Andrew Monkhouse und Terry Camerlengo

Sun Certified Java Developer
Study Guide

25. November 2009

Deutsche Übersetzung von Ralf Wahner

Inhalt

Über die Autoren

Andrew Monkhouse gehört zu den Moderatoren der Java-Ranch (*http://www.javaranch.com*) und betreut zur Zeit für die SCJD- und SCJA-Foren. Andrew ist Träger der folgenden Zertifikate: SCJP 1.2, SCJP 1.4, SCJD, SCWCD, SCBCD sowie des ersten Teiles des SCEA. Er arbeitet schon viel zu lange mit Computern (sein erstes Programm war auf einer Zeichenlochkarte, ähnlich einer Lochkarte, geschrieben).

Andrew hat in verschiedenen Positionen gearbeitet, vom Programmierer bis zum Architekten und IT-Manager, unter VMS, Unix, Macintosh und Windows. Er hat Backend-, Middleware- und Frontendlösungen für verschiedene Industriezweige entwickelt. Andrew ist im Herzen Australier, obwohl er sich durch seine Arbeit häufig in anderen Ländern aufhält.

Terry Camerlengo hat über neun Jahre Erfahrung in der Softwareentwicklung in verschiedenen Unternehmen, darunter Unternehmen aus der Fortune-500-Liste und Internet-Dienstleister. Er hat Erfahrung in allen Phasen des Software-Lebenszyklus mit einem Schwerpunkt auf objektorientierten Technologien wie Java, C#, C++ und .Net. Zu seiner Expertise gehören Frontend-Webdesign, serverseitige Entwicklung von Enterprise-Anwendungen, Datenbankentwicklung und Modellierung rationaler Datenbanksysteme. Terry hat Sun Microsystems- und Microsoft-Zertifikate und besitzt einen Abschluß in Informatik und Philosophie der Ohio State University. Terry arbeitet zur Zeit als Senior-Developer und Wissenschaftler im biomedizinischen Institut des James Cancer Centers der Ohio State University und betreibt fortgeschrittene Studien im Bereich Bioinformatik.

Über den technischen Editor

Jim Yingst hat an der University of Arizona Technische Physik studiert, sich nach seinem Abschluß aber dem IT-Arbeitsmarkt zugewandt weil, ... nun, weil er es damals für eine gute Idee hielt. Er streift nun durch den amerikanischen Westen und hilft technischen Unternehmen dabei, Ihre IT-Probleme zu lösen.

Jim ist Sheriff (Administrator) und seit langer Zeit Mitarbeiter auf der Java-Ranch (*http://www.javaranch.com*), wo er Fragen zu Java beantwortet, vom Thema abweichende Diskussionen umsortiert und sich mit störenden Australiern herumschlägt.

Er scheint den größten Teil seiner Freizeit wie besessen davon zu sein, Buchhandlungen aufzusuchen. In selten Fällen liest er die Bücher, die er kauft (meistens Science Fiction). In der restlichen Zeit hört er wahrscheinlich düstere progressive Rockmusik oder sucht nach neuen Thai-Restaurants. Jim lebt in Boulder, Colorado.

Danksagungen

Wir möchten den folgenden Menschen danken:

- Mehran (Max) Habibi, der die erste Auflage dieses Buches geschrieben, die zweite Auflage auf den Weg gebracht und uns den Verlagsmitarbeitern von Apress vorgestellt hat.
- Unserem technischen Editor Jim Yingst, der nicht nur unseren Text überprüft, sondern auch viele gute Vorschläge gemacht hat.
- Den fantastischen Verlagsmitarbeitern von Apress für die Zusammenarbeit während dieses Projektes; dafür daß sie unseren Rohentwurf akzeptiert und in ein Buch verwandelt haben.
- Uns gegenseitig.

Ohne die Hilfe all dieser Menschen wäre dieses Buch weit davon entfernt, so gut zu sein, wie es nun ist. Wir möchten uns bei unseren Familien, Freunden und Kollegen bedanken, die unsere Schwankungen zwischen völliger Ungeselligkeit angesichts nahender Termine und den anschließenden verzweifelten Versuchen es wieder gut zu machen ertragen haben.

Andrew Monkhouse und Terry Camerlengo

Vorwort

Die Prüfung zum *Sun Certified Java Developer* unter Version 5 des Java Development Kits (JDK 5) bietet Java-Entwicklern die einmalige Gelegenheit, ihre Fertigkeiten praktisch einzusetzen, ohne auf eine bestimmte Entwicklungs- oder Laufzeitumgebung fixiert zu sein. Die Prüfung bietet außerdem eine umfassende Lernumgebung, da viele verschiedene APIs verwenden werden können und viele Lösungsalternativen möglich sind. Dieses Buch führt in eine Reihe der Themengebiete ein, die Sie beherrschen müssen, um die Prüfung zum *Sun Certified Java Developer* zu bestehen.

Viele Entwickler sind angesichts des Prüfungsumfangs eingeschüchtert. Die Prüfung umfaßt die gesamte Bandbreite von der Backend-Datenbank, über Server- und Frontend-Anwendung bis hin zur API-Dokumentation und einer Betriebsanleitung für die Benutzer. Dieses Buch deckt jeden Abschnitt in allen Einzelheiten ab und hilft Ihnen dabei, Schritt für Schritt Ihre Kenntnisse in den einzelnen Gebieten zu vervollständigen, während parallel eine komplette Beispielanwendung entwickelt wird. Dieses Buch führt darüberhinaus in die neuen Eigenschaften und Fähigkeiten von Version 5 des Java Development Kits ein, die im Kontext der Beispielanwendung demonstriert werden.

Sun Microsystems schreibt bewußt kein Betriebssystem oder eine Entwicklungsgebung vor. Sie benötigen lediglich einen Rechner und die aktuelle Version des JDK 5. Die deutsche Übersetzung basiert durchgängig auf dem JDK 5 unter Ubuntu 8.10. Da die in diesem Buch beschriebenen neuen Eigenschaften und Fähigkeiten des JDK 5 in der Beispielanwendung angewendet werden, benötigen Sie ein JDK 5, um die Beispielanwendung starten zu können. Die Beispielanwendung selbst hängt dagegen *nicht* von Ubuntu 8.10 ab.

Wir hoffen, daß Ihnen dieses Buch Freude bereitet und freuen uns auf die Nachricht, daß Sie bestanden haben sowie Ihren Anmerkungen zu diesem Buch. Sie erreichen beide Autoren unter der E-Mailadresse *scjd@apress.com*.

Teil I

Einleitung und Richtlinien für die Arbeit an der Prüfungsaufgabe

Kapitel 1

Einleitung

[0] Willkommen zur deutschen Übersetzung der zweiten Auflage des *Sun Certified Java Developer Study Guide*. Durch die neuen Eigenschaften und Fähigkeiten der J2SE 5 ist die Prüfung zum *Sun Certified Java Developer* leichter als je zuvor. Generische Kollektionen, die erweiterte `for`-Schleife, das Autoboxing zwischen primitiven Typen und ihren Wrappertypen, der neue Sperrmechanismus für Threads im Package `java.util.concurrent.locks` und vieles mehr, bieten dem Entwickler einen reichhaltiger als je zuvor ausgestatteten Werkzeugkasten. Dieses Buch und die begleitende Beispielanwendung *Denny's DVDs* unterstützen Sie dabei, sich das erforderliche Verständnis und Wissen zu erarbeiten, um die Prüfung zum *Sun Certified Java Developer* zu meistern und zugleich den Umgang mit einigen der neuen Spracheigenschaften des J2SE Development Kits (JDK)[1] zu erlernen. Wenn Sie die Prüfung zum *Sun Certified Java Developer* ablegen möchten und bereit sind, sich mit den tieferen Geheimnissen von Java auseinanderzusetzen, haben Sie das richtige Buch gefunden.

[1] Die beste Methode, um etwas Neues zu lernen besteht darin, es zu verwenden. Das gilt für sowohl Tennis und Töpfern als auch für das Programmieren. Aufbauend auf dem Prinzip „Learning by Doing“ hilft Ihnen dieses Buch dabei, sich mit Version 5 des Java Development Kits auseinanderzusetzen und sich die Verfahren, Fertigkeiten und das benötigte Hintergrundwissen anzueignen, um die SCJD-Prüfung zu bestehen. Sun Microsystems hat die SCJD-Prüfung als realistisches Beispiel für die Anforderungen gestaltet, die einen professionellen Java-Entwickler in der Praxis erwarten. Die SCJD-Prüfung deckt einen großen Teil von Version 5 des Java Development Kits ab, beispielsweise Remote Method Invocation (RMI), Threads, die Ein-/Ausgabebibliothek und Swing (graphische Benutzeroberflächen).

[2] Die in Kapitel 3 eingeführte Beispielanwendung *Denny's DVDs* dient als Studienobjekt für die prüfungsrelevanten Themengebiete. Im Gegensatz zur Anleitung einer SCJD-Prüfungsaufgabe werden die angewendeten Gebiete im Rahmen der Beispielanwendung in allen Einzelheiten erklärt. Wenn Sie dieses Buch durchgearbeitet haben sind Sie mit dem erforderlichen Rüstzeug ausgestattet, um die Prüfung anzutreten und zu bestehen. Dieses Einleitungskapitel hat zwei Hauptanliegen:

- Erläuterung, wie Sie Ihre Prüfungsaufgabe herunterladen und mit der Arbeit beginnen können.
- Erläuterung der Zielsetzung dieses Buches.

[1] *Anmerkung des Übersetzers*: Wir verwenden von nun an statt „J2SE“ die äquivalente Bezeichnung „Java Development Kit“ (JDK) beziehungsweise statt „J2SE 5“ die Bezeichnung „Version 5 des Java Development Kits“.

1.1 Version 5 des Java Development Kits

[3] Version 5 des Java Development Kits ist ein Major-Update und wurde mit der Zielsetzung entworfen, die Entwicklung zu erleichtern, die Skalierbarkeit zu verbessern, provide for additional monitoring and manageability sowie um die mit Swing entwickelten graphischen Benutzeroberflächen zu verbessern. Obwohl Version 5 des Java Development Kits eine ganze Reihe neuer Eigenschaften und Fähigkeiten beinhaltet, konzentriert sich dieses Buch auf eher alltägliche Dinge wie Threads, RMI, Sockets, die Verkettung von Ausnahmen, Protokollierung (*logging*) und Serialisierung. Wenn Sie die Grundlagen erst einmal verstanden haben, ergibt sich alles übrige in natürlicher Weise.

1.2 Die Prüfung zum Sun Certified Java Developer

[4] Die Prüfung zum *Sun Certified Java Developer* ist ein umfassendes Examen, mit dem Sun Microsystems die Fähigkeiten fortgeschrittener Java-Entwickler verifiziert. Das SCJD-Zertifikat wird in der Regel als aussagekräftiger Nachweis von Kompetenz betrachtet. Dieses Buch konzentriert sich auf die Eigenschaften und Fähigkeiten von Version 5 des Java Development Kits, die für diese Prüfung relevant sind. Das Bestehen der Prüfung setzt eine angemessene Vorbereitung voraus. Aufgrund des Schwierigkeitsgrades ist die Zulassung zur SCJD-Prüfung nur für Kandidaten möglich, die bereits die Prüfung zum *Sun Certified Java Programmer* (SCJP) abgelegt haben. Dieses Buch erläutert die zum Bestehen der SCJD-Prüfung erforderlichen Themengebiete.

Bemerkung: Gegenwärtig gibt es zwei Versionen der Prüfung zum *Sun Certified Java Programmer*: je eine für die Versionen 5.0 und 6.0 der Java Plattform. Die Fragen zwischen den verschiedenen Versionen unterscheiden sich geringfügig. Dennoch genügt die Zertifizierung bezüglich einer dieser beiden Versionen als Zulassung zur SCJD-Prüfung.

1.2.1 Der Zertifizierungsprozeß

[5] Die Zertifizierung besteht aus zwei Teilen. Der erste Teil ist Ihre Prüfungsaufgabe (eine Programmieraufgabe), bestehend aus einer Anleitung, einer Datei mit Beispieldaten (Datenbankdatei), einem zu implementierenden Interface und einigen aufgabenspezifischen Anforderungen. Sie bearbeiten die Prüfungsaufgabe und reichen Ihre Lösung zur Begutachtung und Bewertung ein. Sie finden im nächsten Unterabschnitt eine Anleitung zum Registrieren und zum Herunterladen Ihrer Prüfungsaufgabe.

[6] Nachdem Sie Ihre Prüfungsaufgabe bearbeitet und Ihre Lösung eingereicht haben, kommen Sie zum zweiten Teil des Zertifizierungsprozesses, nämlich der schriftlichen Prüfung mit der Sie nachweisen sollen, daß Sie die eingereichte Lösung selbst verfaßt haben, indem Sie beschreiben, welche Überlegungen zu den einzelnen Design-Entscheidungen und der Implementierung im ersten Teil des Zertifizierungsprozesses geführt haben. Sie haben 90 Minuten Zeit, um die schriftliche Prüfung abzufassen, da aber gegenwärtig nur vier Fragen gestellt werden, ist die Zeit großzügig bemessen.

[7] Bücher, Notizen oder sonstige Materialien sind während der Prüfung nicht erlaubt. Es ist daher sinnvoll, die schriftliche Prüfung so schnell wie möglich nach dem Einreichen der Lösung abzulegen, wenn Sie noch alles frisch im Kopf haben. Beide Prüfungsteile werden zusammen ausgewertet, obwohl die programmierte Lösung zuerst eingereicht werden muß. Das bedeutet, daß Sie in der

schriftlichen Prüfung generische Fragen beantworten müssen, keine spezifischen Fragen zu Ihrer individuellen Lösung. Sie brauchen ein ganzheitliches Verständnis, um beide Teile der Prüfung zum *Sun Certified Java Developer* zu bestehen.

Warnung: Ihre eingereichte Lösung wird nicht eher einem Gutachter übergeben, als bis Sie auch die schriftliche Prüfung abgelegt haben. Wenn Sie die schriftliche Prüfung nicht ablegen, erhalten Sie keine Warnung, daß Ihre Lösung nicht begutachtet wird. Sie sind solange in der Schwebe, bis Sie die schriftliche Prüfung abgelegt haben.

---

[8] Das Ziel der Prüfung zum *Sun Certified Java Developer* besteht darin, Ihr Verständnis der wichtigsten Eigenschaften und Fähigkeiten von Java zu überprüfen. Dazu gehören Threads, RMI, Sockets, Serialisierung, die Ein-/Ausgabebibliothek und Swing. Jede Prüfungsaufgabe ist ein Einzelexemplar und prüft diese Themen in unterschiedlichem Umfang. Sie müssen beispielsweise einen Server entwickeln, der mehrere gleichzeitig eingehende Anfragen verarbeiten kann. Das vorgegebene Interface kann aber festlegen, welche Klassen sicheren Zugriff durch viele Threads erlauben müssen. Or your requirements could call for strict search requirements versus more general searching ability. Dieses Buch erfaßt alles was Sie über die einzelnen Gebiete wissen müssen und berücksichtigt dabei die relevanten Änderungen in Version 5 des Java Development Kits.

[9] Sun Microsystems verlangt, daß Sie ein aktuelles Java Development Kit verwenden (das heißt eines das nicht in den letzten 18 Monaten von einer neueren Version verdrängt wurde), um Ihre Lösung zu entwickeln, so daß die Prüfungskandidaten mit den neuen Eigenschaften und Fähigkeiten der Sprache in Verbindung bleiben. Sie finden unter der Internetadresse *http://java.sun.com/j2-se/codenames.html* eine Liste von Release-Daten.

1.2.2 Herunterladen Ihrer Prüfungsaufgabe

[10] Sie können sich in Deutschland für die Zuweisung einer Prüfungsaufgabe und die schriftliche Prüfung unter der Internetadresse *http://de.sun.com/training/certification/objectives/index.xml* registrieren. In vielen anderen Ländern ist die Online-Registrierung ebenfalls möglich, siehe *http://www-.sun.com/training/world_ training.html*. Nachdem Sie die Gebühr für Ihre Prüfungsaufgabe entrichtet haben, erhalten Sie ein E-Mail mit einer genauen Anleitung, wie Sie die *.jar* Datei mit Ihrer Prüfungsaufgabe herunterladen können. Das Eintreffen dieses E-Mails kann einige Tage dauern, aber auch noch am selben Tag geschehen. Fertigen Sie unmittelbar nach dem Herunterladen der *.jar* Datei einige Kopien an und bewahren Sie sie sicher auf. Es ist sehr teuer, eine zweite Kopie der Aufgabe zu bekommen to match the subject of this section.

Tipp: Die Website der Schulungsabteilung von Sun Microsystems gibt die Adressen der Internetseiten an, von denen Sie Ihre Prüfungsaufgabe herunterladen können. Eventuell können Sie Ihre Aufgabe auch schon herunterladen, bevor Sie per E-Mail benachrichtigt werden, daß Ihr Konto zum Herunterladen eingerichtet wurde.

1.2.3 Dokumentation und Fragen

[11] Sie werden wahrscheinlich Fragen zu den Anforderungen Ihrer Prüfungsaufgabe haben. Sun Microsystems beantwortet diese Fragen generell nicht. Eventuell möchte Sun Microsystems sehen, wie Sie sich zwischen verschiedenen Möglichkeiten entscheiden und wie gut Sie Ihre Entscheidungen begründen können. Möglicherweise soll auch eine praxisgerechte Situation simuliert werden, in der

ein Kunde nicht immer zur Kommunikation bereit ist. Es könnte aber auch einfach daran liegen, daß es unmöglich ist, die Fragen jedes einzelnen Prüfungskandidaten zu beantworten. Es ist dennoch wichtig, daß Sie Ihre Fragen artikulieren und in der Dokumentation diskutieren, die Sie als Teil Ihrer Lösung einreichen müssen. Beschreiben Sie zumindest Ihre Annahmen und Entscheidungen. Zur weiteren Unterstützung empfehlen wir die Java-Ranch (*http://www.javaranch.com*) sowie die Diskussionsgruppen für Java-Zertifizierungen bei Yahoo!

[12] Das zweite Kapitel enthält Vorschläge für das Anlegen und Schreiben von Dokumentationskommentaren (Javadoc-Kommentaren) und beschreibt eine Auswahl industriespezifischer bewährter Arbeitsmethoden. Verwenden Sie keine ausgefallenen Namen für Ihre Bezeichner oder gar die ungarische Notation. Befolgen Sie bestehende Richtlinien, falls vorhanden. As far as the SCJD exam is concerned, Sun Microsystems really wants you to color inside the lines.

1.3 Die Zielgruppe dieses Buches

[13] Dieses Buch ist für den beruflich aktiven Java-Entwickler gedacht, der eine Einführung in Version 5 des Java Development Kits sucht und sich dafür die interessiert, die Prüfung zum *Sun Certified Java Developer* abzulegen. Die SCJD-Prüfung vermittelt dem Entwickler einen Eindruck davon, wie sich ein echtes Projekt „anfühlt“. Sie müssen bereit sein, sich dieser Herausforderung zu stellen. Ein Entwickler, der die Prüfung zum *Sun Certified Java Programmer* (SCJP) bestanden hat oder prinzipiell bestehen könnte, wird sich hier zu Hause fühlen. Entwickler mit weniger als einem halben Jahr Erfahrung, sollten zur Ergänzung dieses Buch einige der anderen Java-Bücher von Apress (*http://www.apress.com*) oder eines anderen Verlages zu Rate ziehen.

[14] Dieses Buch beschreibt sowohl Eigenschaften und Fähigkeiten des Java Development Kits, die teilweise bereits seit Jahren zum Sprachumfang gehören, als auch solche, die erst seit der Version 5 zur Verfügung stehen. Wir setzen in diesem Buch nicht mehr voraus, als daß der Leser mit Java vertraut genug ist, um die Prüfung zum *Sun Certified Java Programmer* zu bestehen, so daß wir keine Zeit investieren müssen, um die Grundlagen der Sprache zu erklären (zum Beispiel den Unterschied zwischen den Datentypen `int` und `long`). Wir werden aber auch neue Spracheigenschaften detailliert beschreiben, so daß Leser, die die neuen Eigenschaften und Fähigkeiten von Version 5 des Java Development Kits noch nicht kennen, sie hier entdecken können.

1.4 Über dieses Buch

[15] Dieses Buch ist der Prüfung zum *Sun Certified Java Developer* gewidmet, einer von mehreren Zertifierungsprüfungen zu Java, die von Sun Microsystems angeboten werden. Die Prüfungen zum *Sun Certified Java Developer* (SCJD) und *Sun Certified Enterprise Architect* (SCEA) verlangen, daß der Kandidat ein Projekt bearbeitet, während die anderen Zertifikate auf Theorie basieren und der Kandidat typischerweise Multiple-Choice-Fragen beantworten muß. Vom Standpunkt der Entwicklungsarbeit aus betrachtet, ist die Prüfung zum *Sun Certified Java Developer* die größte Herausforderung unter den von Sun Microsystems angebotenen Prüfungen und daher auch der Schwerpunkt dieses Buches.

[16] Dieses Buch besteht aus drei Teilen. Der erste Teil beschreibt allgemeine Gesichtspunkte der Softwareentwicklung und skizziert die Beispielanwendung *Denny's DVDs*. Der zweite Teil lehrt die erforderlichen Grundlagen von Anfang an und erleichert sowohl das Verständnis als auch die Implementierung. Der dritte Teil beschließt das Buch mit einer Diskussion der Entscheidungen hinsichtlich des Designs und der Implementierung und zeigt Alternativen auf.

[17] Das Buch beschreibt eine Beispielanwendung, deren Umfang und Schwierigkeitsgrad einer SCJD-Prüfungsaufgabe entsprechen und gebraucht dabei einige neue Eigenschaften und Fähigkeiten von Version 5 des Java Development Kits. Jedes prüfungsrelevante Thema wird in allen Einzelheiten erläutert und seine Vor- und Nachteile diskutiert. An einigen Stellen werden verschiedene Entwicklungsmöglichkeiten parallel diskutiert und implementiert.

[18] Die Anwendung von Entwurfsmustern wird in den jeweiligen Kapiteln diskutiert und die Entwurfsmuster kurz erklärt. Wir empfehlen nachdrücklich, sich Informationsmaterial über Entwurfsmuster zu beschaffen. Verschiedene Websites bieten aufschlußreiche Anleitungen zu diesem Thema, darunter die Websites von Sun Microsystems (*http://www.sun.com*) und TheServerSide (*http://www.theserverside.com*). Es gibt auch verschiedene ausgezeichnete Bücher zu diesem Thema, zum Beispiel *Head First: Design Patterns* von Elisabeth Freeman, Eric Freeman, Bert Bates und Kathy Sierra (O'Reilly, 2004).

[19] Das Buch präsentiert viele Beispiele, die Ihnen beim Entwickeln einer praxistauglichen Java-Anwendung helfen werden. Jedes Kapitel liefert einen Beitrag zur Entwicklung dieser Anwendung, indem es ein wichtiges Gebiet wie Threads, Swing oder die Entwicklung der Netzwerkschnittstelle entweder per RMI oder per Sockets beschreibt. Das Buch beantwortet die Fragen, die sich beim Studium dieser Themengebiete in natürlicher Weise einstellen und erläutert, wie Sie die zu bewältigenden Probleme lösen können. Darüberhinaus werden die Vorteile, Nachteile und Auswirkungen der bestehenden Wahlmöglichkeiten diskutiert.

- Kapitel 1 (Einleitung): Dieses Kapitel ist eine allgemeine Einführung und Entscheidungshilfe in der Frage, ob dieses Buch Ihren Bedürfnissen entspricht oder nicht. Das Kapitel beschreibt den Aufbau des „Lehrplanes" in diesem Buch, führt in die Prüfungsanforderungen ein und visiert die Themen der technischen Diskussion an, die sich durch die folgenden Kapitel ziehen.

- Kapitel 2 (Untersuchung und Planung der Beispielanwendung): Dieses Kapitel widmet sich den grundlegenden Eigenschaften einer SCJD-Prüfungsaufgabe in allen Merkmalen und Anforderungen, zum Beispiel der Datei- und Verzeichnisstruktur, dem Umgang mit Packages, den Richtlinien für die Benennung von Bezeichnern und die Formatierung von Quelltext und Kommentaren, den verlangten Begleitdateien (zum Beispiel *readme.txt*) sowie allgemein gültigen Schritten zu Beginn einer SCJD-Prüfungsaufgabe.

- Kapitel 3 (Einführung in die Beispielanwendung): Dieses Kapitel führt die Beispielanwendung *Denny's DVDs* ein. Die Beispielanwendung umfaßt Klassen, die den Zugriff auf eine Datei ähnlich dem Zugriff auf eine Datenbank ermöglichen (Datenbankdatei), eine graphische Benutzeroberfläche (Swing) und zwei unterschiedliche Netzwerkschnittstellen (RMI und Sockets) implementieren. Lesen Sie dieses Kapitel sorgfältig: Es hilft bei der präzisen Festlegung des Leistungsumfangs der Beispielanwendung.

- Kapitel 4 (Threads): Dieses Kapitel beginnt „am Reißbrett" und führt Sie zu einem klaren Verständnis für die Threadprogrammierung. Das Kapitel behandelt das Interface `java.lang.Runnable`, die Klasse `java.lang.Thread`, das Sperren von Methoden beziehungsweise Blöcken von Anweisungen, die Synchronisierung von Methoden, Threads im Warte- und im Schlafzustand, die Benachrichtigung wartender Threads sowie Threadscheduling. Wir besprechen den neuen Sperrmechanismus in Version 5 des Java Development Kits (Package `java.util.concurrent.locks`) und zeigen, wie er die Threadprogrammierung erleichtert. Der Fokus richtet sich dabei auf die prüfungsrelevanten Gesichtspunkte der Threadprogrammierung.

- Kapitel 5 (Die Klasse DvdDatabase): Dieses Kapitel dokumentiert die Entwicklung einer für die Beispielanwendung grundlegenden Klasse, die das vorgegebene Interface (`sampleproject.db.DBClient`) implementiert. In diesem Zusammenhang die Entwurfsmuster *Façade*, *Value-Object* (*Transfer-Object*) und *Adapter* auf.

- Kapitel 6 (RMI als Netzwerkschnittstelle): Dieses Kapitel führt in die Entwicklung verteilter Anwendungen und die erste von zwei Varianten für die Entwicklung der Netzwerkschnittstelle Ihrer Prüfungsaufgabe ein. Sie lernen die Remote Method Invocation (RMI) kennen und erlernen die Entwicklung von RMI-Clients und -Servern.
- Kapitel 7 (Sockets als Netzwerkschnittstelle): Dieses Kapitel führt in die andere Variante für die Entwicklung der Netzwerkschnittstelle Ihrer Prüfungsaufgabe ein. Sie lernen, eine Socketverbindung zwischen einem Client- und einem Serversocket aufzubauen und diskutieren die Vor- und Nachteile von Sockets im Vergleich mit RMI. Das Kapitel beinhaltet außerdem eine kurze Diskussion der Themen „Sicherheit“, „Serialisierung“ sowie der Entwurfsmuster *Command* und *Proxy*.
- Kapitel 8 (Die graphische Benutzeroberfläche): Dieses Kapitel führt in die Entwicklung graphischer Benutzeroberflächen mit der Swing-Bibliothek ein und ist für Java-Entwickler mit wenig beziehungsweise keiner Swing-Erfahrung geschrieben. Das Kapitel beginnt ebenfalls ganz von vorne und erläutert die Grundzüge der Swing-Bibliothek, die Rolle des Entwurfsmusters *Model-View-Controller*, die Ereignisbehandlung, die Verwendung der `JTable`-Komponente und wie alle Teile zusammengehören.
- Kapitel 9 (Projektabschluß): Dieses Kapitel betrachtet die Beispielanwendung im Rückblick. Wir erledigen einige abschließende Handgriffe und packen die verlangten *.jar* Dateien. Wir betrachten nochmals die getroffenen Design-Entscheidungen die damit verbundenen Vor- und Nachteile und bereiten die Beispielanwendung stellvertretend für Ihre Prüfungsaufgabe zum Einreichen beim Gutachter vor.

[20] Der Quelltext der Beispielanwendung, sowie verschiedene hilfreiche Diagramme und Dokumente, stehen Ihnen im „Source Code“-Abschnitt der Apress-Website (*http://www.apress.com/book/sourcecode*) zum Herunterladen zur Verfügung.

1.5 Konfiguration der SE 5

[21/22] Die Konfiguration von Version 5 des Java Development Kits ist sehr einfach. Sun Microsystems bietet ausführliche Dokumentation dazu für verschiedene Plattformen an und wir verzichten darauf, an dieser Stelle alles einfach abzuschreiben. Sie finden Informationen zum Herunterladen und Konfigurieren von Version 5 des Java Development Kits unter der Internetadresse *http://java.sun.com/javase/index.jsp*.

1.6 Zusammenfassung

[23] Wir haben dieses erste Kapitel für einen Überblick über die Prüfung zum *Sun Certified Java Developer* genutzt und einige Ratschläge ausgesprochen, von denen Sie Gebrauch machen können, um die Prüfung zu bestehen. Wir haben die einzelnen Themengebiete der Prüfung aufgezählt, den Zertifizierungsprozeß besprochen und den Inhalt der einzelnen Kapitel dieses Buchs vorgestellt.

[24] Sie haben durch das Lesen des ersten Kapitels bereits mit der Prüfungsvorbereitung begonnen. Wenn Sie etwa vier Wochen investieren, um die in diesem Buch diskutierten Themengebiete zu verinnerlichen, sollten Sie ausreichend vorbereitet sein, um die Prüfung zum *Sun Certified Java Developer* zu bewältigen. Kalkulieren für jedes der Hauptgebiete Threads, Swing, Netzwerkprogrammierung und die graphische Benutzeroberfläche eine Woche ein. Der Zeitbedarf hängt natürlich von Ihrem

persönlichen Erfahrungshintergrund ab. Sie haben nun ein Gefühl dafür, was Sie erwartet. Wir wünschen Ihnen viel Glück! Lernen Sie eifrig und schreiben Sie uns ein E-Mail (*scjd@apress.com*), wenn Sie die Prüfung bestanden haben.

1.7 Häufige Fragen

- *Frage*: Bin ich bereit für die Prüfung zum *Sun Certified Java Developer*?

  Antwort: Wenn Sie die Prüfung zum *Sun Certified Java Programmer* bestanden haben oder kurz davor stehen, sind Sie im Prinzip soweit, daß Sie mit der Vorbereitung für die SCJD-Prüfung beginnen können. Es gibt keine zeitliche Beschränkung für das Bearbeiten der Prüfungsaufgaben, Sie können also Ihre Aufgabe herunterladen und das Buch durcharbeiten, während Sie Ihre Aufgabe lösen. Denken Sie allerdings daran, daß die Gültigkeit des Prüfungsgutscheins zeitlich begrenzt ist. Einige Sun Microsystems-Vertretungen verlangen, daß Sie Prüfungaufgabe und -gutschein zusammen bezahlen.

- *Frage*: Hilft mir dieses Buch bei der Vorbereitung auf die Prüfung zum *Sun Certified Java Programmer* (SCJP)?

  Antwort: Dieses Buch kann zusammen mit einem Buch über die SCJP-Prüfungsthemen für Ihr Verständnis der Themen förderlich sein, aber wir empfehlen es nicht als Informationsquelle für SCJP-Prüfungsanwärter. Einige der Themen, die Gegenstand der SCJP-Prüfung sind, werden zwar in diesem Buch detailliert behandelt, andere Themen aber entweder gar nicht oder sie werden ohne Erläuterungen und in der Annahme verwendet, daß ihr Gebrauch bekannt ist. Außerdem werden in diesem Buch viele Dinge behandelt, die für die SCJP-Prüfung nicht relevant sind.

- *Frage*: Ich habe Schwierigkeiten damit, meine Entwicklungsumgebung zu konfigurieren. Wo finde ich Hilfe?

  Antwort: Lesen Sie auf der Java-Website von Sun Microsystems nach (*http://java.sun.com/javase/index.jsp*) und folgen Sie der Dokumentation exakt. Falls dies nicht hilft, kontaktieren Sie Sun Microsystems direkt.

- *Frage*: Welche Themen werden in diesem Buch diskutiert?

  Antwort: Dieses Buch diskutiert und erläutert RMI, Threads, Swing, Sockets, Zusicherungen (*assertions*), die Verkettung von Ausnahmen (*exception chaining*) und Protokollierung (*logging*).

- *Frage*: Wieviel kostet die SCJD-Prüfung?

  Antwort: Die Prüfung kostet etwa 400 US-$. Die Gebühr unterliegt natürlich dem Ermessen von Sun Microsystems.

- *Frage*: Ich habe meine Aufgabe verloren. Was soll ich tun?

  Antwort: Versuchen Sie, die Aufgabe nochmals von der Sun Microsystems-Website herunterzuladen. Wenn es nicht funktioniert, kontaktieren Sie Sun Microsystems direkt.

Kapitel 2

Untersuchung und Planung der Beispielanwendung

[0] Dieses Kapitel beschreibt die bei allen Softwareprojekten erforderliche Explorations- und Designphase im Hinblick auf die Beispielanwendung *Denny's DVDs* sowie auf Ihre eigene Prüfungsaufgabe, die Sie bearbeiten und zur Begutachtung einreichen müssen, um die Prüfung zum *Sun Certified Java Developer* zu bestehen. Wir besprechen in diesem Kapitel folgende Themen:

- Planung der ersten Schritte auf Ihrem Weg zur Prüfung.
- Organizing the layout of your project.
- Dokumentation von Softwareprojekten.
- Industrielle Richtlinien hinsichtlich der Formatierung des Quelltextes sowie der API-Dokumentation (Javadoc). Berücksichtigung dieser Richtlinien bei Ihrer Prüfungsaufgabe von Anfang an.
- Verwendung von Packages zur Aufteilung des Quelltextes nach funktionalen Kriterien.
- Erlernen häufiger Arbeitsmethoden in der Softwareentwicklung, zum Beispiel Zusicherungen (*assertions*) und Protokollierung (*logging*).

[1/2] Wir erfinden das Rad in diesem Kapitel nicht neu, sondern konzentrieren uns auf bewährte Grundsätze wie die von Sun Microsystems abgefaßten Richtlinien zur Formatierung von Quelltexten beziehungsweise zur Namenswahl bei Bezeichnern, die API-Dokumentation (Javadoc) und die Verteilung der Klassen einer Anwendung auf Packages anhand funktionaler Eigenschaften und Fähigkeiten. Ein Teil dieser Dinge ist notwendig, um die Prüfung zum *Sun Certified Java Developer* zu bestehen. Allerdings sollte jedes davon seinen Platz im Werkzeugkasten eines Java-Entwicklers haben. Wenn Sie diese Richtlinien von Anfang an befolgen, sind Sie auf dem besten Weg ein zertifizierter Java-Entwickler zu werden.

2.1 Die Explorations- und Designphase

[3] Es ist sehr verlockend zu Beginn eines Projektes sofort mit dem Programmieren anzufangen, denn es macht Spaß und vermittelt einen unmittelbaren Eindruck von Fortschritt. Diese Vorgehensweise hat allerdings signifikante Nachteile. Der Projektstart ohne vorausgehende angemessene Planung

kann das Projekt an unausgesprochene Voraussetzungen binden und dazu führen, daß Sie entscheidende Dinge übersehen oder Design-Entscheidungen treffen, die sich erst später im Lebenszyklus des Projektes als mangelhaft herausstellen.

[4] In der Regel ist es das Sinnvollste, zu Beginn des Projektes die Anforderungen zu bestätigen, den Datenfluß zu planen und eine Skizze der graphischen Benutzeroberfläche anzufertigen. Anschließend beginnt die Designphase.

Bemerkung: In der Softwareindustrie werden verschiedene Entwicklungsansätze unterschiedlich intensiv praktiziert. Beispiele für häufig eingesetzte Designverfahren sind die „iterative Vorgehensweise“ (*Iterative Process*), der Rational Unified Process, das Spiralmodell nach Barry W. Boehm (*Boehm Spiral Model*) sowie das Extreme Programming (XP). Wir verwenden in diesem Buch ein anderes gängiges Verfahren, nämlich das Wasserfallmodell (*Waterfall Model*). Dieses Verfahren setzt voraus, daß die Entwicklung ein fortschreitender Prozeß ist, wobei eine Version der Software auf der Vorgängerversion aufbaut. Außerdem werden die Anforderungen beim Wasserfallmodell vor der Design- und der Entwicklungsphase bestimmt. Der Zeitraum zur Bearbeitung einer SCJD-Prüfungsaufgabe ist nicht beschränkt und es wird verlangt, daß Sie die Aufgabe alleine bearbeiten. Diese beiden Kriterien passen ausgezeichnet zum Wasserfallmodell.

---

[5] Die Designphase eines Projektes ist ein fließender, von der eigentlichen Entwicklung eingefriedeter Vorgang. Der beste Weg durch die Designphase einer Anwendung besteht darin, die technischen Anforderungen dadurch zu untersuchen, daß Sie ein bißchen programmieren, ein bißchen designen und wieder ein bißchen programmieren. In diesem Sinne ist die Designphase ein iterativer Vorgang. Wir wenden uns nun den Richtlinien zu.

2.1.1 Erste Schritte

[6] Nehmen Sie sich zunächst etwas Zeit, um Ihr Verständnis der Anforderungen des Projektes beziehungsweise Ihrer Prüfungsaufgabe zu verifizieren. Lesen Sie die Anleitung mehrmals und hinterfragen Sie das Gelesene. Klopfen Sie die logische Gliederung der Funktionalität ab und dokumentieren Sie Ihre Annahmen schriftlich. Achten Sie auf die ~~umbrella activities~~, die mehrere unterschiedliche Varianten ~~under a given topic~~ zusammenfassen. Die groben Zusammenhänge helfen oft bei der Packagestruktur. Es ist zum Beispiel sinnvoll ein `gui`-Package anzulegen, das für die Datenvisualisierung verantwortlich ist.

2.1.2 Schriftliches Erfassen der Anforderungen

[7] Die Anforderungen eines Projektes sind die Funktionalität, die der Kunde von der Anwendung erwartet, die Sie entwickeln sollen. Die Anforderungen beschreiben detailliert jede einzelne Funktion, die die Anwendung haben soll. Fragen Sie während der Explorations- und Designphase eines Projektes so viel Sie können. Arbeiten Sie Ihre Fragen schriftlich auf, bevor Sie sie beantworten und achten Sie darauf, daß sowohl die Fragen als auch die Antworten sinnvoll sind. Formulieren Sie Ihre Fragen um und stellen Sie sie erneut. Es ist besser zu Beginn eines Projektes etwas langsamer zu arbeiten, als am Ende festzustellen, daß man unachtsam war.

[8] Bestätigen Sie alle Ihre Annahmen entweder schriftlich ~~or as a GUI layout~~. Bei Ihrer Prüfungsaufgabe haben Sie keinen Ansprechpartner, den Sie persönlich fragen können. Lassen Sie sich dadurch aber nicht davon abhalten, Gedanken oder Risiken zu artikulieren. Es soll Sie im Gegenteil sogar ermutigen, Ihre Fragen zu formulieren und Ihre Gedanken zu ordnen.

Bemerkung: Kapitel 3 beschreibt einige aus den Anforderungen an die Beispielanwendung abgeleitete Anwendungsfälle (*use cases*). Kapitel 8 zeigt anhand der graphischen Benutzeroberfläche, wie die Anwendungsfälle in Funktionalität der Beispielanwendung übertragen werden.

2.1.2.1 Prototyp der graphischen Benutzeroberfläche

[9] Zeichnen Sie, wenn Sie mit einem Prototyp für Ihre graphische Benutzeroberfläche beginnen, einen einfachen Entwurf der verschiedenen Fenster mit Papier und Bleistift. Dadurch bekommen Sie ein Gefühl dafür, was der Benutzer braucht, bevor Sie sich über die interne Funktionsweise der graphischen Benutzeroberfläche Gedanken machen. Dies ist häufig ein wichtiger Schritt für die Abstimmung der Erwartungen der Benutzer mit der späteren Realität durch die implementierte Anwendung. Kapitel 8 zeigt ein Beispiel für einen skizzierten Prototypen einer graphischen Benutzeroberfläche (Abbildung 8-22, Seite 263 (Buch)) und beschreibt die Richtlinien zum Layout von Anwendungsschnittstellen zu menschlichen Benutzern.

[10] Wir empfehlen, den Prototyp Ihrer graphischen Benutzeroberfläche in diesem Stadium des Projektes mit Papier und Bleistift zu skizzieren, statt direkt zu programmatisch zu entwickeln. Die direkte Entwicklung am Rechner beinhaltet die folgenden Risiken:

- Die Tester (siehe eingerahmten Text „Testbenutzer“ auf Seite 15) können ein Design verwerfen, obwohl Sie es für gut halten und viel Zeit investiert haben. In diesem Fall haben Sie Zeit verschwendet.
- Sie werden fast sicher mehr Zeit benötigen, um einen Prototypen zu programmieren, als eine Skizze auf Papier zu zeichnen. Zeigen Sie Ihren Testern (siehe eingerahmten Text „Testbenutzer“) diesen Entwurf. Falls die Tester Empfehlungen für Änderungen äußern (oder im schlimmsten Fall Ihre Skizze verwerfen), haben Sie weniger Zeit verloren, wenn es nur eine Skizze auf Papier ist.
- Eine Skizze auf Papier kann überall mit den Testern diskutiert werden, auch wenn kein Rechner verfügbar ist. Änderungsvorschläge können schnell in die Skizze eingesetzt werden.
- Haben Sie bereits viel Zeit in die programmatische Entwicklung Ihres Prototyps investiert, so regt sich ein natürlicher Widerstand gegen Änderungen, der Sie veranlassen kann, gute Ratschläge Ihrer Tester zu ignorieren.
- Eventuell geben Sie aufgrund von Implementierungsdetails frustriert auf, bevor Ihr programmatisch entwickelter Prototyp fertig ist. Möglicherweise opfern Sie damit einen guten Entwurf, um etwas einfacheres zu programmieren, wodurch Sie bei der Begutachtung Ihrer Prüfungsaufgabe Punkte einbüßen.
- Wenn Sie die graphische Benutzerfläche schon jetzt entwickeln, kommen Sie vielleicht zu einem in Ihren Augen schönen Ergebnis. Lehnen Ihre späteren Benutzer die Oberfläche aber ab, so laufen Sie auch hier Gefahr, noch einmal von vorne beginnen zu müssen.

Testbenutzer

Es ist eine Binsenweisheit, daß Entwickler Programme für Entwickler schreiben, das heißt, daß wir, unabhängig von der gestellten Aufgabe, als Entwickler dazu neigen, ein Programm zu schreiben, das aus unserer Perspektive völlig logisch, für die Mehrzahl derjenigen aber schwierig zu benutzen ist, die selbst keine Entwickler sind. Dieser Effekt tritt in allen Berufsgruppen auf, so daß manche

Unternehmen Mitarbeiter einstellen, sie sich ausschließlich um die ästhetische Wirkung der Produkte kümmern. Beispielsweise gibt es bei manchen Autoherstellern Angestellte, deren Tätigkeit einzig und allein darin besteht, dafür zu sorgen, daß das Auto den Kunden ästhetisch anspricht.

Auch wir müssen diesen Effekt berücksichtigen, um eine graphische Benutzeroberfläche zu entwickeln, die dem Benutzer (in unserem Fall dem Gutachter von Sun Microsystems) gefällt und daher akzeptiert wird (unserem Fall also die volle Punktzahl erhält). Die Akzeptanz durch die Benutzer ist ein extrem wichtiges Kriterium, ohne das wir in einen unendlichen Zyklus von Änderungen an der graphischen Benutzeroberfläche geraten. Wir brauchen Testbenutzer, die (vorzugsweise) selbst keine Entwickler sind, um sich unseren Prototyp und die fertige Anwendung anzusehen und uns zu sagen, was noch getan werden muß, um die Anwendung zu einer erstklassigen statt einer mittelmäßigen Anwendung zu machen.

Dies sind unsere bevorzugten Testbenutzer: Ehepartner, Lebensgefährten, Eltern oder die Sekretärin im Büro. Sie sind die Benutzer, die unter Umständen eine Eigenschaft oder Fähigkeit entdecken, die sie für standardisiert halten, aber die in Ihrer Anwendung noch fehlt. Sie sind es, die sich Ihre Anwendung anschauen und Ihnen sagen, daß etwas nicht am richtigen Platz ist. Sie sind es, die beim Testen der Endversion unerwartete Dinge tun, zum Beispiel die Anwendung zweimal starten oder das Fenster kleiner machen, als Sie je erwartet hätten.

Testkandidaten, die Sie dagegen vermeiden müssen, sind andere Entwickler. Sie sind es, die eine fehlende Eigenschaft oder Fähigkeit nicht erwähnen, weil sie sie eventuell selbst nicht mögen. Außerdem ignorieren sie eventuell einen Aspekt in puncto Anwenderfreundlichkeit, da sie sich daran gewöhnt haben, nicht vorhandene Funktionalität in Programmen zu umgehen.

Versuchen Sie, ein paar Testbenutzer zu finden, denen Sie die Skizzen für Ihre graphische Benutzeroberfläche vorlegen können und fragen Sie sie, was Sie von Ihrem Entwurf halten. Respektieren Sie die Kommentare Ihrer Testbenutzer, kehren Sie an Ihren Schreibtisch zurück und arbeiten Sie die Änderungen ein.

Legen Sie Ihren Testbenutzern möglichst mehrere verschiedene Entwürfe vor und lassen Sie sie wählen, mit welcher graphischen Benutzeroberfläche sie am liebsten arbeiten würden. Die Testbenutzer fühlen sich dadurch stärker beteiligt und es fällt ihnen leichter, Änderungen an Ihren Entwürfen vorzuschlagen, wenn die Arbeit noch nicht abgeschlossen ist.

2.1.3 Verwendung erprobter Entwurfsmuster

[11] Das Abweichen von Richtlinien ist nur selten die Entwicklungszeit wert und kann sogar dazu führen, daß Sie in der Prüfung zum *Sun Certified Java Developer* durchfallen. Im schlimmsten Fall setzen Sie sich der Feindschaft der Entwickler aus, die den kryptischen Quelltext pflegen müssen.

[12] Während das Entwickeln einer individuellen Lösung für ein allgemeines Problem prinzipiell möglich ist und unter Umständen lohnend sein kann, sollten Sie im Hinblick auf Ihre Prüfungsaufgabe von diesem Gedanken Abstand nehmen. In der Praxis ist eine individuelle Lösung gelegentlich schneller und günstiger als eine generische Lösung, etwa wenn eine anwendungsspezifische Methode die Elemente eines Arrays schneller sortiert, als die entsprechenden Methoden der Klasse `java.util.Arrays`.

[13] Da Entwurfsmuster nicht der primäre Schwerpunkt dieses Buches sind, gibt es viele Entwurfsmuster, die wir im Rahmen dieses Buches nicht behandeln können. Wir empfehlen aber mit Nachdruck, daß Sie sich mit Entwurfsmustern vertraut machen und Sie in Ihrer Entwicklerkarriere einsetzen. Das bekannteste Buch zu diesem Thema ist Gamma E., Helm R., Johnson R. E. and Vlissides J.: *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995). Die

Autoren werden häufig als „Viererbande“ (*Gang of Four*) und ihr Buch als GoF-Buch bezeichnet. Das GoF-Buch ist nicht die leichteste Lektüre zu diesem Thema, so daß Sie eventuell nach Alternativen suchen werden. Versuchen Sie es mit *Head First: Design Patterns* von Elisabeth Freeman, Eric Freeman, Bert Bates und Kathy Sierra (O'Reilly, 2004), dem „Portland Pattern Repository“ (*http://c2.com/ppr*) oder den Wikipedia-Artikeln zu den einzelnen Entwurfsmustern (beginnen Sie beim Artikel „Entwurfsmuster“ in der deutschen beziehungsweise „Design pattern (computer science)“ in der englischen Ausgabe). Auch die Entwurfsmuster der Java Enterprise Edition von Sun Microsystems (*http://java.sun.com/blueprints/patterns*) sind sehr nützlich. Beachten Sie aber, daß diese Entwurfsmuster auf die Java Enterprise Edition ausgerichtet sind und eventuell in einer Form beschrieben werden, die für Ihre Prüfungsaufgabe nicht nützlich ist.

2.1.4 Dokumentation Ihrer Design-Entscheidungen

[14] Dokumentieren Sie Ihre während der Arbeit an Ihrer Prüfungsaufgabe getroffenen Entscheidungen und die Überlegungen, die Sie zur jeweiligen Entscheidung veranlaßt haben. Wenn Sie beispielsweise für eine interne Datenstruktur ein `ArrayList`-Objekt anstelle eines `Vector`-Objektes gewählt haben, notieren Sie die Tatsache, daß die Klasse `ArrayList` nicht synchronisiert ist und gewählt wurde, um eine leichter gewichtige Datenstruktur zu verwenden. Übertreiben Sie es aber nicht mit dieser Dokumentation, seien Sie also aufmerksam und lassen Sie die Vernunft walten. Diese Art Dokumentation ist eine wichtige Informationsquelle bei der Fehlersuche und beim Verbessern der Performanz. Darüberhinaus ist sie ein guter Anhaltspunkt für diejenigen, die sich mit dem Quelltext auseinandersetzen müssen.

[15] Warten Sie mit der Dokumentation Ihrer Design-Entscheidungen nicht bis zum Ende des Projektes. Wenn Sie mit dieser Dokumentation warten, bis Sie die Entwicklung abgeschlossen haben, müssen Sie nicht nur aus dem Gedächtnis rekonstruieren, warum Sie eine bestimmte Entscheidung getroffen haben, sondern auch, welche Alternativen Sie damals zur Auswahl hatten. Wenn Sie sich zu Beginn der Arbeit beispielsweise für ein anwendungsspezifisches Dialogfenster entschieden haben, erinnern Sie gegen Ende des Projektes möglichlicherweise nicht mehr, daß Sie ursprünglich auch eine editierbare Zelle im Hauptfenster in Betracht gezogen haben.

[16] Gegen Ende des Projektes haben Sie unter Umständen ein umfangreiches Dokument mit Design-Entscheidungen erarbeitet. Weniger wichtige Entscheidungen können herausgenommen werden, um das Dokument handlicher zu machen.

Tipp: Verwenden Sie Aufzählungspunkte (*bullet points*) für die Dokumentation Ihrer Design-Entscheidungen. Sie reduzieren dadurch nicht nur die Textmenge, sondern können sich die Einträge auch leichter merken, wenn Sie die schriftliche Prüfung ablegen. Das ist besonders für Prüfungskandidaten wichtig, deren Muttersprache nicht Englisch ist, da Sie weniger auf Buchstabierung und Satzbau achten müssen.

---

[17] Beispiel für die Dokumentation zweier Entscheidungen:

- Sockets statt RMI: Stets völlige Kontrolle über Threads.
- `RandomAccessFile` statt separater `DataInputStream`- und `DataOutputStream`-Objekte, da Zugriff auf eine beliebige Position in der Datei möglich.

2.1.5 Modultest und systemweites Testen

[18] Schreiben Sie zu jeder Klasse Ihrer Prüfungsaufgabe einen Modultest (*unit test*) oder verwenden Sie ein Testwerkzeug wie JUnit (*http://www.junit.org*), um die Testklassen automatisch zu generieren. Es ist sinnvoll, die Testklasse vor der Implementierung der zu testenden Methoden anzulegen.

[19] Schreiben Sie Ihre Testklassen, bevor Sie mit der eigentlichen Entwicklung Ihrer Anwendung beginnen und achten Sie darauf, daß Ihre Testklassen die schriftlichen Anforderungen an die Anwendung abdecken. Weist eine unter dieser Voraussetzung angelegte Testklasse nach, daß die getestete Anwendungsklasse die in den Anforderungen beschriebene Funktionalität erfüllt, so war der Test erfolgreich. Zusätzliche Funktionalität über die Anforderungen hinaus ist nicht nötig. Das Entwickeln der Testklasse vor der zugehörigen Anwendungsklasse verhindert, daß ein übereifriger Entwickler bei einer Funktionalität verweilt, die nicht verlangt ist.

[20] Während der Arbeit an Ihrer Prüfungsaufgabe wird es vorkommen, daß Sie in einer Anwendungsklasse eine weitere Methode anlegen oder eine bereits vorhandene Methode ändern. Achten Sie darauf, Ihre Testklassen zusammen mit den zugehörigen Anwendungsklassen zu aktualisieren. Es wird generell empfohlen, Testklassen nicht zu löschen. Einmal geschrieben, liefert eine Testklasse die wertvolle Information, ob die grundlegende Funktionalität der Anwendungsklasse nach einer Änderung noch gegeben ist.

Bemerkung: Ein Modultest testet eines Ihrer Softwaremodule (in der Regel eine Klasse). Hat Ihre Anwendungsklasse zum Beispiel eine Methode namens `setName(String Name)`, dann sollte die Testklasse diese Methode so oft aufrufen, bis der gesamte Wertebereich des Argumentes `dvdName` abgedeckt ist. Verfahren Sie mit jeder Methode auf diese Weise. Modultests gehören nicht zum Prüfungsumfang aber wir empfehlen sie dringend als Probe für den „Gesundheitszustand“ Ihrer Prüfungsaufgabe. Die Testklassen gehören *nicht* zu Ihrer bei Sun Microsystems eingereichten Lösung.

---

[21] Wenn die Anwendung zum systemweiten Test bereit ist, rekrutieren Sie ein paar Freiwillige zur Unterstützung. Es ist in der Regel sinnvoll, daß Sie den systemweiten Test Ihrer Anwendung *nicht* selbst durchführen, sondern ein mitleidloses und unbefangenes Auge auf Ihre Arbeit sehen lassen. Es ist sinnvoll, den systemweiten Test gemeinsam mit dem Kunden zu entwickeln. Sofern dies nicht möglich ist (wie bei Ihrer Prüfungsaufgabe), beziehen Sie sich auf die zu Beginn des Projektes erfaßten Anforderungen.

Bemerkung: Ein systemweiter Test prüft die Zusammenarbeit der verschiedenen Module (in der Regel Klassen). Angenommen, eine Anwendungsklasse ruft die `DVD`-Methode `setName(String name)` auf und Sie wissen aufgrund der Modultests, daß sowohl die Anwendungsklasse als auch die Klasse `DVD` als einzelne Module funktionieren. Dann gilt es noch immer, die *Zusammenarbeit der beiden Klassen* zu testen. Beispielweise können sich Netzwerkprobleme ergeben oder die Anwendungsklasse speichert den Namen einer DVD als Array von Zeichen, während die `DVD`-Klasse hierfür ein `String`-Objekt verwendet. Durch systemweites Testen können Sie gewährleisten, daß alle Anwendungsklassen erwartungsgemäß zusammenarbeiten.

2.2 Datei- und Verzeichnisstruktur

[22] Eine der ersten Aufgaben zu Beginn eines Projektes ist die vernünftige Organisation aller projektbezogenen Dinge, im Falle eines Softwareprojektes eine Datei- und Verzeichnisstruktur, in der sämtliche projektrelevanten Dateien und Dokumente abgelegt werden können. Es ist sehr schwierig eine solche Entscheidung erst später Lebenszyklus des Projektes zu treffen oder zu ändern. Die Datei- und Verzeichnisstruktur sollte frühzeitig während der Explorations- und Designphase des Projektes festgelegt werden. Sun Microsystems schreibt bei der Prüfung zum *Sun Certified Java Developer* keine Verzeichnisstruktur vor, legt allerdings bei einigen Aufgaben bestimmte Top-Level-Verzeichnisse fest, die in der eingereichten Lösung verwendet werden müssen. Tabelle 2.1 schlägt eine grundlegende Verzeichnisstruktur für eine Prüfungsaufgabe vor, die auf der Beispielanwendung *Denny's DVDs* in diesem Buch zugrunde liegt.

Unterverzeichnis	**Empfohlene Verwendung**
src	Enthält sämtliche während der Arbeit an der Prüfungsaufgabe geschriebenen *.java* Dateien mit Ausnahme der *.java* Dateien für Modultests (siehe unten).
classes	Enthält die übersetzten *.class* Dateien und alle gepackten *.jar* Dateien. Der Klassenpfad enthält dieses Verzeichnis beim Aufruf der Anwendung.
bkp	Enthält beliebige Dateien, die eventuell als Sicherungskopien benötigt werden.
tst	Enthält *.java* Dateien aus den Modultests der Anwendung (Testklassen).
tmp	Hilfsverzeichnis für temporäre Dateien.
log	Enthält die Protokolldatei(en).
doc	Enthält die gesamte Dokumentation, darunter die API-Dokumentation (Javadoc), die Benutzerdokumentation sowie die Beschreibung der Design-Entscheidungen.

Tabelle 2.1: Für Ihre Prüfungsaufgabe empfohlene Verzeichnisstruktur.

2.3 Dokumentation

[23] Zusätzlich zur Programmieraufgabe verlangt die Prüfung zum *Sun Certified Java Developer* auch Dokumentation in mehreren Varianten. Zum Zeitpunkt der Drucklegung dieses Buches wurde Dokumentation im folgenden Umfang verlangt:

- API-Dokumentation (Javadoc), siehe Abschnitt 2.5.
- Eine einfache Textdatei namens *version.txt*.
- Eine Betriebsanleitung für den Benutzer, sofern sie nicht eingebaut und zur Laufzeit der Anwendung verfügbar ist.
- Dokumentation Ihrer Design-Entscheidungen, siehe Unterabschnitt 2.3.1.

Die Textdatei *version.txt* dokumentiert

- Die Version des zur Entwicklung Ihrer Prüfungsaufgabe verwendeten Java Development Kits.
- Das verwendete Betriebssystem.

Warnung: Sun Microsystems vergibt zur Zeit mehrere verschiedene Prüfungsaufgaben, wobei sich auch die Anleitungen zweier ähnlicher Aufgabe voneinander unterscheiden können. Sun Microsystems wird auch in der Zukunft ähnliche Aufgaben herausgeben, deren Anleitungen sich geringfügig unterscheiden. Die Informationen in diesem Buch sind allgemein gültig. Lesen Sie die Anleitung zu Ihrer Prüfungsaufgabe sorgfältig und achten Sie darauf, sie zu befolgen.

Welche zusätzlichen Dateien sollten Sie einreichen?

Prüfungskandidaten fragen häufig, ob sie beispielsweise ihre Testklassen und/oder ihre Klassendiagramme ebenfalls einreichen sollen.

Unser Rat: Reichen Sie generell nichts ein wonach Sie nicht gefragt wurden. Die Anleitungen von Sun Microsystems beinhalten gegenwärtig den Hinweis, daß Sie keine zusätzlichen Punkte bekommen, wenn Sie über die Anforderungen hinausgehen. Sie haben also nichts davon, wenn Sie Ihrem Gutachter diese Informationen zur Verfügung stellen. Es ist dagegen nicht auszuschließen, daß Sie durch zusätzliche eingereichte Anlagen Ihre Bewertung beeinträchtigen, wenn der Gutachter einen Fehler in einer Datei findet, die Sie nicht hätten abgeben müssen.

Eine Ausnahme vom obigen Rat besteht, wenn eine zusätzlichen Datei das Verständnis Ihrer Lösung sehr erleichtert, zum Beispiel bei einem Klassendiagramm. Anderseits ist die Prüfungsaufgabe einfach. Wenn Sie ein Klassendiagramm brauchen, um Ihre Lösung verständlich zu machen, ist sie wahrscheinlich zu kompliziert.

[24] Eine Betriebsanleitung für die Benutzer ist unbedingt erforderlich. Wenn Sie dem Benutzer den Umgang mit Ihrer Anwendung nicht erklären, wird sie als nutzlos betrachtet. Gehen Sie daher beim Schreiben dieser Dokumentation mit Sorgfalt vor. Sie dürfen nur eines als gesichert voraussetzen: Der Benutzer hat keine Erfahrung mit Ihrer Anwendung. Jeder Schritt muß hier in allen Einzelheiten beschrieben werden, so geringfügig er auch sein mag. Testen Sie die Betriebsanleitung, indem Sie sie einem nichtsahnenden Freund geben (vorzugsweise einem Freund, der kein Entwickler ist). Wenn er der Dokumentation folgen kann, ist die Beschreibung ausreichend.

Warnung: Die gegenwärtigen Anleitungen zu den Prüfungsaufgaben verlangen, daß die Betriebsanleitung für die Benutzer in einem separaten Verzeichnis deponiert wird oder online verfügbar ist. Diese Forderung hat in der Vergangenheit immer wieder Prüfungskandidaten verunsichert und zu der Fehlannahme verleitet, es werde verlangt, einen Webserver zu betreiben. Das ist ein Mißverständnis. Sun Microsystems verlangt lediglich, daß der Gutachter Zugang zur Betriebsanleitung für die Benutzer hat, indem die Anleitung im dafür vorgesehenen Verzeichnis liegt oder zur Laufzeit der Anwendung aufgerufen werden kann (zum Beispiel führt die „F1“-Taste bei Microsoft Word zur Online-Hilfe, auch wenn Sie keine Verbindung zum Internet haben).

2.3.1 Dokumentation Ihrer Design-Entscheidungen

[25] Sun Microsystems schreibt bestimmte Details beim Design und der Implementierung vor, beispielsweise die Verwendung einer `JTable`-Komponente (Swing) in der graphischen Benutzeroberfläche. Andere Entscheidungen müssen Sie dagegen selbst fällen. Sie haben beispielsweise die Wahl, ob Sie die Netzwerkschnittstelle per RMI oder per Sockets implementieren möchten. Jede Variante hat Vor- und Nachteile. Es ist wichtig, Ihre Design-Entscheidungen zu dokumentieren, um sie vor

dem Gutachter verteidigen zu können, der letztendlich entscheidet, ob Sie mit Ihrer eingereichten Lösung die Prüfung bestanden haben oder nicht.

[26] Halten Sie die Design-Entscheidungen zu Ihrer Prüfungsaufgabe in einem Dokument fest. Diese Datei dokumentiert Beispiele für wesentliche Entscheidungen, etwa warum Sie ein bestimmtes Entwurfsmuster verwenden oder einer Variante den Vorzug vor einer anderen geben. Eventuell müssen Sie eine Design-Entscheidung auf der Grundlage einer unklaren funktionalen Anforderung fällen. Falls sich diese Situation ergibt, dokumentieren Sie Ihre Entscheidung zusammen mit allen Annahmen, die Sie zur Lösung des Problems voraussetzen. Schreiben Sie die Dokumentation Ihrer Design-Entscheidungen zu Ende. Es ist die einzige Grundlage, von der ausgehend Sie Ihre Prüfungsaufgabe verteidigen können.

Tipp: Auch bei den von Sun Microsystems gestellten Prüfungsaufgaben kann es mehr als eine Lösung für ein gegebenes Problem geben. Eventuell ist keine dieser Lösungen optimal. Investieren Sie nicht zuviel Zeit in die Suche nach der besten Lösung. Es ist möglich, daß sie nicht existiert. Sun Microsystems hat absichtlich bestimmte Unklarheiten in den Anleitungen belassen, um den Prüfungskandidaten an einigen wenigen Stellen eine begrenzte Entscheidungsfreiheit zu gewähren. In allen übrigen Fällen kommt es nicht darauf an, welche Wahl Sie treffen, sondern wie Sie Ihre Entscheidung begründen. Diese Punkte gehören detailliert ausgearbeitet in die Dokumentation Ihrer Design-Entscheidungen.

2.4 Benennung von Bezeichnern und Formatierung des Quelltextes

[27/28] Ein gemeinsames Ziel aller Vertreter der Softwareindustrie ist es, eine Anwendung zur zukünftigen Pflege an einen anderen Entwickler zu übergeben. Der Quelltext muß dazu so formatiert sein, daß der Ihr Nachfolger Ihre Arbeit akzeptiert. Sie schaden Ihrem Ruf, wenn Ihr Nachfolger Ihren Quelltext als unverständlich empfindet und verwirft. Ebenso würden wir unserem Ruf schaden, wenn wir dieses Buch nicht in Kapitel, Abschnitte und Absätze eingeteilt hätten. Wenn Sie das Buch nicht lesen können, werden Sie nicht viel daraus lernen.

[29] Die Schöpfer der Programmiersprachen Java, C und C++ haben absichtlich darauf verzichtet, den Entwicklern Richtlinien zur Formatierung ihrer Quelltexte aufzuerlegen. Es gibt syntaktische Regeln zu befolgen, aber solange Sie sich an diese Vorgaben halten, können Sie Ihren Quelltext formatieren, wie Sie möchten, zum Beispiel:

```
public class MyTest {
    public static void main(String[] args) {
        System.out.println("Hello");
    }
}
```

Dieser Quelltext wird vom Compiler in exakt derselben Weise interpretiert wie dieser:

```
public
class
MyTest{public static void main(String
args[]){System.out.println("Hello");}}
```

Sie stimmen zweifellos zu, daß die obere Variante erheblich besser lesbar ist, als die untere.

[30] Man kommt schnell zur Überzeugung, daß es eine Richtlinie für die Formatierung des Quelltextes geben sollte. Es ist allerdings schwierig, sich auf eine gemeinsame Richtlinie zu einigen. Die einen Entwickler bevorzugen beispielsweise die öffnende geschweifte Klammer (`{`) am Zeilenende,

andere aber am Anfang einer neuen Zeile. Beide Entwickler haben gute Argumente für Ihren jeweiligen Standpunkt, aber sinnvollerweise sollte innerhalb eines Projektes genau eine Richtlinie befolgt werden.

[31] In der Regel schreibt das Unternehmen vor, welche Formatierungsrichtlinie gilt. Wir empfehlen für Ihre SCJD-Prüfungsaufgabe die „Sun Microsystems Code Conventions for the Java Programming Language“, die Sie von der Internetadresse *http://java.sun.com/docs/codeconv* herunter laden können. Sun Microsystems definiert elf Kategorien, in denen eine Benennungs- beziehungweise Formatierungsrichtlinie benötigt wird. Wir widmen uns jeder Kategorie in einem eigenen Unterabschnitt. Die Hauptkategorien sind: die Benennung von Bezeichnern (Unterabschnitt 2.4.1), die Struktur einer *.java* Datei (Unterabschnitt 2.4.2), die Formatierung des Quelltextes (Unterabschnitt 2.4.3) und die Formatierung von Kommentaren (Unterabschnitt 2.4.4).

2.4.1 Benennung von Bezeichnern

[32/33] Die Benennungsrichtlinie von Sun Microsystems sieht für Packages, Klassen und Interface, Methoden Felder und Variablen sowie Konstanten unterschiedliche Schemata vor. Vermeiden Sie grundsätzlich Abkürzungen, es sei denn, daß eine Abkürzung häufiger als ihre expandierte Bedeutung verwendet wird. Achten Sie darauf, daß Ihre Bezeichner nicht unnötig lang werden, da sie sonst mühsam zu lesen und zu schreiben sind. Tabelle 2.2 zeigt je ein Beispiel.

2.4.1.1 Bezeichner für Packages

[34] Ein Packagename beginnt mit Ihrem vollqualifizierten Domainnamen in Kleinbuchstaben, wobei die einzelnen „Wörter“ zurückwärts angeordnet werden. Wenn Ihr Unternehmen den Domainnamen `example.com` hat, beginnt Ihr Packagename mit `com.example`.

[35] Anschließend befolgen Sie die Benennungsrichtlinie Ihres Unternehmens. Ein typisches Schema ist der Projektname, gefolgt von einer funktionalen Gruppierung der einzelnen Klassen. Die Prüfung zum *Sun Certified Java Developer* beinhaltet unter anderem eine graphische Benutzeroberfläche. Daher könnten Sie die entsprechenden Klassen in einem Package namens `com.example.edu.scjd.gui` zusammenfassen. Wir haben das Basispackage unserer Beispielanwendung *Denny's DVDs* der Einfachheit halber `sampleproject` genannt. Die Klassen für die graphische Benutzeroberfläche der Beispielanwendung liegen daher im Package `sampleproject.gui`.

Tipp: Bei Ihrer Prüfungsaufgabe ist in der Regel die Packagezugehörigkeit mindestens einer Klasse vorgegeben. Sie brauchen also sich nicht darum zu kümmern, wenn Sie keinen Domainnamen haben, aus dem Sie den Namen des Basispackages ableiten können.

Inhalt	Guter Bezeichner	Schlechter Bezeichner
Kontostand	`accountBalance`	`userAccountBalance` (zu lang), `ab` (keine allgemeine verständlich Abkürzung, Abkürzung ohne Aussage)
HTML-Editorklasse	`HtmlEditor`	`HyperTextMarkupLanguageEditor` („HTML“ ist eine allgemein verständliche Abkürzung, deren Verwendung den Klassennamen leichter lesbar macht).

Tabelle 2.2: Beispiele für gute und schlechte Bezeichner.

2.4.1.2 Bezeichner für Klassen und Interfaces

[36] Der Name einer Klasse oder eines Interfaces beginnt stets mit einem großen Buchstaben und ist ein Substantiv (der Name beschreibt ein Objekt und keine Aktion dieses Objektes). Beispielweise können Sie eine Klasse, die Informationen über ein Buch enthält `Book` nennen.

[37] Häufig werden zwei oder mehr Substantive oder ein Adjektiv und ein Substantiv zu einem Namen zusammengefaßt, wobei Binnenmajuskeln (*camel case*) verwendet werden (der erste Buchstabe jedes Wortes ist ein Großbuchstabe, wodurch sich ein wellenförmiges Muster ergibt, das an die Höcker eines Kamels erinnert). Beispielweise können Sie eine Fabrikklasse, die Socketverbindungen erzeugt `SocketFactory` nennen.

Tipp: Eine Klasse sollte nicht mehr als eine Aufgabe haben. Eine Klasse, die via Remote Method Invocation (RMI) eine Verbindung zu einem Server aufbaut, sollte nicht zugleich für die Datenvisualisierung verantwortlich sein. Hat eine Klasse nur eine Aufgabe, so fällt auch die Wahl ihres Namens leichter. Gut gewählte Bezeichner zahlen sich zum Beispiel später aus, wenn eine Klasse gepflegt oder überarbeitet werden muß. Durch Aufgabenteilung und gute Bezeichner läßt sich die entsprechende Klasse leichter finden.

2.4.1.3 Bezeichner für Methoden

[38] Der Name einer Methode beginnt stets mit einem kleinen Buchstaben und einem Verb (der Name beschreibt eine Aktion des Objektes). Der Name der Methode `getLeadActor()` der `DVD`-Klasse unserer Beispielanwendung gibt beispielsweise an, daß diese Methode den Namen des Hauptdarstellers zurückgibt.

[39] Häufig werden mehrere Wörter kombiniert, um die Funktion einer Methode zu beschreiben. Der Bezeichner `getLeadActor()` stellt unmißverständlich klar, daß wir per Aufruf dieser Methode den Namen des Hauptdarstellers abfragen können (und nicht den Namen einer anderen, mit der DVD verbundenen Person). Auch hier sind die einzelnen Worte durch Binnenmajuskeln miteinander verbunden.

2.4.1.4 Bezeichner für Felder und lokale Variablen

[40] Der Name eines Feldes beziehungsweise einer lokalen Variablen beginnt stets mit einem kleinen Buchstaben, ist kurz und beschreibt, welche Art von Information in diesem Feld beziehungsweise dieser Variablen gespeichert wird. Das Feld `leadActor` der Klasse `DVD` enthält beispielsweise den Namen des Hauptdarstellers des Films auf der DVD.

[41] Wiederum werden häufig mehrere Wörter kombiniert, um die Funktion des Feldes beziehungsweise der lokalen Variablen zu beschreiben und auch hier werden die einzelnen Worte mit Binnenmajuskeln verbunden.

Bemerkung: Die Benennungsrichtlinie von Sun Microsystems verlangt, daß Sie für alle Objektfelder, Klassenfelder und lokalen Variablen konsistent dasselbe Schema anwenden. Eventuell sehen Sie in Quelltexten anderer Entwickler, daß Objekt- oder Klassenfelder durch einen Unterstrich oder ein anderes Zeichen gekennzeichnet sind. Wiederum ein anderes Schema schreibt die Notationen `variable` für lokale Variablen, `this.variable` für Objektfelder und `Class.variable` für Klassenfelder vor. Das letztere Schema verdeutlicht explizit, welche Art von Bezeichner vorliegt.

2.4.1.5 Bezeichner für Konstanten

[42] Der Name einer Konstanten ist stets durchgängig in Großbuchstaben geschrieben, wobei einzelne Wörter durch Unterstriche getrennt werden. Beispielsweise könnte die Konstante `DIRECTOR_LENGTH` die Maximallänge des Namens des Direktors in der Datenbankdatei angeben.

2.4.2 Struktur einer .java Datei

[43] Die Quelltextdateien von Java-Klassen und Interfaces haben stets die folgende standardisierte Struktur (Hauptabschnitte):

- Einleitender Kommentar.
- `package`- und `import`-Anweisungen.
- Deklaration von Klassen Interfaces.

Die Formatierungsrichtlinie von Sun Microsystems sieht zwischen diesen Hauptabschnitten je zwei Leerzeilen vor, das heißt zwei Leerzeilen zwischen dem einleitenden Kommentar und den `package`-/`import`-Anweisungen sowie zwischen diesen und der Klassen- beziehungsweise Interfacedeklaration. In allen übrigen Fällen, in denen eine Leerzeile die Lesbarzeit verbessert (zum Beispiel zwischen zwei Methodendeklarationen), steht in der Regel nur eine einzelne Leerzeile. Beispiel:

```
01. /*
02.   * HelloWorld.java      version 1.0.0     date 2005-06-20
03.   * Copyright (c) Andrew Monkhouse & Terry Camerlengo 2005
04.   *
05.   * This is a version of the hello world program
06.   * The beginning comment, has two blank lines following it
07.   */
08.
09.
10. package com.example.javaExample;
11.
12. import java.util.Date;
13.
14.
15. public class HelloWorld {
16.     public static void main(String[] args) {
17.         sayHello();
18.     }
19.
20.     public static void sayHello() {
21.         System.out.println("Hello, word at " + new Date() + "\n");
22.     }
23. }
```

[44] Zwischen den Hauptabschnitten befinden sich jeweils zwei Leerzeilen. Die Zeilen 8 und 9 trennen den einleitenden Kommentar von den `package`-/`import`-Anweisungen. Die Zeile 13 und 14 trennen die `package`-/`import`-Anweisungen von der Klassendeklaration. Die Unterabschnitte sind stets nur durch eine einzelne Leerzeile voneinander getrennt. Zeile 11 trennt die `package`-Anweisung von der ersten `import`-Anweisung und Zeile 19 trennt den Konstruktor von den Methodendeklarationen.

2.4.2.1 Einleitender Kommentar

[45] Der einleitende Kommentar ist von den Dokumentationskommentaren (Javadoc) getrennt und wird von Java-Entwicklern häufig mißverstanden. Der einleitende Kommentar enthält Informationen, die sich teilweise auch in den Dokumentationskommentaren der Klasse befinden, wobei sich beide Kommentartypen in wesentlichen Punkten unterscheiden: Der einleitende Kommentar steht immer an derselben Stelle und es handelt sich um very specific information. Dokumentationskommentare können zwar Informationen aus dem einleitenden Kommentar enthalten, sind aber nicht an eine bestimmte Zeilennummer gebunden und die gewünschte Information kann unter der API-Dokumentation „verschüttet“ sein. Der einleitende Kommentar enthält die folgenden Informationen:

- Den Klassennamen.
- Die Versionsnummer.
- Das Anlegedatum beziehungsweise Datum der letzten Änderung.
- Autor der Datei beziehungsweise der letzten Änderung.

Versionsnummer, Anlege-/Änderungsdatum und Autor können von Ihrer Versionsverwaltung automatisch ausgefüllt werden. Der Kommentarblock ist ein C-Kommentar, kein Dokumentationskommentar, das heißt der Block beginnt mit `/*` statt `/**`.

2.4.2.2 package- und import-Anweisungen

[46/47] Dem einleitenden Kommentar folgen die `package`-Anweisung, eine Leerzeile und die `import`-Anweisungen, siehe Zeilen 10–12 im obigen Beispiel. Die Formatierungsrichtlinie von Sun Microsystems gibt nicht an, ob jede Klasse einzeln oder das gesamte Package importiert werden soll. Häufig werden bis zu drei Klassen eines Packages einzeln zu importiert, danach das ganze Package.

[48] Die Formatierungsrichtlinie von Sun Microsystems gibt ebenfalls nicht an, ob die `import`-Anweisungen in einer bestimmten Reihenfolge angeordnet werden sollen. Das Nachdenken über solche Dinge liegt wahrscheinlich jenseits des durch die Prüfungsanforderungen gegebenen Rahmens (und kann Ihre Kollegen veranlassen, sich über Sie lustig zu machen). Viele Entwickler ordnen die Packages alphabetisch an, aber machen Sie sich nicht zu viele Gedanken hierüber.

Tipp: Viele integrierte Entwicklungsumgebungen bieten einige der folgenden Eigenschaften beziehungsweise Fähigkeiten: automatisches Hinzufügen fehlender `import`-Anweisungen, automatisches Entfernen nicht benötigter `import`-Anweisungen, automatische Änderung der `import`-Anweisung(en) bei zu vielen oder zu wenigen Importen aus ein und demselben Package. Obwohl diese Automatismen in der Berufspraxis ihre Arbeit unterstützen, empfehlen wir, sie während der Arbeit an Ihrer Prüfungsaufgabe zu deaktivieren. Eines der Probleme, die solche Automatismen mit sich bringen ist, daß sie die Fehlersuche erschweren können. Wenn Sie den manuellen Umgang mit `import`-Anweisungen im Rahmen Ihrer Prüfungsvorbereitung erlernen, wird Ihnen die Fehlersuche später leichter fallen.

---

[49] Häufig werden die `import`-Anweisungen für Packages aus der Java Standard Edition, Packages aus der Java Enterprise Edition, für externe und interne Packages durch eine Leerzeile voneinander getrennt.

2.4.2.3 Deklaration von Klassen und Interfaces

[50] Die Formatierungsrichtlinie von Sun Microsystems gibt an, daß eine Klassen- oder Interfacedeklaration die folgenden Elemente in dieser Reihenfolge enthalten soll:

- Einen Dokumentationskommentar (Javadoc) zu dieser Klassen- oder Interfacedeklaration.
- Das Schlüsselwort `class` beziehungsweise `interface`.
- Klassenfelder.
- Objektfelder.
- Konstruktoren.
- Methoden.

Felder sollen nach ihrem Zugriffskontrollmodifikator sortiert werden: `public`, `protected`, Standardzugriff, `private`, zum Beispiel:

```
public class VariableOrderExample {
    public int aVariableModifiableByAnyOtherClass;
    public String anotherPublicVariable;
    // protected variables appear after public variables
    protected int protectedVariable;
    // now list the variables with default access
    Character defaultAccessVariable;
    // finally list the variables with private access
    private int noOtherClassCanSeeMe;
```

Tipp: Die Formatierungsrichtlinie von Sun Microsystems gibt nicht an, wo Konstanten angeordnet werden sollen. In der Regel werden Konstanten vor den Klassenfeldern deklariert.

[51] Methoden sollen dagegen nach Funktionalität und nicht bezüglich ihres Zugriffskontrollmodifikators sortiert werden, das heißt Sie sollen eine private Methode in der Nähe der öffentlichen Methode platzieren, die sie aufruft.

2.4.3 Formatierung des Quelltextes

[52] Achten Sie während des Programmierens darauf, die folgenden Elemente konsistent zu formatieren:

- Einrückungen.
- Zeilenlänge und Zeilenumbruch.
- Leerraum.
- Anweisungen.
- Feld- und Variablendeklarationen.

[53] Die meisten dieser Formatierungsregeln sind so gewählt, daß andere Menschen die Ihren Quelltext lesen, eine integrierte Entwicklungsumgebung, einen Texteditor und eine Bildschirmauflösung ihrer Wahl benutzen können. Wenn Sie keiner gebräuchlichen Richtlinie folgen, ist Ihr Quelltext unter Umständen für andere schwer lesbar.

Warnung: Vergessen Sie niemals, daß Sie Ihren Quelltext bei einem unbekannten Gutachter einreichen. Formatieren Sie Ihren Quelltext so, daß es eine Freude ist, ihn zu begutachten. Das gilt auch für Ihre richtige Arbeit: Sie sollten Ihre Quelltexte immer so schreiben, daß Ihre Kollegen gerne damit arbeiten.

2.4.3.1 Einrückungen

[54/55] Aus der Formatierungsrichtlinie von Sun Microsystems: „Four spaces should be used as the unit of indentation. The exact construction of the indentation (spaces vs. tabs) is unspecified. Tabs must be set exactly every 8 spaces (not 4)“, übersetzt etwa „Die Einheit für Einrückungen sind vier Leerzeichen. Ob die Einrückung aus Leerzeichen oder Tabulatoren besteht ist nicht festgelegt. Tabulatoren werden alle acht Zeichen (nicht vier) gesetzt.“ Ein Beispiel:

```
1. public class IndentationExample {
2.     /* this line is indented once */
3.     public IndentationExample() {
4.         /* this line is indented twice */
5.     }
6. }
```

[56] Die Zeilen 2, 3 und 5 haben jeweils eine einfache Einrückung, also vier Leerzeichen. Zeile 4 hat eine doppelte Einrückung, also acht Leerzeichen. Acht Leerzeichen sind gleichbedeutend mit einem Tabulator, das heißt Sie können in Zeile auch einen Tabulator anstelle der acht Leerzeichen verwenden. Wir empfehlen acht Leerzeichen anstelle eines Tabulators.

Tipp: Viele integrierte Entwicklungsumgebungen setzen pro Druck der Tabulatortaste eine bestimmte Anzahl von Leerzeichen ein (und berücksichtigen diese Einstellung auch beim Löschen der Einrückung und der Neuformatierung ganzer Blöcke von Anweisungen). Sehen Sie nach, ob Ihre Entwicklungsumgebung oder Ihr Texteditor diese Funktionalität unterstützen und aktivieren Sie sie, falls vorhanden.

2.4.3.2 Zeilenlänge und Zeilenumbruch

[57] Sie wissen nicht, welchen Texteditor oder welche Bildschirmeinstellungen Ihr Gutachter verwendet. indem Sie die Zeilenlänge auf 80 Zeichen beschränken, gewährleisten Sie, daß Ihr Quelltext in den meisten Fällen lesbar ist.

[58] Zeilen mit mehr als 80 Zeichen werden nach einem Komma oder vor einem Operator umgebrochen. Die restliche Zeile (und eventuelle ebenfalls umgebrochene Folgezeilen) werden bis zum Beginn des umgebrochenen Ausdrucks in der vorigen Zeile oder aber um acht Leerzeichen eingerückt, zum Beispiel:

```
int i = myMethod(longNamedVariable1, longNamedVariable2
                 longNamedVariable3, longNamedVariable4);
```

Beim Zeilenumbruch einer `if`-, `for`- oder `while`-Anweisung wird die Folgezeile generell um acht Leerzeichen eingerückt und nicht nach dem Anfang des Ausdrucks ausgerichtet, da sich die zweite Hälfte der umgebrochene Zeile andernfalls optisch nicht deutlich genug von der um vier Leerzeichen eingerückten nächsten Anweisung unterscheidet. Das nächste Beispiel zeigt sowohl das bevorzugte als auch das nicht bevorzugte Umbrechen einer `if`-Anweisung:

```
// nonpreferred way
if (myMethod(longNamedVariable1, longNamedVariable2
    longNamedVariable3, longNamedVariable4)) {
    // code starts here - see how confusing this is ?
    doSomething();
}

// preferred way
if (myMethod(longNamedVariable1, longNamedVariable2
        longNamedVariable3, longNamedVariable4)) {
    // code starts here - now we can see the difference between
    // the condition and the code to be run within the condition
    doSomething();
}
```

Enthält eine Zeile mehrere Anweisungsebenen, etwa wenn das Ergebnis eines Methodenaufrufs als Parameter in einen anderen Methodenaufruf einsetzt wird, so sollte die Zeile an der höheren Ebene umgebrochen werden:

```
int i = myMethod(variable1,
                 callToAnotherMethod(variable1, variable2),
                 (variable1 + variable2));
```

Belassen Sie den äußeren Methodenaufruf möglichst in einer Zeile. In solchen Fällen sollten Sie in Betracht ziehen, den Quelltext umzuschreiben, um seine Lesbarkeit zu verbessern. Einige Vorschläge:

- Rufen Sie die innere Methode in einer separaten Zeile auf.
- Führen Sie die eingeklammerte Addition in einer separaten Zeile durch.
- Wenden beide Alternativen zugleich an.

Sehen Sie selbst, wie viel besser die folgenden Zeilen zu lesen und zu warten sind:

```
int dbValue = callToAnotherMethod(variable1, variable2);
int calculated = variable1 + variable2;
int i = myMethod(variable1, dbValue, calculated);
```

[59] Wenn Sie die dritte oder vierte Einrückungsebene überschritten haben, wird es anstrengend, sich an die Formatierungsrichtlinie zu halten. Ihr Quelltext ist offenbar schwierig zu lesen. Überlegen Sie sich, ob Sie nicht einen Teil der eingerückten Anweisungen in eine separate Methode auslagern.

2.4.3.3 Leerraum

[60] Leerraum wird verwendet, damit der Quelltext leichter lesbar wird, kann aber bei übermäßigem Gebrauch auch die gegenteilige Wirkung haben. Im allgemeinen setzen Sie ein Leerzeichen zwischen einem Schlüsselwort und einer Klammer, nach einem Komma, zwischen den Ausdrücken im Kopf einer `for`-Schleife, nach einer Typumwandlung sowie beidseitig um alle binären Operatoren. Tabelle 2.3 zeigt einige Beispiele.

2.4.3.4 Formatierung von Anweisungen

[61/62] Die meisten Regeln für die Formatierung von Anweisungen ergeben sich aus den bereits diskutieren Richtlinien. Die meisten Entwickler bevorzugen eine bestimmte Notation für zusammengesetzte Anweisungen, zum Beispiel wo sie geschweifte Klammern setzen und wo diese Klammern optional sind. Sun Microsystems stellt diesen Aspekt nicht zur Diskussion, sondern verlangt, daß die

Regel	Beispiel
Ein Leerzeichen zwischen Schlüsselwort und öffnender Klammer.	`while (true)`
Ein Leerzeichen nach jedem Komma.	`myMethod(variable1, variable2);`
Ein Leerzeichen zwischen den Ausdrücken einer `for`-Schleife.	`for (expression1; expression2; expression3) {`
Ein Leerzeichen nach Typumwandlung.	`int i = (int) aLongValue;`
Je ein Leerzeichen zu beiden Seiten jedes binären Operators.	`a = b + c;` oder `c = 5 * 10;`

Tabelle 2.3: Beispiele zur Verwendung von Leerraum.

Anweisungen in einer Kontrollstruktur *grundsätzlich* in geschweifte Klammern gesetzt werden *müssen*, insbesondere also, wenn die Kontrollstruktur nur eine einzige Anweisung enthält. Beispielsweise muß die einzelne Anweisung in der folgenden `if`-Struktur in geschweiften Klammern stehen:

```
if (variable == someVariable) {
    doSomething();
}
```

[63] Die Anweisung zwischen den Klammern ist einfach eingerückt (vier Leerzeichen), die schließende geschweifte Klammer steht in einer separaten Zeile und in derselben Spalte wie `if`.

Tipp: Es gibt viele Werkzeuge, um zu prüfen, ob Ihr Quelltext der Formatierungsrichtlinie von Sun Microsystems gehorcht. Eines davon ist Checkstyle (*http://checkstyle.sourceforge.net*). Checkstyle läßt sich praktischerweise in viele Entwicklungsumgebungen integrieren und unterstützt viele verschiedene Formatierungsrichtlinien, nicht nur die von Sun Microsystems. Ein gutes Formatierungsprüfwerkzeug liefert einen Bericht über die Stellen, die geändert werden müssen, so daß Sie jedes Vorkommen selbst verifizieren können.

Warnung: Vermeiden Sie bei Ihrer Prüfungsaufgabe automatische Formatierungsprogramme. Gelegentlich formatieren solche Werkzeuge Ihren Quelltext zwar gemäß *der Richtlinie* um, allerdings nicht so wie Sie (oder Ihr Gutachter) es wollen. Die Änderungen lassen sich nur mühsam rückgängig machen und die meisten automatischen Formatierungsprogramme gestatten keine einfache Kontrolle der beabsichtigen Änderungen, so daß Sie einen Vorschlag entweder akzeptieren oder ablehnen können.

2.4.3.5 Formatierung von Feld- und Variablendeklarationen

[64/65] Felder und lokale Variablen werden einzeilig deklariert, vorzugsweise mit einem Kommentar im Anschluß an die Deklaration. Die Formatierungsrichtlinie von Sun Microsystems besagt, daß Sie einen Feld- oder Variablennamen entweder durch ein Leerzeichen oder einen Tabulator von der Typangabe trennen können. Zuviele Tabulatoren oder Leerzeichen bringen allerdings zeitraubendes Neuformatieren mit sich, wenn Sie später ein neues Feld oder eine neue lokale Variable deklarieren. Wir empfehlen ein einfaches Leerzeichen zwischen Feld-/Variablentyp und -name.

[66] Lokale Variablen sollten stets bei ihrer Deklaration initialisiert werden. Außerdem sollten Variablen zu Beginn des „engsten“ geschweiften Klammerpaares deklariert werden, welches den benötigen Geltungsbereich umfaßt. Deklarieren Sie ein Feld oder eine lokale Variable *nicht* erst dann, wenn

Sie sie benötigen.

2.4.4 Formatierung von Kommentaren

[67–69] In Java-Quelltexten sind zwei Sorten von Kommentaren erlaubt: Dokumentationskommentare (Javadoc-Kommentare) und Implementierungskommentare (die übrigen Kommentare). Dokumentationskommentare werden im Abschnitt 2.5 in diesem Kapitel detailliert behandelt. Im Augenblick genügt es zu wissen, daß aus Dokumentationskommentaren API-Dokumentation generiert wird, die anderen Entwickler dabei hilft, die Aufgabe Ihre Klasse und die Verwendung ihrer Konstanten, Methoden und eventuell auch Felder zu verstehen, ohne den Quelltext lesen zu müssen. Dokumentationskommentare sind für außenstehende Entwickler vorgesehen und beschreiben die Funktionsweise der Klasse als Ganzes, aber keine Implementierungsdetails. Auf der anderen Seite dienen Implementierungskommentare Entwicklern (und Gutachtern) als Hinweise, wenn sie den Quelltext durchsehen und zu verstehen versuchen, wie er funktioniert.

Warnung: Übertreiben Sie nicht mit Implementierungskommentaren in Ihrem Quelltext. Wenn Sie für Ihre Klassen, Methoden, Felder und lokalen Variablen sinnvolle Namen wählen, dokumentiert sich Ihr Quelltext in der Regel selbst. Zu viele Kommentare sind kontraproduktiv, da sie vom Quelltext ablenken und schnell veralten.

[70/71] Es gibt zwei Varianten von Implementierungskommentaren, nämlich den Blockkommentar (zwischen den Begrenzungssymbolen `/*` und `*/`) sowie den Zeilenkommentar (beginnt mit `//` und reicht bis zum Zeilenende. Für eine einzelne Kommentarzeile oder einen Kommentar am Zeilenende empfiehlt sich der `//`-Kommentar:

```
// this is an example of a comment using a single line of text
doSomething();        // this is an example of a comment at the end of a line
```

[72] Paßt ein Kommentar nicht in eine Zeile, so verwenden Sie einen Blockkommentar:

```
/*
 * This comment explains why the following code must be used instead of a more
 * ”intuitive” way. As it takes more than one line, it is in a block comment.
 */
```

[73] Vermeiden Sie Blockkommentare beim Auskommentieren von Zeilen im Quelltext und verwenden Sie statt dessen `//`-Kommentare. Das Auskommentieren mit Blockkommentaren erschwert anderen Entwicklern zu bestimmen, welcher Teil des Quelltextes verwendet wird und welcher Teil auskommentiert ist. Ein Zeilenkommentar sollte stets so weit eingerückt werden, wie die kommentierten Anweisungen.

2.4.5 Formatierung der neuen Eigenschaften und Fähigkeiten des JDK 5

[74] Die Formatierungsrichtlinie von Sun Microsystems wurde im Hinblick auf die neuen Eigenschaften und Fähigkeiten von Version 5 des Java Development Kits noch nicht aktualisiert. In diesem Abschnitt erklären wir einige dieser Neuerungen und verwenden dabei die Formatierung aus der API-Dokumentation zu Version 5 des Java Development Kits, den Quelltextbeispielen von Sun Microsystems und dem Java Specification Request (JSR), welches die jeweilige neue Eigenschaft oder Fähigkeit beschreibt.

Bemerkung: Vor Beginn der Entwicklung von Version 5 des Java Development Kits hat Sun Mi-

crosystems die Java-Entwickler befragt, welche Eigenschaften und Fähigkeiten sie sich für die neue Version wünschen. Alle Eingaben wurden beachtet und die Entwickler aufgefordert über die aus ihrer Sicht wertvollsten Neuerungen abzustimmen. Den ausgewählten Anforderungen wurden JSR-Nummern zugewiesen und die wichtigsten Anforderungen in Version 5 des Java Development Kits eingebaut. Auf der Website des Java Community Process (JCP), *http://jcp.org/en/home/index* finden Sie weitere Informationen und haben ein Mitspracherecht bei zukünftigen Verbesserungen.

---

[75] Wir belassen es in diesem Kapitel bei einer kurzen Übersicht über die neuen Eigenschaften und Fähigkeiten von Version 5 des Java Development Kits und besprechen die Einzelheiten, während wir sie in den folgenden Kapiteln in den Kontext unserer Beispielanwendung einbetten.

2.4.5.1 Generische Kollektionen

[76] Die Unterstützung zur Laufzeit typsicherer generischer Kollektionen gestattet, bereits zur Übersetzungszeit festzulegen, welchen Elementtyp eine generische Kollektion zur Laufzeit enthalten wird. Die in diesem Unterabschnitt verwendeten Formatierungsregeln sind am Ende zusammengefaßt. Das folgende Beispiel ist *nicht* typsicher:

```
public List getBreed(String breedName) {
    List dogs = new Array();
    // do some work to find the correct dogs
    String dogName = breedName;
    Dog pooch = new Dog(dogName);
    dogs.add(pooch);
    return dogs;
}

public void listDogs() {
    Collection c = getBreed("labrador");
    for (Iterator i = c.iterator(); i.hasNext(); ) {
        String name = ((Dog) i.next()).getName();
        System.out.println(name);
    }
}
```

[77/78] Obwohl wir wissen, daß das von `getBreed()` zurückgegebene *`List`*-Objekt eine Liste von `Dog`-Elementen enthält, kann ein anderer Entwickler den Quelltext erneut übersetzen, wobei die Liste diesesmal `Cat`-Objekte enthält. Die `listDogs()`-Methode läßt sich anstandslos übersetzen, wirft aber zur Laufzeit eine Ausnahme vom Typ `ClassCastException` aus. Zur Abhilfe müßten Sie per `instanceof`-Operator eine explizite Typprüfung veranlassen und/oder die `ClassCastException`-Ausnahme abfangen.

[79] Es ist aber besser, dafür zu sorgen, daß die generische Kollektion zur Laufzeit ausschließlich `Dog`-Objekte aufnehmen kann. Betrachten Sie nun die beiden geänderten Versionen der Methoden `getBreed()` und `listDogs()`:

```
public List<Dog> getBreed(String breedName) {
    List<Dog> dogs = new ArrayList<Dog>();
    // do some work to find the correct dogs
    String dogName = breedName;
    Dog pooch = new Dog(dogName);
    dogs.add(pooch);
    return dogs;
}

public void listDogs() {
```

```
        Collection<Dog> c = getBreed("labrador");
        for (Iterator<Dog> i = c.iterator(); i.hasNext(); ) {
            String name = i.next().getName();
            System.out.println(name);
        }
    }
```

[80–82] Wir deklarieren, daß die `getBreed()`-Methode eine Liste von `Dog`-Objekten zurückgibt. (Lesen Sie *`List<Dog>`* als „*`List`* of `Dog`s“.) Beachten Sie, daß die Zeile `String name = i.next().getName();` keine Typumwandlung mehr enthält. Der Typ der verarbeiteten Objekte ist durch die Definition des Iterators im ersten Ausdruck der `for`-Schleife festgelegt. Der Versuch, eine Referenz aus dieser Kollektion in einen nicht mit `Dog` verwandten Typ umzuwandeln endet mit der folgenden Fehlermeldung des Compilers:

```
GenericExample.java:16: inconvertible types
found   : Dog
required: Cat
            String name = ((Cat) i.next()).getName();
                                ^

1 error
```

[83] Beachten Sie bei der Formatierung der obigen Beispiele:

- Keine Leerzeichen innerhalb der eckigen Klammern (`<` und `>`).
- Keine Leerzeichen zwischen dem Namen der Kollektion und den eckigen Klammern.
- Keine Leerzeichen zwischen den eckigen Klammern und den runden Klammern beim Aufruf des Konstruktors.

[84] Dieses Formatierungsschema wird in den JSRs und verschiedenen Dokumenten von Sun Microsystems verwendet, die die neuen Eigenschaften und Fähigkeiten von Version 5 des Java Development Kits beschreiben. Es gibt aber zur Zeit keine formale Richtlinie.

2.4.5.2 Die erweiterte for-Schleife

[85] Bei den Beispielen zu den generischen Kollektionen im vorigen Unterunterabschnitt haben wir einen Iterator gewählt, um eine Kollektion elementweise zu verarbeiten:

```
for (Iterator<Dog> i = c.iterator(); i.hasNext(); ) {
    String name = i.next().getName();
```

[86] Der Kopf dieser `for`-Schleife ist aber unnötig ausführlich. In der Regel soll eine Kollektion Element für Element durchlaufen werden. Es ist nicht nötig, dem Compiler eine so sorgfältig ausformulierte Anweisung zu geben.

[87] Version 5 des Java Development Kits hat die elementweise Verarbeitung in einer Schleife erheblich vereinfacht. Die folgende Konstruktion liefert dasselbe Ergebnis:

```
for (Dog mutt : c) {
        String name = mutt.getName();
```

[88] Lesen Sie den Doppelpunkt als „in“. Die Anweisung `for (Dog mutt : c)` liest sich dann „Für jedes `mutt` in `c` ...“.

Bemerkung: Viele Entwickler wundern sich darüber, warum Sun Microsystems den Doppelpunkt

anstelle eines Wortes wie `in` oder `foreach` gewählt hat. Die Antwort ist einfach: Sun Microsystems wollte die Einführung eines neuen Schlüsselwortes vermeiden, das möglicherweise mit bereits vorhandenem Quelltext kollidiert. Ihr gesamter Quelltext sollte sich unter Version 5 des Java Development Kits sowohl übersetzen als auch ausführen lassen.

---

[89] Die erweiterte `for`-Schleife funktioniert auch bei gewöhnlichen Arrays. Das folgende Beispiel verarbeitet ein Array von `String`-Objekten ohne die erweiterte `for`-Schleife:

```
public static void main(String[] args) {
    for (int i = 0; i < args.length; i++ ) {
        System.out.println(args[i]);
    }
}
```

[90] Die erweiterte `for`-Schleife ermöglicht eine einfachere Schreibweise:

```
public static void main(String[] args) {
    for (String arg: args) {
        System.out.println(arg);
    }
}
```

[91/92] Beachten Sie, daß Sie die erweiterte `for`-Schleife nicht überall verwenden können. Hinter der erweiterten `for`-Schleife verbirgt sich ein Iterator, so daß Sie keine Methoden mehr aufrufen können, die die unterliegende Kollektion eventuell verändern. Bei der Formatierung der erweiterten `for`-Schleife steht zu beiden Seiten des Doppelpunktes ein Leerzeichen.

2.4.5.3 Autoboxing

[93] Version 5 des Java Development Kits gestattet die automatische Typumwandlung zwischen primitiven Typen und den entsprechenden Wrappertypen, zum Beispiel:

```
Integer myInteger = 5;   // automatically converts 5 (int) into an Integer
```

Hierfür wird keine Formatierungsrichtlinie benötigt.

2.4.5.4 Argumentlisten variabler Länge

[94] Argumentlisten variabler Länge, in der englischen Literatur häufig auch als „VarArgs“ bezeichnet, ermöglichen dem Entwickler, zu deklarieren, daß die Anzahl der Argumente beim Aufruf eines Konstruktors oder einer Methode flexibel ist. Die Verwendung dieser Fähigkeit kann die Anzahl überladener Methoden und Konstruktoren reduzieren.

[95] Auf der anderen Seite haben Argumentlisten variabler Länge auch gewisse Nachteile: Stehen Anzahl und Typ der Parameter einer Methode fest, so kann der Compiler Typprüfungen veranlassen, um zu gewährleisten, daß Ihr Methodenaufruf korrekt ist. Bei Argumentlisten variabler Länge ist dagegen nur eine minimale Prüfung möglich.

[96] Wir betrachten den Konstruktor einer Klasse `Dog` mit zwei Feldern `age` und `name` als Beispiel. Wenn Sie die erforderlichen Konstruktoren anlegen wollen, um jede mögliche Kombination von Argumenten zu erfassen, benötigen Sie die folgenden vier Exemplare:

```
Dog();
Dog(int age);
```

```
Dog(String name);
Dog(int age, String name);
```

[97] Das ist noch nicht alles: Hat Ihre Klasse n Felder, so gibt es nämlich 2^n mögliche Parameterkombinationen (Konstruktoren). Wenn wir die `Dog`-Klasse auf sieben Felder erweitern, zum Beispiel `age`, `height`, `weight`, `name`, `owner`, `color` und `pedigree`, so wären $2^7 = 128$ Konstruktoren erforderlich, um jede mögliche Kombination zu erfassen.

[98] Zusätzliche Probleme ergeben sich daraus, daß Java nicht zwischen Methoden unterscheiden kann, deren Signaturen effektiv identisch sind, zum Beispiel:

```
Dog(int age, String name);
Dog(int age, String owner);
```

[99] Version 5 des Java Development Kits ermöglicht nun, in der Deklaration festzulegen, daß eine Methode beziehungsweise ein Konstruktor eine variable Anzahl von Argumenten *des gleichen Typs* erwartet. Sie können somit deklarieren, daß die exakte Anzahl der `String`-Argumente beim Aufruf des Konstruktors der Klasse `Dog` variabel ist:

```
Dog(int age, String... args) {
    for (String parameter : args) {
        System.out.println("Received parameter " + parameter);
    }
}
```

[100] Die Auslassungspunkte (...) folgen dem Namen des Bezugstyps unmittelbar. Im Methodenkörper können Sie das entsprechende Argument wie ein Array behandeln.

Bemerkung: Die Deklaration einer Methode oder eines Konstruktors darf höchstens eine Argumentliste variabler Länge enthalten und dieser Parameter muß am Ende der Parameterliste des Konstruktors beziehungsweise der Methode deklariert werden. Beispielsweise ist *nicht erlaubt*: `Dog(int... ages, String name)`.

Warnung: Lassen Sie Sorgfalt walten, wenn Sie Konstruktoren oder Methoden mit Versionen überladen, die eine Argumentliste variabler Länge haben. Es ist leicht, Versionen einer überschriebenen Methode zu deklarieren, die der Compiler nicht unterscheiden kann (zum Beispiel sind `Dog(String name, String... args)` und `Dog(String... args)` effektiv identisch) oder eine Version zu deklarieren, die auf mehr Fälle angewendet werden kann, als Sie beabsichtigt haben (zum Beispiel paßt `Dog(Object... args)` per Autoboxing zu *jedem Konstruktoraufruf*.

---

[101] Eventuell ist Ihnen aufgefallen, daß es in den früheren Versionen des JDKs eine ähnliche Möglichkeit gab, nämlich die Übergabe eines Arrays des entsprechenden Typs. Dazu mußte allerdings erst ein neues Array erzeugt werden. Beispiel:

```
public static void lookupDog(String... searchCriteria) {
    for (String criterion : searchCriteria) {
        // do work here
    }
}

public static void lookupCat(String[] searchCriteria) {
    for (String criterion : searchCriteria) {
        // do work here - no different than working with Dog method
    }
}
```

```
public static void main(String[] args) {
    // first the easy code: use the Dog method:
    lookupDog("Breed", "Terrier", "Color", "Brown");
    // now for the Cat method
    lookupCat(new String[] {"Breed", "Burmese", "Coat", "Silky"});
    // or
    String[] criteria = {" Breed", "Burmese", "Coat", "Silky"};
    lookupCat(criteria);
}
```

[102] Der Quelltext der `lookupDog()`-Methode ist viel leichter zu lesen, zu schreiben und zu verstehen, als bei der äquivalenten `lookupCat()`-Methode.

2.4.5.5 Statisches Importieren

[103] Vor Version 5 des Java Development Kits wurden Konstanten mit Hilfe von Interfaces definiert. Indem eine Klasse ein solches Interface implementierte, konnten die Konstanten in dieser Klasse unqualifiziert, das heißt ohne Voranstellung des Interfacenamens, verwendet werden. Beispiel:

```
public interface BadInterface {
    public static final int FIRST_NAME_POSITION = 1;
}

public class BadClass implements BadInterface {
    public static void main(String[] args) {
        System.out.println("First name = " + args[FIRST_NAME_POSITION]);
    }
}
```

[104] Nachteile:

- Implementiert eine Klasse ein Interface, so sagt man, die Klasse sei eine Instanz dieses Interfaces. If you have a reasonable name for your interface, it probably doesn't make sense to say your class "is" an instance of it.
- Implementiert Ihre Klasse ein Interface, so implementiert auch jede von dieser abgeleitete Klasse dieses Interface. Die Konstanten werden dadurch Bestandteil des Namensraumes dieser abgeleiteten Klasse, obwohl die Konstanten möglicherweise gar nicht in dieser Klasse verwendet werden.

[105/106] Version 5 des Java Development Kits gestattet das *statische Importieren*, also das Importieren der statischen Komponenten einer Klasse oder eines Interfaces, um diese Probleme zu umgehen. Beim Schreiben eines Protokolleintrages (siehe Unterabschnitt 2.7.3) müßten Sie das Gewicht des Eintrages (*logging level*) beispielsweise eigentlich qualifiziert angeben:

```
myLogger.log(Level.FINE, "This message is at FINE level");
```

[107] Nach dem statischen Importieren können wir das statische Objekt `FINE` verwenden, als ob es in unserer Klasse definiert worden wäre:

```
import static java.util.logging.Level.*;
// ...
        myLogger.log(FINE, "This message is at FINE level");
```

[108] Wir merken an, daß die statische Importfähigkeit eingeführt wurde, um eine schlechte Programmierangewohnheit zu umgehen. Verwenden Sie diesen Mechanismus, wenn Sie andernfalls lokale Kopien der Konstanten deklarieren oder die Vererbung in der oben beschriebenen Weise mißbrauchen

müßten. Wir empfehlen generell das statische Importieren zu vermeiden und qualifizierte Konstanten zu verwenden.

2.5 Javadoc

[109/110] Javadoc ist ein sehr einfaches aber mächtiges Werkzeug, mit dessen Hilfe Entwickler API-Dokumentation für andere Entwickler bereitstellen können. In seiner Voreinstellung parst Javadoc Ihren Quelltext, achtet dabei auf spezielle Kommentare und generiert aus diesen eine Dokumentation im HTML-Format. Wir beginnen mit einigen Beispielen für Javadoc-Tags und beschreiben ihre Funktion:

```
/**
 * The <b>main</b> starting point for this application.
 * Instantiates an instance of this class, and runs it.
 *
 * @param args an array containing the command line arguments
 * @throws IOException if files cannot be created
 */
public static void main(String[] args) throws IOException {
  // ...
```

Ein Dokumentationskommentar beginnt mit dem Kommentar-Startsymbol `/**` und endet mit dem Schlußsymbol `*/`. Die Tags von Javadoc beginnen mit dem At-Zeichen (`@`). Der gesamte Text zwischen dem Startsymbol und dem ersten Tag wird direkt in die generierte Ausgabe übertragen. Javadoc ignoriert unbekannte Tags nach dem ersten erkannten Tags bis zum nächsten erkannten Schlüsselwort oder Dokumentende. *This bit of magic is how XDoclet can work.*

[111] Die obigen Javadoc-Tags führen zum folgenden HTML-Quelltext:

```
<A NAME=”main(java.lang.String[])”><!-- --></A><H3>
main</H3>
<PRE>
public static void <B>main</B>(java.lang.String[] args)
                 throws java.io.IOException</PRE>
<DL>
<DD>The <b>main</b> starting point for this application.
 Instantiates an instance of this class, and runs it.
<P>
<DD><DL>
<DT><B>Parameter:</B><DD><CODE>args</CODE> - an array containing the command
line arguments.
<DT><B>Throws:</B>
<DD><CODE>java.io.IOException</CODE> - if files cannot be created</DL>
</DD>
</DL>
```

[112] Im Firebox-Browser erscheint die HTML-Seite etwa wie in Abbildung 2.1.

Bemerkung: Javadoc ist viel mächtiger als viele Entwickler wahrnehmen. Javadoc wurde als erweiterbares Werkzeug konzipiert, dessen Verhalten vom angeschlossenen Modul abhängt. In der Voreinstellung parst Javadoc Java-Quelltext und generiert API-Dokumentation. Mit dem XDoclet-Modul (*http://xdoclet.sourceforge.net/xdoclet/index.html*) kann Javadoc, basierend auf Ihren Dokumentationskommentaren, Quelltext generieren (diese Fähigkeit ist beim Entwickeln von J2EE-Anwendungen sehr nützlich, da XDoclet Interfaces und Deployment-Deskriptoren für Sie erzeugen

kann). Mit dem DocCheck-Modul (*http://java.sun.com/j2se/javadoc/doccheck*) können Sie untersuchen, wie gut Sie sich an die Javadoc-Richtlinien von Sun Microsystems gehalten haben.

2.5.1 Richtlinien für Dokumentationskommentare

[113] Sun Microsystems hat einen Artikel mit der Überschrift „How to Write Doc Comments for the Javadoc Tool“ herausgegeben, den Sie unter der Internetadresse *http://java.sun.com/j2se/javadoc/writingdoccomments* finden. Wir fassen diesen Artikel in den folgenden Unterunterabschnitten zusammen.

2.5.1.1 Inhalt von Dokumentationskommentaren

[114] Die Namen Ihrer Klassen und Methoden sollten selbsterklärend sein. Es besteht also keine Veranlassung, einen Dokumentationskommentar anzulegen, der lediglich den Namen einer Klasse oder Methode wiederholt. Dokumentationskommentare enthalten Informationen für Entwickler, die mit den Objekten der dokumentierten Klasse arbeiten beziehungsweise die Klasse verändern, neu schreiben oder eine neue Klasse ableiten wollen, ohne dabei Ihren Quelltext zu sehen.

[115] Wenn Sie beispielsweise eine Klasse, die eine Netzwerkverbindung zu Ihrem Server aufbaut, `NetworkConnection` nennen, ist es nutzlos, im Dokumentationskommentar darauf hinzuweisen, daß diese Klasse eine Netzwerkverbindung herstellt (der Benutzer hat diese Information bereits anhand des Klassennamens). Statt dessen könnten Sie dokumentieren, daß die Klasse „über ein Netzwerk hinweg eine Verbindung zum Datenbankserver aufbaut, über die Fernzugriff auf die Funktionen der Datenbank möglich ist.“

Warnung: Implementierungsspezifische Informationen gehören nicht in die Dokumentationskommentare (API-Dokumentation).

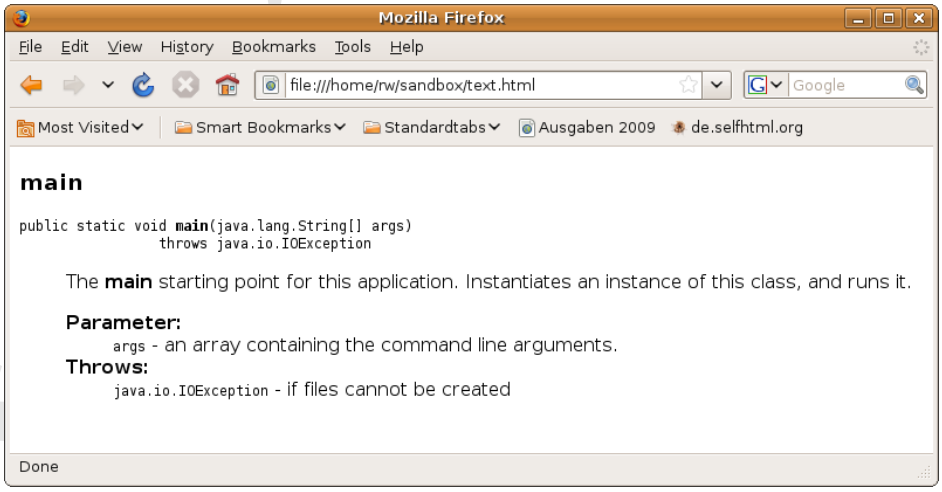

Abbildung 2.1: Beispiel für eine von Javadoc generierte HTML-Dokumentation.

Starttag	Endtag	Beschreibung
`<code>`	`</code>`	Der Text zwischen diese beiden Tags erscheint in nichtproportionaler Schrift in der HTML-Ausgabe. (Siehe auch Tipp auf Seite 39.)
`<pre>`	`</pre>`	Der Text zwischen diese beiden Tags erscheint in nichtproportionaler Schrift in der HTML-Ausgabe, wobei Ihre Einrückungen erhalten bleiben. Wird in der Regel für Quelltextbeispiele verwendet.
`<ul>`	`</ul>`	Diese Tags bezeichnen Anfang und Ende einer ungeordneten (nicht nummerierten) Liste.
`<ol>`	`</ol>`	Diese Tags bezeichnen Anfang und Ende einer geordneten (nummerierten) Liste.
`<li>`	`</li>`	Dieses Tags bezeichnen Anfang und Ende eines Aufzählungspunktes in einer geordneten oder ungeordneten Liste.
`<p>`	`</p>`	Diese Tags bezeichnen Anfang und Ende eines Absatzes.

Tabelle 2.4: Häufige verwendete HTML-Tags in Dokumentationskommentaren.

2.5.1.2 Platzierung von Dokumentationskommentaren

[116] Theoretisch können Sie überall in Ihrem Quelltext Dokumentationskommentare anlegen. Das `javadoc`-Programm berücksichtigt allerdings nur Kommentare, die sich an bestimmten Stellen befinden:

- Kommentare zu einer Klasse stehen unmittelbar vor der Klassendeklaration. (Javadoc gestattet gegenwärtig zwar Kommentare überall vor der Klassendeklaration, ja sogar vor der `package`-Anweisung, Sie sollten sich aber nicht darauf verlassen.)
- Kommentare zu Klassen- und Objektfeldern stehen unmittelbar vor der Felddeklaration, auf die sie sich beziehen.
- Kommentare zu Methoden stehen unmittelbar vor der Signatur der Methode.

[117] An allen anderen Stellen werden Dokumentationskommentare von `javadoc` komplett ignoriert. Ein Dokumentationskommentar im Körper einer Methode erscheint zum Beispiel nicht in der API-Dokumentation. Verwenden Sie in diesem Fall anstelle eines Dokumentationskommentars einen Implementierungskommentar.

2.5.1.3 Formatierungsmöglichkeiten und spezielle Javadoc-Tags

[118] Dokumentationskommentare dürfen beliebige HTML-Tags enthalten. Tabelle 2.4 zeigt eine Auswahl häufig verwendeter HTML-Tags.

Bemerkung: Javadoc erzeugt HTML-Ausgabe gemäß Version 3.2 der HTML-Spezifikation, die bei bestimmten Elementen keine schließenden Tags verlangt (zum Beispiel muß `<li>` nicht durch `</li>` und `<p>` nicht durch `</p>` beendet werden). Die Verwendung der schließenden Tags ist aber erlaubt. Die Version 3.2 der HTML-Spezifikation schreibt außerdem nicht vor, ob Tagnamen in Groß- oder Kleinbuchstaben geschrieben werden sollen. Wir verwenden im gesamten Buch HTML-Tags gemäß Version 3.2, formatieren sie aber nach der späteren Version 4.0 der HTML- beziehungsweise der XHTML-Spezifikation (das heißt wir setzen die schließenden Tags und schreiben Tagnamen in kleinen Buchstaben). Javadoc gestattet die Verwendung der Tags und Eigenschaften von Version 4.0 der HTML-Spezifikation, sie müssen allerdings damit rechnen, daß einige ältere Browser die

generierte HTML-Dokumentation nicht anzeigen können.

[119] Sie können noch viele andere HTML-Tags in Ihren Dokumentationskommentaren verwenden. In der Regel brauchen Sie die meisten davon jedoch nicht, da Sie API-Dokumentation für Entwickler schreiben.

Tipp: Verwenden Sie das `<code>`-Tag für Java-Schlüsselwörter sowie für die Namen von Packages, Klassen, Interfaces, Methoden, Feldern und Argumenten, um diese in nichtproportionaler Schrift darzustellen, falls der jeweilige Browser diesen Modus unterstützt (siehe auch Tabelle 2.4).

[120–123] Neben den HTML-Tags verfügt Javadoc über spezielle eigene Tags, die auf unterschiedliche Weise interpretiert werden. Tabelle 2.5 beschreibt die Javadoc-Tags für Klassen, Interfaces, Konstruktoren, Methoden und Felder. Die Tags sind in der Reihenfolge angeordnet, in der Sie in Ihrem Dokumentationskommentar vorkommen sollen. Die Tags `@see` und `@link` erzeugen Hyperlinks zur/zum ersten passenden Klasse, Methode oder Feld. Der Name der Klasse ist bei Bezügen auf Methoden und Felder innerhalb der Klasse optional. Analog ist der Packagename bei Verweisen innerhalb eines Packages nicht erforderlich. Bezieht sich ein Verweis nicht auf eine spezielle Version einer überladenen Methode, so muß die Parameterliste der Methode nicht angegeben werden. Tabelle 2.6 zeigt einige Beispiele.

Klassen und Interfaces	
Tag	**Beschreibung**
`@author`	Name des Autors dieser Klasse beziehungsweise diese Interfaces.
`@version`	Aktuelle Version dieser Klasse beziehungsweise dieses Interfaces. Ihr Versionskontrollsystem kann dieses Feld bewerten, wenn Sie den Quelltext auschecken.
`@see`	Erzeugt einen Verweis zur/zum entsprechenden/r Klasse, Interface, Konstruktor, Methode oder Feld. Der Verweis erscheint in einem separaten „See also“-Abschnitt der generierten Dokumentation. Siehe auch Tabelle 2.6 und `@link` (unten).
`@since`	Gibt die Versionsnummer des Packages an, in dem diese(s/r) Klasse, Interface, Konstruktor, Methode oder Feld zum ersten mal vorkommt.
`@deprecated`	Gibt an, daß diese(s/r) Klasse, Interface, Konstruktor, Methode oder Feld nicht mehr verwendet werden soll, aber aus Kompatibilitätsgründen noch existiert. Dem `@deprecated`-Tag folgt ein Hinweis, welche(s/r) Klasse, Interface, Konstruktor, Methode oder Feld der Benutzer stattdessen verwenden soll beziehungsweise den Hinweis „No replacement“, falls es keinen Ersatz für die Funktionalität gibt.
`@serial`	Used to specify wether a a class or field that would normally be serializable should be documented as being Serializable.
`{@link` *`reference label`*`}`	Erzeugt einen Hyperlink zur/zum angegebenen Klasse, Interface, Konstruktor, Methode oder Feld. Siehe auch Tabelle 2.6 und `@link` (unten). Der `label` wird in nichtproportionaler Schrift gesetzt.
`{@linkplain` *`reference label`*`}`	Wie `{@link}`, außer daß der `label` in der Standardschrift gesetzt wird.
`{@docRoot}`	Verweist auf das Wurzelverzeichnis Ihrer Javadoc-Dokumentation (dort liegt die Datei *index.html*). Wird stets korrekt aufgelöst, unabhängig davon, wie tief das Verzeichnis verschachtelt ist, in dem dieses Tag verwendet wird.
Konstruktoren und Methoden	

Tag	Beschreibung
`@param`	Beschreibt einen Parameter in der Signatur der Methode. Jeder Parameter wird in einer separaten Zeile beschrieben, wobei die Aufzählung in der Reihenfolge der Parameter in der Signatur angeordnet ist.
`@return`	Beschreibt den/die von der Methode zurückgegebenen Wert beziehungsweise Referenz.
`@exception`	Beschreibt eine Ausnahme, die diese Methode auswerfen kann. (Das äquivalente Tag `@throws` wurde in der Javadoc-Version 1.2 aufgenommen.)
`@see`	Siehe oben (Klassen und Interfaces) und Tabelle 2.6.
`@since`, `@deprecated`, `{@link` *`reference label`*`}`, `{@linkplain` *`reference label`*`}` und `{@docRoot}`	Siehe oben (Klassen und Interface).
Klassen- und Objektfelder	
Tag	**Beschreibung**
`@see`	Siehe oben (Klassen und Interfaces) und Tabelle 2.6.
`@since`, `@serial`, `@deprecated`, `{@link` *`reference label`*`}`, `{@linkplain` *`reference label`*`}` und `{@docRoot}`	Siehe oben (Klassen und Interfaces).
`{@value}`	Gibt den Wert der dokumentierten Konstante an.

Tabelle 2.5: Javadoc-Tags für Klassen, Interfaces, Konstruktoren, Methoden und Felder.

[124] Version 5 des Java Development Kits führt die beiden neuen Tags `{@code text}` und `{@literal text}` ein, die überall in der Dokumentation verwendet werden dürfen. In beiden Fällen wird `text` ohne Interpretation angezeigt. Enthält der Dokumentationstext beispielsweise ein `<b>`-Tag, so würde es in der Regel als Startsymbol für Fettdruck interpretiert werden. Bei `{@code <b>}` dagegen enthält die Ausgabe den Text „`<b>`“, genauer, setzt `{@code <b>}` sein Argument in nichtproportionaler Schrift, während `{@literal <b>}` sein Argument in Normalschrift wiedergibt.

Tag	Beschreibung
`@see #field label`	Erzeugt einen Verweis zum angegebenen Feld mit dem angegebenen ~~label~~.
`@see #method label`	Erzeugt einen Verweis zur ersten passenden Methode und gibt dem Verweis den angegeben ~~label~~.
`@see #method (parameters) label`	Erzeugt einen Verweis zur Methode mit passender Signatur und gibt dem Verweis den angegeben ~~label~~.
`@see class#method label`	Erzeugt einen Verweis zur ersten passenden Methode in der benannten Klasse und gibt dem Verweis den angegeben ~~label~~.
`@see package.class#method label`	Erzeugt einen Verweis zur ersten passenden Methode in der benannten Klasse im benannten Package und gibt dem Verweis den angegeben ~~label~~.

Tabelle 2.6: Beispiele für per `@see`-Tags definierte Verweise.

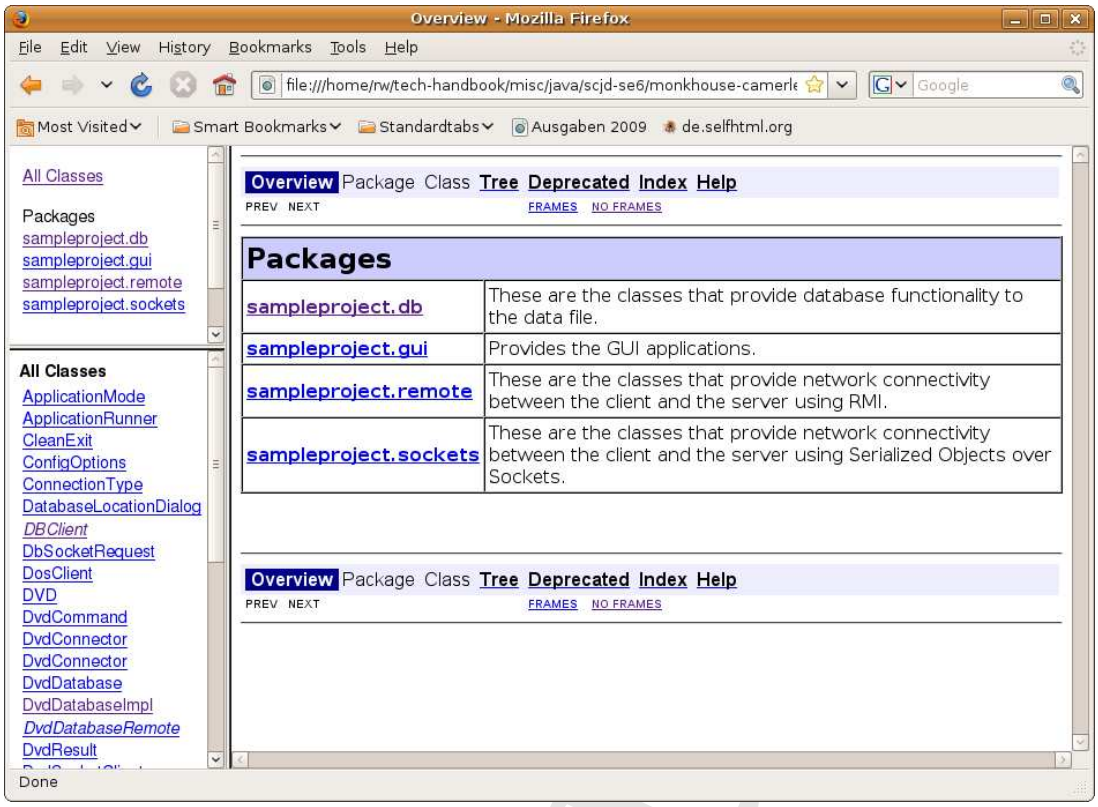

Abbildung 2.2: Die Überblicksseite der Javadoc-Dokumentation der Beispielanwendung *Denny's DVDs* (Beispiel).

2.5.1.4 Packagedokumentation

[125] Javadoc kann auch Dokumentation über Packages in die generierte Ausgabe einsetzen und eine Überblicksseite über Ihr gesamtes Projekt erzeugen. Abbildung 2.2 zeigt die Überblicksseite für unsere Beispielanwendung.

[126] Das Erzeugen von Package-Dokumentation setzt voraus, daß jedes Packageverzeichnis eine Datei namens *package.html* enthält. Ein einfaches Beispiel:

```
<!doctype HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<html>
<head>
<title>sampleproject.db</title>
</head>
<body>
Provides database functionality.
<h2>Package Specification</h2>
These classes are the base that provide the basic data creation, reading, updating,
and deleting functions required to process to process data in a database. Basic
locking functionality exists, but this is not an advanced package.

@author Unattributed person from example.com
@version 1.0
@since 1.0
</body>
</html>
```

2.5.1.5 Änderung in Version 5 des Java Development Kits

[127] Die offensichtlichen Änderungen an Javadoc im Vergleich mit den früheren Versionen belaufen sich auf die beiden neuen Tags `{@code text}` und `{@literal text}` sowie die Unterstützung von generischen Kollektionen, Aufzählungstypen und Argumentlisten variabler Länge.

[128] Sun Microsystems hatte anläßlich der Herausgabe von Version 1.4 des Java Development Kits den Mechanismus geändert, mit dessen Hilfe `javadoc` ermittelt, welcher Anteil eines Kommentars in der Zusammenfassung erscheint (die Trennung verlief zwischen der Zusammenfassung und dem Rest des Kommentars). Nach dem Sun Microsystems das neue Trennverfahren in `javadoc` eingesetzt hatte, wurden unzählige Warnungen hervorgerufen, die dokumentierten, daß die beabsichtige Trennung zu unterschiedlichen Zusammenfassungen führen würde. Sun Microsystems hat diese Pläne revidiert und das Trennverfahren wieder entfernt.

2.5.1.6 Aufruf des javadoc-Programms auf der Kommandozeile

[129] Der `javadoc`-Aufruf über die Kommandozeile hat das folgende Format:

```
javadoc [options] [packages] [sourcefiles] [@files]
```

[130] Das folgende Kommando generiert die API-Dokumentation aller *.java* Dateien, die keinem Package angehören (aufgerufen in dem Verzeichnis, das die *.java* Dateien enthält):

```
javadoc *.java
```

[131] Wenn die *.java* Dateien in Packages liegen, genügt es, die Namen der Packages anzugeben:

```
javadoc com.example.mypackage com.example.more.packages
```

[132] Die beiden obigen Beispiele erzeugen die API-Dokumentation im aktuellen Verzeichnis. Sie können mit Hilfe der Option `-d` ein Wurzelverzeichnis angeben, in das die API-Dokumentation installiert werden soll:

```
javadoc -d doc/api com.example.mypackage com.example.more.packages
```

Tipp: `javadoc` konvertiert Pfadangaben in Unixnotation in plattform-spezifische Syntax. Das ist praktisch, wenn Sie `javadoc` in einem Skript aufrufen, das von unbekannten Benutzern auf beliebigen Plattformen ausgeführt werden kann.

---

[133] Sie können mit Hilfe der Kommandozeilenoptionen von `javadoc` weitere Eigenschaften und Fähigkeiten zu- oder abschalten, zum Beispiel:

```
1 javadoc           \
 2    -d doc/api    \
 3    -version      \
 4    -author       \
 5    -use          \
 6    -source 1.5   \
 7    -windowtitle "Denny's DVDs application"                          \
 8    -doctitle    "<h1>The SCJD Exam with J2SE 1.5 Project</h1>"      \
 9    -bottom      "<i>Developers: Andrew Monkhouse and Terry Camerlengo</i>" \
10    -sourcepath  src                                                 \
11    -linkoffline http://java.sun.com/j2se/1.5.0/docs/api/jdk_1.5/api \
12    sampleproject.gui sampleproject.db sampleproject.remote sampleproject.sockets
```

Zeile	Schalter	Wirkung
2	`-d`	Gibt das Wurzelverzeichnis der generierten API-Dokumentation an.
3	`-version`	Gibt an, daß die Versionsnummer (`@version`-Tag) aus Ihrem Javadoc-Kommentar in die generierte API-Dokumentation eingetragen wird.
4	`-author`	Gibt an, daß der Name des Autors (`@author`-Tag) aus Ihrem Javadoc-Kommentar in die generierte API-Dokumentation eingetragen wird.
5	`-use`	Weist `javadoc` an, zu jeder Klasse eine „Uses of"-Seite anzulegen, die angibt, wo diese Klasse als Parameter oder Rückgabewert einer Methode vorkommt.
6	`-source`	Specifies which version of the JDK the Javadoc should maintain compatibility with.
7	`-windowtitle`	Gibt den Inhalt der Titelzeile des Browserfensters an (falls unterstützt).
8	`-doctitle`	Gibt die Überschrift der „Overview"-Seite (Packageübersicht) an.
9	`-bottom`	Gibt den Text an, der am Fuß jeder generierten Seite angezeigt wird.
10	`-sourcepath`	Gibt das Wurzelverzeichnis an, in dem Ihre Packages liegen.
11	`-linkoffline`	Creates a link to existing Javadoc entries. See the description in the paragraph that follows.

Tabelle 2.7: Bedeutung der häufigsten Kommandozeilenoptionen von `javadoc`.

[134–137] Tabelle 2.7 erläutert die Wirkung dieser Optionen. Die Optionen `-link` und `-linkoffline` gestatten Verweise auf bereits existierende API-Dokumentation. Wenn eine Ihrer Methoden einen `String`-Parameter erwartet, können Sie per `-link`/`-linkoffline` in Ihrer API-Dokumentation zu dieser Methode einen Verweis auf die originale API-Dokumentation der Klasse `String` von Sun Microsystems anlegen. Sie verwenden `-link`, wenn Sie keine lokale Kopie der existierenden API-Dokumentation besitzen, sondern über ein Netzwerk darauf zugreifen. Sie verwenden dagegen `-linkoffline`, wenn Sie eine lokale Kopie der vorhandenen API-Dokumentation oder eine lokale Kopie der *package-list*-Datei zur Verfügung haben oder aber wenn Sie die bereits existierende API-Dokumentation nicht über ein Netzwerk erreichen können.

Tipp: Verwenden Sie lokale Dokumentation statt über das Internet nachzulesen, wann immer es möglich ist, da Sie schlicht und einfach schneller sind. Verweisen Sie aus demselben Grund bei Ihrer Dokumentation auf Ihre lokale Kopie statt auf eine Version im Netz.

[138] Sie wollen nicht jedesmal, wenn Sie Ihre API-Dokumentation neu generieren müssen, ein so aufwendiges Kommando eingeben. Eine Vereinfachung besteht darin, alle Optionen (eine pro Zeile) in einer einfachen Textdatei zu speichern und sich beim Kommandoaufruf auf diese sogenannte Optionsdatei zu beziehen (siehe auch Kapitel 9, Seite 288). Wenn Sie alle erforderlichen `javadoc`-Optionen in die Datei *javadoc.options* eingetragen haben, können Sie Ihre Dokumentation mit dem folgenden Kommando neu generieren:

```
javadoc @javadoc.options sampleproject.gui sampleproject.db sampleproject.remote
```

2.6 Packages

[139] Wenn Sie alle Dateien auf Ihrer Festplatte in einem einzigen Verzeichnis deponieren würden, wäre dieses Verzeichnis schnell unüberschaubar und die Suche nach einer bestimmten Datei ein Alptraum. Sie verwenden daher Verzeichnisse, um zusammengehörigen Dateien zu verwalten, zum Beispiel ein Verzeichnis für Buchführung, eines für die Stellensuche und eines für Musik. Manche Verzeichnisse haben Unterverzeichnisse mit weiteren Subkategorien.

[140] Dieselbe Situation besteht auch bei der Softwareentwicklung. Bei Java-Anwendungen können Dateien in Packages verwaltet werden, einem plattformunabhängigen Äquivalent von Verzeichnissen. Eine Klasse, die zu einem bestimmten Package gehört, wird lokalisiert, indem Sie den Klassennamen „vollqualifiziert“, das heißt mit vorangestelltem vollständigem Packagenamen angeben. Das funktioniert sogar bei Betriebssystemen, die das Konzept eines Verzeichnisses, Ordners oder hierarchischen Dateisystems nicht kennen.

[141] Betrachten Sie zu Beginn eines Projektes, welche logischen Module oder Funktionalitäten das Projekt haben könnte und ordnen Sie jede Klasse nach Funktionalität in ein entsprechendes Package ein. Die Beispielanwendung hat Klassen, die eine graphische Benutzeroberfläche aufbauen, andere Klassen, die eine Netzwerkschnittstelle implementieren und wiederum andere Klassen, die den Zugriff auf die Datenbankdatei ermöglichen. Wir beginnen mit drei hypothetischen Packages:

```
gui
network
database
```

[142] Nach dem Durchlaufen der Explorations- und Designphase des Projektes sind eventuell weitere Packages erforderlich und/oder vorhandene Packages zu ändern. Kapitel 3 beschreibt die Beispielanwendung *Denny's DVDs*. Das `network`-Package könnte zum Beispiel in zwei separate Unterpackages unterteilt werden (eines für RMI und eines für Sockets):

```
network.rmi
network.sockets
```

[143] Es wird empfohlen, Packagenamen zu „qualifizieren“, um zu gewährleisten, daß jede Klasse eindeutig identifiziert werden kann. Wenn Sie ein Objekt der Klasse `StartGui` erzeugen wollen, dann interessieren Sie sich für *Ihre* Klasse und nicht für eine andere Klasse mit identischem Namen. Eindeutigkeit wird dadurch gewährleistet, daß der Packagename mit Ihrem „rückwärts“ und in Kleinbuchstaben geschriebenen Domainnamen beginnt. Wenn Ihr Unternehmen beispielsweise den Domainnamen `example.com` hat, beginnt jeder Packagename mit `com.example`. Der vollqualifizierte Packagename lautet nun `com.example.network.sockets`.

[144] Bei Windows- und Unix-artigen Plattformen entspricht jedem Package ein eigenes Verzeichnis. Zum obigen Package `com.example.network.sockets` gehört demnach ein Verzeichnis *com* für das erste Package, welches ein Unterverzeichnis *example* für das zweite Package enthält, das wiederum ein Unterverzeichnis *network* für das dritte Package beinhaltet und dieses schließlich ein viertes Unterverzeichnis namens *sockets* enthält. Das letzte Unterverzeichnis (*sockets*) enthält alle Java-Klassen, die dem Package `com.example.network.sockets` angehören.

Bemerkung: Viele Betriebsysteme unterscheiden bei Datei- und Verzeichnisnamen zwischen Groß- und Kleinschreibung. Die Datei *numberOfBoxes* unterscheidet sich hierbei von *NumberOfBoxes*. Da sich Java auf Dateien aus dem unterliegenden Dateisystem bezieht und nicht jede mögliche Permutation von Groß- und Kleinbuchstaben ausprobieren kann, hat Sun Microsystems festgelegt, daß die Groß- und Kleinschreibung von Datei- und Verzeichnisnamen mit der Notation der Klassen- und Packagenamen übereinstimmen muß.

[145] Die Java-Werkzeuge von Sun Microsystems arbeiten in konsistenter Weise mit Packages. Findet also der Java-Compiler Ihre Klasse, so finden auch die Laufzeitumgebung und `javadoc` diese Klasse.

[146] Eine *.jar* Datei kann sich wie ein Package verhalten. Wenn Sie im Wurzelverzeichnis Ihres Projektes (hier *~/devProj/classes/*) das folgende Kommando aufrufen

```
~$ jar cf db.jar sampleproject.db
```

erhalten Sie eine komprimierte Datei namens *db.jar*, die den Verzeichnisteilbaum des Packages `sampleproject.db` enthält. Die Datei *db.jar* kann dem Klassenpfad hinzugefügt werden, wie ein Verzeichnis und der Compiler beziehungsweise die Laufzeitumgebung durchsuchen *db.jar* nach referenzierten Klassen und Interfaces.

[147–149] Achten Sie auf das Verhalten der Java-Werkzeuge bei *verteilt definierten Packages*. Stellen Sie sich beispielsweise vor, daß zwei Verzeichnisse im Dateisystem einer Festplatte *verschiedene Teile eines Packages* enthalten, etwa die Klassen des ersten Packages Verzeichnis *~/devProj/classes* und die Klassen des zweiten Packages außerhalb der Verzeichnisstruktur des Projektes im Verzeichnis */tmp/tempClasses*. Das erste Verzeichnis enthält das Package `sampleproject.db`, das zweite Verzeichnis das Package `sampleproject.testclient`. (Die Klassen befinden sich also effektiv in den Verzeichnissen *~/devProj/classes/sampleproject/db* beziehungsweise */tmp/tempClasses/sampleproject/testclient*.) Nun werden beide Verzeichnisse per `jar` zu zwei separaten *.jar* Dateien komprimiert:

```
~$ jar cf db.jar sampleproject.db
~$ jar cf testclient.jar sampleproject.testclient
```

[150] Schließlich werden beide *.jar* Dateien beim Aufruf des Compilers in den Klassenpfad integriert:

```
~$ javac -cp ~/devProj/classes/db.jar /tmp/tempClasses/testclient.jar
```

[151] Der Inhalt der ersten *.jar* Datei überschreibt den Inhalt der zweiten *.jar* Datei *nicht*. Enthalten separate *.jar* Dateien oder separate Verzeichnisse Teile ein und derselben Package-Verzeichnisstruktur (also ein und desselben Packages), so kombinieren der Java-Compiler beziehungsweise die Laufzeitumgebung die Gesamtstruktur.

Warnung: Enthalten zwei *.jar* Dateien eine Klasse mit identischem vollqualifizierten Namen, so „gilt“ die erste Klasse (bezüglich der Anordnung der *.jar* Dateien im Klassenpfad) und die zweite wird ignoriert.

[152] Die Möglichkeit, den Inhalt von Packages auf verschiedene Verzeichnisse zu verteilen, gestattet eine sehr modulare Struktur. Beachten Sie daher gewisse Idealvorstellungen, wenn Sie die Packagestruktur einer Anwendung planen. Ein Package umfaßt alle Klassen, die ähnliche Funktionalität haben oder aufgrund des Designs voneinander abhängig sind. Dadurch wird ein Package zu einer vollständigen funktionalen Einheit. Eine *.jar* Datei kann somit ein ganzes funktionales Package enthalten, das an ein Projekt „angekoppelt“ wird.

Bemerkung: Die in Kapitel 3 eingeführte Beispielanwendung verwendet eine Packagestruktur, die zwei verschiedene Implementierungen der Netzwerkschnittstelle (RMI und Sockets) voneinander trennt. Diese Packagestruktur gestattet, die eine Netzwerkschnittstelle ohne Auswirkungen auf die übrigen Packages der Anwendung durch die andere zu ersetzen.

[153] Die Klassen im Wurzelverzeichnis einer Packagestruktur sind die Bausteine der Funktionalität des Packages und daher die stabilsten (das heißt den wenigsten Änderungen ausgesetzten) Bestandteile des Packages. Packages in der Nähe der Wurzelebene enthalten die stabilen Klassen auf denen alle anderen Klassen aufbauen. Packages sollten nach diesem Prinzip aufgebaut sein.

[154] Falls es nicht möglich ist, nur stabile, das heißt sich nicht ändernde Klassen, im Wurzelverzeichnis zu deponieren, so sollten diese Klassen durch Interfaces ersetzt und deren unbeständige Implementierungen in eine höhere Ebene der Packagestruktur oder ein eigenes Package verlegt werden.

[155] Achten Sie darauf, daß Abhängigkeiten zwischen Klassen innerhalb eines Packages aufgelöst werden. Ein Package ist eine autonome Einheit, die separat übersetzt werden kann. Müssen die Klassen in einem Package übersetzt werden, um ein anderes Package übersetzen zu können, so sollte in der Regel die Packagestruktur geändert werden, damit keine Abhängigkeiten zwischen Klassen über die Grenzen eines Packages hinaus bestehen. Eine gute Packagestruktur gewährleistet, daß das Fundament einer Anwendung so stabil wie möglich ist.

[156] Zusammenfassung: Packages sind ein Mittel, um eine Anwendung in funktionale Blöcke aufzuteilen. Eine gute Packagestruktur gewährleistet, daß Teile der Anwendung entfernt oder anders angeordnet werden können, ohne sich auf den Quelltext der übrigen Teile auszuwirken.

2.7 Bewährte Arbeitsmethoden

[157] Sie haben im Laufe Ihrer Entwicklerkarriere eventuell Arbeitsmethoden beobachtet, die sich nicht in die obigen Kategorien einordnen lassen. Obwohl diese Methoden nicht verlangt werden, sollten Sie sie für Ihre Prüfungsaufgabe in Betracht ziehen.

2.7.1 Dokumentieren Sie während der Entwicklung

[158] Arbeiten Sie an Ihrer Dokumentation während Sie die Lösung zu Ihrer Prüfungsaufgabe entwickeln. Am besten schreiben Sie die Dokumentation, bevor Sie mit der Entwicklung beginnen. Zu Ihrer Prüfungsaufgabe werden drei Arten von Dokumentation verlangt:

- Design-Entscheidungen
- Javadoc
- Betriebsanleitung für die Benutzer

2.7.1.1 Design-Entscheidungen

[159] Die Dokumentation Ihrer Design-Entscheidungen enthält eine Kurzübersicht über die *wesentlichen Entscheidungen* während der Arbeit an Ihrer Prüfungsaufgabe, die betrachteten Alternativen und eventuell Erläuterungen, warum Sie die Alternativen verworfen haben.

[160] Sie brauchen kein Buch über Ihre Entscheidungen schreiben. Der Gutachter interessiert sich vornehmlich für zwei Dinge:

1. Haben Sie Alternativen betrachtet?
2. Stammt die eingereichte Lösung tatsächlich von Ihnen?

[161] Wenn Sie sich zum Systemarchitekten weiterentwickeln möchten, wird von Ihnen erwartet, daß Sie mehrere Wege *beherrschen*, um ein Ziel in der Softwareentwicklung zu erreichen, häufig unter Berücksichtigung verschiedener Architekturen und Programmiersprachen. Als Entwickler müssen Sie ebenfalls in der Lage sein, Lösungsalternativen im Rahmen Ihrer Erfahrung (J2SE) zu betrachten und gegebenenfalls zu verwerfen. Beispielsweise müssen Sie in einer Situation, in der sowohl ein Radiobutton als auch ein Ankreuzfeld in Frage kommt, erkennen daß beide Alternativen möglich sind und sich für die richtige Wahl entscheiden. (Beachten Sie aber, daß die Entscheidung zwischen einem Radiobutton und einem Ankreuzfeld wahrscheinlich eine untergeordnete Rolle spielt und kein Punkt ist, den Sie zu dokumentieren brauchen.)

Tipp: Dokumentieren Sie Ihre Design-Entscheidungen möglichst auch in Form von Implementierungskommentaren in der Nähe der entsprechenden Stelle im Quelltext. Wenn Sie sich beispielsweise für einen Radiobutton anstelle eines Ankreuzfeldes entschieden haben, erspart ein entsprechender Implementierungskommentar beim Konstruktor des Radiobuttons dem Wartungsprogrammierer Ihres Quelltextes die Zeit, um festzustellen, ob dies die richtige Stelle im Quelltext ist oder nicht.

---

[162] Sun Microsystems steht vor einer schwierigen Aufgabe: Wie läßt sich nachprüfen, ob *Sie* die eingereichte Lösung entwickelt haben? Ein Teil des „Nachweisprotokolls“ besteht darin, daß es eine Reihe von Prüfungsaufgaben gibt (unterschiedliche Geschäftsbereiche), von denen jede wiederum in verschiedenen Versionen ausgegeben wird (unterschiedliche Interfaces, die implementiert werden müssen). Es ist unwahrscheinlich, daß jemand den Sie kennen dieselbe Prüfungsaufgabe bearbeitet wie Sie. Ein anderer Teil besteht darin, zu prüfen, daß die Informationen, die Sie in der schriftlichen Prüfung (bei der Sie sich ausweisen müssen) angeben, mit der Dokumentation Ihrer Design-Entscheidungen übereinstimmen und daß dieses Dokument tatsächlich Ihren Quelltext beschreibt. Dies kombiniert, kann Sun Microsystems einigermaßen sicher sein, daß die Person, die die schriftliche Prüfung abgelegt hat, die Person ist, die den Quelltext geschrieben hat.

[163] Es ist sehr wichtig, daß Sie Ihre Design-Entscheidungen aufschreiben, während Sie an Ihrer Prüfungsaufgabe arbeiten. Wenn Sie diese Arbeit bis zum Projektende vor sich herschieben, wird es Ihnen sehr schwer fallen, sich an Ihre Entscheidungen und die Begründung zu erinnern.

2.7.1.2 Javadoc

[164–167] Es hat sich wiederholt gezeigt, daß Dokumentation nicht zeitgleich mit dem Quelltext aktualisiert wird und sich als nutzlos erweist, wenn beide voneinandert getrennt sind. Javadoc und ähnliche Dokumentationswerkzeuge sind entwickelt worden, um dieses Problem zu lösen. Befindet sich die Dokumentation im Quelltext selbst, so ist es viel leichter beide zugleich zu aktualisieren. Dies setzt allerdings voraus, daß Sie die Dokumentationskommentare zeitgleich mit dem Quelltext schreiben und aktualisieren. Wenn Sie bis zum Ende einer dreimonatigen Projektlaufzeit warten, um den Quelltext zu kommentieren, haben Sie eventuell weitere drei Monate vor sich, schlicht und einfach weil Sie den gesamten Quelltext nochmals durchsehen müssen, um zu verstehen was Sie gemacht haben und es zu dokumentieren.

2.7.1.3 Betriebsanleitung für die Benutzer

[168] Im Unterunterabschnitt 2.1.2.1 haben wir empfohlen die graphische Benutzeroberfläche Ihrer Prüfungsaufgabe grob zu skizzieren, bevor Sie mit der Entwicklung beginnen. Ein Vorteil besteht darin, daß Sie ein definiertes Ziel haben, auf das Sie zuarbeiten können und das Risiko verringern,

mit einer weniger anwenderfreundlichen graphischen Benutzeroberfläche dazustehen, nur weil sie sich leichter entwickeln ließ.

[169/170] Wenn Sie die Betriebsanleitung für die Benutzer schreiben, bevor Sie mit dem Entwickeln der Prüfungsaufgabe anfangen, erhalten Sie denselben positiven Effekt. Sie haben im Voraus aufgeschrieben, was die Anwendung leisten soll und haben ein klares Ziel auf das Sie sich zubewegen. Insbesondere haben Sie ein definiertes Ende: Entspricht die Anwendung Ihrer Betriebsanleitung, dann wissen Sie, daß sie reif für die Testbenutzer ist.

[171] Die Betriebsanleitung ist die Anlaufstelle, die der Benutzer aufsucht, wenn er ein Problem hat oder wissen möchte, wie eine fortgeschrittene Funktionalität benutzt wird. Eine fehlerhafte oder schlecht geschriebene Betriebsanleitung frustriert den Benutzer und sorgt bei Ihrer Arbeitsstelle für Supportanrufe, die viel ärgerlicher sind als sie sein müßten. Wenn Sie Ihre Dokumentation im Voraus schreiben fühlen Sie sich weniger gedrängt und nehmen sich die Dokumentation während der Entwicklungsarbeit immer wieder einmal vor, um daran zu arbeiten.

2.7.2 Zusicherungen

[172] Zusicherungen (*assertions*) sind seit Version 1.4 des Java Development Kits verfügbar und stellen für den Entwickler eine nützliche Bestätigung und Dokumentation von Annahmen dar, ohne sich nach dem Deployment der Anwendung auszuwirken. Die beiden Syntaxen von Zusicherungen lauten:

```
assert <expression with a boolean result>;
```

beziehungsweise

```
assert <expression with a boolean result> : <any statement>;
```

[173] Liefert `<expression with a boolean result>` bei aktivierter Prüfung von Zusicherungen `false`, so wird ein Fehlerobjekt vom Typ `java.lang.AssertionError` ausgeworfen.

[174] Entwickler machen zuweilen Annahmen über das Zusammenwirken zweier Teile ihres Programms, zum Beispiel könnte ein Entwickler annehmen, daß der Wert eines Parameters in einer privaten Methode niemals über einen bestimmten Vergleichswert hinaus wächst. Falls eine solche Annahme ungerechtfertigt ist, können die dadurch verursachten Probleme für lange Zeit unentdeckt bleiben. Hier kann es sich lohnen, zu Beginn der Methode eine `assert`-Anweisung einzusetzen, die beim Testen verwendet wird, um die getroffene Annahme zu bestätigen, aber in der deployten Anwendung keine Wirkung hat.

Warnung: Verwenden Sie Zusicherungen *niemals*, um die Argumente einer öffentlichen Methode zu prüfen. Öffentliche Methoden werden von Entwicklern aufgerufen, die sich eventuell nicht um Ihre Randbedingungen kümmern. Sie sollten die Argumente einer öffentlichen Methode aber *immer* validieren, unabhängig davon, ob Zusicherungen aktiviert oder deaktiviert sind.

---

[175] Ein Beispiel für den korrekten Gebrauch einer Zusicherung:

```
public class AssertionTest {
    public static void main(String[] args) {
        new AssertionTest(11);
    }
    public AssertionTest(int withdrawalAmount) {
        int balance = reduceBalance(withdrawalAmount);

        // reduceBalance should never return a number less than zero
```

```
        assert (balance < 0) : "Business rule: balance cannot be < 0";
    }
}
```

Bemerkung: Bei Version 1.4 des Java Development Kits mußte der Compiler explizit mit der Option `-source 1.4` aufgerufen werden, um Zusicherungen zu übersetzen. Ab Version 5 ist diese Option nicht mehr erforderlich.

[176] Zusicherungen werden per Voreinstellung nicht ausgewertet. Dafür gibt es zwei Hauptgründe:

- Der Typ `AssertionError` ist von `Error` abgeleitet. Das Auswerfen eines Fehlerobjektes führt zum Programmabbruch und ist daher im produktiven Betrieb unerwünscht.
- Das Auswerten des Ausdrucks einer Zusicherung kann zeitaufwendig sein. Die voreingestellte Deaktivierung verbessert die Performanz.

Warnung: Verwenden Sie Zusicherungen *niemals*, um Aktionen auszuführen, die für die Funktionsweise einer Methode notwendig sind. Zusicherungen sollten lediglich dazu verwendet werden, um zu prüfen, ob bestimmte Werte korrekt sind. Da Zusicherungen im normalen Betrieb deaktiviert sind, würde eine zusicherungsabhängige Aktion in der Regel nicht ausgeführt werden.

[177] Sie aktivieren Zusicherungen zur Laufzeit, indem Sie die Laufzeitumgebung mit den Schaltern `-ea` oder `-enableassertions` aufrufen:

```
java -ea AssertionTest
```

[178] Ohne den Schalter `-ea` läuft das obige `AssertionTest`-Programm fehlerlos. Mit dem Schalter `-ea` wird ein Fehlerobjekt vom Typ `AssertionError` ausgeworfen.

[179] Es ist möglich, Zusicherungen nur für eine bestimmte Klasse oder ein bestimmtes Package zu aktivieren, indem der Klassen- oder Packagename auf der Kommandozeile angegeben wird:

```
java -ea:<packageName> -ea:<className>
```

2.7.3 Protokollmeldungen

[180/181] Bei der Fehlersuche ist es häufig nützlich, zu wissen ob eine bestimmte Methode erreicht wurde und welche Werte einige Parameter und Felder beziehungsweise lokale Variablen haben. Ein Debugger gestattet das Setzen von Haltepunkten (*break points*) an bestimmten Stellen sowie das Überwachen von Variablen. Die Arbeit mit dem Debugger zur Fehlersuche in einem Programm kann allerdings mühsam sein.

[182] Der nächste logische Schritt sind über den Quelltext verstreute `System.out.println()`-Anweisungen. Durch Beobachten der Ausgaben im Konsolenfenster haben Sie einen Eindruck davon, was Ihr Programm tut. Allerdings ist dies keine gute Lösung für ein Programm, mit dem jemand anders (zum Beispiel Ihr Kunde oder Gutachter) arbeitet. Im günstigsten Fall gerät der Betrachter nur aus dem Konzept. Im schlimmsten Fall hält er die „normalen“ Kontrollmeldungen für Anzeichen von Fehlern. Insbesondere haben Sie bei Anwendungen die auf Serverrechnern laufen eventuell keinen Zugriff auf ein Fenster in dem diese Meldungen angezeigt werden. Falls mit Ihrem Programm etwas schief geht, ist es schwierig, den Benutzer dazu zu bewegen, die entsprechenden Meldung herauszukopieren und Ihnen zu senden. Außerdem ist ein Teil der Meldungen nur bei der Fehlersuche

sinnvoll, nicht aber nach dem die Anwendung deployt wurde, das heißt Sie möchten eventuell einen Teil der Meldungen abschalten.

[183] Diese Probleme treten häufig auf und es gibt eine allgemeine Lösung: Verwenden Sie eine Protokollbibliothek. Dadurch sind Sie in der Lage:

- Protokollmeldungen an einem bestimmten Empfänger (zum Beispiel Datei, Bildschirm oder Drucker) zu senden.
- Protokollmeldungen auch dann aufzuzeichnen, wenn eine Anwendung im „Servermodus“ läuft und kein Konsolenfenster zur Anzeige der Meldungen vorhanden ist.
- zu gewährleisten, daß Protokollmeldungen über die gesamte Anwendung hinweg in konsistenter Weise ausgegeben werden.
- den Protokollmechanismus für die gesamte Anwendung selektiv an- und abzuschalten.

[184] Seit Version 1.4 des Java Development Kits gibt es eine standardisierte Protokollbibliothek. Rufen Sie in Ihrer Prüfungsaufgabe stets dann eine Methode aus dieser Bibliothek auf, wenn Sie dabei sind eine Meldung an den Standardausgabe- oder den Standardfehlerkanal auszugeben.

[185] Bevor Sie eine Protokollmeldung aufzeichnen können, benötigen Sie eine Referenz auf ein `java.util.logging.Logger`-Objekt. `Logger`-Objekte haben in der Regel einen Namen (es gibt aber auch ein anonymes `Logger`-Objekt gibt), so daß Sie mehrere `Logger`-Objekte unterschiedlich konfigurieren können. Die folgende Anweisung fordert eine Referenz auf das `Logger`-Objekt im Kontext `example.com.testApplication` an:

```
Logger myLogger = Logger.getLogger("example.com.testApplication");
```

[186] Jede Klasse, die ein `Logger`-Objekt anfordert und dabei den Namen `example.com.testApplication` angibt, erhält eine Referenz auf das obige von `myLogger` referenzierte Objekt. This ensures that the configuration applied to that logger will apply to all instances of that logger.

Bemerkung: Der Name eines `Logger`-Objektes ist Teil eines hierarchischen Namensraumes, das heißt `example.com.testApplication` ist im Namensraum `example.com` enthalten. (Siehe auch Beschreibung der `Logger`-Methode `setUseParentHandlers()` auf Seite 53.)

---

[187] Sie können jeder Protokollmeldung ein Gewicht (*logging level*) zuweisen. Es gibt verschiedene vordefinierte Gewichte und wir empfehlen, diese zu verwenden, statt eigene zu definieren. Tabelle 2.8 zeigt die vordefinierten Gewichte, wobei die Ausführlichkeit der Meldungen von oben nach unten zunimmt.

[188] Die Klasse `Logger` verfügt über eine Reihe von Methoden, mit denen Sie Protokollmeldungen

Gewicht	Empfohlende Verwendung
`SEVERE`	Schwerwiegender Funktionsfehler
`WARNING`	Mögliches Problem
`INFO`	Informationsbenachrichtigung
`CONFIG`	Konfigurationsbenachrichtigung
`FINE`	Protokollmeldung
`FINER`	Ausführliche Protokollmeldung
`FINEST`	Sehr ausführliche Protokollmeldung

Tabelle 2.8: Vordefinierte Gewichte für Protokollmeldungen.

aufzeichnen können, zum Beispiel `info(<msg>)`, `warning(<msg>)` und `severe(<msg>)`. Die folgende Zeile dokumentiert einen schwerwiegenden Funktionsfehler:

```
myLogger.severe("Sending message to standard error");
```

[189] Wenn Sie diese Zeile zum Ausprobieren in ein kleines Testprogramm eingesetzt haben, fragen Sie sich eventuell, wozu das ganze Aufhebens gut ist. Wir haben nichts vorgeführt, das Sie nicht genauso gut mit `System.out.println(<msg>)`-Anweisungen hätten bewerkstelligen können. Beachten Sie aber, daß Sie die Ausgabe von Protokollmeldungen in Ihrer *gesamten* Anwendung mit nur *einer Anweisung* abstellen können:

```
myLogger.setLevel(Level.OFF);
```

[190] Selbst wenn jede Ihrer Klassen Protokollmeldungen mit verschiedenen Gewichten aufzeichnet, genügt diese eine Zeile, um sämtliche Aufzeichnungen zu deaktivieren. Sie müssen nicht nach verstreuten `System.out.println(<msg>)`-Anweisungen suchen.

[191] Unter Umständen möchten Sie einen Block von Protokollanweisungen nur dann ausführen, wenn Sie wissen, daß auch tatsächlich Protokollmeldungen aufgezeichnet werden. Die `Logger`-Methode `isLoggable(level)` prüft, ob eine Meldung mit dem Gewicht `level` von „ihrem“ `Logger`-Objekt aufgezeichnet werden würde. Sie können diese Methode aufrufen, um festzustellen, ob Ihre eventuell performanzlastigen Protokollanweisungen überhaupt ausgeführt werden müssen.

[192] Zu den oben angeführten Vorteilen der Protokollbibliothek gehört, daß die Protokollmeldungen in einer Datei aufgezeichnet werden können. Dafür ist ein `java.util.logging.FileHandler`-Objekt (die `Logger`-Methode `addHandler()` erwartet ein Objekt vom Typ `java.util.logging.Handler`; siehe API-Dokumentation der Klasse `Logger`) erforderlich, das sich über eine einfache Anweisung mit dem `Logger`-Objekt verknüpfen läßt:

```
import java.util.logging.*;
import java.io.IOException;

public class TestLogging {
    public static void main(String[] args) throws IOException {
        Logger myLogger = Logger.getLogger("Test");
        myLogger.addHandler(new FileHandler("temp.log"));
        myLogger.severe("My program did something bad");
    }
}
```

Bemerkung: Ein `FileHandler`-Objekt kann Protokolldateien plattformunabhängig in einem der typischerweise dafür verwendeten Verzeichnisse deponieren und sich um die üblichen Angelegenheiten wie die Rotation der Protokolldateien kümmern (siehe API-Dokumentation der Klasse `FileHandler`).

[193] Das obige Programm liefert die Ausgabe:

```
~$ java TestLogging

Sep 20, 2009 4:22:45 PM TestLogging main

SEVERE: My program did something bad
~$
```

Bemerkung: Ihr Datum und ihre Zeit weichen selbstverständlich ab.

[194] Ihr aktuelles Arbeitsverzeichnis enthält außerdem eine Datei namens *temp.log*, die die Protokollmeldung im XML-Format enthält:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  <date>2009-09-20T16:22:45</date>
  <millis>1253456565708</millis>
  <sequence>0</sequence>
  <logger>Test</logger>
  <level>SEVERE</level>
  <class>TestLogging</class>
  <method>main</method>
  <thread>10</thread>
  <message>My program did something bad</message>
</record>
</log>
```

[195] Die Ausgabe im XML-Format ist praktisch, wenn Sie die Protokollmeldung mit Hilfe eines Programms auswerten möchten. Wenn Sie die Meldungen dagegen selbst untersuchen wollen, eignet sich die Formatierung als einfacher Text besser. Ergänzen Sie das obige Programm folgendermaßen um ein `SimpleFormatter`-Objekt:

```
import java.util.logging.*;
import java.io.IOException;

public class TestLogging {
    public static void main(String[] args) throws IOException {
        FileHandler myFileHandler = new FileHandler("temp.log");
        myFileHandler.setFormatter(new SimpleFormatter());
        Logger myLogger = Logger.getLogger("Test");
        myLogger.addHandler(myFileHandler);

        myLogger.severe("My program did something bad");
    }
}
```

Die Ausgabe auf dem Bildschirm lautet

```
Sep 20, 2009 4:31:02 PM TestLogging main
SEVERE: My program did something bad
```

und stimmt mit dem Inhalt der Datei *temp.log* überein:

```
Sep 20, 2009 4:31:02 PM TestLogging main
SEVERE: My program did something bad
```

[196] Wenn Sie eine andere Ausgabeform haben möchten, könnten Sie sich überlegen, eine eigene Formatierungsklasse zu schreiben. Der Quelltext der Klasse `SimpleFormatter` steht in der Quelltextdistribution des JDKs zur Verfügung und ist nur 80 Zeilen lang, das heißt es ist nicht schwer, eine eigene Formatierungsklasse anzulegen.

[197] Sie haben sicher bemerkt, daß die Protokollmeldungen auf dem Bildschirm ausgegeben werden, obwohl wir dieses Verhalten nicht konfiguriert haben. Das liegt daran, daß jedes `Logger`-Objekt per Voreinstellung den log handler seines Elternobjektes mit seinem eigenen kombiniert. Das voreinstellte anonyme `Logger`-Objekt sendet seine Ausgabe an den Bildschirm. Wenn Sie ein `Logger`-Objekt an das Package `com.example` und ein zweites an `com.example.test` binden, ist das `Logger`-Objekt von `com.example.test` ein Kind des `Logger`-Objektes von `com.example` und jede Meldung an das `Logger`-Objekt von `com.example.test` wird auch an das `Logger`-Objekt von `com.example` gesendet. Sie

können dieses Verhalten deaktivieren, indem Sie die `Logger`-Methode `setUseParentHandlers(boolean useParentHandlers)` des Kind-`Logger`-Objektes aufrufen. (Siehe auch Bemerkung auf Seite 50.)

[198] Entfernen Sie keine Protokollanweisungen, bevor die Anwendung in den produktiven Betrieb übergeht. Es ist nicht auszuschließen, daß Sie eine Protokollanweisung versehentlich löschen und bei späterem Bedarf wieder einsetzen müssen. Es ist besser, die Rotation von Protokolldateien zu nutzen, um zu gewährleisten, daß die Dateien nicht zu groß werden und außerdem das Gewicht der Protokollmeldung so zu setzen, daß nicht zu viele Meldungen aufgezeichnet werden, üblicherweise nur `Level.WARNING` und `Level.SEVERE`. In der Regel wird auch die Ausgabe der Protokollmeldungen auf dem Bildschirm unterbunden (falls Sie noch nicht selbst daran gedacht haben).

2.8 Zusammenfassung

[199] Etwas Zeit in die Explorations- und Designphase eines Projektes zu investieren, kann die Gesamtzeit zur Durchführung eines Projektes erheblich verkürzen. Gut les- und pflegbarer Quelltext bringt Ihnen nicht nur mehr Punkte bei der Bewertung Ihrer Prüfungsaufgabe ein, sondern auch mehr Respekt von Ihren Arbeitskollegen. Der Einsatz der von Sun Microsystems zur Verfügung gestellten Werkzeuge macht Ihre Lösung professioneller.

[200] Wir haben einige Werkzeuge, Fähigkeiten und Eigenschaften aus Version 5 des Java Development Kits vorgestellt, die Ihre Prüfungsaufgabe professioneller machen und Ihre tägliche Arbeit unterstützen.

2.9 Häufige Fragen

- *Frage*: Müssen die Verfahren und Richtlinien aus diesem Kapitel befolgt werden?

  Antwort: Die Anleitung zu Ihrer Prüfungsaufgabe stellt Anforderungen, an die Sie sich halten müssen. Javadoc ist eine dieser Anforderungen. Der Gebrauch von Verfahren und Richtlinien ist aber nicht immer komplett festgelegt. Wenn Ihre Anleitung nicht auf der Einhaltung einer Richtlinie besteht, können Sie sie, allerdings auf eigenes Risiko, beiseite lassen.

- *Frage*: Soll ich meine Modultests auch einreichen?

  Antwort: Zur Zeit enthält jede Prüfungsaufgabe den Hinweis, daß Sie keine Zusatzpunkte bekommen, wenn Sie über die Aufgabenstellung hinausgehen. Wir empfehlen daher, daß Sie Ihre Modultests nicht zusammen mit Ihrer Lösung einreichen. Im besten Fall ignoriert der Gutachter die zusätzlichen Dateien und im schlimmsten Fall zieht der Gutachter Punkte ab, wenn er einen Fehler darin entdeckt. Wenn Sie JUnit zum Testen verwenden, lassen sich Ihre Testklassen ohne *junit.jar* nicht übersetzen, wobei *junit.jar* externer Programmcode ist und Sie keinen externen Programmcode in Ihrer Lösung verwenden dürfen.

- *Frage*: Soll ich die Protokollanweisungen in meiner Lösung stehen lassen?

  Antwort: Wie im Unterabschnitt 2.7.3 beschrieben, raten wir Ihnen, Protokollanweisungen stehen zu lassen. Sie sparen sich dadurch die Arbeit, sämtliche Protokollanweisungen zu entfernen und wichtiger noch, die Arbeit, diese Anweisungen bei Bedarf später wieder einzusetzen. (Wir wissen, daß dies scheinbar der Antwort zu Modultests widerspricht, aber Modultests sind vom Quelltext der eigentlichen Anwendung getrennt, während Protokollanweisungen in die Anwendung integriert sind. Außerdem ist das Package `java.util.logging` Teil der Standardbiblio-

thek, nicht aber JUnit. Ein weiterer Grund, warum es in Ordnung ist, Protokollanweisungen im Quelltext zu belassen, JUnit-Anweisungen aber zu entfernen.)

- *Frage*: Kann ich statt der Datei- und Verzeichnisstruktur im Buch eine eigene verwenden?

  Antwort: Lesen Sie sorgfältig in der Anleitung Ihrer Prüfungsaufgabe nach. Wenn dort eine Datei- und Verzeichnisstruktur angegeben ist, *müssen* Sie sie verwenden. Über die Anleitung hinaus ist jede beliebige Verzeichnisstruktur erlaubt. Wenn Sie die Beispielanwendung in diesem Buch verfolgen, aber eine eigene Verzeichnisstruktur verwenden möchten, müssen Anweisungen und Quelltextbeispiele eventuell überarbeitet werden, um sich wie erklärt zu verhalten. Verwenden nur dann eine eigene Verzeichnisstruktur, wenn Sie bereits mit Javadoc, Klassenpfaden und Packagestrukturen vertraut sind.

Kapitel 3

Einführung in die Beispielanwendung

[0] In diesem Kapitel stellen wir das Beispielprojekt *Denny's DVDs* vor, an dem wir die einzelnen für die Prüfung zum *Sun Certified Java Developer* erforderlichen Gebiete studieren werden. Das Beispielprojekt, bisweilen auch als „die Beispielanwendung" bezeichnet, entspricht in Aufbau und Umfang der Aufgabe, die Sie für Ihre Prüfung bearbeiten müssen und beinhaltet alle wesentlichen Konzepte, die zur Lösung Ihrer eigenen Prüfungsaufgabe erforderlich sind. Jedes Kapitel beschreibt eine der Hauptkomponenten des Beispielprojektes und baut auf den vorangegangenen Kapiteln auf, so daß Sie am Ende des Buches ein vollständiges und funktionstüchtiges Exemplar vom *Denny's DVDs*, Version 1.0[1] installiert und verstanden haben und bedienen können. Das Beispielprojekt nutzt die SE 5 (J2SE 5). Ein Teil der neuen Eigenschaften und Fähigkeiten, zum Beispiel Autoboxing und generische Kollektionen wird in den folgenden Kapiteln erläutert.

3.1 Die wesentlichen Prüfungsanforderungen

[1] Ihr Projekt muß die folgenden Anforderungen erfüllen, damit Sie von Sun Microsystems Kompetenz auf Entwicklerniveau bescheinigt bekommen:

- Sie müssen ein von Sun Microsystems vorgegebenes Interface implementieren.
- Sie müssen entweder Remote Method Invocation (RMI) oder Sockets plus Serialisierung als Netzwerkschnittstelle verwenden.
- Sie müssen zur Datenvisualisierung die Swing-Bibliothek und eine `JTable`-Komponente verwenden.
- Ihre Anwendung muß in einer einzigen ausführbaren *.jar* Datei verpackt sein und daher ohne Kommandozeilenoptionen lauffähig sein.
- Änderungen an der Konfiguration müssen zwischen zwei Aufrufen der Anwendung dauerhaft gespeichert werden.

Ihre zur Begutachtung eingereichte Anwendung muß aus den folgenden Teilen bestehen:

- Eine ausführbare *.jar* Datei, aus der der Client sowohl im *stand-alone*-Betriebsmodus als auch im Netzwerkmodus gestartet werden kann.

[1] *Anmerkung des Übersetzers*: Siehe **Bemerkung**-Box auf Seite 61 zur Versionsbezeichnung der Beispielanwendung.

- Eine ausführbare *.jar* Datei, aus der der per Netzwerk erreichbare Server gestartet werden kann.
- Eine *.jar* Datei, die von Client und Server gemeinsam genutzt wird und enthält:
  - Ein *src*-Verzeichnis, das den Quelltext des Projektes enthält.
  - Die originale (unveränderte) Datenbankdatei, die Sie zusammen mit der Anleitung zu Ihrer Aufgabe erhalten haben.
- Ein *doc*-Verzeichnis mit folgendem Inhalt:
  - Die per `javadoc` generierte API-Dokumentation.
  - Betriebsanleitung für die Benutzer.
- Eine Datei, die Ihre Design-Entscheidungen dokumentiert und begründet.

Ihre gesamte Lösung muß in Form einer *.jar* Datei verpackt werden (siehe Kapitel 9).

Bemerkung: *.jar* Dateien wurden ursprünglich verwendet, um Applets mittels einer einzigen HTTP-Anfrage, statt mehrerer Anfragen und Antworten für jede einzelne Komponente (zum Beispiel Klassen und Graphiken), herunterladen zu können. In der Regel verknüpft Ihre graphische Oberfläche ausführbare *.jar* Dateien unter Windows mit `javaw -jar` und unter Unix/Linux mit `java -jar`.

Die Beispielanwendung wird letztendlich zu einer einzigen *.jar* Datei namens *submission.jar* verpackt. Diese *.jar* Datei enthält wiederum eine ausführbare *.jar* Datei namens *sampleproject.jar*, welche die Datenbankdatei, die Dokumentation und den Quelltext umfaßt. Das Aufrufen der Beispielanwendung aus *sampleproject.jar* wird Abschnitt 9.9 erklärt.

Bemerkung: Die vorige Auflage dieses Buches enthielt ein Kapitel über die neue Ein-/Ausgabebibliothek von Java (das `java.nio`-Package). Damals war die NIO-Bibliothek in Java Version 1.4 neu und das Ausmaß, in dem sie für Zertifizierungsaufgaben verwendet werden durfte unklar. Nach der Veröffentlichung der ersten Auflage dieses Buches und der Herausgabe von J2SE 1.4 untersagte Sun Microsystems explizit den Gebrauch der NIO-Bibliothek *als Netzwerkschnittstelle* bei Zertifizierungsprojekten, erlaubte sie hingegen als Mechanismus zur Dateiein-/ausgabe. Die Motivation für die Aufnahme der NIO-Bibliothek in die vorige Auflage dieses Buch bestand darin, die neuen Eigenschaften und Fähigkeiten vorzuführen. Die damaligen Beispiele im Buch (im „Source Code"-Abschnitt der Apress-Website (*http://www.apress.com/book/sourcecode*) zum Herunterladen) enthielten keine *netzwerkbezogenen* Anweisungen aus der NIO-Bibliothek. Allerdings hat die Präsenz von Anweisungen aus der NIO-Bibliothek viele Leser zu dem irrtümlichen Schluß verleitet, unsere Musterlösung sei notwendig auf die NIO-Bibliothek angewiesen. Wir haben uns daher entschieden, die Diskussion der NIO-Bibliothek in dieser Auflage zu streichen, um erneute Verunsicherung zu vermeiden. Es sei nochmals darauf hingewiesen, daß Sie entweder Sockets oder Remote Method Invocation (RMI) als Netzwerkschnittstelle wählen *müssen*, während Sie die traditionelle Ein-/Ausgabebibliothek nach eigenem Ermessen verwenden dürfen.

3.1.1 Einführung in das Beispielprojekt

[2] Die Beispielanwendung *Denny's DVDs* simuliert eine DVD-Videothek mit kleinem Kundenkreis. Sun Microsystems verlangt in den Zertifizierungsaufgaben, daß die zu entwickelnde Anwendung

auf einem vorgegebenen Interface aufbaut. Sie erhalten das Interface zusammen mit Ihrer Aufgabe und sind dafür verantwortlich, es zu implementieren und eine funktionstüchtige Anwendung zu erarbeiten. Wir verfahren mit unserem Beispielprojekt nach demselben Ansatz. Wir nehmen an, daß Denny, der Inhaber von *Denny's DVDs* ein Inferface definiert hat und von Ihnen (dem Leser) eine Implementierung erwartet. Dieses Interface, `sampleproject.db.DBClient` (*DBClient.java*), ist der Ausgangspunkt unserer Arbeit. Abbildung 3-1, Seite 59 (Buch), zeigt die Anwendungsfälle (*use cases*), die das System unterstützen muß, in einem UML-Diagramm.

Bemerkung: Die in diesem Buch häufig auftretendende Abkürzung UPC steht für „Universal Product Code" und ist eine offizielle Produktbezeichnung, etwa für ein Buch, eine CD oder eine DVD. Der UPC ist eine eindeutige Nummer, über die ein Produkt weltweit identifiziert werden kann. Sie finden bei Wikipedia (*http://en.wikipedia.org/wiki/Main_Page*) sowie in verschiedenen online erreichbaren UPC-Datenbanken viele wissenswerte Informationen über den UPC und ähnliche Kennzeichnungen, wie das European Article Numbering (EAN) und Prüfsummen. Ein ausführlichere Beschreibung geht über den Rahmen dieses Buches hinaus. Für die Zwecke unserer Beispielanwendung ist der UPC ein praktischer systemweiter Identifikator und Primärschlüssel. (Siehe auch **Bemerkung**-Box auf Seite 115.)

---

[3] Die Anwendungsfälle aus Abbildung 3-1 münden in das öffentliche Interface `DBClient`, das die Beispielanwendung implementieren muß. Beschreibung der in `DBClient` deklarierten Methoden:

- `addDVD()`: Erwartet eine Referenz auf ein Objekt vom Typ `DVD` als Eingabeparameter. Ist die per UPC eindeutig identifierbare DVD noch nicht in der Datenbankdatei registriert, fügt die Methode die DVD hinzu. Andernfalls wirft die Methode eine Ausnahme aus.
- `getDVD()`: Erwartet einen UPC als Eingabeparameter. Ist die entsprechende DVD in der Datenbankdatei registriert, so gibt die Methode eine Referenz auf das entsprechende `DVD`-Objekt zurück. Andernfalls gibt die Methode `null` zurück.
- `modifyDVD()`: Erwartet eine Referenz auf ein Objekt vom Typ `DVD` als Eingabeparameter. Ist in der Datenbankdatei eine DVD mit dem übergebenen UPC registriert, so werden die Felder des entsprechenden Datensatzes aktualisiert. Ist in der Datenbankdatei dagegen *keine* DVD mit dem übergebenen UPC registriert, so gibt die Methode `false` zurück und die Datenbankdatei wird *nicht* modifiziert.
- `getDVDs()`: Erwartet keinen Eingabeparameter. Gibt eine Kollektion mit allen in der Datenbankdatei registrierten DVDs zurück. Sind keine DVDs in Datenbankdatei registriert, so ist die zurückgelieferte Kollektion leer.

Bemerkung: Die Beispielanwendung, die Sie aus dem „Source Code"-Abschnitt der Apress-Website (*http://www.apress.com/book/sourcecode*) herunterladen können, enthält eine Datenbankdatei mit einigen Mustereinträgen, auf die wir uns in den Beispielen in diesem Buch beziehen.

- `findDVD()`: Erwartet einen regulären Ausdruck (Referenz auf ein `String`-Objekt) als Eingabeparameter, der auf die Verknüpfung aller Felder eines DVD-Datensatzes angewendet wird. Der Eingabeparameter ist der reguläre Ausdruck selbst, das heißt die Anwendung muß das Suchkriterium des Benutzers *vor* dem Aufruf der Methode `findDVDs()` in einen regulären Ausdruck umwandeln. Die Umwandlung des Suchkriteriums in einen regulären Ausdruck kann entweder in der graphischen Benutzeroberfläche oder in einer Hilfsklasse vorgenommen werden, nicht aber in `findDVD()` selbst. Selbstverständlich dürfen Sie nicht vom Benutzer erwarten, den regulären Ausdruck selbst als Suchkriterium zu formulieren. Die Methode gibt eine Kollektion

von Referenzen auf `DVD`-Objekte zurück.

- `releaseDVD()`: Erwartet einen UPC als Eingabeparameter. Inkrementiert den Bestandszähler einer ausleihbaren DVD. Diese Methode entspricht dem Zurückgeben einer DVD. Die Methode gibt `true` zurück, wenn die DVD in der Datenbankdatei registriert ist und noch Exemplare zum Ausleihen verfügbar sind. Andernfalls gibt die Methode `false` zurück.
- `reserveDVD()`: Erwartet einen UPC als Eingabeparameter. Dekrementiert den Bestandszähler einer ausleihbaren DVD und zeigt an, daß eines der verfügbaren Exemplare ausgeliehen wurde. Ist der übergebene UPC nicht in der Datenbankdatei registriert, so gibt die Methode `false` zurück. Die Methode gibt auch dann `false` zurück, wenn keine ausleihbaren Exemplare der DVD mehr verfügbar sind.
- `removeDVD()`: Erwartet einen UPC als Eingabeparameter. Löscht die DVD mit dem entsprechenden UPC aus der Datenbankdatei und gibt `true` zurück. Ist der UPC nicht in der Datenbankdatei registriert, so wird keine DVD gelöscht und die Methode gibt `false` zurück.

Tipp: Legen Sie mit Hilfe der obigen Beschreibung eine eigene Version des Interfaces ***DBClient*** an, bevor Sie das Beispielprojekt aus dem „Source Code"-Abschnitt der Apress-Website (*http://www-.apress.com/book/sourcecode*) herunterladen und vergleichen Sie anschließend wie gut Ihr Interface mit *DBClient.java* übereinstimmt. (Ihre Prüfungsaufgabe enthält das zu implementierende Interface bereits.)

---

[4] Die Beispielanwendung benötigt eine graphische Benutzeroberfläche, über welche die im Interface ***DBClient*** deklarierten Methoden aufgerufen werden können. Die graphische Benutzeroberfläche muß sowohl im Netzwerkmodus mit einem entfernten Server, als auch im *stand-alone*-Betriebsmodus mit einem lokalen Server kommunizieren können. Beim Entwurf der Anwendung muß gleichzeitiger Zugriff durch mehrere Benutzer und das Sperren von Datensätzen berücksichtigt werden. Der Quelltext des Interfaces ***DBClient*** lautet:

```
package sampleproject.db;

import java.io.*;
import java.util.regex.*;
import java.util.*;

/**
  * An interface implemented by classes that provide access to the Dvd
  * data store, including DvdDatabase.
  *
  * @author Denny's DVDs
  * @version 2.0
  */
public interface DBClient {

    /**
      * Adds a DVD to the database or inventory.
      *
      * @param dvd The DVD item to add to inventory.
      * @return Indicates the success/failure of the add operation.
      * @throws IOException Indicates there is a problem accessing the database.
      */
    public boolean addDVD(DVD dvd) throws IOException;

    /**
      * Locates a DVD using the UPC identification number.
      *
```

```
  * @param UPC The UPC of the DVD to locate.
  * @return The DVD object which matches the UPC.
  * @throws IOException if there is a problem accessing the data.
  */
public DVD getDVD(String UPC) throws IOException;

/**
  * Changes existing information of a DV item.
  * Modifications can occur on any of the attributes of DVD except UPC.
  * The UPC is used to identify the DVD to be modified.
  *
  * @param dvd The Dvd to modify.
  * @return Returns true if the DVD was found and modified.
  * @throws IOException Indicates there is a problem accessing the data.
  */
public  boolean modifyDVD(DVD dvd) throws IOException;

/**
  * Removes DVDs from inventory using the unique UPC.
  *
  * @param UPC The UPC or key of the DVD to be removed.
  * @return Returns true if the UPC was found and the DVD was removed.
  * @throws IOException Indicates there is a problem accessing the data.
  */
public  boolean removeDVD(String UPC) throws IOException;

/**
  * Gets the store's inventory.
  * All of the DVDs in the system.
  *
  * @return A List containing all found DVD's.
  * @throws IOException Indicates there is a problem accessing the data.
  */
public List<DVD> getDVDs() throws IOException;

 /**
   * A properly formatted <code>String</code> expressions returns all
   * matching DVD items. The <code>String</code> must be formatted as a
   * regular expression.
   *
   * @param query The formatted regular expression used as the search
   * criteria.
   * @return The list of DVDs that match the query. Can be an empty
   * Collection.
   * @throws IOException Indicates there is a problem accessing the data.
   * @throws PatternSyntaxException Indicates there is a syntax problem in
   * the regular expression.
  */
public Collection<DVD> findDVD(String query)
       throws IOException, PatternSyntaxException;

/**
  * Lock the requested DVD. This method blocks until the lock succeeds,
  * or for a maximum of 5 seconds, whichever comes first.
  *
  * @param UPC The UPC of the DVD to reserve
  * @return true if the DVD was reserved
  * @throws InterruptedException Indicates the thread is interrupted.
  * @throws IOException on any network problem
  */
```

```
    boolean reserveDVD(String UPC) throws IOException, InterruptedException;

    /**
      * Unlock the requested record. Ignored if the caller does not have
      * a current lock on the requested record.
      *
      * @param UPC The UPC of the DVD to release
      * @throws IOException on any network problem
      */
    void releaseDVD(String UPC) throws IOException;
}
```

Bemerkung: Die Beispielanwendung ist keine umfangreiche E-Commerce-Applikation, sondern dient zur Demonstration der Gebiete, die Sie beherrschen müssen, um den Entwicklungsanteil der Prüfung zum *Sun Certified Java Developer* erfolgreich abzuschließen.

3.1.2 Technischer Überblick

[5] In diesem Abschnitt präsentieren wir einen Überblick über die Beispielanwendung und beschreiben die erforderlichen Schritte, um die Anwendung zu implementieren.

[6] Die Beispielanwendung ist hinsichtlich ihrer Architektur ein traditionelles Client-Server-System und besteht im wesentlichen aus drei Komponenten, nämlich einer serverseitigen Datenbankdatei zusammen mit der Funktionalität eines Netzwerkservers, einer clientseitigen graphischen Benutzeroberfläche und einem clientseitigen Datenbankinterface, das sich im Auftrag der Benutzerschnittstelle um das Netzwerk kümmert. Abbildung 3-2, Seite 63 (Buch), zeigt die Beispielanwendung im Grobüberblick. Die zentralen Anforderungen an die Beispielanwendung lauten:

- Die Benutzer der Anwendung, also die Mitarbeiter von *Denny's DVDs*, müssen die im Interface `DBClient` deklarierten Methoden über eine graphische Benutzeroberfläche ausführen können.
- Die Anwendung muß Mehrbenutzerzugriff auf eine zentrale Datenbankdatei gestatten. Der Zugriff auf diese Datenbankdatei muß threadsicher sein.
- Die Benutzer müssen beim Zugriffsmodus auf die Datenbankdatei über ein Netzwerk zwischen Remote Method Invocation (RMI) und Sockets wählen können.

Tipp: Der Quelltext der Beispielanwendung, den Sie aus dem „Source Code"-Abschnitt der Apress-Website (*http://www.apress.com/book/sourcecode*) herunterladen können, enthält eine Datei namens *DosClient.java*. Die Klasse `DosClient` ist ein nützliches Hilfsprogramm für die Kommandozeile und stammt aus der ersten Auflage dieses Buches, wird aber, infolge einiger Änderungen, bei Version 2.0 der Beispielanwendung nicht mehr benötigt. Wir gehen im Buch nicht mehr auf `DosClient` ein, obwohl das Programm noch zur Distribution der Beispielanwendung gehört. Stellen Sie sich `DosClient` als eine „Kommandozeilenversion der graphischen Benutzeroberfläche" vor.

3.1.2.1 Graphische Benutzeroberfläche

[7] Ein Mitarbeiter von *Denny's DVDs* muß über die graphische Benutzeroberfläche die Felder eines DVD-Datensatzes abfragen können, zum Beispiel den UPC, den Namen des Films, den Verleihstatus, den Regisseur, die Schauspieler und Schauspielerinnen, den Komponisten und die Anzahl der

verfügbaren Exemplare. Die graphische Benutzeroberfläche muß dem Mitarbeiter die Suche nach DVDs bezüglich dieser Felder ermöglichen. Abbildung 3-3, Seite 65 (Buch), zeigt alle benötigten Felder einer DVD in Form eines Klassendiagramms. Die graphische Benutzeroberfläche muß dem Mitarbeiter gestatten, sich entweder mit einer lokalen oder über ein Netzwerk mit einer entfernten Datenbankdatei zu verbinden. Sicherheitsaspekte wie Authentifizierung oder Anmeldung müssen nicht berücksichtigt werden müssen. Da die Prüfung zum *Sun Certified Java Developer* zur Zeit eine graphische Benutzeroberfläche basierend auf der Swing-Bibliothek verlangt, entwickeln wir die graphische Schnittstelle der Beispielanwendung ebenfalls mit Hilfe von Swing.

Bemerkung: Swing ist eine Bibliothek für graphische Benutzeroberflächen und baut auf dem Abstract Window Toolkit (AWT) auf, einem Teil der Java Foundation Classes (JFC). Die Prüfung zum *Sun Certified Java Programmer* verlangt keine AWT-Kenntnisse mehr. Die Prüfung zum *Sun Certified Java Developer* verlangt keine direkten Kenntnisse über die Verbindung zwischen AWT und Swing.

3.1.2.2 Datenbankdatei und Netzwerkschnittstelle

[8] Version 1.0 der Beispielanwendung kann entweder lokal, das heißt im *stand-alone*-Betriebsmodus oder über ein Netzwerk hinweg betrieben werden. Im lokalen Modus verbindet sich die graphische Benutzeroberfläche nur dann mit der Datenbankdatei, wenn sich diese auf demselben Rechner befindet. Im Netzwerkmodus verbindet sich die graphische Benutzeroberfläche dagegen von einem beliebigen Rechner im Netzwerk aus mit dem dem Server, der die Datenbankdatei pflegt. Die Netzwerkschnittstelle kann entweder mit Sockets oder per RMI implementiert werden, siehe Abbildung 3-4, Seite 65 (Buch). Wir führen beide Ansätze in den Kapiteln 5 und 6 vor.

Bemerkung: Obwohl es nur eine Version der Beispielanwendung *Denny's DVDs* gibt (insbesondere also keine Version 2.0), bezeichnen wir die beschriebene Fassung der Beispielanwendung stets als Version 1.0. Vielleicht wird es in der dritten Auflage dieses Buches eine Version 2.0 geben.

---

[9] Da sich mehrere Clients über ein Netzwerk mit dem Server verbinden und versuchen können, ein und denselben Datensatz gleichzeitig zu bearbeiten, muß die Beispielanwendung threadsicher sein. Wie besprechen Threadsicherheit detailliert in Kapitel 4. Im Augenblick genügt es zu wissen, daß eine threadsichere Anwendung den Zustand eines Objektes schützt, wenn mehrere Clients auf dieses Objekt zugreifen und dabei versuchen, seinen Zustand zu ändern. Sie sind im Rahmen Ihrer Zertifizierung dafür verantwortlich, daß Ihre eingereichte Anwendung threadsicher ist (siehe Abschnitt 4.2).

[10] Clients müssen nicht benachrichtigt werden, wenn ein angezeigter Datensatz modifiziert wurde. Versuchen dagegen zwei Mitarbeiter ein und dieselbe DVD auszubuchen, so kann die DVD nur einem Kunden ausgeliehen werden. Hier ist das Sperren von Datensätzen (*record locking*) erforderlich. Kapitel 4 widmet sich detailliert den Sperrmechanismen für Objekte. Kapitel 5 beschreibt das Sperren von Datensätzen an einem Beispiel.

[11] Die Anwendung muß auch ohne Netzwerk lauffähig sein. Im *stand-alone*-Betriebsmodus werden die Datenbankdatei und die graphische Benutzeroberfläche von ein und derselben Laufzeitumgebung betrieben, es findet keine Kommunikation mit dem Netzwerk statt und es sollen keine Sockets erzeugt werden. In den Kapiteln 6 und 7 werden separate Implementierungen für RMI und Sockets vorgestellt.

3.2 Zusammenfassung

[12] In diesem Kapitel haben wir das Interface `DBClient` der Beispielanwendung *Denny's DVDs* besprochen. Wir haben die Anforderungen an das Projekt und alle in `DBClient` deklarierten Methoden besprochen, für deren Implementierung wir verantwortlich sind. Sie können den Quelltext der Beispielanwendung aus dem „Source Code"-Abschnitt der Apress-Website (*http://www.apress-.com/book/sourcecode*) herunterladen.

[13] Die Datenbankdatei der Beispielanwendung muß sowohl im *stand-alone*-Betriebsmodus als auch im Netzwerkmodus von mehreren Clients aus erreichbar sein. Die graphische Benutzeroberfläche soll intuitiv und leicht zu benutzen sein. Die Beispielanwendung muß threadsicher sein. Der Rest dieses Buches beschreibt und diskutiert die in diesem Kapitel angesprochenen Anforderungen, wobei die Beispielanwendung von Kapitel zu Kapitel weiterwächst.

3.3 Häufige Fragen

- *Frage*: Die Anleitungen zu den Zertifizierungsprojekten verlangen, daß die Anleitungen selbst und die Datenbankdatei mit eingereicht werden müssen. Warum ist das erforderlich?

  Antwort: Sun Microsystems vergibt verschiedene Prüfungsaufgaben, von denen einige wiederum mehrere Versionen haben. Manche Änderungen in den Prüfungsaufgaben wirken sich auf das Format der Datenbankdatei, das vorgegebene Interface, die vorgeschriebenen Namen von Klassen oder einer Kombination dieser Faktoren aus. Indem Sie *Ihre Anleitung* und *Ihre Datenbankdatei* einreichen, können Sie davon ausgehen, daß der Gutachter Ihre Lösung mit Ihrer Anleitung vergleicht und nicht mit der Anleitung eines anderen Prüfungskandidaten.

- *Frage*: Ich würde gerne einen Ausnahmetyp verwenden, der im vorgegebenen Interface nicht deklariert ist. Ist das zulässig?

  Antwort: Nein. Die Anleitung zu Ihrer Aufgabe weist darauf hin, daß sich andere Anwendungen eventuell auf Ihr vorgegebenes Interface beziehen, den zusätzlichen Ausnahmetyp also abfangen müßten. Ein zusätzlicher Ausnahmetyp könnte die Funktionstüchtigkeit einer anderen Anwendung gefährden.

- *Frage*: Ich bin der Ansicht, daß meine Anwendung besser läuft, wenn ich dem Interface eine weitere Methode hinzufüge. Ist das erlaubt?

  Antwort: Ja, aber wir raten in den Prüfungsaufgaben davon ab. Das Hinzufügen einer neuen Methode zum Interface verletzt den Vertrag zwischen dem Datenformat und der Benutzerschnittstelle. Eventuell fällt Ihre eingesendete Lösung damit sogar durch. Es ist sinnvoller, die Methode einer Klasse hinzuzufügen, die das Interface implementiert. Wir möchten Sie ausdrücklich ermutigen, zusätzliche Methoden in Klassen (nicht aber in dem von Sun Microsystems vorgegebenen Interface) anzulegen oder bereits implementierte Klassen zu ändern, um das Beispielprojekt und seinen Quelltext besser zu verstehen.

- *Frage*: Wie ähnlich ist die Beispielanwendung der von Sun Microsystems gestellten Aufgabe?

  Antwort: Es gibt keine Garantie dafür, wie zukünftige Prüfungsaufgaben von Sun Microsystems aussehen, aber das Beispielprojekt führt die Gebiete vor, die Sie beherrschen müssen, um die Entwicklerzertifizierung zu bestehen und demonstriert außerdem neue Spracheigenschaften und -fähigkeiten der J2SE 5.

- *Frage*: Wie soll ich mit den Quelltextbeispielen umgehen?

Antwort: Die vollständige Distribution der Beispielanwendung steht Ihnen im „Source Code"-Abschnitt der Apress-Website (*http://www.apress.com/book/sourcecode*) zum Herunterladen zur Verfügung. Jedes Kapitel beschreibt einen Entwicklungsschritt der Beispielanwendung, ausgehend von einem Teil des Interfaces `DBClient`, bis hin zur vollständigen Anwendung. Arbeiten Sie, während Sie das Buch lesen, die entsprechenden Teile des Quelltextes durch, um die gesamte Implementierung zu verstehen. Kapitel 9 erläutert detailliert, wie Sie die Beispielanwendung übersetzen und ausführen (analog zum Übersetzen und Ausführen Ihrer Prüfungsaufgabe).

Alternativ können Sie zuerst den Quelltext der Beispielanwendung durcharbeiten und anschließend das Buch lesen, um die Begründungen für die Design-Entscheidungen nachzulesen und zu verstehen, wie diese Entscheidungen mit ähnlichen Entscheidungen zusammenhängen, die Sie bei Ihrer Prüfungsaufgabe treffen müssen. Wir empfehlen, erst das Buch zu lesen und anschließend den Quelltext durchzuarbeiten, aber auch die umgekehrte Reihenfolge sollte zum Erfolg führen. Es kommt darauf an, welche Vorgehensweise mehr Ihrem Lernverhalten entspricht.

- *Frage*: Kann ich das Beispielprojekt um Eigenschaften oder Fähigkeiten ergänzen, die nicht im Buch diskutiert werden?

  Antwort: Selbstverständlich können Sie die Beispielanwendung verändern und ausfeilen. Tatsächlich sind Neugier und Experimentierfreudigkeit sehr wichtige Eigenschaften für einen Entwickler. Das Beispielprojekt wurde allerdings mit Sorgfalt entworfen, um zwei Zielen zu dienen: Das erste und wichtigere Ziel besteht darin, alle für die Prüfung zum *Sun Certified Java Developer* erforderlichen Themen abzudecken. Denken Sie an den Zweck des Beispielprojektes, wenn Sie improvisieren. Das zweite Ziel besteht darin, die neuen Eigenschaften der J2SE 5 vorzuführen, zum Beispiel Autoboxing und generische Kollektionen.

Teil II

Implementierung der Prüfungsaufgabe

Kapitel 4

Threads

[0] In diesem Kapitel besprechen wir Threads, indem wir das Thema in überschaubare Abschnitte und Unterabschnitte aufteilen, in denen wir realistische Beispiele diskutieren und anschauliche Vergleiche anführen. Das Kapitel erläutert die konzeptionellen und technischen Einzelheiten von Threads, die Sie verstehen müssen, um sowohl den Entwicklungsteil als auch die schriftliche Prüfung der Zertifizierung zum *Sun Certified Java Developer* zu bestehen.

[1] Die ersten Abschnitte dieses Kapitels führen in das Gebiet „Threads“ und die Probleme durch die Anwesenheit mehrerer Threads (Multithreading) ein. Der Wartezustand eines Threads und das Sperren von Objekten werden in allen Einzelheiten erklärt. Die Einführung in das Gebiet „Threads“ endet mit einem Abschnitt über Threadsicherheit, in dem die Themen „Verklemmungen“ (*deadlocks*), „Verhungern“ (*starvation*), „Wettlaufsituationen“ (*race conditions*) und „Monitore“ (*monitors*) diskutiert werden.

[2] Der beiden folgenden Abschnitte diskutieren verschiedene direkte Anwendungsfälle für `Thread`-Objekte, Threads im Hinblick auf graphische Benutzeroberflächen (Swing) und eine Auswahl bewährter Verfahren beim Programmieren mit Threads. Das Kapitel endet mit einem Abschnitt über Fragen, die häufig im Zusammenhang mit Threads gestellt und diskutiert werden. Die folgenden Themen werden behandelt:

- Einführung Threads und Multithreading.
- Sperren von Objekten und Synchronisieren von Threads.
- Die Sperrmechanismen in Version 5 des Java Development Kits.
- Effektives Koordinieren von Threads mit Hilfe der Methoden `wait()`, `sleep()` und `yield()`.
- Die Bedeutung von Threadsicherheit.
- Sperren von Datensätzen in der Beispielanwendung.

Bemerkung: Dieses Kapitel dient als Einführung in das Thema „Threads“. Zum vertieften Studium empfehlen wir Allen Holubs Buch *Taming Java Threads*, Apress (2000).

4.1 Grundlagen

[3] Wir fassen, bevor wir uns mit dem eigentlichen Inhalt dieses Kapitels auseinandersetzen, einige Eigenschaften von Threads zusammenfassen, die Ihnen bereits vertraut sein sollten. Bei Java gibt es zwei Möglichkeiten, um einen Ausführungsfaden zu erzeugen, nämlich durch Ableiten der Klasse `java.lang.Thread` und durch Implementieren des Interfaces *`java.lang.Runnable`*. Sie starten einen Thread, indem Sie seine `start()`-Methode aufrufen. Threads können scheinbar parallel verarbeitet werden. Diese grundlegenden Tatsachen über Threads müssen Ihnen bekannt sein, um die Prüfung zum *Sun Certified Java Programmer* zu bestehen und Sie müssen sie beherrschen, wenn Sie die Prüfung zum *Sun Certified Java Developer* anstreben.

4.1.1 Motivation zur Verwendung von Threads

[4] Ein Thread ist ein unabhängiger Ausführungsfaden der einige Anweisungen verarbeitet. Stellen Sie sich zum Beispiel ein Textverarbeitungsprogramm vor, das zwei Operationen ausführen muß, etwa das aktuelle Dokument drucken und sichern, siehe Abbildung 4-1, Seite 72 (Buch).

[5] Ein Textverarbeitungsprogramm, das stets höchstens eine Anweisung ausführen kann (*single-threaded* im Gegensatz zu *multithreaded*), wäre nicht in der Lage, mit dem Speichern des Dokumentes zu beginnen, bevor der Druckvorgang beendet ist (Abbildung 4-1, Seite 72 (Buch)). Obwohl dieser Ansatz zweifellos funktioniert, ist er unnötig ineffizient (diejenigen unter uns, die sich an die ersten Textverarbeitungsprogramme erinnern können, werden das bestätigen).

[6] Stellen Sie sich nun vor, daß das Textverarbeitungsprogramm zwei voneinander unabhängige Threads erzeugen kann. Der eine Thread kümmert sich um das Drucken, der andere sichert das Dokument im Hintergrund. Dieser Ansatz führt sehr wahrscheinlich zu einem effizienteren Ergebnis, da für die beiden Aufgaben keine Notwendigkeit besteht, aufeinander zu warten. Beispielsweise kann die Datei bereits gesichert werden, während der Druckthread noch die Netzwerkverbindung aufbaut. Dieses Multithreading-Konzept ist in Abbildung 4-2, Seite 72 (Buch) dargestellt und wird im nächsten Unterabschnitt detailliert beschrieben.

4.1.2 Multithreading

[7] Die Koordination der verschiedenen in einer Anwendung laufenden Threads wird als „Multithreading“ bezeichnet und macht das System effizienter. Multithreading äußert sich durch abwechselnde oder gemeinsame Verwendung von Resourcen, verringerte Unkosten hinsichtlich der Laufzeit und des Speicherbedarfs sowie dadurch, daß die einzelnen Threads gegenseitig ihre Grenzen respektieren.

[8] Die Anwesenheit mehrerer Threads verursacht, bedingt durch die Art und Weise in der Prozessoren Threads verarbeiten, einige inherente Schwierigkeiten. Wir besprechen diese Probleme im Abschnitt 4.2 in allen Einzelheiten.

[9] Ein Prozessor verarbeitet einige Anweisungen eines Threads, schaltet um und verarbeitet einige Anweisungen eines anderen Threads, schaltet wieder um, ... Da das Umschalten schnell vor sich geht, entsteht der Eindruck, daß die Threads gleichzeitig und unabhängig voneinander verarbeitet werden. Insbesondere wirkt sich ein inaktiv wartender Thread, der zum Beispiel auf eine Netzwerkverbindung wartet, nicht als Engpaß für andere Threads aus. Die verbesserte Effizienz ist einer der größten Vorteile des Multithreadings, fordert aber auch ihren Preis: Ohne entsprechende Vorkehrungen kann ein Thread Daten modifizieren, die ein anderer Thread benötigt. Wir demonstrieren diese Probleme im Abschnitt 4.2 und beschreiben, wie Sie durch defensives Programmieren dagegen vorgehen können.

Bemerkung: In diesem Kapitel diskutieren wir das Thema „Threads“ aus der Perspektive einer Laufzeitumgebung auf einem Rechner mit nur einem Prozessor. Bei Rechnern mit mehr als einem Prozessor ist es möglich, daß mehrere Threads auf verschiedenen Prozessoren tatsächlich gleichzeitig verarbeitet werden. Die auftretenden Probleme sind aber identisch.

[10] Stellen Sie sich in dem Beispiel mit dem Textverarbeitungsprogramm vor, daß ein weiterer Thread den Text im Blocksatz formatiert, sich aber nicht mit dem Druckthread koordiniert. Als Ergebnis erhalten Sie eventuell einen Teil des gedruckten Dokumentes im Blocksatz, einen anderen Teil dagegen nicht.

[11] Dies waren einige Beispiele für die Probleme, die Sie erwarten, wenn Sie das Gebiet „Multithreading“ betreten. Die Prüfung zum *Sun Certified Java Developer* setzt voraus, daß Sie Multithreading und die damit verbundenen Schwierigkeiten sorgfältig durchdacht haben.

4.1.3 Multithreadingunterstützung bei Java

[12] Java ist eine der wenigen Programmiersprachen mit eingebauter Unterstützung für Multithreading. Diese Fähigkeit bringt zwei wichtige Verhaltensmerkmale mit sich: Erstens kann jedes Java-Objekt für den exklusiven Zugriff durch genau einen Thread gesperrt werden (Synchronisierung von Threads bezüglich eines Objektes). Ein gesperrtes Objekt ist für alle Threads unzugänglich, die die Synchronisierung respektieren, mit Ausnahme des einen Threads, der die Sperrung verursacht hat. Zweitens kann ein Java-Objekt alle Threads die exklusiven Zugriff wünschen mit Hilfe einer Art Warteliste „im Auge behalten“.

[13] Das grundsätzliche Problem bei Anwesenheit mehrerer Threads ähnelt der Erziehung kleiner Kinder: Wie verhindern Sie, daß sich die Threads (Kinder) sich um einen begrenzten Resourcenvorrat (Spielzeug) streiten?

[14] Bei Java besteht die Lösung aus zwei Hälften: Zum einen muß jeder Thread, der Zugriff auf ein Objekt benötigt, dem Protokoll gehorchen und prüfen, ob das Objekt gerade von einem anderen Thread verwendet wird. Zum anderen muß jeder Thread, der den Zustand des Objektes ändert, das Objekt markieren, um anzuzeigen, daß das Objekt im Augenblick verwendet wird. (Wenn es doch mit Kindern genauso einfach wäre.)

[15] Ein anschaulicher Vergleich aus der Realität für die zweite Hälfte ist die Anmeldung für das Laufband im Fitneßstudio. Wenn Sie (der Thread) ein allgemein verfügbares Objekt (das Laufband) verwenden möchten, müssen Sie nachsehen, ob gerade ein anderer Sportler (Thread) das Laufband benutzt. Ist das Gerät besetzt, so müssen Sie warten. Andernfalls, registrieren Sie sich und benutzen es.

4.1.3.1 Threads im Wartezustand

[16] Ein wartender Thread „tritt beiseite“, wenn seine Ausführung vorübergehend nicht fortgesetzt werden kann und geht in einen inaktiven Zustand über, das heißt ein wartender Thread verbraucht keine Rechenzeit. Ein wartender Thead stellt sein Kontingent an Rechenzeit anderen Threads zur Verfügung, da er es mit Sicherheit nicht braucht.

[17] Stellen Sie sich eine Gruppe Kinder vor, die bei einem Eisverkäufer ein Eis kaufen wollen. Wenn der Junge, der gerade mit dem Eisverkäufer spricht, nicht genug Geld dabei hat, wird er die anderen nicht zwingen, solange zu warten, bis sein Vater kommt und das benötigte restliche Geld bringt. Er

wird statt dessen aus dem Weg gehen und wahrscheinlich leise weinend warten. Dann wird er sich wieder ins Gewühl stürzen und versuchen, den Eisverkäufer auf sich aufmerksam zu machen. Die anderen Kinder werden nicht auf ihn warten, da er kein Eis bestellen kann.

Bemerkung: Der Vergleich mit den Kindern, von denen ein jedes begehrlich versucht, die Aufmerksamkeit des Eisverkäufers auf sich zu lenken, ist nicht zufällig gewählt. Wetteifernde Threads verhalten sich nicht wie höfliche Erwachsene in der Schlange beim Gemüsehändler. Es gibt keine Reihenfolge. Alle Threads versuchen sich gegenseitig auszustechen und einer gewinnt, abhängig von den Eigenschaften des unterliegenden Betriebssystems, Prioritäten oder anderen Kriterien.

---

[18] Es gibt zwei Arten von Wartezuständen, während deren Dauer die Verarbeitung eines Threads unterbrochen wird, bis eine bestimmtes Ereignis eintritt. In der ersten Variante werden die vom Thread verursachten Objektsperren aufrechterhalten, während die zweite Variante die Sperren vor dem Eintritt in den Wartezustand aufhebt. Für die erste Variante ist kennzeichnend, daß der Thread auf ein Ereignis warten muß, um weiterarbeiten zu können, beispielsweise auf Daten, die über eine Netzwerkverbindung gesendet werden. Die zweite Variante liegt dagegen vor, wenn der Thread eine Methode aufruft, deren Dokumentation explizit aussagt, daß die (von diesem Thread bewirkten) Objektsperren aufgehoben werden. Ruft ein Thread beispielsweise die `Object`-Methode `wait()` auf, so hebt `wait()` die von diesem Thread verursachte Sperre desjenigen Objektes auf, zu dem die `wait()`-Methode gehört.

Bemerkung: Wir besprechen das Sperren von Objekten im gleichnamigen Unterabschnitt in diesem Kapitel (Seite 83). Stellen Sie sich eine Sperre bis dahin wie eine „wird gerade verwendet"-Markierung an einem Objekt vor.

---

[19] Geht Ihr Thread durch einen Aufruf der `Thread`-Methoden `yield()` oder `sleep()` oder das Warten auf ein Ereignis, beispielsweise eine Ein-/Ausgabeoperation, in den Wartezustand über, so erhält der Thread alle Sperren aufrecht. Kein anderer Thread kann eines der gesperrten Objekte verwenden. Versucht ein Thread auf ein gesperrtes Objekt zuzugreifen, so wird er blockiert, bis die Resource wieder freigegeben wird.

[20] Ruft ein Thread die `wait()`-Methode eines Objektes auf, so wird die von diesem Thread verursachte Sperrung dieses Objektes wieder aufgehoben, falls dieser Thread zuvor die Sperrung dieses Objektes veranlaßt hat. Andernfalls, wenn das Objekt also nicht zuvor von diesem Thread gesperrt wurde, bewirkt `wait()`, daß eine Ausnahme vom Typ `IllegalMonitorStateException` ausgeworfen wird. Das Aufheben der Sperre ist praktisch, da andere Threads nun in der Lage sind, den Zustand des Objektes zu ändern. Das Eisverkäufer-Beispiel in diesem Kapitel veranschaulicht diese Möglichkeit.

[21] Stellen Sie sich im folgenden Beispiel zum Verständnis der `wait()`-Methode einen Pfarrer vor, der mit einem Teller für die Kollekte an den Teilnehmern des Gottesdienstes vorbei geht. Der Pfarrer erwartet, daß die Anwesenden eine Spende auf den Teller legen und verharrt solange, bis sie es tun. Legen die Anwesenden keine Spende auf den Teller, so wartet der Pfarrer solange, bis sie es doch tun. Die Situation stellt sich in Form von Quelltext etwa wie im nächsten Beispiel dar. Lassen Sie sich nicht entmutigen, wenn sie dieses und die folgenden Quelltextbeispiele noch nicht komplett verstehen. Lesen Sie das ganze Kapitel und alles findet seinen Platz:

```
01. public class Minister {
02.     private CollectionPlate collectionPlate = new CollectionPlate();
03.
04.     public static void main(String[] args) {
```

```
05.         Minister minister = new Minister();
06.
07.         // create a Thread that checks  the amount on
08.         // money in the collection plate.
09.         minister.new CollectionChecker().start();
10.
11.         //create several threads to accept contributions.
12.         for (int i = 0; i < 6; i++) {
13.             minister.new CollectionAcceptor(20).start();
14.         }
15.     }
16.
17.     /**
18.      * the collection plate that get passed around
19.      */
20.     private class CollectionPlate {
21.         int amount = 0;
22.     }
23.
24.     /**
25.      * Thread that accepts collections.
26.      */
27.     private class CollectionAcceptor extends Thread {
28.         int contribution = 0;
29.
30.         public CollectionAcceptor(int contribution) {
31.             this.contribution = contribution;
32.         }
33.
34.         public void run() {
35.             //Add the contributed amount to the collectionPlate.
36.             synchronized (collectionPlate) {
37.                 int amount = collectionPlate.amount + contribution;
38.                 String msg = "Contributing: current amount: " + amount;
39.                 System.out.println(msg);
40.                 collectionPlate.amount = amount;
41.                 collectionPlate.notify();
42.             }
43.         }
44.     }
45.
46.     /**
47.      * Thread that checks  the collections made.
48.      */
49.     private class CollectionChecker extends Thread {
50.         public void run() {
51.             // check the amount of money in the collection plate. If it's less
52.             // than  100, then release the collection plate, so other Threads
53.             // can modify it.
54.             synchronized (collectionPlate) {
55.                 while (collectionPlate.amount < 100) {
56.                     try  {
57.                         System.out.println("Waiting ");
58.                         collectionPlate.wait();
59.                     } catch  (InterruptedException ie) {
60.                         ie.printStackTrace();
61.                     }
```

```
62.                 }
63.                 // getting past the while statement means that the
64.                 // contribution goal has been met.
65.                 System.out.println("Thank you");
66.             }
67.         }
68.     }
69. }
```

[22] In diesem Beispiel repräsentiert ein Thread den Pfarrer, der darauf wartet, daß die Kollekte wenigstens 100$ beträgt. Weitere sechs Threads repräsentieren die Teilnehmer des Gottesdienstes, von denen jeder 20$ für die Kollekte spendet. Alle Threads sind bezüglich des von `collectionPlate` referenzierten `CollectionPlate`-Objektes synchronisiert (nach dem Fremdwörterbuch „zeitlich aufeinander abgestimmt"). Da die Threads, die eine Spende zur Kollekte geben, die Sperre dieses Objektes aufnehmen müssen, muß der Pfarrer-Thread diese Sperre hin und wieder vorrübergehend aufheben, indem er die `wait()`-Methode aufruft. Jeder Thread, der eine Spende zur Kollekte gibt, „weckt den Pfarrer auf", indem er die `notify()`-Methode aufruft.

Bemerkung: Es kommt in Zeile 41 im obigen Beispiel nicht darauf an, ob die Spender-Threads `notify()` oder `notifyAll()` aufrufen. Der Pfarrer-Thread wird in beiden Fällen aufgeweckt. Wir rufen `notify()` auf, da nur ein Thread auf die Änderung einer Bedingung (Summe der Spenden auf dem Teller) wartet. Warten mehrere Threads auf unterschiedliche Bedingungen, so ist es sinnvoll `notifyAll()` aufzurufen.

---

[23] Ein anderes Beispiel: Stellen Sie sich eine Erzeuger/Verbraucher-Beziehung vor, wobei ein Thread den Zustand eines Objektes setzt und ein anderer Thread das Objekt im bewerteten Zustand verwendet. Der Verbraucher ist darauf angewiesen, daß die entsprechende Änderung des Objektzustandes stattfindet, möchte also benachrichtigt werden, wenn die Änderung durchgeführt wurde. Die `Object`-Methoden `wait()`, `notify()` und `notifyAll()` dienen dieser Art des Wartens.

[24] Ein Aufruf der `wait()`-Methode wirkt anders als ein Aufruf der Methoden `sleep()` oder `yield()` oder das Warten auf eine Ein-/Ausgabeoperation. Die Methode `wait()` gestattet anderen Threads, „ihr" Objekt zu sperren. Dies gilt *nicht* für die Methoden `sleep()` und `yield()` sowie beim Warten auf eine Ein-/Ausgabeoperation.

Bemerkung: Die Methoden `wait()` und `notify()`/`notifyAll()` treten fast immer gemeinsam auf. Ruft ein Thread `wait()` auf, so gibt es in der Regel einen anderen Thread, der `notify()` oder `notifyAll()` aufruft. Eine Ausnahme von dieser Regel kommt in sehr seltenen Situationen vor, wenn Sie mit Zeitüberschreitungen arbeiten und die Sperrung eines Objektes aufheben müssen. Wird die `wait()`-Methode nicht aufgerufen, so gibt es auch keinen Grund `notify()` oder `notifyAll()` aufzurufen.

4.1.3.2 Die statische Thread-Methode yield()

[25] Ein Thread kann durch Aufrufen der statischen `Thread`-Methode `yield()` von der Nutzung seines Kontingentes an Rechenzeit zurücktreten, um einem anderen Thread den Vortritt zu lassen, dabei aber die Sperren seiner Resourcen aufrechterhalten. Stellen Sie zum Beispiel vor, Sie stehen vor dem Bankautomaten (*automated teller machine, ATM*) und wissen, daß Sie ihn für einige Zeit beanspruchen werden, da Sie eine Überweisung durchführen, den Kontostand prüfen und eine Einzahlung vornehmen müssen. Ein Aufruf der `yield()`-Methode ist damit vergleichbar, daß Sie

(der in Ausführung befindliche Thread) der Person hinter Ihnen anbieten, den Automaten zuerst zu benutzen, wobei Sie dieser Person selbstverständlich nicht Ihre Resourcen geben (etwa Geld oder Bankkarte). Sie bieten aber Ihren Platz am Bankautomaten (analog zum Prozessor) an. Es ist möglich, daß dieses Angebot nicht genutzt wird.

[26] Es ist möglich, daß der Thread, der die `yield()`-Methode aufgerufen hat, unmittelbar danach wieder zur Ausführung gebracht wird. Auf das Bankautomatenbeispiel bezogen, kann die Person, die hinter Ihnen in der Schlange steht, zu höflich sein, um Ihr Angebot anzunehmen. Für die Threads bedeutet ein Aufruf der `yield()`-Methode, daß nun alle Threads (auch derjenige, der `yield()` aufgerufen hat) um die Sperre des Objektes wetteifern, wobei Ihr Thread keine schlechtere oder bessere Aussicht hat, das Objekt zu sperren, als die anderen. Letztendlich entscheidet der Threadscheduler, welcher Thread „gewinnt“ und das Objekt sperrt.

[27] Der Threadscheduler ist eine Art Büroleiter. Er überwacht die Threads und entscheidet unter Berücksichtigung des unterliegenden Betriebsystem, der Threadprioritäten und weiterer Faktoren, welcher Thread verarbeitet wird.

[28/29] Zwei wichtige Bemerkungen zum Aufruf der `yield()`-Methode. Erstens: Es besteht keine Garantie dafür, daß der Thread, der die `yield()`-Methode aufgerufen hat, als nächster Thread in der Reihenfolge ausgewählt und zur Ausführung gebracht wird, obwohl er freiwillig von seinem Kontigent an Rechenzeit zurückgetreten ist. Nach einem Aufruf der `yield()`-Methode und nach dem ein anderer Thread zur Verarbeitung ausgewählt wurde, hat der ursprüngliche Thread keinen Sonderstatus unter den anderen Threads, sondern muß warten, wie alle übrigen Threads. Zweitens: Der Thread, der die `yield()`-Methode aufgerufen hat, erhält alle Objektsperren aufrecht. Selbst wenn Sie am Bankautomaten jemand anderem den Vortritt lassen, geben Sie ihm nicht Ihre Bankkarte. Ein Thread nimmt nach dem Aufruf der `yield()`-Methode keine Rechenzeit in Anspruch, behält aber die Kontrolle über alle Resourcen, die er ursprünglich zur exklusiven Verwendung gesperrt hatte.

[30] Unter welchen Umständen ist ein Aufruf der `yield()`-Methode sinnvoll? Stellen Sie sich vor, daß Ihr Thread sechs kleinere Banktransaktionen abarbeitet: Kontostand prüfen, Geld von Ihrem einen Girokonto auf Ihr zweites Girokonto überweisen, Kontostand prüfen, Geld vom Sparkonto auf das zweite Girokonto überweisen, Kontostand prüfen und Bargeld abheben. Eventuell überweist der Thread, dem Sie Vortritt gewähren, das Geld, welches er Ihnen schuldet auf Ihr zweites Girokonto, so daß Ihre eigene Überweisung nicht benötigt wird und Sie unter Umständen nicht fortfahren müssen. Hätte Ihr Thread den anderen Thread nicht vorgelassen, so wäre dessen Überweisung erst nach Abschluß Ihrer sechs Transaktionen eingegangen. Führt ein Thread eine langwierige oder teuere Operation aus, so ist es sinnvoll, hin und wieder per `yield()` einem anderen Thread den Vortritt zu gewähren.

Warnung: Verlassen Sie sich bei der Steuerung Ihrer Threads weder auf Threadprioritäten noch auf ausgiebigen Gebrauch der `yield()`-Methode. Die Implementierung des Threadschedulers, der auch die `yield()`-Methode interpretiert, liegt vollkommen in den Händen des Anbieters der Laufzeitumgebung. Es ist möglich, daß sich verschiedene Betriebssysteme oder sogar zwei verschiedene Laufzeitumgebungen unter ein und demselben Betriebssystem hinsichtlich der Wirkung der `yield()`-Methode völlig verschieden verhalten. Verwenden Sie `yield()` in Ihren Programmen im Bewußtsein, sich nicht darauf verlassen zu können, daß die Wirkung tatsächlich eintritt.

4.1.3.3 Threads im blockierten Zustand, Teil 1

[31] Ist die Verarbeitung eines Threads unterbrochen bis dieser ein benötigtes Objekt sperren kann, so befindet sich der Thread im blockierten Zustand. Versucht ein Thread ein Objekt zu sperren, das bereits von einem anderen Thread gesperrt wurde, so wird der erstere Thread solange blockiert, bis der andere Thread die Sperre aufhebt. Der blockierte Thread befindet sich in einer Art „Winterschlaf“ (*hibernation*), bis das Ereignis „Sperre verfügbar“ eintritt.

[32] Welche Auswirkungen hat das Blockieren eines Threads auf die übrigen Threads? Die übrigen Threads können die Rechenzeit des blockierten Threads in Anspruch nehmen, die dieser nicht benötigt. Der blockierte Thread behält allerdings die alleinige Kontrolle über alle zuvor gesperrten Resourcen. Die übrigen Threads sollten darauf vorbereitet sein, daß der blockierte Thread weiterverarbeitet wird, da das auslösende Ereignis jederzeit eintreten kann.

Bemerkung: Die Laufzeitumgebung ist dafür zuständig, Threads vom Zustand `Thread.State.BLOCKED` in den Zustand `Thread.State.RUNNABLE` zu überführen. Sie brauchen den anderen Threads nicht programmatisch mitzuteilen, daß Sie dabei sind, die Sperrung eines Objektes aufzuheben.

4.1.3.4 Die statische Thread-Methode sleep()

[33] Ein Thread „schläft“, wenn er wenigstens für eine gegebene Zeitdauer im Wartezustand verbleibt. Stellen Sie sich vor, der Bankautomat zeigt eine Meldung an, daß sich das System gerade im Wartungsmodus befindet. Sie könnten sich entscheiden, für fünf Minuten durch die Schalterhalle zu wandern und es anschließend erneut zu versuchen. Fünf Minuten später gehen Sie wieder zum Bankautomaten und stellen sich in der Warteschlange an.

Bemerkung: Ein schlafender Thread wartet wenigstens so lange, wie beim Aufruf der `sleep()`-Methode festgelegt wurde. Die tatsächliche Dauer des Wartezustandes liegt im Ermessen des Threadschedulers.

---

[34] Der Unterschied zwischen einem schlafenden Thread (`sleep()`) und einem Thread, der einem anderen Thread den Vortritt gelassen hat (`yield()`) besteht darin, daß der durch `yield()` zurückgesetzte Thread nicht „weiß“, wie kurz oder lang seine Wartezeit ausfallen wird. Ein schlafender Thread „weiß“ dagegen, daß er wenigstens so lange warten muß, als beim Aufruf der `sleep()`-Methode vereinbart.

4.1.3.5 Eisverkäuferbeispiel: Die Klasse Child

[35] Das folgende Eisverkäufer-Beispiel ist eventuell komplizierter als die Anforderungen, die Sie in Ihrer Prüfungsaufgabe erwarten, eignet sich aber hervorragend, um einen Thread im Wartezustand vorzuführen. Wenn Sie dieses Beispiel sorgfältig durcharbeiten, brauchen Sie die Anforderungen Ihrer Prüfungsaufgabe im Hinblick auf Threads nicht zu fürchten.

[36] Die ersten 41 Zeilen sind nicht schwierig zu verstehen. Die Klasse `IceCreamMan` ist von `Thread` abgeleitet (siehe Seite 74ff). Wir erzeugen ein `IceCreamMan`-Objekt, also einen neuen Thread, der

in einer `while`-Schleife auf Kinder wartet, die ihm `IceCreamDish`-Objekte zum füllen übergeben[1]. Das `IceCreamMan`-Objekt ist ein statisches Objekt, das heißt, daß es unabhängig von der Anzahl der Kinder stets nur einen Eisverkäufer gibt.

[37] In Zeile 15 charakterisieren wir den `IceCreamMan`-Thread mit Hilfe der Methode `setDaemon()` als sogenannten „Hintergrundthread“ (*daemon thread*). Die Laufzeitumgebung wird beendet, wenn nur noch Hintergrundthreads verarbeitet werden. Außerdem erzeugen wir explizit drei sogenannte „Vordergrundthreads“ (*user threads*), nämlich je einen für jedes der drei Kinder. Der `main`-Thread ist ebenfalls ein Vordergrundthread. Sind diese vier Threads beendet, dann ist der einzige explizit erzeugte Thread, der immer noch läuft, der `IceCreamMan`-Thread (Hintergrundthread) und die Laufzeitumgebung und damit auch das Programm werden beendet.

Bemerkung: Der Typ eines Threads (Vorder- oder Hintergrundthread) kann nach seinem Start nicht mehr geändert werden. Soll ein Thread als Hintergrundthread ablaufen, so müssen Sie ihn als solchen kennzeichnen, *bevor* Sie ihn starten. Ein von einem Vordergrundthread erzeugter Thread ist per Voreinstellung wiederum ein Vordergrundthread. Ebenso ist ein von einem Hintergrundthread erzeugter Thread per Voreinstellung wiederum ein Hintergrundthread. In beiden Fällen kann der voreingestellte Typ mit Hilfe der `setDaemon()`-Methode geändert werden.

---

[38] Da die Klasse `IceCreamMan` von `Thread` abgeleitet ist, können wir in Zeile 16 die `start()`-Methode direkt aufrufen.

[39] Zeile 25 zeigt ein Anwendungsbeispiel für die erweiterte `for`-Schleife zur Iteration über die Elemente eines Arrays. Pro Schleifendurchlauf wird ein `Child`-Objekt erzeugt, welches zugleich einen neuen Thread repräsentiert. Jeder `Child`-Thread hat die Aufgabe, beim Eisverkäufer ein Eis zu bestellen.

[40] In Zeile 27 wird ein `Thread`-Objekt erzeugt, das ein Objekt der Klasse `Child` enthält (`Child` implementiert das Interface ***`Runnable`***). Der neue Thread wird per `start()`-Methode in Zeile 28 gestartet, wie jeder andere Thread auch.

[41] In diesem Beispiel ist lediglich wichtig, daß jedes Kind sein Eis bekommt (was danach passiert ist unbedeutend). Daher beenden wir das Programm, nachdem alle drei Kinder ihr Eis aufgegessen haben. Der `main`-Thread muß solange warten, bis die drei `Child`-Threads vollständig verarbeitet worden sind. Dies wird dadurch erreicht, daß der `main`-Thread in Zeile 34 die `join()`-Methode jedes einzelnen `Child`-Threads aufruft.

Tipp: Wenn es Ihnen schwerfällt, die Wirkung der Methode `join()` zu verstehen, sehen Sie sich nochmals Abbildung 4-2 an. Seite 72 (Buch). Im Diagramm sieht es so aus, als ob sich jeder vom Hauptthread abzweigende Thread nach seiner Verarbeitung wieder mit dem Hauptthread vereinigt.

---

[42] Nachdem die Verarbeitung aller `Child`-Threads beendet ist, gibt der `main`-Thread eine Statusmeldung aus und wird beendet. Wie zuvor bereits beschrieben, sind an dieser Stelle keine Vordergrundthreads mehr aktiv und das Programm wird beendet.

Bemerkung: Die Anwendung würde sich ohne die `join()`-Methodenaufrufe in Zeile 34 sehr ähnlich verhalten. Der Hauptunterschied würde darin bestehen, daß der `main`-Thread vor den `Child`-Threads

[1] *Anmerkung des Übersetzers*: Üblicherweise betreten Kinder eine Eisdiele nicht mit einer leeren Eiswaffel in der Hand, um den Eisverkäufer zu bitten, sie zu füllen. Stellen Sie sich vor, daß jedes Kind von seiner Mutter eine große Keramikschlüssel bekommen hat.

beendet werden würde. Die `Child`-Threads würden dennoch ausgeführt werden, da sie keine Hintergrundthreads sind. Jedes Kind würde sein Eis bekommen. Nachdem alle Kinder ihr Eis bekommen haben, würde das Programm beendet werden.

```
01. /**
02.   * a Child object, designed to consume ice cream
03.   */
04. public class Child implements Runnable {
05.     private static IceCreamMan iceCreamMan = new IceCreamMan();
06.     private IceCreamDish myDish = new IceCreamDish();
07.     private String name;
08.
09.     public Child(String name) {
10.         this.name = name;
11.     }
12.
13.     public static void main(String args[]) {
14.         // start the ice cream man's thread.
15.         iceCreamMan.setDaemon(true);
16.         iceCreamMan.start();
17.
18.         String[] names = {"Ricardo", "Sally", "Maria"};
19.         Thread[] children = new Thread[names.length];
20.
21.         // create some child objects
22.         // create a thread for each child
23.         // get the Child threads started
24.         int counter = -1;
25.         for (String name : names) {
26.             Child child = new Child(name);
27.             children[++counter] = new Thread(child);
28.             children[counter].start();
29.         }
30.
31.         // wait until all children have eaten their ice cream
32.         for (Thread child : children) {
33.             try {
34.                 child.join();
35.             } catch (InterruptedException ie) {
36.                 ie.printStackTrace();
37.             }
38.         }
39.
40.         System.out.println("All children received ice cream");
41.     }
```

[43] Jedes Kind (`Child`-Thread) gibt dem Eisverkäufer in Zeile 44 die Schlüssel seiner Mutter (`IceCreamDish`-Objekt). Anschließend ißt das Kind sein Eis, wenn es seine Schlüssel zurückbekommt (`eatIceCream()`-Methode):

```
43.     public void run() {
44.         iceCreamMan.requestIceCream(myDish);
45.         eatIceCream();
46.     }
47.
```

Bemerkung: Die Formatierungsrichtlinie für Java-Quelltexte von Sun Microsystems empfiehlt eine Leerzeile zwischen je zwei aufeinander folgenden Methoden. Die Wiedergabe solcher Leerzeilen im Buch ist nutzlos, wenn ein Quelltext unterbrochen wird, um einen Absatz mit Erläuterungen einzufügen. Da die Leerzeilen aber in der Quelltext-Distribution, die Sie aus dem „Source Code"-Abschnitt der Apress-Website (*http://www.apress.com/book/sourcecode*) herunterladen können, vorhanden sind, kann es vorkommen, daß Leerzeilen und Dokumentationskommentare (Javadoc) im Buch nicht wiedergegeben sind, damit die Nummerierung der Zeilen zwischen Buch und Quelltext übereinstimmt. Sie können also sicher sein, daß alle relevanten Zeilen abgedruckt sind.

[44] Im Rahmen der Diskussion der Klasse `IceCreamMan` im folgenden Unterunterabschnitt wird sich zeigen, daß der Eisverkäufer (`IceCreamMan`-Thread) die erhaltene Schüssel (`IceCreamDish`-Objekt) füllt und anschließend anzeigt, daß die Eisportion fertig ist. Jedes Kind (`Child`-Thread) wartet solange, bis der Eisverkäufer seine Eisportion fertig gemacht und das Kind über diese Änderung benachrichtigt hat. Sobald diese Benachrichtigung eintrifft, ißt das Kind sein Eis.

[45] Beachten Sie, daß jeder `Child`-Thread die Sperrung seines `IceCreamDish`-Objektes aufhebt, indem es in Zeile 60 die Methode `myDish.wait()` aufruft. Dadurch hat der `IceCreamMan`-Thread die Gelegenheit, seinerseits das `IceCreamDish`-Objekt zu sperren und die Eisportion fertigzumachen.

Bemerkung: Sie können die `wait()`-Methode eines Objektes nur aufrufen, wenn Sie sich in einem Block oder einer Methode befinden, der beziehungsweise die bezüglich *dieses Objektes* synchronisiert ist.

[46] Die nächste interessante Stelle befindet sich in den Zeilen 48–69:

```
48.     private void eatIceCream() {
49.         String msg = name + " waiting for the IceCreamMan to fill dish";
50.         /*
51.          * The IceCreamMan will notify us when the dish is full, so we should
52.          * wait until we have received that notification. Otherwise we could
53.          * get a dish that is only half full (or even empty).
54.          */
55.         synchronized (myDish) {
56.             while (myDish.readyToEat == false) {
57.                 // wait for the ice cream man's attention
58.                 try {
59.                     System.out.println(msg);
60.                     myDish.wait();
61.                 }  catch (InterruptedException ie) {
62.                     ie.printStackTrace();
63.                 }
64.             }
65.             myDish.readyToEat = false;
66.         }
67.         System.out.println(name +": yum");
68.     }
69. }
70.
71. class IceCreamDish {
72.     public boolean readyToEat = false;
73 }
```

[47] Die Verarbeitung des `Child`-Threads wird in den Zeilen 55–66 bezüglich des von `myDish` re-

ferenzierten `IceCreamDish`-Objektes synchronisiert. Das bedeutet, daß die Ausführung des `Child`-Threads nicht über Zeile 55 hinaus geht, wenn der Thread keinen exklusiven Zugriff auf sein `IceCreamDish`-Objekt hat.

[48] Denken Sie daran, daß sich ein `Child`-Thread den Zugriff auf sein `IceCreamDish`-Objekt mit dem `IceCreamMan`-Thread teilt. Der `IceCreamMan`-Thread kann den Zustand des `IceCreamDish`-Objektes jederzeit ändern. Wir wollen aber verhindern, daß ein `Child`-Thread sein `IceCreamDish`-Objekt „ißt", bevor der `IceCreamMan`-Thread damit fertig ist.

[49] Wir nehmen nun an, daß der `Child`-Thread Zugriff auf sein `IceCreamDish`-Objekt erhalten hat. Dies kann aus zwei Gründen geschehen. Erstens: Der `IceCreamMan`-Thread verwendet dieses `IceCreamDish`-Objekt im Augenblick nicht (der Eisverkäufer hat schließlich noch weitere Kinder zu bedienen). Zweitens: Der `IceCreamMan`-Thread hat die Verarbeitung dieses `IceCreamDish`-Objektes beendet. Zeile 56 fragt der `Child`-Thread den Wert des Feldes `myDish.readyToEat` ab. Dieser Wert gibt dem `Child`-Thread an, ob das Eis gegessen werden kann. Das `readyToEat`-Feld funktioniert als sogenannter Monitor zwischen einem `Child`-Thread und dem `IceCreamMan`-Thread. Der Begriff „Monitor" wird in Kürze erklärt. Stellen Sie sich unter einem Monitor fürs erste einen Benachrichtigungsmechanismus zwischen einem `Child`-Thread und dem `IceCreamMan`-Thread vor.

[50] Enthält das Feld `myDish.readyToEat` noch den Wert `false`, so hat der Eisverkäufer die Eisportion noch nicht fertig gemacht. Der `Child`-Thread kann die Sperrung seines `IceCreamDish`-Objektes aufheben, indem er dessen `wait()`-Methode aufruft (Zeile 60).

[51] Beachten Sie, daß der `Child`-Thread das `readyToEat`-Feld *nicht* mittels `if`, sondern per `while`-Schleife abfragt. Der Grund hierfür besteht in einem Effekt, der sich ergeben kann, wenn die Laufzeitumgebung nach einem `wait()`-Aufruf die Kontrolle an die Anwendung zurückgibt. Die API-Dokumentation der `Object`-Methode `wait()` sagt aus: „A thread can also wake up without being notified, interrupted, or timing out, a so-called *spurious wakeup*. While this will rarely occur in practice, applications must guard against it by testing for the condition that should have caused the thread to be awakened, and continuing to wait if the condition is not satisfied. In other words, waits should always occur in loops." Übersetzt etwa: „Ein Thread kann aus seinem Wartezustand zurückkehren, ohne zuvor benachrichtigt oder unterbrochen worden zu sein oder die für seinen Wartezustand festgesetzte Mindestzeitdauer überschritten zu haben. Dieser Effekt wird als „Spurious Wakeup" bezeichnet. Obwohl dieser Effekt in der Praxis selten eintritt, müssen Anwendung entsprechend vorbereitet werden, indem sie die Bedingung prüfen, die das Erwachen des Threads auslöst und den Wartezustand fortsetzen, wenn die Bedingung nicht erfüllt ist, das heißt *`wait()`-Aufrufe sollten stets in Schleifen stehen*."

4.1.3.6 Eisverkäuferbeispiel: Die Klasse IceCreamMan

[52] Die `run()`-Methode des `IceCreamMan`-Threads prüft in einer `while`-Schleife, ob `IceCreamDish`-Objekte vorhanden sind, die verarbeitet werden müssen (Zeilen 15–32):

```
01. import java.util.* ; 02.
03. public class IceCreamMan extends Thread {
04.     /**
05.      * a list to hold all the children's IceCreamDish objects
06.      */
07.     private List<IceCreamDish> dishes = new ArrayList<IceCreamDish>();
08.
09.     /**
10.      * Start a thread that waits for ice cream bowls to be given to it.
11.      */
12.     public void run() {
```

```
13.         String clientExists = "IceCreamMan: has a client";
14.         String clientDoesntExist = "IceCreamMan: does not have a client";
15.
16.         while (true) {
17.             if (!dishes.isEmpty()) {
18.                 System.out.println(clientExists);
19.                 serveIceCream();
20.             } else {
21.                 try {
22.                     System.out.println(clientDoesntExist);
23.                     // sleep so that children have a chance to add their
24.                     // dishes. See note in book about why it is not this
25.                     // is not a yield statement.
26.                     sleep(1000);
27.                 }  catch(InterruptedException ie) {
28.                     ie.printStackTrace();
29.                 }
30.             }
31.         }
32.     }
```

[53] Der Thread tritt in Zeile 16 in eine Endlosschleife ein. Beachten Sie, daß der `IceCreamMan`-Thread ein Hintergrundthread ist, das Programm also trotz der Endlosschleife beendet wird, nachdem alle `Child`-Threads vollständig verarbeitet worden sind.

[54] Zeile 17 prüft, ob es `IceCreamDish`-Objekte gibt, die auf ihre Verarbeitung warten. Falls keine `IceCreamDish`-Objekte vorhanden sind, „schläft“ der `IceCreamMan`-Thread für eine Sekunde und wiederholt anschließend die Prüfung. Befindet sich aber ein `IceCreamDish`-Objekt in der Warteschlange, so ruft der `IceCreamMan`-Thread die Methode `serveIceCream()` auf.

[55] In Zeile 26 hatten wir die Wahl zwischen den Methoden `yield()` und `sleep()` (siehe Kommentar in den Zeilen 24 und 25). Der `IceCreamMan`-Thread könnte sowohl durch Aufrufen der `yield()`-Methode einem anderen Thread den Vortritt lassen, als auch durch einen Aufruf der `sleep()`-Methode für eine Mindestzeitdauer in den Wartezustand zurückgesetzt werden. In beiden Fällen würden die `Child`-Threads verarbeitet werden. Bei der `sleep()`-Methode ist jedoch das Umschalten zu einem der `Child`-Threads wahrscheinlicher, da die Laufzeitumgebung „weiß“, daß der `IceCreamMan`-Thread für wenigstens eine Sekunde ausscheidet. Bei der `yield()`-Methode könnte dagegen einer der `Child`-Threads aber auch unmittelbar wiederum der `IceCreamMan`-Thread selbst zur Verarbeitung ausgewählt werden. Davon abgesehen wird die `yield()`-Methode in der Regel vor dem Aufruf oder während der Verarbeitung einer länglichen oder komplizierten Operation aufgerufen, um sich eventuell vorhandenen anderen Threads gegenüber „kollegial“ zu verhalten, wofür hier aber kein Bedarf ist. Falls es keine anderen Threads gibt, die verarbeitet werden müssen, übergibt die Laufzeitumgebung die Kontrolle sofort wieder an die `while`-Schleife, wodurch der `IceCreamMan`-Thread soviel Rechenzeit wie möglich beansprucht. Dadurch kann der Eindruck entstehen, der Rechner habe „sich aufgehängt“.

[56] Die Methode `serveIceCream()` ist der interessanteste Abschnitt der Klasse `IceCreamMan`. Befindet sich ein `IceCreamDish`-Objekt in der Warteschlange, so wird der `IceCreamMan`-Thread in Zeile 49 bezüglich dieses `IceCreamDish`-Objektes synchronisiert. Der `Child`-Thread zu dem das gerade gesperrte `IceCreamDish`-Objekt gehört, muß warten, bis der `IceCreamMan`-Thread die Sperre wieder aufhebt. Insbesondere veranlaßt diese Synchronisierung den `Child`-Thread in Zeile 55 (von *Child.java*) solange zu warten bis der `IceCreamMan`-Thread in Zeile 53 (von *IceCreamMan.java*) das Ende des synchronisierten Blocks erreicht hat. Wartet ein `Child`-Thread nach dem Aufrufen der `wait()`-Methode in Zeile 60 (von *Child.java*), so kann seine Verarbeitung erst dann fortgesetzt

werden, wenn der `IceCreamMan`-Thread Zeile 53 (von *IceCreamMan.java*) erreicht hat.

[57] Ruft, im umgekehrten Fall, ein `Child`-Thread seine `eatIceCream()`-Methode auf, die einen bezüglich seines `IceCreamDish`-Objektes synchronisierten Block enthält (Zeilen 55–66 in *Child.java*), so wird der `IceCreamMan`-Thread beim Erreichen seiner Zeile 49 (in *IceCreamMan.java*) veranlaßt, zu Warten, bis der `Child`-Thread seinen synchronisierten Block vollständig verarbeitet oder aber durch Aufrufen der `wait()`-Methode die Sperrung seines `IceCreamDish`-Objektes aufgehoben hat.

[58] Dies ist ein Beispiel dafür, daß Threads ihre gegenseitigen Grenzen respektieren. Durch die Synchronisierung der Threads bezüglich ein und desselben Objektes, nämlich eines `IceCreamDish`-Objektes, ist gewährleistet, daß die Threads nicht gleichzeitig dasselbe Objekt verwenden oder ändern.

[59] Hat der `IceCreamMan`-Thread ein `IceCreamDish`-Objekt gesperrt, so kann der Eisverkäufer die Eisportion fertig machen und anschließend den entsprechenden `Child`-Thread benachrichtigen (Zeile 52 in *IceCreamMan.java*), daß das Eis fertig ist. Hat der `IceCreamMan`-Thread Zeile 52 verarbeitet, so braucht der entsprechende `Child`-Thread nicht länger zu warten und kann vom Scheduler der Laufzeitumgebung zur Verarbeitung gewählt werden. Der `Child`-Thread hat allerdings keine Chance, sein `IceCreamDish`-Objekt zu sperren, bevor der `IceCreamMan`-Thread seine Sperrung in Zeile 53 aufhebt und bleibt solange blockiert.

Bemerkung: Die `notify()`-Methode eines Objektes kann nur in einem Block oder einer Methode aufgerufen werden, der beziehungsweise die bezüglich dieses Objektes synchronisiert ist.

```
34.     /**
35.      * Serve Ice Cream to a Child object.
36.      */
37.     private void serveIceCream() {
38.         // get an ice cream dish
39.         IceCreamDish currentDish = dishes.get(0);
40.
41.         // wait sometimes, don't wait sometimes
42.         if (Math.random() > .5) {
43.             delay();
44.         }
45.
46.         String msg = "notify client that the ice cream is ready";
47.         System.out.println("IceCreamMan: " + msg);
48.
49.         synchronized (currentDish) {
50.             currentDish.readyToEat = true;
51.             // notify the dish's owner that the dish is ready
52.             currentDish.notify();
53.         }
54.
55.         // remove the dish from the queue of dishes that need service
56.         dishes.remove(currentDish);
57.     }
```

[60] Für die synchronisierten Blöcke in den Zeilen 49–53 der `IceCreamMan`-Methode `serveIceCream()` beziehungsweise in den Zeilen 55–66 der `Child`-Methode `eatIceCream()` ist wesentlich, daß der `IceCreamMan`-Thread und die `Child`-Threads bezüglich desselben Objektes synchronisiert sind. Wir müssen dafür sorgen, daß ein fremder Entwickler beim Arbeiten an den Klassen `Child` und `IceCreamMan` die synchronisierten Blöcke beibehält. Allerdings besteht das Risiko, daß die Synchronisierung des Blocks in der Klasse `Child` entfernt wird, wodurch es möglich wird, daß ein

`Child`-Thread nach seinem `IceCreamDish`-Objekt „greift“, bevor die Eisportion fertig ist. Dadurch besteht aber eine Sicherung gegen Programmänderungen: Wir erklären dem fremden Entwickler, daß durch das Entfernen der Synchronisierung ein Kind eventuell nur eine halb fertige oder gar keine Eisportion bekommen könnte.

Tipp: Wenn die Funktionstüchtigkeit eines Quelltextabschnittes davon abhängt, daß diese auf eine bestimmte Art und Weise implementiert ist, sollten Sie einen Implementierungskommentar anlegen, um fremde Entwickler über diese Tatsache und eventuelle Einzelheiten zu informieren, siehe beispielsweise Zeilen 50–54 in *Child.java*, Seite 77.

---

[61] Es ist vergleichsweise einfach, einem Kind zu erklären, daß es nicht nach seiner Eisportion greifen soll, bevor es gesagt bekommt, daß das Eis fertig ist (weil es nicht zu kurz kommen will). Es dagegen schwieriger, die Kinder davon zu überzeugen, daß sie nicht alle zugleich dem Eisverkäufer ihre Schüsseln geben sollen (aus der Perspektive eines Kindes wird die Schüssel umso früher gefüllt, je eher es sie abgibt). Ein Kind kümmert sich nicht um Fairness gegenüber den anderen Kindern oder darum, wie der Eisverkäufer mit mehreren Schüsseln zugleich zurecht kommt.

[62] Aus diesem Grund haben wir die `IceCreamMan`-Methode `requestIceCream()` bezüglich des statischen `IceCreamMan`-Objektes synchronisiert. Dadurch wird gewährleistet, daß stets nur ein Kind dem Eisverkäufer seine Schüssel gibt:

```
59.     /**
60.      * Allow client objects to add dishes
61.      */
62.     public synchronized void requestIceCream(IceCreamDish dish) {
63.             dishes.add(dish);
64.     }
65.
66.     /**
67.      * build in a delay
68.      */
69.     private void delay() {
70.         try {
71.             System.out.println("IceCreamMan: delayed");
72.             Thread.sleep((long) (Math.random()* 1000) );
73.         } catch (InterruptedException ie) {
74.             ie.printStackTrace();
75.         }
76.     }
77. }
```

[63] Die Bildschirmausgaben eines Programmaufrufs lauten:

```
~$ java Child

IceCreamMan: does not have a client
Ricardo waiting for the IceCreamMan to fill dish
Sally waiting for the IceCreamMan to fill dish
Maria waiting for the IceCreamMan to fill dish
IceCreamMan: has a client
IceCreamMan: delayed
IceCreamMan: notify client that the ice cream is ready
Ricardo: yum
IceCreamMan: has a client
IceCreamMan: notify client that the ice cream is ready
```

```
Sally: yum
IceCreamMan: has a client
IceCreamMan: notify client that the ice cream is ready
Maria: yum
All children received ice cream
IceCreamMan: does not have a client
```

Der Eisverkäufer (`IceCreamMan`-Thread) wartet auf Kunden. Jedes Kind (`Child`-Objekt) ist zugleich ein Thread. Ein `Child`-Thread interagiert mit dem `IceCreamMan`-Thread, indem er eine Referenz auf sein `IceCreamDish`-Objekt übergibt und anschließend in den Wartezustand übergeht. Der `IceCreamMan`-Thread macht das `IceCreamDish`-Objekt fertig und benachrichtigt den zugehörigen `Child`-Thread, um zu erwachen und das Eis zu essen.

4.1.3.7 Zusammenfassung: Threads im Wartezustand

[64] Die Verarbeitung eines Threads kann beim Zugriff auf eine Resource ohne eine vom Entwickler erteilte Anweisung unterbrochen werden. Im Gegensatz dazu kann die Verarbeitung eines Threads durch Aufrufen der Methoden `yield()`, `wait()` oder `sleep()` *programmatisch* unterbrochen werden. In beiden Fällen geht der Thread in einen Wartezustand über.

[65] Wartet ein Thread auf eine nicht verfügbare Resource oder läßt ein Thread per `yield()` einem anderen Thread den Vortritt, so wissen Sie nicht, wann die Verarbeitung dieses Threads fortgesetzt wird. Im ersten Fall wird die Verarbeitung fortgesetzt, nachdem die Resource wieder verfügbar geworden ist, im zweiten Fall nach Ermessen des Threadschedulers der Laufzeitumgebung. Beim Aufruf der `wait()`-Methode ohne Parameter wissen Sie nicht, wann die Verarbeitung des wartenden Threads wieder aufgenommen wird, da die Fortsetzung der Verarbeitung von der Benachrichtung mittels `notify()` oder `notifyAll()` durch einen anderen Thread abhängt. Wenn Sie die `wait()`-Methode mit der Angabe eines Zeitraumes in Millisekunden aufrufen, dann wissen Sie, daß die Verarbeitung des Threads entweder bei Benachrichtigung oder kurz nach Ablauf der Wartezeit fortgesetzt wird. Beachten Sie, daß die exakte Einhaltung derartiger Zeiträume nicht erzwungen werden kann. Die Laufzeitumgebung unterbricht die Verarbeitung des Threads mindestens so lange, wie gefordert und versucht die Verarbeitung nach Ablauf der Wartezeit fortzusetzen, aber ein exakter Zeitpunkt für die Wiederaufnahme der Verarbeitung kann nicht garantiert werden. Beim Aufruf der `sleep()`-Methode wissen Sie, daß die Verarbeitung des Threads kurz nach Ablauf der vereinbarten Wartezeit wieder aufgenommen wird. Auch hier kann die Fortsetzung der Threadverarbeitung nicht an einem bestimmten Punkt nach Verstreichen des geforderten Zeitraumes erzwungen werden. Stellen Sie sich im Eisverkäuferbeispiel vor, daß Sally den Eisverkäufer zwar auf sich aufmerksam gemacht und ihm Geld gegeben, sich aber noch nicht für ihre Eissorten entschieden hat. Würde Sally (per `yield()`) freiwillig einem anderen Kind Vortritt gewähren, so könnte sie sicher sein, daß der Eisverkäufer ihre Eisportion nicht verkauft. Wenn kein anderes Kind mehr wartet, tritt Sally wieder vor und ihre Eisportion wird fertig gemacht.

[66] Den Sally-Thread „schlafen zu schicken“ (per `sleep()`) ist, als ob sie das Geld für ein Eis bereits bezahlt hat, die Bestellung aber um eine Minute hinauszögert, etwa um auf ihren kleinen Bruder zu warten. Nach Verstreichen der Minute versucht sie es erneut (unabhängig davon, ob der kleine Bruder eingetroffen ist oder nicht). Wenn Sally schließlich die Aufmerksamkeit des Eisverkäufers für sich hat, kann sie sich darauf verlassen, daß die reservierte Schüssel noch da ist. Die Tatsache, daß die Minute verstrichen ist, bedeutet allerdings nicht, daß Sally ihr Eis sofort bekommt, da möglicherweise gerade ein anderes Kind sein Eis erhält. Der Sally-Thread wird blockiert, wenn sie den Eisverkäufer nicht auf sich aufmerksam machen kann. Nach einer Minute Wartezeit des Sally-Threads, bedient der Eisverkäufer gerade ein anderes Kind. Der Sally-Thread besitzt zwar die Sperre der Eisportion, nicht aber die Sperre des Eisverkäufers und wird solange blockiert, bis sie dessen

Sperre aufnehmen kann (eventuell taucht auch Sallys Bruder in der Zwischenzeit auf).

[67] Hätte Sally aber auf ihren Vater warten müssen, der das Geld mitbringt, so hätte sie keine Eiswaffel reservieren können und während der Wartezeit hätten alle Waffeln verkauft sein können.

4.1.4 Der Sperrmechanismus eines Objektes

[68] Jedes Java-Objekt besitzt eine „Sperre“. Diese Sperre ist ein „abstrakter Gegenstand“ der stets nur von einem einzigen Thread verwendet werden kann. Sie können sich die Sperre eines Objektes wie eine Markierung vorstellen, durch die ein Thread den anderen Threads mitteilt, daß der Zustand dieses Objektes nicht verändert werden darf, bevor der erstere Thread die Sperrung wieder aufhebt (vorausgesetzt, daß die anderen Threads die Sperre respektieren). Wir sagen, daß ein Thread die Synchronisierung eines anderen Threads respektiert, wenn beide Threads bezüglich desselben Objekts synchronisiert (nach dem Fremdwörterbuch also „zeitlich aufeinander abgestimmt“) sind.

[69] Bei Java können Sie ein Objekt sperren, indem Sie eine Methode oder einen Block von Anweisungen bezüglich dieses Objektes synchronisieren, zum Beispiel:

```
public void addElement(Object item) {
    synchronized (myArrayList) {
        // do stuff
    }
}
```

[70] Die Laufzeitumgebung erkennt daran, daß kein anderer Thread den Zustand des von `myArrayList` referenzierten Objektes verändern darf. Diese Sicherung gilt nur für synchronisierte Methoden. Die Methode `elementExists()` im folgenden Beispiel muß eventuell nicht synchronisiert werden, da sie nur eine lesende Operation ausführt:

```
public boolean elementExists(Object item) {
    return myArrayList.contains(item);
}
```

[71] Ist dagegen erforderlich, daß `elementExists()` stets absolut genau funktionieren muß (wenn Sie sich also keinen Fehler leisten können, der dadurch verursacht wird, daß ein anderer Thread zwischenzeitlich Elemente zur abgefragten Kollektion hinzugefügt oder gelöscht hat), dann müssen Sie auch den Zugriff durch die Methode `elementExists()` synchronisieren.

[72] Dieser Abschnitt faßt einige der bereits angesprochenen Konzepte zusammen. Das Konzept des Sperrens von Objekten ist grundlegend für das Verständnis von Threads und Sie werden sehr davon profitieren, diesen Abschnitt sorgfältig durchzuarbeiten.

4.1.4.1 Sperren von Objekten

[73] Das Sperren von Objekten kommt im Alltag häufig vor. Im Kino „sperren“ Sie beispielsweise ihren Platz, indem Sie Ihre Jacke liegen lassen. Selbst wenn Sie Ihren Platz verlassen, um Popcorn zu holen, ist klar, daß sich niemand auf diesen Platz setzen soll. Das Liegenlassen der Jacke bedeutet, daß Sie den Sitz für sich alleine beanspruchen, bis Sie ihn wieder freigeben.

[74] Jedes Java-Objekt kann von einem Thread zur exklusiven Verwendung beansprucht werden. Dies ist ein expliziter Mechanismus, durch den Java Multithreading unterstützt. Weitere Mechanismen sind die Fähigkeit von Objekten, ihren Verfügbarkeitsstatus bekannt zu geben und die Eigenschaft,

Threads, die auf eine gesperrte Resource warten, in den Wartezustand zu versetzen, bis die Resource verfügbar wird.

[75] Ist ein Objekt nicht ausdrücklich gesperrt, so kann es von jedem Thread verwendet werden, ebenso wie ein nicht explizit gekennzeichneter Platz im Kino jederzeit besetzt werden kann. Wenn Sie Ihren Platz verlassen, um eine Erfrischung zu kaufen, den Platz aber nicht markieren, indem Sie Ihre Jacke zurücklassen, sollten Sie nicht überrascht sein, wenn Sie zurückkommen und feststellen, daß jemand anders ihren Platz besetzt hat. Das Verlassen Ihres ungekennzeichneten Platzes im Kino ist nicht sicher, auch wenn es bisher stets funktioniert hat. Dasselbe gilt für Java. Falls eine Ihrer Klassen nicht explizit threadsicher ist, dürfen Sie aus der Tatsache, daß die Objekte dieser Klasse in der Vergangenheit tadellos funktioniert haben *nicht* schließen, daß die Fehlerfreiheit auch in der Zukunft gewährleistet ist.

[76] Der anschauliche Vergleich mit dem Platz im Kino verdeutlicht noch einen Aspekt von Threads. Es ist nämlich möglich, daß jemand Ihren Platz beansprucht, *obwohl* Sie ihn mit Ihrer Jacke gekennzeichnet haben. Wiederum ist dasselbe auch bei Java möglich. Der Zustand eines gesperrten Objektes kann verletzt werden, wenn der darauf zugreifende Thread die Synchronisierung nicht respektiert.

[77] Die folgende Methode `goodMethod()` ist beispielsweise bezüglich des von `myObject` referenzierten Objektes synchronisiert:

```
public void goodMethod() {
    synchronized (myObject) {
        // do stuff to myObject
    }
}
```

[78] Jede Methode, die nicht bezüglich `myObject` synchronisiert ist, kann den Zustand des von `myObject` referenzierten Objektes ändern. Eine Methode bezüglich eines Objektes zu synchronisieren bedeutet, daß die Methode die Sperrung dieses Objektes durch andere Threads respektiert und „hofft“, daß die anderen Threads die Sperrung ihres Objektes ebenfalls respektieren. Die zweite Satzhälfte gilt nicht zwangsläufig. Die folgende Methode `badMethod()` respektiert die Synchronisierung von `goodMethod()` nicht und läßt sich ohne Schwierigkeiten ausführen. Probleme sind dann möglich, wenn `badMethod()` den Zustand des von `myObject` referenzierten Objektes ändert, während ein anderer bezüglich `myObject` synchronisierter Thread „glaubt“, exklusiven Zugriff auf das gesperrte Objekt zu haben:

```
public boolean badMethod() {
    // do stuff to myObject
}
```

[79] Die Methodennamen im obigen Beispiel bedeuten selbstverständlich nicht, daß die eine Lösung stets gut, die andere Lösung dagegen stets schlecht ist. Fragt `badMethod()` beispielsweise nur eine Eigenschaft des von `myObject` referenzierten Objektes ab, kann es unter Umständen völlig ausreichen, auf die Synchronisierung bezüglich `myObject` zu verzichten. In der Regel hängen „gut“ und „schlecht“ vom Kontext ab.

4.1.4.2 Synchronisierung bezüglich des Klassenobjektes

[80] Java gestattet das Sperren von Klassen ebenso wie das Sperren von Objekten. Der Gedanke mag überraschen, aber das Konzept ist unkompliziert. Eine Klasse ist selbst ein Objekt, welches dazu verwendet wird, weitere Objekte zu erzeugen (auch eine Axt ist ein „Objekt“, um Feuerholz zu „erzeugen“).

[81] Beim erstmaligen Laden eines Programms erzeugt die Laufzeitumgebung für jede im Programm verwendete Klasse ein `Class`-Objekt (Klassenobjekt). Pro Laufzeitumgebung existiert für alle Objekte einer Klasse nur ein einziges Klassenobjekt, auf das Sie eingeschränkten Zugriff haben. Da das Klassenobjekt wiederum ein Objekt ist, können Sie es wie jedes andere Objekt sperren. Das Klassenobjekt ist durch seine Eigenschaft interessant, statische Felder und statische Methoden zu enthalten.

[82] Ein statisches Feld steht jedem Objekt der zugehörigen Klasse zur Verfügung, das heißt es gibt ein für alle Objekte dieser Klasse gemeinsames Feld. Beispielsweise könnte eine Klasse namens `McBurgerPlace` ein statisches Feld namens `chiefExecutiveOfficer` besitzen. Gleichgültig in welchem Restaurant (`McBurgerPlace`-Objekt) Sie essen, es gibt nur einen Bereichsleiter (`chiefExecutiveOfficer`), dem alle Filialen untergeordnet sind. Ändert eines der `McBurgerPlace`-Objekte den Wert von `chiefExecutiveOfficer`, so gilt die Änderung unmittelbar auch für alle anderen `McBurgerPlace`-Objekte.

[83] Auch eine statische Methode steht jedem Objekt der zugehörigen Klasse zur Verfügung, das heißt alle Objekte dieser Klasse teilen sich diese nur einmal pro Klasse und Laufzeitumgebung definierte Methode. Die Klasse `McBurgerPlace` könnte zum Beispiel eine statische Methode namens `createFranchise()` besitzen, vorausgesetzt, daß jedes einzelne Restaurant befugt ist, eine neue Lizenz zu erteilen. Jedes `McBurgerPlace`-Objekt kann einen neuen Lizenznehmer zulassen, indem es (per `createFranchise()`) die Firmenleitung informiert. Unabhängig vom spezifischen `McBurgerPlace`-Objekt wird stets ein und dieselbe `createFranchise()`-Methode aufgerufen.

[84] Statische Methoden und Objektmethoden unterscheiden sich in einer wesentlichen Eigenschaft: Statische Methoden gehören zur Klasse, nicht zu den einzelnen Objekten. Der „Beweis“ besteht in der Tatsache, daß Sie einen Lizenznehmer auch dann zulassen können, wenn es noch kein Restaurant gibt, indem Sie direkt mit der Firmenleitung in Verbindung treten.

[85] Wie hängt dies alles mit dem Sperren von Objekten zusammen? Stellen Sie sich vor, daß Sie eine einzelne `McBurgerPlace`-Filiale sperren müssen, während eine bestimmte Methode, zum Beispiel `waxFloor()`, ausgeführt wird, da Sie während des Fußbodenwachsens keine andere Tätigkeit im Restaurant erlauben dürfen.

[86] Wie läßt sich durchsetzen, daß ausschließlich die `waxFloor()`-Methode ausgeführt wird? In der Realität verbieten Sie, daß in dem entsprechenden Restaurant andere Tätigkeiten ausgeübt werden, während der Fußboden gewachst wird: Es dürfen keine Burger gebraten, keine Pommes Frites verkauft und keine Mahlzeiten ausgegeben werden. Alle Arbeitsschritte warten, bis der Fußboden gewachst ist. In einem Java-Programm bewerkstelligen Sie derartige Ausschließlichkeit, indem Sie die fragliche Methode mit Hilfe des Schlüsselwortes `synchronized` als synchronisierte Methode deklarieren:

```
public synchronized void waxFloor()
```

[87] Jede andere synchronisierte Methode dieses `McBurgerPlace`-Objektes wartet nun, bis der Fußboden gewachst ist.

Bemerkung: Unsynchronisierte Methoden respektieren synchronisierte Methoden *nicht*. Somit kann eine unsynchronisierte Methode jederzeit ausgeführt werden. Es ist wichtig, darauf zu achten, daß eine unsynchronisierte Methode den Zustand des Objektes nicht in Mitleidenschaft ziehen kann. Beispielweise kann `countMoney()` vermutlich unsynchronisiert gelassen werden, da sich ihre Aufgabe nicht mit `waxFloor()` überschneidet. Die Methode `wipeDownTables()` sollte dagegen synchronisiert werden, weil diese Tätigkeit nicht zugleich mit dem Wachsen des Fußbodens ausgeübt werden darf.

---

[88] Derartige Ausschließlichkeit hat nichts mit den anderen Restaurants (`McBurgerPlace`-Objekten) zu tun. In einem anderen Restaurant können Burger gebraten und Pommes Frites verkauft werden, auch wenn in der ersteren Filiale gerade der Fußboden gewachst wird.

[89] Wie bilden Sie eine Situation ab, die auch die übrigen Filialen betrifft, beispielsweise ein Besuch des Bereichsleiters der Restaurantkette? Natürlich kann der Bereichsleiter nur eine Filiale auf einmal besuchen, auch der Chef kann nicht an zwei Orten zugleich sein. Wie läßt sich ausschließlicher Zugriff in dieser Form in Java realisieren? Um bei unserem Beispiel zu bleiben, müssen Sie verhindern, daß der Bereichsleiter von einem Restaurantleiter unterbrochen wird, während er sich gerade mit einem anderen Restaurantleiter unterhält.

[90] Die Lösung dieses Problems besteht aus zwei Schritten. Erstens muß das Feld `chiefExecutiveOfficer` ein statisches Feld sein, also nur auf Klassenebene existieren. Sie deklarieren ein statisches Feld durch Verwendung des Schlüsselwortes `static`:

```
private static Object ceo = new Object();
```

[91] Zweitens müssen Sie die statische Methode synchronisieren, die auf das Feld `chiefExecutiveOfficer` zugreift:

```
public static synchronized Object receiveCeo() {
    return ceo;
}
```

[92] Bei einer statischen Methode bezieht sich die Synchronisierung auf die Klassenebene statt der Objektebene.

4.1.4.3 Synchronisierung bezüglich der this-Referenz

[93] Es gibt noch eine feiner einstellbare Möglichkeit, um Objekte (sowohl Klassenobjekte als auch „gewöhnliche“ Objekte) zu sperren: Sie können einen Block von Anweisungen bezüglich eines Objektes Ihrer Wahl synchronisieren. Stellen Sie sich in unserem Restaurantbeispiel vor, daß die Klasse `McBurgerPlace` eine Methode namens `getSoda()` besitzt, die stets alleinigen Zugriff auf das vom Feld `sodaFountain` referenzierte Objekt benötigt. Dann können Sie das ganze `McBurgerPlace`-Objekt sperren oder aber nur das von `sodaFountain` referenzierte Objekt:

```
public void getSodaEfficiently() {
    synchronized (sodaFountain) {
        // do stuff
    }
}
```

[94] Das Sperren des von `sodaFountain` referenzierten Objektes bewirkt, daß nur diejenigen Kunden warten müssen, die dieses Objekt benötigen, während das Sperren des `McBurgerPlace`-Objektes bewirkt, daß *alle* Kunden im Restaurant warten müssen (auch die Kunden, die nur einen Burger haben wollen).

[95] Das Synchronisieren des Zugriffs auf eine Methode ist eine Form der Sperrung von Objekten, genauer, des von `this` referenzierten Objektes. Das Synchronisieren einer Methode ist eigentlich nur eine abgekürzte Schreibweise für das Synchronisieren eines Blocks von Anweisungen bezüglich des von `this` referenzierten Objektes. Gleichbedeutend sind:

```
// Synchronisierung einer Methode

public synchronized void myMethod() {
    // code
}
```

und

```
// Synchronisierung eines Blocks von Anweisungen bezüglich der this-Referenz
public void myMethod() {
    synchronized (this) {
        // code
    }
}
```

[96] Die Synchronisierung der Verarbeitung eines Blocks von Anweisungen bezüglich eines anderen als des von `this` referenzierten Objektes kann die Anzahl gleichzeitiger Zugriffe erhöhen, da bezüglich verschiedener Objekte synchronisierte Blöcke von Anweisungen gleichzeitig verarbeitet werden können. Die Interpretation des Begriffs „Effizienz“ hängt allerdings vom Betrachter ab. Muß ein Thread zum Verarbeiten eines synchronierten Blocks mehrere Objekte sperren, so kann das Sperren und Entsperren der einzelnen Objekte aufwendiger sein, als das Synchronisieren der ganzen Methode.

[97] Die Methode `synchThisObject()` im folgenden Beispiel ist effizienter als die Methode `synchAllLocksExample()`, da sie den Aufwand des dreimaligen Sperrens und Entsperrens vermeidet:

```
01. public class SynchExample {
02.
03.     private Resource resourceOne   = new Resource();
04.     private Resource resourceTwo   = new Resource();
05.     private Resource resourceThree = new Resource();
06.
07.     public synchronized void synchThisObjectExample() {
08.         resourceOne.value   = -3;
09.         resourceTwo.value   = -2;
10.         resourceThree.value = -1;
11.     }
12.
13.     public void synchAllLocksExample() {
14.         synchronized (resourceOne) {
15.             resourceOne.value   = -3;
16.         }
17.
18.         synchronized (resourceTwo) {
19.             resourceOne.value   = -2;
20.         }
21.
22.         synchronized (resourceThree) {
23.             resourceOne.value   = -1;
24.         }
25.     }
26.
27.     private static class Resource {
28.         int value;
29.     }
30. }
```

Wenn Sie Ihre Klasse mit Sorgfalt so entworfen haben, daß kein unsynchronisierter Zugriff auf die gesperrten Resourcen möglich ist, dann ist das Synchronisieren ganzer Methoden wahrscheinlich der einfachere Ansatz.

4.1.4.4 Die Object-Methoden notify() und notifyAll()

[98] Jedes Java-Objekt kann die Threads auf seiner Warteliste benachrichtigen, daß ein für sie möglicherweise interessantes Ereignis eingetreten ist. Ein Aufruf der `notify()`-Methode des Objektes versetzt *einen* der wartenden Threads vom Wartezustand in den blockierten Zustand, bis der Thread das Objekt sperren und seine Verarbeitung fortgesetzt werden kann. Ein Aufruf der `notifyAll()`-Methode versetzt dagegen *alle* Threads auf der Warteliste des Objektes vom Wartezustand in den blockierten Zustand. Die Threads verbleiben im blockierten Zustand bis sie das Objekt sperren und verarbeitet werden können.

Warnung: Zur Unterscheidung zwischen Wartezustand und blockiertem Zustand eines Threads: Bei Java geht ein Thread in den Zustand `Thread.State.WAITING` über, nachdem er eine der Methoden `wait()` oder `join()` aufgerufen hat. Erfolgt der Zustandsübergang durch einen Aufruf der `wait()`-Methode eines Objektes, so beansprucht der aufrufende Thread keine Rechenzeit, bis ein anderer Thread die `notify()`- oder `notifyAll()`-Methode *desselben Objektes* aufruft oder der wartende Thread unterbrochen wird. Erfolgt der Zustandsübergang durch einen Aufruf der `join()`-Methode eines Objektes, so beansprucht der aufrufende Thread auch in diesem Fall keine Rechenzeit, bis der Thread, dessen Ende der aufrufende Thread abwartet terminiert oder unterbrochen wird. Versuchen dagegen mehrere Thread zugleich, ein Objekt zu sperren, so hat nur ein Thread Erfolg, während die übrigen in den Zustand `Thread.State.BLOCKED` versetzt werden, bis die Sperre aufgehoben wird, woraufhin ein anderer Thread das Objekt sperrt. Der Scheduler der Laufzeitumgebung wählt dazu automatisch einen Thread aus, der sich im Zustand `BLOCKED` befindet.

---

[99] Das folgenden Beispiel veranschaulicht den Unterschied zwischen den Methoden `notify()` und `notifyAll()`:

```
public class NotifyVersusNotifyAll extends Thread {
    private static Object mutex = new Object();

    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < 5; i++) {
            new NotifyVersusNotifyAll().start();
        }

        Thread.sleep(2000);
        synchronized(mutex) {
            System.out.println("Main thread kicking off other threads");
           mutex.notifyAll();
            // mutex.notify();
        }
    }

    public void run() {
        // Note: This code has changed significantly from what was
        // published in the book, as the sample in the book does not show
        // that the multiple threads are all woken.
        try {
            System.out.println(getName() + " waiting");
            synchronized(mutex) {
                mutex.wait();
            }
            // if we called notifyAll(), then all threads will get to this line
            // at roughly the same time. If we called notify() then only one
            // thread will get here, then when it calls notify() another thread
            // will get here, and so on.
```

```
                System.out.println(getName() + " woken up");
                Thread.sleep(2000);
                System.out.println(getName() + " waking up another thread");

                // need to get back inside a synchronized block to call notify()
                synchronized(mutex) {
                    mutex.notify();
                }
            } catch (InterruptedException ie) {
                ie.printStackTrace();
            }
        }
    }
```

[100] Die Ausgabe lautet:

```
~$ java NotifyVersusNotifyAll

Thread-0 waiting
Thread-1 waiting
Thread-2 waiting
Thread-3 waiting
Thread-4 waiting
Main thread kicking off other threads
Thread-0 woken up
Thread-0 waking up another thread
Thread-1 woken up
Thread-1 waking up another thread
Thread-2 woken up
Thread-2 waking up another thread
Thread-3 woken up
Thread-3 waking up another thread
Thread-4 woken up
Thread-4 waking up another thread

~$

~$ java NotifyVersusNotifyAll

Thread-0 waiting
Thread-1 waiting
Thread-2 waiting
Thread-3 waiting
Thread-4 waiting
Main thread kicking off other threads
Thread-0 woken up
Thread-1 woken up
Thread-2 woken up
Thread-3 woken up
Thread-4 woken up
Thread-0 waking up another thread
Thread-1 waking up another thread
Thread-2 waking up another thread
Thread-3 waking up another thread
Thread-4 waking up another thread

~$
```

Beim ersten Aufruf des Programms `NotifyVersusNotifyAll` ruft der `main`-Thread die `notify()`-Methode auf (`notifyAll()` auskommentiert). Beim zweiten Aufruf wird `notifyAll()` aufgerufen (`notify()` auskommentiert). Die Methode `notify()` versetzt einen wartenden Thread in den Zu-

stand `Thread.State.RUNNABLE`, während `notifyAll()` alle wartenden Threads in den Zustand `RUNNABLE` versetzt.

Warnung: Obwohl die Threads in der Ausgabe in der Reihenfolge benachrichtigt werden, in der sie die `wait()`-Methode des Mutexobjektes aufgerufen haben, dürfen Sie sich niemals auf diese Reihenfolge verlassen. Die Reihenfolge, in der die Threads benachrichtigt werden und die Reihenfolge in der die Threads das Mutexobjekt sperren, obliegt ausschließlich dem Ermessen des Threadschedulers der Laufzeitumgebung.

---

[101] In der Regel rufen Sie `notify()` nur dann auf, wenn Sie wissen, daß der benachrichtigte Thread die Benachrichtigung auch nutzen kann. Sind mehrere Threads vorhanden, deren Benachrichtung sinnvoll ist, wobei aber nur ein Thread in den Zustand `RUNNABLE` versetzt werden soll, dann verwenden Sie `notify()`. Sie müssen in diesem Fall aber dafür sorgen, daß die übrigen Threads nicht in der Schwebe bleiben, sondern rechtzeitig von einem anderen Thread benachrichtigt werden, damit ihre Verarbeitung fortgesetzt werden kann. Dies ist eine komplizierte Aufgabe und ihre korrekte Lösung erfordert sorgfältige Untersuchung. Existieren mehrere blockierte Threads, die darauf warten ein Objekt sperren zu können, um anschließend verschiedene Bedingungen zu prüfen, so verwenden Sie besser `notifyAll()`, damit alle Threads ihre Bedingungen prüfen können, wie im folgenden Erzeuger-Verbraucher-Beispiel:

```
// Erzeugerthread
synchronized (lock) {
    value = Math.random();
    lock.notifyAll();
}

// Verbraucherthread 1
synchronized (lock) {
    while (value < 0.5) {
        lock.wait();
    }
}

// Verbraucherthread 2
synchronized (lock) {
    while (value => 0.5) {
        lock.wait();
    }
}
```

[102] Dies ist ein Beispiel für eine Situation, in der sich `notifyAll()` besser eignet als `notify()`. Durch Aufrufen der `notifyAll()`-Methode nach dem Bewerten des `value`-Feldes kann der Erzeugerthread sicher sein, daß beide Verbraucherthreads den aktuellen Wert der `value`-Variablen prüfen und die Verarbeitung eines von beiden Threads fortgesetzt wird. Wäre `notify()` anstelle von `notifyAll()` aufgerufen worden, so wäre nach dem Bewerten der `value`-Variablen eventuell der falsche Thread gewählt und wieder in den Zustand `Thread.State.WAITING` zurückgesetzt worden, weil `value` nicht den gewünschten Wert enthält, das heißt keiner der Verbraucherthreads würde verarbeitet werden.

4.1.4.5 Objektsperren wirken nicht implizit

[103] Das Sperren eines Objektes bewirkt *nicht*, daß die von seinen Objektfeldern referenzierten Objekte ebenfalls gesperrt werden. Diese Tatsache mag auf den ersten Blick überraschen, ist aus

der Perspektive der Laufzeitumgebung aber sinnvoll, da sie einem Thread gestattet, das rekursive Sperren tief verschachtelter Objekte zu vermeiden. Stellen Sie den Aufwand vor, ein Objekt zu sperren, dann alle von seinen Objektfeldern referenzierten Objekte zu sperren, dann die von den Objektfeldern dieser Objekte referenzierten Objekte zu sperren .... Das folgende Beispiel zeigt, daß die von den Objektfeldern eines gesperrten Objektes referenzierten Objekte nicht gesperrt werden:

```
01. import java.util.*;
02.
03. public class LockObjectNotMemberVariables{
04.     private List myList = new ArrayList();
05.
06.     public static void main(String args[]){
07.         LockObjectNotMemberVariables lonmv =
08.                 new LockObjectNotMemberVariables();
09.         lonmv.lockTest();
10.     }
11.
12.     public synchronized void lockTest(){
13.         System.out.println("Is the THIS object locked? " +
14.                             Thread.holdsLock(this));
15.
16.         System.out.println("Is the list object locked? " +
17.                             Thread.holdsLock(myList));
18.     }
19. }
```

Ausgabe:

```
~$ java LockObjectNotMemberVariables

Is the THIS object locked? true
Is the list object locked? false

~$
```

[104] Das Sperren eines Klassenobjektes bewirkt *nicht*, daß die Objekte dieser Klassen ebenfalls gesperrt werden (siehe nächstes Beispiel). Beachten Sie, daß die Methode `lockTest()` sowohl statisch als auch synchronisiert ist, also die Sperre des Klassenobjektes aufnimmt:

```
01. public class ClassLockNotObjectLock {
02.     public static void main(String args[]) {
03.         lockTest();
04.     }
05.
06.     public static synchronized void lockTest() {
07.         ClassLockNotObjectLock clnoc = new ClassLockNotObjectLock();
08.         System.out.println("Is the class object locked? " +
09.                 Thread.currentThread().holdsLock(clnoc.getClass()));
10.
11.         System.out.println("Is the object Instance locked? " +
12.                 Thread.currentThread().holdsLock(clnoc));
13.     }
14. }
```

Ausgabe:

```
~$ java ClassLockNotObjectLock

Is the class object locked? true
Is the object Instance locked? false
```

```
~$
```

4.1.5 Der neue Sperrmechanismus in Version 5 des Java Development Kits

[105] Version 5 des Java Development Kits enthält ein paar Packages, die einen Sperrmechanismus sowie das Warten auf Bedingungen unabhängig von der Synchronisierung und dem Sperren von Objekten im vorigen Unterabschnitt ermöglichen. Das neuen Verfahren gestattet beispielsweise, beim Zugriff der einzelnen Threads auf die Sperre eines Objektes für Gleichberechtigung zu sorgen. (Der Ansatz im vorigen Unterabschnitt gewährleistet bei der Benachrichtigung wartender Threads keinerlei Reihenfolge. Ein Thread, der gerade erst in den Wartezustand zurückgesetzt wurde, kann vor einem Thread zur Weiterverarbeitung gewählt werden, der bereits längere Zeit gewartet hat.)

[106] Das in Version 5 des Java Development Kits neu hinzugekommene Package `java.util.concurrent.locks` beinhaltet einige im Hinblick auf die Prüfung zum *Sun Certified Java Developer* nützliche Fähigkeiten. *`ReadWriteLock`* gestattet beispielsweise, daß mehrere Threads einen Datensatz zum Lesen sperren können, aber nur ein Thread zum Schreiben. Wir besprechen das Interface *`ReadWriteLock`* in Kapitel 5.

[107] Unter Version 1.4 des Java Development Kits war es nicht möglich, nach einem Aufruf der Methode `wait(milliseconds)` direkt in Erfahrung zu bringen, ob der Thread benachrichtigt worden oder die vereinbarte Wartezeit abgelaufen war. Die *`Lock`*-Methode `tryLock(time, unit)` gibt `true` zurück, wenn der aufrufende Thread das Objekt in der gegebenen Zeit sperren konnte und `false`, sonst. Sie finden ein Anwendungsbeispiel für die ähnliche *`Lock`*-Methode `await(time, unit)` in Unterabschnitt 5.4.2. Sie können nun mehr als nur eine Bedingung festlegen, bei deren Eintreten ein Thread benachrichtigt wird (für die Prüfung zum *Sun Certified Java Developer* nicht verlangt).

[108] Obwohl die neuen Kommandos anders aussehen als die Synchronisierung von Blöcken, ist es nicht schwierig die eine Variante in die andere zu übertragen. Der folgende Ausschnitt verwendet zum Beispiel einen synchronisierten Block:

```
Object lock = new Object();
public void soSomething() {
    synchronized (lock) {
        while (true) {
            try {
                lock.wait();
                // something happens here
            } catch (InterruptedException ie) {
                // handle exception
            }
        }
    }
}
```

[109] Umgeschrieben mit den neuen Kommandos:

```
private static Lock lock = new ReentrantLock();
private static Condition lockReleased = lock.newCondition();
public void doSomething() {
    lock.lock();
    try {
        lockReleased.await();
    } catch (InterruptedException ie) {
        // handle exception
    } finally {
```

```
            lock.unlock();
        }
    }
```

[110] Ein Anwendungsbeispiel für den neuen Sperrmechanismus finden Sie in Unterabschnitt 5.4.2, weiterführende Überlegungen im Unterunterabschnitt 5.4.3.3.

[111] Damit eine Sperre sicher aufgehoben wird, empfehlen wir, daß Sie den Aufruf der `unlock()`-Methode wie im obigen Beispiel, in einen `finally`-Block setzen.

[112] Die Betrachtungen zur Threadsicherheit im folgenden Abschnitt gelten sowohl für traditionell synchronisierte Blöcke, als auch für die neuen Klassen.

4.1.6 Zusammenfassung: Sperren von Objekten

[113] Zwischen dem Sperren eines Objektes und dem Sperren eines von einem seiner Felder referenzierten Objektes besteht keine Wechselwirkung. Mit anderen Worten bewirkt das Sperren eines Objektes nicht, daß die von dessen Feldern referenzierten Objekte implizit ebenfalls gesperrt werden. Das Sperren eines von einem Feld eines Objektes referenzierten anderen Objektes bewirkt nicht, daß das Objekt zu dem das Feld gehört ebenfalls gesperrt wird. Threads die verschiedene Objekte sperren, können gleichzeitig verarbeitet werden, solange sie nicht die Sperre eines gemeinsam verwendeten Objektes benötigen. Das Sperren des Klassenobjektes bewirkt nicht, daß die Objekte dieser Klasse ebenfalls gesperrt werden. Die Threadsicherheit kann durch unsynchronisierten Zugriff auf Objektfelder verletzt werden.

4.2 Threadsicherheit

[114] Beim Programmieren mit Threads gibt es einige Fallen, die häufig unerwartet und scheinbar zufällig auftreten. Wir stellen in diesem Abschnitt einige der häufigeren Probleme vor und geben Möglichkeiten an, wie sie vermieden werden können. Die folgenden Unterabschnitte erklären den Begriff „Threadsicherheit“ und die Effekte, die durch Threadsicherheit verhindert werden können.

4.2.1 Verklemmungen (Deadlocks)

[115] Eine Verklemmung liegt vor, wenn Threads dauerhaft blockiert sind und auf eine Bedingung warten, die nicht eintreten kann. Beispiel: Daffy Duck und Bugs Bunny sind auf einer Insel gestrandet. Daffy hat eine Konservendose mit Nahrungsmitteln und Bugs hat einen Dosenöffner. Daffy gibt die Dose nicht her, ohne den Dosenöffner zu bekommen und Bugs gibt den Dosenöffner nicht her, ohne die Dose zu bekommen. Die Situation ist unlösbar.

[116] Im folgenden Beispiel sperrt `thread1` das in Zeile 8 von `lockA` referenzierte Objekt, muß aber auch das in Zeile 9 von `lockB` referenzierte Objekt sperren. `thread2` sperrt dagegen das von `lockB` referenzierte Objekt, muß aber auch das von `lockA` referenzierte Objekt sperren. Keiner der beiden Threads gestattet dem anderen seine Verarbeitung fortzusetzen, setzt aber auch seine eigene Verarbeitung nicht fort. Das ist eine Verklemmung:

```
01. public class DeadlockExample {
02.     /**
03.      * Entry point to the application. Creates 2 threads that will deadlock.
04.      */
05.     public static void main(String args[]){
```

```
06.         DeadlockExample dle = new DeadlockExample();
07.
08.         Object lockA = ”Lock A”;
09.         Object lockB = ”Lock B”;
10.
11.         Runner thread1 = new Runner(lockA, lockB);
12.         Runner thread2 = new Runner(lockB, lockA);
13.
14.         thread1.start();
15.         thread2.start();
16.     }
17.
18.     /**
19.       * Lock two objects in the order they were specified in the constructor.
20.       */
21.     static class Runner extends Thread {
22.         private Object lock1;
23.         private Object lock2;
24.
25.         public Runner(Object firstLockToGet, Object secondLockToGet) {
26.             this.lock1 = firstLockToGet;
27.             this.lock2 = secondLockToGet;
28.         }
29.
30.         public void run() {
31.             String name = Thread.currentThread().getName();
32.             synchronized(lock1) {
33.                 System.out.println(name + ”: locked ” + lock1);
34.                 delay(name);
35.                 System.out.println(name + ”: trying to get ” + lock2);
36.                 synchronized(lock2) {
37.                     System.out.println(name + ”: locked ” + lock2);
38.                 }
39.             }
40.         }
41.     }
42.
43.     /**
44.       * build in a delay to allow the other thread time to lock the object
45.       * the delaying thread would like to get.
46.       */
47.     private static void delay(String name) {
48.         try {
49.             System.out.println(name + ”: delaying 1 second”);
50.             Thread.currentThread().sleep(1000L);
51.         } catch(InterruptedException ie) {
52.             ie.printStackTrace();
53.         }
54.     }
55. }
```

Ausgabe:

```
~$ java DeadlockExample

Thread-0: locked Lock A
Thread-0: delaying 1 second
Thread-1: locked Lock B
```

```
Thread-1: delaying 1 second
Thread-0: trying to get Lock B
Thread-1: trying to get Lock A
~$
```

[117] Das obige Programm ist ein Beispiel für schlechten Entwurf. Es gibt fast immer eine bessere Lösung als verschachtelte Sperren. Wenn Sie keine einfachere Lösung für einen Teil Ihrer Aufgabe finden, betrachten Sie den Entwurf Ihrer Anwendung im Großen. Sun Microsystems stellt zwar schwierige Aufgaben in der Prüfung zum *Sun Certified Java Developer*, aber die meisten Aufgaben lassen sich elegant lösen.

4.2.2 Wettlaufsituationen (Race Conditions)

[118] Eine Wettlaufsituation liegt vor, wenn zwei oder mehr Threads um den Zugriff auf dieselbe Resource wetteifern und das Verhalten des Programms davon abhängt, welcher Thread „gewinnt“. Stellen Sie als Beispiel sich vor, daß die Kollegen Johnson und Smith morgens ins Büro hetzen, um der erste am Testserver zu sein. Je nach dem wer gewinnt, wird der Server für ein paar Minuten oder für einige Stunden intensiv beansprucht. Das Programmverhalten hängt also von zufälligen Faktoren ab, beispielsweise davon, wer morgens im Berufsverkehr steckenbleibt. Solches Verhalten ist nur selten nützlich. Es kann sehr schwierig sein, Wettlaufsituationen aufzuspüren, da ihre Wirkung nur selten auftritt und daher schwierig zu reproduzieren ist.

[119] Wettlaufsituationen können Verklemmungen auslösen, wenn Objekte in einer anderen als er erwarteten Reihenfolge gesperrt werden. Ihre Anwendung kann stets Objekt 1 vor Objekt 2 gesperrt haben, sperrt die Objekte aber diesesmal umgekehrt und gibt die Sperren in einer unerwarteten Reihenfolge wieder frei.

[120] Das Vorhandensein einer Wettlaufsituation äußert sich dadurch, daß Ihre Anwendung scheinbar plötzlich ein anderes Verhalten zeigt, zum Beispiel:

```
01. public class RaceConditionExample {
02.     public static void main(String args[]) {
03.         //create an instance of this object
04.         RaceConditionExample rce = new RaceConditionExample();
05.
06.         //create two runners
07.         Runner johnson = rce.new Runner("Johnson");
08.         Runner smith = rce.new Runner("Smith");
09.
10.         //point both runners to the same resource
11.         smith.server =  "the common object";
12.         johnson.server = smith.server;
13.
14.         //start the race, based on a random factor, one thread
15.         //or the other gets to start first.
16.         if (Math.random() > .5) {
17.             johnson.start();
18.             smith.start();
19.         } else {
20.             smith.start();
21.             johnson.start();
22.         }
23.     }
24.
25.     /**
```

```
26.       * Creates a thread, then races for the resource
27.       */
28.     class Runner extends Thread {
29.         public Object server;
30.
31.         public Runner(String name) {
32.             super(name);
33.         }
34.
35.         public void run() {
36.             System.out.println(getName() + ": trying for lock on " + server);
37.             synchronized (server) {
38.                 System.out.println(getName() + ": has lock on " + server);
39.                 // wait 2 seconds: show the other thread really is blocked
40.                 try {
41.                     Thread.sleep(2000);
42.                 } catch (InterruptedException ie) {
43.                     ie.printStackTrace();
44.                 }
45.                 System.out.println(getName() + ": releasing lock ");
46.             }
47.         }
48.     }
49. }
```

Ausgabe:

```
~$ java RaceConditionExample

Smith: trying for lock on the common object
Smith: has lock on the common object
Johnson: trying for lock on the common object
Smith: releasing lock
Johnson: has lock on the common object
Johnson: releasing lock

~$ java RaceConditionExample

Johnson: trying for lock on the common object
Johnson: has lock on the common object
Smith: trying for lock on the common object
Johnson: releasing lock
Smith: has lock on the common object
Smith: releasing lock

~$ java RaceConditionExample

Johnson: trying for lock on the common object
Johnson: has lock on the common object
Smith: trying for lock on the common object
Johnson: releasing lock
Smith: has lock on the common object
Smith: releasing lock

~$
```

Wettlaufsituationen sind schwierig zu finden und zu reproduzieren. Aus diesem Grund ist es so wichtig, bei der Planung einer Anwendung wohldurchdachte Entscheidungen zu fällen, um das gefährliche Gebiet der Wettlaufsituationen zu umschiffen.

[121] In diesem Beispiel ist offensichtlich nicht vorhersagbar, welcher Thread zuerst Zugriff auf den

Server erhält. Das kann, abhängig vom Kontext, völlig in Ordnung sein, aber auch zerstörerische Auswirkungen haben. Nicht jede Wettlaufsituation muß vermieden werden, um Threadsicherheit zu gewährleisten, aber es wichtig, sich dieses Effektes bewußt zu sein.

4.2.3 Verhungern (Starvation)

[122] Ein Thread „verhungert“, wenn er keine Chance bekommt, um verarbeitet zu werden. Dieser Effekt zeigt sich am häufigsten, wenn Threads mit höherer Priorität bevorzugt vor anderen Threads mit geringerer Priorität verarbeitet werden. Stellen Sie sich zum Beispiel vor, daß Sie mit dem technischen Leiter Ihres Unternehmens sprechen müssen. Aber jedesmal, wenn der Chef Zeit hat, springt ein leitender Angestellter ein und beansprucht den Zeitraum, den Sie nutzen wollten. Obwohl jeder der leitenden Angestellten einen guten Grund gehabt haben kann, hatten Sie schließlich keine Gelegenheit mit dem technischen Leiter zu sprechen (Ihr Thread hatte keine Chance ausgeführt zu werden).

[123] Das nächste Programm ist ein künstliches Beispiel mit drei Threads hoher Priorität und einem Thread niedriger Priorität, die alle eine gemeinsame Resource zu verwenden versuchen. Beispiel für das Verhungern eines Threads:

```
01. /**
02.   * Demonstrate the concept of a starving thread.
03.   */
04. public class StarvationExample{
05.     public static void main(String args[]){
06.         // Ensure the main thread competes with the other threads
07.         Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
08.         // Create an instance of this object
09.         StarvationExample se = new StarvationExample();
10.         // Create 5 threads, marking number 1 as a very low priority
11.         for(int i=1; i< 5; i++){
12.             //create a runner
13.             Runner r = se.new Runner();
14.             r.setPriority(Thread.MAX_PRIORITY);
15.
16.             //set the first thread to starve
17.             if (i == 1) {
18.                 r.setPriority(Thread.MIN_PRIORITY);
19.                 r.setName("Starvation Thread");
20.             }
21.             //start the thread.
22.             r.start();
23.         }
24.
25.             // Exit as soon as we possibly can
26.             System.exit(0);
27.     }
28.
29.     /**
30.       * Create a thread, then cycle through its command ten times.
31.       */
32.     class Runner extends Thread{
33.         public void run(){
34.             for (int count = 10; count > 0; count-) {
35.                 System.out.println(getName() + ": is working " + count);
36.             }
```

```
37.         }
38.     }
39. }
```

[124] Der `Starvation Thread` wurde nicht aufgerufen, wie Sie der Ausgabe des obigen Beispiels entnehmen können:

```
~$ java StarvationExample

Thread-1: is working 10
Thread-2: is working 10
Thread-3: is working 10
Thread-1: is working 9
Thread-2: is working 9
Thread-3: is working 9
Thread-1: is working 8
Thread-2: is working 8
Thread-3: is working 8
Thread-1: is working 7
Thread-2: is working 7
Thread-3: is working 7
Thread-1: is working 6
Thread-2: is working 6
Thread-3: is working 6

    ⋮   // Ausgabe gekürzt

~$
```

Bemerkung: Die exakte Ausgabe variiert bei jedem Programmaufruf, da die Berücksichtigung der Threadprioritäten durch den Threadscheduler nicht garantiert ist. In der Mehrzahl der Aufrufe dieses Programms sollte sich aber zeigen, daß der `Starvation Thread` entweder gar nicht oder erheblich seltener als die anderen Threads ausgeführt wird. Wenn der `Starvation Thread` aufgerufen wird, ist die Verarbeitung der übrigen Threads in der Regel bereits beendet.

4.2.4 Atomare Operationen

[125] Atomare Operationen sind unteilbar. Die Verarbeitung eines Threads wird während einer atomaren Operation nicht unterbrochen. Die einzigen natürlichen atomaren Operation unter Java sind Zuweisungen. Zum Beispiel ist

```
x = 45;
```

eine atomare Operation, wenn `x` ein `int`-Feld beziehungsweise eine lokale `int`-Variable ist. Zuweisungen mit `double`- oder `long`-Werten, sind dagegen *nicht* atomar, repräsentieren also eine Ausnahme von der obigen Regel.

[126] Operationen mit `double`- oder `long`-Werten zerfallen aufgrund der Größe der Operanden in zwei Operationen. Die erste Teiloperation setzt die höherwertigen 32 Bits und die zweite Teiloperation die niederwertigen 32 Bits. Das bedeutet, daß die Verarbeitung eines Threads der eine solche Zuweisung ausführt, zwischen den beiden Teiloperationen unterbrochen werden kann.

[127] Ein Thread verarbeitet eine Folge programmatischer Anweisungen. Jede dieser Anweisungen besteht wiederum aus atomaren Operationen. Der Thread kann überall ausgesetzt werden, außer während der der Ausführung einer atomaren Operation. Beispiel: Die beiden Anweisungen

```
1. x = 7;
2. y = x++;
```

bestehen eigentlich aus vier Operationen:

```
1.  x = 7;
2a. int temp = x + 1;
2b. x = temp;
2c. y = x;
```

[128] Angenommen, `x`, `y` und `z` seien Felder eines Objektes oder einer Klasse. Das Umschalten zwischen den Zeilen 2a und 2b kann zu einem unerwarteten Ergebnis führen, wenn der zwischengeschaltete Thread das Feld `x` beispielsweise mit dem Inhalt 13 bewertet. Dies kann zu dem unverständlichen Ergebnis führen, daß das Feld `y` den Wert 13 enthält. Dieses Problem würde natürlich nicht entstehen, wenn `x`, `y` und `z` lokale Variablen wären, weil sie dann für keine andere Methode erreichbar wären.

[129] Indem Sie Ihre Methode in einen synchronisierten Block setzen, definieren Sie, aus der Perspektive der übrigen Threads betrachtet, die auf dieses Objekt zugreifen, effektiv die gesamte Methode als eine einzige atomare Operation. Selbst wenn ein anderer Thread zwischengeschaltet wird, bevor die Verarbeitung Ihrer Methode abgeschlossen ist, kommt es garantiert nicht zu fehlerhaften Daten wie im obigen Beispiel, da der andere Thread keine synchronisierten Methoden des Objektes aufrufen kann, bevor die Verarbeitung Ihrer Methode beendet ist.

[130] Das nächste Programm zeigt ein künstliches Beispiel mit zehn Threads, die zu einem ungünstigen Zeitpunkt einem anderen Thread den Vortritt lassen und dadurch ungültige Ergebnisse erzeugen:

```
01. public class NonAtomic {
02.     static int x;
03.
04.     public static void main(String[] args) {
            NonAtomic na = new NonAtomic();
05.         for (int i = 0; i < 10; i++) {
06.             na.new Runner().start();
07.         }
08.     }
09.
10.     class Runner extends Thread {
11.         private int validCounts = 0;
12.         private int invalidCounts = 0;
13.
14.         public void run() {
15.             for (int i = 0; i < 10; i++) {
16. //              synchronized (NonAtomic.class) {
17.                 int reference = (int) (Math.random() * 100);
18.                 x = reference;
19.
20.                 // either yielding or doing something intensive
21.                 // should cause the problem to manifest.
22.                 yield();
23. //                  for (int y = 0; y < 10000; y++) {
24. //                      Math.tan(200);
25. //                  }
26.
27.                 if (x == reference) {
28.                     validCounts++;
29.                 } else {
```

```
30.                     invalidCounts++;
31.                 }
32.             }
33. //            }
34.
35.             System.out.println(getName()
36.                                + " valid: " + validCounts
37.                                + " invalid: " + invalidCounts);
38.         }
39.     }
40. }
```

[131] Wenn Sie das Programm starten, werden Sie beim Abfragen in Zeile 27 feststellen, daß in der Mehrzahl der Fälle ein anderer Thread den in Zeile 18 gesetzten Wert von `x` geändert hat:

```
~$ java NonAtomic

Thread-0 valid: 2 invalid: 8
Thread-1 valid: 2 invalid: 8
Thread-3 valid: 10 invalid: 0
Thread-4 valid: 8 invalid: 2
Thread-2 valid: 10 invalid: 0
Thread-5 valid: 8 invalid: 2
Thread-6 valid: 10 invalid: 0
Thread-8 valid: 6 invalid: 4
Thread-9 valid: 7 invalid: 3
Thread-7 valid: 3 invalid: 7

~$
```

[132] Durch Aufrufen der `yield()`-Methode wird das Problem deutlicher sichtbar, als unter normalen Umständen, wobei nicht `yield()` selbst das Problem darstellt, sondern die benachbarten *unsynchronisierten* Anweisungen. Sie können diese Behauptung nachprüfen, indem Sie den `yield()`-Aufruf in Zeile 22 aus- und statt dessen die `for`-Schleife in den Zeilen 23–25 einkommentieren. Auch nach dieser Änderung verursacht die nicht-atomare Programmstruktur viele Fehler, obwohl der laufende Thread nicht mehr zurücktritt, um anderen Threads Vorrang zu gewähren. Sie können außerdem feststellen, daß diese Änderung eine erheblich längere Laufzeit und deutlich mehr Rechenleistung erfordert.

[133] Wenn Sie nun die Kommentarzeichen vor den Zeilen 16 und 33 entfernen, können Sie beobachten, daß sich das Programm durch die Synchronisierung wie eine atomare Anweisung verhält, gleichgültig ob Sie `yield()` aufrufen oder die `for`-Schleife verwenden.

4.2.5 Zusammenfassung: Threadsicherheit

[134] Sie können jede der vier in diesem Abschnitt vorgestellten Situationen programmatisch lösen, wobei aber die beste Vorgehensweise darin besteht, Situationen zu vermeiden, in denen Threadsicherheit nicht gewährleistet ist. Obwohl nahezu jede Regel eine Ausnahme hat, sollten Sie im allgemeinen keine verschachtelten Sperren verwenden, sich nicht auf Threadprioritäten verlassen und auf Wettlaufsituationen achten. Der Abschnitt 4.4 am Ende dieses Kapitels hilft Ihnen dabei, die Gefahr einzuschränken, in eine Situation zu geraten, in denen sich solche Probleme ergeben können.

4.3 Thread-Objekte

[135] Dieser Abschnitt enthält einige wichtige allgemeine Informationen, die Sie im Kopf haben sollten, wenn Sie direkt mit `Thread`-Objekten arbeiten.

4.3.1 Die Methoden stop(), suspend(), destroy() und resume()

[136] Die `Thread`-Methoden `stop()`, `suspend()`, `resume()` und `destroy()` sind von Natur aus unsicher und wurden als *deprecated* deklariert. Verwenden Sie diese Methoden unter keinen Umständen.

[137] Anstelle einer dieser Methoden verwenden Sie besser ein Feld, welches sowohl für den Thread erreichbar ist, dessen Zustand geändert werden soll, als auch für den Thread, der die Zustandsänderung bewirken soll. Der Thread dessen Zustand geändert wird, sollte dieses Feld periodisch auswerten und seinen Zustand in sicherer Weise ändern, wenn sich der Inhalt des Feldes geändert hat (durch Freigeben von Resourcen, Schließen von Dateien, ...).

[138] Mehr Informationen finden Sie in der Beschreibung, warum diese Methoden als *deprecated* deklariert wurden unter der Internetadresse *http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html*.

4.3.2 Threadzustände

[139] Es gibt sechs Threadzustände: `Thread.State.NEW`, `Thread.State.RUNNABLE`, `Thread.State.WAITING`, `Thread.State.TIMED_WAITING`, `Thread.State.BLOCKED` und `Thread.State.TERMINATED`. Der Übergang von einem Threadzustand in einen anderen geschieht bei Java nach bestimmten Regeln. Es ist wichtig, diese Regeln zu kennen.

[140] Die wichtigste Regel lautet, daß ein Thread, der sich im Zustand `TERMINATED` befindet, keinen der anderen Zustände mehr annehmen kann. Die zweitwichtigste Regel lautet, daß ein Thread nur Anweisungen ausführen kann, wenn er sich im Zustand `RUNNABLE` befindet. Nach dem Aufruf

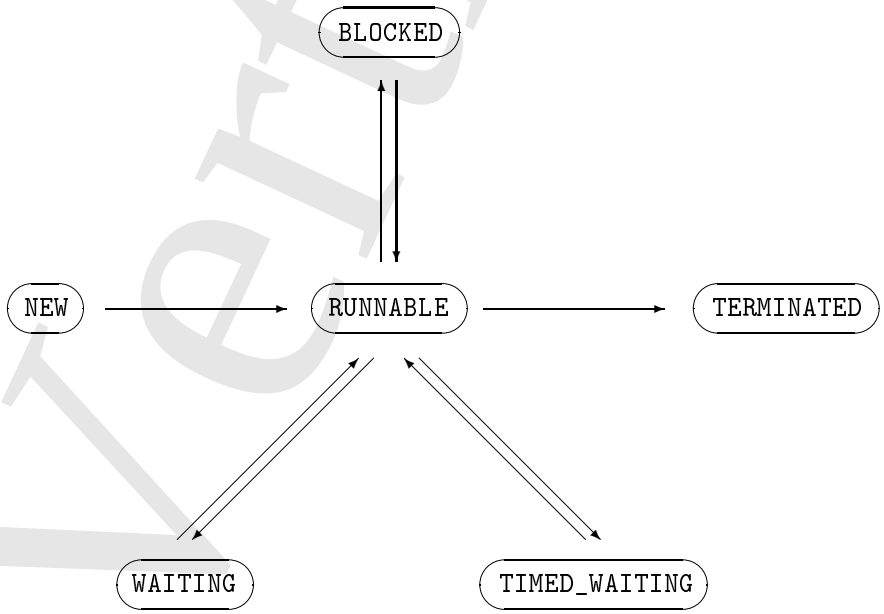

Abbildung 4.1: Threadzustände und Zustandsübergänge.

der `start()`-Methode versetzt der Scheduler den Thread in den Zustand `RUNNABLE`. Anschließend bringt der Scheduler den Thread nach eigenem Ermessen zur Ausführung, so daß die Anweisungen im Körper der `run()`-Methode ausgeführt werden.

[141] Es gibt vier mögliche Zustandsübergänge für einen Thread im Zustand `RUNNABLE`, siehe Abbildung 4.1:

- Der Thread kann bis zu seinem Ende verarbeitet werden, woraufhin er in den Zustand `TERMINATED` übergeht.
- Der Thread kann durch einen Aufruf der Methode `wait()` (ohne Zeitangabe) oder `join()` in den Zustand `WAITING` übergehen.
- Der Thread kann durch einen Aufruf der Methode `wait()` oder `sleep()` (*mit* Zeitangabe) in den Zustand `TIMED_WAITING` übergehen.
- Der Thread kann eine benötigte Sperre nicht aufnehmen und geht in den Zustand `BLOCKED` über.

4.3.3 Threads im blockierten Zustand, Teil 2

[142] Der blockierte Zustand (`Thread.State.BLOCKED`) ist ein wichtiges Phänomen bei Threads und verdient somit einige zusätzliche erklärende Worte. Kurze Wiederholung: Ein Thread wird im blockierten Zustand nicht verarbeitet und beansprucht daher keine Rechenzeit. Ein Thread wartet im blockierten Zustand nicht darauf, daß eine Zeitspanne verstreicht, sondern auf das Eintreten eines Ereignisses, genauer darauf, daß ein benötigtes Objekt gesperrt werden kann. Geht ein Thread in den blockierten Zustand über, wird seine Verarbeitung also unterbrochen, so bleibt die Sperrung der von diesem Thread gesperrten Objekte erhalten. Ein Thread, der sich im blockierten Zustand befindet und die benötigte Resource nicht sperren kann, kann anderen Threads den Zugriff auf benötigte Resourcen verweigern und unter Umständen sogar dauerhaft in seinem „Winterschlaf“ verharren. Beispiel für einen permanent blockierten Thread:

```
01. import java.net.*;
02.
03. public class BlockingExample extends Thread {
04.     private static Object mylock = new Object();
05.
06.     public static void main(String args[]) throws Exception {
07.         LockOwner lo = new LockOwner();
08.         lo.setName(”Lock owner”);
09.         lo.start();
10.
11.         // Wait for a little while for the lock owner thread to start
12.         Thread.sleep(200);
13.
14.         // Now start the thread that will be blocked
15.         BlockingExample be = new BlockingExample();
16.         be.setName(”Blocked thread”);
17.         be.start();
18.
19.         // Wait for a little while for the blocked thread to start
20.         Thread.sleep(200);
21.
22.         // Now print the two threads states
23.         printState(lo);
24.         printState(be);
```

```
25.     }
26.
27.     // start a thread.
28.     public void run() {
29.         // wait for the mylock object to be freed,
30.         // which will never happen
31.         synchronized (mylock) {
32.             System.out.println(getName() + " owns lock");
33.             System.out.println("doing Stuff");
34.         }
35.         System.out.println(getName() + " released lock");
36.     }
37.
38.     private static void printState(Thread t) {
39.         System.out.println();
40.         System.out.println("State of thread named: " + t.getName());
41.         System.out.println("State: " + t.getState());
42.         System.out.println("begin trace");
43.         for (StackTraceElement ste : t.getStackTrace()) {
44.             System.out.println("\t" + ste);
45.         }
46.         System.out.println("end trace");
47.     }
48.
49.     static class LockOwner extends Thread {
50.         public void run() {
51.             synchronized (mylock) {
52.                 System.out.println(getName() + " owns lock");
53.                 try {
54.                     ServerSocket ss = new ServerSocket(8080);
55.                     ss.accept();
56.                 } catch (Exception e) {
57.                     e.printStackTrace();
58.                 }
59.             }
60.             System.out.println(getName() + " released lock");
61.         }
62.     }
63. }
```

Ausgabe:

```
~$ java BlockingExample

lock owner owns lock

State of thread named: Lock owner
State: RUNNABLE
begin trace
        java.net.PlainSocketImpl.socketAccept(Native Method)
        java.net.PlainSocketImpl.accept(PlainSocketImpl.java:384)
        java.net.ServerSocket.implAccept(ServerSocket.java:453)
        java.net.ServerSocket.accept(ServerSocket.java:421)
        BlockingExample$LockOwner.run(BlockingExample.java:55)
end trace

State of thread named: Blocked thread
State: BLOCKED
begin trace
```

```
        BlockingExample.run(BlockingExample.java:32)
end trace

~$
```

Bemerkung: In den Zeilen 54/55 wird ein Socket erzeugt und beginnt an Port 8080 zu lauschen. Es gibt keinen besonderen Grund für die Wahl dieses Ports, mit Ausnahme der Tatsache, daß dieser Port nicht offiziell einem Protokoll zugeordnet und daher wahrscheinlich verfügbar ist. Wenn auf Ihrem Rechner eine Anwendung läuft, die den Port 8080 verwendet, zum Beispiel Tomcat, der per Voreinstellung diesen Port benutzt, so wird eine Ausnahme vom Typ `java.net.BindException` ausgeworfen. Tritt diese Ausnahme auf, dann wählen Sie einfach in Zeile 54 statt 8080 eine andere Portadresse, beispielsweise 9090.

[143] Die Sperre des von `myLock` referenzierten Objektes wird niemals aufgehoben, da der `LockOwner`-Thread endlos läuft. Da die Sperre niemals aufgehoben wird, befindet sich der auf diese Sperre wartende `BlockingExample`-Thread dauerhaft im blockierten Zustand und erhält keine Gelegenheit zur Verarbeitung.

[144] Die Verarbeitung des `main`-Threads würde fortgesetzt werden, wenn er nicht von der gesperrten Resource abhängen würde. *Again, this is because blocking threads do not use CPU cycles.*

Warnung: In vielen englischsprachigen Büchern und Zeitschriftenartikeln wird die Sprechweise „a thread or application is blocked for I/O“ verwendet, um auszudrücken, daß der Thread beziehungsweise die Anwendung darauf wartet, daß eine Ein-/Ausgabeoperation möglich wird. Dieser Sprechweise folgend, ist der `LockOwner`-Thread im obigen Beispiel „*blocked for* a client connecting to the socket“, übersetzt etwa: „solange blockiert, bis sich ein Client mit dem Socket verbindet.“ Das vorige Beispiel zeigt, daß ein auf eine Ein-/Ausgabeoperation wartender Thread den Zustand `RUNNABLE`, oder auch einen anderen Zustand haben kann, je nach Implementierung der Klasse, die für die Ein-/Ausgabeoperation zuständig ist. Denken Sie bei der Fehlersuche oder wenn Sie den Quelltext einer anderen Anwendung durchsehen daran, daß Java keinen separaten Zustand für Threads kennt, die auf eine Ein-/Ausgabeoperation warten.

4.3.3.1 Der Ausnahmetyp InterruptedException

[145] Eine Ausnahme vom Typ `java.lang.InterruptedException` wird typischerweise dann ausgeworfen, wenn sich ein Thread zu lange in einem durch durch `sleep()`, `wait()` oder `join()` verursachten Wartezustand befunden hat und ein anderer Thread die `interrupt()`-Methode des ersteren aufgerufen hat, um den Wartezustand zu unterbrechen. Stellen Sie sich, wenn eine Ausnahme von Typ `InterruptedException` ausgeworfen wird, vor, daß ein Thread einen anderen mit Gewalt wachgerüttelt hat. Ein wachgerüttelter Thread führt zuerst die Anweisungen in der `catch`-Klausel (zu `InterruptedException`) aus.

[146] Stellen Sie sich zum Beispiel eine Klasse namens `Data` mit einer `lockRecord()`-Methode vor, die auf einen Datensatz warten muß, bevor sie ihn sperren kann, aber keine Zeitüberschreitung hat. Nach dem die Entwicklung der Klasse `Data` abgeschlossen ist, erhalten Sie den Auftrag, eine neue Klasse von `Data` abzuleiten, die beim Sperren von Datensätzen eine Zeitüberschreitung berücksichtigt. Sie könnten nun den Sperrmechanismus neu schreiben oder aber lediglich einen separaten Thread erzeugen, welcher den Thread unterbricht, der den Datensatz sperrt, wenn die Sperre nicht während der Zeitüberschreitung aufgehoben wird.

Bemerkung: Ein Thread kann jederzeit unterbrochen werden, auch wenn er die Methoden `sleep()`, `wait()` oder `join()` nicht aufgerufen hat. In diesem Fall wird die Verarbeitung des Threads normal fortgesetzt, bis der Thread `sleep()`, `wait()` oder `join()` aufruft, woraufhin die aufgerufene Methode sofort eine Ausnahme vom Typ `InterruptedException` auswirft.

4.3.4 Synchronisierung

[147] Der Begriff „Synchronisierung“ umfaßt zwei Fälle: Das Synchronisieren ganzer Klassen beziehungsweise Objekt sowie das Synchronisieren von Blöcken und Methoden.

4.3.4.1 Synchronisierte Klassen

[148] Die Klasse `Vector` ist synchronisiert, die Klasse `ArrayList` dagegen nicht. Was bedeutet „Synchronisierung“ in diesem Kontext? Eine Klasse heißt synchronisiert, wenn alle Methoden, die interne Daten der Klasse abfragen oder ändern, synchronisiert sind. Jede Änderung am Zustand eines `Vector`-Objektes durch einen Thread ist unmittelbar danach für alle anderen Threads sichtbar. Bei `ArrayList`-Objekten besteht diese Garantie dagegen nicht. Insbesondere bedeutet die Synchronisierung der Klasse `Vector` *nicht*, daß die Verwendung eines `Vector`-Objektes automatisch Threadsicherheit gewährleistet. Wenn Sie threadsichere Verwendung eines `Vector`-Objektes in Ihrem Programm gewährleisten möchten, müssen Sie alle Zugriffe bezüglich des `Vector`-Objektes synchronisieren, analog wie Sie die Zugriffe auf das `ArrayList`-Objekt bezüglich des `ArrayList`-Objektes synchronisieren müssen.

Bemerkung: Die Klasse `Vector` ist ein Beispiel für eine threadsichere Kollektion. Allerdings ist die von `Vector`-Objekten gewährleistete Threadsicherheit selten erforderlich (der folgende Unterunterabschnitt erläutert welche Art von Threadsicherheit `Vector` garantiert). Falls Sie diese Art von Threadsicherheit nicht benötigen, fahren Sie eventuell besser, wenn Sie die `Collections`-Methode `synchronizedCollection()` verwenden, um eine threadsichere Kollektion mit den Eigenschaften Ihrer benötigten Ausgangskollektion zu erhalten.

[149] Eine synchronisierte Klasse verspricht nur eines: Jede Methode eines Objektes dieser Klasse verhält sich atomar. Stellen Sie sich zwei Threads und zwei Felder `myVector` (referenziert ein `Vector`-Objekt) und `myArrayList` (referenziert ein `ArrayList`-Objekt) vor. `Thread1` hat begonnen, Zeile 1 auszuführen, `Thread2` ist im Begriff, mit der Verarbeitung von Zeile 2 beginnen:

Thread 1:

```
1. myVector.add("Hello");
2. myArrayList.add("Hello");
```

Thread 2:

```
A. myVector.add("Cruel");
B. myArrayList.add("Cruel");
```

[150] Angenommen, die Verarbeitung von `Thread1` wird in der Mitte von Zeile 1 vom Threadscheduler unterbrochen. Dann kann der zwischengeschaltete `Thread2` Zeile A nicht verarbeiten, da die Klasse `Vector` synchronisiert ist. Sie können also sicher sein, daß `Thread2` das Element `Cruel` nach `Hello` einsetzt.

[151] Diese Garantie besteht nicht für das von `myArrayList` referenzierte `ArrayList`-Objekt. Genauer, wenn der Threadscheduler die Verarbeitung von `Thread1` in der Mitte von Zeile 2 unterbricht und auf `Thread2` umschaltet, ist die Reihenfolge der Elemente `Cruel` und `Hello` unbestimmt. Bewirkt einer der Threads, daß die Größe des internen Arrays verändert wird, können viele merkwürdige Dinge geschehen. Ausnahmen vom Typ `NullPointerException` oder `ArrayIndexOutOfBoundsException` sind ebenso möglich, wie der Verlust eines der beiden hinzugefügten Elemente. Unsynchronisierter Zugriff durch mehrere Threads kann alle Arten von unvorhersagbaren Schwierigkeiten mit sich bringen.

Bemerkung: Der Grund für die Betonung, daß der Threadscheduler die Verarbeitung der Threads *in der Mitte* der Zeilen 1 beziehungsweise 2 umschaltet, besteht darin, daß die `add()`-Methoden der Klassen `Vector` und `ArrayList` mehrere Operationen durchführen, bevor das übergebene Element tatsächlich in die unterliegende Datenstruktur eingesetzt wird.

---

[152] Die Klasse `Vector` ist der Klasse `ArrayList` ebenso wenig generell überlegen wie umgekehrt. Es gibt Situationen, in denen Sie auf die Unkosten der Synchronisierung einer Kollektion verzichten können und wiederum Situationen, in denen die Synchronisierung unbedingt erforderlich ist. Beispielsweise könnte eine lokale Variable ein `ArrayList`-Objekt referenzieren. Da das Objekt nur im Geltungsbereich dieser Methode existiert, hat kein anderer Thread die Möglichkeit, das Objekt zu modifizieren.

[153] Wir empfehlen auf `Vector` zu verzichten, wann immer es möglich ist. Beispielsweise könnte ein unerfahrener Entwickler fälschlicherweise glauben, daß das Verwenden eines `Vector`-Objektes eine andere als die tatsächliche Art von Threadsicherheit garantiert. Ein Entwickler könnte zu einem späteren Zeitpunkt `Vector` durch eine andere Klasse ersetzen, ohne zu bemerken, daß die Synchronisierung hier erforderlich ist.

[154] Synchronisierte Klassen und unsynchronisierte Klassen haben jeweils ihren individuellen Verwendungszweck. Ein Teil Ihrer Prüfungsvorbereitung zum *Sun Certified Java Developer* besteht darin, zu verstehen, welches der jeweilige Zweck ist.

4.3.4.2 Synchronisierte Blöcke und Methoden

[155] Die obige Methode `myVector.add(Object)` ist zwar eine synchronisierte Operation, aber die interne Synchronisierung des `Vector`-Objektes garantiert *nicht*, daß kein anderer Thread die Elemente des von `myVector` referenzierten Objektes löscht, bevor die nächste Anweisung Ihrer Methode ausgeführt wird. Beispiel:

```
1. public boolean addDVD(DVD dvd) {
2.     for (int i = 0; i < myVector.length(); i++) {
3.         doSomethingWith(myVector.get(i));
4.     }
5. }
```

Durch das Synchronisieren einer Methode beziehungsweise eines Blocks von Anweisungen (siehe unten) gewährleisten Sie, daß kein anderer Thread eine Datenstruktur während der Verarbeitung einer Abfolge von Anweisungen verfälschen kann.

[156] Während Sie sicher sein können, daß Zeile 2 während ihrer Verarbeitung nicht unterbrochen wird, gibt es keine Garantie dafür, daß nicht ein anderer Thread die Elemente des von `myVector` referenzierten `Vector`-Objektes löscht, bevor Zeile 3 verarbeitet wird. Der Threadscheduler könnte die Verarbeitung des aktuellen Threads zwischen Zeile 2 und 3 unterbrechen und der nächste zur

Verarbeitung ausgewählte Thread das von `myVector` referenzierte Objekt unbrauchbar machen. Die Gefahr ist zwar gering, aber grundsätzlich vorhanden.

[157] Seien Sie sich dieser Risiken bewußt, insbesondere dann, wenn Sie sich dafür entscheiden, sie in Kauf zu nehmen: Es ist möglich, daß ein anderer Thread zwischen zwei unsynchronisierte Zeilen geschaltet wird, auch wenn beide Zeilen synchronisierte Methoden aufrufen.

[158] Es gibt zwei Möglichkeiten, um zu gewährleisten, daß stets nur ein Thread Zugriff auf eine Abfolge von Anweisungen erhält. Erstens können Sie die Methode synchronisieren, die die Anweisungen enthält:

```
1. public synchronized boolean addDVD(DVD dvd) {
2.     for (int i = 0; i < myVector.length(); i++) {
3.         doSomethingWith(myVector.get(i));
4.     }
5. }
```

Somit kann kein anderer Thread eine synchronisierte Methode des Objektes aufrufen, zu dem die Methode `addDVD()` gehört, bevor die Verarbeitung dieser Methode beendet ist. Je nach Aufbau Ihrer Klasse kann diese Synchronisierung genügen.

[159] Zweitens können Sie die Anweisungen im Methodenkörper bezüglich des von `myVector` referenzierten Objektes synchronisieren:

```
1. public boolean addDVD(DVD dvd) {
2.     synchronized (myVector) {
3.         for (int i = 0; i < myVector.length(); i++) {
4.             doSomethingWith(myVector.get(i));
5.         }
6.     }
7. }
```

[160] Dadurch wird jede andere Methode während der Verarbeitungsdauer der Zeilen 3–6 daran gehindert, das von `myVector` referenzierte Objekt zu modifizieren. Diese Gewährleistung bleibt sogar bestehen, wenn die Verarbeitung Ihres Threads während dieser Zeilen vom Threadscheduler unterbrochen wird.

[161] Der Zugriff durch stets nur einen einzigen Thread ist nur dann garantiert, wenn alle erforderlichen Methoden entsprechend synchronisiert sind. Eine nicht bezüglich des von `myVector` referenzierten Objektes synchronisierte Methode ist nicht an das „Protokoll“ gebunden und kann die Threadsicherheit aushebeln.

[162] Dies ist, nebenbei bemerkt, eine Rechtfertigung für den kontrollierten Zugriff auf die Felder einer Klasse durch Kapselung. Sind alle Felder als `private` deklariert, so können Sie den Zugriff auf sensitive Daten durch Synchronisierung aller relevanten Methoden steuern. Genau dies tut die Klasse `Vector`. Synchronisierte Klassen, wie `Vector`, synchronisieren nicht etwa ihre internen Datenstrukturen, sondern sie synchronisieren die Methoden, die diese Daten abfragen beziehungsweise ändern. Sun Microsystems empfiehlt und unterstützt Kapselung, wie am Beispiel der JavaBeans besonders deutlich zu sehen ist.

4.3.5 Threads und Swing

[163] Swing-Komponenten sind in der Regel nicht threadsicher. Swing verwendet einen einzigen Thread, um die vom unterliegenden Betriebssystem verursachten Ereignisse von der graphischen Benutzeroberfläche zu verarbeiten, nämlich den sogenannten Ereignisbehandlerthread (*event dispatcher thread*). Das heißt, daß Swing-Komponenten im allgemeinen nicht threadsicher zu sein

brauchen, da sie typischerweise nur dem Zugriff durch einen einzigen Thread ausgesetzt sind. Die Verarbeitung eines Ereignisses, etwa das Anklicken einer Schaltfläche, bewirkt, daß die übrigen Ereignisse solange warten müssen, bis das erstere Ereignis vollständig verarbeitet ist. Bei einem aufwendigen Ereignisbehandler kann Ihre graphische Benutzeroberfläche träge werden.

[164] Die Kernaussage lautet, daß die Geschwindigkeit mit der Ihre graphische Benutzeroberfläche Ereignisse bearbeitet, bei einer aufwendigen Operation deutlich zurückgehen kann. Eine solche Verlangsamung kann in einer Prüfungsaufgabe für die Zertifizierung zum *Sun Certified Java Developer* durchaus vertretbar sein. Sie müssen sich dieses Effektes aber bewußt sein und ihn gegebenenfalls dokumentieren. Verlangt Ihre Anleitung allerdings, daß die graphische Benutzeroberfläche schnell reagiert, so müssen Sie die Verarbeitung aufwendiger Operationen in einem eigenen Thread in Betracht ziehen, sich also über Threadsicherheit und Synchronisierung Gedanken machen.

4.3.5.1 Grundlagen

[165] Eine Swing-Komponente deren `setVisible(true)`- beziehungsweise `pack()`-Methode bereits aufgerufen wurde, sollte ausschließlich über den Ereignisbehandlerthread aktualisiert werden. Dadurch verhindern Sie Wettlaufsituationen infolge verschiedener Anfragen an das `javax.swing.RepaintManager`-Objekt.

[166] In der Regel müssen Sie sich bei einer Swing-basierten graphischen Benutzeroberfläche nur über Threads Gedanken machen, wenn Sie Swing-Komponenten über einen anderen als den Ereignisbehandlerthread aktualisieren. Dieser Fall tritt zum Beispiel dann ein, wenn Ihre graphische Benutzeroberfläche schnell reagieren muß und nicht auf entfernte Methodenaufrufe über ein Netzwerk per RMI oder Sockets warten kann.

[167] Vergewissern Sie sich mit aller Sorgfalt, daß Sie sich das Warten auf den oder die entfernten Methodenaufrufe wirklich nicht leisten können, bevor Sie diesen Weg wählen: Der Ansatz ist langwierig, fehleranfällig und verkompliziert Ihren Weg zum *Sun Certified Java Developer* übermäßig. Falls Sie sich für diesen Weg entscheiden, sollten Sie die zeitaufwendigen Methodenaufrufe mit Hilfe eines separaten Threads (*worker threads*) bewerkstelligen und die Ergebnisse mit Hilfe des Ereignisbehandlerthreads in die graphische Benutzeroberfläche übernehmen. Sie finden ein hervorragendes Beispiel für diesen Ansatz unter der Internetadresse *http://java.sun.com/products/jfc/tsc/articles/threads/threads2.html*.

Warnung: Sie dürfen die Klasse `javax.swing.SwingWorker` in Ihrer Prüfungsaufgabe nicht direkt verwenden. In der Aufgabenstellung ist festgelegt, daß Sie nur selbstgeschriebenen Quelltext einreichen dürfen, sich also durch Ableiten einer Klasse von `SwingWorker` disqualifizieren würden. Sie können die Klasse `SwingWorker` allerdings heranziehen, um die benötigten Prinzipien zu verstehen, bevor Sie Ihre eigene Lösung implementieren.

---

[168] Das Aufrufen der verschiedenen Änderungsmethoden einer Swing-Komponente ist *kein* threadsicherer Ansatz, um die entsprechenden Ereignisse in die Warteschlange des Ereignisbehandlerthreads einzusetzen. Das Aufrufen einer Änderungsmethode bewirkt zwar, daß ein Ereignis erzeugt wird, aber der Ereignisbehandlerthread kümmert sich nicht immer um derartige Ereignisse.

4.3.5.2 Aktualisieren von Komponenten per Ereignisbehandlerthread

[169] Sie registrieren Ereignisse in der Warteschlange des Ereignisbehandlerthreads, indem Sie die `SwingUtilities`-Methoden `invokeAndWait()` beziehungsweise `invokeLater()` aufrufen. Die Ver-

arbeitung der ersten Methode (`invokeAndWait()`) wird solange blockiert, bis das Ereignis ausgelöst werden kann. Die Verarbeitung der zweiten Methode (`invokeLater()`) wird nicht blockiert, sondern ihr Argument lediglich der Warteschlange der zu erzeugenden Ereignisse hinzugefügt. Beide Methoden erwarten ein Argument vom Typ *`Runnable`*. Der Ereignisbehandlerthread ruft die `run()`-Methode des Arguments auf.

4.4 Threads: Bewährte Verfahren

[170] Die folgenden praktischen Ratschläge sollen Ihnen dabei helfen, die Schattenseiten der Threadprogrammierung zu umgehen:

- Vermeiden Sie das Schachteln von Sperren. Wenn Ihr Ausführungspfad zwei bezüglich verschiedener Objekte synchronisierte Blöcke durchläuft, sind Sie eventuell auf dem falschen Weg.
- Vermeiden Sie Multithreading bei Swing, falls möglich. Sie können als Gegenargument anführen, daß die Verwendung mehrerer Threads „hinter“ einer Swing-Oberfläche die Verarbeitung von Ereignisbehandlern nur *scheinbar* beschleunigt. Schnelles Reagieren der graphischen Benutzeroberfläche bedeutet nicht, daß auch die darunter liegende Anwendung schnell reagiert. Wollen Sie dem Anwender tatsächlich vorgaukeln, daß eine Operation beendet ist, die tatsächlich noch verarbeitet wird? Dient die Illusion einem sinnvollen Zweck?
- Führt ein Thread eine zeitaufwendige Operation aus, so ist es sinnvoll, gelegentlich die `yield()`-Methode aufzurufen. Verarbeitet ein Thread beispielsweise eine langwierige Operation und der Benutzer entscheidet sich, die Verarbeitung abzubrechen, so wird das Ereignis „Verarbeitung abbrechen“ eventuell erst bemerkt, nachdem die Verarbeitung des aufwendigen Threads bereits beendet ist. Durch gelegentliche `yield()`-Aufrufe geben Sie anderen Threads eine Chance, zu einem Zeitpunkt verarbeitet zu werden, der für den Zustand des Threads günstig ist, der die `yield()`-Methode aufgerufen hat. Heben Sie nach Möglichkeit am besten alle Sperren auf, bevor Sie `yield()` aufrufen.
- Eine Methode, die den Zustand eines Objektes nicht verändert, braucht nicht synchronisiert zu werden. Synchronisieren Sie eine Methode nicht, wenn sie die Felder ihres Objektes, die von ihrem Objekt referenzierten Objekte und die als Argumente übergebenen Objekte nicht modifiziert.
- Gehen Sie sparsam mit der Synchronisierung um. Achten Sie darauf, daß Sie genau wissen, warum Sie eine Methode oder den Zugriff auf eine Datenstruktur synchronisieren. Wenn Sie die Notwendigkeit einer Synchronisierung nicht für sich selbst befriedigend erklären können, überdenken Sie die Ausgangssituation noch einmal von Anfang an.
- Der Zugriff auf unveränderliche Objekte (zum Beispiel `String`- und `Integer`-Objekte) braucht nicht synchronisiert zu werden. Das gilt vor allem für Klassen- und Objektfelder, die solche Objekte referenzieren, da der Zustand eines unveränderlichen Objektes nicht modifiziert werden kann. Der Zugriff auf ein Feld selbst, das ein unveränderliches Objekt referenziert, muß dagegen unter Umständen synchronisiert werden, da der Feldinhalt verändert werden kann.
- Vermeiden Sie unnötiges Multithreading. Ist die Anzahl der Threads im blockierten Zustand gering, so ist eventell ein Ansatz schneller, der stets nur eine einzige Anweisung verarbeitet.
- When working with Swing, feel free to create and insert components into any container that hasn't been realized yet. To be realized, a component must be able to receive paint

or validation events — this means, before `show()`, `setVisible(true)` or `pack()` has been called on the component.

- Interpretieren Sie das Wesen intern synchronisierter Klassen wie `Vector` nicht falsch. Eine intern synchronisierte Klasse verspricht nicht mehr, als daß ihre Methoden atomar sind. Führt ein Thread die Anweisung `myVector.add("test")` aus und wird ein anderer Thread zwischengeschaltet, der die Anweisung `myVector.contains("test")` ausführt, so gibt die `contains()`-Methode des zweiten Threads `true` zurück. Dieses Verhalten ist zum Beispiel bei einem `ArrayList`-Objekt *nicht* garantiert. In der Regel sind synchronisierte Methoden erforderlich, um mit solchen Objekten zu kommunizieren.

- Setzen Sie so wenige Anweisungen wie möglich in einen synchronisierten Block, da kein anderer Thread das Objekt sperren kann und die gleichzeitige Verarbeitung verschiedener Threads behindert wird.

- Vermeiden Sie Threadprioritäten als Mechanismus zur Threadsteuerung. Verschiedene Betriebssysteme verwenden unterschiedliche Algorithmen, um Prioritäten von Threads auszuwerten und unterschiedliche Laufzeitumgebungen verwenden wiederum unterschiedliche Algorithmen. Dadurch kann sich Ihr Programm auf verschiedenen Plattform völlig unterschiedlich verhalten. Am besten vermeiden Sie die Uneindeutigkeit von Threadprioritäten, insbesondere in Ihrer Prüfungsaufgabe für die Zertifizierung zum *Sun Certified Java Developer*.

- Die `run()`-Methode eines Threads wird *nicht* synchronisiert. Das Synchronisieren der `run()`-Methode eines Threads kann viele Komplikationen verursachen.

- Verwenden Sie eine `while`-Schleife statt einer `if`-Klausel, um in einem synchronisierten Block eine Bedingung abzufragen. Falls die Verarbeitung Ihres Threads in der Mitte der `if`-Anweisung unterbrochen wird, wird seine Verarbeitung später an genau dieser Stelle fortgesetzt. Wider der Intuition ist dies keine gute Lösung, weil die Bedingung, die vor dem Aussetzen des ersten Threads wahr war durch die Verarbeitung des zweiten Threads zwischenzeitlich falsch geworden sein könnte.

- Die Bedingung einer `while`-Schleife wird dagegen erneut geprüft, bevor Ihr Thread den Schleifenkörper verläßt. Relevante Änderungen am Zustand des Objektes werden somit erkannt. Beispiel:

```
synchronized (lockRecords) {
    while (lockedRecords.contains(upc)) {
        lockedRecords.wait();
    }
    // do stuff
    lockedRecords.notifyAll();
}
```

- Verwenden Sie `notifyAll()` anstelle von `notify()`. Die `notify()`-Methode weckt nur einen einzigen wartenden Thread auf, während `notifyAll()` alle wartenden Threads aufweckt.

4.5 Zusammenfassung

[171] In diesem Kapitel haben wir einige der Möglichkeiten besprochen, die Ihnen die Threadprogrammierung bietet und einige Probleme behandelt, die in diesem Kontext auftreten können. Wir haben Threadsicherheit, Threads und Swing, den blockierten und den Wartezustand von Threads und viele andere Themen diskutiert.

[172] Sie verstehen das Wesen von Threads am besten, wenn Sie eine Lösung zu einer Aufgabe ausarbeiten, Voraussagen über deren Funktionsweise treffen und Ihre Voraussagen anschließend mit Hilfe einer passenden `Thread`-Methode überprüfen. Die `Thread`-Methoden `holdsLock(Object)`, `getState()` und `getStackTrace()` sind sehr praktisch, um zu prüfen, was ein anderer Thread tut. Wenn sich Ihre Threads nicht erwartungsgemäß verhalten, dokumentieren Sie Ihre Annahmen (vorzugsweise schriftlich) und untersuchen Sie eine nach der anderen. Threads sind wie Grammatik: Es gibt zwar viele Regeln, aber mit der Zeit entwickeln Sie ein Gefühl dafür, was funktioniert und was nicht.

[173] Zögern Sie beim Durcharbeiten der folgenden Kapitel nicht, zum Kapitel „Threads“ zurückzublättern, sofern erforderlich.

4.6 Häufige Fragen

- *Frage*: Was geschieht, wenn ein synchronisierter Block/eine synchronisierte Methode einen unsynchronisierten Block/eine unsynchronisierte Methode aufruft?

  Antwort: Der unsynchronisierte Block/die unsynchronisierte Methode wird behandelt, als sei er/sie synchronisiert.

- *Frage*: Was geschieht, wenn ein synchronisierter Block/eine synchronisierte Methode einen synchronisierten Block/eine synchronisierte Methode aufruft, dessen/deren Synchronisierung sich auf dasselbe Objekt bezieht?

  Antwort: Wenn beide Methoden bezüglich desselben Objektes synchronisiert sind, geschieht nicht ungewöhnliches. Es gibt keine doppelte Synchronisierung und es besteht keine Gefahr für eine Verklemmung. Der Thread sperrt das Objekt ein zweitesmal. Die zweite Sperre wird zusammen mit der ersten aufgehoben.

- *Frage*: Bewirkt das Sperren eines Objektes, daß die von seinen Feldern referenzierten Objekte ebenfalls gesperrt werden.

  Antwort: Nein! Das wäre ein enormer Aufwand für die Laufzeitumgebung, da nicht nur die von den Feldern des Objektes referenzierten Objekte gesperrt werden müßten, sondern auch die von den Feldern *dieser* Objekte referenzierten Objekte .... Java verlangt, daß Sie sich dieser Tatsache bewußt sind und nur die tatsächlich erforderlichen Objekte sperren.

- *Frage*: Um welchen Faktor ist eine synchronisierte Methode langsamer, verglichen mit einer unsynchronisierten Methode?

  Antwort: Diese Frage ist schwierig zu beantworten. Nicht auf wissenschaftlicher Basis durchgeführte Tests scheinen zu belegen, daß eine synchronisierte Methode zwischen 1.5 und 150 mal langsamer verarbeitet wird, als eine unsynchronisierte Methode. Die Verarbeitungsgeschwindigkeit hängt von der „Wettbewerbssituation“ ab. Der Bedarf an zusätzlicher Laufzeit durch Synchronisieren einer Methode, die stets von nur einem Thread aufgerufen wird, ist unter Umständen vernachlässigbar.

- *Frage*: Können Daten synchronisiert werden?

  Antwort: Nein. Nur der Zugriff auf Daten.

- *Frage*: Sind `Vector`-Objekte nicht synchronisierte Daten?

  Antwort: Nein. Ein `Vector`-Objekt besitzt *synchronisierte Methoden* zum Zugriff auf ihre Daten. Das ist ein wesentlicher Unterschied.

- *Frage*: Worin besteht der Unterschied zwischen der Synchronisierung bezüglich eines von einem statischen Feld referenzierten Objektes und der Synchronisierung bezüglich eines von einem Objektfeld referenzierten Objektes?

  Antwort: Es gibt keinen Unterschied. Ihr Thread sperrt ein Objekt. Die Synchronisierung bezüglich eines von einem statischen Feld referenzierten Objektes wirkt sich auf alle Objekte aus, die ein statisches Feld besitzen, welches auf das gesperrte Objekt verweist.

- *Frage*: Was bedeutet die Aussage „Die Klasse `Vector` ist synchronisiert"?

  Antwort: Alle öffentlichen Methoden einer synchronisierten Klasse, die gemeinsam genutzte Daten abfragen oder ändern, sind synchronisiert. Das Aufrufen einer Methode eines synchronisierten Objektes ist eine atomare Operation. Jeder andere Thread, der eine synchronisierte Methode dieses Objektes aufruft, wird solange blockiert bis Ihr Thread seine Anweisungen bezüglich dieses Objektes vollständig verarbeitet hat.

- *Frage*: Angenommen, eine Methode ist bezüglich eines von einem Feld referenzierten Objektes synchronisiert und einen andere Methode bezüglich der `this`-Referenz. Beide Methoden beinflussen sich gegenseitig.

  Antwort: Sie sperren zwei *verschiedene* Objekte, nämlich das von `this` und das von dem Feld referenzierte Objekt. *In dieser Situation ist Sorgfalt sehr wichtig*, sonst droht die Gefahr einer Verklemmung. Überlegen Sie sich gewissenhaft, ob die Sperrung zweier verschiedener Objekte tatsächlich notwendig ist, oder ob es ausreicht, nur eines der Objekte zu sperren.

- *Frage*: Wie kann ich einen zu Ende verarbeiteten Thread (`TERMINATED`) neu starten?

  Antwort: Überhaupt nicht. Sie müssen einen neuen Thread erzeugen.

- *Frage*: Bei den meisten Beispielen wartet der `main`-Thread implizit bis alle erzeugten Threads beendet sind, bevor die Anwendung selbst beendet wird. Gibt es eine Möglichkeit, daß die Anwendung zu beenden, ohne auf das Ende eines Threads zu warten?

  Antwort: Ja. Sie können explizit die `System`-Methode `exit()` aufrufen, oder vor dem `start()`-Aufruf die `Thread`-Methode `setDaemon(true)` des Threads aufrufen, auf den Sie nicht warten wollen. Das Eisverkäuferbeispiel mit dem `IceCreamMan`-Thread zeigt, wie ein Thread als Hintergrundthread deklariert wird (Seite 76).

Kapitel 5

Die Klasse DvdDatabase

[0/1] In diesem Kapitel beginnen wir mit der Arbeit an unserem Beispielprojekt *Denny's DVDs*. Wir entwickeln die Klasse `sampleproject.db.DvdDatabase`, die das in Kapitel 3 beschriebene Interface *`DBClient`* implementiert. Es ist logisch mit `DvdDatabase` zu beginnen, da sowohl der Client als auch der Server diese Klasse benötigen. Wir machen dabei Gebrauch von verschiedenen Entwurfsmustern und generischen Kollektionen (siehe Kapitel 2).

[2] Das Beispielprojekt und seine Implementierung weichen stellenweise von den typischen Eigenschaften und Anforderungen einer typischen Prüfungsaufgabe von Sun Microsystems ab. Dies gilt in besonderem Maße für die in diesem Kapitel diskutierten Klassen. Einerseits ist der Quelltext in diesem Bereich unseres Beispielprojektes ausführlicher als bei Ihrer Prüfungsaufgabe verlangt. Andererseits gebrauchen wir Abkürzungen, die Sie bei Ihrer Prüfungsaufgabe nicht verwenden dürfen. Die Methoden unserer Klassen und Interfaces können beispielsweise Ausnahmen vom Typ `java.io.IOException` auswerfen, wodurch die Entwicklung erheblich vereinfacht wird. Andererseits ist unsere Lösung aufwendiger, da wir Sperren mit Zeitüberschreitung und *`java.util.concurrent.locks.ReadWriteLock`*-Objekte verwenden, um die parallele Verarbeitung mehrerer Threads zu unterstützen. Dennoch finden Sie alle Informationen in diesem Kapitel, die Sie benötigen, um eine gute Lösung für *Ihre* Prüfungsaufgabe zu entwickeln.

[3] Darüberhinaus diskutieren wir einige Dinge, die für die Lösung Ihrer Prüfungsaufgabe zwar nicht erforderlich sind, sie aber professioneller machen, beispielsweise das Zwischenspeichern von Daten, das Vermeiden von Verklemmungen, den Umgang mit clientseitigen Verbindungsabbrüchen, und die Verwendung vieler Sperrobjekte, um die Prozessorauslastung durch Threads zu reduzieren, die versuchen, einen Datensatz logisch zu reservieren (sperren).

5.1 Die Hilfsklassen der Klasse DvdDatabase

[4] Die Klasse `DvdDatabase` bezieht sich auf weitere Klassen, etwa für Eingaben, Ausgaben und Ausnahmen. Wir entwickeln zunächst diese Hilfsklassen, bevor wir `DvdDatabase` selbst anlegen.

5.1.1 Die Klasse DVD und das Entwurfsmuster Value-Objekt

[5] Die Klasse `DvdDatabase` verwendet Objekte einer Hilfsklasse namens `sampleproject.db.DVD`, die sämtliche Informationen über eine `DVD` enthalten. Derartige Objekte werden als „Value-Objekte“ bezeichnet. Das Entwurfsmuster *Value-Object* ist auch unter der Bezeichnung *Transfer-Object* bekannt, wobei „Value-Objekt“ den Verwendungszweck des Objektes beschreibt, nämlich alle Werte

eines bestimmten Objektes zu enthalten (hier alle Felder einer DVD). Die Bezeichnung „Transfer-Objekt“ beschreibt dagegen einen häufigen Anwendungsfall dieses Entwurfsmusters, nämlich das Anlegen einer Klasse, deren Objekte Daten zwischen zwei Systemen übertragen, üblicherweise zwischen einem Client und einem Server.

[6] Die Motivation zur Verwendung eines Value- beziehungsweise Transfer-Objektes ist leicht zu einzusehen: Der Quelltext ist leichter lesbar und die Performanz der Anwendung wird verbessert. Der Quelltext bezieht sich auf die Felder eines einzigen Objekttyps, so daß der Zusammenhang zwischen den einzelnen Feldern deutlicher erkennbar wird. Da Sie einer Methode ein ganzes Value-Objekt übergeben beziehungsweise aus einer Methode zurückgeben können, wird der Methodenaufruf performanter verarbeitet, als wenn alle Felder einzeln übergeben werden müssen. Das zahlt sich besonders dann aus, wenn Value-Objekte über ein Netzwerk übertragen werden, da nur ein einziger Methodenaufruf über das Netzwerk erforderlich ist, statt mehrerer Aufrufe, um jedes Feld separat zu übertragen.

[7] Wir besprechen nun die einzelnen Abschnitte der Klasse `DVD`. Da sich Teile des Quelltextes wiederholen, geben wir nicht den vollständigen Quelltext wieder, sondern nur die wesentlichen Abschnitte. Sie können den vollständigen Quelltext der Klasse `DVD` aus dem „Source Code“-Abschnitt der Apress-Website (*http://www.apress.com/book/sourcecode*) herunterladen. In Kapitel 9 finden Sie detaillierte Angaben darüber, was Sie herunterladen können und wie der Quelltext der Beispielanwendung *Denny's DVDs* in Packages organisiert ist beziehungsweise übersetzt werden kann. Die Zeilennummerierung im Buch entspricht den tatsächlichen Zeilennummern im Quelltext (Lücken in der Folge der Zeilennummern repräsentieren Kommentare oder sich wiederholende Anweisungen).

```
01. package sampleproject.db;
02.
03. import java.io.*;
04. import java.util.*;
05. import java.util.logging.*;
06.
...
13. public class DVD implements Serializable {
...
19.     private static final long serialVersionUID = 5165L;
```

[8] Die Objekte der Klasse `DVD` dienen in der Beispielanwendung dazu, die Informationen, die eine DVD repräsentieren, zwischen Client und Server auszutauschen, wobei sich Client und Server auf verschiedenen Rechnern befinden können. Damit dieser Datenaustausch funktioniert, müssen Client und Server darin übereinstimmen, was ein `DVD`-Objekt ist. Verwendet eine Partei ein andere interne Datenstruktur (zum Beispiel könnte der Server das Verleihdatum als `Date`-Objekt betrachten, der Client dagegen als `String`), so muß das gesamte System der Datenübertragung in Frage gestellt werden. Wenn Sie Glück haben, wird eine Ausnahme vom Typ `ClassCastException` ausgeworfen. Wenn Sie kein Glück haben, arbeiten Client und Server mit den inkonsistenten Daten weiter und verfälschen dadurch den Inhalt der Datenbankdatei.

[9] Java verfügt über einen Mechanismus, um zu prüfen, ob ein serialisiertes Objekt einer gegebenen Struktur entspricht: Das `serialVersionUID`-Feld. Versucht eine Klasse, ein serialisiertes Objekt mit nicht übereinstimmender Versionsbezeichnung zu deserialisieren, so wirft die Laufzeitumgebung eine Ausnahme vom Typ `java.io.InvalidClassException` aus.

Bemerkung: Die Serialisierung wird im ersten Abschnitt von Kapitel 6 in allen Einzelheiten behandelt.

---

[10] Zeile 19 deklariert ein `static final long` Feld namens `serialVersionUID`. Beachten Sie, daß die Modifikatoren `static`, `final` und `long` zwingend erforderlich sind. Jeder Zugriffskontrollmodifikator ist erlaubt, insbesondere `private` (siehe oben). Sie können anhand des `serialVersionUID`-Feldes entscheiden, ob die clientseitige Version einer serialisierbaren Klasse nach einer Änderung neu übersetzt werden muß. Angenommen, wir ergänzen die Klasse `DVD` um ein Feld, das die Anzahl der Emmy-Nominierungen der DVD angibt. Dann wären die clientseitigen Klassen, die vor dem Hinzufügen dieses Feldes übersetzt wurden, nicht von dieser Änderung betroffen, sondern könnten nach wie vor jedes Feld abfragen oder ändern, das sie zuvor erreichen konnten. Dies würde jedoch nicht gelten, wenn wir ein Feld löschen, selbst wenn die clientseitigen Klassen dieses Feld zur Zeit nicht verwenden. Das Feld *könnte* benutzt werden, so daß die Änderung der Klasse das erneute Übersetzen der clientseitige Klassen erforderlich macht.

[11] Wenn Sie in einer Ihrer serialisierbaren Klassen kein `serialVersionUID`-Feld anlegen, generiert der Compiler eine Versionsbezeichnung für diese Klasse, die von verschiedenen Eigenschaften der Klasse abhängt, zum Beispiel vom Klassennamen sowie den Namen der Methoden und Felder. Ändert sich eines dieser Merkmale, so ändert sich auch die Versionsbezeichnung der Klasse, also auch dann, wenn wir, wie im vorigen Absatz beschrieben, ein unbedeutendes zusätzliches Feld anlegen. Es ist daher stets ratsam, daß Sie selbst ein `serialVersionUID`-Feld anlegen und pflegen. (Falls Sie sich wundern: Der Wert unseres `serialVersionUID`-Feldes, 5165, besteht aus den letzten vier Ziffern der ISBN dieses Buches.)

Tipp: Wenn Sie mehr über Serialisierung von Objekten erfahren möchten, können Sie in der Dokumentation von Sun Microsystems nachlesen: *http://java.sun.com/j2se/1.5.0/docs/guide/serialization/*.

---

[12] Anschließend deklariert die `DVD`-Klasse alle Felder, die wir zur Beschreibung einer DVD verwenden wollen. Wiederum wurden die Kommentare fortgelassen, so daß die Zeilennummern nicht in lückenloser Reihenfolge notiert sind:

```
89.  private String upc = "";                  // The record UPC identifier
94.  private String name = "";                 // Holds the movie title
100. private String composer = "";             // The music composer
106. private String director = "";             // The director's name
112. private String leadActor = "";            // The lead actor's name
118. private String supportingActor = "";      // The supporting actor's name
123. private String year = "";                 // The movie's release date
129. private int copy = 1;                     // The number of Dvds in stock
```

Bemerkung: Der UPC (Universal Product Code) ist eine eindeutige Bezeichnung die jedem im Einzelhandel vertriebenen Produkt in den Vereinigten Staaten und in Kanada zugewiesen wird. Das japanische Äquivalent ist der JAN-Code. Der nächste Standard und ein weiterer Kandidat für eine weltweit gültige Kennzeichnung ist der europäische EAN-Code. Wir verwenden in diesem Buch den originalen UPC, um zu garantieren, daß jeder Datensatz in unserer Datenbankdatei einen eindeutigen Index hat. (Siehe auch **Bemerkung**-Box auf Seite 57.)

---

[13] Die Klasse `DVD` besitzt drei Konstruktoren, nämlich einen (argumentlosen) Standardkonstruktor (bei serialisierbaren Klassen erforderlich), einen Konstruktor für DVDs, von denen nur ein Exemplar in der Datenbankdatei registriert ist und einen Konstruktor für DVDs mit mehreren registrierten Exemplaren:

```
140. public DVD() {
141.     log.finer("DVD empty constructor called");
```

```
142. }
...
157. public DVD(String upc, String name, String composer,
158.            String director, String lead, String supportActor,
159.            String year) {
160.     this(upc, name, composer, director, lead, supportActor, year, 1);
161. } 162.
163. /**
164.   * Creates an instance of this object with a specified list of initial
165.   * values.
166.   *
167.   * @param upc The UPC value of the Dvd.
168.   * @param name The title of the Dvd.
169.   * @param composer The name of the movie's composer.
170.   * @param director The name of the movie's director.
171.   * @param leadActor The name of the movie's leading actor.
172.   * @param supportActor The name of the movie's supporting actor.
173.   * @param year The date the of the movie's release (yyyy-mm-dd).
174.   * @param copies The number of copies in stock.
175.   */
176. public DVD(String upc, String name, String composer, String director,
177.            String leadActor, String supportActor, String year,
178.            int copies) {
179.     log.entering("DVD", "DVD",
180.                  new Object[]{upc, name, composer, director, leadActor,
181.                               supportingActor, year, copies});
182.     this.upc = upc;
183.     this.name = name;
184.     this.composer = composer;
185.     this.director = director;
186.     this.leadActor = leadActor;
187.     this.supportingActor = supportActor;
188.     this.year = year;
189.     this.copy = copies;
190.     log.exiting("DVD", "DVD");
191. }
```

[14] Beim dritten Konstruktor sind die Kommentare nicht ausgeblendet, um zu zeigen, wie Parameter in Dokumentationskommentaren (Javadoc) deklariert werden. Wie Sie sehen, muß der Typ eines Parameters nicht angegeben werden.

[15] Beachten Sie, daß der zweite Konstruktor keine Protokollmeldungen erzeugt. Da der zweite Konstruktor unmittelbar den dritten aufruft, können wir die dort implementierte `Logger`-Funktionalität nutzen.

[16] Zu jedem in den Zeilen 89–129 deklarierten Felder gehört je eine Abfrage- und eine Änderungsmethode. Die Abfrage- und Änderungsmethoden für das Feld `composer` (Zeile 100) heißen zum Beispiel `getComposer()` und `setComposer()`:

```
199. public String getComposer() {
200.     log.entering("DVD", "getComposer");
201.     log.exiting("DVD", "getComposer", this.composer);
202.     return this.composer;
203. }
...
211. public void setComposer(String composer) {
212.     log.entering("DVD", "setComposer", composer);
213.     this.composer = composer;
```

```
214.     log.exiting("DVD", "setComposer", this.composer);
215. }
```

[17] Die Verwendung von Abfrage- und Änderungsmethoden, im Englischen wegen der ersten Silbe `get` beziehungsweise `set` auch als „Getter“ und „Setter“ bezeichnet, gestattet uns, das Feld selbst zu verbergen und den Zugriff nur mit Hilfe dieser Methoden zu erlauben und zu kontrollieren. Es ist zum Beispiel sinnvoll, die Anzahl der Exemplare einer DVD auf einen ganzzahligen *nicht negativen* Wert einzuschränken. Bei unmittelbarem Zugriff könnte dem Feld ein beliebiger Wert zugewiesen werden, also auch ein negativer Wert. Bei einer Änderungsmethode für die Anzahl der Exemplare können wir dagegen eine entsprechende Prüfung vorschalten.

Bemerkung: Wir rufen in Zeile 200 die `Logger`-Methode `entering()` und unmittelbar darauf in Zeile 201 bereits die `Logger`-Methode `exiting()` auf. Wir mußten wiederum eine Design-Entscheidung fällen: Schreiben wir eine eigene Protokollmeldung mit Hilfe der `Logger`-Methode `finer()` oder wählen wir die Methoden `entering()` und `exiting()`, obwohl sie eigentlich keine nützlichen Informationen liefern? Auch hier gibt es keine einzige „richtige“ Lösung. Der Nachteil unserer Entscheidung besteht darin, daß wir unnötige aussagelose Meldungen protokollieren. Der Vorteil besteht allerdings darin, daß wir stets dasselbe standardisierte Protokollformat verwenden (`entering()`/`exiting()` für alle Abfrage- und Änderungsmethoden). Würden wir dagegen für jede Abfrage- und Änderungsmethode eine individuelle Protokollmeldung schreiben, so würden sich die Formate dieser Meldungen wahrscheinlich mit der Zeit von Methode zu Methode unterscheiden. Falls zu einem späteren Zeitpunkt die Protokollmeldung zu einer Anfrage- oder Änderungsmethode erweitert werden muß, kann die Änderung direkt zwischen den Methoden `entering()` und `exiting()` eingesetzt werden. Eine selbstentwickelte Protokollmeldung würde wahrscheinlich entfernt und durch eine standardisierte Meldung ersetzt werden müssen.

---

[18] Das Protokollieren der Abfrage- und Änderungsmethoden ist eventuell zuviel des Guten, stellt aber ein wichtiges Werkzeug zur Fehlersuche dar: Protokollmeldungen sind sehr hilfreich, um nachzuvollziehen, welche Argumente einer Methode übergeben werden und wie das Ergebnis des Methodenaufrufs lautet. Durch das Protokollieren des Parameters `composer` in Zeile 212 erhalten wir einen Eintrag mit dem Parameterwert, der `setComposer()` beim Aufruf übergeben wurde und durch `this.composer` in Zeile 214 einen weiteren Eintrag der das Ergebnis des Aufrufs der `setComposer()`-Methode dokumentiert. Auch wenn die Methode später geändert wird (zum Beispiel, um eine Anweisung erweiter wird, die dafür sorgt, daß jeder Namensbestandteil des Komponisten mit einem Großbuchstaben beginnt), geben die Protokolleinträge noch immer den Eingabeparameter und das Ergebnis an.

[19] Die übrigen Abfrage- und Änderungsmethoden werden nicht gezeigt. Sie folgen aber dem Schema von `getComposer()` und `setComposer()`.

Tipp: Die meisten integrierten Entwicklungsumgebungen können Abfrage- und Änderungsmethoden mit rudimentären Vorlagen für Dokumentationskommentare (Javadoc) generieren. Anschließend können Sie die einzelnen Methoden ausprogrammieren und die Dokumentationskommentare ausfüllen.

---

[20] Wir betrachten abschließend noch, wie sich zwei `DVD`-Objekte vergleichen lassen, um entscheiden zu können, ob sie identisch sind. Würde die Beispielanwendung ausschließlich im *stand-alone*-Betriebsmodus aufgerufen, so könnten wir uns eventuell auf ein einziges `DVD`-Objekt pro registrierter DVD beschränken und den Vergleich mit Hilfe des `==`-Operators bewerkstelligen. Da im Netzwerkmodus zwei oder mehr serialisierte, von verschiedenen Clients gesendete `DVD`-Objekte ein

und dieselbe DVD repräsentieren können, scheidet der Vergleich per `==`-Operator allerdings aus. Wir überschreiben daher die von `Object` geerbte Methode `equals()`:

```
416. public boolean equals(Object aDvd) {
417.     if (!(aDvd instanceof DVD)) {
418.         return false;
419.     }
420.
421.     DVD otherDvd = (DVD) aDvd;
422.
423.     return (upc == null) ? (otherDvd.getUPC() == null)
424.                          : upc.equals(otherDvd.getUPC());
425. }
```

[21] Zeile 417 kontrolliert, ob die übergebene Referenz `aDvd` auf ein Objekt vom Typ `DVD` verweist. Nach dieser Prüfung und nur, wenn die Prüfung positiv ausgefallen ist, wandeln wir `aDvd` in den Typ `DVD` um (Zeile 421). Schließlich nutzen wir in den Zeilen 423 und 424 die Eindeutigkeit der UPCs für DVDs, um die `DVD`-Objekte zu vergleichen.

Bemerkung: Wir verwenden in den Zeilen 423 und 424 den ternären Operator, um zu ermitteln, ob die überschriebene `equals()`-Methode `true` oder `false` zurückgibt. Die Meinungen darüber, ob der ternäre Operator die Lesbarkeit verbessert oder nicht, sind geteilt. Sie müssen zu einer eigenen, eventuell vom Einzelfall abhängigen Entscheidung kommen. Wir sind der Auffassung, daß der ternäre Operator in diesem Fall die Lesbarkeit verbessert. Vergleichen Sie die Zeilen 423 und 424 mit der Alternative:

```
if (upc == null) {
    return otherDvd.getUPC() == null;
} else {
    return upc.equals(otherDvd.getUPC());
}
```

In der zweiten `return`-Anweisung können wir die `String`-Methode `equals()` sicher aufrufen, um die Inhalte der beide `upc`-Felder miteinander zu vergleichen. Wir können die `equals()`-Methode dagegen nicht vorher aufrufen, da wir nicht wissen, ob das lokale `upc`-Feld den Wert `null` enthält (in diesem Fall hätte die letzte `return`-Anweisung eine Ausnahme vom Typ `NullPointerException` ausgeworfen). Die Logik kann auch nicht vertauscht werden (`otherDvd.getUPC.equals(upc)`), da wir nicht wissen, ob das externe `upc`-Feld den Wert `null` enthält, wodurch wiederum eine `NullPointerException` hervorgerufen würde. Die `equals()`-Methode kann erst verwendet werden, nachdem wir festgestellt haben, daß mindestens eines der `upc`-Felder nicht `null` enthält.

---

[22] Würde der UPC eine DVD nicht eindeutig identifizieren, so müßten wir ein komplexeres Kriterium konstruieren, zum Beispiel können wir den UPC, den Titel des Films und den Hauptdarsteller wählen (es ist unwahrscheinlich, daß ein Schauspieler in zwei verschiedenen, aber gleichnamigen Filmen der Hauptdarsteller ist und außerdem beide DVDs übereinstimmende UPCs haben). Wir berücksichtigen dagegen die Anzahl der verfügbaren Exemplare einer DVD *nicht*, weil sie offensichtlich kein kennzeichnendes Kriterium für eine DVD ist.

[23] Wenn Sie die `equals()`-Methode überschreiben, sollten Sie auch die `hashCode()`-Methode überschreiben, da beide Methoden häufig gemeinsam verwendet werden. Wir stützen uns zu diesem Zweck nochmals auf die Tatsache, daß der UPC eine DVD eindeutig identifiziert und verwenden den Hashwert der UPC als Hashwert des zugehörigen `DVD`-Objektes:

```
462. public int hashCode() {
463.     return upc.hashCode();
```

```
464. }
```

[24] Es ist wichtig, daß `hashCode()` bezüglich eines Programmaufrufs für verschiedene Objekte einer Klasse *vorzugsweise* auch verschiedene Hashwerte, bei identischen Objekten dagegen stets ein und denselben Hashwert liefert, gleichgültig wie oft die Methode aufgerufen wird (vorausgesetzt, daß sich der Zustand keines der beiden identischen Objekte ändert). Hätten wir in unserer `equals()`-Methode eine zusätzliche Prüfung implementiert (zum Beispiel den Namen des Hauptdarstellers), so ist es sinnvoll, auch die `hashCode()`-Methode entsprechend anzupassen. Beispielsweise könnten wir die Hashwerte von UPC und dem Namen des Hauptdarstellers addieren und die Summe als neuen Hashwert zurückgeben.

Tipp: Es ist sehr schwierig, eine Hashfunktion zu finden, die für jedes Objekt einer Klasse einen eindeutigen Hashwert liefert. Selbst wenn Sie viel Zeit investieren, um einen Algorithmus zu konstruieren, der mit hoher Wahrscheinlichkeit eindeutige Hashwerte erzeugt, würde die Effizienz der Anwendung durch die benötigte Zeit, um den Algorithmus auszuführen, vermutlich leiden. Investieren Sie nicht zu viel Zeit in eine eigene `hashCode()`-Methode, *es sei denn*, Sie stellen beim Untersuchen des Laufzeitverhaltens per Profiler fest, daß ein signifikanter Anteil der Laufzeit von Methoden beansprucht wird, die mit dem Hashwert zusammenhängen (zum Beispiel die `HashMap`-Methode `get()`).

5.1.2 Diskussion: Behandlung im Interface nicht deklarierter Ausnahmen

[25] Eventuell stellen Sie während der Arbeit an Ihrer Prüfungsaufgabe fest, daß Sie Methoden aufrufen müssen, die Ausnahmen auswerfen können, welche in Ihrem von Sun Microsystems vorgegebenen Interface nicht deklariert sind. Sie müssen bei jeder solchen Ausnahme entscheiden, wie Sie vorgehen wollen. Es gibt keine universelle perfekte Lösung, dafür aber verschiedene schlechte.

[26] Sie müssen selbst entscheiden, welchen Ansatz Sie für richtig halten. Beachten Sie die Anleitung zu *Ihrer* Prüfungsaufgabe, denken Sie daran, daß sich die Anleitungen geringfügig voneinander unterscheiden können und entscheiden Sie sich so, daß sie *Ihre* Anforderungen bestmöglich erfüllen. Vergessen Sie nicht, Ihre Entscheidungen sowohl in der Dokumentation Ihrer Design-Entscheidungen als auch in Form von Implementierungskommentaren im Quelltext zu vermerken.

[27] Wir veranschaulichen die verschiedenen Möglichkeiten anhand des folgenden Quelltextes:

```
01. import java.util.logging.*;
02.
03. public class InterruptedExceptionExample extends Thread {
04.     static Logger log = Logger.getAnonymousLogger();
05.
06.     public static void main(String[] args) throws InterruptedException {
07.         InterruptedExceptionExample iee = new InterruptedExceptionExample();
08.         iee.start();
09.
10.         while (iee.isAlive()) {
11.             log.info("main: waiting 5 seconds for other thread to finish");
12.             iee.join(5000);
13.
14.             if (iee.isAlive()) {
15.                 log.info("main: interrupting other thread.");
16.                 iee.interrupt();
17.             }
18.         }
```

```
19.         log.info("main: finished");
20.     }
21.
22.     public void run() {
23.         try {
24.             getLock();
25.         } catch (LockAttemptFailedException dle) {
26.             log.log(Level.WARNING, "Lock attempt failed", dle);
27.         }
28.     }
29.
30.     public void getLock() throws LockAttemptFailedException {
31.         // try to get some resource that we will presumably never get.
32.         for (;;) {
33.             try {
34.                 synchronized (InterruptedExceptionExample.class) {
35.                     log.info(getName() + ": waiting for some resource.");
36.                     InterruptedExceptionExample.class.wait();
37.                 }
38.             } catch (InterruptedException ie) {
39.                 // this is the bit we are interested in
40.             }
41.         }
42.     }
43.
44.     public class LockAttemptFailedException extends Exception {
45.         public LockAttemptFailedException(String msg, Throwable t) {
46.             super(msg, t);
47.         }
48.     }
49. }
```

Bemerkung: Beim Aufruf der `join()`-Methode in Zeile 12 kann eine Ausnahme vom Typ `InterruptedException` ausgeworfen werden. Da diese Ausnahme aber nichts mit der Diskussion in diesem Unterabschnitt zu tun hat, deklarieren wir sie per `throws`-Klausel als Ausnahme der `main()`-Methode.

[28] Die Rückkehr aus der `getLock()`-Methode hängt vom Eintreten einer Bedingung ab, wie der Kommentar in Zeile 31 andeutet. Für dieses Beispiel haben wir keine „richtige“ Bedingung festgelegt. Bei einer realen Anwendung könnte es das Warten auf die Sperre eines Objektes oder das Freiwerden einer Resource sein. Im vorliegenden Fall kommt es alleine darauf an, daß die erwartete Bedingung niemals eintritt.

[29] Die Betonung liegt bei diesem Beispiel auf der in Zeile 36 aufgerufenen `wait()`-Methode, genauer darauf, daß `wait()` eine Ausnahme vom Typ `InterruptedException` auswerfen kann, wobei die Signatur der `getLock()`-Methode aber nur eine Ausnahme vom Typ `LockAttemptFailedException` deklariert, der nicht von `InterruptedException` abgeleitet ist. Wir behandeln die Ausnahme in der `catch`-Klausel in den Zeilen 38–40.

Tipp: Sie können Threads bezüglich eines Klassenobjektes synchronisieren (siehe Zeilen 34 und 36). Dazu ist kein Objekt der Klasse erforderlich, sondern es genügt, den Klassennamen mit der Endung `.class` anzugeben. Zumeist sind bestimmte Objekte vorhanden, auf die sich die Synchronisierung bezieht. Ist kein offensichtlicher Kandidat vorhanden, sind Sie allerdings nicht automatisch gezwun-

gen, ein Objekt lediglich zu erzeugen, um seine Sperre zu benutzen, damit stets nur ein Thread einen Block von Anweisungen verarbeiten kann. (Ein Objekt, dessen Sperre verwendet wird, um den Zugriff aller anderen Threads zu verhindern, wenn ein Thread das Objekt gesperrt hat, heißt Mutex-Objekt.) Verwenden Sie die Synchronisierung bezüglich eines Klassenobjektes aber nicht unpassenden Situationen. Wenn Sie den Zweck eines Mutex-Objektes verdeutlichen wollen, ist es unter Umständen sinnvoll, ein `Object`-Objekt mit einem sprechenden Namen zu wählen, um den Quelltext klarer zu gestalten. Solche Entscheidungen gehören zur Aufgabe eines Entwicklers.

5.1.2.1 Unterdrücken der Ausnahme

[30] Die `catch`-Klausel in den Zeilen 38–40 des obigen Beispiels enthält keine Anweisungen, um die Ausnahme zu behandeln. Dieses Nichtstun wird häufig als „Unterdrücken der Ausnahme“ (*swallowing the exception*) bezeichnet. Nach der „Verarbeitung“ der Zeilen 38–40 hat das Programm die Ausnahme scheinbar verschluckt. Es bleibt keine Spur von der Ausnahme zurück.

[31] Das Unterdrücken von Ausnahmen ist bei Java zwar erlaubt, gilt aber als armseliger Programmierstil und kostet Sie bei der Bewertung Ihrer Prüfungsaufgabe ziemlich sicher einige Punkte (und sorgt außerdem für hitzige Diskussionen zwischen Ihnen und dem Entwickler, der Ihre Anwendung später einmal pflegen muß). Eine Ausnahme ist eine Situation, die eigentlich nicht eintreten sollte. Durch das Unterdrücken einer Ausnahme wird die Ausnahme aber nicht nur nicht behandelt, sondern auch jeder Anhaltspunkt für die eventuelle spätere Fehlerursache verworfen.

[32] Wenn wir das obige Beispiel mit leerer `catch`-Klausel aufrufen, kehrt die `main()`-Methode nicht zurück:

```
~$ java InterruptedExceptionExample

Aug 14, 2009 4:33:41 PM InterruptedExceptionExample main
INFO: main: waiting 5 seconds for other thread to finish
Aug 14, 2009 4:33:41 PM InterruptedExceptionExample getLock
INFO: Thread-1: waiting for some resource.
Aug 14, 2009 4:33:46 PM InterruptedExceptionExample main
INFO: main: interrupting other thread.
Aug 14, 2009 4:33:46 PM InterruptedExceptionExample main
INFO: main: waiting 5 seconds for other thread to finish
Aug 14, 2009 4:33:46 PM InterruptedExceptionExample getLock
INFO: Thread-1: waiting for some resource.
Aug 14, 2009 4:33:51 PM InterruptedExceptionExample main
INFO: main: interrupting other thread.
Aug 14, 2009 4:33:51 PM InterruptedExceptionExample main
INFO: main: waiting 5 seconds for other thread to finish
Aug 14, 2009 4:33:51 PM InterruptedExceptionExample getLock
INFO: Thread-1: waiting for some resource.

~$
```

5.1.2.2 Protokollieren der Ausnahme

[33] Es kann aus Gründen des betrieblichen Ablaufs erforderlich sein, eine Ausnahme zu ignorieren. Beispielsweise könnte eine Geschäftsregel (*business rule*) festlegen, daß Transaktionen aus Gründen der Rechnungsprüfung nicht abgebrochen werden dürfen: Eine Transaktion muß stets vollständig verarbeitet werden. Anschließend kann eine neue Transaktion gestartet werden, um die vorige Transaktion zu widerrufen (häufige Anforderung bei Geschäftsanwendungen). In einem solchen Fall

könnten wir entscheiden, die Unterbrechungsversuche unbeachtet zu lassen und fortzufahren, auf die Sperre zu warten.

[34] Es ist in einer solchen Situation nicht sinnvoll, die Ausnahme einfach zu unterdrücken, da sämtliche Hinweise auf den Versuch, den Thread zu unterbrechen, verloren gehen würden. Statt dessen ist es hier angebracht, die Ausnahme zu protokollieren, zum Beispiel:

```
38. } catch (InterruptedException ie) {
39.     log.info("Ignoring InterruptedException in transaction");
40. }
```

Die sehr schlichte Protokollmeldung lautet:

```
~$ java InterruptedExceptionExample

Aug 14, 2009 4:52:23 PM InterruptedExceptionExample main
INFO: main: waiting 5 seconds for other thread to finish
Aug 14, 2009 4:52:23 PM InterruptedExceptionExample getLock
INFO: Thread-1: waiting for some resource.
Aug 14, 2009 4:52:28 PM InterruptedExceptionExample main
INFO: main: interrupting other thread.
Aug 14, 2009 4:52:28 PM InterruptedExceptionExample main
INFO: main: waiting 5 seconds for other thread to finish
Aug 14, 2009 4:52:28 PM InterruptedExceptionExample getLock
INFO: Ignoring InterruptedException in transaction
Aug 14, 2009 4:52:28 PM InterruptedExceptionExample getLock
INFO: Thread-1: waiting for some resource.
Aug 14, 2009 4:52:33 PM InterruptedExceptionExample main

~$
```

[35] Diese Protokollmeldung transportiert nicht mehr Informationen, als daß sich der Entwickler entschieden hat, diese Ausnahme nicht zu beachten. Im vorliegenden Fall läßt sich mühelos feststellen, welche `catch`-Klausel diese Ausnahme abgefangen hat und was anschließend mit der Ausnahme geschehen ist. Liefe die Anwendung aber auf einem Server und könnte die `getLock()`-Methode von verschiedenen Methoden aufgerufen werden, so wäre der Stackinhalt unter Umständen nützlich, um herauszufinden, warum `getLock()` aufgerufen wurde. Die folgenden drei Zeilen bewirken einen detaillierteren Protokolleintrag:

```
38. } catch (InterruptedException ie) {
39.     log.log(Level.WARNING, "Ignoring InterruptedException in transaction", ie);
40. }
```

Die Ausgabe lautet:

```
~$ java InterruptedExceptionExample

NFO: Thread-1: waiting for some resource.
Aug 14, 2009 5:38:50 PM InterruptedExceptionExample main
INFO: main: waiting 5 seconds for other thread to finish
Aug 14, 2009 5:38:55 PM InterruptedExceptionExample main
INFO: main: interrupting other thread.
Aug 14, 2009 5:38:55 PM InterruptedExceptionExample main
INFO: main: waiting 5 seconds for other thread to finish
Aug 14, 2009 5:38:55 PM InterruptedExceptionExample getLock
WARNING: Ignoring InterruptedException in transaction
java.lang.InterruptedException
        at java.lang.Object.wait(Native Method)
        at java.lang.Object.wait(Object.java:485)
        at InterruptedExceptionExample.getLock(InterruptedExceptionExample.java:36)
```

```
        at InterruptedExceptionExample.run(InterruptedExceptionExample.java:24)
Aug 14, 2009 5:38:55 PM InterruptedExceptionExample getLock
INFO: Thread-1: waiting for some resource.
Aug 14, 2009 5:39:00 PM InterruptedExceptionExample main
INFO: main: interrupting other thread.end trace

~$
```

[36] Falls Sie sich entscheiden eine Ausnahme zu ignorieren, sollten Sie Ihre Entscheidung in Form eines Dokumentationskommentars (Javadoc) erwähnen, damit die Benutzer sich dieser Tatsache bewußt sein können (und nicht mitten in der Nacht anrufen, um zu fragen, warum sie einen erzeugten Thread nicht unterbrechen können).

5.1.2.3 Verpacken der Ausnahme in einer deklarierten Ausnahme

[37] In der Regel können Ausnahmen nicht einfach ignoriert werden, wie beim letzten Beispiel. Wenn beim Schreiben in eine Datei eine Ausnahme vom Typ `java.io.IOException` ausgeworfen wird, dürfen Sie diese Ausnahme nicht einfach unbeachtet lassen, da Sie andernfalls verfälschte Daten riskieren. Im vorigen Beispiel haben wir eine frei erfundene Geschäftsregel verwendet, die uns gestattet, Ausnahmen vom Typ `InterruptedException` zu ignorieren. Existiert eine solche Regel nicht, so müssen die Benutzer einer Anwendung davon ausgehen dürfen, daß sie ihre Threads unterbrechen können.

[38] Eine Behandlungsmöglichkeit besteht darin, die abgefangene Ausnahme in einer behandelten oder deklarierten Ausnahme zu verpacken, zum Beispiel:

```
38. } catch (InterruptedException ie) {
39.     throw new LockAttemptFailedException("InterruptedException in getLock", ie);
40. }
```

Die Ausgabe lautet:

```
~$ java InterruptedExceptionExample

Aug 14, 2009 5:58:39 PM InterruptedExceptionExample getLock
INFO: Thread-1: waiting for some resource.
Aug 14, 2009 5:58:39 PM InterruptedExceptionExample main
INFO: main: waiting 5 seconds for other thread to finish
Aug 14, 2009 5:58:44 PM InterruptedExceptionExample main
INFO: main: interrupting other thread.
Aug 14, 2009 5:58:44 PM InterruptedExceptionExample run
WARNING: Lock attempt failed
InterruptedExceptionExample$LockAttemptFailedException:
        Ignoring InterruptedException in getLock
        at InterruptedExceptionExample.getLock(InterruptedExceptionExample.java:42)
        at InterruptedExceptionExample.run(InterruptedExceptionExample.java:24)
Caused by: java.lang.InterruptedException
        at java.lang.Object.wait(Native Method)
        at java.lang.Object.wait(Object.java:485)
        at InterruptedExceptionExample.getLock(InterruptedExceptionExample.java:36)
        ... 1 more
Aug 14, 2009 5:58:44 PM InterruptedExceptionExample main

INFO: main: finished ~$
```

[39] Wie Sie an der Ausgabe sehen, erhalten wir eine Protokollmeldung vom Typ `WARNING`, die eine Ausnahme vom Typ `LockAttemptFailedException` mit dem erwarteten Stackinhalt dokumentiert. Diese Meldung stammt aus der `catch`-Klausel in den Zeilen 25–27. Noch wichtiger sind aber die

Zeilen ab „`Caused by:`“. Es sind dieselben Ausgabezeilen, die wir bereits auf Seite 122 als „detaillierteren Protokolleintrag“ erhalten haben. Beim Verpacken der `InterruptedException` in der `LockAttemptFailedException` bleibt der gesamte Stackinhalt der `InterruptedException` erhalten.

Warnung: Überlegen Sie sich genau, ob das Verpacken einer Ausnahme in einer anderen sinnvoll ist. Falls Sie nur einen einzigen Ausnahmetyp, beispielsweise `RecordNotFoundException`, verwenden dürfen, *kann* es sinnvoll sein, eine Ausnahme vom Typ `EOFException` in einer Ausnahme vom Typ `RecordNotFoundException` zu verpacken (nämlich dann, wenn Sie das Ende der Datei erreichen, ohne den gesuchten Datensatz zu finden). Wird die `EOFException` dagegen beim Lesen eines Datensatzes ausgeworfen, so ist die Datei möglicherweise beschädigt und die Fehlermeldung, daß der Datensatz nicht gefunden werden konnte bestensfalls irreführend, kann aber auch Probleme verursachen (beispielsweise verfälschte Daten). Es ist ebenfalls nicht sinnvoll, eine Ausnahme vom Typ `InterruptedException` in einer Ausnahme vom Typ `RecordNotFoundException` zu verpacken. Da Ihr Thread darauf wartet, einen Datensatz sperren zu können, ist dieser Datensatz voraussichtlich auch vorhanden.

5.1.2.4 Verpacken der Ausnahme in einer RuntimeException

[40] Es gibt Situationen, in denen das Verpacken der abgefangenen Ausnahme in einer deklarierten Ausnahme nicht sinnvoll ist (siehe obige Warnung).Wir können der Signatur einer Methode aber unter Umständen keine (weitere) geprüfte Ausnahme hinzufügen, ohne die Funktion anderer Anwendungen zu beeinträchtigen. Das gilt insbesondere wenn eine Klasse ein Interface implementiert: Ein anderer Entwickler könnte eine solche Klasse in seiner Anwendung gebrauchen, sich dabei auf ein gegebenes Interface beziehen und feststellen, daß sich sein Programm nicht erwartungsgemäß verhält oder nicht mehr übersetzen läßt, weil Sie das Interface geändert haben. Sie machen sich in jedem Fall schnell unbeliebt.

[41] Ausnahmen vom Typ `RuntimeException` und davon abgeleiteten Typen müssen weder in der Signatur einer Methode deklariert noch behandelt werden. Sie können die abgefangene Ausnahme also in einer Ausnahme vom Typ `RuntimeException` verpacken:

```
38. } catch (InterruptedException ie) {
39.     throw new RuntimeException("InterruptedException in getLock", ie);
40. }
```

[42] Das Verpacken einer Ausnahme in einer Ausnahme vom Typ `RuntimeException` erschwert dem Entwickler, der Ihre Klasse verwendet aber das Abfangen und Behandeln der tatsächlichen Ausnahme. Eine ziemlich armselige Ausnahmebehandlung ist:

```
22. public void run() {
23.     try {
24.         getLock();
25a.    } catch (RuntimeException re) {
25b.        log.log(Level.WARNING, "Caught the interrupt", re);
25c.    } catch (LockAttemptFailedException ex) {
26.         log.log(Level.WARNING, "Lock attempt failed", ex);
27.     }
28. }
```

[43] Indem wir in der `catch`-Klausel den Typ `RuntimeException` angeben, riskieren wir, sämtliche Untertypen von `RuntimeException` abzufangen, die wir in dieser Klausel gar nicht behandeln wollen (wir interessieren uns nur für den Ausnahmetyp `InterruptedException`). Angenommen, während der Verarbeitung der `getLock()`-Methode werde eine Ausnahme vom Typ `NullPointerException`

ausgeworfen. Da `NullPointerException` ein von `RuntimeException` abgeleiteter Typ ist, wird die `NullPointerException` in Zeile 25a abgefangen. Die `catch`-Klausel ist aber nicht für den Typ `NullPointerException` einrichtet, die Ausnahme wird also nicht angemessen behandelt. Um zu gewährleisten, daß wir nur die „Sorte“ der Ausnahmen vom Typ `RuntimeException` behandeln, für die wir uns interessieren, werten wir die Ursache der `RuntimeException` aus und werfen die Ausnahme erneut aus, wenn sie nicht von der erwarteten „Sorte“ ist:

```
22. public void run() {
23.     try {
24.         getLock();
25a.    } catch (RuntimeException re) {
25b.        if (re.getCause() instanceof InterruptedException) {
25c.            log.log(Level.WARNING, "Caught the interrupt", re);
25d.        } else {
25e.            throw re;
25f.        }
25g.    } catch (LockAttemptFailedException ex) {
26.         log.log(Level.WARNING, "Lock attempt failed", ex);
27.     }
28. }
```

[44] Mit dieser Änderung liefert das Programm die bereits vertraute Ausgabe:

```
~$ java InterruptedExceptionExample

Aug 14, 2009 5:58:39 PM InterruptedExceptionExample getLock
INFO: Thread-1: waiting for some resource.
Aug 14, 2009 5:58:39 PM InterruptedExceptionExample main
INFO: main: waiting 5 seconds for other thread to finish
Aug 14, 2009 5:58:44 PM InterruptedExceptionExample main
INFO: main: interrupting other thread.
Aug 14, 2009 5:58:44 PM InterruptedExceptionExample run
WARNING: Lock attempt failed
InterruptedExceptionExample$LockAttemptFailedException:
        Ignoring InterruptedException in transaction
        at InterruptedExceptionExample.getLock(InterruptedExceptionExample.java:42)
        at InterruptedExceptionExample.run(InterruptedExceptionExample.java:24)
Caused by: java.lang.InterruptedException
        at java.lang.Object.wait(Native Method)
        at java.lang.Object.wait(Object.java:485)
        at InterruptedExceptionExample.getLock(InterruptedExceptionExample.java:36)
        ... 1 more
Aug 14, 2009 5:58:44 PM InterruptedExceptionExample main
INFO: main: finished

~$
```

5.1.2.5 Verpacken der Ausnahme in einem Untertyp von RuntimeException

[45/46] Fünf Zeilen sind erforderlich, um die `RuntimeException` mit darin enthaltener `InterruptedException` zu behandeln, die Komplexität des Quelltextes hat also zugenommen. Andererseits müssen von `RuntimeException` abgeleitete Ausnahmetypen bekanntlich weder deklariert noch behandelt werden. Eine naheliegende Lösung besteht darin, einen neuen Ausnahmetyp von `RuntimeException` abzuleiten:

```
public class UserInterruptionException extends RuntimeException {
    public UserInterruptionException(String msg, Throwable t) {
```

```
            super(msg, t);
        }
    }
```

[47] Der neuen Ausnahmetyp kann analog zu `RuntimeException` im vorigen Beispiel verwendet werden:

```
38. } catch (InterruptedException ie) {
39.     throw new UserInterruptionException("InterruptedException in getLock", ie);
40. }
```

[48] Die Ausnahmebehandlung wird aber viel übersichtlicher:

```
22. public void run() {
23.     try {
24.         getLock();
25a.    } catch (UserInterruptionException uie) {
25b.        log.log(Level.WARNING, "Caught the interrupt", uie);
25c.    } catch (LockAttemptFailedException ex) {
26.         log.log(Level.WARNING, "Lock attempt failed", ex);
27.     }
28. }
```

Tipp: Eine von `RuntimeException` abgeleitete Ausnahme, die nicht abgefangen werden soll, wird der standardisierten Programmierpraxis folgend, nicht in der Signatur einer Methode deklariert. Dient die Ausnahme allerdings dazu eine Einschränkung durch das vorgegebene Interface zu umgehen, so sollten Sie sie sowohl in der Signatur der Methode deklarieren, als auch im Dokumentationskommentar der Methode beschreiben. Viele gängige Entwicklungsumgebungen zeigen das Ausnahmeverhalten einer Methode beim Eingeben des Methodennamens an. Einige Entwicklungsumgebungen erzeugen basierend auf der Signatur einer Methode sogar Vorlagen für `try`/`catch`-Klauseln. In solchen Fällen unterstützt die Deklaration des Ausnahmeverhaltens andere Entwickler, auch bei Untertypen von `RuntimeException`.

Wenn Sie die `RuntimeException` im Ausnahmeverhalten der Methode deklarieren und/oder in deren Dokumentationskommentar beschreiben, ist es sinnvoll, auch einen entsprechenden Implementierungskommentar im Quelltext anzulegen, der eventuellen späteren Entwicklern erläutert, warum Sie sich für diesen Ausnahmetyp entschieden haben. Bei Ihrer Prüfungsaufgabe sollten Sie auch im Dokumentationskommentar eine Notiz zu Ihrer Entscheidung einfügen.

---

[49] Daß Ausnahmen vom Typ `RuntimeException` nicht abgefangen werden müssen, ist ein großer Nachteil. Läßt der Entwickler die zusätzlichen Anweisungen in den Zeilen 25a–c aus, so läuft das Programm trotzdem klaglos, aber die `RuntimeException` wird durch den Stack ans obere Ende durchgereicht:

```
~$ java InterruptedExceptionExample

Aug 14, 2009 7:34:59 PM InterruptedExceptionExample getLock
INFO: Thread-1: waiting for some resource.
Aug 14, 2009 7:34:59 PM InterruptedExceptionExample main
INFO: main: waiting 5 seconds for other thread to finish
Aug 14, 2009 7:35:04 PM InterruptedExceptionExample main
INFO: main: interrupting other thread.
Exception in thread "Thread-1" java.lang.RuntimeException:
        Ignoring InterruptedException in transaction
        at InterruptedExceptionExample.getLock(InterruptedExceptionExample.java:48)
        at InterruptedExceptionExample.run(InterruptedExceptionExample.java:24)
```

```
Caused by: java.lang.InterruptedException
        at java.lang.Object.wait(Native Method)
        at java.lang.Object.wait(Object.java:485)
        at InterruptedExceptionExample.getLock(InterruptedExceptionExample.java:42)
        ... 1 more
Aug 14, 2009 7:35:04 PM InterruptedExceptionExample main
INFO: main: finished

~$
```

Warnung: Es ist wichtig, zu verstehen, daß die `RuntimeException` zum oberen Endes des Stacks *desjenigen Threads durchgereicht wird, der die Ausnahme verursacht hat*, nicht zum oberen Ende des Stacks der gesamten Laufzeitumgebung. Sie können der obigen Ausgabe entnehmen, daß der `main`-Thread noch läuft, nachdem die `RuntimeException` ausgeworfen wurde und `Thread-1` beendet ist. Dies hat wesentliche Auswirkungen auf die Entwicklung des Servers: Wenn Sie eine `RuntimeException` auswerfen und nicht behandeln, kann der Server weiterlaufen, aber nicht mehr in der Lage sein, Anfragen entgegen zu nehmen.

5.2 Die Klasse DvdDatabase und das Entwurfsmuster Façade

[50] Vor dem Anlegen einer Klasse ist es sinnvoll, die Aufgabe der Klasse zu betrachten. Dasselbe gilt für Methoden: Überlegen Sie sich bei jeder Methode welchen Zweck die erfüllen soll. Wenn Sie bei der Beschreibung einer Klasse oder Methode das Wort „und“ verwenden, so übernimmt sie vermutlich mehr Funktionalität als sie sollte. Ziehen Sie in diesem Fall in Betracht die Klasse oder Methode in zwei oder mehr Klassen beziehungsweise Methoden aufzuteilen. Ihr Quelltext wird dadurch übersichtlicher und ist leichter zu pflegen.

[51/52] Die Anleitung zu unserem Beispielprojekt verlangt, daß die Klasse `DvdDatabase` öffentlich ist und von allen anderen Klassen für den Zugriff auf die Datenbankdatei verwendet wird. Beim genauen Hinsehen zeigt sich allerdings, daß die Klasse *zwei unterschiedliche Aufgaben hat*, nämlich den physikalischen Zugriff auf die Datenbankdatei und das logische Sperren von Datensätzen. Die Beschreibung der Funktionalität der Klasse `DvdDatabase` enthält also das Wort „und“.

[53] Wir *könnten* diese beiden Funktionen zwar in einer Klasse implementieren, aber der Quelltext ist später besser zu pflegen, wenn wir die Aufgaben aufteilen. Wenn Sie später eine Methode ändern oder verstehen müssen, die physikalisch auf die Datenbankdatei zugreift, genügt es, wenn Sie sich die Klasse vornehmen, die sich um den Dateizugriff kümmert (es besteht keine Veranlassung, sich außerdem mit dem Sperrmechanismus auseinanderzusetzen).

[54] Es gibt ein Entwurfsmuster, das die Situation abbildet, von der wir gerade sprechen: *Façade*. Die Fassade (frz. *façade*) ist die Vorderseite eines Objektes, typischerweise eines Gebäudes. Die Fassade eines Ladens oder Gebäudes ist die Ansicht, mit der sich ein Mensch konfrontiert sieht. Die Klasse `DvdDatabase` hat für Entwickler dieselbe Funktion: Sie verbirgt Klassen über deren Existenz und Funktionsweise der Entwickler nicht Bescheid zu wissen braucht.

Bemerkung: Das Entwurfsmuster *Façade* verhindert zwar nicht das Wort „und“ in der Beschreibung der Funktionalität einer Klasse, bewirkt aber daß die *Façade*-Klasse die Funktionalität nicht selbst implementiert, sondern an verschiedene Klassen hinter den Kulissen überträgt.

[55] Unsere Klasse `DvdDatabase` fällt somit vergleichsweise einfach aus. Wir beginnen damit, Referenzen auf Objekte der Klassen anzulegen, die die eigentliche Arbeit verrichten:

```
package sampleproject.db;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Collection;
import java.util.List;
import java.util.regex.PatternSyntaxException;

public class DvdDatabase implements DBClient {
    private static ReservationsManager reservationsManager
            = new ReservationsManager();

    private static DvdFileAccess database = null;

    public DvdDatabase() throws FileNotFoundException, IOException {
        this(System.getProperty("user.dir"));
    }

    public DvdDatabase(String dbPath)
            throws FileNotFoundException, IOException {
        database = new DvdFileAccess(dbPath);
    }
```

[56] Wir kommen zum Anlegen der Konstruktoren der Klasse `DvdDatabase`. Die Anleitung unseres Beispielprojektes enthält keine Angaben zu den Konstruktoren von `DvdDatabase`, so daß wir eine gewisse Freiheit haben. Es gibt zwei primäre Fragen: Welche Parameter werden dem Konstruktor übergeben? Wirft der Konstruktor Ausnahmen aus und falls ja, welche?

[57] Hinsichtlich der Parameter des Konstruktors betrachten wir zwei unterschiedliche Anwendungsfälle: Die Klasse `DvdDatabase` kann im (netzwerkunabhängigen) *stand-alone*-Betriebsmodus der Anwendung verwendet werden, wobei keine Parameter benötigt werden und vorausgesetzt werden kann, daß die Datenbankdatei im aktuellen Arbeitsverzeichnis liegt. Die Klasse `DvdDatabase` wird anderseits auch im Netzwerkmodus der Anwendung verwendet, wobei sich Datenbankdatei und Anwendung in verschiedenen, eventuell sogar physikalisch getrennten Verzeichnissen (Festplatten) befinden. In diesem Fall müssen wir das Verzeichnis angeben können, in dem sich die Datenbankdatei befindet.

[58] Der erste (parameterlose) Konstruktor ist lediglich ein Spezialfall des zweiten (parameterbehafteten) Konstruktors, könnte also auch fortgelassen werden. Die Entscheidung sollte davon abhängen, wie oft der erste Konstruktor benötigt wird. Bei häufiger Verwendung werden die Benutzer dankbar sein, wenn es ihn gibt. Andernfalls lassen wir den Konstruktor fort und vereinfachen dadurch den Quelltext. In den seltenen Fällen, in denen die Datenbankdatei im aktuellen Arbeitsverzeichnis liegt, kann der parameterbehaftete Konstruktor mit dem Namen des Arbeitsverzeichnisses aufgerufen werden (`System.getProperties("user.dir")`).

[59] Wir haben uns hauptsächlich deswegen dafür entschieden, beide Konstruktoren zu belassen, um vorführen zu können, wie ein Konstruktor einen anderen Konstruktor aufruft. Diesen Ansatz findet man häufig bei überladenen Methoden und Konstruktoren, da er gewährleistet, daß stets dieselben Anweisungen ausgeführt werden, unabhängig davon, welche überladene Version aufgerufen wird.

[60] Der zweite Konstruktor öffnet (implizit) die Datenbankdatei, das heißt, daß eventuell eine Ausnahme vom Typ `FileNotFoundException` ausgeworfen wird, wenn sich die Datei nicht im angegebenen Verzeichnis befindet, oder eine Ausnahme vom Typ `IOException`, wenn beim Öffnen der Datei ein Problem auftritt (etwa ungenügende Zugriffsberechtigung). Damit stehen wir vor der Entscheidung, ob wir diese Ausnahmen im Konstruktor behandeln oder an die aufrufende Methode

weitergeben wollen.

[61] Was können wir tun, wenn eine dieser beiden Ausnahmen ausgeworfen wird? Wir können in der Klasse `DvdDatabase` nicht mehr tun, als die fehlgeschlagene Operation erneut zu versuchen, wobei die beim ersten Versuch nicht gefundene Datei wahrscheinlich auch beim zweiten Versuch nicht auftauchen wird. Wir können diese beiden Ausnahmen aber auch nicht einfach protokollieren und unbeachtet lassen, da der Benutzer fälschlicherweise davon ausgehen würde, daß das `DvdDatabase`-Objekt korrekt erzeugt wurde. Wir haben uns daher entschieden, diese beiden Ausnahmen an die Methode weiterzugeben, die das `DvdDatabase`-Objekt erzeugt.

Tipp: Wir haben uns dafür entschieden, die beiden Ausnahmen vom Typ `FileNotFoundException` beziehungsweise `IOException` per `throws` als Ausnahmeverhalten des Konstruktors zu deklarieren, dadurch aber impliziert bekannt gegeben, daß wir eine Datei zur Datenspeicherung verwenden. Eine zukünftige Verbesserung wäre, die Daten in einer Datenbank zu sichern, wobei die obigen Ausnahmetypen nicht mehr vorkommen würden. Eine bessere Lösung wäre daher, einen eigenen Ausnahmetyp namens `DatabaseFailureException` anzulegen und die Ausnahmen vom Typ `FileNotFoundException` beziehungsweise `IOException` in einer Ausnahme des neuen Typs zu verpacken. Mit dieser Lösung könnten wir bei einem späteren Umstieg auf eine Datenbank die SQL-Ausnahmen einfach in Ausnahmen vom Typ `DatabaseFailureException` verpacken und die Anwendung würde wie zuvor funktionieren. Wir überlassen diese Änderung dem Leser als Übungsaufgabe.

---

[62] Die Klasse `DvdDatabase` implementiert jede im Interface *`DBClient`* deklarierte Methode, die wiederum die entsprechende Methode in einer der Hilfsklassen aufruft:

```
public boolean addDVD(DVD dvd) throws IOException {
    return database.addDvd(dvd);
}

public DVD getDVD(String upc) throws IOException {
    return database.getDvd(upc);
}

public boolean removeDVD(String upc) throws IOException {
    return database.removeDvd(upc);
}

public boolean modifyDVD(DVD dvd) throws IOException {
    return database.modifyDvd(dvd);
}

public List<DVD> getDVDs() throws IOException {
    return database.getDvds();
}

public Collection<DVD> findDVD(String query)
        throws IOException, PatternSyntaxException {
    return database.find(query);
}

public boolean reserveDVD(String upc) throws InterruptedException {
    return reservationsManager.reserveDvd(upc, this);
}

public void releaseDVD(String upc) {
    reservationsManager.releaseDvd(upc, this);
}
```

5.3 Die Klasse DvdFileAccess

[63] Wir besprechen die Klasse `DvdFileAccess` Abschnitt für Abschnitt, statt den gesamten Quelltext auf einmal vorzustellen. Sie können den Quelltext der Beispielanwendung *Denny's DVDs* aus dem „Source Code"-Abschnitt der Apress-Website (*http://www.apress.com/book/sourcecode*) herunterladen.

[64] Da es nur eine physikalische Datenbankdatei gibt, könnten wir die Klasse `DvdFileAccess` als *Singleton* anlegen, also so gestalten, daß stets höchstens ein einziges Objekt vorhanden sein kann. Andererseits könnten viele Arbeitsschritte parallel verrichtet werden, wenn die Anwendung von mehreren Benutzern bedient und auf einem Mehrprozessorsystem betrieben wird, beispielsweise die Umwandlung eines `DVD`-Objektes in das Datensatzformat der Datenbankdatei oder die Verarbeitung einer Suchanfrage an die Datenbankdatei. Das Anlegen der Klasse `DvdFileAccess` als *Singleton* würde die Implementierung jeder Klasse beeinflussen, die auf das `DvdFileAccess`-Objekt zugreift. Fiele irgendwann in der Zukunft die Entscheidung, daß die Klasse `DvdFileAccess` kein *Singleton* mehr sein, sondern den Zugriff auf mehrere Datenbankdateien ermöglichen soll, so müßten wir alle Klassen überarbeiten, in denen ein Objekt der Klasse `DvdFileAccess` vorkommt. Aus diesen Gründen haben wir uns dagegen entschieden, `DvdFileAccess` als *Singleton* anzulegen.

[65] Wie im vorigen Abschnitt beschrieben, implementiert die Klasse `DvdDatabase` das Entwurfsmuster *Façade* und alle anderen Klassen sollen per `DvdDatabase` auf die Datenbankdatei zugreifen. Keine der übrigen Klassen soll `DvdDatabase` direkt verwenden. Diese Absicherung ist dadurch gegeben, daß wir die Klasse `DvdFileAccess` unter Standardzugriff (Packagezugriff) stellen, siehe Zeile 31 (nur Klassen die im Package `sampleproject.db` liegen, haben Zugriff auf die Klasse `DvdFileAccess`). Wie bereits beschrieben, hat die Folge der Zeilennummern Lücken, da Kommentare im Quelltext ausgeblendet sind:

```
01. package sampleproject.db;
02.
03. import java.io.*;
04. import java.util.*;
05. import java.util.concurrent.locks.*;
06. import java.util.logging.*;
07. import java.util.regex.*;
...
31. class DvdFileAccess {
...
35.     private static final String DATABASE_NAME = "dvd_db.dvd";
...
41.     private Logger log = Logger.getLogger("sampleproject.db");
...
46.     private RandomAccessFile database = null;
...
51.     private static Map<String, Long> recordNumbers
52.             = new HashMap<String, Long>();
```

[66] Die Typen der Felder `DATABASE_NAME` (`String`), `log` (`Logger`) und `database` (`RandomAccessFile`) standen auch schon bei den früheren Versionen des JDKs zur Verfügung. *`Map<String, Long>`* dagegen, der Typ des vom `recordNumbers`-Feldes referenzierten Kollektion, bezeichnet eine der erst seit Version 5 des Java Development Kits vorhandenen generischen Kollektionen, die dem Compiler gestatten, *bereits zur Übersetzungszeit zu prüfen*, ob eine Kollektion *typsicher* verwendet wird. Damit sind Ausnahmen vom Typ `ClassCastException` zur Laufzeit weitestgehend ausgeschlossen. Wir besprechen die Anwendung der von `recordNumbers` referenzierten generischen Kollektion im Anschluß an die Diskussion der Konstruktoren auf Seite 134f.

```
58.     private static ReadWriteLock recordNumbersLock
59.             = new ReentrantReadWriteLock();
...
66.     private static String emptyRecordString = null;
...
71.     private static String dbPath = null;
...
77.     static {
78.         emptyRecordString = new String(new byte[DVD.RECORD_LENGTH]);
79.     }
```

[67] Beim Schreiben eines Datensatzes in die Datenbankdatei haben wir zwei Möglichkeiten: Wir können jedes Feld einzeln schreiben und gegebenenfalls mit Leerzeichen auffüllen oder einen kompletten Datensatz auf einmal in einer einzigen Operation schreiben.

[68] Da die Festplatte permanent in Bewegung ist, würde das feldweise Schreiben deutlich mehr Zeit beanspruchen, als das Schreiben eines kompletten Datensatzes, da sich die Festplatte nach jeder Schreiboperation weiterbewegt und eine Verzögerung entsteht, um den Schreibkopf erneut an die richtige Position zu setzen (die Positionierung wird zwar vom Festplatten-Controller ausgeführt, verlangsamt die Schreiboperation aber dennoch). Die Verzögerungen durch das Schreiben einzelner Felder sind zwar gering, aber dennoch vorhanden und wären ein Engpaß in einem Mehrbenutzersystem, da stets nur ein Benutzer auf die Festplatte schreiben kann. Wir haben uns daher entschieden, den gesamten Datensatz auf einmal zu schreiben.

[69] Wir besprechen zusammen mit der Beschreibung der Methode `persistDvd()` auf Seite 138f einen Ansatz, um einen Datensatz zu konstruieren, bevor er in die Datenbankdatei geschrieben wird. Sowohl aus Geschwindigkeitserwägungen als auch der Einfachheit halber, haben wir uns entschieden, die Konstruktion eines Datensatzes mit einem `StringBuilder`-Objekt von bekannter Länge zu beginnen und die Bytes durch die Werte der DVD-Felder zu ersetzen. Da wir ein `StringBuilder`-Objekt mit bekannter Länge erzeugen wollen, das vor dem Einsetzen des ersten Feldwertes nur Nullen enthält, haben wir ein Feld `emptyRecordString` angelegt, dessen Inhalt dem `StringBuilder`-Konstruktor übergeben werden kann. `emptyRecordString` ist ein statisches Feld und wird mit Hilfe eines statischen Initialisierungsblocks bewertet (siehe oben).

```
90.     public DvdFileAccess(String suppliedDbPath)
91.             throws FileNotFoundException, IOException {
92.         log.entering("DvdFileAccess", "DvdFileAccess", suppliedDbPath);
93.         if (database == null) {
94.             database = new RandomAccessFile(
95.                 suppliedDbPath + File.separator + DATABASE_NAME, "rw");
96.             getDvdList(true);
97.             dbPath = suppliedDbPath;
98.             log.fine("database opened and file location table populated");
99.         } else if (dbPath != suppliedDbPath) {
100.             log.warning("Only one database location can be specified. "
101.                         + "Current location: " + dbPath + " "
102.                         + "Ignoring specified path: " + suppliedDbPath);
103.         }
104.         log.exiting("DvdFileAccess", "DvdFileAccess");
105.     }
```

[70] Der Konstruktor der Klasse `DvdFileAccess` enthält Anweisungen, die nicht bei jedem Aufruf verarbeitet werden brauchen. Beispielsweise muß die Initialisierung des `database`-Feldes nicht wiederholt werden. Beim ersten Aufruf des Konstruktors wird das `database`-Feld mit einer Referenz auf ein `RandomAccessFile`-Objekt bewertet, das die Datenbankdatei repräsentiert. Danach besteht kein Bedarf mehr, diese Initialisierung zu wiederholen. Wir prüfen in Zeile 93, ob das `database`-

Feld bereits konfiguriert ist und führen die Initialisierung im Positivfall nicht mehr aus. Allerdings besteht die Möglichkeit, daß der Konstruktor mehrmals mit verschiedenen Pfaden aufgerufen wird. In diesem Fall wird eine Warnung protokolliert, die aussagt, daß der neuere Pfad nicht beachtet wurde.

[71] Wir müssen uns als Entwickler stets darüber im Klaren sein, wie schnell unsere Anwendung arbeitet. Es kommt nicht darauf an, sich über jede einzelne Zeile den Kopf zu zerbrechen, wohl aber darauf, typische Stellen zu erkennen, die die Geschwindigkeit beschränken.

Warnung: Eines Tages (falls nicht bereits geschehen) wird man Sie auffordern, das Laufzeitverhalten Ihrer Anwendung zu verbessern. Es ist zwar gut und richtig, problematische Stellen durch Lesen des Quelltextes sozusagen „manuell“ zu suchen, aber Sie sollten stets auch einen Profiler benutzen. Ein Profiler ist ein Programm, welches Ihre Anwendung während der Ausführung überwacht und ermittelt, an welchen Stellen der größte Teil der Laufzeit beansprucht wird. Wenn Sie wissen, welche Methoden den größten Teil der Laufzeit benötigen, können Sie gezielt an der Geschwindigkeit Ihrer Anwendung arbeiten, statt sich darauf zu verlassen, daß Sie zufällig die richtige Stelle finden. Die hier wiedergegebenen Stichpunkte zur Verbesserung des Laufzeitverhaltens der Klasse `DvdDatabase` sind so gewählt, daß wir bestimmte Abschnitte des Quelltextes und bestimmte Eigenschaften von Version 5 des Java Development Kits diskutieren können. Es sind keinesfalls die einzigen (oder gar die besten) Stellen, an denen sich das Programm verbessern läßt.

---

[72] Das Lesen und Schreiben von der beziehungsweise auf die Festplatte ist eine der langsamsten Operationen, die bei unserer Beispielanwendung vorkommen. Benötigen mehrere Benutzer Zugriff auf verschiedene Datensätze, so müssen die Anfragen bei nur einer Datenbankdatei in einer Warteschlange aneinandergereiht werden, wodurch sich ein Engpaß ergibt.

[73] Das Lesen beziehungsweise Schreiben eines Datensatzes läßt sich beschleunigen, wenn die Position des Datensatzes in der Datenbankdatei bekannt ist. Wären die Datensätze durchnummiert, der Primärschlüssel also ganzzahlig, so könnten wir die Position eines Datensatzes in der Datenbankdatei berechnen, indem wir den Primärschlüssel mit der einheitlichen Datensatzlänge multiplizieren. Unsere Beispielanwendung verwendet allerdings den UPC als Primärschlüssel, so daß wir eine Abbildung der UPCs auf die Datensatzpositionen in der Datenbankdatei benötigen. Diese Abbildung läßt sich nur dadurch definieren, daß wir die Datenbankdatei einmal komplett einlesen und dabei die UPCs und Datensatzpositionen erfassen. Wir könnten nun eine Methode anlegen, die diese Aufgabe übernimmt, wobei das zu implementierende Interface *`DBClient`* bereits die Methode `getDVDs()` deklariert, die die gesamte Datenbankdatei einliest und eine Referenz auf ein *`List`*-Objekt zurückgibt, das die in der Datenbankdatei gespeicherten DVDs enthält. Wir *könnten* diese Methode verwenden, um auch die benötigte Abbildung zu definieren.

[74] Dieser Ansatz birgt allerdings eine Gefahr: Der Konstruktor der Klasse `DvdFileAccess` verläßt sich nämlich darauf, daß die Methode `getDVDs()` die Abbildung der UPCs auf die Datensatzpositionen definiert. Falls `getDVDs()` zu einem späteren Zeitpunkt in einer von `DvdFileAccess` abgeleiteten Klasse überschrieben wird, ist die Definition dieser Abbildung eventuell nicht mehr gewährleistet. Mögliche Auswege sind, `getDVDs()` als finale Methode zu deklarieren, so daß keine abgeleitete Klasse diese Methode überschreiben kann oder eine *zusätzliche* private Methode (`getDvdList()`) aufzurufen, die sich um die Definition der Abbildung kümmert. (Private Methoden können, wie finale Methoden, nicht überschrieben werden, da sie für andere Klassen nicht sichtbar sind.)

[75] Die Methode `getDVDs()` repräsentiert einen sogenannten „Geschäftsprozeß“ (*business method*), ist also nur aufgrund des vereinbarten „geschäftlichen“ Rahmens vorhanden (die Anwendung könnte mühelos ohne diese Methode entwickelt werden). Unsere Implementierung der Methode `getDVDs()` liest die gesamte Datenbankdatei von Anfang bis Ende ein, könnte aber auch vollkommen anders

aus programmiert werden, beispielsweise so, daß sie nicht direkt auf die Datenbankdatei zugreift, sondern zu diesem Zweck eine andere öffentliche Methoden wie `getDVD()` aufruft. Es ist nicht auszuschließen, daß die Methode `getDVDs()` zu einem späteren Zeitpunkt überschrieben werden muß, so daß es nicht sinnvoll ist, `getDVDs()` als finale Methode zu deklarieren.

[76] Die Methode `getDVDs()` muß aber als öffentliche Methode (`public`) deklariert werden, da sie im Interface *`DBClient`* deklariert ist.

[77] Es gibt eine einfache Lösung für das eben beschriebene Dilemma: Wir legen eine zusätzliche private Methode namens `getDvdList()` an, die sich um die Definition der Abbildung der UPCs auf die Datensatzpositionen in der Datenbankdatei kümmert und rufen diese Methode implizit über `getDVDs()` auf (*`DBClient`*/`DvdDatabase.getDVDs()`→`DvdFileAccess.getDvds()`→`DvdFileAccess.getDvdList()`). Falls `getDVDs()` später einmal überschrieben wird, existiert die `getDvdList()`-Methode weiterhin und kann zur Initialisierung der Abbildung aufgerufen werden.

[78] Dies ist keine perfekte Lösung. Durch das Aufrufen der `getDvdList()`-Methode, um unsere Abbildung zu definieren, erzeugen wir ein nicht benötigtes *`List`*-Objekt und verwerfen es sofort wieder. Obwohl in der Regel unvernünftig, ist es im vorliegenden Fall gerechtfertigt, eine Kollektion zu erzeugen und wieder zu verwerfen, ohne sie verwendet zu haben. Es ist lediglich der Konstruktor, der das *`List`*-Objekt verschwendet und die Alternative besteht darin eine nahezu identische Methode anzulegen, deren einziger Zweck darin besteht, unsere Abbildung zu definieren. Unsere `getDvdList()`-Methode liest alle Datensätze aus der Datenbankdatei, definiert die Abbildung von UPCs auf die Datensatzpositionen und wird von der öffentlichen `DvdFileAccess`-Methode `getDvds()` aufgerufen:

```
113. public List<DVD> getDvds() throws IOException {
114.     return getDvdList(false);
115. }
```

Bemerkung: Die `DvdFileAccess`-Methode `getDvds()` weicht vom Namensschema der *`DBClient`*-Methode `getDVDs()` ab. Das ist in Ordnung, da die Klasse `DvdFileAccess` „hinter“ der *Façade*-Klasse `DvdDatabase` verborgen liegt und das Interface *`DBClient`* nicht implementiert. Wir haben uns dafür entschieden, bei den Methodennamen unserer nicht-öffentlichen Klasse der Formatierungsrichtlinie von Sun Microsystems zu folgen, auch wenn diese Methodennamen dadurch nicht mit den Methodennamen in den öffentlichen Klassen und dem Interface *`DBClient`* übereinstimmen. Die Entscheidung hat aber auch einen Nachteil: Ein erfahrener Entwickler wird an der Klasse `DvdFileAccess` schätzen, daß die Klassen- und Methodennamen der Formatierungsrichtlinie von Sun Microsystems genügen. Andererseits können die zwischen öffentlichen und nicht-öffentlichen Klassen uneinheitlichen Namensschemata einen unerfahrenen Entwickler verwirren.

```
130. private List<DVD> getDvdList(boolean buildRecordNumbers)
131.         throws IOException {
132.     log.entering("DvdFileAccess", "getDvdList", buildRecordNumbers);
133.     List<DVD> returnValue = new ArrayList<DVD>();
134.
135.     if (buildRecordNumbers) {
136.         recordNumbersLock.writeLock().lock();
137.     }
138.
139.     try {
140.         for (long locationInFile = 0;
141.              locationInFile < database.length();
142.              locationInFile += DVD.RECORD_LENGTH) {
143.             DVD dvd = retrieveDvd(locationInFile);
```

```
144.             log.fine("retrieving record at " + locationInFile);
145.             if (dvd == null) {
146.                 log.fine("found deleted record ");
147.             } else {
148.                 log.fine("found record " + dvd.getUPC());
149.                 if (buildRecordNumbers) {
150.                     recordNumbers.put(dvd.getUPC(), locationInFile);
151.                 }
152.                 returnValue.add(dvd);
153.             }
154.         }
155.     } finally {
156.         if (buildRecordNumbers) {
157.             recordNumbersLock.writeLock().unlock();
158.         }
159.     }
160.
161.     log.exiting("DvdFileAccess", "getDvdList");
162.     return returnValue;
163. }
```

[79] Ebenso wie die Datenbankdatei, wird auch die Abbildung der UPCs auf die Datensatzpositionen durch ein Objekt repräsentiert, das von vielen Threads verwendet wird. In den meisten Fällen fragen die Threads das von `recordNumbers` referenzierte `HashMap`-Objekt nur ab. Änderungen kommen viel seltener vor. Vor Version 5 des Java Development Kits wären sämtliche Zugriffe auf das `HashMap`-Objekt synchronisiert worden, so daß stets höchstens ein einzelner Thread Zugriff auf das Objekt erhält. Seit Version 5 des Java Development Kits haben wir spezielle Sperrobjekte vom Typ *`java.util.concurrent.locks.ReadWriteLock`* zur Verfügung, welche die gleichzeitige Verarbeitung mehrerer Threads begünstigen. Anstelle der Synchronisierung eines Blocks von Anweisungen, schließen wir die entsprechenden Anweisungen zwischen den Methoden `lock()` und `unlock()` ein. Ein *`ReadWriteLock`*-Objekt kann immer von mehreren Threads zum Lesen gesperrt werden, aber stets nur von einem Thread zum Schreiben.

[80] Beim Aufruf der Methode `getDvdList()` vom Konstruktor aus, wird das von `recordNumbers` referenzierte `HashMap`-Objekt initialisiert. Eine Schreibsperre gewährleistet, daß zwischenzeitlich keine Zugriffe durch andere Threads erfolgen. Die Sperre wird in Zeile 136 gesetzt und in der `finally`-Klausel in Zeile 157 wieder aufgehoben. Es ist wichtig, daß die Sperre wieder aufgehoben wird, selbst wenn eine Ausnahme ausgeworfen wird. Durch Platzieren der `unlock()`-Methode in der `finally`-Klausel wird die Sperre garantiert stets wieder aufgehoben. Dieses Aufheben der Sperre in der `finally`-Klausel wird bei Verwendung der neuen Sperrobjekte generell empfohlen.

[81] Zeile 150 legt im `HashMap`-Objekt einen neuen Eintrag an, bestehend aus einem UPC und der Datensatzposition in der Datenbankdatei. Die Verwendung einer generischen Kollektion sowie des Autoboxingmechanismus' (siehe Kapitel 2) sorgt für übersichtlichen Quelltext bei gleichzeitiger Garantie, daß keine ungültigen Einträge in das `HashMap`-Objekt aufgenommen werden. Zu Beginn der Klasse `DvdFileAccess` haben wir deklariert, daß das von `recordNumbers` referenzierte `HashMap`-Objekt nur `String`-Referenzen als Schlüssel und `Long`-Referenzen als Werte akzeptiert:

```
51. private static Map<String, Long> recordNumbers
52.         = new HashMap<String, Long>();
```

[82] Der Compiler validiert anhand dieser Deklaration *bereits zur Übersetzungszeit*, daß wir `String`-Referenzen als Schlüssel und `Long`-Referenzen als Werte für unser `HashMap`-Objekt verwenden. Der Autoboxingmechanismus gestattet uns, einen Eintrag anzulegen, der aus einem `String`-Schlüssel und einem *primitiven* `long`-Wert besteht, da Java den primitiven Wert automatisch in den für die

generische Kollektion erforderlichen Wrappertyp `Long` umwandelt.

[83] Vor Version 5 des Java Development Kits hatte der Compiler keine Möglichkeit, um zu prüfen, ob der Typ eines einer Kollektion übergebenen Elementes der tatsächlich gewünschte Typ war. Wenn Sie heute ein Element eines falschen Typs in eine typisierte Kollektion einzutragen versuchen, meldet der Compiler einen Fehler. Wenn Sie ein Beispiel für diese Fehlermeldung sehen möchten, ändern Sie Zeile 150 folgermaßen:

```
recordNumbers.put(dvd.getUPC(), (int) locationFile);
```

[84] Ein Compiler ab Version 5 des Java Development Kits gibt nun eine Fehlermeldung aus, weil der übergebene Datentyp nicht mit dem für diese Kollektion deklarierten Inhalt verträglich ist:

```
sampleproject/db/DvdFileAccess.java:150: put(java.lang.String,java.lang.Long)
    in java.util.Map<java.lang.String,java.lang.Long>
    cannot be applied to (java.lang.String,int)
                        recordNumbers.put(dvd.getUPC(), (int) locationInFile);
                                      ^

1 error
```

[85] Vergegenwärtigen Sie sich, daß dies lediglich eine Prüfung zur Übersetzungszeit ist. Es ist immer noch möglich, zur Laufzeit Elemente eines ungültigen Typs einzutragen, wodurch eine Ausnahme vom Typ `ClassCastException` hervorgerufen wird.

[86] In Zeile 143 wird die private Methode `retrieveDvd()` aufgerufen, um den Datensatz einer DVD in Abhängigkeit von seiner Position in der Datenbankdatei abzufragen. Auch die öffentliche Methode `getDvd()`, die eine DVD anhand ihres UPCs abfragt, verwendet hinter den Kulissen `retrieveDvd()`:

```
172. public DVD getDvd(String upc) throws IOException {
173.     log.entering("DvdFileAccess", "getDvd", upc);
174.
175.     recordNumbersLock.readLock().lock();
176.     try {
177.         // Determine where in the file this record should be.
178.         // note: if this is null the record does not exist
179.         Long locationInFile = recordNumbers.get(upc);
180.         return (locationInFile != null) ? retrieveDvd(locationInFile)
181.                                         : null;
182.     } finally {
183.         recordNumbersLock.readLock().unlock();
184.         log.exiting("DvdFileAccess", "getDvd");
185.     }
186. }
```

[87] Zeile 175 sperrt das von `recordNumbersLock` referenzierte `ReentrantReadWriteLock`-Objekt vor der Anfrage an das von `recordNumbers` referenzierte `HashMap`-Objekt. Beachten Sie, daß stets mehr als Thread verarbeitet werden kann und das Anfordern einer Lesesperre weiteren Threads gestattet, ebenfalls eine Lesesperre anzufordern.

[88] Falls der angeforderte UPC nicht in der Datenbankdatei existiert, wird `null` an die aufrufende Methode zurückgegeben (Zeile 181). Andernfalls wird in Zeile 180 die Methode `retrieveDvd()` aufgerufen und das erhaltende Ergebnis an die aufrufende Methode zurückgegeben. Wenn `retrieveDvd()` eine Ausnahme vom Typ `IOException` oder `RuntimeException` auswirft, ist es wichtig, daß die Lesesperre des `ReentrantReadWriteLock`-Objektes garantiert wieder aufgehoben wird. Daher befindet sich der Aufruf der `unlock()`-Methode wiederum in einer `finally`-Klausel (Zeile 183). Denken Sie daran, daß die `finally`-Klausel ausgeführt wird, *obwohl* die `return`-Anweisung bereits in Zeile 180 steht.

```
196. private DVD retrieveDvd(long locationInFile) throws IOException {
197.     log.entering("DvdFileAccess", "retrieveDvd", locationInFile);
198.     final byte[] input = new byte[DVD.RECORD_LENGTH];
...
202.     synchronized (database) {
203.         database.seek(locationInFile);
204.         database.readFully(input);
205.     }
```

[89] Die Mehrzahl der Threads, die auf das von `recordNumbers` referenzierte `HashMap`-Objekt zugreifen, fragen das Objekt nur ab. Da mehrere Leseoperationen zugleich ausgeführt werden können, ohne daß sich die entsprechenden Threads gegenseitig beeinflussen, ist der Zugriff auf das `HashMap`-Objekt ein perfektes Beispiel für die Verwendung eines Sperrobjektes vom Typ *`ReadWriteLock`*.

[90] Aber auch beim *Lesen* der Datenbankdatei kann ein Thread einen anderen gleichzeitig operierenden Thread beeinflussen. Das Einlesen eines Datensatzes aus der Datenbankdatei besteht aus zwei Schritten, nämlich dem Positionieren des Dateizeigers auf den richtigen Datensatz und das anschließende Lesen. Es ist sehr wichtig, daß diese beiden Schritte als eine atomare Operation ausgeführt werden. Andernfalls könnte die Position des Dateizeigers zwischen den Zeilen 203 und 204 von einem anderen Thread geändert worden sein und wir würden nicht den richtigen Datensatz lesen. Die Verwendung eines Sperrobjektes vom Typ *`ReadWriteLock`* würde sich in der vorliegenden Situation kontraproduktiv auswirken, da für beide Schritte eine Schreibsperre erforderlich ist und die Geschwindigkeit der Anwendung durch den Aufwand beinträchtigt werden würde, zu Prüfen ob noch aktive Lesesperren vorhanden sind. Die traditionelle Synchronisierung ist in diesem Fall die bessere Lösung (Zeilen 202–205).

[91] Da ein synchronisierter Block von Anweisungen alle anderen Threads bei Zugriffsversuchen auf die Datenbankdatei blockiert, definieren wir den Block so klein wie möglich, um die Wartezeit für die übrigen Threads zu verringern. Der synchronisierte Block dauert daher nur solange, bis der Datensatz vollständig eingelesen ist. Beachten Sie, daß die `RandomAccessFile`-Methode `read()` lediglich garantiert, daß wenigstens ein Byte, nicht aber der gesamte Datensatz eingelesen wird. Wir verwenden daher die `RandomAccessFile`-Methode `readFully()`, um zu garantieren, daß stets ein vollständiger Datensatz eingelesen wird.

[92] Nachdem wir einen Datensatz komplett gelesen und in dem von der lokalen Variablen `input` referenzierten finalen `byte`-Array (Zeile 198) gespeichert haben, können wir die einzelnen Feldinhalte mit Hilfe der lokalen inneren Klasse `RecordFieldReader` extrahieren:

```
211.     class RecordFieldReader {
212.         private int offset = 0;
213.         String read(int length) throws UnsupportedEncodingException {
214.             String str = new String(input, offset, length, "UTF-8");
215.             offset += length;
216.             return str.trim();
217.         }
218.     }
219.
220.     RecordFieldReader readRecord = new RecordFieldReader();
221.     String upc = readRecord.read(DVD.UPC_LENGTH);
222.     String name = readRecord.read(DVD.NAME_LENGTH);
223.     String composer = readRecord.read(DVD.COMPOSER_LENGTH);
224.     String director = readRecord.read(DVD.DIRECTOR_LENGTH);
225.     String leadActor = readRecord.read(DVD.LEAD_ACTOR_LENGTH);
226.     String supportingActor = readRecord.read(DVD.SUPPORTING_ACTOR_LENGTH);
227.     String year = readRecord.read(DVD.YEAR_LENGTH);
228.     int copy = Integer.parseInt(readRecord.read(DVD.COPIES_LENGTH));
```

```
229.
230.     DVD returnValue = ("DELETED".equals(upc))
231.                       ? null
232.                       : new DVD(upc, name, composer, director, leadActor,
233.                                 supportingActor, year, copy);
234.
235.     log.exiting("DvdFileAccess", "retrieveDvd", returnValue);
236.     return returnValue;
237. }
```

[93] Wenn der Datensatz nicht als gelöscht markiert ist (`DELETED`), erzeugen wir schließlich ein neues `DVD`-Objekt und geben eine Referenz darauf an die aufrufende Methode zurück. Bei der Initialisierung des `DVD`-Objektes entfernen wir sämtliche führenden und endständigen Leerzeichen und Nullen der einzelnen Feldinhalte.

[94] Das von `recordNumbers` referenzierte `HashMap`-Objekt muß nach seiner Initialisierung nur selten geändert werden. Aktualisierungen sind nur erforderlich, wenn ein neuer Datensatz angelegt oder ein vorhandener Datensatz gelöscht wird. Die Methode `addDvd()` ruft eine weitere private Methode namens `persistDvd()` auf, die auch beim Ändern eines DVD-Datensatzes gebraucht wird:

```
public boolean addDvd(DVD dvd) throws IOException {
    return persistDvd(dvd, true);
}
```

[95] Die `persistDvd()`-Methode beginnt mit der Prüfung, ob der übergebene UPC bereits in der Datenbankdatei registriert ist und ob ein neuer Datensatz angelegt oder ein vorhandener Datensatz aktualisiert werden soll. Beim Anlegen eines neuen Datensatzes, darf der UPC noch nicht registriert sein. Beim Aktualisieren eines vorhandenen Datensatzes dagegen, muß der UPC registriert sein. In allen übrigen Fällen geben wir `false` zurück, um anzuzeigen, daß die Operation gescheitert ist. Da eventuell ein neuer Eintrag angelegt wird, fordern wir die Schreibsperre des von `recordNumbersLock` referenzierten `ReentrantReadWriteLock`-Objektes an.

Bemerkung: In der Regel sollen Rückgabewerte nicht verwendet werden, um Erfolg oder Mißerfolg eines Methodenaufrufs anzuzeigen. Es ist besser eine Ausnahme auszuwerfen, wenn der Programmablauf nicht fortgesetzt werden kann. Eine Ausnahme gestattet eine detaillierte Beschreibung des aufgetretenen Fehlers und außerdem kann der Stackinhalt ausgegeben und ausgewertet werden. Allerdings verlangt das Interface *`DBClient`*, daß die Methoden `addDVD()`, `modifyDVD()` und `removeDVD()` je eine `Boolean`-Referenz zurückgeben müssen, deren Inhalt Erfolg oder Scheitern des Methodenaufrufs anzeigt. Wir müssen dem Diktat des vorgegebenen Interfaces gehorchen, um nicht zu riskieren, daß eine Anwendung eines anderen Entwicklers scheitert. Wie bei der Prüfungsaufgabe, die Sie von Sun Microsystems erhalten, verwenden wir ein Interface, bei dem die Rückgabewerte, Parameter und das Ausnahmeverhalten der Methoden nicht (immer vollständig) beschrieben sind.

---

[96] Das Verlassen der `persistDvd()`-Methode, ohne die Schreibsperre des von `recordNumbersLock` referenzierten `ReentrantReadWriteLock`-Objektes aufzuheben, kann verheerende Folgen haben: Kein anderer Thread wäre in der Lage, eine Lese- oder Schreibsperre von diesem `ReentrantReadWriteLock`-Objekt anzufordern, wodurch die meisten Methoden nicht mehr funktionsfähig würden. Wir legen daher unmittelbar nach der Zeile, die die Schreibsperre anfordert eine `try-finally`-Kombination an und heben die Sperre in der `finally`-Klausel in Zeile 357 wieder auf:

```
private boolean persistDvd(DVD dvd, boolean create) throws IOException {
    log.entering("DvdFileAccess", "persistDvd", dvd);

    // Perform as many operations as we can outside of the synchronized
    // block to improve concurrent operations.
```

```
Long offset = 0L;
recordNumbersLock.readLock().lock();
try {
    offset = recordNumbers.get(dvd.getUPC());
} finally {
    recordNumbersLock.readLock().unlock();
}
if (create == true && offset == null) {
    recordNumbersLock.writeLock().lock();
    try {
        offset = database.length();
        recordNumbers.put(dvd.getUPC(), offset);
    } finally {
        recordNumbersLock.writeLock().unlock();
    }
    log.info("creating record " + dvd.getUPC());
} else if (create == false && offset != null) {
    log.info("updating existing record " + dvd.getUPC());
} else {
    return false;
}
```

[97] Die `persistDvd()`-Methode besitzt eine lokale Variable `out` vom Typ `StringBuilder`, die wir verwenden, um einen vollständigen Datensatz darzustellen. Vor Version 5 des Java Development Kits wäre die Wahl möglicherweise auf ein `StringBuffer`-Objekt gefallen. (Die Klasse `StringBuilder` ist erst seit Version 5 des Java Development Kits verfügbar.) `StringBuffer` ist allerdings eine synchronisierte Klasse und somit threadsicher. Da `out` eine lokale Variable ist, ist kein synchronisierter Zugriff erforderlich. Die unsynchronisierte Klasse `StringBuilder` ist außerdem performanter als `StringBuffer`.

[98] Alle Datensätze haben eine feste einheitliche Länge. Wir erzeugen zunächst eine Datensatzvorlage entsprechender Länge und ersetzen die dabei verwendeten Platzhalterzeichen (das nicht druckbare ASCII-Zeichen `NUL`) anschließend durch die Feldinhalte. Wir haben in Zeile 66 das statische Feld `emptyRecordString` angelegt und in den Zeilen 77–79 mit Hilfe eines statischen Initialisierungsblocks bewertet (Seite 130). Das von `emptyRecordString` referenzierte `String`-Hilfsobjekt gestattet uns, ein `StringBuilder`-Objekt mit einheitlicher Länge und Platzhaltern als Inhalt zu erzeugen:

```
360. final StringBuilder out = new StringBuilder(emptyRecordString);
```

[99] Wir verwenden nun die `StringBuilder`-Methode `replace()`, um die Platzhalterzeichen in der `StringBuilder`-Vorlage durch die Inhalte der einzelnen DVD-Felder zu ersetzen. Die lokale innere Klasse `RecordFieldWriter` vermeidet dabei das Duplizieren der benötigten Anweisungen.

```
362. class RecordFieldWriter {
363.     private int currentPosition = 0;
364.     void write(String data, int length) {
365.         out.replace(currentPosition,
366.                     currentPosition + data.length(),
367.                     data);
368.         currentPosition += length;
369.     }
370. }
371. RecordFieldWriter writeRecord = new RecordFieldWriter();
```

[100] Anschließend verwenden wir diese Hilfsklasse, um das `DVD`-Objekt in ein gleichwertiges `StringBuilder`-Objekt umzuwandeln:

```
373. writeRecord.write(dvd.getUPC(), DVD.UPC_LENGTH);
```

```
374. writeRecord.write(dvd.getName(), DVD.NAME_LENGTH);
375. writeRecord.write(dvd.getComposer(), DVD.COMPOSER_LENGTH);
376. writeRecord.write(dvd.getDirector(), DVD.DIRECTOR_LENGTH);
377. writeRecord.write(dvd.getLeadActor(), DVD.LEAD_ACTOR_LENGTH);
378. writeRecord.write(dvd.getSupportingActor(),
379.                   DVD.SUPPORTING_ACTOR_LENGTH);
380. writeRecord.write(dvd.getYear(), DVD.YEAR_LENGTH);
381. writeRecord.write("" + dvd.getCopy(), DVD.COPIES_LENGTH);
```

Warnung: Fällen Sie Ihre Design-Entscheidungen bei der Arbeit an Ihrer Prüfungsaufgabe so, daß Ihre Entscheidungen auf Sie sinnvoll wirken und Sie sie in der schriftlichen Prüfung verteidigen können. Dies kann bedeuten, daß eine Ihrer Entscheidungen, einer unserer Entscheidungen widerspricht. Das Beispielprojekt *Denny's DVDs* und seine Anforderungen unterscheiden sich schließlich von Ihrer Prüfungsaufgabe und den Anforderungen die Sie erfüllen müssen. Zögern Sie nicht, Alternativen in Betracht zu ziehen. Die Java-Ranch (*http://www.javaranch.com*) ist ein gutes Forum, um Ihre und unsere Design-Entscheidungen zu diskutieren.

[101] Schließlich schreiben wir den Datensatz in die Datenbankdatei und geben `true` zurück, um anzuzeigen, daß der Datensatz persistent gespeichert wurde:

```
384.     // now that we have everything ready to go, we can go into our
385.     // synchronized block & perform our operations as quickly as possible
386.     // ensuring that we block other users for as little time as possible.
387.
388.     synchronized (database) {
389.         database.seek(offset);
390.         database.write(out.toString().getBytes());
391.     }
392.
393.     log.exiting("DvdFileAccess", "persistDvd", true);
394.     return true;
395. }
```

[102] Die meisten der übrigen Methoden benötigen keine Erläuterungen. Wir beschließen diesen Abschnitt mit der `find()`-Methode, die sowohl die in Version 5 des Java Development Kits eingeführte erweiterte `for`-Schleife als auch einen regulären Ausdruck (seit Version 1.4 des Java Development Kits vorhanden) verwendet:

```
311. public Collection<DVD> find(String query)
312.         throws IOException, PatternSyntaxException {
313.     log.entering("DvdFileAccess", "find", query);
314.     Collection<DVD> returnValue = new ArrayList<DVD>();
315.     Pattern p = Pattern.compile(query);
316.
317.     for (DVD dvd : getDvds()) {
318.         Matcher m = p.matcher(dvd.toString());
319.         if (m.find()) {
320.             returnValue.add(dvd);
321.         }
322.     }
323.
324.     log.exiting("DvdFileAccess", "find", returnValue);
325.     return returnValue;
326. }
```

[103] In Zeile 315 wird die von der graphischen Benutzeroberfläche erfaßte Zeichenkette in ein `java-`

`.util.regex.Pattern`-Objekt umgewandelt. Wir benötigen außerdem ein `java.util.regex.Matcher`-Objekt, um einen Datensatz mit dem Muster vergleichen zu können. Das `Matcher`-Objekt wird für jeden Datensatz neu erzeugt (Zeile 318). In Zeile 319 suchen wir nach dem nächsten Vorkommen des Musters in der Zeichenkettendarstellung der aktuellen DVD. Falls ein Treffer gefunden wird, nehmen wir das `DVD`-Objekt in die von `find()` zurückgegebene Kollektion von `DVD`-Objekten auf. Wir interessieren uns lediglich dafür, ob ein Teil der Zeichenkettendarstellung eines `DVD`-Objektes mit dem Muster übereinstimmt. Das `Matcher`-Objekt kann aber auch noch viele andere Vergleichsoperationen durchführen, zum Beispiel zwei Zeichenketten auf gesamter Länge vergleichen oder feststellen, ob eine Zeichenkette mit einem bestimmten Anfangsstück beginnt.

[104] Zeile 317 demonstriert die Verwendung der erweiterten `for`-Schleife. Die `for`-Schleife verhindert, daß wir manuell einen eigenen Iterator erzeugen müssen und die Verwendung einer generischen Kollektion erspart uns die Typumwandlung der Elemente. Vergleichen Sie die elegante Lösung in Zeile 317 mit der Alternative ohne erweiterte `for`-Schleife:

```
317a. List<DVD> dvds = getDVDs();
317b. for (Iterator i = dvds.iterator(); i.hasNext(); ) {
317c.     DVD dvd = i.next();
318.      Matcher m = p.matcher(dvd.toString());
319.      if (m.find()) {
320.          returnValue.add(dvd);
321.      }
322.  }
```

5.3.1 Diskussion: Zwischenspeicherung von Datensätzen

[105] Wir haben die Frage, ob und wie sich der Zugriff auf die Datenbankdatei beschleunigen läßt, bis jetzt nur am Rande berücksichtigt. Jeder Datensatz wird aus einer physikalischen Datei auf der Festplatte gelesen (unter Umständen die langsamste Operation in der gesamten Beispielanwendung). Wir haben dieses Thema bewußt nicht in die Diskussion eingeflochten, um dieses Kapitel nicht übermäßig kompliziert zu machen. Dennoch wollen wir in diesem Unterabschnitt einige Diskussionsansätze für die Frage zusammenstellen, ob sich die Zwischenspeicherung von Datensätzen in der Beispielanwendung lohnt oder nicht.

[106] Die erste Frage, mit der wir uns auseinandersetzen sollten, lautet: „Reduziert das Zwischenspeichern der Datensätze die Anzahl der Festplattenzugriffe?“ Beinhalten die meisten Operationen auf den Daten Schreibzugriffe auf die Festplatte, so ist die Zwischenspeicherung nicht sinnvoll, da wir bei diesen Operationen sowohl den Zwischenspeicher als auch die Datenbankdatei aktualisieren müssen. Tatsächlich könnte die Performanz einer Anwendung, verglichen mit einer Lösung ohne Zwischenspeicherung, in diesem Fall sogar geringer ausfallen. Bei unserer Beispielanwendung sind aber viele Such- und Anzeigeoperationen für passende Datensätze zu erwarten, bevor ein Datensatz in der Datenbankdatei überschrieben wird. Die Mehrzahl der Operationen könnte also von der Zwischenspeicherung der Datensätze profitieren.

[107] Die nächste Frage lautet: „Spricht der Bedarf an Arbeitsspeicher gegen die Zwischenspeicherung von Datensätzen?“ Ein DVD-Datensatz ist relativ klein und wahrscheinlich wird die Zwischenspeicherung von 1000 DVD-Datensätzen deutlich weniger als 10MB des Arbeitsspeichers beanspruchen. Da die meisten heutigen Rechner mit Arbeitspeicher in der Größenordnung von Giga-Bytes (GB) ausgeliefert werden, ist die Zwischenspeicherung von Datensätzen im Hinblick auf den Speicherbedarf unproblematisch.

[108] Die letzte Frage in diesem Unterabschnitt lautet: „Wie wirkt sich die Implementierung der Zwischenspeicherung für die Datensätze auf die Komplexität des Quelltextes aus?“ Verschlechtert die

Implementierung einer neuen Eigenschaft oder Fähigkeit einer Anwendung die Lesbarkeit des Quelltextes erheblich, so lohnt sich die Überlegung, ob der Vorteil durch die Existenz dieser Eigenschaft oder Fähigkeit den Nachteil des komplizierteren Quelltextes rechtfertigt (auch im Hinblick auf den zukünftigen Wartungsprogrammierer, der den Quelltext pflegen muß). Das Anlegen eines Zwischenspeichers im Quelltext der Beispielanwendung ist aber recht einfach: Wir brauchen nicht mehr, als ein zusätzliches `Map`-Objekt, um die UPCs (Schlüssel) auf die DVD-Datensätze (Werte) abzubilden. Die Threadsicherheit beim Zugriff auf diesen Zwischenspeicher wird analog wie bei der Abbildung der UPCs auf die Datensatzpositionen durch ein Sperrobjekt vom Typ `ReentrantReadWriteLock` gewährleistet.

[109] Lesen Sie in der Anleitung Ihrer Prüfungsaufgabe sorgfältig nach, bevor Sie sich für oder gegen die Zwischenspeicherung von Datensätzen entscheiden. Suchen Sie, ob es Anforderungen hinsichtlich der Performanz oder der Einfachheit des Quelltextes gibt. Wenn die Anleitung keine entsprechenden Anforderungen enthält, haben Sie die Wahl, ob Sie Ihre Datensätze zwischenspeichern oder nicht. Denken Sie aber daran, Ihre Entscheidung in der Dokumentation Ihrer Design-Entscheidungen festzuhalten.

5.4 Die Klasse ReservationsManager

[110] Die Klasse `sampleproject.db.ReservationsManager` implementiert das logische Reservieren (Sperren) eines Datensatzes. Ein logisch reservierter Datensatz kann stets nur von einem einzigen Benutzer (Thread) geändert werden. Andere Threads, die ebenfalls eine Modifikation an diesem Datensatz durchzuführen haben, müssen warten, bis die logische Reservierung wieder aufgehoben wird.

[111] Zur Erläuterung der Notwendigkeit einer logischen Sperre für die Datensätze betrachten wir zunächst eine Beispielsituation ohne einen solchen Mechanismus. Zwei Benutzer versuchen, das einzige verfügbare Exemplar einer bestimmten DVD auszuleihen:

- Benutzer A (Thread) ruft den Datensatz für den Film „Alles Routine“ (orig. „Office Space“) ab.
- Benutzer B (Thread) ruft den Datensatz für „Alles Routine“ ab.
- Benutzer A verifiziert, daß noch ein Exemplar vorhanden ist.
- Benutzer B verifiziert, daß noch ein Exemplar vorhanden ist.
- Benutzer A verleiht die DVD.
- Benutzer B verleiht die DVD.

[112/113] Problem: Die beiden elektronischen Datensätzen sagen aus, daß beide Benutzer ein und dieselbe DVD ausgeliehen haben. Eine mögliche Lösung besteht darin, die Operation „Abrufen, Verifizieren und Ausleihen“ atomar zu machen, läßt sich allerdings nur serverseitig realisieren, da Thread 1 in Laufzeitumgebung 1 nicht feststellen kann, ob sich Thread 2 in Laufzeitgebung 2 in einem synchronisierten Block befindet oder nicht. Einerseits würde die serverseitige Verarbeitung dieser Operationen die Entwicklung eines Thin-Clients (siehe unten) deutlich vereinfachen. Andererseits ist aber bekannt, daß wir eine Swing-basierte graphische Benutzeroberfläche einreichen müssen, der Client-Rechner also auch Thick-Clients (siehe unten) unterstützt. Das größere Problem besteht darin, daß die Zusammenfassung der obigen drei Operationen zu einer atomaren Operation die parallele Verarbeitung mehrerer Threads beeinträchtigt.

Thin-Clients und Thick-Clients

Der Begriff „Thin-Client“ bezeichnet eine Benutzerschnittstelle, die auf einem Rechner mit minimaler Prozessorleistung betrieben werden kann. Webbrowserbasierte Anwendungsschnittstellen sind ein typisches Beispiel: Ein Rechner auf dem eine solche Benutzerschnittstelle läuft, kommt mit sehr wenig Leistung aus und benötigt eventuell weder eine Festplatte noch ein Diskettenlaufwerk, um auf eine Anwendung zugreifen zu können, wenn die Anwendung einen Thin-Client zur Verfügung stellt. Ein PC mit 368er-Prozessor, 16MHz Taktfrequenz und 4MB Arbeitsspeicher genügt, um einen Webbrowser zu betreiben, ein E-Mail-Programm zu bedienen, Newsgruppen zu lesen und im Internet zu suchen.

Für diejenigen Leser, die jünger als die Autoren sind: Vor den heutigen PCs mit Pentium-IV-Prozessoren, 2GHz Taktfrequenz und Arbeitspeicher in der Größenordnung von GB, gab es PCs mit Pentium-III-Prozessoren, davor Pentium-II-, Pentium-, 468er- und 368er-Prozessoren. Wiederum vor den 368er- gab es 268er-, 8068er- und 8088er-Prozessoren. Aber auch wir würden keinen Webbrowser auf einem so alten Rechner betreiben (das erste unixartige System auf dem einer von uns gearbeitet hat, war „Coherent“ auf einem 268er-Prozessor).

Der Begriff „Thick-Client“ (oder „Fat-Client“) bezeichnet eine Benutzerschnittstelle, bei der ein beträchtlicher Anteil der Verarbeitung auf dem benutzerseitigen Rechner durchgeführt wird und daher mehr Rechenleistung erforderlich ist. Beispielsweise empfiehlt Microsoft für Microsoft Office einen Rechner mit wenigstens einem Pentium-III-Prozessor und 128MB Arbeitspeicher.

Der Trend geht in Unternehmen aus verschiedenen Gründen in Richtung Thin-Clients. Beispielsweise sind Thin-Clients typischerweise auf älteren Rechnern lauffähig (der 368er-Prozessor kam 1985 auf den Markt, der Pentium-III-Prozessor 1999; 14 Jahre Prozessorentwicklung wären technisch überholt, wenn Software grundsätzlich einen Thick-Client mit Pentium-III-Prozessor voraussetzen würde) und Systemadministratoren haben weniger Arbeit, wenn sie nur einen oder zwei Server mit den Anwendungen betreuen müssen und die Rechner der Benutzer nur Thin-Clients benutzen.

Warnung: Lesen Sie sorgfältig in der Anleitung zu Ihrer Prüfungsaufgabe nach, bevor Sie sich für einen Thin- oder einen Thick-Client entscheiden. Die Anleitung kann diesbezügliche Anforderungen enthalten, die Sie berücksichtigen müssen. Es gibt auf der Java-Ranch (*http://www.javaranch.com*) viele Diskussionen zu diesem Thema und Sie sind willkommen, sich an daran zu beteiligen.

---

[114] Wir betrachten nun die obige Situation nochmals, diesesmal aber mit einer logischen Reservierung für die Datensätze:

- Benutzer A (Thread) reserviert den Datensatz für den Film „Alles Routine“ (orig. „Office Space“) logisch.
- Benutzer B (Thread) versucht, den Datensatz für den Film „Alles Routine“ logisch zu reservieren. Die Reservierung kann aber erst erfolgen, nachdem Benutzer A die logische Sperre des Datensatzes wieder aufgehoben hat. Benutzer B muß warten, bis Benutzer A fertig ist.
- Benutzer A ruft den Datensatz für den Film „Alles Routine“ ab.
- Benutzer A verifiziert, daß noch ein Exemplar vorhanden ist.
- Benutzer A verleiht die DVD an seinen Kunden.
- Benutzer A hebt die logische Reservierung des Datensatzes von „Alles Routine“ wieder auf.
- Benutzer B sperrt den Datensatz für den Film „Alles Routine“ logisch.

- Benutzer B ruft den Datensatz für den Film „Alles Routine“ ab.
- Benutzer B stellt fest, daß kein Exemplar mehr vorhanden ist.
- Benutzer B hebt die logische Sperre des Datensatzes von „Alles Routine“ wieder auf und sucht nach einer anderen DVD.

[115] Einer der wesentlichen Vorteile einer logischen Reservierung (Sperre) für die Datensätze besteht in der besseren parallelen Verarbeitung mehrerer Threads. Der Mechanismus kann auch über ein Netzwerk hinweg verwendet werden, so daß die Rechenleistung der Thick-Clients benutzt wird.

5.4.1 Diskussion: Identifizierung des Besitzers einer Sperre

[116] Das logische Reservieren (Sperren) eines Datensatzes und das Aufheben der Reservierung reichen in der Regel nicht aus. Sie benötigen typischerweise die Information, welcher Benutzer (Thread) den spezifischen Datensatz reserviert hat, um zu gewährleisten, daß nur dieser Benutzer die Reservierung wieder aufheben kann.

[117] Ein praktisches Beispiel zur Erläuterung, warum diese Information wichtig ist: Stellen Sie sich vor, Sie rufen an der Theaterkasse an und reservieren den letzten freien Platz für eine Vorstellung. Nun kommt ein anderer Theatergast an die Kasse und behauptet, er habe Ihren Platz reserviert. Wenn die Kassiererin keine Möglichkeit hat, den Kunden der den Platz reserviert hat zu identifizieren, verlieren Sie eventuell Ihren Platz.

[118] Wenn wir den Besitzer einer Reservierung nicht identifizieren, könnte ein gewissenloser Entwickler ein Programm schreiben, daß die Reservierung nach Ablauf einer bestimmten Wartezeit aufhebt, falls es der Besitzer nicht selbst tut.

[119] Wie Sie den Besitzer eine Reservierung identifizieren, hängt davon ab, was in Ihrer Anleitung steht, eventuell auch davon, wie Sie den Server Ihrer Anwendung implementieren. Wir bieten im Folgenden einige Vorschläge an. Sie müssen allerdings selbst ermitteln, welcher Ansatz zu den Anforderungen Ihrer Prüfungsaufgabe paßt.

5.4.1.1 Identifizierung anhand eines Merkmals

[120] Wenn die Reservierungsmethode ein Merkmal an der Client zurückgibt, der sie aufgerufen hat, können wir festlegen, daß der Client dieses Merkmal bei allen Operationen übergeben muß, insbesondere also zum Aufheben der logischen Reservierung.

[121] Das Merkmal (auch als „Cookie“ oder „Magic Cookie“ bezeichnet) ist eine Kennzeichnung, die der Server verwendet, um den Besitzer der logischen Reservierung zu identifizieren. Das Merkmal muß kein bedeutungstragendes Objekt sein, da es keine andere Aufgabe hat, als vom Client gespeichert und beim Aufrufen von Methoden wieder übergeben zu werden. Es ist sogar besser, wenn das Merkmal zufällig gewählt wird. Falls es nämlich eine Bedeutung hat, könnte sie von einem skrupellosen Entwickler erraten und die Reservierung dennoch aufgehoben werden.

[122] Unser Interface *`DBClient`* deklariert allerdings, daß die Reservierungsmethode `reserveDVD()` einen `boolean`-Wert zurückgibt und die Methode `releaseDVD()` kein Merkmal, sondern nur einen UPC erwartet. Daher scheidet dieser Ansatz für das Beispielprojekt aus.

5.4.1.2 Identifizierung anhand des Threads

[123] Wählen wir Sockets als Netzwerkschnittstelle, so haben wir vollständige Kontrolle über die von den einzelnen Clients serverseitig gestarteten Threads. Verbindet sich der Client mit dem Server, so wird serverseitig ein neuer Thread erzeugt, dessen Namen wir als eindeutige Kennzeichnung des Clients verwenden können.

[124] Bei diesem Ansatz speichern wir den Namen des Threads zusammen mit einer eindeutigen Kennzeichnung des Datensatzes, wenn wir den Datensatz logisch reservieren. Versucht ein Client eine Operation auf einem Datensatz auszuführen, für die dessen Reservierung erforderlich ist, so vergleichen wir den Namen des entsprechenden Threads mit dem gespeicherten Namen. Stimmen beide Namen überein, lassen wir die Operation zu.

[125] Dieser Ansatz erspart dem Client zwar das Speichern und Übergeben eines Merkmals, funktioniert aber nur im *stand-alone*-Betriebsmodus der Anwendung sowie im Netzwerkmodus, wenn Sockets als Netzwerkschnittstelle gewählt wurden. Bei Remote Method Invocation (RMI) als Netzwerkschnittstelle, scheidet dieser Ansatz aus.

5.4.1.3 Identifizierung anhand eines individuellen DvdDatabase-Objektes

[126] Abschnitt „3.2 Thread Usage in Remote Method Invocations“ der RMI-Spezifikation weist darauf hin, daß nicht garantiert werden kann, welcher Thread zur Ausführung einer entfernten Methode verwendet wird. Die folgende Situation ist gemäß der RMI-Spezifikation möglich:

- Client A verwendet den serverseitigen Thread 1, um einen Datensatz logisch zu reservieren.
- Client B verwendet den serverseitigen Thread 1, um zu versuchen, denselben Datensatz logisch zu reservieren. Thread 1 wartet, bis die logische Reservierung aufgehoben wird.
- Client A verwendet den serverseitigen Thread 2, um die logische Reservierung des Datensatzes aufzuheben.
- Client B verwendet den serverseitigen Thread 1, um die logische Reservierung des Datensatzes aufzuheben.

[127] Auch kompliziertere Situationen sind denkbar. Das folgende Beispiel führt die mehrfache Verwendung eines serverseitigen Threads bei RMI vor:

```
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

interface ServerReference extends Remote {
    public void serverThreadNumber(String id) throws RemoteException;
}

class Server extends UnicastRemoteObject implements ServerReference {
    public Server() throws RemoteException {
        // do nothing constructor
    }

    public void serverThreadNumber(String id) throws RemoteException {
        System.out.println(id + " running in thread "
                           + Thread.currentThread().hashCode());
        try {
            Thread.sleep(2000);
        } catch (InterruptedException ie) {
```

```
                ie.printStackTrace();
                System.exit(1);
            }
        }
    }

    public class RmiProblem extends Thread {
        public RmiProblem(String id) {
            super(id);
        }

        public static void main(String[] args) throws Exception {
            LocateRegistry.createRegistry(1099);
            Naming.rebind("rmi://localhost:1099/RmiProblem", new Server());

            Thread a = new RmiProblem("A");
            a.start();

            Thread.sleep(1000);

            Thread b = new RmiProblem("B");
            b.start();

            a.join();
            b.join();

            System.exit(0);
        }

        public void run() {
            try {
                ServerReference remoteCode =
                        (ServerReference) Naming.lookup("RmiProblem");

                for (int i = 0; i < 5; i++) {
                    remoteCode.serverThreadNumber(getName());
                    Thread.sleep(2000);
                }
            } catch (Exception e) {
                e.printStackTrace();
                System.exit(0);
            }
        }
    }
```

[128/129] Machen Sie sich keine Sorgen, wenn Sie diesen Quelltext nicht komplett verstehen. Kapitel 6 diskutiert RMI ausführlich und in allen Einzelheiten. Kehren Sie nach dem Lesen von Kapitel 6 zu diesem Beispiel zurück. Das Ergebnis dieses Programms variiert von Rechner zu Rechner (sogar von Aufruf zu Aufruf). Die Ausgabe eines unserer Aufrufe lautet:

```
~$ java RmiProblem

A running in thread 26780757
B running in thread 21764429
A running in thread 21764429
B running in thread 26780757
A running in thread 26780757
B running in thread 21764429
A running in thread 21764429
B running in thread 26780757
A running in thread 26780757
B running in thread 21764429
```

```
~$
```

[130/131] Die Ausgabe dokumentiert, daß sowohl eine Anfrage von Client A (Aufruf der entfernten Methode `serverThreadNumber()`) als auch eine Anfrage von Client B vom serverseitigen Thread `26780757` verarbeitet wurden (analog für Thread `21764429`). Falls Merkmale zum Identifizieren der Clients nicht erlaubt sind, wir aber RMI als Netzwerkschnittstelle wählen möchten, müssen wir uns nach einem anderen Verfahren umsehen, um den Verursacher einer Reservierung eindeutig feststellen zu können.

[132] Ein möglicher Ansatz besteht darin, daß der Server das Entwurfsmuster *Factory* implementiert, wobei das Fabrikobjekt für jeden mit dem Server verbundenen Client ein eindeutiges `DvdDatabase`-Objekt erzeugt, über das der Client identifiziert werden kann (siehe Seite 166f).

[133] Die Implementierung des *Factory*-Entwurfsmuster kann sowohl bei Sockets als auch bei RMI als Netzwerkschnittstelle verwendet werden, ist bei Sockets aber übertrieben. Da dieses Buch beide Ansätze beschreibt, haben wir uns für eine RMI-Lösung als *Factory* entschieden (siehe Kapitel 6).

5.4.2 Die logische Methode reserveDvd()

[134] Die Funktionsweise der für die logische Reservierung eines Datensatz beziehungsweise deren Aufhebung verantwortlichen `ReservationsManager`-Methoden `reserveDvd()` und `releaseDvd()` (nächster Unterabschnitt) setzt drei statische Felder voraus. Erstens müssen wir den Besitzer einer Reservierung protokollieren, wofür sich ein *`Map`*-Objekt anbietet, das die UPC auf die eindeutige Besitzerkennzeichnung abbildet (hier eine Referenz auf das individuelle `DvdDatabase`-Objekt). Zweitens brauchen wir ein von allen Threads gemeinsam verwendetes Sperrobjekt, um zu verhindern, daß zwei verschiedene Threads gleichzeitig denselben Datensatz reservieren. Drittens brauchen wir eine Bedingung, die die Threads beobachten können, um festzustellen, wann sie erneut versuchen können, einen Datensatz zu reservieren. Die drei statischen Felder `reservations`, `lock` und `lockReleased` sind wie folgt definiert:

```
private static Map<String, DvdDatabase> reservations
        = new HashMap<String, DvdDatabase>();

private static Lock lock = new ReentrantLock();
private static Condition lockReleased  = lock.newCondition();
```

Der Quelltext der Methode `reserveDvd()` lautet:

```
boolean reserveDvd(String upc, DvdDatabase renter)
        throws InterruptedException {
    log.entering("ReservationsManager", "reserveDvd", new Object[]{upc, renter});

    lock.lock();
    try {
        long endTimeMSec = System.currentTimeMillis() + TIMEOUT;
        while (reservations.containsKey(upc)) {
            long timeLeftMSec = endTimeMSec - System.currentTimeMillis();
            if (!lockReleased.await(timeLeftMSec, TimeUnit.MILLISECONDS)) {
                log.fine(renter + " giving up after 5 seconds: " + upc);
                return false;
            }
        }
        reservations.put(upc, renter);
        log.fine(renter + " got Lock for " + upc);
        log.fine("Locked record count = " + reservations.size());
        log.exiting("ReservationsManager", "reserveDvd", true);
```

```
            return true;
        } finally {
            // ensure lock is always released, even if an Exception is thrown
            lock.unlock();
        }
    }
```

[135] Wir haben uns dafür entschieden, daß jeder Datensatz reserviert werden darf, insbesondere also ein neu angelegter Datensatz, der noch nicht in der Datenbankdatei registriert ist. Dadurch kann ein DVD-Datensatz bereits beim Neuanlegen in der Datenbankdatei reserviert werden. Alternativ könnte zu Beginn der `reserveDvd()`-Methode geprüft werden, ob der Datensatz vorhanden ist und eine Ausnahme ausgeworfen werden, falls er nicht existiert. Hätten wir eine Methode zum Löschen von Datensätzen, so müßten wir nach dem Reservieren eines Datensatzes ebenfalls verifizieren, daß der Datensatz bereits existiert. Außerdem müßten wir auch die Logik der beiden `DvdFileAccess`-Methoden `addDvd()` und `modifyDvd()` ändern (beide rufen die private `DvdFileAccess`-Methode `persistDvd()` auf, die eventuell ebenfalls geändert werden muß).

[136] Die `reserveDvd()`-Methode sperrt das von `lock` referenzierte `ReentrantLock`-Objekt und tritt anschließend in eine `while`-Schleife ein. In der Schleife wird entweder die Reservierung des Datensatzes aufgehoben oder die Methode wegen Zeitüberschreitung abgebrochen.

[137] Das Interface *`DBClient`* legt fest, daß die Methode `reserveDVD()` den Rückgabewert `false` liefern muß, wenn der Datensatz nicht innerhalb von fünf Sekunden reserviert werden kann. In Version 1.4 des Java Development Kits gab es keine Möglichkeit, um garantiert zu bestimmen, ob während eines `wait(timeout)`-Aufrufs die Zeitüberschreitung eingetreten oder der Thread benachrichtigt worden war. Version 5 des Java Development Kits verfügt über die *`Condition`*-Methode `await()`, die `true` zurückgibt, falls der Thread per `unlock()` benachrichtigt wurde, aber `false` bei Zeitüberschreitung.

[138/139] Kann der Datensatz reserviert werden, so fügen wir ihn zusammen mit dem von `renter` referenzierten clientspezifischen `DvdDatabase`-Objekt der Buchhaltung der reservierten Datensätze hinzu (von `reservations` referenziertes `HashMap`-Objekt). Schließlich wird die Sperre des `ReentrantLock`-Objektes wieder aufgehoben.

5.4.3 Die logische Methode releaseDvd()

[140] Die `ReservationsManager`-Methode `releaseDVD()` ist das Gegenstück zu `reserveDVD()` (voriger Unterabschnitt) und ähnlich programmiert:

```
void releaseDvd(String upc, DvdDatabase renter) {
    log.entering("ReservationsManager", "releaseDvd", new Object[]{upc, renter});
    lock.lock();
    if (reservations.get(upc) == renter) {
        reservations.remove(upc);
        log.fine(renter + " released lock for " + upc);
        lockReleased.signal();
    } else {
        log.fine(renter + " can't release " + upc + " lock (not owner)");
    }
    lock.unlock();
    log.exiting("ReservationsManager", "releaseDvd");
}
```

[141] Zunächst sperren wir das `ReentrantLock`-Objekt. Stimmt der Besitzer der Reservierung mit dem Thread überein, der die Reservierung aufheben möchte, so entfernen wir die Reservierung aus

der Buchhaltung. Anschließend werden alle wartenden Threads benachrichtigt, daß sie versuchen können, den Datensatz zu reservieren. Wird der Methode dagegen nicht das richtige `DvdDatabase`-Objekt übergeben (`renter`), wird eine Warnmeldung protokolliert. Schließlich wird die Sperre des `ReentrantLock`-Objektes wieder aufgehoben.

5.4.3.1 Diskussion: Behandlung von Verklemmungen

[142] Eine Verklemmung (*dead lock*) liegt vor, wenn ein Thread dauerhaft blockiert ist, also auf eine Bedingung wartet, die nicht erfüllt werden kann. Kapitel 4 diskutiert die Behandlung von Verklemmungen aus der Perpektive der einzelnen Threads, aber die dort beschriebenen Probleme können auch auf der Ebene der Anwendungslogik auftreten. Beispiel: Zwei Clients versuchen, dieselben beiden Datensätze in entgegengesetzter Reihenfolge logisch zu reservieren:

- Client A reserviert Datensatz 1 logisch.
- Client B reserviert Datensatz 2 logisch.
- Client A versucht, Datensatz 2 logisch zu reservieren.
- Client B versucht, Datensatz 1 logisch zu reservieren.

[143] Unsere `reserveDvd()`-Methode implementiert eine Zeitüberschreitung von fünf Sekunden. Versuchen Client A und Client B unmittelbar danach, ihren jeweiligen Datensatz logisch zu reservieren, so ergibt sich dieselbe Situation wie ohne Zeitüberschreitung: Beide Threads wären effektiv verklemmt.

[144] Es gibt eine Reihe von Lösungsansätzen, um Verklemmungen zu verhindern, darunter:

- Ein Client darf stets höchstens einen Datensatz logisch reservieren. Wenn kein Client mehr als einen Datensatz logisch reservieren kann, ist eine logische Verklemmung nicht möglich.
- Clients dürfen Datensätze nur in numerischer Reihenfolge logisch reservieren. Bei dieser Regel ist es Client B nicht erlaubt, Datensatz 1 logisch zu reservieren. Client B hebt die Reservierung von Datensatz 2 irgendwann auf, woraufhin Client A fortfahren kann.
- Verwenden Sie einen Ansatz, um zu Protokollieren, welche Sperren aktiv und welche inaktiv sind. Jedesmal, wenn eine neue Sperre aktiviert wird, prüft das Verfahren, ob eine Verklemmung möglich ist und verweigert gegebenenfalls die Sperre. Dies ist zwar die komplizierteste der möglichen Lösungen, aber die benötigten Anweisungen basieren auf Rekursion und lassen sich mit etwas Überlegung leicht entwickeln.
- Ignorieren Sie das Problem. Ernsthaft: Verlangt Ihre Prüfungsaufgabe, daß Sie dieses Problem lösen? Wie wahrscheinlich führt der Versuch, dieses Problem zu lösen dazu, daß Sie einen Fehler machen, durch den Sie in der Bewertung Punkte verlieren? Haben Sie den Eindruck, daß dieses Problem nicht zu Ihrer Aufgabe gehört?

[145/146] Wir empfehlen auch zu dieser Frage, die Anleitung zu Ihrer Prüfungsaufgabe aufmerksam zu lesen, um festzustellen, ob die Behandlung von Verklemmungen zu Ihrer Aufgabe gehört. Wenn Ihre Anleitung diesen Punkt nicht anspricht, haben Sie die Wahl, ob Sie Verklemmungen behandeln wollen oder nicht. Eventuell beschäftigen Sie die Fragen, ob es nicht professioneller ist, Verklemmungen zu behandeln, ob die Behandlung den Quelltext unnötig verkompliziert oder ob Sie Verklemmungen mit Hilfe der Ausnahmen behandeln können, die Ihre Methoden auswerfen dürfen. Unabhängig davon wie Sie sich entscheiden: Dies ist eine Design-Entscheidung, die Sie dokumentieren sollten.

5.4.3.2 Diskussion: Behandlung von clientseitigen Verbindungsabbrüchen ~~(Baustelle)~~

[147] Wir haben in diesem Kapitel bereits besprochen, welche Möglichkeiten der Einsatz von Thick-Clients bietet (Seite 141), wenn der Client für das logische Sperren beziehungsweise Entsperren der Datensätze verantwortlich ist. Wir betrachten nun die Situation, daß der Client die Verbindung zum Server abbricht bevor die logische Reservierung eines Datensatzes wieder aufgehoben wird. Die Reservierung des Datensatzes kann nicht mehr aufgehoben, der Datensatz kann also von keinem anderem Client mehr reserviert werden. Es gibt wiederum mehrere Lösungsansätze, darunter:

- Verwenden Sie einen „dünneren“ Client (das heißt, einen Client, der nur eine serverseitige `rentDvd()`-Methode aufruft und der Server die Methoden `reserveDvd()` und `releaseDvd()` aufruft). Mit diesem Ansatz umgehen Sie das Problem voll und ganz. Eine Reservierung wird bei dieser Lösung niemals völlig unerreichbar. Lesen Sie die kurze Diskussion über Thin-/Thick-Clients und ihre Auswirkungen auf Seite 142f noch einmal und nehmen Sie an den entsprechenden Diskussionen auf der Java-Ranch teil.

- Bei Sockets als Netzwerkschnittstelle wird der serverseitige Thread, der die Anfrage des Clients verarbeitet in Form einer Ausnahme „benachrichtigt“, wenn der Client die Verbindung abbricht. Wenn sich der serverseitige Thread die gewährten logischen Reservierungen „merkt“, kann er sie beim Abfangen dieser Ausnahme aufheben.

- Bei RMI als Netzwerkschnittstelle und einer Server-Fabrik mit eindeutigen `DvdDatabase`-Objekten als Client-Identifikatoren, können wir die Reservierung in einem `java.util.WeakHashMap`-Objekt speichern (mit dem Client-Identifikator als Schlüssel). Bricht der Client die Verbindung ab, so wird der entsprechende Client-Identifikator (`DvdDatabase`-Objekt) letztendlich der automatischen Speicherbereinigung übergeben und die Reservierung automatisch aus dem `WeakHashMap`-Objekt gelöscht. Ein separater Thread kann das `WeakHashMap`-Objekt beobachten und die wartenden Threads benachrichtigen, wenn eine Reservierung aufgehoben wird.

- Bei RMI als Netzwerkschnittstelle und einer Server-Fabrik mit eindeutigen ~~workers~~ pro RMI-Client kann die ~~worker~~-Klasse das Interface *`java.rmi.server.Unreferenced`* implementieren. Die *`Unreferenced`*-Methode `unreferenced()` wird aufgerufen, nachdem der RMI-Client die Verbindung abgebrochen hat. Wenn sich das ~~worker~~-Objekt die gewährten Reservierungen „merkt“, kann es sie beim Aufrufen dieser Methode wieder aufheben.

- Ignorieren Sie das Problem. Ernsthaft (wie bei den Verklemmungen): Wie wahrscheinlich führt der Versuch, dieses Problem zu lösen dazu, daß Sie einen Fehler machen, durch den Sie in der Bewertung Punkte verlieren? Haben Sie den Eindruck, daß dieses Problem nicht zu Ihrer Aufgabe gehört?

[148/149] Wir empfehlen auch zu dieser Frage, die Anleitung zu Ihrer Prüfungsaufgabe aufmerksam zu lesen, um festzustellen, ob die Behandlung von clientseitigen Verbindungsabbrüchen zu Ihrer Aufgabe gehört. Wenn Ihre Anleitung diesen Punkt nicht anspricht, haben Sie die Wahl, ob Sie clientseitige Verbindungsabbrüche behandeln wollen oder nicht. Eventuell beschäftigen Sie die Fragen, ob es nicht professioneller ist, solche Verbindungsabbrüche zu behandeln oder ob die Behandlung den Quelltext unnötig verkompliziert. Unabhängig davon wie Sie sich entscheiden: Dies ist eine Design-Entscheidung, die Sie dokumentieren sollten.

5.4.3.3 Diskussion: Viele Sperrobjekte ~~(Baustelle)~~

[150] Angenommen, 100 Threads warten darauf, verschiedene Datensätze zu reservieren. Hebt nun einer der Threads seine Reservierung eines Datensatzes auf, so ruft er die Methode `lockRelea-`

`se.signal()` auf. Dadurch werden alle 100 Threads benachrichtigt, daß eine Reservierung aufgehoben wurde und versuchen, das von `lock` referenzierte `ReentrantLock`-Objekt zu sperren (siehe Quelltext der Methode `reserveDvd()`, Seite 146). Anschließend prüft der „Gewinner“, ob es sich um den Datensatz handelt, den er benötigt. Wahrscheinlich ist der reservierte Datensatz für die meisten Threads aber uninteressant. Jedesmal, wenn eine Reservierung aufgehoben wird, kommt es zu einer plötzlichen Häufung der Prozessorlast, wodurch die Leistung des Servers beeinträchtigt werden kann.

[151/152] Es ist offenbar sinnvoll, einen Thread nur dann zu benachrichtigen, wenn die Reservierung eines für ihn interessanten Datensatzes aufgehoben wird. Unter Version 1.4 des Java Development Kits war dieser Ansatz nur schwierig zu realisieren: Jeder Thread hätte bezüglich eines anderen Objektes synchronisiert werden müssen, um zu gewährleisten, daß stets nur ein bestimmter Thread benachrichtig wird. Version 5 des Java Development Kits gestattet dagegen, daß alle Threads zwar ein gemeinsames Objekt sperren, aber abhängig von individuellen Bedingungen benachrichtigt werden.

[153] Die reentranten („eintrittsinvarianten“, „wiedereintrittsfähigen“) Sperrobjekte aus Version 5 des Java Development Kits, mit ihrer von der Synchronisierung bezüglich eines Blocks von Anweisungen abweichenden Syntax, gestatten verschränkte Sperren (*hand-over-hand locking*), also verkettete Sperren, wobei eine Sperre erst aufgehoben wird, nachdem die nachfolgende Sperre aktiviert wurde.

[154] Das folgende Beispiel zeigt einen Ersatz für die `reserveDvd()`-Methode, der dem beschriebenen Konzept entspricht. Beachten Sie, daß die Zeilennummern nur für diese Diskussion gelten und kein Zusammenhang mit den Zeilennummern in der ursprünglichen `reserveDvd()`-Methode besteht. Eine weniger CPU-lastige `reserveDvd()`-Methode:

```
01. private static Map<String, LockInformation> reservations
02.         = new HashMap<String, LockInformation>();
03.
04. private static Lock masterLock = new ReentrantLock();
05.
06. public boolean reserveDvd(String upc, DvdDatabase renter)
07.         throws InterruptedException {
08.     LockInformation dvdLock = null;
09.     masterLock.lock();
10.     try {
11.         dvdLock = reservations.get(upc);
12.         if (dvdLock == null) {
13.             dvdLock = new LockInformation();
14.             reservations.put(upc, dvdLock);
15.         }
16.         dvdLock.lock();
17.     } finally {
18.         masterLock.unlock();
19.     }
20.
21.     try {
22.         long endTimeMSec = System.currentTimeMillis() + 5000;
23.         Condition dvdCondition = dvdLock.getCondition();
24.         while (dvdLock.isReserved()) {
25.             long timeLeftMSec = endTimeMSec - System.currentTimeMillis();
26.             if (!dvdCondition.await(timeLeftMSec, TimeUnit.MILLISECONDS)) {
27.                     return false;
28.             }
29.         }
30.         dvdLock.setReserver(renter);
```

```
31.     } finally {
32.         dvdLock.unlock();
33.     }
34.     return true;
35. }
```

[155] Zeile 9 sperrt das von `masterlock` referenzierte `ReentrantLock`-Objekt („Hauptsperrobjekt"), so daß wir entweder eine Referenz auf das `LockInformation`-Objekt („Sperrobjekt") dieses Datensatzes anfordern (Zeile 11) oder ein neues Sperrobjekt erzeugen und der Reservierungstabelle hinzufügen können (Zeilen 13 und 14). Ohne Hauptsperrobjekt könnte ein anderer Thread die Reservierungstabelle ändern, während wir eine Referenz auf das Sperrobjekt dieses Datensatzes anfordern.

[156] Bei Version 1.4 des Java Development Kits stünden wir nun vor dem folgenden Problem: Die Sperrung des Hauptsperrobjektes wird nicht mehr länger benötigt. Heben wir aber die Sperrung des Hauptsperrobjektes auf, bevor wir das von `dvdLock` referenzierte, für den aktuellen Datensatz spezifische Sperrobjekt gesperrt haben, so besteht die Gefahr, daß `dvdLock` von einem anderen Thread gesperrt (der Datensatz also eventuell geändert) wird, bevor unser Thread `dvdLock` sperren kann. Da der Sperrmechanismus in Version 1.4 des Java Development Kits auf der Synchronisierung von Blöcken beruht, ist es nicht möglich, das individuelle Sperrobjekt zu sperren und anschließend die Sperre des Hauptsperrobjektes aufzuheben.

[157] Dieses Problem läßt sich mittels verschränkter Sperren lösen. Bevor wir die Sperre des individuellen Sperrobjektes aufheben, sperren wir in Zeile 16 das Hauptsperrobjekt. Im Augenblick sind zwei Sperrobjekte gesperrt. Nach dem Aufheben der Sperre des Hauptsperrobjektes in Zeile 18 bleibt eine Sperre übrig. Stets ist mindestens ein Sperrobjekt gesperrt und die Existenz von zwei Sperren wurde umgehend auf das Minimum reduziert.

[158] In Zeile 26 wartet jeder Client auf die Bedingung für „sein" Sperrobjekt. Der Clients wird nicht mehr benachrichtigt, wenn *irgendeine* Sperre aufgehoben wird.

[159] Diese Funktionalität ist in der inneren Klasse `LockInformation` implementiert:

```
class LockInformation extends ReentrantLock {
    private DvdDatabase reserver = null;
    private Condition notifier = newCondition();

    void setReserver(DvdDatabase reserver) {
        this.reserver = reserver;
    }

    void releaseReserver() {
        this.reserver = null;
    }

    Condition getCondition() {
        return notifier;
    }

    boolean isReserved() {
        return reserver != null;
    }
}
```

[160] Die `releaseDvd()`-Methode verwendet die in der Klasse `LockInformation` implementierte Bedingung (von `notifier` referenziertes *`Condition`*-Objekt), um nur die Threads zu benachrichtigen, die auf diesen speziellen Datensatz warten. Warten viele Threads auf Datensätze, so kann dieser

Ansatz die Prozessorauslastung beim Entsperren eines Datensatzes signifikant reduzieren.

[161] Wir empfehlen wiederum, sorgfältig nachzulesen, welche Anforderung die Anleitung zu Ihrer Prüfungsaufgabe im Hinblick auf die gleichzeitige Benachrichtigung vieler Threads stellt. Insbesondere ist dies wiederum eine Design-Entscheidung, die Sie dokumentieren sollten.

Tipp: Wir haben diese Diskussion über viele Sperrobjekte in das Buch aufgenommen, da die von Sun Microsystems ausgegebenen Prüfungsaufgaben hin und wieder entsprechende Anforderungen stellen. Zum Zeitpunkt der Drucklegung dieses Buches scheint Sun Microsystems Prüfungsanwärter, die diesen Gesichtspunkt nicht beachten, aber nicht zu bestrafen.

5.5 Zusammenfassung

[162] In diesem Kapitel haben wir die Entwicklung von Klassen demonstriert, die die Datenbankdatei lesen und Datensätze sperren können. Wir haben einige typische Probleme diskutiert und erklärt, wie sie sich vermeiden lassen. Außerdem haben wir Beispiele für einige Entwurfsmuster besprochen und ihre Anwendung vorgeführt.

[163] Wir haben viele technische Handgriffe vorgestellt, die Sie zur Bearbeitung Ihrer Prüfungsaufgabe benötigen werden, etwa das Lesen aus und das Schreiben in Dateien bei beliebiger vorheriger Positionierung des Dateizeigers sowie das Sperren und Entsperren von Datensätzen. Wir haben einige neue Eigenschaften und Fähigkeiten von Version 5 des Java Development Kits verwendet und den Gebrauch diverser neuer Klassen und Interfaces erläutert.

[164] Wir haben bereits in Kapitel 3 erwähnt, daß das Beispielprojekt *Denny's DVDs* in mancher Hinsicht komplizierter ist, als die von Sun Microsystems gestellten Prüfungsaufgaben. Eventuell finden Sie für ein Problem in Ihrer Prüfungsaufgabe eine viel einfachere Lösung als unsere hier gezeigten Vorschläge (und wir empfehlen mit Nachdruck, einfachere Ansätze zu beachten). Ebenso können Teile Ihrer Prüfungsaufgabe komplizierter sein, als unser Beispielprojekt. Die Klasse, die das Interface implementiert, das Sie zusammen mit Ihrer Prüfungsaufgabe erhalten haben, kann auch Methoden benötigen, die wir nicht in diesem Buch nicht genannt haben. Wir sind aber davon überzeugt, daß wir Sie in diesem Buch mit allen grundlegenden Informationen ausgerüstet haben, die Sie brauchen, um die benötigten Methoden selbst zu schreiben.

5.6 Häufige Fragen

- *Frage*: Muß ich für meine Prüfungsaufgabe ein eigenes Format für die Datenbankdatei entwerfen?

  Antwort: Nein. Zum Zeitpunkt der Drucklegung dieses Buches liefert Sun Microsystems eine Datenbankdatei mit Beispieldaten sowie eine Beschreibung des Dateiformates, die Sie benötigen, um die Datenbankdatei lesen und schreiben zu können.

- *Frage*: Soll ich meine Testklassen und Build-Skripte zusammen mit meiner Lösung einreichen?

  Antwort: Wir empfehlen, über die Anforderungen hinaus nichts einzureichen. Zum Zeitpunkt der Drucklegung dieses Buches weist die von Sun Microsystems ausgegebene Anleitung zu den Prüfungsaufgaben darauf hin, daß Lösungsteile, die über die Anforderungen hinausgehen, bei der Bewertung keine zusätzlichen Punkte einbringen. Sie gewinnen also nichts, wenn Sie Ihren Testklassen und Build-Skripte mit einreichen.

- *Frage*: Soll ich die Klasse, die das vorgegebene Interface implementiert, stufenweise entwickeln?

  Antwort: Wir empfehlen nachdrücklich, daß Sie alle Klassen stufenweise entwickeln. Dadurch gewährleisten Sie, daß ein Abschnitt korrekt ist, bevor Sie mit dem nächsten Abschnitt beginnen.

- *Frage*: Ist es erforderlich beziehungsweise erlaubt, daß die Klasse, die das vorgegebene Interface implementiert, ein Value-Objekt verwendet?

  Antwort: Lesen Sie in Ihrer Anleitung nach, ob ein Value-Objekt erlaubt ist oder nicht. Bis jetzt waren die Anleitungen hinsichtlich der Signaturen der verlangten Methoden der Klasse, die das vorgegebene Interface implementiert, sehr deutlich. Falls Value-Objekte in dieser Klasse tatsächlich nicht erlaubt sein sollten, gestattet Ihre Anleitung eventuell, sie überall sonst in Ihrer Anwendung zu verwenden.

- *Frage*: Muß die Klasse, die das vorgegebene Interface implementiert, das *Façade*-Entwurfsmuster implementieren?

  Antwort: Es gibt kein bestimmtes Entwurfsmuster, daß Sie in Ihrer Anwendung implementieren müssen. Sie können jedes Entwurfsmuster wählen, das zur jeweiligen Situation paßt. Insbesondere wird *nicht gefordert*, daß Sie die Verwendung eines Entwurfsmusters dokumentieren, obwohl es als guter Programmierstil gilt, wenn Sie ein verwendetes Entwurfsmuster in der Dokumentation Ihrer Design-Entscheidungen und/oder in Ihrem Dokumentationskommentaren erwähnen, da Sie hierdurch anderen Programmieren das Verständnis Ihres Quelltextes erleichtern.

- *Frage*: Soll ich in der Lösung meiner Prüfungsaufgabe einen Zwischenspeicher verwenden?

  Antwort: In der Regel weisen die Anleitungen von Sun Microsystems darauf hin, daß ein klarer Entwurf im Vergleich mit einem performanteren Entwurf bevorzugt wird. Entscheiden Sie selbst, wie sich das Anlegen eines Zwischenspeichers für die Datensätze auf die Lesbarkeit Ihres Quelltextes auswirkt und ob sich die Mühe lohnt.

- *Frage*: Was soll ich tun, wenn die Verwendung eines Zwischenspeichers bewirkt, daß der Laufzeitumgebung beziehungsweise dem Rechner der Arbeitsspeicher ausgeht?

  Antwort: Laden Sie Datensätze nur bei Bedarf (*lazy loading*) und leiten Sie Ihre Datensatzklasse von `SoftReference` ab, damit die Laufzeitumgebung Datensätze löschen kann, wenn der Arbeitsspeicher knapp wird. Wenn die Laufzeitumgebung einen Datensatz aus dem Arbeitspeicher löscht, bedeutet „Laden bei Bedarf“, daß der Datensatz erforderlichenfalls erneut geladen (und dafür eventuell ein anderes seltener gebrauchtes Objekt gelöscht) wird. Überschlagen Sie aber auch, wieviele Datensätze Sie benötigen, um den Arbeitsspeicher Ihres Rechners zu verbrauchen und überlegen Sie sich, ob Sie derart viele Datensätze überhaupt mit Ihrer Anwendung verarbeiten möchten.

- *Frage*: Darf ich die Packages `java.nio` (NIO) und `java.util.concurrent` in meiner Anwendung verwenden?

  Antwort: In der Vergangenheit wies Sun Microsystems auf ihrer Website darauf hin, daß die `java.nio`-Packages in den Prüfungsaufgaben für die Zertifizierung zum *Sun Certified Java Developer nicht* verwendet werden dürfen. Andererseits weist Sun Microsystems darauf hin, daß die Anleitung zu Ihrer Prüfungsaufgabe maßgeblich ist. Sagt Ihre offizielle Anleitung aus, daß Sie ein bestimmtes Package nicht verwenden dürfen, so ist sein Gebrauch verboten. Verbietet Ihre Anleitung ein bestimmtes Package dagegen nicht, können Sie es bedenkenlos verwenden.

Kapitel 6

RMI als Netzwerkschnittstelle

[0] In diesem Kapitel entwickeln wir das erforderliche Hintergrundwissen, um eine vollständige Netzwerkschnittstelle mittels Remote Method Invocation (RMI) zu implementieren, nämlich RMI selbst und in etwas geringerem Umfang Serialisierung. Abbildung 6-1, Seite 163 (Buch), veranschaulicht, wie sich die Netzwerkschnittstelle zwischen der Datenbankdatei („Datenbankschicht“) und der graphischen Benutzeroberfläche in die Architektur der Beispielanwendung einfügt.

[1] Das Verstehen von RMI setzt voraus, mit den grundlegenden Eigenschaften und Fähigkeiten der Serialisierung vertraut zu sein, da RMI auf der Serialisierung aufbaut. Vermutlich erinnern Sie sich noch aus Ihrer Prüfung zum *Sun Certified Java Programmer* an das Thema „Serialisierung“, wir fassen die Grundzüge aber noch einmal kurz zusammen. Wir behandeln in Kapitel 6 die folgenden Themen:

- Serialisierung in der Beispielanwendung *Denny's DVDs*.
- Vor- und Nachteile der Wahl von RMI als Netzwerkschnittstelle.
- Implementierung einer entfernten Klasse und Definition eines entfernten Interfaces.
- Diskussion des *Factory*-Entwurfsmusters und seiner Verwendung im RMI-Kontext.
- Marshalling und Demarshalling bei RMI.
- RMI-Registratur und registrieren entfernter Objekte.

Warnung: Java-Sockets, also die Netzwerkschnittstelle, die Sie als Prüfungskandidat alternativ zu RMI wählen können, sind Gegenstand von Kapitel 7 (nächstes Kapitel). Beachten Sie bei Ihrer Entscheidung, daß Sie in Ihrer Prüfungsaufgabe serialisierte Objekte übertragen *müssen*, wenn Sie Sockets als Netzwerkschnittstelle wählen. Während RMI auf der Serialisierung aufbaut, ist sie bei Sockets nicht zwingend erforderlich. Sie könnten bei Sockets beispielsweise ein eigenes Übertragungsformat festlegen und von einem beliebigen Client aus Anfragen an Ihren Socketserver senden. Wir behandeln die Übertragung serialisierter Objekte über eine Socketverbindung im nächsten Kapitel. Auch wenn Sie sich für Sockets als Netzwerkschnittstelle entscheiden, ist die Einführung in das Thema „Serialisierung“ im folgenden Abschnitt für Sie nützlich.

6.1 Serialisierung

[2] Was ist Serialisierung und welche Aufgabe hat sie in einer Client/Server-Anwendung? Es seien A und B zwei Rechner im selben Netzwerk und A sende eine Nachricht an B. Wie empfängt B diese Nachricht und welche Schritte sind erforderlich, damit B die Nachricht auswerten kann? Die Nachrichtenübertragung zwischen A und B setzt ein Protokoll voraus, das die zu übertragenden Daten verflacht, das heißt, in ein Format umwandelt, das die Versendung über ein Netzwerk hinweg gestattet.

[3] Die Serialisierung ist ein Teil der Datenübertragung über das Netzwerk. Durch den Vorgang der Serialisierung wird ein Objekt auf der sendenden Seite in die Angabe seines Typs und seinen Zustand „zerlegt". Auf der empfangenden Seite wird aus diesen Informationen eine Kopie des ursprünglichen Objektes rekonstruiert. Die Übertragung eines serialisierten Objektes wird häufig als Marshalling bezeichnet, die Rekonstruktion des Objektes auf der empfangenden Seite als Demarshalling oder auch Unmarshalling. Serialisierte Objekte können mit Hilfe der I/O-Bibliothek von Java dauerhaft im lokalen Dateisystem gespeichert oder per Socket über das Netzwerk versendet werden. In beiden Fällen wird das Objekt in einen seriellen Bytestrom umgewandelt, über das Netzwerk gesendet oder im Dateisystem gespeichert, auf der empfangenden Seite in den Arbeitspeicher geladen und sein Abhängigkeitsgraph rekonstruiert. Die beiden vorigen Sätze drücken im Java-Jargon aus, daß auf der Empfängerseite ein mit dem ursprünglichen Objekt identisches Objekt erzeugt wird. Ein aus einem Bytestrom in den Arbeitsspeicher geladenes Objekt befindet sich im *aktiven Zustand*, ein aus dem Arbeitspeicher der Laufzeitumgebung in eine Datei geschriebenes Objekt dagegen im *passiven Zustand*. Die Sprechweise eines Objektes im aktiven beziehungsweise passiven Zustand wird sich als nützlich erweisen, wenn wir später in diesem Kapitel die abstrakte Klasse `java.rmi.activation.Activatable` kennenlernen (Seite 166). Wir besprechen die Serialisierung nun im Detail.

[4] Die Serialisierung besteht aus drei Schritten: Erstens wird das zu serialisierende Objekt in einen seriellen Bytestrom umgewandelt, dessen Header den Typ des Objektes angibt, worauf der Zustand des Objektes folgt. Der Zustand eines Objektes ist durch seine Feldinhalte definiert, mit Ausnahme der statischen sowie der transienten Felder (Standardserialisierung vorausgesetzt, das heißt die Methoden `readObject()` und `writeObject()` wurden nicht implementiert; siehe unten). Zweitens fragt das empfangende entfernte Objekt den Typ des serialisierten Objektes ab und erzeugt ein entsprechendes neues Objekt. Falls die Laufzeitumgebung (entfernt oder lokal) die Klasse nicht finden kann, wirft sie eine Ausnahme vom Typ `java.lang.ClassNotFoundException` aus. Drittens werden, nachdem das Objekt erzeugt wurde, die Feldinhalte aus dem Bytestrom abgefragt und die entsprechenden Felder des neu erzeugten Objektes initialisiert.

Bemerkung: Ein serialisiertes Objekt ist eine Kopie des ursprünglichen Objektes, nicht aber eine Referenz darauf. Bei RMI werden Parameter und Rückgabewerte als Werte (Kopien) übergeben. Nur entfernte Objekte, also Objekte, die mit Hilfe der `UnicastRemoteObject`-Methode `export()` exportiert wurden oder deren Klasse von `java.rmi.server.UnicastRemoteObject` abgeleitet ist, können über die Grenze ihrer Laufzeitumgebung hinweg referenziert werden.

6.1.1 Das Hilfsprogramm serialver

[5] Die Serialisierung eines Objektes setzt voraus, daß seine Klasse das Interface *`Serializable`* implementiert. Dieses Interface fällt dadurch auf, daß es keinerlei Methoden deklariert. Eine Klasse implementiert *`Serializable`* hauptsächlich, um der Laufzeitumgebung anzuzeigen, daß ihre Objekte serialisiert werden können. Interfaces, die wie *`Serializable`* keine Methoden deklarieren, werden

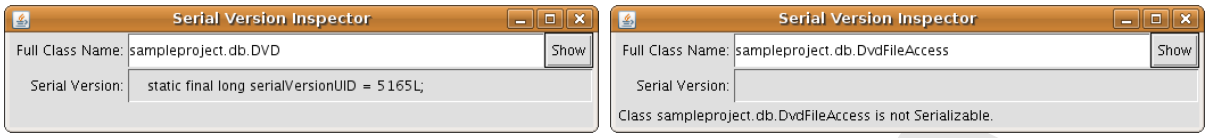

Abbildung 6.1: Ausgabe des Hilfsprogramms `serialver` für die Klassen `DVD` (links, serialisierbar) und `DvdFileAccess` (rechts, nicht serialisierbar) der Beispielanwendung.

häufig als Marker-Interfaces bezeichnet. Ein Marker-Interface deklariert formal keine Methoden, die von implementierenden Klassen oder ihren Unterklassen ausprogrammiert werden müssen, sondern kennzeichnet das Objekt als einem bestimmten Typ zugehörig. Sie können mit Hilfe des Hilfsprogramms `serialver` herausfinden, ob eine Klasse serialisierbar ist. Wechseln Sie zum Beispiel in das Wurzelverzeichnis des Beispielprojektes *monkhouse-camerlengo/src* und führen Sie das folgende Kommando aus:

```
serialver -classpath classes/. -show
```

[6] Geben Sie nun den vollqualifizierten Namen einer Klasse aus der Beispielanwendung ein, zum Beispiel `sampleproject.db.DVD` (serialisierbar) oder `sampleproject.db.DvdFileAccess` (nicht serialisierbar). Abbildung 6.1 zeigt die Ausgabe von `serialver` für `DVD` (links) und `DvdFileAccess` (rechts).

[7/8] Die direkte Prüfung der Klassen der Beispielanwendung auf Serialisierbarkeit ist möglich, da wir `serialver` im Wurzelverzeichnis des Projektes aufgerufen haben (das im Klassenpfad liegt). Das Kommando läßt sich aber auch in einem anderen Verzeichnis aufrufen, wenn das Wurzelverzeichnis der Beispielanwendung in den Klassenpfad eingetragen wird. Das Prüfen unserer eigenen Klassen auf Serialisierbarkeit ist nicht besonders interessant, da wir den Quelltext zur Verfügung haben und wissen, welche Klassen serialisierbar sind und welche nicht (das heißt wir können einfach nachschauen, ob eine Klasse das Interface *`Serializable`* implementiert). Das Hilfsprogramm `serialver` ist dagegen bei Klassen praktisch, deren Quelltext nicht zugänglich ist, zum Beispiel bei Klassen in *.jar* Dateien, die zur Laufzeitumgebung gehören. Untersuchen Sie einmal ein paar Klassen aus Ihrem Klassenpfad, zum Beispiel `java.util.Date` oder `java.lang.String` aus Ihrem JDK.

[9] Wenn Sie im Quelltext der beiden Klassen `DVD` und `DvdFileAccess` nachsehen, werden Sie feststellen, daß `DVD` tatsächlich das Interface *`Serializable`* implementiert, `DvdFileAccess` dagegen nicht. (Sie können den Quelltext der Beispielanwendung aus dem „Source Code"-Abschnitt der Apress-Website (*http://www.apress.com/book/sourcecode*) herunterladen.)

6.1.2 Der Serialisierungsvorgang

[10] Zwei Klassen aus dem Package `java.io` verrichten beim Schreiben und Einlesen serialisierter Objekte den größten Teil der Arbeit: `ObjectOutputStream` und `ObjectInputStream`, genauer die `ObjectOutputStream`-Methode `writeObject()` und die `ObjectInputStream`-Methode `readObject()`. In der Beispielanwendung dient die Serialisierung zum Austauschen von Methodenargumenten zwischen den graphischen Clients und dem Server über das Netzwerk hinweg, wobei wir die Standardserialisierung verwenden (falls der Server ein RMI- oder Socketserver ist). Bei RMI ist keine explizite Serialisierung der Methodenargumente erforderlich. Angenommen (beachten Sie die Warnung auf Seite 158 unter den beiden folgenden Beispielen), wir wollten den Zustand unserer `DVD`-Objekte persistent im Dateisystem speichern und würden diese Funktionalität in den beiden *hypothetischen* `DvdFileAccess`-Methoden `persistDvd()` und `retrieveDvd()` implementieren. Der Client würde, um beispielsweise nach einem Aufruf der Methode `setRecordNumber()` einen Datensatz oder den Zustand eines `DVD`-Objektes zu sichern, die Methode `persistDvd()` auf rufen. Die

`persistDvd()`-Methode serialisiert das `DVD`-Objekt mit Hilfe eines `ObjectOutputStream`-Objektes und dessen `writeObject()`-Methode (siehe unten). Beachten Sie, daß das serialisierte Objekt der Klasse `DVD` angehört. Die `readObject()`-Methode liest das serialisierte Objekt ein und erzeugt ein neues `DVD`-Objekt mit dem Zustand des ursprünglichen Objektes. Die *hypothetische Methode* ***`persistDvd()`*** serialisiert ein `DVD`-Objektes per `FileOutputStream`:

```
private boolean persistDvd(DVD dvd) throws IOException {
    boolean retVal = false;
    // open a FileOutputStream associated with the data
    // notice that if the DVD does not already exist,
    // it will be created.
    String filePath = dbName + "/" + dvd.getUPC() + fileExtension();
    FileOutputStream fos = new FileOutputStream(filePath);
    try {
        ObjectOutputStream oos = new ObjectOutputStream();
        // Read in the data from the object
        oos.writeObject(dvd);
    } finally {
        // close all references
        oos.close();
        fos.close();
    }
}
```

Die *hypothetische Methode* ***`retrieveDvd()`*** deserialisiert ein `DVD`-Objektes per `FileInputStream`:

```
private DVD retrieveDvd(String upc, String fileExtension)
    throws IOException, ClassNotFoundException {
    DVD retVal = null;
    // get the path to the object's serialized state.
    try {
        String filePath = dbName + "/" + fileExtension;
        FileInputStream fis = new FileInputStream(filePath);
        // Read in the data from the object
        ObjectInputStream ois = new ObjectInputStream(fis);
        retVal = (DVD) ois.readObject();
    } finally {
        ois.close();
        fis.close();
    }
    return retVal;
}
```

Warnung: Die obigen *hypothetischen* Methoden `persistDvd()` und `retrieveDvd()` gehören *nicht* zum tatsächlichen Quelltext der Beispielanwendung. Die Beispielanwendung macht lediglich von der Standardserialisierung Gebrauch. `persistDvd()` und `retrieveDvd()` erfüllen einen rein pädagogischen Zweck, nämlich vorzuführen, wie Objekte explizit serialisiert werden.

[11] Bei der Serialisierung werden Informationen über den Zustand des Objektes über einen Bytestrom persistent in einer Datei gespeichert. Der Objektzustand ist aber nicht die einzige Information, die gesichert wird. Damit der Klassenlader ein serialisiertes und in einer Datei gesichertes Objekt korrekt rekonstruieren kann, werden auch der Typ des Objektes und eine Versionsbezeichnung gespeichert. Besitzt eine serialisierbare Klasse kein `serialVersionUID`-Feld, so generiert das `ObjectOutputStream`-Objekt eine eindeutige Versionsnummer. Bei jeder anschließenden Änderung an den Komponenten dieser Klasse wird beim Übersetzen eine neue Versionsnummer erzeugt. Die

Versionsnummer der Klasse `DVD` ist als privates Feld deklariert:

```
private static final long serialVersionUID = 5165L;
```

Bemerkung: In der Regel beinträchtigt jede Änderung einer serialisierbaren Klasse die Kompatibilität der serialisierten Objekte. Hier ist die Verwendung einer selbstdefinierten Versionsnummerierung per `serialVersionUID`-Feld sinnvoll. Sie können per `serialver` einen Anfangswert generieren und in Ihre Klasse einsetzen. Das `serialVersionUID`-Feld gestattet Ihnen, ein eigenes Versionsschema für Ihre serialisierbaren Klassen festzulegen. Interessanterweise „weiß" die Laufzeitumgebung, daß dieses Feld existiert, obwohl `serialVersionUID` als privat deklariert ist.

6.1.2.1 Anpassung der Standardserialisierung

[12] Hin und wieder gibt es Situationen, in denen Sie sich mehr Kontrolle über die Serialisierung des Zustandes eines Objektes wünschen. Eventuell möchten Sie nur einen Teil des Objektzustandes persistent sichern, beispielsweise wenn ein privates Feld eine sitzungsabhängige Datenbankverbindung referenziert, die nicht gesichert zu werden braucht und bei Bedarf angelegt wird. Es ist unsinnig, die Datenbankverbindung für die nächste Sitzung mit der Anwendung persistent zu speichern. Statt dessen weisen Sie dem Feld beim nächsten Programmstart eine neue Datenbankverbindung zu. Eine andere Situation, in der Sie einen Teil des Zustandes eines Objektes nicht persistent speichern können, sind Felder, die Objekte eines nicht serialisierbaren Typs referenzieren. Beispielsweise referenziert die Klasse `DVD` aus unserer Beispielanwendung ein Objekt der nicht serialisierbaren Klasse `Logger`.

[13] Woran erkennt die Laufzeitumgebung, welche Felder (genauer, die von diesen Feldern referenzierten Objekte) beim persistenten Speichern eines Objektes einer Klasse serialisiert beziehungsweise nicht serialisiert werden sollen? Java stellt zu diesem Zweck das Schlüsselwort `transient` zur Verfügung. Von transienten Felder referenzierte Objekte werden bei der Serialisierung nicht beachtet.

[14] Wir verwenden das Schlüsselwort `transient` in unserer Beispielanwendung, da `Logger`-Objekte nicht serialisierbar sind (Rufen Sie das Hilfsprogramm `serialver` für die Klasse `java.util.logging.Logger` auf). Die Nicht-Serialisierbarkeit der Klasse `Logger` paßt ins Bild, da ein `Logger`-Objekt kein wesentlicher Bestandteil eines `DVD`-Objektes ist. In der Regel gehören `Logger`-Objekte zu den Dingen, die nicht dauerhaft gespeichert und beim Deserialisieren eines `DVD`-Objektes einfach neu erzeugt werden. Die folgende Zeile zeigt die Deklaration des transienten `log`-Feldes aus *DVD.java*:

```
private transient Logger log = Logger.getLogger("sampleproject.db");
```

[15] Es gibt noch einen weiteren Serialisierungsmechanismus, der in diesem Zusammenhang erwähnt werden sollte. Anstelle des Interfaces *`Serializable`* können Sie das von *`Serializable`* abgeleitete Interface *`java.io.Externalizable`* implementieren. Der große Vorteil von *`Externalizable`*, verglichen mit *`Serializable`*, besteht in der Performanz. Marshalling und Demarshalling verwenden bei der Serialisierung systematisch den Reflexionsmechanismus, um die nicht-transienten und nicht-statischen Felder einer Klasse zu ermitteln.

[16] Durch die dynamische Bestimmung der Klasseneigenschaften zur Laufzeit per Reflexion kann die Serialisierung sowohl im Hinblick auf die Prozessorauslastung als auch auf den Speicherbedarf teuer sein. Ist Performanz wichtiger als Flexibilität, so sollten Sie den reflexionslosen *`Externalizable`*-Ansatz als Alternative zu *`Serializable`* in Betracht ziehen. Weitere Informationen über den Reflexionsmechanismus von Java finden Sie im folgenden Sun Microsystems-Tutorial: *http://java.sun.com/docs/books/tutorial/reflect/*.

6.1.3 Performante Serialisierung mit dem Interface Externalizable

[17] Ist die Performanz ausschlaggebend, so sollten Sie das Interface *Externalizable* statt *Serializable* zur Serialisierung verwenden. Auf welche Weise verbessert *Externalizable* den Standardserialisierungsmechanismus? Die Implementierung des Interfaces *Externalizable* verlangt, daß *Sie* die Einzelheiten über den Zustand des Objektes in den Bytestrom schreiben beziehungsweise von dort lesen. Diese Vorgehensweise ist zwar langwieriger, als sich auf den Reflexionsmechanismus zu verlassen, bewirkt aber letzten Endes eine Geschwindigkeitsexplosion. Die Klasse `ObjectOutputStream` vereinfacht den Serialisierungsvorgang nicht mehr. Statt dessen müssen Sie die *Externalizable*-Methoden `readExternal()` und `writeExternal()` implementieren und dabei den Typ des jeweiligen Feldes berücksichtigen, also ob es sich um einen primitiven Typ, ein `String`-Feld oder einen anderen serialisierbaren Typ handelt. Da Sie eine systemnahe Operation implementieren, müssen Sie die Felder in der Reihenfolge aus dem Bytestrom lesen, in der Sie sie zuvor geschrieben haben.

[18] Der folgenden Quelltextauszug zeigt einen Ausschnitt aus der Klasse `DVD` für den *hypothetischen Fall*, daß `DVD` das Interface *Externalizable* implementiert (beachten Sie die Warnung auf Seite 161). Wir zeigen nur die Methoden `readExternal()` und `writeExternal()`, um Platz zu sparen. Beide Methoden können als private Methoden deklariert werden, da der Serialisierungsmechanismus die üblicherweise gelten Zugriffsregeln für Klassen umgeht (das heißt die Laufzeitumgebung kann die privaten Serialisierungsmethoden eines Objektes aufrufen).

Bemerkung: Implementiert eine serialisierbare Klasse die beiden Methoden `readObject()` und `writeObject()`, so wird der Reflexionsmechanismus nicht aufgerufen und Sie können, verglichen mit der Standardserialisierung (das heißt die Klasse implementiert *Serializable*, ohne die Methoden `readObject()` und `writeObject()` auszuprogrammieren) einen Performanzzuwachs feststellen.

Die Implementierung der *Serializable*-Methoden `readObject()` und `writeObject()` ähnelt der Implementierung von *Externalizable* mit einigen kleineren Unterschieden hinsichtlich der Vererbung und der Behandlung von Klassenmetadaten. Sie können die Signaturen der beiden zu implementieren *Externalizable*-Methoden dem folgenden Quelltextauszug entnehmen. Wenn Sie diese Methoden überschreiben, ist es wichtig, die Klassenattribute in derselben Reihenfolge in den Bytestrom zu schreiben, in der sie daraus wieder gelesen werden. Falls die serialisierbare Klasse keiner Vererbungshierarchie angehört, werden `readObject()` und `writeObject()` wie `readExternal()` und `writeExternal()` implementiert (siehe unten). Die Signaturen der Methoden `readObject()` und `writeObject()` lauten:

```
private void writeObject(java.io.ObjectOutputStream out) throws IOException
private Object readObject(java.io.ObjectInputStream in) throws IOException,
    ClassNotFoundException
```

Eine hypothetische *Externalizable*-Version der Klasse `DVD` (nur die Methoden `writeExternal()` und `readExternal()`):

```
/**
  * Required method for Externalizable interface.
  * Specifies how the object graph gets converted to a byte stream.
  */
public void writeExternal(ObjectOutput out) throws IOException {
    out.writeUTF(upc);
    out.writeUTF(name);
    out.writeUTF(composer);
```

```
        out.writeUTF(director);
        out.writeUTF(leadActor);
        out.writeUTF(supportingActor);
        out.writeUTF(year);
        out.writeUTF(copy);
    }

    /**
      * Required method for Externalizable interface.
      * Specifies how to recreate the object graph from
      * a byte stream. The order members are read must
      * match the order in which the object members were
      * written to the stream.
      */
    public void readExternal(ObjectInput in)
            throws IOException, ClassNotFoundException {
        in.readUTF(upc);
        in.readUTF(name);
        in.readUTF(composer);
        in.readUTF(director);
        in.readUTF(leadActor);
        in.readUTF(supportingActor);
        in.readUTF(year);
        in.readUTF(copy);
    }
```

Warnung: Wie die hypothetischen Methoden `persistDvd()` und `retrieveDvd()` in Unterabschnitt 6.1.2 (siehe Warnung auf Seite 158) befinden sich auch die obigen Methoden `readExternal()` und `writeExternal()` *nicht* im Quelltext der Beispielanwendung, sondern erfüllen einen rein pädagogischen Zweck, nämlich das Vorführen einer Implementierung des Interfaces *`Externalizable`*.

[19] Das Interface *`Externalizable`* wird als fortgeschrittenes Serialisierungsverfahren betrachtet und daher in diesem Buch nicht weiter behandelt. Wir haben uns der Einfachheit halber für *`Serializable`* entschieden, *`Externalizable`* aber vorgestellt, damit Sie die mögliche Alternative verstehen. *`Serializable`* ist leichter zu implementieren, opfert dafür aber Performanz und Flexibilität. Die Implementierung von *`Externalizable`* führt zu schwer pflegbarem Quelltext. Angenommen, wir würden in der `DVD`-Klasse ein neues Feld anlegen, zum Beispiel von einem serialisierbaren Typ wie `boolean`, so müßten wir unsere *`Serializable`*-Methoden `readObject()` und `writeObject()` nicht ändern. Implementierte `DVD` aber das Interface *`Externalizable`*, so müßten wir die Methoden `readExternal()` und `writeExternal()` anpassen, damit das neue Feld berücksichtigt wird. Außerdem sollten die *`Externalizable`*-Methoden die Version ihrer Klasse prüfen, um feststellen zu können, wenn sie mit einer veralteten Version arbeiten. Das ist bei einem Produktivsystem ein Ärgernis, das viele Klassen enthält, die sich von Zeit zu Zeit ändern. Tabelle 6.1 vergleicht die Serialisierungsverfahren *`Serializable`* und *`Externalizable`*.

Tipp: Die Verwendung des Schlüsselwortes `transient` in einer Klasse die *`Externalizable`* implementiert, ist nicht erforderlich. Das Interface *`Externalizable`* verlangt, daß Sie die Einzelheiten des Schreibens und Einlesens des Objektzustandes definieren, so daß `transient` nicht benötigt wird. Das Schlüsselwort `transient` wird in *`Externalizable`*-Klassen ignoriert.

Bemerkung: Da hinsichtlich der Performanz der Beispielanwendung keine Anforderungen beste-

Serialisierung (Serializable)	Externalisierung (Externalizable)	Vorteil liegt bei
Einfachere Verwendung. Es genügt, `Serializable` zu implementieren und (gegebenenfalls) die Methoden `writeObject()` und `readObject()` auszuprogrammieren, um den Zustand des Objektes speichern beziehungsweise wieder einlesen zu können.	Kompliziertere Verwendung. Die im Interface `Externalizable` deklarierten Methoden `readExternal()` und `writeExternal()` *müssen* implementiert werden.	`Serializable`.
Weniger effizienter Algorithmus. Verwendet Reflexion, um ein Objekt zu untersuchen.	Leistungsfähigerer Algorithmus. Der Reflexionmechanismus wird nicht benötigt, da Sie das Schreiben und Einlesen der Informationen über den Zustand des Objektes selbst implementieren.	`Externalizable`.
More data gets serialized due to larger class description and versioning information.	Mehr Kontrolle über den Umfang der Serialisierung.	`Externalizable`.

Tabelle 6.1: Vergleich der Serialisierungsverfahren „Serialisierung“ und „Externalisierung“.

hen, haben wir uns bei Version 2.0 von *Denny's DVDs* für Serialisierung und gegen Externalisierung entschieden.

6.2 Remote Method Invocation (RMI)

[20] Eine wesentliche Anforderung in der Prüfung zum *Sun Certified Java Developer* besteht darin, daß die zu entwickelnde Anwendung Rechnern gestatten muß, über ein Netzwerk hinweg Nachrichten auszutauschen. Derartige Kommunikation, auch als „Verteilung von Anwendungen“ (*distributed computing*) bezeichnet, kann eine schwierige Herausforderung sein, wobei Ihnen RMI aber hilfreich zur Seite steht. RMI ist aber nicht der einzige Lösungsansatz. Alternative Verfahren sind der Remote Procedure Call (RPC), die Common Object Request Broker Architecture (CORBA) und das .NET Framework von Microsoft. Im Grunde ist RMI eine objektorientierte Version von RPC.

Bemerkung: Das .NET-Framework beinhaltet eine ähnliche, mit RMI konkurrierende Technologie namens .NET Remoting.

[21] Wir beginnen damit, einige neue Begriffe einzuführen. Der Begriff „Server“ bezeichnet im RMI-Kontext ein entferntes Objekt, dessen Methoden über eine *andere* Laufzeitumgebung aufgerufen werden können, als diejenige, in der sich das entfernte Objekt befindet. Ein „Client“ ist ein Objekt, das die entfernten Methoden eines solchen Servers aufruft. Ein „System verteilter Objekte“ besteht aus entfernten Objekten, deren Methoden von Clients aufgerufen werden können. Die Kommunikation zwischen Client und Server ist bidirektional, das heißt ein Client muß in der Lage sein, entfernte Objekte zu lokalisieren, Methoden [mit Argumenten] aufzurufen und deren Rückgabewerte zu empfangen.

[22] RMI befähigt Java-Objekte auf verschiedenen Rechnern miteinander zu kommunizieren. Eines der Ziele von RMI besteht darin, daß ein Entwickler mit einem entfernten Objekt arbeiten kann, als wäre es ein lokales Objekt. Der tatsächliche Ort des entfernten Objektes ist für den Entwickler transparent. Diese Netzwerktransparenz ist eine äußerst reizvolle Eigenschaft, da sie die komplexen Vorgänge in einem System verteilter Objekte abstrahiert, so daß sich entfernte Objekte wie lokale Objekte verhalten. In einem RMI-System läuft der Client lokal, während das restliche System auf

einem entfernten Rechner läuft. Derartige Systeme werden häufig als Systeme verteilter Objekte bezeichnet. Abbildung 6-5 veranschaulicht diesen Systemtyp.

Bemerkung: Der Remote Procedure Call (RPC) ist eine Alternative zu Sockets und RMI. RPC ist sowohl sprach- als auch prozessorunabhängig und setzt voraus, daß Parameter network representations einfacher Datentypen wie `int` oder `char` sind, die es auf nahezu jeder Plattform gibt. RMI ist dagegen nur prozessorunabhängig, setzt ein objektorientiertes Framework voraus und verlangt den Einsatz von Java als Programmiersprache.

Neben RPC gewährleisten auch die Common Object Request Broker Architecture (CORBA) und das Simple Object Access Protocol (SOAP) Sprach- und Prozessorunabhängigkeit zwischen verschiedenen Plattformen. Da RMI Java voraussetzt, sind viele ermüdende Schritte auf Protokollebene bereits in RMI eingebaut, so daß sich der Entwickler auf die Logik seiner Anwendung konzentrieren kann. RMI eignet sich hervorragend für Java-Systeme, da hier auf beiden Seiten der Netzwerkverbindung eine Java-Laufzeitumgebung vorausgesetzt werden kann. Daher ist RMI bei komplett in Java geschriebenen Systemen mit verteilten Objekten, verglichen mit RPC und CORBA, die bessere Wahl. Dennoch eine schändliche Randnotiz: CORBA-Systeme sind tendentiell performanter als RMI-Systeme, hauptsächlich deshalb, weil sie in C oder C++ geschrieben werden können. Das Remote Method Invocation Run Over Internet Inter-ORB Protocol (RMI-IIOP) ist eng mit RMI verwandt und wurde von Sun Microsystems und IBM gemeinsam entwickelt, um die Zusammenarbeit CORBA-kompatibler Anwendungen mit Java zu ermöglichen. Die Bordmittel von Java gestatten Ihnen, eine Beschreibungsdatei im Format der Interface Definition Language (IDL) zu generieren, um die Zusammenarbeit mit CORBA-Anwendungen zu ermöglichen, die beispielsweise in C++ geschrieben sind. Der `rmic`-Compiler generiert mit dem Schalter `-iiop` IDL-Dateien.

6.2.1 Ebenenmodell und Übertragungsprotokolle

[23] Ein Transportprotokokoll (*transport protocol*) baut direkt auf der Vermittlungsebene auf. Die beiden häufigsten Beispiele für Transportprotokolle sind TCP/IP und UDP. Über der Transportebene liegt die Übertragungsebene (*transfer layer*), die für die transparente Datenübertragung zwischen Geräten, [↑!] ... (*flow control*) und durchgängige Fehlerbehebung (*end-to-end error recovery*) verantwortlich ist. Die Transportebene fordert Dienste aus der Vermittlungsebene an. Die Abfolge der Protokollebenen lautet stark vereinfacht:

- Anwendungsebene (*application layer*): HTTP, SSH, Telnet.
- Sitzungsebene (*session layer*): RPC, NetBIOS, SSH.
- Transportebene (*transport layer*): TCP, UDP.
- Vermittlungsebene (*network layer*): IP.
- Physikalische Ebene (*physical layer*): Kabel.

RMI baut auf der Vermittlungsebene auf.

[24] Übertragungsprotokolle (*transfer protocol*) bauen auf Transportprotokollen auf und legen fest, wie Informationen zwischen Geräten übertragen werden. Das dem World-Wide-Web zugrunde liegende Hypertext Transfer Protocol (HTTP) ist ein häufig genanntes Beispiel für ein Übertragungsprotokoll. RMI verwendet nicht HTTP, sondern das Java Remote Method Protocol (JRMP) als Übertragungsprotokoll. Eine Schwäche von JRMP besteht darin, daß sowohl die Sender- als auch die Empfängerseite Java „verstehen“ müssen, wodurch die Lösung an Java und RMI gebunden ist. Die Abhängigkeit von Java wird dadurch verringert und die Verwendung von CORBA-kompatiblen

Protokoll	Beschreibung
RMI-JRMP	Standardübertragungsprotokoll für RMI. Setzt bei einer Client-Server-Anwendung auf beiden Seite Java als Programmiersprache voraus. Eine reine Java-Lösung.
Java-IDL	Für CORBA-Entwickler, die Java zusammen mit Schnittstellen verwenden wollen, die in einer CORBA-kompatiblen Anwendung definiert sind. Im wesentlichen zur Anbindung von CORBA an Java gedacht.
Java RMI-IIOP	Für Java-Entwickler, die die Kompatibilität mit älteren Anwendungen pflegen müssen. Zu Anbindung von Java an Nicht-Java-Systeme gedacht.

Tabelle 6.2: RMI Übertragungsprotokolle.

Clients ermöglicht, daß der `rmic`-Compiler bei Erzeugen der Stubobjekte mit dem Schalter `-iiop` aufgerufen werden kann. IIOP gestattet CORBA-kompatiblen Clients, entfernte Java-Objekte zu benutzen, so daß die Abhängigkeit von einer reinen Java-Lösung reduziert wird.

[25] Tabelle 6.2 faßt die drei Möglichkeiten zusammen, aus denen der Entwickler einer RMI-Anwendung ein Übertragungsprotokoll auswählen kann.

[26] Argumente und Rückgabewerte von Methoden sind Kopien. Exportierte Objekte oder Objekte, die ein entferntes Interface implementieren, werden über ihr Stubobjekt (clientseitiges Stellvertreterobjekt) referenziert. Um ein Objekt zu exportieren, leiten Sie eine Klasse von der abstrakten Klasse `java.rmi.server.UnicastRemoteObject` ab oder rufen Sie die `UnicastRemoteObject`-Methode `export(<entferntes Objekt>)` auf.

6.2.2 Vor- und Nachteile von RMI als Netzwerkschnittstelle

[27] Sie können bei der Netzwerkschnittstelle Ihrer Prüfungsaufgabe zwischen RMI und der Übertragung serialisierter Objekte über Sockets wählen. Es folgen einige Gründe, warum Sie RMI Sockets vorziehen könnten:

- Objektbasierte Semantik: Entfernte Objekte wirken und verhalten sich wie lokale Objekte. RMI verbirgt die technische Komplexität des Zugriffs auf entfernte Objekte.
- Kein Protokoll erforderlich: Im Gegensatz zu Sockets müssen Sie bei einer RMI-Lösung kein Protokoll für die Kommunikation zwischen Client und Server entwickeln, sparen also einen fehleranfälligen Schritt.
- Bei RMI sind Methodenaufrufe typsicher: RMI ist stärker typisiert als Sockets. Fehler können bereits zur Übersetzungszeit abgefangen werden.
- Bei RMI können mühelos Objekte hinzugefügt oder das Protokoll erweitert werden. Es ist kein Problem, ein neues entferntes Inferface anzulegen oder in der Klasse eines bereits vorhandenen entfernten Objektes eine weitere Methode hinzuzufügen.
- Bei Endpunkten ohne Java können Sie IIOP verwenden. Sie sind nicht an eine beidseitige Java-Lösung gebunden.
- Das Erzeugen von Stubobjekten ist mit der J2SE 5 einfacher geworden. Wenn Sie JRMP verwenden, brauchen Sie keine Stubobjekte mehr explizit per `rmic` zu erzeugen. (Dennoch verlangt die Anleitung unserer Beispielanwendung, daß `rmic` verwendet wird.)

Warnung: Obwohl die J2SE 5 das dynamische Erzeugen von Stubobjekten ermöglicht, um RMI-Entwicklern das explizite Aufrufen des `rmic`-Compilers auf den entfernten Klassen vor dem Programmstart zu ersparen, **wird die Verwendung des `rmic`-Compilers in den Prüfungsaufgaben noch immer verlangt**.

[28] Die intuitive Objektsemantik von RMI geht allerdings zu Lasten des Netzwerks. RMI ist durch den Kommunikationsbedarf über das Netzwerk teuer: Anfragen an die RMI-Registratur sowie die clientseitigen Stubobjekte (Stellvertreterobjekte), um die Methodenaufrufe bei entfernten Objekten transparent zu machen. Zu jedem entfernten Objekt benötigt ein clientseitiges Stellvertreterobjekt, wodurch die Performanz beeinträchtigt wird.

[29] [Baustelle ??, Seite ??] Es ist wichtig, sich darüber im Klaren zu sein, daß die Verwaltung von Threads Probleme verursachen kann, wenn Klassen vorhanden sind, die nicht im Hinblick auf Threadsicherheit konzipiert sind. Die Kontrolle und Verwaltung von Threads unterliegt bei RMI der Laufzeitumgebung, nicht aber dem Programm.

6.2.3 Klassen und Interfaces von RMI

[30] Abbildung 6-7 zeigt die RMI-Klassen und -Interfaces im Überblick.

[31] *`java.rmi.Remote`* ist wie *`Serializable`* ein Marker-Interface. Der Namensbestandteil „remote“ deutet bei Klassen und Interfaces im RMI-Kontext an, daß die Methoden eines solchen Objektes aus einer anderen Laufzeitumgebung aufgerufen werden, als derjenigen, die dieses Objekt enthält. Ein Objekt wird als entferntes Objekt betrachtet, wenn seine Klasse das Interface *`Remote`* implementiert. Die Klasse implementiert *`Remote`* allerdings *nicht direkt*, sondern ein von *`Remote`* abgeleitetes Interface.

[32] Jede in einem von *`Remote`* abgeleiteten Interface deklarierte Methode muß per `throws` eine Ausnahme vom Typ `java.rmi.RemoteException` oder einem Basistyp von `RemoteException` wie `IOException`, `Throwable` oder `Exception` deklarieren. `RemoteException` selbst ist der Basistyp aller Ausnahmen, die beim Aufrufen einer entfernten Methode ausgeworfen werden können. `RemoteException` ist eine geprüfte Ausnahme und muß daher entweder per `try-catch`-Klausel behandelt oder per `throws` in der Signatur der Methode deklariert werden, die eine Methode eines entfernten Objektes aufruft. Eine `RemoteException` kann eine der beiden folgenden Ursachen haben:

- Der Server wurde heruntergefahren, ist nicht erreichbar oder hat die Kommunikation mit dem Client abgebrochen.
- Beim Marshalling von Argumenten oder Rückgabewerten ist ein Fehler aufgetreten.

[33] Die Klasse eines entfernten Objektes muß außerdem von der abstrakten Klasse `java.rmi.server.RemoteObject` abgeleitet werden. Ein entferntes Objekt verhält sich zu `RemoteObject` wie ein „gewöhnliches“ Objekt zu `java.lang.Object`, das heißt `RemoteObject` liefert die „entfernten Äquivalente“ der `Object`-Methoden `toString()`, `hashCode()` und `equals()` für entfernte Objekte. Abbildung 6-8 zeigt die Ebenen, die beim Aufrufen einer entfernten Methode unter RMI durchlaufen werden.

[34] Die Basisklasse für RMI-Server, `java.rmi.server.RemoteServer`, ist eine weitere für die Entwicklung von RMI-Anwendungen erforderliche abstrakte Klasse. `RemoteServer` liefert den Rahmen für die Unterstützung der Semantik „entfernter Referenzen“, das heißt für Referenzen auf entfernte Objekte. `RemoteServer` definiert die zum Anlegen und Exportieren von entfernten Objekten (RMI-Servern) benötigten Methoden in abstrakter Form. Die abstrakten `RemoteServer`-Methoden

sind in den von `RemoteServer` abgeleiteten Klassen konkret implementiert. Zwei wichtige Klassen sind von `RemoteServer` abgeleitet: `java.rmi.server.UnicastRemoteObject` und `java.rmi.activation.Activatable`.

[35] Die beiden Klassen `UnicastRemoteObject` und `Activatable` sind für das Exportieren entfernter Objekte zuständig. Ein `Activatable`-Objekt wird bei Bedarf (beim Aufrufen einer Methode) aktiviert und kann nach der Verarbeitung des Methodenaufrufs wieder deaktiviert werden. Ein `UnicastRemoteObject`-Objekt ist dagegen entweder aktiviert oder nicht vorhanden. Ein Objekt heißt „aktiv", wenn es erzeugt und in eine Laufzeitumgebung exportiert wurde. Ein Objekt heißt dagegen „passiv", wenn es sich noch im persistent gespeicherten Zustand befindet. „Aktivierung" (*activation*) ist der Vorgang durch den ein passives Objekt in den aktiven Zustand übergeht. „Verzögerte Aktivierung" (*lazy activation*) bedeutet, daß ein Objekt erst bei Bedarf aktiviert wird. Die Klasse `Activatable` gestattet RMI, den Zugriff auf persistent gespeicherte Objekte bis zum ersten tatsächlichen Zugriff aufzuschieben. Ein RMI-Server soll keine teueren Systemresourcen vergeuden, indem er viele entfernte Objekte lädt, die selten oder gar nicht benötigt werden. Im Idealfall haben verteilte Systeme Zugriff auf tausende persistent gespeicherte Objekte über lange, eventuell sogar unbeschränkte Zeiträume hinweg.

[36] Ist die Klasse eines RMI-Servers von `UnicastRemoteObject` abgeleitet, so müssen die entfernten Objekte erzeugt worden sein, bevor ein Client eine entfernte Methode eines solchen Objektes aufrufen kann. Falls aber viele entfernte Objekte benötigt werden und der Aufwand des Erzeugens all dieser Objekte nicht akzeptabel ist, ist die Ableitung der Serverklassen von `Activatable` eine ausgezeichnete Alternative. Der RMI-Daemon `rmid` startet den Daemon des Aktivierungssystems, so daß Objekte von Typ `Activatable` bei Bedarf aktiviert werden.

[37] Beim Implementieren des RMI-Servers für unsere Beispielanwendung brauchen wir uns über diesen Aspekt der Resourcenverwaltung zum Glück nicht den Kopf zu zerbrechen, da wir nur ein einziges entferntes Objekt vom Typ `DvdDatabaseImpl` verwenden, das sowohl die Aufgabe des RMI-Servers übernimmt als auch den Zugriff auf die Datenbankdatei bewerkstelligt. Die Serverklasse `DvdDatabaseImpl` der Beispielanwendung ist daher nicht von `Activatable`, sondern von `UnicastRemoteObject` abgeleitet.

6.2.4 RMI und das Factory-Entwurfsmuster

[38] Eine „RMI-Fabrik", genauer ein „RMI-Fabrikobjekt", ist eine Implementierung des Entwurfsmusters *Factory* im RMI-Kontext. Wir haben bereits im Rahmen der Diskussion über das Reservieren (Sperren) von Datensätzen in Kapitel 5 angesprochen, daß in der Beispielanwendung ein RMI-Fabrikobjekt verwendet wird, um den Verursacher der Reservierung eines Datensatzes zu identifizieren (Seite 146). Indem jeder RMI-Client mit seinem eigenen `DvdDatabase`-Objekt kommuniziert, verhindern wir, daß ein einziges `DvdDatabase`-Objekt von mehreren Threads verwendet wird. Dieser Effekt tritt bei RMI häufig auf und läßt sich nicht über eine Eigenschaft der Laufzeitumgebung oder eine RMI-Eigenschaft kontrollieren. Da wir die Zuweisung der serverseitigen Threads zur Verarbeitung clientseitiger Anfragen (Aufrufe entfernter Methoden) nicht kontrollieren können und keine Cookies oder Merkmale zur Identifizierung der Quelle einer Anfrage verwenden dürfen, nutzen wir das Entwurfsmuster *Factory*, um *clientspezifische* `DvdDatabase`-Objekte zu erzeugen. Das Ablaufdiagramm in Abbildung 6-9 zeigt die typischen Aktionen und Akteure des *Factory*-Entwurfsmusters.

6.2.4.1 Das Entwurfsmuster

[39] Ein Fabrikobjekt ist ein Stück Software, das andere Stücke Software erzeugt. Ein Client fordert beim Fabrikobjekt ein Objekt eines bestimmten Typs an und das Fabrikobjekt erzeugt es. Verlangt

der Client ein Objekt eines anderen Typs, so erzeugt das Fabrikobjekt auch dieses. Das Fabrikobjekt muß natürlich „wissen“, wie ein Objekt des verlangten Typs erzeugt wird. Anderfalls meldet das Fabrikobjekt, daß es die Anfrage nicht bearbeiten kann. Sie können beispielsweise keinen PC bei einer Fabrik bestellen, die Autos baut. Unser Fabrikobjekt „versteht“ analog, wie entfernte Objekte erzeugt werden, genauer wie entfernte Objekte vom Typ `DvdDatabase` erzeugt werden.

[40] Unser Fabrikobjekt erzeugt nur Objekte eines einzigen Typs, nämlich `sampleproject.db.DvdDatabase`. Einer der häufigsten Gründe, sich für ein Fabrikobjekt zu entscheiden besteht darin, die Anzahl der zu registrierenden entfernten Objekte zu reduzieren. Das RMI-Fabrikobjekt muß nur einmal registiert werden und erzeugt anschließend neue entfernte Objekte, die nicht registiert werden müssen. Insbesondere muß die RMI-Registratur nicht neu gestartet werden. Dies ist ein erfreulicher Nebeneffekt bei RMI-Fabrikobjekten, den wir in unserer Beispielanwendung aber nicht nutzen.

[41] Das RMI-Fabrikobjekt in der Beispielanwendung ist eine parametrisierte Version des *Factory*-Entwurfsmusters. (*Factory* gehört zu den sogenannten erzeugenden Entwurfsmustern (*creational patterns*), welche Objekte und/oder Typen erzeugen. Ein anderes bekanntes Beispiel aus dem Kreis der erzeugenden Entwurfsmuster ist *Singleton*.) Unser RMI-Fabrikobjekt weicht von der eigentlichen Form des parametrisierten Entwurfsmusters ab, da die `getClient()`-Methode ohne Argument aufgerufen wird. Wir hätten zwar einen `String`-Parameter definieren können, etwa `getClient(String class_name_to_create)`, da es in der Beispielanwendung aber nur ein entferntes Interface gibt (*`DvdDatabaseRemote`*), ist der Typ der erzeugten Objekte offensichtlich (`DvdDatabase`) und muß nicht angegeben werden. Wir könnten die `getClient()`-Methode oder die Klasse unseres RMI-Fabrikobjektes aber mühelos ändern, um Objekte verschiedener Typen erzeugen zu können.

6.2.4.2 Die Implementierung des Entwurfsmusters

Bemerkung: In diesem Abschnitt diskutieren wir das Entwurfsmuster *Factory* im Kontext der RMI-Netzwerkschnittstelle unserer Beispielanwendung. Das Entwurfsmuster eignet sich aber ebenso gut für Sockets als Netzwerkschnittstelle. Der Socketserver funktioniert aber schon von sich aus wie ein ein Fabrikobjekt, indem er pro Clientverbindung einen eindeutigen Thread erzeugt. Es besteht also keine Veranlassung zu jeder Clientverbindung ein eindeutiges Objekt zu erzeugen. Die Funktionsweise des multithreadfähigen Socketservers verhindert die Probleme durch mehrfache Verwendung serverseitiger Threads bei RMI.

---

[42] Wir entwickeln nun die serverseitige Klasse, die das *Factory*-Entwurfsmuster implementiert, deren Objekte also Fabrikobjekte sind. Wir legen zunächst das Interface *`sampleproject.remote.DvdDatabaseFactory`* an, das lediglich die Methode `getClient()` deklariert. Die `getClient()`-Methode gibt eine Referenz auf ein `DvdDatabaseImpl`-Objekt zurück (siehe unten):

```
interface DvdDatabaseFactory extends Remote {
    public DvdDatabaseRemote getClient() throws RemoteException;
}

class DvdDatabaseFactoryImpl extends UnicastRemoteObject
        implements DvdDatabaseFactory {
    /**
     * A version number for this class so that serialization can occur
     * without worrying about the underlying class changing between
     * serialization and deserialization.
     */
    private static final long serialVersionUID = 5165L;
```

```
        public DvdDatabaseFactoryImpl() throws RemoteException {
            // do nothing constructor
        }

        public DvdDatabaseRemote getClient() throws RemoteException {
            return new DvdDatabaseImpl();
        }
    }
```

Wir legen nun das entfernte Marker-Interface *sampleproject.remote.DvdDatabaseRemote* an, welches wie alle entfernten Typen das Interface *Remote* erweitert. Die Klasse `sampleproject.remote.DvdDatabaseImpl` implementiert *DvdDatabaseRemote* (indirekt also auch *Remote*) und ist von `UnicastRemoteObject` abgeleitet:

```
    public interface DvdDatabaseRemote extends Remote, DBClient {}

    public class DvdDatabaseImpl extends UnicastRemoteObject
        implements DvdDatabaseImpl {
        // ... refer to code base for the rest of implementation ...
    }
```

[43] Schließlich können wir das Fabrikobjekt mit Hilfe der `register()`-Methode in der Hilfsklasse `RegDvdDatabase` bei der RMI-Registratur anmelden. Die statische `LocateRegistry`-Methode `createRegistry()` startet die RMI-Registratur programmatisch:

```
    public static void register(String dbLocation, int rmiPort)
            throws RemoteException {
        Registry r = java.rmi.registry.LocateRegistry.createRegistry(rmiPort);

        // make a dvd database instance on a random port and register
        // our service name and our port number on the RMI registry.
        r.rebind("DvdMediator", new DvdDatabaseFactoryImpl(dbLocation));
    }
```

Bemerkung: Der programmatische Start der RMI-Registratur ist in allen neuen Prüfungsaufgaben Pflicht. Das Starten in einem separaten Schritt wird von Sun Microsystems nicht mehr gestattet.

---

[44] Die Hilfsklasse `RegDvdDatabase` verknüpft das entfernte Fabrikobjekt (`DvdDatabaseFactoryImpl`) mit dem Namen „DvdMediator“. Der Client, eine in Swing entwickelte graphische Benutzeroberfläche (siehe übernächstes Kapitel), braucht die Klasse `DvdConnector`, um eine entfernte Referenz auf die Datenbankdatei anzufordern. Die `getRemote()`-Methode der Klasse `DvdConnector` gibt entweder eine Referenz auf ein entferntes `DvdDatabaseFactoryImpl`-Objekt oder ein `DvdDatabase`-Objekt zurück. (`DvdDatabase` ist der Wrappertyp von `DvdFileAccess`, beide Rückgabetypen von `getRemote()` sind *DBClient*.)

[45] Beachten Sie, daß nur das Fabrikobjekt (`DvdDatabaseFactoryImpl`) registriert wird. Nicht einmal das `DvdDatabaseImpl`-Objekt muß registiert werden. Die `getClient()`-Methode der Klasse `DvdDatabaseFactoryImpl` gibt eine `DvdDatabaseImpl`-Referenz zurück:

```
    public DvdDatabaseRemote getClient() throws RemoteException {
        return new DvdDatabaseImpl();
    }
```

[46] And the reason for the factory in the first place is the new instance of the database itself, which can be found in the constructor of our DvdDatabase remote implementation:

```
    public DvdDatabaseImpl(String dbLocation) throws RemoteException {
        try {
```

```
            db = new DvdDatabase(dbLocation);
        } catch (FileNotFoundException e) {
            throw new RemoteException(e.getMessage(), e);
        } catch (IOException e) {
            throw new RemoteException(e.getMessage(), e);
        }
    }
```

[47] Wir erhalten somit zu jedem verbundenen Client ein Objekt der Klasse `DvdDatabaseImpl`. Ist jedes `DvdDatabaseImpl`-Objekt außerdem über ein `db`-Feld mit seinem eigenen `DvdDatabase`-Objekt verknüpft, so läßt dieses `DvdDatabase`-Objekt dazu verwenden, daß das `ReservationsManager`-Objekt den Client identifizieren kann.

[48] Da sich das *Factory*-Entwurfsmuster sowohl für Sockets als auch für RMI als Netzwerkschnittstelle eignet und wir in diesem Buch beide Ansätze behandeln, we will present the ReservationsManager code as though a factory solution is being used.

[49] Wir wollen nun eine Lösung mit Fabrikobjekt vorführen und anschließend mit einer Lösung ohne Fabrikobjekt vergleichen. Die Musterlösung unserer Beispielanwendung, die Sie aus dem „Source Code"-Abschnitt der Apress-Website (*http://www.apress.com/book/sourcecode*) herunterladen können, enthält die beiden Klassen `sampleproject.remote.RmiFactoryExample` und `sampleproject.remote.RmiNoFactoryExample`. Wie Sie anhand der Klassennamen vermutet haben werden, zeigt die Klasse `RmiFactoryExample` die Verwendung eines Fabrikobjektes, die Klasse `RmiNoFactoryExample` dagegen nicht. Die Klasse `RmiFactoryExample` bewirkt, daß pro Anfrage ein `DvdDatabase`-Objekt erzeugt wird. Die Klasse `RmiNoFactoryExample` bewirkt dagegen, daß beide Anfragen durch „Wiederverwendung" eines einzigen `DvdDatabase`-Objektes verarbeitet werden.

[50] Sowohl `RmiFactoryExample` als auch `RmiNoFactoryExample` ist von `Thread` abgeleitet und implementiert die `run()`-Methode. Die `main()`-Methoden erzeugen jeweils zwei Threads, um mehrere Clients zu simulieren. Die Klassen unterscheiden sich nur durch das registierte entfernte Objekt. `RmiNoFactoryExample` registriert ein entferntes `DvdDatabase`-Objekt, während `RmiFactoryExample` das entfernte Fabrikobjekt (`DvdDatabaseFactoryImpl`) registriert. Dementsprechend müssen die `run()`-Methoden die von der `lookup()`-Methode erhaltene Referenz umwandeln. Die Methoden `main()` und `run()` der Klasse `RmiNoFactoryExample` lauten:

```
    public static void main(String[] args) throws Exception {

        LocateRegistry.createRegistry(1099);
        Naming.rebind("RmiNoFactoryExample", new DvdDatabaseImpl(".."));

        Thread a = new RmiNoFactoryExample("A");
        a.start();

        Thread.sleep(1000);

        Thread b = new RmiNoFactoryExample("B");
        b.start();

        a.join();
        b.join();

        System.exit(0);
    }

    public void run() {
        try {
            System.out.println("Getting a remote handle to a DvdDatabase."
                + this.hashCode());
            DvdDatabaseRemote remote
                = (DvdDatabaseRemote) Naming.lookup("RmiNoFactoryExample");
```

```
        } catch (Exception e) {
            System.err.println(e);
            e.printStackTrace();
        }
    }
```

[51] Der Quelltext der Klasse `RmiFactoryExample` ist sehr ähnlich. Die `main()`-Methode unterscheiden sich nur in der Zeile

```
Naming.rebind("RmiFactoryExample",
              new DvdDatabaseFactoryImpl("../../../../dvd_db.dvd"));
```

sowie beim zweimaligen Aufruf des jeweiligen Konstruktors. Die `run()`-Methode der Klasse `RmiFactoryExample` lautet:

```
public void run() {
    try {
        System.out.println("Getting a remote handle to a factory."
            + this.hashCode());
        DvdDatabaseFactory factory
            = (DvdDatabaseFactory) Naming.lookup("RmiNoFactoryExample");
        DvdDatabaseRemote worker = factory.getClient();
    } catch (Exception e) {
        System.err.println(e);
        e.printStackTrace();
    }
}
```

[52] Rufen Sie das folgende Kommando im Verzeichnis *classes* des Beispielprojektes auf, um das Testprogramm `RmiFactoryExample` zu starten:

```
java sampleproject.remote.RmiNoFactoryExample
```

Für das Testprogramm `RmiNoFactoryExample` lautet die Kommandozeile:

```
java sampleproject.remote.RmiFactoryExample
```

Die Ausgaben der beiden Programme lauten:

```
~$ java sampleproject.remote.RmiFactoryExample

Getting a remote handle to a factory. 15842168
 constructing a DvdDatabase object 19106770
Getting a remote handle to a factory. 29131495
 constructing a DvdDatabase object 31822120

~$ java sampleproject.remote.RmiNoFactoryExample

 constructing a DvdDatabase object 14867177
getting a remote handle to a DvdDatabase.12227392
getting a remote handle to a DvdDatabase.22048196

~$
```

[53] Die Klasse `DvdDatabase` erhält eine zusätzliche Ausgabezeile, um anzuzeigen, daß der Konstruktor aufgerufen wurde:

```
System.out.println(" constructing a a DvdDatabase object " + this.hashCode());
```

[54] Die Ausgaben der beiden Programme verdeutlichen einen wesentlichen Unterschied, nämlich die Anzahl der erzeugten `DvdDatabase`-Objekte. `RmiFactoryExample` erzeugt zwei Objekte, `RmiNoFactoryExample` dagegen nur eines. Dieses Beispiel demonstriert die mehrfache Verwendung eines serverseitigen Threads bei RMI. Zu jedem Thread gehört ein separates `DvdDatabase`-Objekt. Da das

RmiNoFactoryExample-Beispiel einen der Threads wiederverwendet, wird das DvdDatabase-Objekt von zwei Aufrufen gemeinsam verwendet. Threadsicherheit ist nicht garantiert und unser Sperransatz scheitert. (Lesen Sie in Kapitel 5 ab Seite 144 nach, warum die Reservierung von Datensätzen ein eindeutiges `DvdDatabase`-Objekt voraussetzt.)

[55] Das `RmiFactoryExample`-Beispiel erzeugt dagegen zwei separate `DvdDatabaseImpl`-Objekte, wie Sie an der Meldung des Konstruktoraufrufs nachvollziehen können.

[56] Wir wollen die öffentlichen Methoden als entfernte Methoden sichtbar machen, damit sie von der graphischen Oberfläche aus aufgerufen werden können. Abbildung 6-11 zeigt ein Klassendiagramm für die entfernte Klasse `DvdDatabaseImpl`.

Bemerkung: Die Vorteile durch das Ableiten einer entfernten Klasse von der abstrakten Klasse `Activatable` gehen über die Anforderungen an unsere Beispielanwendung hinaus. Wir haben uns daher entschieden, die entfernten Klassen `DvdDatabaseImpl` und `DvdDatabaseFactoryImpl` von der einfacheren Klasse `UnicastRemoteObject` abzuleiten.

---

[57] Der Quelltext des entferntes Interfaces *`DvdDatabaseRemote`* ist kurz und bündig:

```
package sampleproject.remote;

import java.rmi.Remote;
import sampleproject.db.DBClient;

/**
 * The remote interface for the GUI-Client.
 * Exactly matches the DBClient interface in the db package.
 *
 * @author Denny's DVDs
 * @version 2.0
 */
public interface DvdDatabaseRemote extends Remote, DBClient {}
```

[58] Sie verstehen das von *`DvdDatabaseRemote`* beschriebene Verhalten, wenn wir das vorgegebene Interface *`DBClient`* nochmals betrachten. Abbildung 6-12 beschreibt die Zusammenhänge zwischen *`DBClient`* und den anderen Typen der Beispielanwendung *Denny's DVDs*. Der Quelltext des Interfaces *`DBClient`* lautet:

```
public interface DBClient {

    /**
     * Adds a DVD to the database or inventory.
     *
     * @param dvd The DVD item to add to inventory.
     * @return Indicates the success/failure of the add operation.
     * @throws IOException Indicates there is a problem accessing the database.
     */
    public boolean addDVD(DVD dvd) throws IOException;

    /**
     * Locates a DVD using the UPC identification number.
     *
     * @param UPC The UPC of the DVD to locate.
     * @return The DVD object which matches the UPC.
     * @throws IOException if there is a problem accessing the data.
     */
    public DVD getDVD(String UPC)throws IOException;
```

```
/**
 * Changes existing information of a DVD item.
 * Modifications can occur on any of the attributes of DVD except UPC.
 * The UPC is used to identify the DVD to be modified.
 *
 * @param dvd The Dvd to modify.
 * @return Returns true if the DVD was found and modified.
 * @throws IOException Indicates there is a problem accessing the data.
 */
public  boolean modifyDVD(DVD dvd) throws IOException;

/**
 * Removes DVDs from inventory using the unique UPC.
 *
 * @param UPC The UPC or key of the DVD to be removed.
 * @return Returns true if the UPC was found and the DVD was removed.
 * @throws IOException Indicates there is a problem accessing the data.
 */
public  boolean removeDVD(String UPC) throws IOException;

/**
 * Gets the store's inventory.
 * All of the DVDs in the system.
 *
 * @return A List containing all found DVD's.
 * @throws IOException Indicates there is a problem accessing the data.
 */
public List<DVD> getDVDs() throws IOException;

 /**
  * A properly formatted <code>String</code> expressions returns all
  * matching DVD items. The <code>String</code> must be formatted as a
  * regular expression.
  *
  * @param query The formatted regular expression used as the search
  * criteria.
  * @return The list of DVDs that match the query. Can be an empty
  * Collection.
  * @throws IOException Indicates there is a problem accessing the data.
  * @throws PatternSyntaxException Indicates there is a syntax problem in
  * the regular expression.
 */
public Collection<DVD> findDVD(String query)
        throws IOException, PatternSyntaxException;

/**
 * Lock the requested DVD. This method blocks until the lock succeeds,
 * or for a maximum of 5 seconds, whichever comes first.
 *
 * @param UPC The UPC of the DVD to reserve
 * @return true if the DVD was reserved
 * @throws InterruptedException Indicates the thread is interrupted.
 * @throws IOException on any network problem
 */
boolean reserveDVD(String UPC) throws IOException, InterruptedException;

/**
 * Unlock the requested record. Ignored if the caller does not have
 * a current lock on the requested record.
 *
```

```
         * @param UPC The UPC of the DVD to release
         * @throws IOException on any network problem
         */
        void releaseDVD(String UPC) throws IOException;
    }
```

[59] Wir brauchen noch eine Klasse, damit unserer RMI-Implementierung komplett ist: `sampleproject.remote.DvdDatabaseImpl`. Diese Klasse implementiert das entfernte Interface *`DvdDatabaseRemote`*. Ein großer Teil des Quelltextes wurde ausgelassen, um Platz zu sparen. Sie finden *DvdDatabaseImpl.java* in der Distribution der Beispielanwendung, die Sie aus dem „Source Code"-Abschnitt der Apress-Website (*http://www.apress.com/book/sourcecode*) herunterladen können. Beachten Sie, daß wir die Klasse `DvdDatabaseImpl` von `java.rmi.server.UnicastRemoteObject` abgeleitet haben, nicht von `java.rmi.activation.Activatable`. Der Quelltext der Klasse `DvdDatabaseImpl` lautet:

```
    public class DvdDatabaseImpl extends UnicastRemoteObject
            implements DvdDatabaseRemote {
        /**
         * A version number for this class so that serialization can occur
         * without worrying about the underlying class changing between
         * serialization and deserialization.
         */
        private static final long serialVersionUID = 5165L;

        /**
         * The Logger instance. All log messages from this class are routed through
         * this member. The Logger namespace is <code>sampleproject.remote</code>.
         */
        private static Logger log = Logger.getLogger("sampleproject.remote");

        /**
         * The database handle.
         */
        private DBClient db = null;

        /**
         * DvdDatabaseImpl default constructor.
         *
         * @param dbLocation the location of the database.
         * @throws RemoteException Thrown if a <code>DvdDatabaseImpl</code>
         * instance cannot be created.
         */
        public DvdDatabaseImpl(String dbLocation) throws RemoteException {
            try {
                db = new DvdDatabase(dbLocation);
            } catch (FileNotFoundException e) {
                throw new RemoteException(e.getMessage(), e);
            } catch (IOException e) {
                throw new RemoteException(e.getMessage(), e);
            }
        }

        /**
         * Returns the sampleproject.db.Dvd object matching the UPC.
         *
         * @param upc The upc code of the Dvd to retrieve.
         * @return The matching Dvd object.
         * @throws RemoteException  Thrown if an exception occurs in the
         * <code>DvdDatabaseImpl</code> class.
```

```
 * @throws IOException Thrown if an <code>IOException</code> is
 * encountered in the <code>db</code> class.
 * <br>
 * For more information, see {@link DvdDatabase}.
 * <br>
 * For more information, see {@link DvdDatabase}.
 */
public DVD getDVD(String upc) throws RemoteException, IOException {
    return db.getDVD(upc);
}

/**
 * Gets the store's inventory.
 * All of the Dvds in the system.
 *
 * @return A collection of all found Dvd's.
 * @throws IOException Indicates there is a problem accessing the data.
 */
 public List<DVD> getDVDs() throws IOException {
    return db.getDVDs();
}

/**
 * A properly formatted <code>String</code> expressions returns all matching
 * Dvd items. The <code>String</code> must be formatted as a regular
 * expression.
 *
 * @param query A regular expression searcg string.
 * @return A <code>Collection</code> of <code>Dvd</code> objects that match
 * the search criteria.
 * @throws IOException Thrown if an <code>IOException</code> is
 * encountered in the <code>db</code> class.
 * @throws PatternSyntaxException Thrown if an
 * <code>PatternSyntaxException</code> is encountered in the
 * <code>db</code> class.
 */
 public  Collection<DVD> findDVD(String query)
        throws IOException, PatternSyntaxException {
    return db.findDVD(query);
 }

/**
 * Modifies a Dvd database entry specified by a Dvd object.
 *
 * @param item The Dvd to modify.
 * @return A boolean indicating the success or failure of the modify
 * operation.
 * @throws RemoteException  Thrown if an exception occurs in the
 * <code>DvdDatabaseImpl</code> class.
 * @throws IOException Thrown if an <code>IOException</code> is
 * encountered in the <code>db</code> class.
 * <br>
 * For more information, see {@link DvdDatabase}.
 */
public boolean modifyDVD(DVD item) throws RemoteException, IOException {
    return db.modifyDVD(item);
}

/**
```

```
     * Removes a Dvd database entry specified by a UPC.
     *
     * @param upc The UPC number of the Dvd to remove.
     * @return A boolean indicating the success or failure of the removal
     * operation.
     * @throws RemoteException  Thrown if an exception occurs in the
     * <code>DvdDatabaseImpl</code> class.
     * @throws IOException Thrown if an <code>IOException</code> is
     * encountered in the <code>db</code> class.
     * <br>
     * For more information, see {@link DvdDatabase}.
     */
    public boolean removeDVD(String upc) throws RemoteException, IOException {
        return db.removeDVD(upc);
    }

    /**
     * Lock the requested Dvd. This method blocks until the lock succeeds,
     * or for a maximum of 5 seconds, whichever comes first.
     *
     * @param upc The upc of the Dvd to reserve
     * @return Indicates the success/failure of the add operation.
     * @throws InterruptedException Indicates the thread is interrupted.
     * @throws IOException Thrown if an <code>IOException</code> is
     * encountered in the <code>db</code> class.
     */
    public boolean reserveDVD(String upc)
            throws InterruptedException, IOException {
        return db.reserveDVD(upc);
    }

    /**
     * Unlock the requested record. Ignored if the caller does not have
     * a current lock on the requested record.
     *
     * @param upc The upc of the Dvd to release
     * @throws IOException Thrown if an <code>IOException</code> is
     * encountered in the <code>db</code> class.
     */
    public void releaseDVD(String upc) throws IOException {
        db.releaseDVD(upc);
    }

    /**
     * Adds a dvd to the database or inventory.
     *
     * @param dvd The Dvd item to add to inventory.
     * @return Indicates the success/failure of the add operation.
     * @throws IOException Indicates there is a problem accessing the database.
     * @throws RemoteException  Thrown if an exception occurs in the
     * <code>DvdDatabaseImpl</code> class.
     */
    public boolean addDVD(DVD dvd) throws IOException, RemoteException {
        return db.addDVD(dvd);
    }
}
```

[60] Die Klasse `DvdDatabaseImpl` ist dadurch interessant, daß sie die Datenbankdatei mit Hilfe eines Wrappers (Adapters) anspricht, nämlich über ein Objekt der Klasse `DvdDatabase`. Es wäre

auch in Frage gekommen, die Methoden für den Zugriff auf die Datenbankdatei aus der Klasse `DvdFileAccess` direkt in der entfernten Klasse `DvdDatabaseImpl` anzulegen. Wir hätten auch die Klasse `DvdFileAccess` von der abstrakten Klasse `UnicastRemoteObject` ableiten und das `DvdFileAccess`-Objekt als entferntes Objekt wählen können. Unsere Entscheidung für die Adapterlösung beruht auf den folgenden Überlegungen:

- Eine zusätzliche Abstraktionsebene zwischen der RMI-Implementierung und den eigentlichen Anweisungen für den Zugriff auf die Datenbankdatei verdeutlicht den objektorientierten Entwurf und unterstützt die Trennung der RMI-basierten Netzwerkschnittstelle von der Anwendungslogik. Es war unser Ziel, die Anwendungslogik durch eine Abstraktionsebene von den beiden Varianten der Netzwerkschnittstelle (RMI und Sockets) zu trennen. Dieser Ansatz gestattet, einfach zwischen den beiden Varianten zu wählen. Es gibt keine Abhängigkeiten zwischen den Packages `sampleproject.remote` beziehungsweise `sampleproject.sockets` und `sampleproject.db` mehr. Wir greifen diesen Gesichtspunkt in Kapitel 9 noch einmal auf.

- Die korrekte Funktionsweise unseres Reservierungsverfahrens setzt voraus, daß jede Anfrage, die eine Änderung eines Datensatzes bewirkt, diesen reserviert, bevor die Änderung durchgeführt wird, wobei die Quelle der Anfrage mit Hilfe eindeutiger `DvdDatabase`-Objekte identifiziert werden kann. Wir brauchen ein eindeutiges `DvdDatabase`-Objekt, um den Verursacher einer Reservierung feststellen zu können. Andernfalls könnte nicht gewährleistet werden, daß nur der Client, der einen Datensatz reserviert hat, diesen Datensatz ändern, löschen oder seine Reservierung aufheben darf. Nach erfolgter Reservierung kann der Datensatz sicher bearbeitet werden. Da diese Operationen nicht in einer synchronisierten Methode stehen dürfen (Unterabschnitt 4.1.4), die Änderungsmethoden für den Zugriff auf die Datenbankdatei aber synchronisiert sein müssen, sind wir gezwungen, die Datenbankdatei mit Hilfe eines Adapters anzusprechen.

[61] Die Klassen einer RMI-Implementierung müssen threadsicher sein, da ein entferntes Objekt gleichzeitig von mehr als einem Thread beansprucht werden kann, wenn die RMI-Laufzeitumgebung mehrere Threads startet. Nach Abschnitt „3.2 Thread Usage in Remote Method Invocations“ der RMI-Spezifikation gibt es keine Garantie dafür, daß eine Anfrage von einem bestimmten Thread verarbeitet wird. Als Entwickler sind Sie dafür verantwortlich, daß entfernte Objekte threadsicher sind. In unserem Fall gewährleistet die Anwendung des *Factory*-Entwurfsmuster in der Netzwerkschnittstelle zusammen mit den `DvdDatabase`-Methoden `reserveDVD()` und `releaseDVD()`, daß die Implementierung threadsicher ist.

[62] Wenn Sie Ihre Klasse nicht von `UnicastRemoteObject` ableiten, sondern lieber die statische `UnicastRemoteObject`-Methode `exportObject()` im Konstruktor aufrufen möchten, müssen Sie die `Object`-Methoden `hashCode()`, `toString()` und `equals()` implementieren. Der folgende kurze Quelltextauszug zeigt, wie der `DvdDatabaseImpl`-Konstruktor diese Möglichkeit implementiert (der restliche Quelltext wurde ausgelassen, um Platz zu sparen):

```
public class DvdDatabaseImpl implements DvdDatabaseRemote {
    public DvdDatabaseImpl() throws RemoteException {
        UnicastRemoteObject.exportObject(this);
        this.dvdDatabase = new DvdDatabase();
    }
}
```

6.2.4.3 Stub- und Skeletonobjekte

[63] Ein RMI-Client ruft lediglich die Methoden sogenannter „Stubobjekte“ auf, den lokalen Äquivalenten (Stellvertretern) der entfernten Objekte. Obwohl der RMI-Client scheinbar direkt eine

entfernte Methode aufruft, wird eigentlich und ohne Wissen des RMI-Clients eine Stellvertretermethode des Stubobjektes aufgerufen, welches sich um die Kommunikation mit dem entfernten Objekt in dessen Laufzeitumgebung kümmert. Das Stubobjekt ist dafür zuständig, die beim Aufruf der entfernten Methode übergebenen Argumente zu verpacken, bevor sie über das Netzwerk übertragen werden. Das Verpacken der Argumente vor dem Transport über das Netzwerk wird als Marshalling bezeichnet.

[64] Auf der Serverseite der Netzwerkverbindung nimmt die entfernte reference layer den eingegangenen Methodenaufruf entgegen, packt dessen Argumente aus und leitet den Aufruf an den tatsächlichen RMI-Server weiter. Anschließend verpackt der RMI-Server den Rückgabewert und sendet ihn an die Quelle der Anfrage zurück.

[65] Die J2SE 5 befreit die Entwickler von der Notwendigkeit, clientseitig per `rmic`-Compiler Stubklassen generieren zu müssen. Die Stubklassen werden nun dynamisch zur Laufzeit generiert. (Wir sind allerdings bei unserem Beispielprojekt noch immer an die Verwendung des `rmic`-Compilers gebunden.) Wenn Sie unbedingt dynamische Stubklassen generieren möchten, setzen Sie beim Aufruf der Laufzeitumgebung die Eigenschaft `java.rmi.server.ignoreStubClasses` auf `true`:

```
java -Djava.rmi.server.ignoreStubClasses=true
```

Bemerkung: Sie *müssen* auf den `rmic`-Compiler zurückgreifen, um Stubklassen zu erzeugen, wenn Sie Rückwärtskompatibilität mit Clients vor der J2SE 5 gewährleisten wollen oder müssen.

---

[66] Der `rmic`-Compiler kann je nach Kontext auf verschiedene Weise verwendet werden. Der `rmic`-Compiler gestattet, Rückwärtskompatibilität mit früheren Java-Versionen herzustellen, zum Beispiel Java 1.1 (Stub- und Skeletonobjekte erforderlich), Java 1.2 (nur Stubobjekte erforderlich). Die J2SE 5 verlangt keine Skeletonobjekte mehr. Sie sind dennoch gezwungen, bei Ihrer Prüfungsaufgabe den `rmic`-Compiler zu verwenden.

Bemerkung: Das Interface *`java.rmi.server.Skeleton`* ist seit Java-Version 1.2 als *deprecated* deklariert. Der folgende Aufruf des `rmic`-Compilers generiert eine Stubklasse zu `DvdDatabaseImpl` speziell für die Java-Version 1.2:

```
rmic -v1.2 sampleproject.remote.DvdDatabaseImpl
```

Bemerkung: RMI macht vom *Proxy*-Entwurfsmuster (*Stellvertreter*) Gebrauch. Dieses Entwurfsmuster bezeichnet ein Stellvertreterobjekt, um den Zugriff auf das eigentlich Objekt zu steuern und besteht aus drei wesentlichen Komponenten: einem „Subjekt“, einem „Stellvertreter“ und einem „realen Subjekt“. Der Stellvertreter referenziert das reale Subjekt und ist für die Kommunikation zwischen beiden zuständig. Das Subjekt ist die vom realen Subjekt und vom Stellvertreter gemeinsam verwendete Schnittstelle. Das reale Subjekt ist die Implementierung des vom Stellvertreter repräsentierten Objektes und entspricht der durch das Subjekt festgelegten Schnittstelle. Bei unserer RMI-Lösung entspricht das entfernte Interface *`DvdDatabaseRemote`* dem Subjekt, das clientseitige Stubobjekt dem Stellvertreter und das entfernte `DvdDatabaseImpl`-Objekt (Implementierung von *`DvdDatabaseRemote`*) dem realen Subjekt.

```
                                +------------+
                                |  Subjekt   |
+----------+                    +------------+
|  Client  |  ---------->       |            |
+----------+                    +------------+
```

```
                        | request()  |
                        +------------+
                             +  +
                             |  |
              +--------------+  +--------------+
              |                                |

+------------------+            +------------------+
|  Reales Subjekt  |            |  Stellvertreter  |
+------------------+            +------------------+
|                  |  <-------  |                  |
+------------------+            +------------------+
|  request()       |            |  request()       |
+------------------+            +------------------+
```

Abbildung 6-13: Das Proxy-Entwurfsmuster (Stellvertreter).

Unter den folgenden Internetadressen finden Sie Informationen über den `rmic`-Compiler und Rückwärtskompatibilität zu älteren Java-Versionen: *http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/rmic.html* (Windows), *http://java.sun.com/j2se/1.5.0/docs/tooldocs/solaris/rmic.html* (Solaris, Unix).

6.2.4.4 Übergabe von Argumenten und Rückgabewerten

[67] Das Aufrufen einer entfernten Methode per RMI setzt voraus, daß Objekte miteinander kommunizieren, die in verschiedenen Laufzeitumgebungen und/oder sogar auf verschiedenen Rechnern liegen, obwohl der Methodenaufruf syntaktisch wie ein lokaler Methodenaufruf aussieht. Dieser Unterunterabschnitt beschreibt wie Argumente und Rückgabewerte zwischen verschiedenen Laufzeitumgebungen übertragen werden. Argument und Rückgabewerte lassen sich in drei Gruppen einteilen: primitive Typen, serialisierbare Objekte und entfernte Objekte.

Bemerkung: Auch Arrays von Elementen primitiven Typs und Arrays von Elementen serialisierbaren Typs können beim Aufruf entfernter Methoden übergeben werden.

---

[68] RMI arbeitet bei Argumenten und Rückgabewerten primitiven Typs mit Kopien. Wird eine Kopie eines Wertes eines primitiven Typs über das Netzwerk übertragen, so sagen wir, der Parameter oder Rückgabewert werde „als Wert“ (*by-value*) übergeben.

[69] Wird ein Objekt als Argument oder Rückgabewert einer Methode über das Netzwerk übertragen, so erzeugt RMI auf der Empfängerseite eine Kopie des Objektes und aller von ihm rekursiv referenzierten Objekte. Die einzelnen Feldinhalte des Objektes werden als Werte übertragen, wie bei den primitiven Typen.

[70] Das zu transportierende Objekt muß serialisierbar sein. Andernfalls wird eine Ausnahme vom Typ `java.io.NotSerializableException` ausgeworfen. Außerdem müssen alle von diesem Objekt referenzierten Objekte wiederum serialisierbar sein, es sei denn, die entsprechenden Felder sind als transiente Felder deklariert oder die Klasse implementiert nicht ***`Serializable`***, sondern das Interface ***`Externalizable`***.

[71] Die Rückgabe eines entfernten Objektes ist der komplizierteste Fall. Gibt eine Methode ein entferntes Objekt zurück, also ein Objekt, dessen Klasse das Interface ***`Remote`*** implementiert, so liefert die Methode anstelle des entfernten Objektes eine Referenz auf dessen Stellvertreterobjekt

Argument/Rückgabewert	Übertragungsmodus
Entferntes Objekt	Übergabe als Referenz (*pass-by-reference*): Anstelle einer Kopie wird eine Referenz auf ein Stellvertreterobjekt (Stubobjekt) des entfernten Objektes übergeben.
Lokales serialisierbares Objekt	Übergabe als Wert (*pass-by-value*): Per Serialisierung wird eine Kopie des lokalen Objektes übergeben.
Wert primitiven Typs	Übergabe als Wert: Eine Kopie wird übergeben.

Tabelle 6.3: Übergabe von Argumenten und Rückgabewerten bei RMI.

(Stubobjekt) zurück. Entfernte Objekte werden also „als Referenz“ (*by-reference*) übergeben ~~and the stub is marshaled back and forth~~.

[72] Entfernte Objekte sind stets serialisierbar. Tabelle 6.3 faßt zusammen, wie RMI die verschiedenen Typen von Argumenten und Rückgabewerten behandelt. RMI hat zwei Möglichkeiten, um eine Referenz auf ein entferntes Objekt zu übergeben, nämlich die RMI-Registratur und die oben diskutierte Übergabe eines Stellvertreterobjektes (Stubobjekt). RMI bedient sich der statischen Methoden der Klasse `java.rmi.Naming`, um Referenzen auf entfernte Objekte zu registrieren. Die statischen Methoden `bind()` und `rebind()` der Klasse `Naming` verknüpfen eine URL-artig notierte Referenz auf Objekte auf einem bestimmten Rechner unter einer bestimmten Portadresse mit einem frei wählbaren Namen. RMI ist in der Lage, Bytecode mit Hilfe eines URL-basierten Protokolls wie HTTP oder FTP herunterzuladen. Die Kommunikation eines RMI-Clients mit einem entfernten Objekt findet nicht direkt mit dem eigentlichen entfernten Objekt statt, sondern mit dem Interface des entfernten Objektes.

6.2.4.5 Sicherheit und dynamisches Laden

[73] Die Funktion einer RMI-basierten Anwendung über ein Netzwerk hinweg setzt voraus, daß die Klassenlader auf der Server- und der Clientseite bestimmte Dateien erreichen können. Tabelle 6.4 faßt zusammen, welche Dateien benötigt werden.

[74] Im einfacheren Fall liegen die Stubklassen der entfernten Objekte und eventuell weitere benötigte Klassen beim Client lokal vor, so daß wir die Situation zur Laufzeit nicht weiter betrachten müssen. Hat der Client aber nur das entfernte Interface zur Verfügung, so ist Java in der Lage, die benötigten entfernten Klassen dynamisch zu laden. Das dynamische Laden ist eine Fähigkeit von RMI, die gestattet, lokal nicht verfügbare Klassen zu beschaffen und wird durch zwei Klassen ermöglicht: `java.rmi.server.RMIClassLoader` und `java.rmi.RMISecurityManager`. Außer diesen beiden Klassen muß bekannt sein, wo sich die zu ladenden Klassen befinden.

[75] Der Ort an dem sich die zu ladenden Klassen befinden wird als „Codebase“ (*code base*) bezeich-

Clientseitiger Klassenlader	Serverseitiger Klassenlader
Entfernte Interfaces.	Entfernte Interfaces und entfernte Klassen, die diese Interfaces implementieren.
Stubklassen entfernter Objekte.	Stubklassen entfernter Objekte.
Serverklassen, die als Rückgabetypen verwendet werden.	Skeletonklassen entfernter Objekte (nur Java-Version 1.1).
Verschiedene clientseitige Klassen.	Verschiedene serverseitige Klassen.

Tabelle 6.4: Vom Klassenlader auf der Client- beziehungsweise Serverseite benötigte Dateien.

net. Sie können sich die Codebase wie einen Klassenpfad vorstellen, abgesehen davon, daß sie auf einen anderen Rechner verweist und das Format einer URL hat. Der Klassenpfad ähnelt einer lokalen Codebase. Die entsprechende Eigenschaft der Laufzeitumgebung heißt `java.rmi.server.codebase` und wird folgendermaßen bewertet:

```
java -Djava.rmi.server.codebase <URL>
```

[76/77] Das dynamische Klassenladen bei RMI setzt die Klasse `java.rmi.server.RMIClassLoader` voraus. `RMIClassLoader` wertet die Eigenschaft `java.rmi.server.codebase` aus. Die Klasse `RMIClassLoader` kann Klassen für Applets oder RMI-Anwendungen laden. Im Hinblick auf die Bedürfnisse unserer Beispielanwendung und die Anforderungen an Ihre Prüfungsaufgabe fällt die Diskussion über Sicherheit nur kurz aus. Da unsere Beispielanwendung keinen Sicherheitsmanager (*security manager*) verwendet, müssen wir uns über Sicherheitsmanager und *.policy* Dateien nicht den Kopf zerbrechen. Java 2 kann aus Kompatibilitätsgründen mit früheren Versionen einen Sicherheitsmanager verwenden. Bei einer professionellen RMI-Anwendung ist aber in der Regel ein Sicherheitsmanager vorgeschrieben. Wenn Ihre netzwerkfähige Anwendung dynamisches Klassenladen verwendet, ist ein Sicherheitsmanager unumgänglich.

6.2.4.6 Firewalls

[78] Befindet sich zwischen Client und Server eine Firewall, so verbietet die Transportebene das Anlegen von Socketverbindungen. Wenn Socketverbindungen verboten sind, weicht RMI auf sogenanntes HTTP-Tunneling aus. Beim Tunneling verpackt RMI seine Aufrufe in HTTP `POST`-Anfragen, die von Firewalls in der Regel gestattet werden.

[79] Das Tunneling wird automatisch durchgeführt. Wenn die Socketverbindung von der Transportschicht verboten wird, unternimmt RMI einen letzten Versuch und sendet die Anfrage per HTTP-Tunneling.

Bemerkung: Ein Teil der Berechtigungen von Java-Programmen werden durch sogenannte *.policy* Dateien eingestellt. Das Sicherheitsmodell von Java 2 verlangt, daß sowohl Programme als auch die RMI-Registratur berechtigt sind, Sockets anzulegen. RMI baut letztendlich auf Sockets auf. Der Client muß beim Start eine *.policy* Datei angeben. Das folgende Kommando startet die RMI-Registratur mit einer anderen, als der voreingestellten *.policy* Datei:

```
rmiregistry -J -Djava.security.policy=ourSecurityFile.policy
```

---

[80] Port 80 ist voreingestellt, falls mit der Servereigenschaft `http.proxyHost` kein anderer Port festgelegt wird. HTTP-Tunneling verursacht allerdings Unkosten. Performanz wird geopfert und die Sicherheit kann beeinträchtigt werden. Das Tunneling kann unterbunden werden, indem die Servereigenschaft `java.rmi.server.disableHttp` auf `true` gesetzt wird.

6.3 Zusammenfassung

[81] In diesem Kapitel haben wir viel gelernt. Zuerst haben wir das Thema „Serialisierung“ wiederholt. Jedes Objekt, das über ein Netzwerk hinweg versendet wird, muß vor dem Transport serialisiert werden. Sowohl Sockets (siehe folgendes Kapitel) als auch RMI setzen Serialisierung voraus. Damit eine RMI-Anwendung funktionieren kann, müssen sowohl die Argument als auch die Rückgabewerte der entsprechenden Methoden serialisierbar seien.

[82] Wir haben RMI in allen Einzelheiten behandelt und eine RMI-basierte Netzwerkschnittstelle für unsere Beispielanwendung entwickelt. Angesichts des Aufwandes für eine socketbasierte Netzwerkschnittstelle haben wir die Arbeit schätzen gelernt, die RMI unter der Haube verrichtet. Die Entwicklung eines multithreadfähigen Servers ist nicht erforderlich, da sich RMI von sich aus um diesen Gesichtspunkt kümmert. Insbesondere sind keine Kommandoobjekte (*command objects*) und Ergebnisobjekte (*result objects*) erforderlich, um einem Anwendungsprotokoll zu gehorchen.

[83] Andererseits verlangt RMI die Definition eines entfernten Interfaces und die Serialisierbarkeit sämtlicher Argumente und Rückgabewerte von Methoden. Entfernte Objekte müssen beim RMI-Namensdienst registriert und die RMI-Registratur gestartet werden, um die entfernten Objekte für entfernte Clients verfügbar zu machen. Wir hoffen, daß Sie genug über Serialisierung, Sockets und RMI gelernt haben, um eine komplette Netzwerkschnittstelle für Ihre Prüfungsaufgabe planen und entwickeln zu können. Mit dem `sampleproject.remote`-Package der Beispielanwendung als Richtlinie und diesem Kapitel sollten Sie angemessen vorbereitet sein.

6.4 Häufige Fragen

- *Frage*: Sind RMI-Serverobjekte threadsicher?

  Antwort: Nein. Die RMI-Spezifikation zur Java-Version 1.4 beinhaltet keine Garantie hinsichtlich der Anzahl von Threads, die Zugriff auf eines Ihrer entfernten Objekte erhalten. Sie müssen die Threadsicherheit Ihrer RMI-Serverobjekte daher bei Ihrer Planung berücksichtigen. Schlagen Sie zu den Themen „Sperren“ und „Threadsicherheit“ in Kapitel 4 nach. Der Ansatz in diesem Kapitel besteht darin, ein Fabrikobjekt zu verwenden, das dafür sorgt, daß jedem reservierten Datensatz ein eindeutig identifizierbares Objekt der Wrapperklasse `DvdDatabase` zugeordnet wird.

- *Frage*: Welcher Ansatz zur Entwicklung einer Netzwerkschnittstelle ist besser, RMI oder Sockets?

  Antwort: Jeder Ansatz hat Vor- und Nachteile. Sockets sind günstig, um große Datenmengen (mit oder ohne Protokoll) ohne größere Unkosten zu versenden. Bei RMI ist es leichter, einen multithreadfähigen Server zu implementieren, da RMI das Erzeugen der Threads selbst übernimmt. Ein entferntes Objekt verhält sich wie ein lokales Objekt, komplett mit Übertragungsprotokoll (öffentliche Methoden), während Sie bei Sockets selbst ein Protokoll entwickeln müssen. Entscheiden Sie selbst. Beide Ansätze funktionieren gut, wenn Sie die jeweiligen wesentlichen technischen Eigenschaften verstanden haben.

- *Frage*: Ist ein Sicherheitsmanager oder eine Form von Authentifizierung erforderlich?

  Antwort: Das hängt sehr von Ihrer spezifischen Prüfungsaufgabe ab. Lesen Sie die Anleitung sehr sorgfältig. Jede Aufgabe ist anders. Wir haben in diesem Kapitel einige Sicherheitsaspekte kurz diskutiert: *.policy* Dateien, das dynamische Klassenladen, HTTP-Tunneling, Firewalls und Sicherheitsmanager. Wir verwenden in unserem Beispielprojekt weder Sicherheitsmanager noch *.policy* Dateien.

- *Frage*: Muß ich die Skeletonklassen meiner RMI-Implementierung unter der J2SE 5 mit einreichen?

  Antwort: Ja. Skeletonklassen sind nur bei Java-Versionen vor 1.2 erforderlich, das heißt bei Version 1.1. In seiner Voreinstellung generiert der `rmic`-Compiler *keine* Skeletonklassen mehr, Sie können das Standardverhalten aber mit Hilfe des Schalters `-v1.1` ändern (bei dieser Schalterstellung werden Skeletonklassen erzeugt).

Zum Zeitpunkt der Drucklegung dieses Buches wird bei den Prüfungsaufgaben noch immer verlangt, daß Sie die Stubklassen mit Ihrer Lösung einreichen und sich nicht auf die dynamische generierten Klassen der J2SE 5 verlassen.

- *Frage*: Muß ich die RMI-Registratur manuell starten?

  Antwort: Nein, im Gegenteil: Die RMI-Registratur *muß* programmatisch gestartet werden. Lesen Sie die Anleitung zu Ihrer Prüfungsaufgabe sorgfältig. Sehr wahrscheinlich wird aber verlangt, daß Sie die Registratur so starten, wie in der Klasse `RegDvdDatabase` in der Beispielanwendung. Die Klasse `RegDvdDatabase` definiert eine `register()`-Methode, die ein Objekt der entfernten Klasse per `LocateRegistry`-Methode `register()` beim RMI-Namensdienst registriert. Nachdem Sie diese Methode aufgerufen haben, können Sie in Ihrer `main()`-Methode die Konnektormethode `getRemote()` aufrufen (in der Beispielanwendung in der Klasse `DvdConnector`).

Kapitel 7

Sockets als Netzwerkschnittstelle

[0] In diesem Kapitel entwickeln wir eine socketbasierte Netzwerkschnittstelle für unsere Beispielanwendung *Denny's DVDs*. Sockets sind eine Softwareabstraktion, die Anwendungen gestattet, über ein Netzwerk hinweg miteinander zu kommunizieren. Jede Verbindung zwischen zwei Rechnern auf der Basis des Internet Protocols (IP) beruht auf Sockets. Java versetzt den Entwickler zwar in die Lage, Anwendungen über Sockets miteinander zu verbinden, aber eine effektive Kommunikation zwischen Client und Server über Standardsockets setzt viel Arbeit voraus, wie wir in diesem Kapitel sehen werden. Java beinhaltet alternativ das auf Sockets aufbauende RMI-Framework, um dem Entwickler einen Teil dieser Arbeit zu ersparen. RMI bringt andererseits eigene Probleme mit sich (siehe Kapitel 6). Beide Netzwerkschnittstellen haben Vor- und Nachteile und als Entwickler müssen Sie bestimmen, welcher Ansatz besser zu einer gegebenen Situation paßt. Wir entwickeln in diesem Kapitel eine einfache socketbasierte Netzwerkschnittstelle für unsere Beispielanwendung und besprechen das erforderliche Hintergrundwissen, damit Sie eine sachkundige Entscheidung für Ihre eigene Prüfungsaufgabe treffen können.

[1] Wir behandeln die folgenden Themen:

- Überblick über das Thema „Sockets".
- Vor- und Nachteile von Sockets als Netzwerkschnittstelle.
- Grundlagen: Aufbau einer Socketverbindung.
- Entwickeln eines Socketclients (TCP).
- Entwickeln eines Socketservers (TCP).
- Serialisierung und Sockets.
- Der Lebenszyklus eines Socketobjekts.

7.1 Überblick

[2] Ein Socket ist einer der beiden Endpunkte der Kommunikationsverbindung zwischen zwei Programmen und stets an einen sogenannten Port gebunden. Eine Socketverbindung dient der Anbindung an ein entferntes Gerät sowie der Datenübertragung von und zu diesem Gerät. Das Aufbauen einer Socketverbindung erfordert die IP-Adresse des Zielrechners und eine Portadresse. Sockets nicht spezifisch für Java. Alle Java-Versionen ab 1.0 sind aber in der Lage, Socketverbindungen aufzubauen. Wie bei RMI diskutieren wir Socketclients und Socketserver separat.

[3] Der Socketserver ist ein lauschender Dienst und wartet auf Verbindungsanfragen von Socketclients. Akzeptiert der Socketserver eine solche Anfrage, so baut er eine Verbindung zwischen zwei Sockets auf. Der Socketclient benötigt den Namen des Rechners (oder dessen IP-Adresse) auf dem der Socketserver läuft und eine Portadresse, um eine Verbindung anzufordern. Auf der Serverseite wartet ein Objekt der Klasse `java.net.ServerSocket` auf Verbindungsanfragen von Socketclients. Auf der Clientseite einer Socketverbindung befindet sich ein Objekt der Klasse `java.net.Socket`. Both the server and the client will have an instance of java.net.Socket per socket connection.

7.1.1 Vor- und Nachteile von Sockets als Netzwerkschnittstelle

[4] Die Wahl zwischen Sockets und RMI gehört zu den größten Entscheidungen, die Sie im Rahmen Ihrer Prüfungsaufgabe treffen müssen. Beide Wahlmöglichkeiten haben Vor- und Nachteile. Ziehen wir das Forum zum *Sun Certified Java Developer* auf der Java-Ranch (*http://www.javaranch.com*) als Indikator heran, so scheinen sich die Prüfungskandidaten mehrheitlich für RMI zu entscheiden. Die erste Auflage dieses Buches enthielt die Empfehlung, sich für RMI anstelle von Sockets zu entscheiden, da wir der Auffassung waren, RMI sei für Java-Entwickler intuitiver. Seither halten wir uns mit derartigen Empfehlungen zurück. Sowohl RMI als auch Sockets funktionieren gut und sind relativ unkompliziert zu implementieren. Für die Abenteuerlustigen diskutieren wir nun einige Argumente, warum Sie sich für Sockets entscheiden könnten.

[5] Socketserver skalieren besser und sind performanter als RMI-Server. Das Anfordern einer Referenz auf ein entferntes Objekt erfordert eine Anfrage über das Netzwerk, nämlich das Nachschlagen in der RMI-Registratur. Zu jedem entfernten Objekt gehört ein clientseitiges Stellvertreterobjekt (Stubobjekt), so daß effektiv eine Ebene von Objekten zwischen dem Client und den entfernten Objekten liegt. Die Kommunikation zwischen Stellvertreter und entferntem Objekt verursacht Unkosten zulasten der Performanz.

[6] Ein Argument, das typischerweise *gegen* Sockets vorgebracht wird, ist das erforderliche Protokoll für die Kommunikation zwischen den Socketclients und dem Socketserver. Wir diskutieren dieses Thema im Unterabschnitt 7.3.4. Wenn das Protokoll nun aber sehr einfach ist? Bei einem einfachen socket interface sind Sockets eine ausgezeichnete Wahl.

[7] Ein weiteres Argument dafür, eine socketbasierte Netzwerkschnittstelle in Betracht zu ziehen, betrifft Threads. Jede von einem Socketclient ausgehende Anfrage bewirkt, daß auf der Serverseite ein neuer Thread erzeugt wird, den wir zum Sperren des `DvdDatabase`-Objektes verwenden können, das heißt wir brauchen kein Fabrikobjekt wie bei RMI (siehe Kapitel 6). Dieses grundsätzliche Merkmal von Socketservern umgeht diese potentiell komplizierte Angelegenheit. Natürlich müssen Sie sich nun mit der Entwicklung eines multithreadfähigen Servers auseinandersetzen. Die Beispielanwendung enthält einen multithreadfähigen Socketserver und wir beschreiben unsere Musterlösung in diesem Kapitel.

7.2 Grundlagen

[8] In diesem Abschnitt lernen wir die beiden Typen von Sockets kennen, die Ihnen als Java-Entwickler zur Verfügung stehen (TCP- und UDP-Sockets) und besprechen einige grundlegende Konzepte der Socket-Programmierung.

7.2.1 IP-Adressen

[9] Eine IP-Adresse („IP“ ist die Abkürzung für „Internet Protocol“) ist eine eindeutige Zahlenkombination, mit deren Hilfe ein mit einem Netzwerk verbundenes Gerät identifiziert werden kann, ebenso wie eine Postadresse den Ort eines Gebäudes in einer Stadt beschreibt. Eine IP-Adresse besteht aus vier Blöcken mit je drei dezimalen Stellen beziehungsweise aus acht Blöcken mit je vier hexadezimalen Stellen, entspricht also technisch einem vorzeichenlosen binären Wert von 32 oder 128 Bit Länge. IP-Adressen werden sowohl vom Internet Protocol (IP) selbst, als auch zum Austauschen von Nachrichten zwischen den Socketadressen über die Protokolle TCP und UDP verwendet. Die Klasse `java.net.InetAddress` kapselt eine IP-Adresse zur Verwendung bei einem Socket.

Bemerkung: Das von TCP/IP verwendete 32-Bit-Adressierungsmodell stammt aus den 1970er Jahren. Schätzungen gehen davon aus, daß der gegenwärtige Vorrat an IP-Adressen durch den zunehmenden Bedarf zwischen 2016 und 2023 erschöpft sein wird (andererseits lagen frühere Schätzungen zum Wachstum des Internets weit hinter der Wirklichkeit zurück, das heißt die obige Schätzung ist nicht verläßlich). Zur Lösung dieses Problems wurde ein neues Adressierungsmodell empfohlen, das sich langsam aber sicher durchsetzt. Das traditionelle 32-Bit-Modell ist unter der Bezeichnung „IPv4“ bekannt, das neue Modell dagegen als „IPv6“. Java unterstützt beide Modelle durch die Klassen `java.net.Inet4Address` beziehungsweise `java.net.Inet6Address`. Wir verwenden in diesem Kapitel IPv4, da es gegenwärtig noch das häufigere der beiden Modelle ist.

7.2.2 Transmission Control Protocol und User Datagram Protocol

[10] Im Laufe der Jahre haben sich die Programmiersprachen vom manuellen Setzen binärer Anweisungen (1GL-Sprachen, Sprachen der ersten Generation) über Assemblersprachen (2GL-Sprachen, Sprachen der zweiten Generation) und menschenlesbare Sprachen wie Java oder C++ (3GL-Sprachen, Sprachen der dritten Generation) bis hin zu den Spezifikationssprachen wie SQL (4GL-Sprachen, Sprachen der vierten Generation) entwickelt. Die moderneren Sprachen machen die älteren, systemnäheren Sprachen nicht überflüssig, sondern übernehmen die anstrengende Aufgabe, die Anweisungen des Entwicklers in systemnähere Anweisungen zu übersetzen. Beispielsweise übersetzt der Java-Compiler Ihren Quelltext in sogenannten Bytecode (entspricht grob einem Assemblerprogramm) der später von der Laufzeitumgebung wiederum in prozessorspezifische binäre Anweisungen übersetzt wird. Sie können noch immer direkt in Maschinensprache programmieren, aber viele Entwickler empfinden diese Arbeit als langsam und fehleranfällig. Die Wahl einer höheren Programmiersprache wie Java wirkt sich erheblich auf die Produktivität des Entwicklers und die Qualität seiner Arbeit aus.

[11] Die Netzwerkprotokolle haben sich in analoger Weise entwickelt, so daß wir uns nicht mehr mit den systemnahen Einzelheiten der Netzwerkprogrammierung auseinandersetzen müssen. Beim Entwickeln eines Java-Programms brauchen Sie sich nicht darum zu kümmern, wie Signale physikalisch über ein Netzwerkkabel (oder ein anderes Medium) übertragen werden oder ob die Rechner in Ihrem Netzwerk per Ethernet oder Token Ring kommunizieren. Es genügt, wenn Sie wissen, wie Sie eine Verbindung zu einem anderen Rechner herstellen können. Wir verwenden bei der Socketprogrammierung das Internet Protocol anstelle eines der anderen Protokolle in der Vermittlungsebene (*network layer*), wie X.25, das Internet Control Message Protocol (ICMP) oder Internetwork Packet Exchange (IPX). Wenn Sie sich für Sockets als Netzwerkschnittstelle entschieden haben, die systemnahe Kommunikation über die Vermittlungsebene und die darunterliegenden Schichten nicht also mehr berücksichtigt zu werden braucht, müssen Sie noch zwischen dem TCP- und dem UDP-Protokoll wählen.

7.2.2.1 User Datagram Protocol (UDP)

[12] Das User Datagram Protocol (UDP) gestattet Rechnern in einem Netzwerk, sich gegenseitig Datagramm-Pakete zu senden. UPD funktioniert wie die gewöhnliche Post. Nachrichten (Briefe) werden an eine bestimmte Adresse gesendet, wobei keine unmittelbare Antwort des Servers (Empfänger) beziehungsweise Endpunktes erforderlich ist. UPD setzt keine bidirektionale Verbindung voraus. Ebenso ist bei der traditionellen Post nicht erforderlich, daß der Empfänger bei der Zustellung eines Briefes anwesend ist.

[13] Wir bleiben bei der Analogie mit der Post. Ein Brief wird am Briefkasten eingeworfen und zur angegebenen Adresse befördert. Der Briefträger wirft den Brief dort in den Briefkasten. Die Post verlangt nicht, daß jemand zu Hause ist, um den Brief entgegenzunehmen (natürlich gibt es Briefsendungen, deren Eingang mit einer Unterschrift bestätigt werden muß, aber wir lassen solche Sendungen in dieser Diskussion beiseite). Wenn der Empfänger nach Hause kommt, nimmt er den Brief aus dem Briefkasten.

[14] UDP verwirft fehlerhafte Nachrichten (Datagramme), garantiert also *nicht*, daß eine gesendete Nachricht ihr Ziel erreicht. Auch bei der Post besteht keine Garantie, daß ein aufgegebener Brief beim beabsichtigten Empfänger ankommt. Die beiden Hauptklassen der entsprechenden Java-API sind `java.net.DatagramSocket` und `java.net.DatagramPacket`. Ein „Paket“ ist eine Information in Form einer Folge von Bytes, die über ein Netzwerk hinweg versendet werden. Da die „Pakete“ bei UDP nicht garantiert ausgeliefert werden und eventuell nicht in der richtigen Reihenfolge eintreffen, muß sich der Entwickler bei UDP-Sockets mit der Sortierung und dem Verlust von Datagrammen auseinandersetzen.

[15] Die Hauptgründe, um sich für UDP zu entscheiden, sind Effizienz und Flexibilität. Wir haben uns aus den folgenden Gründen gegen UDP entschieden: Erstens hat Effizienz bei unserer Beispielanwendung keine so hohe Priorität. Zweitens (und noch wichtiger) ist UDP in den Prüfungsaufgaben nicht erlaubt. Damit steht unsere Entscheidung fest. Hätten wir uns aber für UDP entschieden, so würden sowohl der Client als auch der Server auf der Klasse `DatagramSocket` basieren und Objekte vom Typ `DatagramPacket` austauschen. Der Rest dieses Kapitels widmet sich TCP statt UDP (auch unsere Beispielanwendung basiert auf TCP), so daß wir die Diskussion von UDP hier beenden.

Warnung: Die Prüfungsanforderungen neigen in diesem Punkt zur Undeutlichkeit. Sehr wahrscheinlich enthält Ihre Anleitung den folgenden Satz: „You must use either serialized objects over a simple socket connection, or RMI.“ Genau genommen ist nicht klar, was ein „einfacher Socket“ ist und ob UDP ausgeschlossen ist. Wir interpretieren diesen Satz so, daß Sie die Klasse `java.net.Socket` statt `java.net.DatagramSocket` verwenden sollen. Es wäre zwar technisch machbar, UDP in der Beispielanwendung zu verwenden, aber eine solche Lösung würde hinsichtlich der Paketverwaltung viel Arbeit bedeuten und sich gegen die typische sachgerechte Verwendung von UDP richten (einfache, kurze Nachrichten, häufig weniger als 100 Byte). Wir empfehlen daher, daß Sie das einfachere TCP-Protokoll wählen, falls Sie sich für Sockets als Netzwerkschnittstelle entscheiden. Achten Sie dennoch genau auf die Anleitung zu Ihrer individuellen Prüfungsaufgabe.

---

[16] UDP wurde für Anwendungen mit minimalem Datenaustauschvolumen entwickelt, etwa für die sehr kleinen Nachrichten, die beim Überwachen von Anwendungen im Netzwerk anfallen, zum Beispiel beim Simple Network Management Protocol (SNMP), oder für sehr einfache Anfragen/Antworten, zum Beispiel beim Domain Name Service (DNS). Bei voluminöseren Nachrichten, etwa den Antworten auf Anfragen an die Datenbankdatei in unserer Beispielanwendung, eignet sich TCP besser. Der folgende Unterunterabschnitt beschreibt das TCP-Protokoll.

7.2.2.2 Transmission Control Protocol (TCP)

[17] Das Transmission Control Protocol (TCP) überwacht die Übertragung der Pakete, so daß Nachrichten garantiert zugestellt werden und die einzelnen Pakete in der Reihenfolge ankommen, in der sie gesendet werden. Um diese Garantien zu gewährleisten, verursacht TCP etwas mehr Datenverkehr über das Netzwerk als UDP (da RMI noch mehr Eigenschaften, Fähigkeiten und Garantien beinhaltet, beansprucht RMI wiederum mehr Netzwerkverkehr als TCP). Nachdem TCP bereits entsprechende Prüfungen beinhaltet, müssen wir die Reihenfolge der eingetroffenen Pakete nicht verifizieren. Ein gutes Argument, um sich bei der Beispielanwendung für TCP zu entscheiden.

[18] Ein TCP-Socket verwendet eine TCP/IP-Verbindung als Übertragungsprotokoll (*transfer protocol*). Beide Enden der Verbindung sind anhand einer IP-Adresse und einer Portadresse identifizierbar. TCP-Socketclients senden Anfragen, während TCP-Socketserver auf Anfragen warten („lauschen“ oder auch „horchen“). Die drei grundlegenden Schritte in der Kommunikation über einen TCP-Socket sind:

1. Erzeugen eines Objektes einer Socketklasse.
2. Senden von serialisierten Nachrichten über den Ein-/Ausgabestrom des Socketobjekts.
3. Schließen der Socketverbindung.

Bemerkung: Der Verbindungsaufbau setzt einen Client voraus, der eine Verbindungsanfrage sendet sowie einen Server, der auf eintreffende Verbindungsanfragen wartet. Ist die Verbindung einmal eingerichtet, so besteht sie bidirektional zwischen zwei gleichwertigen Sockets, das heißt jeder Socket kann sowohl Daten senden als auch empfangen. Das Kommunikationsprotokoll bestimmt, welcher Socket zu einem gegebenen Zeitpunkt Daten sendet beziehungsweise empfängt. Wir besprechen das Protokoll der Beispielanwendung im Unterabschnitt 7.3.4. Sie können sich vorstellen, daß die Punkt-zu-Punkt-Kommunikation scheitert, wenn der Client der der Server dem Protokoll nicht gehorcht.

7.2.3 Socketclients

[19] Eine Verbindung zwischen zwei Sockets wird mit Hilfe der Klasse `java.net.Socket` implementiert. Der Konstruktor der Klasse `Socket` versucht, die Verbindung zum Zielrechner aufzubauen und benötigt sowohl dessen Rechnernamen als auch die zu verwendende Portadresse in Form einer URL. Tabelle 7.1 zeigt die verschiedenen öffentlichen Konstruktoren der Klasse `Socket`.

[20] Nachdem eine Socketverbindung aufgebaut wurde, können mit Hilfe von Abfragemethoden eine Vielzahl von Verbindungseigenschaften ausgewertet werden. Der clientseitige Socket muß nicht dieselbe Portadresse verwenden wie der serverseitige Socket. Der clientseitige Socket erhält vielmehr eine lokale Portadresse, die er zur Kommunikation mit dem Server verwendet.

[21] Das Interface *`java.net.SocketOptions`* definiert eine Reihe von statischen `int`-Feldern (keine finalen Felder), die zur Konfiguration clientseitiger Sockets verwendet werden können. Wir implementieren das Interface *`SocketOptions`* nicht direkt, können die Felder aber durch Abfrage- und Änderungsmethoden der Klasse `Socket` erreichen.

7.2.3.1 Das Feld SO_TIMEOUT

[22] Beim Lesen von Daten aus einem Socket über die `read()`-Methode des Eingabestromobjektes blockiert der Aufruf der `read()`-Methode andere Anfragen entsprechend der durch das Feld

Konstruktor der Klasse Socket	Beschreibung
`public Socket()`	Erzeugt einen Socket ohne Verbindung.
`public Socket(InetAddress address, int port)`	Der am häufigsten verwendete Konstruktor: Erzeugt einen mit der übergebenen entfernten Adresse `address` und dem übergebenen entfernten Port `port` verbundenen Socket.
`public Socket(InetAddress address, int port, InetAddress localAddress, int localPort)`	Erzeugt einen mit der entfernten Adresse `address` und dem übergebenen entfernten Port `port` verbundenen Socket. Verwendet auf der lokalen Seite die Netzwerkkarte mit der IP-Adresse `localAddress` und den Port `localPort`.
`public Socket(String host, int port)`	Erzeugt einen mit dem entfernten Gerät `host` und dem entfernten Port `port` verbundenen Socket.
`public Socket(String host, int port, InetAddress localAddress, int localPort)`	Erzeugt einen mit dem entfernten Gerät `host` und dem entfernten Port `port` verbundenen Socket. Verwendet auf der lokalen Seite die Netzwerkkarte mit der IP-Adresse `localAddress` und den Port `localPort`.

Tabelle 7.1: Öffentliche Konstruktoren der Klasse `Socket`.

`SO_TIMEOUT` festgelegten Konfiguration. Das `SO_TIMEOUT`-Feld legt in Millisekunden fest, wie lange eine Operation den Socket blockieren darf. Falls die Operation nicht innerhalb der bewilligten Zeit beendet wird, wirft die Laufzeitumgebung eine Ausnahme vom Typ `java.net.SocketTimeoutException` aus:

```
~$ java sampleproject.sockets.DvdSocketServer

Aug 29, 2009 4:50:35 PM sampleproject.sockets.DvdSocketServer listenForConnections
INFO: a server socket created on port 3000
java.net.SocketTimeoutException: Accept timed out
  at java.net.PlainSocketImpl.socketAccept(Native Method)
  at java.net.PlainSocketImpl.accept(PlainSocketImpl.java:384)
  at java.net.ServerSocket.implAccept(ServerSocket.java:453)
  at java.net.ServerSocket.accept(ServerSocket.java:421)
  at sampleproject.sockets.DvdSocketServer.listenForConnections(DvdSocketServer.java:83)
  at sampleproject.sockets.DvdSocketServer.run(DvdSocketServer.java:60)

~$
```

Ist das `SO_TIMEOUT`-Feld nicht bewertet oder auf 0 gesetzt, so wird die Blockierung bis zum Abschluß der Operation aufrechterhalten und auch bei Zeitüberschreitung nicht abgebrochen.

[23] Die Signaturen der entsprechenden Abfrage- beziehungsweise Änderungsmethode aus der Klasse `Socket` lauten:

```
public void setSoTimeout(int timeout) throws SocketException
public int getSoTimeout() throws SocketException
```

[24] Wir bewerten das `SO_TIMEOUT`-Feld in unserer `DvdSocketServer`-Klasse. Die Bewertung wäre äquivalent über unseren Socketclient (`DvdSocketClient`) möglich gewesen. Die folgende Anweisung beendet den Socketserver nach einer Minute (60000 ms) Inaktivität:

```
serverSocket.setSoTimeout(60000);
```

Nach der festgelegten Zeit wird eine Ausnahme vom Typ `SocketTimeoutException` ausgeworfen (siehe oben).

7.2.3.2 Das Feld SO_SNDBUF

[25] Das Feld `SO_SNDBUF` legt die Puffergröße für die Datenübertragung von diesem Socket aus fest. Der Wert von `SO_SNDBUF` ist lediglich ein Vorschlag beziehungsweise ein Richtwert für das Betriebssystem, welche Puffergröße die Anwendung für die Ausgabeoperationen über diesen Socket benötigt. Die zugehörige Abfragemethode liefert die *tatsächliche* Größe des Ausgabepuffers. Sie können die Änderungs- und die Abfragemethode kombinieren, um nach der Ausgabe zu ermitteln, ob das Betriebssystem den vorgeschlagenen Richtwert übernommen hat. Die Signaturen der Abfrage- beziehungsweise Änderungsmethode aus der Klasse `Socket` lauten:

```
public int getSendBufferSize() throws SocketException
public void setSendBufferSize(int size) throws SocketException
```

[26] Die Größe des Ausgabepuffers gibt an, wieviele Pakete gesendet werden, bevor von der Empfängerseite eine Empfangsbestätigung über diese Pakete erwartet wird. In einem betriebssicheren (*reliable*) Netzwerk können Sie diesen Wert hoch ansetzen, um den bestmöglichen Datendurchsatz zu erhalten. Im schlimmsten Fall können Sie die Puffergröße auf die Größe eines Paketes setzen, so daß Sie für jedes gesendete Paket die Empfangsbestätigung abwarten können, das heißt wenn Sie 100 Pakete senden, werden 100 Bestätigungspakete zurückgesendet. Vergleichen Sie diese Konfiguration mit einem Puffer, der einhundertmal so groß ist, wie die Paketgröße. Wenn Sie nun 100 Pakete senden, erhalten Sie nur eine Empfangsbestätigung, das heißt die Übertragung beansprucht nur die halbe Zeit.

7.2.3.3 Das Feld SO_RCVBUF

[27] Das Feld `SO_RCVBUF` wirkt ähnlich wie `SO_RCVBUF`, stellt aber die Puffergröße für eingehende Pakete ein. Wie beim Ausgabepuffer ist auch `SO_RCVBUF` nur ein unverbindlicher Richtwert für das Betriebssystem. Wie zuvor können Sie die *tatsächlich* verwendete Puffergröße per `getReceiveBufferSize()`-Methode abfragen. Sie erhöhen die Performanz einer Socketverbindung mit hohem Übertragungsvolumen, indem Sie den Eingabepuffer vergrößern. Durch Verringern der Puffergröße reduzieren Sie den Nachholbedarf durch verlorene Pakete. Die Signaturen der Abfrage- beziehungsweise Änderungsmethode aus der Klasse `Socket` lauten:

```
public void setReceiveBufferSize(int size) throws SocketException
public int getReceiveBufferSize() throws SocketException
```

7.2.3.4 Das Feld SO_BINDADDR

[28] Das Feld `SO_BINDADDR` gibt die *lokale* IP-Adresse an, an die der Socket gebunden ist und ist nur lesbar. Die lokale IP-Adresse eines Sockets kann nach dessen Erzeugung nicht mehr geändert werden und muß dem Konstruktor übergeben werden. Die Signatur der Abfragemethode aus der Klasse `Socket` lautet:

```
public InetAddress getLocalAddress()
```

[29] Sie können diese Methode gebrauchen, um auf einem Rechner mit mehr als einer IP-Adresse (also einem Server mit mehr als einer Netzwerkkarte, einem PC mit einer lokalen Adresse und einer Netzwerkadresse oder einem Rechner der sowohl eine Netzwerkadresse besitzt, als auch einem virtuellen privaten Netzwerk angehört) zu ermitteln, an welche IP-Adresse der Socket gebunden wurde.

7.2.4 Die Klasse DvdSocketClient

[30] Einige wichtige Hinweise zur Klasse `sampleproject.sockets.DvdSocketClient` aus unserer Beispielanwendung: Bei `DvdSocketClient` sind die Portadresse `3000` sowie der Rechnername `localhost` voreingestellt. Das ist praktisch, wenn wir unsere Anwendung auf nur einem Rechner aber dennoch im Netzwerkmodus betreiben möchten (also keinen Zugang zu einem Netzwerk haben). Wenn wir unsere Anwendung auf einem anderen Rechner starten wollen, müssen wir die IP-Adresse des Rechners angeben, auf dem der Socketserver läuft.

Tipp: Bei Linux- und Unixrechnern können Sie auch `localhost.localdomain` als Rechnernamen angeben. Insbesondere können Sie die Loopbackadressen (127.0.0.0–127.0.0.254) anstelle des Rechnernamens verwenden. In der Regel wird die Standard-Loopbackadresse 127.0.0.1 gewählt und die übrigen Loopbackadressen für besondere Zwecke reserviert (etwa das gleichzeitige Testen mehrerer identischer Anwendungen über verschiedene Adressen).

---

[31] Beachten Sie ebenfalls, daß die Klasse `DvdSocketClient` das Interface *`sampleproject.db.DBClient`* implementiert. Damit stehen die Verleih- und Rückgabemethoden `rent()` beziehungsweise `returnRental()` jedem Client vom Typ `DvdSocketClient` zur Verfügung. Der Quelltext *DvdSocketClient.java* der Klasse `DvdSocketClient` lautet:

```
public class DvdSocketClient implements DBClient {

    /**
     * The socket client that gets instantiated for the socket connection.
     */
    private Socket socket = null;

    /**
     * The outputstream used to write a serialized object to a socket server.
     */
    private ObjectOutputStream oos = null;

    /**
     * The inputstream used to read a serialized object (a response)
     * from the socket server.
     */
    private ObjectInputStream ois = null;

    /**
     * The ip address of the machine the client is going to attempt a
     * connection.
     */
    private String ip = null;

    /**
     * The port number we will be connecting on.
     */
    private int port = 3000;

    /**
     * Default constructor.
     *
     * @throws UnknownHostException if unable to connect to ''localhost''.
     * @throws IOException on network error.
     * @throws NumberFormatException if portNumber is not valid (never happens).
     */
    public DvdSocketClient()
```

```
        throws UnknownHostException, IOException, NumberFormatException {
    this("localhost", "3000");
}

/**
 * Constructor takes in a hostname of the server to connect.
 *
 * @param hostname The hostname to connect to.
 * @param portNumber the string representation of the port to connect on.
 * @throws UnknownHostException if unable to connect to "localhost".
 * @throws IOException on network error.
 * @throws NumberFormatException if portNumber is not valid.
 */
public DvdSocketClient(String hostname, String portNumber)
        throws UnknownHostException, IOException, NumberFormatException {
    ip = hostname;
    this.port = Integer.parseInt(portNumber);
    this.initialize();
}

/**
 * Adds a dvd to the database or inventory.
 *
 * @param dvd The DVD item to add to inventory.
 * @return A boolean value that indicates the success/failure of the
 * add operation.
 * @throws IOException Indicates there is a problem accessing the data.
 */
public boolean addDVD(DVD dvd) throws IOException {
    DvdCommand cmdObj = new DvdCommand(SocketCommand.ADD, dvd);
    return getResultFor(cmdObj).getBoolean();
}

/**
 * Gets a <code>DVD</code> from the system using a upc.
 *
 * @param upc The upc of the DVD you want to view.
 * @return A DVD that matches the supplied upc.
 *
 * @throws IOException Indicates there is a problem accessing the data.
 */
public DVD getDVD(String upc) throws IOException {
    DVD dvd = new DVD();
    dvd.setUPC(upc);

    DvdCommand cmdObj = new DvdCommand(SocketCommand.GET_DVD, dvd);
    return getResultFor(cmdObj).getDVD();
}

/**
 * Attempts to rent the DVD matching the provided UPC.
 *
 * @param upc is the upc of the DVD you want to rent.
 * @return true if the DVD was rented. false if it cannot be rented.
 *
 * @throws IOException Thrown if an <code>IOException</code> is
 * encountered in the <code>db</code> class.
 * <br>
 * For more information, see {@link DvdDatabase}.
 */
```

```
public boolean rent(String upc) throws IOException {
    DVD dvd = new DVD();
    dvd.setUPC(upc);

    DvdCommand cmdObj = new DvdCommand(SocketCommand.RENT, dvd);
    return getResultFor(cmdObj).getBoolean();
}

/**
 * Attempts to return the DVD matching the provided UPC.
 *
 * @param upc The upc of the DVD you want to rent.
 * @return true if the DVD was rented. false if it cannot be rented.
 *
 * @throws IOException Thrown if an <code>IOException</code> is
 * encountered in the <code>db</code> class.
 * <br>
 * For more information, see {@link DvdDatabase}.
 * <br>
 * For more information, see {@link DvdDatabase}.
 */
public boolean returnRental(String upc) throws IOException {
    DVD dvd = new DVD();
    dvd.setUPC(upc);

    DvdCommand cmdObj = new DvdCommand(SocketCommand.RETURN, dvd);
    return getResultFor(cmdObj).getBoolean();
}

/**
 * Gets the store's inventory.
 * All of the DVDs in the system.
 *
 * @return A collection of all found DVD's.
 * @throws IOException Indicates there is a problem accessing the data.
 */
public List<DVD> getDVDs() throws IOException {
    DvdCommand cmdObj = new DvdCommand(SocketCommand.GET_DVDS);
    return getResultFor(cmdObj).getList();
}

/**
 * A properly formatted <code>String</code> expressions returns all matching
 * DVD items. The <code>String</code> must be formatted as a regular
 * expression.
 *
 * @param query A regular expression searcg string.
 * @return A <code>Collection</code> of <code>DVD</code> objects that match
 * the search criteria.
 * @throws IOException Thrown if an <code>IOException</code> is
 * encountered in the <code>db</code> class.
 * @throws PatternSyntaxException if requested query is not a valid regular
 * expression.
 */
public Collection<DVD> findDVD(String query)
        throws IOException, PatternSyntaxException {
    DvdCommand cmdObj = new DvdCommand(SocketCommand.FIND);
    cmdObj.setRegex(query);

    DvdResult serialReturn = getResultFor(cmdObj);
```

```
    if (serialReturn.isException()
        && serialReturn.getException() instanceof PatternSyntaxException) {
        throw (PatternSyntaxException) serialReturn.getException();
    } else {
        return serialReturn.getCollection();
    }
}

/**
 * Removes a <code>DVD</code> from the system using a upc.
 *
 * @param upc The upc of the DVD you want to remove from the database.
 * @return true if the item was removed, false if it was not removed.
 * @throws IOException Indicates there is a problem accessing the data.
 */
public boolean removeDVD(String upc) throws IOException {
    DVD dvd = new DVD();
    dvd.setUPC(upc);

    DvdCommand cmdObj = new DvdCommand(SocketCommand.REMOVE, dvd);
    return getResultFor(cmdObj).getBoolean();
}

/**
 * Modifies a DVD database entry specified by a DVD object.
 *
 * @param dvd The DVD to modify.
 * @return A boolean indicating the success or failure of the modify
 * operation.
 * @throws IOException Thrown if an <code>IOException</code> is
 * encountered in the <code>db</code> class.
 * <br>
 * For more information, see {@link DvdDatabase}.
 */
public boolean modifyDVD(DVD dvd) throws IOException {
    DvdCommand cmdObj = new DvdCommand(SocketCommand.MODIFY, dvd);
    return getResultFor(cmdObj).getBoolean();
}

/**
 * Lock the requested DVD. This method blocks until the lock succeeds,
 * or for a maximum of 5 seconds, whichever comes first.
 *
 * @param upc The UPC of the DVD to reserve
 * @throws InterruptedException Indicates the thread is interrupted.
 * @throws IOException on any network problem
 * @return true if DVD reserved.
 */
public boolean reserveDVD(String upc)
        throws IOException, InterruptedException {
    DVD dvd = new DVD();
    dvd.setUPC(upc);
    DvdCommand cmdObj = new DvdCommand(SocketCommand.RESERVE, dvd);
    return getResultFor(cmdObj).getBoolean();
}

/**
 * Unlock the requested record. Ignored if the caller does not have
 * a current lock on the requested record.
 *
```

```
     * @param upc The UPC of the DVD to release
     * @throws IOException on any network problem
     */
    public void releaseDVD(String upc) throws IOException {
        DVD dvd = new DVD();
        dvd.setUPC(upc);
        DvdCommand cmdObj = new DvdCommand(SocketCommand.RELEASE, dvd);
        getResultFor(cmdObj).getBoolean();
    }

    /**
     * Method that does the work of sending our request to the client and
     * getting the response back, doing any necessary conversions between
     * a DvdCommand object, the Serialized Objects sent and received over
     * the Socket, and the DvdResult needed.
     *
     * @param command the command to be performed on the remote database.
     * @return a value object containing the result of the command requested.
     * @throws IOException on network error.
     */
    private DvdResult getResultFor(DvdCommand command) throws IOException {
//        this.initialize();
        try {
            oos.writeObject(command);
            DvdResult result = (DvdResult) ois.readObject();
            Exception e = result.getException();

            if (!result.isException()) {
                return result;
            } else if (e instanceof ClassNotFoundException) {
                throw (ClassNotFoundException) e;
            } else if (e instanceof IOException) {
                throw (IOException) e;
            } else {
                // well, we still have an exception, but it is up to the
                // calling method to handle it
                return result;
            }
        } catch (ClassNotFoundException cnfe) {
            IOException ioe
                    = new IOException("problem with demarshalling DvdResult)");
            ioe.initCause(cnfe);
            throw ioe;
//        } finally {
//            closeConnections();
        }
    }

    /**
     * Performs any clean-up necessary when this connection is no longer used.
     * E.g. closing any open connections.
     *
     * @throws IOException on network error.
     */
    public void finalize() throws java.io.IOException {
        closeConnections();
    }

    /**
```

```
         * A helper method which initializes a socket connection on specified port.
         *
         * @throws UnknownHostException if the IP address of the host could not be
         *         determined.
         * @throws IOException Thrown if the socket channel cannot be opened.
         */
        private void initialize() throws UnknownHostException, IOException {
            socket = new Socket(ip, port);

            oos = new ObjectOutputStream(socket.getOutputStream());
            ois = new ObjectInputStream(socket.getInputStream());
        }

        /**
         * A helper method which closes the socket connection.
         * Needs to be called from within a try-catch
         *
         * @throws IOException Thrown if the close operation fails.
         */
        private void closeConnections() throws IOException {
            oos.close();
            ois.close();
            socket.close();
        }
    }
```

Sie können den gesamten Quelltext der Beispielanwendung aus dem „Source Code“-Abschnitt der Apress-Website (*http://www.apress.com/book/sourcecode*) herunterladen. Einzelheiten über den Umfang der Distribution, das Übersetzen und die Aufteilung des Quelltextes in Packages finden Sie in Kapitel 9.

7.3 Socketserver

[32] In diesem Abschnitt diskutieren wir die beiden Typen von Socketservern (Multicast- und Unicastserver) und die Möglichkeiten how to build them. Außerdem besprechen wir das Konzept des Kommunikationsprotokolls und die Implementierung des Protokolls für unsere Beispielanwendung.

7.3.1 Multicast- und Unicast-Server

[33] Ein Unicastserver gestattet nur 1:1-Kommunikation zwischen Client und Server, das heißt der Server akzeptiert höchstens einen Client. Der Socketserver in der Beispielanwendung ein Unicastserver. Das Gegenteil von Unicastservern sind Multicastserver.

[34] Ein Multicastserver erlaubt Verbindungen zu mehr als einem Client. Multicasting (auch Broadcasting) wird verwendet, um Daten von einem Server aus an viele Clients zu senden. Die Klasse `java.net.MulticastSocket` wird verwendet, um Programme zu schreiben, die Nachrichten an viele Clients versenden. Wir belassen die Diskussion der Multicastserver bei diesem kurzen Absatz. Sie sollten allerdings das Prinzip verstanden haben.

7.3.2 Multitasking

[35] Ein sequentieller (iterativer, nur single-threadfähiger) Server verarbeitet eine Clientanfrage nach der anderen. Trifft eine Anfrage ein, so wird sie vom Server verarbeitet und beantwortet

woraufhin der Server auf die nächste Anfrage wartet. Geht eine Anfrage ein, während der Server eine frühere Anfrage verarbeitet, so wird die neu eingetroffene Anfrage in eine Warteschlange gesetzt und kommt an die Reihe, wenn alle ihr vorausgegangenen Anfragen verarbeitet worden sind. Solange ein Client mit einem sequentiellen Server verbunden ist, kann kein anderer Client eine Verbindung aufbauen. Wenn die bestehende Verbindung genug Zeit in Anspruch nimmt, kann ein wartender Client durch Zeitüberschreitung leer ausgehen (siehe `SO_TIMEOUT`-Feld, Seite 187). Der Lebenszyklus eines sequentiellen Socketservers verläuft wie folgt:

1. Das `ServerSocket`-Objekt (Server) wird erzeugt.
2. Der Server wartet auf eingehende Verbindungsanfragen.
3. Der Server ruft die Methoden `getInput()` beziehungsweise `getOutput()` auf.
4. Server und Client kommunizieren dem vereinbarten Protokoll entsprechend.
5. Server oder Client beendet die Verbindung. Weiter bei 2.

[36] Multitaskingfähige Socketserver können mehrere Clientanfragen gleichzeitig verarbeiten. Wenn Sie viele oder zeitaufwendige Anfragen erwarten, ist Multithreadfähigkeit eine nützliche Eigenschaft für Ihren Socketserver. Der Socketserver der Beispielanwendung ist multithreadfähig.

[37] Jede Clientanfrage wird mit einem eigenen Thread verarbeitet, wodurch sich der Socketserver ähnlich wie unser RMI-Fabrikobjekt verhält. We can thus be assured that each client has its own DvdDatabase object. (Siehe Diskussion des RMI-Fabrikobjektes in Kapitel 6, Seite 166ff.) Der nächste Unterabschnitt beschreibt die Details unseres multitaskingfähigen Socketservers.

7.3.3 Die Klasse ServerSocket

[38] Die Klasse `java.net.ServerSocket` ermöglicht das Erzeugen von Socketserverobjekten, die an einem bestimmten Port auf eintreffende Verbindungsanfragen warten. Ein `ServerSocket`-Objekt kann Daten senden, empfangen und verarbeiten. Die öffentlichen Konstruktoren der Klasse `ServerSocket` erwarten eine Portadresse, die Länge der Warteschlange (*backlog*) und eine lokale IP-Adresse. Der Socketserver implementiert das Kommunikationsprotokoll, also die Regeln, denen Client und Server während der Kommunikation gehorchen müssen. Ist der Port, den Sie beim Erzeugen eines `ServerSocket`-Objektes angeben besetzt, so wird eine Ausnahme vom Typ `java.io.IOException` ausgeworfen. Wurde das `ServerSocket`-Objekt dagegen erfolgreich erzeugt, so wartet der Socketserver nach dem Aufrufen seiner `accept()`-Methode an diesem Port auf eintreffende Verbindungsanfragen. Die `ServerSocket`-Methode `accept()` gibt eine Referenz auf ein `Socket`-Objekt zurück, das die Serverseite der Socketverbindung repräsentiert. Der Lebenszyklus eines multithreadfähigen Socketservers verläuft wie folgt:

1. Das `ServerSocket`-Objekt (Server) wird erzeugt.
2. Der Server wartet auf eingehende Verbindungsanfragen.
3. Der Server erzeugt einen neuen Thread, um die Clientanfrage zu verarbeiten,
4. Der Server ruft die Methoden `getInput()` beziehungsweise `getOutput()` auf.
5. Server und Client kommunizieren dem vereinbarten Protokoll entsprechend.
6. Server oder Client beendet die Verbindung. Weiter bei 2.

Bemerkung: Genau genommen ist die Ausnahme, die der `ServerSocket`-Konstruktor auswirft,

wenn der gewählte Port besetzt ist, vom Typ `java.net.BindException`. Der `ServerSocket`-Konstruktor deklariert aber den Basistyp `IOException`, da während der Objekterzeugung auch andere Ausnahmen ausgeworfen werden können.

---

[39] Danach öffnet der Server die Ein- und Ausgabeströme des Sockets (`ObjectInputStream` und `ObjectOutputStream`, siehe Hilfsklasse `sampleproject.sockets.DbSocketRequest`) und kann nun durch Lesen aus beziehungsweise Schreiben in den entsprechenden Datenstrom des Sockets mit dem Client kommunizieren. Die `ServerSocket`-Methode `accept()` blockiert den Port solange, bis sich ein Client verbindet und gibt anschließend eine Referenz auf ein `Socket`-Objekt zurück, das den serverseitigen Socket repräsentiert. Die Socketverbindung wird verwendet, um die Clientanfrage zu verarbeiten und kann danach geschlossen werden.

[40] Die Klassen `ServerSocket` und `Socket` haben viele Eigenschaften gemeinsam, beispielsweise das `SO_TIMEOUT`-Feld. Bei `ServerSocket` gibt `SO_TIMEOUT` an, wie lange die `accept()`-Methode des Socketservers auf Verbindungsanfragen warten soll.

[41] Wenn die `accept()`-Methode ihre zulässige Wartezeit überschreitet, wird eine Ausnahme vom Typ `java.net.SocketTimeoutException` ausgeworfen. Abbildung 7-4, Seite 215 (Buch), veranschaulicht die socketbasierte Netzwerkschnittstelle der Beispielanwendung. Der Quelltext der Klasse `DvdSocketServer` lautet:

```
package sampleproject.sockets;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.logging.Level;
import java.util.logging.Logger;
import sampleproject.db.*;

/**
 * DvdSocketServer is the class that handles socket client requests and
 * passes the request to the database. The class recieves parameters in
 * <code>DVDCommand</code> objects and returns results in
 * <code>DVDResult</code> objects.
 *
 * @author Denny's DVDs
 * @version 2.0
 */
public class DvdSocketServer extends Thread {
    private String dbLocation = null;
    private int port = 3000;

    /**
     * Starts the socket server.
     *
     * @param argv Command line arguments.
     * @throws IOException thrown if the socket server fails to start.
     */
    public static void main(String[] argv) {
        register(".", 3000);
    }

    public static void register(String dbLocation, int port) {
        new DvdSocketServer(dbLocation, port).start();
    }

    public DvdSocketServer(String dbLocation, int port) {
```

```
            this.dbLocation = dbLocation;
            this.port = port;
        }

        public void run() {
            try {
                listenForConnections();
            } catch (IOException ioe) {
                ioe.printStackTrace();
                System.exit(-1);
            }
        }

        public void listenForConnections() throws IOException {
            ServerSocket aServerSocket = new ServerSocket(port);
            //block for 60,000 msecs or 1 minute
            aServerSocket.setSoTimeout(60000);

            (Logger.getLogger("sampleproject.sockets")).log(Level.INFO,
                "a server socket created on port "
                + aServerSocket.getLocalPort());

            while (true) {
                Socket aSocket = aServerSocket.accept();
                DbSocketRequest request = new DbSocketRequest(dbLocation, aSocket);
                request.start();
            }
        }
    }
```

[42] Die Methode `listenForConnections()` erzeugt ein mit der Portadresse `3000` verknüpftes `ServerSocket`-Objekt. Die `while`-Schleife wartet auf Verbindungsanfragen an diesem Port, die von Java-Clients aber auch von Clients gesendet werden können, die in einer anderen Programmiersprache geschrieben sind. Trifft eine Verbindungsanfrage ein, so wird sie akzeptiert, das heißt die `accept()`-Methode hebt die Blockierung des Ports auf und gibt eine Referenz auf ein `Socket`-Objekt zurück, das den serverseitigen Socket repräsentiert. Diese `Socket`-Referenz wird dem `DbSocketRequest`-Konstruktor übergeben. Der Socketserver ruft die `start()`-Methode des zuvor erzeugten `DbSocketRequest`-Objektes auf, um die Clientanfrage in einem neuen Thread zu verarbeiten. Dadurch können sich mehrere Clients mit dem Socketserver verbinden, da jede Anfrage in Form eines separaten Threads verarbeitet wird.

[43] Die Klasse `DbSocketRequest` ist von `Thread` abgeleitet (siehe unten). Denken Sie daran, daß mehrere Clients die Methoden des `DvdDatabase`-Objektes aufrufen können.

[44] Der Socketserver (`DvdSocketServer`-Objekt) kann bei Bedarf mehrere Threads erzeugen. Nachdem die `run()`-Methode über Port `3000` ein serialisiertes Kommandoobjekt (`DvdCommand`) empfangen hat, wird die Methode `execCmdObject()` aufgerufen, im Prinzip eine große `switch`-Anweisung, die durch Auswerten des Kommandoobjektes feststellt, welches Kommando bezüglich der Datenbankdatei ausgeführt werden soll und die entsprechende Methode des `DvdDatabase`-Objektes aufruft. Wir zeigen nur die `run()`-Methode, um Platz zu sparen:

```
    /**
     * Required for a class that extends thread, this is the main path
     * of execution for this thread.
     */
    public void run() {

        try {
            ObjectOutputStream out = new ObjectOutputStream(client.getOutputStream());
```

```
            ObjectInputStream  in = new ObjectInputStream(client.getInputStream());

            DvdCommand cmdObj = (DvdCommand) in.readObject();
            out.writeObject(execCmdObject(cmdObj));
            if (client != null) {
                client.close();
            }
            out.flush();
        } catch (SocketException e) {
            logger.log(Level.SEVERE,
                "SocketException in Socket Server: " + e.getMessage());
        } catch (Exception e) {
            logger.log(Level.SEVERE,
                "General Exception in Socket Server: " + e.getMessage());
        }
    }
```

Sie finden den vollständigen Quelltext der Klasse `DbSocketRequest` in der Distribution der Beispielanwendung, die Sie aus dem „Source Code"-Abschnitt der Apress-Website (*http://www.apress-.com/book/sourcecode*) herunterladen können.

[45] Eine abschließende Bemerkung zur Klasse `DbSocketRequest`: Das Kommunikationsprotokoll ist in `DbSocketRequest` implementiert, also vom eigentlichen Socket (`DvdSocketServer`) getrennt.

7.3.4 Das Kommunikationsprotokoll

[46] Der Socketclient überträgt serialisierte Objekte zum Socketserver. Aber woher „weiß" der Server, wie ein solches Objekt verarbeitet werden soll? Ein serialisiertes Objekt ist technisch betrachtet lediglich ein Bytestrom. An dieser Stelle kommt das Kommunikationsprotokoll ins Spiel, Regeln, die definieren, wie Client und Server miteinander kommunizieren. Das Kommunikationsprotokoll der Beispielanwendung funktioniert in groben Zügen wie folgt:

1. Der Client sendet eine Anfrage zum Server, zum Beispiel um eine DVD auszuleihen oder eine ausgeliehene DVD zurückzugeben.
2. Der Server verarbeitet die Anfrage.
3. Der Status der abgeschlossenen Verarbeitung oder ein Rückgabewert wird an den Client zurückgesendet.

[47] Beim Nachdenken über die obige Liste sollten sich zwei Fragen herauskristallisieren: „Wie interpretiert der Server die Anfrage?" und „Wie wird das Ergebnis der Anfrage an den Client zurückgesendet?"

[48] Wir haben für die Beispielanwendung einen Kapselungsansatz gewählt. Die Anfrage befindet sich gekapselt in einem Kommandoobjekt vom Typ `DvdCommand`. Die Antwort wird gekapselt in einem Ergebnisobjekt vom Typ `DvdResult` zurückgesendet. Wir besprechen die Kommando- und Ergebnisobjekte nun im Detail.

7.3.4.1 Kommandoobjekte: Die Klasse DvdCommand

[49] Die Klasse `sampleproject.sockets.DvdCommand` kapselt eine Anfrage in einem Feld namens `commandId` vom Typ `SocketCommand`. Ruft ein GUI-Client eine der *`DBClient`*-Methoden des `DvdSocketClient`-Objektes auf, so erzeugt der Socketclient ein neues Kommandoobjekt (`DvdCommand`-Objekt), bewertet über den Konstruktoraufruf das `commandId`-Feld und überträgt das Kommando-

objekt zum Socketserver (`DvdSocketServer`-Objekt). Da die Kommandoobjekte über das Netzwerk gesendet werden, muß die Klasse `DvdCommand` serialisierbar sein. Empfängt der Server ein Kommandoobjekt, so wertet er das `commandId`-Feld aus, um die entsprechende Methode des serverseitigen `DvdDatabase`-Objektes aufrufen (ein bezüglich des Servers lokaler Methodenaufruf). Sämtliche für die Anfrage eventuell benötigten Parameter (beispielsweise der UPC) werden in dem vom Feld `dvd` der Klasse `DvdCommand` referenzierten `DVD`-Objekt transportiert. Das `regex`-Feld der Klasse `DvdCommand` wird ausschließlich von der `findDVD()`-Methode ausgewertet.

Bemerkung: Die Klasse `DvdCommand` ist ein Beispiel für das *Command*-Entwurfsmuster (*Kommando*). Ein Kommandoobjekt kapselt eine Anfrage. Weiterführende Informationen über das *Command*-Entwurfsmuster finden Sie im Buch von Gamma E., Helm R., Johnson R. E. and Vlissides J.: *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995). Auch die Entwurfsmuster *Proxy* (*Stellvertreter*) und *Adapter* sind in diesem Buch beschrieben.

---

[50] Zwei von drei Konstruktoren der Klasse `DvdCommand` erwarten ein Argument vom Typ `SocketCommand` (siehe Kasten „Aufzählungstypen“, Seite 201f). Der dritte Konstruktor erwartet zusätzlich ein Argument vom Typ `DVD`. Das `DVD`-Objekt enthält den zum Ausleihen beziehungsweise zur Rückgabe erforderlichen UPC. Der `DVD`-Parameter der `modifyDVD()`-Methode gibt die neuen Inhalte der zu ändernden DVD-Felder an. We do not actually use the modifyDVD method publicly in our implementation, but the class has been designed with this enhancement in mind. Die öffentlichen Methoden der Klasse `DvdCommand` sind:

```
/**
 * Default constructor.
 */
public DvdCommand() {
    this(SocketCommand.UNSPECIFIED);
}

/**
 * Constructor that requires the type of command to execute as a parameter.
 *
 * @param command The id of the command the server is to perform.
 */
public DvdCommand(SocketCommand command) {
    this (command, new DVD());
}

/**
 * Constructor that requires the type of command and the DVD object.
 *
 * @param command The id of the command the server is to perform.
 * @param dvd the DVD object that the command will process.
 */
public DvdCommand(SocketCommand command, DVD dvd) {
    setCommandId(command);
    this.dvd = dvd;
}

/**
 * Gets the query that was used for searching.
 *
 * @return The string representing the regular expression to use in find().
 */
public String getRegex() {
    return regex;
```

```
}

/**
 * Sets the regular expression.
 *
 * @param re The regular expression to use in find().
 */
public void setRegex(String re) {
    regex = re;
}
```

[51] Die Klasse `DvdSocketClient` verwendet bei allen *`DBClient`*-Methoden Kommandoobjekte. Beispielsweise wird das Kommandoobjekt für die `findDVD()`-Methode mit dem Konstantenwert `FIND` des Aufzählungstyps `SocketCommand` erzeugt. Anschließend wird das `regex`-Feld mit dem Parameter der `findDVD()`-Methode bewertet:

```
DvdCommand cmdObj = new DvdCommand(SocketCommand.FIND);
cmdObj.setRegex(query);
```

7.3.4.2 Ergebnisobjekte: Die Klasse DvdResult

[52] Die Klasse `DvdResult` kapselt das Ergebnis einer Anfrage. Nachdem das serverseitige `DvdDatabase`-Objekt eine Anfrage verarbeitet hat, wird das Ergebnis zunächst an das `DvdSocketServer`-Objekt zurückgesendet, welches das Ergebnis wiederum in einem `DvdResult`-Objekt verpackt und an den Socketclient zurückschickt. Ebenso wie `DvdCommand` muß auch die Klasse `DvdResult` serialisierbar sein, damit Ergebnisobjekte über das Netzwerk gesendet werden können. Auch Ausnahmen werden im `DvdResult`-Objekt verpackt und an den Client zurückgesendet.

[53] Die Funktionsweise der Ergebnisobjekte ähnelt der Funktionsweise der Kommandoobjekte. Die Klasse `DvdResult` verfügt über fünf Konstruktoren, welche die verschiedenen Typen von Rückgabewerten der *`DBClient`*-Methoden erwarten. Sie finden den Quelltext der Klasse `DvdResult` in der Distribution der Beispielanwendung, die Sie aus dem „Source Code“-Abschnitt der Apress-Website (*http://www.apress.com/book/sourcecode*) herunterladen können. Die folgenden Zeilen zeigen die Anwendung eines Ergebnisobjektes in der `findDVD()`-Methode der Klasse `DvdSocketClient`:

```
DvdResults serialReturn = (DvdResult) ois.readObject();
if (!serialReturn.isException()) {
    retVal = serialReturn.getCollection();
}
```

[54] Das Ergebnisobjekt wurde vom Socketserver gesendet und auf Ausnahmen geprüft. Falls das Ergebnisobjekt keine Ausnahmen beinhaltet, wird die enthaltene Kollektion extrahiert. Wir wissen, daß es sich um eine Kollektion handelt, weil die *`DBClient`*-Methode `findDVD()` diesen Rückgabetyp deklariert.

Bemerkung: Wie bei einem guten Entwurf zu erwarten, ist das Kommunikationsprotokoll vom Socketserver getrennt. Änderungen am Protokoll und sogar weitere Protokollschritte können vorgenommen beziehungsweise aufgenommen werden, ohne den Socketserver direkt zu beeinträchtigen.

Aufzählungstypen

Das Kommunikationsprotokoll verlangt unter anderem die Angabe des gewünschten Kommandos. Diese erfolgt beim Aufruf des Konstruktors der Klasse `DvdCommand`:

```
public DvdCommand(SocketCommand command, DVD dvd) {

    setCommandId(command);
    this.dvd = dvd;

}
```

Vor Version 5 des Java Development Kits wäre das `commandId`-Feld in der Klasse `DvdCommand` wahrscheinlich mit einer Konstanten wie diesen bewertet worden:

```
public final static int FIND = 0;
public final static int RENT = 1;
public final static int RETURN = 2;
```

In dieser veralteten Weise, Konstanten zu definieren, lauern einige Gefahren, beispielsweise könnte ein Entwickler das `commandId`-Feld direkt mit einem ganzzahligen Wert belegen, der nicht zum Protokoll gehört oder etwas unlogisches mit den Konstanten anstellen (zum Beispiel die Werte addieren). Außerdem können die einzelnen Kommandos nicht aufgezählt werden und beim Abfragen des `commandId`-Feldes wird eine Zahl zurückgegeben, deren Bedeutung in der Dokumentation oder im Quelltext gesucht werden muß.

Seit Version 5 des Java Development Kits verfügt Java über eine bessere Möglichkeit, um Konstanten zu definieren, nämlich die sogenannten Aufzählungstypen (*enumerated types*). Der Typ `SocketCommand` ist folgendermaßen definiert:

```
package sampleproject.gui;

public enum SocketCommand {

    UNSPECIFIED,   /* indicates that the command object has not been set. */
    FIND,          /* request will be performing a Find action.           */
    RENT,          /* renting a DVD.                                      */
    RETURN,        /* returning a DVD.                                    */
    MODIFY,        /* updating status of a DVD.                           */
    ADD,           /* creating a new DVD record.                          */
    REMOVE,        /* delete a DVD record.                                */
    GET_DVD,       /* retrieve a single DVD from database.                */
    GET_DVDS,      /* retrieve multiple DVDs from database.               */
    RESERVE,       /* Reserve a DVD.                                      */
    RELEASE        /* Release a DVD.                                      */

}
```

Ein Feld oder eine lokale Variable vom Typ `SocketCommand` kann ausschließlich diese festgelegten Werte haben, andere Werte sind nicht möglich.

Wenn wir den Wert eines Feldes oder einer Variablen vom Typ `SocketCommand` ausgeben oder protokollieren wollen, lautet die Ausgabe `UNSPECIFIED`, `FIND`, `RENT`, `RETURN`, `MODIFY`, `ADD`, `REMOVE`, `GET_DVD`, `GET_DVDS`, `RESERVE` oder `RELEASE`. Damit ist aus der Ausgabe unmittelbar ersichtlich, welchen Inhalt das Feld beziehungsweise die lokale Variable gehabt hat.

Aufzählungstypen haben noch viele andere Vorteile. Wir empfehlen die Freigabevermerke (*release notes*) zu den Aufzählungstypen unter der Internetadresse *http://java.sun.com/j2se/1.5.0/docs/guide/language/enums.html*.

7.4 Zusammenfassung

[55] In diesem Kapitel haben wir die socketbasierte Netzwerkschnittstelle unserer Beispielanwendung *Denny's DVDs* diskutiert. Wir haben einen kurzen Überblick über die verschiedenen Typen von Sockets gegeben (UDP und TCP) and the various types of TCP socket development strategies. Wir haben unser Kommunikationsprotokoll besprochen und mit Hilfe eines Aufzählungstyp eine Anwendung des *Command*-Entwurfsmusters vorgeführt. Betrachten Sie Sockets als Netzwerkschnittstelle für Ihre eigene Prüfungsaufgabe nicht als etwas esoterisches oder furchteinflößendes, das um jeden Preis vermieden werden muß. Auch wenn sich die meisten Prüfungskandidaten gegen Sockets entscheiden, sind wir der Auffassung, daß eine socketbasierte Netzwerkschnittstelle nicht schwieriger zu entwickeln ist, als eine RMI-Lösung, wenn Sie den Quelltext zu diesem Buch als Grundlage verwenden. Insbesondere sind Sockets die technische Grundlage der meisten Netzwerkprotokolle: Die meisten beliebten neuen Technologien wie Webservices und Enterprise JavaBeans (EJBs) basieren letztendlich auf Sockets. Sogar RMI baut auf Sockets auf, wie wir bereits diskutiert haben. Sie werden davon profitieren, sich mit Sockets vertraut zu machen, und sei es nur, um Ihr Verständnis für die anderen Netzwerktechnologien zu vertiefen. Das nächste Kapitel widmet sich der graphischen Benutzeroberfläche der Beispielanwendung.

7.5 Häufige Fragen

- *Frage*: Muß mein Socketserver multithreadfähig sein?

  Antwort: Ja. Die Prüfungsanforderungen verlangen, daß der Server den Zugriff durch mehrere Benutzer gestattet. Wenn Ihr Socketserver nur single-threadfähig ist, gibt es keine Möglichkeit, um gleichzeitigen Zugriff vorzuführen. Ihre Anwendung würde die Anfragen seriell blockieren, bis eine Anfrage nach der anderen und stets nur eine Anfrage verarbeitet ist. Dies kann eine interessante Lösung sein und umgeht die Notwendigkeit, sich einen Reservierungsmechanismus für einzelne Datensätze zu überlegen, ist aber von Sun Microsystems nicht erlaubt.

- *Frage*: Sollte ich bei meinem Socketserver einen Threadpool verwenden?

  Antwort: Entscheiden Sie selbst. Sie können einen Threadpool verwenden und sich damit für eine gute Lösung entscheiden. Jedesmal, wenn der Socketserver eine neue Socketverbindung akzeptiert, erzeugt er einen neuen Thread, um die Anfrage zu verarbeiten. Sie bekommen das Erzeugen und Zerstören von Threadobjekten aber nicht umsonst. Bei einem tatsächlich ausgelasteten Server mit hohem Transfervolumen kann es sinnvoll sein, das Erzeugen von Threads zu kontrollieren. Ein Threadpool erzeugt beim Starten der Anwendung eine Anzahl von Threads, die nach den Anforderungen an die Performanz Ihrer Anwendung bemessen werden kann. Trifft eine Anfrage beim Server ein, so wird sie von einem Thread aus dem Pool verarbeitet. Nachdem die Anfrage verarbeitet ist, wird der Thread zur späteren Wiederverwendung wieder in den Pool „zurückgelegt". Ein Threadpool verhindert die Unkosten durch das Erzeugen und Zerstören von Threadobjekten. Wir verwenden in unserer Beispielanwendung keinen Threadpool. Sie müssen bei Ihrer Aufgabe keinen Threadpool verwenden, da die Prüfungsanforderungen nicht verlangen, daß sich Ihr Server unter hoher Auslastung performant verhält.

- *Frage*: Was bedeutet die Aussage, TCP sei ein verbindungsorientiertes (*connection-oriented*) Protokoll?

  Antwort: In der Literatur kommt hin und wieder im Zusammenhang mit TCP-Sockets der Begriff „verbindungsorientiert" vor. Dieser Begriff bedeutet, daß zwischen zwei Endpunkten

eine Verbindung existieren muß, bevor die Kommunikation möglich ist. Vergleichen Sie diese Bedingung mit dem verbindungslosen (*connectionless*) User Datagram Protocol (UDP).

- *Frage*: Ist das `SocketCommand` notwendig, um anzuzeigen, welches Kommando bezüglich der Datenbankdatei ausgeführt werden soll?

  Antwort: Nein. Wir haben uns zugunsten der Klarheit dafür entschieden: Die Klasse `SocketCommand` ist ein schönes Beispiel für das Aufzählen der gültigen Kommandos und führt das *Command*-Entwurfsmuster (*Kommando*) vor. Der Nachteil besteht darin, daß der Client beim Hinzufügen neuer Kommandos eine aktualisierte Version der Klasse `SocketCommand` benötigt, um weiterhin Anfragen senden zu können, da die älteren `SocketCommand`-Objekte auf der Serverseite nicht mehr deserialisiert werden können. (Wir könnten die Serialisierungskompatibilität natürlich auf das `serialVersionUID`-Feld beziehen, um Rückwärtskompatibilität zu gewährleisten, aber Sie haben das Problem sicher erkannt.) Alternativ könnte das Kommando als Zeichenkette in einem `DvdCommand`-Objekt verpackt werden. Auf diese Weise können neue Kommandos hinzugefügt werden, ohne die Klasse `SocketCommand` zu ändern, der Server wäre aber dennoch in der Lage das neue Kommando zu interpretieren.

- *Frage*: Sollte ich dem Benutzer erlauben, den von meiner Anwendung verwendeten Port zu ändern?

  Antwort: Diese Flexibilität ist zwar nicht zwingend erforderlich, in der Regel aber eine gute Idee. Andernfalls kann Ihre Anwendung nicht gestartet werden, wenn bereits eine andere Anwendung Ihren fest eingestellten Port verwendet.

- *Frage*: Gibt es Portadressen, die ich bei meiner Wahl vermeiden sollte (unabhängig davon, ob ich die Portadresse hartkodiere oder voreinstelle)?

  Antwort: Die Internet Assigned Numbers Authority (IANA) legt sogenannte „well-known Ports“ (Port 0 bis 1023), „registrierte Ports“ (Port 1024 bis 49151) und „dynamisch bereitgestellte und/oder private Ports“ (Port 49152 bis 65535) fest. Es empfiehlt sich, eine Portadresse aus dem privaten Bereich zu wählen, um Konflikte mit anderen Diensten zu vermeiden. Vermeiden Sie die „well-known ports“, da Sie je nach Betriebssystem, ohne Administratorberechtigung keine dieser Portadressen verwenden dürfen.

- *Frage*: Wie kann ich aufräumen, wenn ein Client seine Verbindung zum Server abbricht?

  Antwort: Wartet der einem bestimmten Client zugeordnete Thread auf eine neue Anweisung von diesem Client, so erhält der Thread eine Ausnahme (das heißt, der Thread kann eine Ausnahme abfangen), wenn der Client die Verbindung abbricht. Bricht der Client die Verbindung ab, bevor der Server die zuvor eingegangene Anfrage beantwortet hat, so erhält der diesem Client zugeordnete Thread beim Versuch die Antwort zu senden eine Ausnahme. In beiden Fällen können Sie Anweisungen zum Aufräumen in die `catch`-Klausel einsetzen.

  Wenn Sie nur bestehende Reservierungen aufheben müssen, können Sie den dem Client zugeordneten Thread als Schlüssel eines `WeakHashMap`-Objektes verwenden, welches die Reservierungen enthält. Wenn der Client die Verbindung unterbricht (und der Thread stirbt) wird die Reservierung schließlich automatisch aus dem `WeakHashMap`-Objekt gelöscht. Schlagen Sie in Kapitel 5 nach (Seite 149).

- *Frage*: Wie kann ich sämtliche Clients aktualisieren, nachdem ein Client eine Buchung auf dem Server durchgeführt hat?

  Antwort: Diese Funktionalität wird von Sun Microsystems nicht gefordert. Sie müßten eine weitere Socketverbindung zwischen Client und Server anlegen, so daß der Server den Client benachrichtigen kann. Sie können dies mit dem Entwurfsmuster *Observer* (*Beobachter*, siehe

Kapitel 8, Seite 252f) kombinieren, so daß sich Clients registrieren können, um bei Buchungen benachrichtigt zu werden.

Kapitel 8

Die graphische Benutzeroberfläche

[0] In den vorangegangenen Kapiteln haben wir die Datenbank- und die Netzwerkschicht der Beispielanwendung detailliert besprochen. Es ist an der Zeit, die letzte Schicht zu entwickeln: die graphische Benutzeroberfläche (Präsentationsschicht). Dieses Kapitel beinhaltet die folgenden Themen:

- Entwicklung einer einfachen aber zweckmäßigen graphischen Benutzeroberfläche.
- Diskussion der wichtigsten Swing-Komponenten und des Ereignismodells.
- Implementing a JTable.
- Implementierung des Entwurfsmusters *Model-View-Controller* (*MVC*).
- Implementierung des Entwurfsmusters *Observer* (*Beobachter*).

[1] Sie müssen die Kapitel 6 und 7 über Remote Method Invocation (RMI) beziehungsweise Sockets als Netzwerkschnittstelle noch nicht gelesen haben, um dieses Kapitel über die graphische Benutzeroberfläche lesen zu können. Der größte Teil dieses Kapitel widmet sich dem Design von graphischen Benutzeroberflächen, wozu die Netzwerkkapitel 6 und 7 nicht benötigt werden. Die Abschnitte dieses Kapitels, in denen eine Verbindung zur Datenbankdatei erforderlich ist, stützen sich auf das *Factory*-Entwurfsmuster, welches ein Objekt erzeugt, dessen Klasse unser connection interface implementiert. Sie werden den Zusammenhang zwischen der graphischen Benutzeroberfläche und der Datenbankschicht allerdings nicht verstehen, ohne die Netzwerkkapitel vorher durchzuarbeiten.

Tipp: Die Verwendung der connection interfaces in unserem Beispielprojekt zeigt einen der Vorteile von Interfaces: Das Interface legt einen „Vertrag“ fest auf dessen Einhaltung wir uns verlassen können. Wir brauchen keine konkrete Implementierung, um unsere Fabrikklasse zu entwickeln und zu verwenden.

---

[2] Wenn Sie mit der Arbeit an Ihrer Prüfungsaufgabe beginnen, können Sie die graphische Benutzeroberfläche vor der Netzwerkschnittstelle entwickeln. Diese Vorgehensweise ist sinnvoll und eventuell in der Praxis zu bevorzugen, da der Kunde bereits im *stand-alone*-Betriebsmodus mit der Anwendung arbeiten kann, während Sie die restlichen Komponenten entwickeln.

[3] Für dieses Buch war es sinnvoll, die Netzwerkschnittstelle vor der graphischen Benutzeroberfläche zu entwickeln, as we will be connecting to the database via direct connection and via the various networking options from within our GUI.

Warnung: Die Benutzer nehmen häufig fälschlicherweise an, daß ein Softwareprojekt kurz vor dem Abschluß steht, wenn sie die graphische Benutzeroberfläche gesehen haben. Sie können viel Ärger vermeiden, indem Sie Ihrem Kunden klar machen, wieviel zusätzliche Arbeit noch erforderlich ist, wenn Sie die graphische Benutzeroberfläche vorstellen. Eine Möglichkeit, derartige Mißverständnisse zu vermeiden ist, das Napkin Look-and-Feel für Swing-Anwendungen zu verwenden (siehe *http://napkinlaf.sourceforge.net*).

[4] Keine Komponente einer Anwendung wirkt sich so deutlich auf den Benutzer aus wie die graphische Benutzeroberfläche. Sie ist das Medium, über das der Benutzer die Anwendung bedient. Bedauerlicherweise wird die graphische Benutzeroberfläche häufig als unwichtigster Teil bei der Entwicklung einer Anwendung betrachtet. Dies ist ein Trugschluß, da die graphische Benutzeroberfläche den Ausschlag dafür geben kann, ob eine Anwendung von den Benutzern akzeptiert oder abgelehnt wird. Ist die graphische Benutzeroberfläche verschachtelt und schwierig zu bedienen, so reagieren die Benutzer schnell frustriert und halten die Anwendung insgesamt für schlecht. Eine Anwendung ist letztendlich nur dann erfolgreich, wenn sie von den Benutzern mit Erfolg bedient werden kann.

[5] Das Ziel dieses Kapitel besteht darin, diejenigen Leser in die Richtlinien für das Layout, das Design und die Entwicklung graphischer Benutzeroberflächen einzuführen, die keine oder nur wenige Erfahrung auf diesem Gebiet haben. Die graphische Benutzeroberfläche ist ein Pflichtbestandteil der Prüfung zum *Sun Certified Java Developer* und dieses Kapitel rüstet Anfänger mit allen benötigten Informationen aus, um eine graphische Benutzeroberfläche zu entwickeln und in Betrieb zu nehmen.

8.1 Richtlinien für graphische Benutzeroberflächen

[6] Die Anforderungen an die Prüfung zum *Sun Certified Java Developer* verlangen, daß die gesamte Programmierarbeit von *einem* Entwickler verrichtet wird. Das betrifft auch die dreischichtige Architektur der Prüfungsaufgabe, also Datenbankschicht, Geschäftslogik und Präsentationsschicht. Die Prüfungsanforderungen weichen hierbei von der häufigen Arbeitspraxis ab, bei der sich ein Entwickler auf eine der drei Schichten spezialisiert. Häufig sind die Entwickler bei einem dreischichtigen Projekt in eine Backend-Gruppe und eine separate Frontend-Gruppe aufgeteilt. Bei der Prüfung zum *Sun Certified Java Developer* ist *ein* Entwickler für alle drei Schichten zuständig.

Tipp: Die Tatsache, daß Sie Entwicklungsarbeit verrichten, die in der Regel auf drei Teams verteilt wird, kann beim Studium der Anleitung zu Ihrer Prüfungsaufgabe Verwirrung stiften. Prüfungskandidaten haben häufig den Eindruck, daß eine Anforderung an die graphische Benutzeroberfläche einer Anforderung an die Datenbankschicht widerspricht. Solche scheinbaren Widersprüche lösen sich aber auf, wenn Sie die Anforderung aus der Perspektive des jeweiligen separaten Teams betrachten. Das Oberflächenteam setzt voraus, daß sich das Datenbankteam an bestimmte Vereinbarungen hält und konzentriert sich auf seine eigenen Anforderungen.

[7] Die Aufgabe, eine graphische Benutzeroberfläche zu entwickeln, kann eine einschüchternde Wirkung haben, vor allem für diejenigen, die nicht primär Frontend-Entwickler sind. Es gibt viele Java-Entwickler, die noch nie eine Oberflächenanweisung geschrieben haben, aber dennoch äußerst fähige Programmierer sind. Erschwerend kommt hinzu, daß auf einer graphischen Benutzeroberfläche nur sehr wenig Beschreibungstext untergebracht werden kann und die Erwartungen der Benutzer nicht bekannt sind. Die Richtlinien in den beiden folgenden Unterabschnitte sollen Ihnen die Entwicklung einer graphischen Benutzeroberfläche erleichtern.

[8] Die beiden folgenden Unterabschnitte vermitteln einen groben Überblick über die Richtlinien für das Layout von graphischen Benutzeroberflächen im allgemeinen sowie für Anwendungsschnittstellen zu menschlichen Benutzern. Eventuell können Sie diese Richtlinien bei der graphischen Benutzeroberfläche Ihrer Prüfungsaufgabe gebrauchen.

Bemerkung: Die Richtlinien für das Layout von graphischen Benutzeroberflächen sowie für Anwendungsschnittstellen zu menschlichen Benutzern sind selbst umfangreiche Themen. Die folgenden beiden Unterabschnitten stellen lediglich die prüfungsrelevanten Grundzüge dar.

8.1.1 Aufbau einer graphischen Benutzeroberfläche

[9] Einer häufig geäußerten Falschannahme zufolge, ist das Layout graphischer Benutzeroberflächen mehr eine Kunst als eine Wissenschaft. Der Vorgang des Layouts und die Informationsarchitektur (*information architecture*) werden aber durch diese Blickrichtung entstellt. Am Anfang des Computerzeitalters wurde die Programmierung ebenfalls mehr als Kunst denn als Wissenschaft betrachtet, eine, gemessen am heutigen Stand, völlig unsinnige Sichtweise. Nachdem sich in der Kunst des Programmierens eine ingenieurmethodische Arbeitsweise durchgesetzt hatte, wurde das Programmieren schnell zur Wissenschaft beziehungsweise zum Ingenieurfach. Dasselbe gilt für die Richtlinien zum Aufbau graphischer Benutzeroberflächen.

[10] Der Kernpunkt beim Layout und Design einer graphischen Benutzeroberfläche ist die klare und präzise Visualisierung von Informationen. Der Benutzer soll in der Lage sein, die von einer graphischen Benutzeroberfläche präsentierten Informationen mühelos zu erfassen. Die Undeutlichkeit dieser Aussage ist beabsichtigt. Unabhängig davon, wie stark der Vorgang der Erfassung von Benutzereingaben standardisiert ist, beinhaltet die Konstruktion der graphischen Benutzeroberfläche stets einen gewissen Grad an Intuition. Nicht alle Möglichkeiten können vorausgeahnt und in der Standardisierung des Datenerfassungsvorgangs berücksichtigt werden. Die Intuition betrifft den künstlerischen Anteil am Layout einer graphischen Benutzeroberfläche, das heißt die Grauzonen in der Entwicklung, die noch niemand zuvor betreten hat.

[11] Glücklicherweise haben sich bereits viele Entwickler mit dem Entwurf einer graphischen Benutzeroberfläche für eine SCJD-Prüfungsaufgabe beschäftigt. Das Layout der Daten hat im Laufe der Zeit eine so deutliche Standardisierung erfahren, daß die Anleitung der Prüfungsaufgaben sogar eine Swingkomponente für die Visualisierung dieser „datenbankartigen“ Datensätze angibt: `javax.swing.JTable`. Sun Microsystems *verlangt* sogar, daß bei der Prüfung zum *Sun Certified Java Developer* eine `JTable`-Komponente verwendet wird. Die `JTable`-Komponente ist die wichtigste Komponente in der Datenvisualisierung der Beispielanwendung und wird in diesem Kapitel noch ausführlich behandelt.

[12] Die `JTable`-Komponente repräsentiert ein Arbeitsblatt (*spreadsheet*) wie bei einer Tabellenkalkulation und visualisiert Daten, die in Zeilen und Spalten unterteilt sind, siehe Abbildung 8-1, Seite 228 (Buch). Bei dieser Darstellung genügt die Angabe von Zeile und Spalte, um einen Eintrag zu lokalisieren. Der Eintrag befindet sich beim Schnittpunkt der Zeile mit der Spalte. Dieses Schema ist die bestmögliche Zusammenführung (*ultimate reconciliation*) zwischen den Daten und dem zu ihrer Visualisierung verfügbaren Raum. Erhält die Tabelle eine zusätzliche Zeile, so nimmt die Höhe der Tabelle etwa um die Höhe des verwendeten Zeichensatzes zu. Das Hinzufügen einer zusätzlichen Spalte wirkt sich stärker aus, als die Hinzunahme einer weiteren Zeile. Das Tabellenschema ist äußerst flexibel. Änderungen an der unterliegenden Datenstruktur gehen mühelos in Datenvisualierung über. Das Tabellenschema stellt viel Information auf wenig Raum dar.

[13] Diese Diskussion trägt nicht nur scheinbar Offensichtliches zusammen, sondern betont, daß sich

die Tabelle zur Visualisierung entsprechend strukturierter Daten anbietet. Die Betrachtungen zur `JTable`-Komponente können und sollen auf das Gesamtlayout der graphischen Benutzeroberfläche übertragen werden. Berücksichtigen Sie beim Layout einer graphischen Benutzeroberfläche daher die folgenden Kriterien:

- Die Daten werden auf minimalem Raum visualisiert, ohne daß die Visualisierung überfrachtet oder unorganisiert wirkt.
- Der Benutzer muß benötigte Informationen schnell finden können.
- In der Regel muß sich eine graphische Benutzeroberfläche an unterschiedliche Datenmengen anpassen können.

8.1.2 Anwendungschnittstellen zu menschlichen Benutzern

[14] Die Richtlinien für Anwendungsschnittstellen zu menschlichen Benutzern gehen über die schlicht organisatorischen Ansätze zur Datenvisualisierung im vorigen Unterabschnitt hinaus. Das Design einer Anwendungsschnittstelle zu menschlichen Benutzern organisiert die Abfolge der Arbeitsschritte des Benutzers, der eine Aufgabe erledigen möchte. Die Anordnung der Daten im Hinblick auf bestmögliche Lesbarkeit ist eine layoutspezifische Entscheidung. Die Organisation des gesamten Vorgangs, einen Eintrag auszuwählen und dann zu ändern, ist dagegen eine Entscheidung, die die Anwendungsschnittstelle zum menschlichen Benutzer betrifft. Kurz: Eine Anwendungsschnittstelle zu menschlichen Benutzern bestimmt, wie ein Benutzer mit einer Anwendung interagiert.

[15] Der Zweck einer graphischen Benutzeroberfläche besteht darin, die mühelose Interaktion zwischen Benutzer und Anwendung zu ermöglichen. Eine graphische Benutzeroberfläche schirmt die eigentliche Funktionalität einer Anwendung vom Benutzer ab. Klickt der Benutzer eine Schaltfläche an, um ein Dokument zu speichern, so führt die Anwendung in der Regel mehrere Schritte aus, zum Beispiel:

1. Die Anwendung überprüft die Integrität der Datei.
2. Die Anwendung ermittelt die physikalische Größe der zu speichernden Datei.
3. Die Anwendung prüft, ob auf der Festplatte genügend Platz vorhanden ist, um die Datei zu speichern.
4. Die Anwendung schreibt die Datei ins Dateisystem.

[16] Fast jede Anwendung führt diese Schritte aus, um eine Datei zu speichern. Bei den meisten Anwendungen ist nur eine einzige Benutzereingabe erforderlich, um diese Abfolge von Schritten auszulösen, nämlich das Auswählen der Option „Sichern“ im Menü „Datei“.

[17] Stellen Sie im Gegensatz dazu eine Anwendung vor, die vom Benutzer verlangt, die „Datei“-Optionen „Prüfe Integrität“, „Ermittle Größe“, „Prüfe verfügbaren Speicherplatz“ (wobei der im vorigen Schritt ermittelt Platzbedarf eingesetzt werden muß), und schließlich (nach der Bestätigung, daß der verfügbare Platz auf der Festplatte ausreicht) „Schreibe Datei“ nacheinander aufzurufen. Diese Anwendung macht das Speichern einer Datei umständlicher als nötig.

[18] Dieses Beispiel veranschaulicht den Begriff der bestmöglichen Abschirmung des Benutzers von den Arbeitsschritten der Anwendung. Die Anwendung sollte die obigen Schritte ohne Benutzereingabe ausführen können. Der Benutzer muß nur dann über die Aktivitäten einer Anwendung informiert werden, wenn ein Fehler auftritt. Reicht beispielsweise der verfügbare Platz auf der Festplatte nicht aus, um die Datei zu speichern, so muß der Benutzer darüber informiert werden, daß die Anwendung

diesen Schritt nicht alleine durchführen kann. Andernfalls soll der Benutzer lediglich die „Sichern“-Option wählen müssen und die Anwendung alle Zwischenschritte verbergen.

[19] Das primäre Ziel beim Design einer Anwendungsschnittstelle für menschliche Benutzer besteht darin, die anwendungsinternen Arbeitsschritte und die vom Benutzer ausgelösten Aktionen aufeinander abzustimmen. Diese Aufgabe klingt offensichtlich und simpel, ist aber alles andere als einfach. Die Anwendung muß aus der Perspektive eines Durchschnittsmenschen betrachtet werden, der noch nie mit dieser Anwendung gearbeitet hat. Diese Objektivität ist für viele Anwendungsdesigner und -entwickler eine Herausforderung. Die Vertrautheit mit der Anwendung steht der Urteilsfähigkeit darüber im Weg, welche Aktionen für den Benutzer und welche internen Schritte für die Anwendung notwendig sind.

[20] Hier kann es sich auszahlen, die Anleitung zu Ihrer Prüfungsaufgabe noch einmal sorgfältig zu lesen. Trennen Sie sich vom Design Ihrer Anwendung, betrachten Sie die Anwendung aus der Perspektive des Benutzers und sehen Sie die Anforderungen im Hinblick auf Hinweise zur Interaktion zwischen Benutzer und Anwendung durch:

- Wer sind die primären Benutzer der Anwendung?
- Zu welchem Zweck arbeiten die Benutzer mit der Anwendung? Welche Aktionen müssen die Benutzer mit der Anwendung veranlassen können?
- Welche internen Schritte muß die Anwendung ausführen, um diese Aktionen veranlassen zu können?

[21] Beantworten Sie diese Fragen schriftlich. Stellen Sie sich beispielsweise eine Anwendung vor, mit der die Benutzer in einem Dokument Fußnoten anlegen können. Die Liste der vom Benutzer benötigten Aktionen lautet zum Beispiel „Fußnote anlegen“, „Fußnote anzeigen“, „Fußnote ändern“ und „Fußnote löschen“.

[22] Diejenigen Leser die sich mit der Anforderungsanalyse bei Projekten auskennen, haben sicherlich erkannt, daß wir einige einfache sogenannte „Anwendungsfälle“ (*use cases*) erfaßt haben. Ein Anwendungsfall dokumentiert eine Interaktion zwischen Benutzer und Anwendung beziehungsweise eine Aktion, die Benutzer der Anwendung veranlassen kann. Ein Anwendungsfall beschreibt in der Regel detailliert, welche Schritte zur Ausführung einer Aktion erforderlich sind. Abbildung 8-2, Seite 230 (Buch), zeigt ein einfaches Diagramm der obigen Anwendungsfälle.

Bemerkung: Eine Anwendungsschnittstelle für menschliche Benutzer überbrückt die Lücke zwischen den Anwendungsfällen und der Funktionalität der Anwendung. Das Design einer Anwendung sollte alle in den Anwendungsfällen beschriebenen Aktionen berücksichtigen. Die Aufgabe der Schnittstelle besteht darin, dem Benutzer die Ausführung dieser Aktionen so leicht wie möglich zu machen. Gestattet die Architektur der Anwendung nicht, daß alle in den Anwendungsfällen dokumentierten Aktionen ausgeführt werden können, dann hat das Design der Anwendung einen ernsthaften Fehler.

---

[23] Anschließend folgt die detaillierte Beschreibung jeder einzelnen Aktion, zum Beispiel: „Der Benutzer legt eine Fußnote an, indem er zuerst die Textstelle auswählt, zu der die neue Fußnote gehört. Danach kann ein optionaler Name für die Fußnote vergeben und der Fußnotentext eintragen werden. Schließlich sichert der Benutzer die Fußnote.“

[24] Die Beschreibung ist nun detailliert genug, um mit dem Design der Schnittstelle zu beginnen. Betrachten Sie zunächst, wie sich die benötigten anwendungsinternen Schritte mit den erforderlichen Benutzeraktionen in Einklang bringen lassen. Planen Sie die Schnittstelle so, daß der Benutzer so

wenig Schritte wie möglich benötigt, um eine Aktion durchzuführen. Nehmen Sie dies als Leitprinzip bei der Entwicklung Ihrer Anwendungsschnittstelle für menschliche Benutzer.

Tipp: In der Regel gestattet eine Anwendungsschnittstelle für menschliche Benutzer dem Benutzer, eine Aktion auf mehr als eine Weise auszulösen. Eine Anwendung kann das Speichern eines Dokumentes zum Beispiel über einen Menüpunkt (*menu item*), eine bekannte Tastenkombination (*keystroke*) oder eine Schaltfläche (*button*) erlauben. Ein solches Design gilt als gut und der einzelne Benutzer merkt sich seine bevorzugte Variante. Von den Prüfungskandidaten wird *erwartet*, standardisierte Funktionalität wie Tastenkombinationen, Schaltflächen und Menüpunkte anzulegen, auch wenn die Anleitungen zu den Prüfungsaufgaben dies *nicht ausdrücklich verlangen*.

---

[25] Eine weitere Richtlinie betrifft die Art, wie der Benutzer eine Anwendungsschnittstelle für menschliche Benutzer tatsächlich durchläuft und interpretiert. In den meisten westlichen Gesellschaften beginnt der Benutzer in der linken oberen Ecke, arbeitet von oben nach unten und von links nach rechts. Die Ursache dafür besteht in der in westlichen Ländern üblichen Schreibweise: Die Worte werden von links nach rechts gelesen, so daß die meisten westlichen Benutzer dazu neigen, die linke obere Ecke des Bildschirms als Anfang und die rechte untere Ecke als Ende wahrzunehmen. Alles dazwischen sind die erforderlichen Schritte, um vom Anfang zum Ende zu kommen, ~~siehe Abbildung 8-3, Seite 231 (Buch)~~.

Tipp: Selbst wenn Sie ein asiatischer Entwickler sind und in Asien arbeiten, sollten Sie berücksichtigen, daß Ihre Prüfungsaufgabe wahrscheinlich nach westlichen Erwartungen bewertet wird.

---

[26] Es ist vorteilhaft, bei der Planung des Arbeitsablaufes in einer Anwendungsschnittstelle für menschliche Benutzer die einzelnen Schritte entlang des Verarbeitungspfades nach Wichtigkeit anzuordnen. Die Komponenten, auf die der Benutzer zuerst aufmerksam werden soll, sollten daher in der Nähe der linken oberen Ecke angebracht werden. Die Anordnung der Komponenten der Schnittstelle relativ zur linken oberen Ecke vermittelt den Eindruck, daß die entsprechenden Aktionen in dieser Reihenfolge ausgeführt werden sollen, ~~siehe zum Beispiel Abbildung 8-4, Seite 232 (Buch)~~.

[27] Der in ~~Abbildung 8-4~~ eingezeichnete Pfad vermittelt den Eindruck, daß der Benutzer zuerst Eintragungen und Operationen in der Tabelle ausführt und anschließend die „Enter“-Schaltfläche anklickt. Wäre die Schaltfläche *vor* der Tabelle platziert worden, würde der Benutzer die Reihenfolge der Schritte wahrscheinlich trotzdem richtig interpretieren, aber die Schnittstelle wäre nicht intuitiv und offensichtlich schlecht organisiert.

[28] Die Gruppierung von Elementen wird als ähnliche Funktionalität verstanden. Daher befinden sich die Schaltflächen „Dokument öffnen“, „Dokument sichern“ und „Neues Dokument“ bei den meisten Textverarbeitungsprogrammen in unmittelbarer Nachbarschaft.

Tipp: Wenn Sie unsicher sind, welches Design Sie für eine Aktion wählen sollen, so daß sie vom Benutzer als intuitiv empfunden wird, dann gibt es hin und wieder eine sehr nahe liegende Lösung: Sehen Sie bei einem bekannten Programm mit ähnlicher Funktionalität nach. Wenn Sie sich beim Entwurf Ihrer Anwendungsschnittstelle für menschliche Benutzer nach einer Vorlage richten, an die die Benutzer gewöhnt sind, wird ihre Anwendung wahrscheinlich ebenfalls als intuitiv empfunden werden.

---

[29] Die Platzierung einer Schaltfläche neben einem Texteingabefeld legt nahe, daß die Schaltfläche eine mit dem Texteingabefeld verknüpfte Aktion ausführt. Dieser Grundsatz kann zu einem gro-

ben Fehler führen, den Sie unter allen Umständen vermeiden sollten: Da der Benutzer gruppierte Elemente als funktional zusammengehörig deutet, ist eine „Programm beenden“-Schaltfläche direkt neben einer „Sichern“-Schaltfläche ungünstig. Das Beenden der Anwendung ist eine destruktive Aktion, weil die Programmausführung abgebrochen wird, während das Speichern nicht nur keine destruktive, sondern eine sehr häufig aufgerufene Aktion ist. Durch diesen Fehler könnten Benutzer irrtümlich das Programm beenden, obwohl sie nur ihre Änderungen speichern wollten.

[30] Denken Sie stets daran, daß der durchschnittliche Benutzer „unfallgefährdet“ ist. Als Entwickler müssen Sie berücksichtigen, daß der Benutzer jederzeit einen Fehler machen oder eine falsche Schaltfläche anklicken kann. Durch gutes Design mit sinnvoll gruppierten und platzierten Schaltflächen läßt sich dieses Risiko verringern. Bei einer „richtigen“ Anwendung können Sie beim Design der Schnittstelle noch einen Schritt weitergehen und mit Hilfe des `javax.swing.undo`-Packages einen Undo-Mechanismus implementieren. Diese Erweiterung der Funktionalität gestattet dem Benutzer die Anwendung in einen früheren Zustand zurück zu versetzen und die Auswirkungen einer irrtümlichen oder fehlerhaften Aktion zurückzunehmen. Obwohl nützlich, geht diese Funktionalität weit über die Anforderungen der Prüfung zum *Sun Certified Java Developer* hinaus.

[31] Beachten Sie beim Design Ihrer Benutzerschnittstelle auch, daß Sie eventuell ungerechtfertigte Annahmen über die Betriebsumgebung Ihrer Anwendung machen. Nur weil Ihr Rechner eine Maus hat, dürfen Sie nicht davon ausgehen, daß jeder Rechner auf dem Ihre Anwendung installiert wird, ebenfalls eine Maus hat. Dies gilt in besonderem Maße für Java-Anwendungen, da sie auf jeder beliebigen Plattform betrieben werden können. Verknüpfen Sie daher jede funktionale Komponente auf dem Bildschirm mit einer mnemonischen Tastenkombination zur Menüsteuerung, einer Funktionstaste oder einer gewöhnlichen Tastenkombination, um die Bedienbarkeit Ihrer Benutzerschnittstelle zu gewährleisten. Auf diese Weise kann der Benutzer die Anwendung über die Tastatur steuern, wenn sie auf einer Plattform ohne Maus läuft.

[32] Eine gute abschließende Faustregel besagt, daß Komponenten an festen, vorhersagbaren Orten platziert werden sollen. Das Einhalten dieser Regel unterstützt den Benutzer dabei, sein „motorisches Gedächtnis“ (*muscle memory*) zu gebrauchen. Die Wirkung dieses Phänomens tritt ein, wenn sich der Benutzer so sehr an den Platz einer Komponente auf dem Bildschirm gewöhnt hat, daß sein Gedächtnis unterbewußt weiß wo sie sich befindet. Die Bedienung der Benutzerschnittstelle erfordert somit weniger Überlegung und wird für den Benutzer zur Gewohnheit. Denken Sie beispielsweise an den Webbrowser, den Sie während der vergangenen drei Jahre verwendet haben. Selbst wenn Sie mehrmals den Browser gewechselt haben, sind die Schaltflächen zum Vorwärts-/Rückwärtsblättern fast immer an derselben Stelle. Das Lokalisieren dieser Schaltflächen mit dem Auge und ihre Bedienung mit der Maus sind schneller geworden, weil Sie wissen, wo sich die Schaltflächen befinden. Es ist wie Fahrradfahren: Auch wenn Sie einige Jahre lang nicht gefahren sind, vergessen Sie doch nicht, wie man das Gleichgewicht hält.

Bemerkung: Benutzer passen sich an neue Richtlinien für Benutzerschnittstellen an, so daß sich der Begriff des Intuitiven stetig wandelt. Im Laufe der Jahre wurden beliebte neue Eigenschaften und Fähigkeiten für Benutzerschnittstellen eingeführt. Beispielsweise wurde das Äquivalent von Karteikarten mit Reitern, eine sehr effizientes Verfahren, um verschiedene Ebenen von Informationen in ein und demselben Fenster darzustellen, sofort von den Entwicklern akzeptiert. Aufgrund ihrer Effizienz wurden und werden Karteikarten mit Reitern von den Entwicklern noch immer häufig verwendet. Bedingt durch das häufige Vorkommen haben sich die Benutzer daran gewöhnt. Zwei weitere Beispiele für Eigenschaften und Fähigkeiten, die von den Benutzern graphischer Benutzeroberflächen in der Regel verwendet werden sind eine Leiste mit ikonisierten Schaltflächen und das webbrowserartige Vorwärts-/Rückwärtsblättern.

---

[33] Vergessen Sie während des Designs und der Entwicklung der Benutzerschnittstelle Ihrer Anwendung nicht das Testen. Das Testen einer Benutzerschnittstelle ist weniger schablonenhaft, als das Testen der restlichen Anwendung. Am besten testen Sie die graphische Benutzeroberfläche, indem Sie jemanden mit der Anwendung arbeiten lassen, der sie noch nie zuvor gesehen und verwendet hat.

[34] Entwerfen Sie zuerst einen Prototyp der Benutzerschnittstelle. Der Prototyp ist eine einfache Version der geplanten Schnittstelle, noch ohne Funktion. In diesem Stadium genügt auch eine Skizze auf einem Blatt Papier. Investieren nicht zuviel Arbeit in den Prototyp, bevor Sie von seiner Benutzerfreundlichkeit/Gebrauchstauglichkeit (*usability*) überzeugt sind.

[35] Geben Sie Ihrem Tester eine schriftliche Aufgabenliste und bitten Sie ihn, die Liste abzuarbeiten. Beobachten Sie den Tester während er sich durch die Liste arbeitet beim Umgang mit dem Prototyp Ihrer Benutzerschnittstelle. Geben Sie dem Tester während Ihrer Beobachtung keine Anleitung, um Mißverständnisse zu klären oder zu helfen, wenn er nicht weiter kommt. Gehen Sie davon aus, daß der Tester hängen bleibt oder einen Teil der Aufgaben nicht bewerkstelligen kann. Das kommt häufig vor. Achten Sie darauf, nicht verärgert zu reagieren. Wenn der Tester nicht weiterkommt fragen Sie ihn nach dem beabsichtigten Schritt und wo er die nächste Aktion erwartet. Nehmen Sie die Kommentare des Testers ernst, denn es sind Vorschläge, wie Sie die Benutzerschnittstelle vor dem nächsten Tester verbessern können. Wiederholen Sie diesen Testvorgang sooft Sie es für nötig halten oder bis der Augenblick eintritt, daß alle Ihre Tester die Schnittstelle mühelose oder nur mit geringen Schwierigkeiten bedienen können. Ausführlichere Informationen zum Testen der Benutzerfreundlichkeit/Gebrauchstauglichkeit (*usability*) sowie entsprechende Designtips finden Sie im Abschnitt 8.7.

8.2 Das Entwurfsmuster Model-View-Controller (MVC)

[36] Der vorige Abschnitt hat zeigt, daß die anwendungsinterne Funktionalität und die Interaktion zwischen Anwendung und Benutzer über eine graphische Benutzeroberfläche aufeinander abgestimmt werden müssen. Das Entwurfsmuster *Model-View-Controller* (*MVC*) beschränkt die Abhängigkeit zwischen der Benutzerschnittstelle und der Implementierung einer Anwendung und schirmt die Datenbankschicht und die Geschäftslogik gegen Änderungen in der Präsentationsschicht ab.

8.2.1 Motivation zur Verwendung des Entwurfsmusters

[37] Der Zweck des *MVC*-Entwurfsmusters besteht hauptsächlich darin, die verschiedenen funktionalen Bereiche einer Benutzerschnittstelle voneinander zu trennen. Eine Benutzerschnittstelle läßt sich typischerweise in die folgenden funktionalen Bereiche untergliedern:

- Schnittstelle zur Anwendung.
- Visualisierung der Daten für den Benutzer.
- Erfassen, Parsen und Verarbeiten von Benutzereingaben.

Diese drei funktionalen Bereichen entsprechen der „Datenmodellkomponente“ (*model*), der „Präsentationskomponente“ (*view*) sowie der „Programmsteuerungskomponente“ (*controller*) des *MVC*-Entwurfsmusters.

[39] Das *MVC*-Entwurfsmuster gestattet die logische Trennung dieser drei Bereiche. In einem großen Projekt kann jeder dieser Bereiche einem separaten Team zugeordnet werden, dessen Entwickler sich

auf die entsprechende Aufgabe spezialisiert haben. Entwickler mit Talent für das Layout von graphischen Benutzeroberflächen arbeiten an der Präsentationskomponenten, während anderere Entwickler, die sich auf das Parsen und Verarbeiten von Informationen aus verschiedenen Quellen verstehen, an der Programmsteuerungskomponente arbeiten.

[40] Der Einsatz des *MVC*-Entwurfsmusters gestattet schnelle Änderungen an der Benutzerschnittstelle. Sowohl die Beispielanwendung als auch die von Sun Microsystems ausgegebenen Prüfungsaufgaben verfügen über eine graphische Benutzeroberfläche, um die Anwendung im *stand-alone*-Betriebsmodus betreiben zu können. Durch Austauschen der Präsentations- und der Steuerungskomponente läßt sich die Anwendung mühelos zu einer Webapplikation umbauen oder eine Schnittstelle für einen Webservice hinzufügen.

Tipp: Häufig wird beim Erlernen eines neuen Gegenstandes empfohlen, das Erlernte praktisch anzuwenden. Wenn Sie mittelfristig auch die Prüfung zum *Sun Certified Web Component Developer* anstreben, können Sie Ihre Prüfungsaufgabe um eine Webschnittstelle erweitern. Das ist unter Umständen einfacher, als ein völlig neues Projekt aufzusetzen, da die Geschäftslogik bereits vorhanden ist und Sie sich nur auf den Teil konzentrieren müssen, der mit Ihrer übernächsten Prüfung zusammenhängt.

8.2.2 Das Entwurfsmuster im Detail

[41] Das *MVC*-Entwurfsmuster besteht aus drei Komponenten, siehe Tabelle 8.1. Das Entwurfsmuster wirkt zwar kompliziert, das Verfahren ist aber, wenn es an einem praktischen Beispiel erläutert wird, tatsächlich sehr einfach und leicht verständlich. Wir besprechen das *MVC*-Entwurfsmuster am Beispiel einer Bestellung in einem Fast-Food-Restaurant.

[42] Beim Restaurant eingetroffen, werfen Sie einen Blick auf die Speisekarte und entscheiden sich für einen Salat. Sie bestellen bei der Bedienung einen Salat und einen Milchshake. Die Bedienung geht in die Küche, holt Ihre Bestellung ab und bringt sie zu Ihnen. Dieses häufige Szenario ist eine Ausprägung des *MVC*-Entwurfsmusters. Wir betrachten die Hauptakteure nun einzeln und bilden sie auf ihre Äquivalente im Entwurfsmuster ab.

[44] Die Präsentationskomponente in diesem Beispiel ist die Speisekarte. Der Präsentationsmodus kann variieren: Im Restaurant bekommen Sie ein gedrucktes Exemplar, während die Speisekarte am Autoschalter an der Hauswand angebracht ist. Beide Modi präsentieren dieselbe Information, aber in unterschiedlichen Formaten: Eine gedruckte Version für Gäste im Restaurant beziehungsweise eine an der Wand befestigte Version am Autoschalter.

[45] Die Programmsteuerungskomponente in diesem Beispiel ist die Bedienung, die Ihre Bestellung entgegennimmt. Die Bedienung funktioniert wie ein Puffer zwischen Ihnen und den tatsächlichen Arbeitsabläufen im Restaurant. Die Bedienung nimmt Ihre Bestellung auf und veranlaßt, daß das Essen zubereitet wird.

Bemerkung: Die Bedienung kann auch als Teil der Präsentationskomponente betrachtet werden, da sie den Kunden befragt und die Bezahlung für das Essen entgegennimmt. Wir lassen der Einfachheit halber beiseite, daß die Bedienung mehr als eine Rolle spielen kann.

---

[46] Die Bedienung (Programmsteuerungskomponente) kümmert sich darum, daß Sie Ihr Essen bekommen, wobei sie weder sehen noch verstehen müssen, wie die Bearbeitung Ihrer Bestellung abläuft. Die Bedienung kann direkt in die Küche gehen und den Koch beauftragen, den Salat und

Komponente	Funktion
Datenmodell (*model*)	Die Datenmodellkomponente ist die Schnittstelle zum Rest der Anwendung, gestattet der beziehungsweise den Programmsteuerungskomponente(n) das konsistente Abfragen und/oder Ändern der Daten und gibt die Daten im angeforderten Format zurück. Nicht selten implementiert die Datenmodellkomponente wiederum ein Entwurfsmuster, beispielsweise *Façade*, um systemnahe Details der Anwendung zu verbergen, den beobachtenden Teil von *Observer*, um die Präsentationskomponente automatisch aktualisieren zu können oder *Singleton*, um zu erzwingen, daß stets höchstens ein Exemplar der Datenmodellkomponente existiert.
Präsentation (*view*)	Die Präsentationskomponente bewirkt die eigentliche Visualisierung der Daten. Das unterliegende System kann Daten auf mehrere Arten speichern, aber es ist die Aufgabe der Präsentationskomponente, diese Daten zu interpretieren und in ein passendes Format umzuwandeln, beispielsweise in eine HTML-Seite, ein PostScript-Dokument oder für die graphische Benutzeroberfläche einer Java-Anwendung. Unabhängig davon, welcher Präsentationsmodus gewählt wird, ist einzig und allein die Präsentationskomponente dafür zuständig, das anwendungsintern verwendete Datenformat in das Präsentationsformat zu transformieren. Die Präsentationskomponente muß die Interaktion mit einem menschlichen Benutzer erlauben. Klickt der Benutzer eine Schaltfläche auf der graphischen Benutzeroberfläche an, so ist die Präsentationskomponente dafür zuständig, diese Aktion zu behandeln und ihre Verarbeitung zu veranlassen, das heißt in einen Methodenaufruf in der Programmsteuerungskomponente umzuwandeln. Es ist wichtig, zu verstehen, daß die Präsentationskomponente nicht zwingend eine Bildschirmausgabe liefert. Die Ausgabe kann beispielsweise auch ein XML-Dokument sein, daß an eine entfernte Anwendung gesendet wird.
Programmsteuerung (*controller*)	Die Programmsteuerungskomponente vermittelt zwischen der Datenmodellkomponente und der Präsentationskomponente einer Anwendung. Die Präsentationskomponente ruft die Programmsteuerungskomponente auf und fordert die Ausführung einer bestimmten Aktion an. Für diese Aktion kann es erforderlich sein, daß Daten von der Präsentationskomponente übergeben oder an sie zurückgegeben werden. Stets gilt: Jede Interaktion zwischen Präsentationskomponente und Datenmodellkomponente verläuft durch die Programmsteuerungskomponente. Die Erhaltung der durch *MVC* implementierten hohen Abstraktionsstufe setzt voraus, daß die Präsentationskomponente niemals an der Programmsteuerungskomponente vorbei direkt mit den unter ihr liegenden Komponenten kommuniziert. Die Präsentationskomponente darf beispielsweise die Datenbank (in der Beispielanwendung die Datenbankdatei) nicht direkt aufrufen und eine Anfrage veranlassen. Statt dessen soll die Präsentationskomponente die Programmsteuerungskomponente aufrufen und anweisen, mit der Datenmodellkomponente zu kommunizieren.

Tabelle 8.1: Die Komponenten der *MVC*-Architektur.

den Milchshake zuzubereiten oder beide Tätigkeiten an zwei weitere Mitarbeiter übertragen. Wichtig ist, daß Sie, wenn Sie ein Essen bestellen, die Küche nicht selbst betreten und den Angestellten sagen, daß sie Ihre Bestellung zubereiten sollen und die Zubereitung der Bestellung mit Sicherheit nicht selbst in die Hand nehmen. Die Organisation der „Nahrungsmittelversorgung“ ist die Aufgabe der Bedienung. Sie kümmern sich nur um den nächsten Präsentationsmodus (die Mahlzeit selbst).

[47] Die dritte *MVC*-Komponente ist die Datenmodellkomponente, im Restaurantbeispiel die Küche. Die Küche stellt verschiedene Präsentationsmodi zur Verfügung, zum Beispiel das Standardessen oder die Spezialität des Tages (auf einer eingelegten Seite in der Speisekarte oder einer Tafel am Autoschalter), nimmt die Bestellung entgegen und liefert die Daten (das Essen) für den nächsten Präsentationsmodus.

8.2.3 Vorteile von MVC

[48] Der Vorteil des *MVC*-Entwurfsmusters besteht hauptsächlich in der Abstraktion zwischen der Präsentation von Daten und den anwendungsinternen Operationen. Diese Trennung hat eine Reihe von Auswirkungen: *MVC* ist die ideale Lösung für eine Anwendung, die Daten in vielen verschiedenen Ausgabeformaten liefern muß. Während die Datenmodellkomponente bei *MVC* unverändert bleibt, kann der Präsentationsmodus leicht geändert werden. Die Anwendung kann sowohl eine graphische Benutzeroberfläche als auch eine webbrowserbasierte HTML-Oberfläche besitzen, ohne das restliche System modifizieren zu müssen.

[49] Das *MVC*-Entwurfsmuster beschränkt die Tragweite von Änderungen in einer Anwendung. Der Benutzer beschäftigt sich zumeist mit der Funktionalität in der Präsentationsschicht, dem Teil der Anwendung, mit dem er tatsächlich arbeitet. Kunden wünschen sich daher oft Änderungen am Design der graphischen Benutzeroberfläche. Das *MVC*-Entwurfsmuster begrenzt die Auswirkungen dieser Änderungen und verhindert häufig, daß andere Teile der Anwendung beeinträchtigt werden. Dies bedeutet keineswegs, daß *MVC* Änderungen an der Anwendung zu einem Kinderspiel macht. Das Frontend der Anwendung wird aber mit Sicherheit erheblich flexibler.

8.2.4 Nachteile von MVC

[50] Jedes Ding hat Vor- und Nachteile und das *MVC*-Entwurfsmuster ist keine Ausnahme. *MVC* ist nicht immer die beste Lösung. Eine Anwendung, welche die Fähigkeit, Daten auf unterschiedliche Weise zu liefern, nicht benötigt, ist kein Kandidat für dieses Entwurfsmuster. Das *MVC*-Entwurfsmuster bringt einen beträchtlichen Abstraktionsbedarf mit sich und der damit verbundene Aufwand wiegt die Entwicklungszeit und -performanz nicht immer auf.

[51] Der Einsatz von *MVC* zahlt sich in der Regel aus. Außerdem ist *MVC* eines der eher leichtgewichtigen Entwurfsmuster. Es wird häufig verwendet und von den meisten Entwicklungsumgebungen (*development platforms*) unterstützt. Der Nutzen von *MVC* korreliert praktisch direkt mit der Größe des Projektes. Erfordert ein Projekt mehrere Entwickler und erhebliche Erweiterbarkeit, so ist der Einsatz des *MVC*-Entwurfsmusters sicherlich ein Gewinn. Ist ein Projekt dagegen klein und wird wahrscheinlich nie überarbeitet, so ist der Einsatz von *MVC* vermutlich nicht nötig.

8.2.5 Alternativen zu MVC

[52] Die offensichtlichste Alternative zu *MVC* besteht darin, das Entwurfsmuster komplett zu vermeiden. Design und Implementierung einer Anwendung können die anwendungsinterne Funktionalität fest mit der Benutzerschnittstelle verdrahten. Dadurch wird allerdings die Flexibilität der graphischen Benutzeroberfläche beeinträchtigt. Jede Änderung wirkt sich unmittelbar auf jeden Bereich der Anwendung aus, da das gesamte System als eine Einheit angelegt ist.

[53] Bei manchen Entwicklungsumgebungen ist die feste Bindung (*tight integration*) der Präsentationsschicht an die Datenbankschicht möglich. Durch den ereignisgetriebenen Ansatz der Java-Plattform sind einige Aspekte des *MVC*-Entwurfsmusters bei allen Benutzerschnittstellen inherent. Jede Komponente einer graphischen Benutzeroberfläche löst Ereignisse aus und zu jedem Ereignis gehört eine Klassen, deren Objekte dieses Ereignis behandeln. *MVC* ist in diesem Sinne eingebaut, aber der Entwickler kann diesen Effekt einsetzen und muß nicht die gesamte *MVC*-Architektur nutzen.

8.3 Swing und das Abstract Window Toolkit (AWT)

[54] In den ersten beiden Abschnitten dieses Kapitels haben wir einige abstrakte Richtlinien für das Design von Benutzerschnittstellen kennengelernt. Die meisten Ansätze, beispielsweise das Entwurfsmuster *Model-View-Controller*, sind plattformunabhängig und beziehen sich auf die Architektur eines Softwareprojektes, das anschließend in nahezu jeder Programmiersprache entwickelt werden kann. Der Rest dieses Kapitels beschreibt dagegen die Entwicklung einer graphischen Benutzeroberfläche mit den Bordwerkzeugen von Java, nämlich dem Abstract Window Toolkit (AWT) und der Swing-Bibliothek.

[55] Über das Programmieren mit AWT und Swing sind ganze Bücher geschrieben worden. Dieser Abschnitt faßt lediglich die Grundzüge überblicksartig zusammen. Eine ausführlichere Darstellung finden Sie unter der Internetadresse *http://java.sun.com/docs/books/tutorial/uiswing*.

8.3.1 Die Layoutmanager BorderLayout und FlowLayout

[56] Die Komponenten einer graphischen Benutzeroberfläche (zum Beispiel Schaltflächen, Textbereiche und Graphiken) werden in einem sogenannten „Container“ (zum Beispiel `JFrame`, `JPanel` oder `JWindow` aus dem Package `javax.swing`) angelegt beziehungsweise hinzugefügt. Die Container verwenden sogenannte „Layoutmanager“, um die Anordnung ihrer Komponenten festzulegen. Die Container ~~JWindow~~ und ~~JFrame~~ verwenden per Voreinstellung den Layoutmanager `java.awt.BorderLayout`. Die Funktionsweise dieses Layoutschemas hat einige eigentümliche Merkmale, die wir wiederholen wollen.

[57/58] Der Layoutmanager `BorderLayout` unterteilt den Container in fünf Bereiche, ~~siehe Abbildung 8-5, Seite 237 (Buch)~~. Eine Komponente (eine Referenz auf ein Objekt der abstrakten AWT-Klasse `java.awt.Component`) kann mit Hilfe der `java.awt.Container`-Methode `add()` in einen dieser Bereiche eingesetzt werden. Die Methode erwartet zwei Parameter. Der erste Parameter von `add()` ist eine Referenz auf die hinzuzufügende Komponente und hat den AWT-Typ `Component`. Der zweite Parameter von `add()`, die Formatierungseigenschaft (*constraint*), gibt an, in welchen der fünf Bereiche die Komponente platziert werden soll. Beim Layoutmanager `BorderLayout` sind hierfür die folgenden Konstanten erlaubt:

- `BorderLayOut.NORTH`
- `BorderLayOut.SOUTH`
- `BorderLayOut.EAST`
- `BorderLayOut.WEST`
- `BorderLayOut.CENTER`

[59] Eine in einem dieser fünf Bereiche platzierte Komponente füllt den gesamten verfügbaren Raum aus, wobei die für den spezifischen Bereich geltenden Formatierungseigenschaften berücksichtigt werden (siehe unten). Das folgende Beispiel platziert je eine Schaltfläche in jedem der fünf Bereiche, Abbildung 8.1 zeigt das Ergebnis:

```
import java.awt.*;
import javax.swing.*;

public class BorderLayoutExample extends JFrame {
    public static void main(String[] args) {
        new BorderLayoutExample().setVisible(true);
    }
```

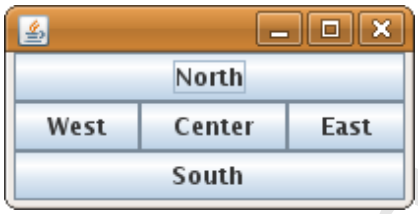

Abbildung 8.1: Ein Beispiel für den Layoutmanager `BorderLayout`.

```
    public BorderLayoutExample() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        add(new JButton("North"), BorderLayout.NORTH);
        add(new JButton("South"), BorderLayout.SOUTH);
        add(new JButton("East"), BorderLayout.EAST);
        add(new JButton("West"), BorderLayout.WEST);
        add(new JButton("Center"), BorderLayout.CENTER);
        pack();
    }
}
```

Bemerkung: Vor Version 5 des Java Development Kits war es notwendig, zunächst eine Referenz auf die Inhaltsebene (*content pane*) eines `JFrame`-Containers anzufordern, um *dieser* anschließend die Komponenten hinzufügen zu können. Seit Version 5 des Java Development Kits überschreibt die Swing-Klasse `JFrame` die von der AWT-Klasse `Container` geerbten `add()`-Methoden. Im obigen Beispiel gestattet eine dieser `add()`-Methoden *scheinbar*, die Komponenten direkt im `JFrame`-Container anzulegen. Die überschriebene `add()`-Methode legt die Komponenten aber eigentlich in der Inhaltsebene an. (Siehe auch Unterabschnitt 8.5.3.)

[60] Die Schaltflächen in den Bereichen `NORTH` und `SOUTH` füllen die gesamte Breite des `JFrame`-Containers von links nach rechts aus, nicht aber den Raum oberhalb und unterhalb der jeweiligen Schaltfläche. Die Schaltfläche in den Bereichen `EAST` und `WEST` füllen die verbleibende Höhe zwischen den ersteren beiden Schaltflächen aus, nicht aber den Raum links beziehungsweise rechts neben der jeweiligen Schaltfläche. Die Schaltfläche im Bereich `CENTER` dehnt sich in alle vier Richtungen aus.

Warnung: Wird in einem Bereich mehr als eine Komponente angelegt, so wird nur die zuletzt angelegte Komponente im Vordergrund angezeigt. Dieses Verhalten stiftet häufig Verwirrung, wenn ein Entwickler zum ersten Mal Komponenten anlegt, da einige Komponenten scheinbar verloren gegangen sind. There is probably no reason why you ever want to intentionally add two components in the same region.

[61] Es gibt verschiedene Möglichkeiten, um in einem Container Komponenten anzulegen und dabei den in der obigen Warnung beschriebene Effekt zu vermeiden. Die beste und vielleicht einfachste Möglichkeit, um die Anordnung mehrerer Komponenten festzulegen besteht darin, zuerst alle Komponenten in einem `JPanel`-Container anzulegen und diesen anschließend in einem der fünf `BorderLayout`-Bereiche zu platzieren. Im Gegensatz zu `BorderLayout` verwendet der `JPanel`-Container per Voreinstellung den Layoutmanager `java.awt.FlowLayout`. Die Eigenschaften dieses Layoutmanagers sind viel leichter nutzbar als bei `java.awt.GridLayout` oder gar `java.awt.GridBagLayout` (siehe Seite 247ff). Jede unter `FlowLayout` platzierte Komponente behält ihre vorgeschlagene Größe bei und ist horizontal beweglich.

[62] Der Layoutmanager `FlowLayout` verfügt über einige Ausrichtungsparameter. Sie können den

Abbildung 8.2: Der Layoutmanager `FlowLayout`. Links oben: Voreinstellung des Layoutmanagers. Jede Komponente hat ihre vorgeschlagene Größe. Die Komponenten sind nebeneinander angeordnet. Rechts oben: Expandierter Container. Links unten: Gestauchter Container. Rechts unten: Die Komponenten sind rechtsbündig ausgerichtet.

Konstruktor `FlowLayout(int index)` mit einer der folgenden, in der Klasse `FlowLayout` selbst definierten Konstanten aufrufen:

- `FlowLayout.CENTER`
- `FlowLayout.RIGHT`
- `FlowLayout.LEFT`

[63] Das nächste Beispiel kombiniert einen `JPanel`- mit einem `JFrame`-Container, um drei Schaltflächen anzuordnen:

```
import java.awt.*;
import javax.swing.*;

public class ExampleFlowLayout extends JFrame {
    public static void main(String[] args) {
        new ExampleFlowLayout().setVisible(true);
    }
    public ExampleFlowLayout() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        JPanel thePanel = new JPanel();
        thePanel.add(new JButton("One"));
        thePanel.add(new JButton("Two"));
        thePanel.add(new JButton("Three"));
        add(thePanel, BorderLayout.CENTER);
        pack();
    }
}
```

Ändern Sie die Größe des `JFrame`-Containers, um die Wirkung auf den `JPanel`-Container zu beobachten. Abbildung 8.2 zeigt einige Beispiele.

[64] Beachten Sie, daß die Abmessungen der Schaltflächen vom Layoutmanager `FlowLayout` berechnet und genau in der richtigen Größe gezeichnet werden, um ihre Beschriftung anzuzeigen. Beachten Sie ebenfalls, daß `FlowLayout` die Beschriftung per Voreinstellung zentriert ausrichtet. Das folgende Beispiel ändert lediglich die Ausrichtung der Schaltflächen aus dem vorigen Beispiel, siehe wiederum Abbildung 8.2:

```
import java.awt.*;
import javax.swing.*;

public class MyFrame extends JFrame {
    public static void main(String[] args) {
```

```
        new MyFrame().setVisible(true);
    }
    public MyFrame() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        JPanel thePanel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
        thePanel.add(new JButton("One"));
        thePanel.add(new JButton("Two"));
        thePanel.add(new JButton("Three"));
        add(thePanel, BorderLayout.CENTER);
        pack();
        setSize(300, 70);
    }
}
```

Tipp: Die Kombination der Containertypen `JFrame` und `JPanel` ist eine mächtige aber zugleich einfache Möglichkeit, um die Komponenten einer graphischen Benutzeroberfläche anzuordnen. Es gibt sicherlich kompliziertere Wege, um eine Benutzerschnittstelle zu gestalten, aber die Kombination dieser beiden Containertypen ist wahrscheinlich mehr als ausreichend, um eine graphische Benutzeroberfläche für die Prüfung zum *Sun Certified Java Developer* zu entwickeln.

8.3.2 Das Look-and-Feel der graphischen Benutzeroberfläche

[65] Swing unterstützt austauschbares (*pluggable*) Look-and-Feel. Dies ist eine willkommene Nebenwirkung der leichtgewichtigen Swing-Komponenten, deren Design und Funktionalität vom Entwickler überschrieben werden können. Ein Beispiel für diese Eigenschaft ist das „Ocean"-Look-and-Feel. Dieses Look-and-Feel der graphischen Benutzeroberfläche steht auf jeder Plattform zur Verfügung, die Java unterstützt, bietet sich also als plattformunabhängige Voreinstellung an. Wenn sich eine graphische Benutzeroberfläche unabhängig von der unterliegenden Plattform absolut identisch verhalten soll und aussehen muß, ist das „Ocean"-Look-and-Feel die richtige Wahl.

[66] Das Look-and-Feel einer Swing-basierten graphischen Benutzeroberfläche kann *on the fly* programmatisch von der Anwendung selbst geändert werden. Häufige Beispiele für `javax.swing.LookAndFeel`-Unterklassen sind:

- `javax.swing.plaf.metal.MetalLookAndFeel`
- `com.sun.java.swing.plaf.windows.WindowsLookAndFeel`
- `com.sun.java.swing.plaf.motif.MotifLookAndFeel`
- `com.sun.java.swing.plaf.gtk.GTKLookAndFeel`
- `com.apple.mrj.swing.MacLookAndFeel`

Warnung: Das einzige, garantiert auf jeder Plattform vorhandene Look-and-Feel ist `javax.swing.plaf.metal.MetalLookAndFeel` aus den Java-Standardpackages. Alle anderen Look-and-Feels liegen in Packages, die nicht zum Standardumfang von Java gehören, beispielsweise `com.sun` und `com.apple`. Diese herstellerspezifischen Packages sind in Laufzeitumgebungen anderer Hersteller wahrscheinlich nicht vorhanden and may not even exist in all JVMs produced by a particular vendor. Das `WindowsLookAndFeel` ist beispielsweise nur auf Microsoft Windowsplattformen vorhanden und das `GTKLookAndFeel` nur auf Plattformen, die GTK unterstützen (typischerweise Unix und unixartige Systeme).

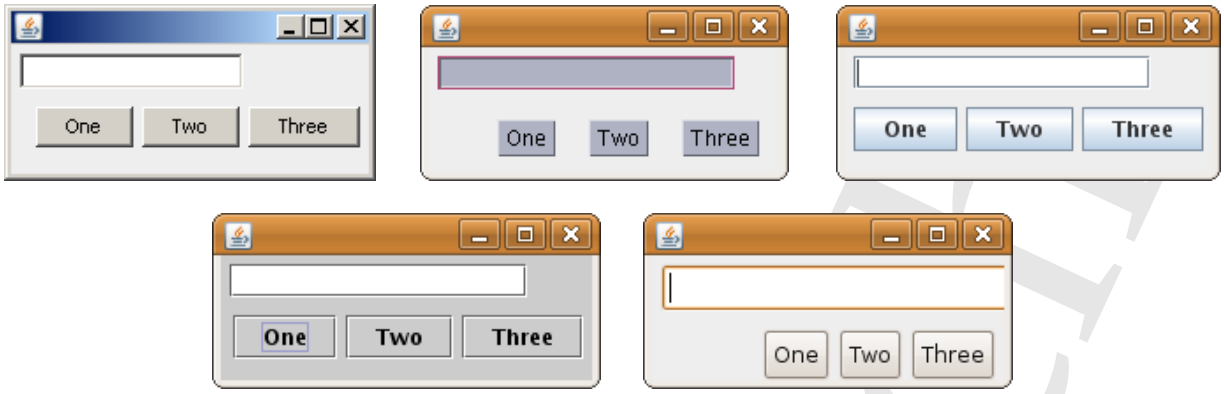

Abbildung 8.3: Verschiedene Look-and-Feels: Oben links: Microsoft Windows. Oben Mitte: Motif. Oben rechts: Metal mit „Ocean"-Thema. Unten links: Metal mit „Steel"-Thema. Unter rechts: GTK (auf Unix-Plattformen).

[67] In den bisherigen Beispielen haben wir das „Ocean"-Thema des „Metal"-Look-and-Feels verwendet. Im folgenden Beispiel lann ein Look-and-Feel beim Programmstart auf der Kommandozeile übergeben werden. Voreinstellung ist `WindowsLookAndFeel`, falls das Programm ohne Angabe eines Look-and-Feels aufgerufen wird:

```
import java.awt.*;
import javax.swing.*;

public class MyFrame extends JFrame {
    public static void main(String[] args) throws Exception {
        new MyFrame(args).setVisible(true);
    }
    public MyFrame(String[] args) throws Exception {
        String lookAndFeelName = (args.length > 0)
            ? args[0]
            : "com.sun.java.swing.plaf.windows.WindowsLookAndFeel";
        UIManager.setLookAndFeel(lookAndFeelName);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        Panel topPanel = new Panel(new FlowLayout(FlowLayout.LEFT));
        topPanel.add(new JTextField(15));
        add(topPanel, BorderLayout.NORTH);
        Panel centerPanel = new Panel(new FlowLayout(FlowLayout.RIGHT));
        centerPanel.add(new JButton("One"));
        centerPanel.add(new JButton("Two"));
        centerPanel.add(new JButton("Three"));
        add(centerPanel, BorderLayout.CENTER);
        pack();
        setSize(210, 100);
    }
}
```

Abbildung 8.3 zeigt verschiedene Look-and-Feels.

[68] Eventuell ist Ihnen aufgefallen, daß die Fenster rechts oben und links unten in Abbildung 8.3 das „Metal"-Look-and-Feel mit verschiedenen Themen (*themes*) verwenden. Vor Version 5 des Java Development Kits hatte das „Metal"-Look-and-Feel nur ein Thema, nämlich „Steel" (links unten). Seit Version 5 des Java Development Kits hat Sun Microsystems das Standard-Look-and-Feel durch das neue Thema „Ocean" (rechts oben) verbessert. Der folgende Programmaufruf wurde verwendet, um das traditionelle „Steel"-Thema vorzuführen:

```
java -Dswing.metalTheme=steel MyFrame javax.swing.plaf.metal.MetalLookAndFeel
```

Warnung: Die Layoutmanager haben die Aufgabe, die Positionierung der Komponenten relativ zu einander sowie die Größe des Containers zu bestimmen. Komponenten können mit der `Component`-Methode `setLocation()` explizit positioniert und ihre Größe per `setSize()`-Methode (siehe voriges Beispiel) festgelegt werden. Die einzelnen Fenster in Abbildung 8.3 zeigen allerdings, daß die Abmessungen der Komponenten abhängig vom verwendeten Look-and-Feel variieren, so daß eine absichtlich festgelegte Größe oder Position für eine Komponente zu sonderbaren Ergebnissen führen kann. Wir empfehlen daher mit Nachdruck, das Positionieren der Komponenten und die Bestimmung der Containergröße dem Layoutmanager zu überlassen.

[69] Bei Verwendung eines anderen Look-and-Feels nimmt die graphische Benutzeroberfläche nicht nur die visuelle Erscheinungsform einer Anwendung unter der entsprechenden Plattform an, sondern paßt auch die Funktionsweise der Komponenten der Benutzerschnittstelle an. Eine Dropdown-Liste (*dropdown menu*) im „Motif"-Look-and-Feel unterscheidet sich deutlich von einer Dropdown-Liste im „Windows"- oder „Metal"-Look-and-Feel. Das liegt daran, daß „Motif" die Erscheinungsform und Funktionalität einer graphischen Benutzeroberfläche im X Window System repräsentiert.

8.3.3 Die Komponente JLabel

[70/71] Graphische Benutzeroberflächen führen in der Regel zu jeder Eingabekomponente eine Beschriftung (*label*), um dem Benutzer mitzuteilen, welche Information mit der jeweiligen Komponente erfaßt wird. Beispielsweise kann ein Texteingabefeld für den Nachnamen mit „Nachname" beschriftet werden, um den Benutzer darauf hinzuweisen, welche Information in das Texteingabefeld eingetragen werden soll. Die Swing-Komponente für Beschriftungen heißt `javax.swing.JLabel`. Die folgende Zeile erzeugt eine Beschriftung:

```
JLabel zipCodeLabel = new JLabel("Zip code");
```

[72] Die Beschriftung ist für sich alleine noch nicht aufregend. Sie können daher einen Buchstaben festlegen, der als mnemonische Tastenkombination (unterstrichenes Zeichen) dargestellt wird. Da die Beschriftung selbst nichts mit dem Fokus anfangen kann, definieren Sie in der Regel zusammen mit der mnemonischen Tastenkombination die Eingabekomponente, die durch Anschlagen dieser Tastenkombination den Fokus erhält:

```
zipCodeLabel.setDisplayedMnemonics('Z');
zipCodeLabel.setLabelFor(zipCode);
```

[73/74] Drückt der Benutzer im vorliegenden Fall gleichzeitig die Tasten „Alt" und „Z", so erhält das durch `zipCode` referenzierte Feld den Fokus, siehe Beispiel im folgenden Unterabschnitt sowie Abbildung 8.4.

8.3.4 Die Komponente JTextField

[75] Die Swing-Komponente für ein einfaches Texteingabefeld heißt `javax.swing.JTextField` und erlaubt die Eingabe von einfachem Text bis zu einer festgelegten Länge. Die folgende Zeile erzeugt ein Texteingabefeld mit einer Breite von 15 Zeichen:

```
JTextField zipCode = new JTextField(15);
```

[76] Die anderen Konstruktoren der Klasse `JTextField` erlauben die Angabe eines Standardwertes, der im Texteingabefeld angezeigt wird beziehungsweise die Übergabe eines *`javax.swing.text.Document`*-Objektes zur Validierung des eingegebenen Wertes (Die Validierung von Benutzereingaben wird im Unterunterabschnitt 8.3.4.1 behandelt.)

Abbildung 8.4: Beispiel für die Komponenten `JLabel` und `JTextField`.

[77] Das folgende Beispiel führt die Komponenten `JLabel` und `JTextField` vor. Abbildung 8.4 zeigt das mit diesen Anweisungen implementierte Fenster. Das Fenster enthält zusätzlich eine Schaltfläche, damit Sie mit den Fokus aus dem Texteingabefeld entfernen und über die Tastenkombination „Alt“ und „Z“ wieder dorthin zurücksetzen können:

```
import java.awt.*;
import javax.swing.*;
import javax.swing.text.*;

public class MyFrame extends JFrame {
    public static void main(String[] args) throws Exception {
        new MyFrame().setVisible(true);
    }
    public MyFrame() throws Exception {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        JLabel zipCodeLabel = new JLabel("Zip code");
        JTextField zipCode = new JTextField(15);
        zipCodeLabel.setDisplayedMnemonic('Z');
        zipCodeLabel.setLabelFor(zipCode);
        this.add(zipCodeLabel);
        this.add(zipCode);
        this.add(new JButton("A button for focus"));
        pack();
    }
}
```

8.3.4.1 Validierung des Inhaltes bei Texteingabefeldern

[78] Wäre die Welt perfekt, die Benutzer unserer Anwendung würden stets gültige Daten eingeben. In der Realität machen die Menschen aber Fehler und die Eingabe ungültiger Daten kann erhebliche Probleme verursachen, wenn die Fehler nicht rechtzeitig abgefangen werden. Es ist gerechtfertigt, einen angemessen Aufwand in Kauf nehmen, um die Eingabe ungültiger Daten zu verhindern und die Daten nach ihrer Eingabe zu prüfen.

[79] Im obigen Beispiel gibt es ein `zipCode`-Feld. In den Vereinigten Staaten sind Postleitzahlen (*zip codes*) fünfstellig und dienen dazu, das Zielgebiet eines Briefes oder Paketes zu identifizieren. Unser Beispielprogramm erlaubt allerdings die Eingabe beliebiger Daten, ungeachtet ihrer Länge oder Ihres Inhaltes.

[80] Die Einschränkung des Texteingabefeldes auf Postleitzahlen läßt sich durch Kombination einer `javax.swing.JFormattedTextField`-Komponente (die Klasse `JFormattedTextField` ist von `JTextField` abgeleitet) mit einem Hilfsobjekt vom Typ `javax.swing.text.MaskFormatter` erwirken. Ein Texteingabefeld vom Typ `JFormattedTextField` läßt nur Eingaben zu, die einer definierten Maske entsprechen. Das `MaskFormatter`-Hilfsobjekt gestattet, mit einfachen Mitteln eine maskenbasierte Formatvorlage zu definieren. Ein Zeichen, daß nicht zur Maske paßt wird nicht als Eingabe akzeptiert. Die Maske für eine Ziffer ist das Zeichen `#`. Damit können wir die Definition unseres

`zipCode`-Feldes folgendermaßen ändern:

```
MaskFormatter fiveDigits = new MaskFormatter("#####");
JTextField zipCode = new JFormattedTextField(fiveDigits);
zipCode.setColumns(5);
```

[81] Nach diesen Änderungen werden Sie feststellen, daß Sie keine Zeichen mehr eingeben können, die keine Ziffern sind und Sie können nicht mehr als fünf Ziffern eingeben.

[82] Wir wollen noch einen Schritt weitergehen: Angenommen, der Benutzer möchte zusätzlich zur Postleitzahl nach einem Bindestrich vier weitere Stellen angeben, um den Lieferbezirk stärker einzugrenzen. Stellen Sie sich vor, daß die ersten fünf Ziffern das Postamt angeben, die nächsten beiden Ziffern einen Bezirk von Blocks entlang einer größeren Straße und die beiden letzten Ziffern den genauen Block.

[83] Die um vier Stellen erweiterte Postleitzahl wird nicht überall verwendet und es nicht vorgeschrieben, dieses Format zu benutzen, selbst wenn es verfügbar ist (wenn wir schon dabei sind, scheint die Angabe der Postleitzahl bei keiner Sendung mit dem US Postal Service zwingend erforderlich zu sein, ist aber vermutlich ratsam, wenn Ihre Sendung zügig befördert werden soll).

[84] Eine Lösung besteht darin, einen eigenen Typ von `JTextField` abzuleiten, der ein eigenes Dokumentmodell (*document model*) besitzt, dessen `insertString()`-Methode wir überschreiben. Die Definition unseres `zipCode`-Feldes ändert sich damit noch einmal:

```
JTextField zipCode = new ZipTextField(9);
```

[85] Die Klasse `ZipTextField` ist von `JTextField` abgeleitet, überschreibt aber nur die Methode `createDefaultModel()` und legt zwei Konstruktoren an:

```
private class ZipTextField extends JTextField {
    ZipTextField() {
        super();
    }
    ZipTextField(int columns) {
        super(columns);
    }
    protected Document createDefaultModel() {
        return new ZipDocument();
    }
}
```

[86/87] Sie können weitere Konstruktoren anlegen oder überhaupt keinen eigenen Konstruktor verwenden (also den Standardkonstruktor verwenden und die geerbte Methode `setColumns()` aufrufen). Die aufwendige Validierungsarbeit findet im Dokumentmodell des `ZipTextField`-Objektes statt. Für die Basisklasse `JTextField` ist das Dokumentmodell `javax.swing.text.PlainDocument` voreingestellt. Wir leiten unser eigenes Dokumentmodell `ZipDocument` von `PlainDocument` ab. `ZipDocument` validiert den in das Texteingabefeld eingegebenen Inhalt.

Bemerkung: Die Klasse `JTextField` implementiert, wie viele andere Swing-Komponenten, das Entwurfsmuster *Model-View-Controller*. Damit gibt es zwei mögliche Bereiche, um die Eingabe von Daten zu kontrollieren, nämlich in der Datenmodellkomponente, die die eingegebenen Daten enthält oder in der Programmsteuerungskomponente, bevor die Daten an die Datenmodellkomponente übergeben werden.

```
private class ZipDocument extends PlainDocument {
    public void insertString(int offs, String str, AttributeSet a)
```

```
            throws BadLocationException {
            if (str == null) {
                return;
            }
            for (char c: str.toCharArray()) {
                if (! ((Character.isDigit(c) && offs < 10 && offs !=5)
                        || (c == '-' && offs == 5))) {
                    return;
                }
            }
            super.insertString(offs, str, a);
        }
    }
```

[88] Wir haben der Einfachheit halber nur die `insertString()`-Methode überschrieben und prüfen lediglich, ob jedes eingegebene Zeichen an seiner Position gültig ist. Ist die Eingabe gültig, so rufen wir die `insertString()`-Methode der Basisklasse auf, um die Zeichenkette tatsächlich in das Dokument einzusetzen.

[89] Da dieses Beispiel einfach sein sollte, haben wir das Löschen von eingegebenen Daten nicht implementiert (wozu die Methode `remove()` überschrieben werden müßte). Die Distribution der Beispielanwendung, die Sie aus dem „Source Code"-Abschnitt der Apress-Website (*http://www.apress.com/book/sourcecode*) herunterladen können, enthält ein aufwendigeres Beispiel in dem eine Portadresse validiert wird.

Warnung: Die diskutierte Kombination aus `ZipTextField` und `ZipDocument` hindert den Benutzer keineswegs daran, zum Beispiel 12345-67 einzugeben und mit dem nächsten Texteingabefeld oder einer anderen Komponente in der graphischen Benutzeroberfläche fortzufahren. Folglich sind zusätzliche Prüfungen erforderlich. Sie finden ein Beispiel in der Distribution der Beispielanwendung, die Sie aus dem „Source Code"-Abschnitt der Apress-Website (*http://www.apress.com/book/sourcecode*) herunterladen können

---

8.3.5 Die Komponente JButton

[90] Die Komponente `JButton` erzeugt eine Schaltfläche, mit der eine anwendungsspezifische Aktion ausgelöst werden kann. Beispiel:

```
JButton exitButton = new JButton("Exit");
```

[91] Es gibt weitere `JButton`-Konstruktoren, etwa um ein Icon (ein Objekt vom Typ *`Icon`*) anstelle oder zusätzlich zur Beschriftung anzubringen sowie um die Schaltfläche mit einer Aktion (einem Objekt vom Typ *`Action`*) zu verknüpfen.

[92] Die Schaltfläche ist noch wirkungslos, da wir noch nicht festgelegt haben, was beim Anklicken geschehen soll. Zu diesem Zweck müssen wir einen Ereignisbehandler (Objekt vom Typ *`ActionListener`*) mit der Schaltfläche verknüpfen:

```
exitButton.addActionListener(anActionListener);
```

[93] Das Interface *`java.awt.event.ActionListener`* gestattet auf Aktionen (Ereignisse) zu reagieren, zum Beispiel das Anklicken einer Schaltfläche entweder alleine oder zusammen mit „Ctrl", „Alt", der Umschalt- oder der Metataste. Wird eine solche Aktion ausgeführt, so ruft der Ereignisbehandlerthread die Methode `actionPerformed()` auf.

[94] Das Interface *ActionListener* kann als anonyme innere Klasse, als private Klasse, als gewöhnliche äußere Klasse oder von Ihrer Präsentationskomponente implementiert werden. Das nächste Beispiel zeigt eine Implementierung von *ActionListener* als anomyme innere Klasse:

```
JButton exitButton = new JButton("Exit");
exitButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            System.out.println("Somebody clicked the Exit button");
            System.exit(1);
        }
    });
```

Bemerkung: Die letzte Zeile im obigen Quelltextauszug sieht seltsam aus. Wenn Sie aber rückwärts zählen, zeigt sich, daß die schließende geschweifte Klammer (`}`) zur öffnenden geschweiften Klammer bei `new ActionListener() {` gehört. Die schließende runde Klammer (`)`) gehört zu öffnenden runden Klammer von `addActionListener(`.

[95] Anonyme innere Klassen haben Vor- und Nachteile. Einige ihrer Vorteile sind:

- Es ist nicht erforderlich, eine separate Klasse anzulegen, nur um die vom Benutzer ausgelösten Ereignisse zu behandeln.
- Die Prüfung, welche Komponente das Ereignis ausgelöst hat, entfällt.
- Die Ereignisbehandlung befindet sich in unmittelbarer Nähe zur Komponente, die das Ereignis auslöst.

[96] Wenn Sie das *MVC*-Entwurfsmuster implementieren, bevorzugen Sie eventuell, daß die vom Benutzer ausgelösten Ereignisse in der Programmsteuerungskomponente behandelt werden. Alternativ kann eine einfache anonyme Klasse eine Methode aus der Programmsteuerungskomponente aufrufen, wodurch die Abhängigkeit der Programmsteuerungskomponente von der graphischen Benutzeroberfläche verringert wird.

[97] Der letzte Punkt kann sich auch nachteilig auswirken. Eventuell möchten Sie nicht, daß die Anweisungen zur Ereignisbehandlung in unmittelbarer Nachbarschaft zu den Anweisungen zur Konfiguration der Präsentationskomponente stehen. Es ist unter Umständen sinnvoller, die Anweisungen zur Ereignisbehandlung separat anzulegen.

[98] Falls Sie sich für eine separate Klasse oder Methode zur Ereignisbehandlung entscheiden, können Sie der Komponente, die das Ereignis auslöst, per `setActionCommand()` eine Zeichenkette zuweisen, um die auslösende Komponente später identifizieren zu können. Diese Zeichenkette kann aus dem `ActionEvent`-Objekt abgefragt werden, das der `actionPerformed()`-Methode beim Aufruf übergeben wird. Das Beispiel im nächsten Unterabschnitt führt diesen Ansatz vor.

8.3.6 Die Komponente JRadioButton

[99] Radiobuttons sind kleine logisch gruppierte Knöpfe, wobei stets höchstens ein Knopf gedrückt sein kann. Die Bezeichnung „Radiobutton“ stammt von den älteren Radiogeräten mit voreingestellten Sendern. Da Sie stets nur einen Sender hören können, konnte nicht mehr als ein Knopf (voreingestellter Sender) gedrückt werden.

[100] Die folgende Anweisung erzeugt eine `JRadioButton`-Komponente:

```
JRadionButton serverButton = new JRadioButton("Server");
```

[101] Es gibt weitere `JRadioButton`-Konstruktoren, etwa um ein Icon (ein Objekt vom Typ *`Icon`*) anstelle oder zusätzlich zur Beschriftung anzubringen, den Anfangszustand des Radiobuttons einzustellen oder die Komponente mit einer Aktion (einem Objekt vom Typ *`Action`*) zu verknüpfen.

[102] Analog zur `JButton`-Komponente kann jeder `JRadioButton`-Komponente ein Ereignisbehandler (ein Objekt vom Typ *`ActionListener`*) zugeordnet werden. Sie können bei Radiobuttons einen Ereignisbehandler zum Beispiel dazu verwenden, um je nach gewähltem Radiobutton bestimmte Felder zu aktivieren oder deaktivieren. Nicht jeder Radiobutton braucht einen Ereignisbehandler. Eventuell genügt es, den Zustand erst auswerten, nach dem der Benutzer eine andere Aktion ausgeführt hat, indem Sie per `isSelected()` ermitteln, welche Wahl der Benutzer eingestellt hat.

[103] Radiobuttons erfüllen ihren Zweck in der Regel nur dann, wenn mehrere von ihnen logisch zu einer Gruppe zusammengefaßt sind, so daß stets höchstens ein Radiobutton gedrückt sein kann. Sie deklarieren eine Gruppe von Radiobuttons, indem Sie ein `ButtonGroup`-Objekt erzeugen, dem Sie anschließend Referenzen auf die einzelnen `JRadioButton`-Komponenten übergeben:

```
ButtonGroup applicationMode = new ButtonGroup();
applicationMode.add(serverButton);
```

[104] Das folgende Beispiel beinhaltet zwei Radiobuttons und eine Schaltfläche. Abbildung 8.5 zeigt das Ergebnis:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MyFrame extends JFrame {
    private static final String EXIT_COMMAND = "EXIT";
    private static final String CLIENT_COMMAND = "CLIENT";
    private static final String SERVER_COMMAND = "SERVER";

    public static void main(String[] args) throws Exception {
        new MyFrame().setVisible(true);
    }

    public MyFrame() throws Exception {
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        ActionListener buttonHandler = new MyFrameActionListener();

        JButton exitButton = new JButton("Exit");
        exitButton.setActionCommand(EXIT_COMMAND);
        exitButton.addActionListener(buttonHandler);

        JRadioButton serverButton = new JRadioButton("Server");
        serverButton.setActionCommand(SERVER_COMMAND);
        serverButton.addActionListener(buttonHandler);

        JRadioButton clientButton = new JRadioButton("Client");
        clientButton.setActionCommand(CLIENT_COMMAND);
        clientButton.addActionListener(buttonHandler);

        ButtonGroup clientServerGroup = new ButtonGroup();
        clientServerGroup.add(serverButton);
        clientServerGroup.add(clientButton);

        JPanel clientServerPanel = new JPanel();
        clientServerPanel.add(serverButton, BorderLayout.NORTH);
        clientServerPanel.add(clientButton, BorderLayout.SOUTH);

        this.add(clientServerPanel, BorderLayout.CENTER);
        this.add(exitButton, BorderLayout.SOUTH);
```

Abbildung 8.5: Beispiel für die Komponenten `JButton` und `JRadioButton`.

```
        pack();
    }

    private class MyFrameActionListener implements ActionListener {
        public void actionPerformed(ActionEvent ae) {
            if (EXIT_COMMAND.equals(ae.getActionCommand())) {
                System.exit(0);
            } else if (SERVER_COMMAND.equals(ae.getActionCommand())) {
                System.out.println("Server selected");
            } else if (CLIENT_COMMAND.equals(ae.getActionCommand())) {
                System.out.println("Client selected");
            }
        }
    }
}
```

8.3.7 Die Komponente JComboBox

[105] Es ist unter Umständen sinnvoller, dem Benutzer klare Optionen zur Auswahl anzubieten, als seine Auswahl selbst eingeben zu lassen (ein fehleranfälliger Ansatz). Die Komponente `JComboBox` erzeugt ein einfaches Element, das beim Anklicken nach unten aufklappt und eine Liste von Optionen anzeigt, aus denen der Benutzer wählen kann.

[106] Sind die Listeneinträge bereits bekannt, so besitzt die Klasse `JComboBox` Konstruktoren, denen Sie die initialen Einträge übergeben können. Alternativ können Sie mit Hilfe der Methoden `addItem()` und `removeItem()` dynamisch Listeneinträge hinzufügen beziehungsweise entfernen.

[107] Analog zu `JButton` und `JRadioButton` kann auch jeder `JComboBox`-Komponente ein Ereignisbehandler (ein Objekt vom Typ *`ActionListener`*) zugewiesen werden. Sie können den Ereignisbehandler verwenden, um unmittelbar nach der Auswahl einer Option durch den Benutzer eine Aktion auszulösen. Andererseits ist nicht immer ein Ereignisbehandler nötig. Eventuell genügt es, die gewählte Option auszuwerten, nach dem der Benutzer eine andere Aktion ausgeführt hat, indem Sie per `getSelectedIndex()` den Index der gewählten Option beziehungsweise per `getSelectedItem()` die Option selbst abfragen (enthalten in einem Objekt vom Typ `Object`).

[108] Die beiden Beispiele im nächsten Unterabschnitt führen die Verwendung einer `JComboBox`-Komponenten vor. Abbildung 8.6 zeigt das Ergebnis.

8.3.8 Die Klasse BorderFactory

[109] Alle bis jetzt vorgestellten Komponenten lagen gleichberechtigt in ein und demselben Fenster, das heißt, es gab keine logische Trennung zwischen einer Komponenten (oder eine Gruppe von Komponenten) und den übrigen Komponenten.

[110] In manchen Situation ist es angebracht, einen Rahmen um eine Gruppe von Komponenten zu zeichnen. Eines der häufigsten Anwendungsbeispiele ist die Gruppierung von Radiobuttons oder Ankreuzfeldern (*check boxes*) mit Hilfe eines Rahmens. Wir haben im vorigen Unterabschnitt bewußt keinen Rahmen um die Radiobuttons angelegt, um zu betonen, daß ein Rahmen beim Benutzer lediglich den *Eindruck* hervorruft, daß die Elemente in logischer Hinsicht zusammengehören. Es ist allerdings möglich, einen Rahmen um Komponenten anzulegen, die nicht logisch zusammengehören, das heißt, das alleinige Einrahmen definiert *keinen* logischen Zusammenhang.

[111] Im Gegensatz zu den meisten Swing-Komponenten wird ein Rahmen nicht direkt erzeugt. Wir rufen statt dessen eine statische Methode der Klasse `javax.swing.BorderFactory` auf, um einen Rahmen zu erzeugen, den wir anschließend in einer `JPanel`- oder `JFrame`-Komponente verwenden.

Bemerkung: Das von `BorderFactory` implementierte Entwurfsmuster *Factory* tritt häufig auf, wenn viele ähnliche Objekte erzeugt werden müssen, der Anwender dieser Objekte aber keine Implementierungsdetails kennen muß. Es genügt, wenn der Anwender weiß, wie er mit dem Objekt umgehen muß. Alle `create`-Methoden der Klasse `BorderFactory` erzeugen ein *`Border`*-Objekt (*`Border`* ist ein Interface und gehört zum Package `javax.swing.border`). *`Border`* ist der Basistyp aller Rahmentypen, so daß wir den erzeugten Rahmen in jeder `JPanel`- oder `JFrame`-Komponente einsetzen können, ohne uns damit auseinandersetzen zu müssen, welchen Typ ein spezifischer Rahmen hat. Außerdem brauchen wir uns keine Gedanken darüber zu machen, wie ein bestimmter Rahmen unter einem bestimmten Betriebssystem erzeugt wird (auch hierum kümmert sich die Fabrikklasse).

---

[112] Die Ausprägung des erzeugten Rahmens obliegt Ihrer Wahl. Bei Statusleisten wird häufig ein vertiefter Rahmen (*lower bevel border*) angelegt (verwenden Sie hierfür die Methode `createLoweredBevelBorder()`), bei Gruppen von Radiobuttons ein Rahmen mit Titel (*titled border*, `createTitleBorder()`) und bei Gruppen von Schaltflächen ein vertiefter oder erhabender Rahmen (*bevel border*) ohne Titel (`createBevelBorder()`).

[113] Das folgende Beispiel zeigt einen Rahmen mit Titel um eine `JComboBox`-Komponente. Abbildung 8.6 zeigt das Ergebnis. Vier aus Leerzeichen bestehende Beschriftungen wurden angebracht, um den Rahmen durch Vergrößerung des umschlossenen Bereichs deutlicher hervortreten zu lassen:

```
import java.awt.*;
import javax.swing.*;

public class MyFrame extends JFrame {
    private static final String TITLE = ”Title goes here”;

    public static void main(String[] args) throws Exception {
        new MyFrame().setVisible(true);
    }

    public MyFrame() throws Exception {
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        String[] items = {”One”, ”Two”, ”Three”, ”Four”, ”Five”};
        JComboBox choosableItems = new JComboBox(items);

        JPanel clientServerPanel = new JPanel();

        clientServerPanel.setBorder(BorderFactory.createTitledBorder(TITLE));
        clientServerPanel.add(new JLabel(”Pick a number: ”), BorderLayout.EAST);
        clientServerPanel.add(choosableItems, BorderLayout.CENTER);

        this.add(new JLabel(”     ”), BorderLayout.NORTH);
        this.add(new JLabel(”     ”), BorderLayout.SOUTH);
        this.add(new JLabel(”     ”), BorderLayout.EAST);
```

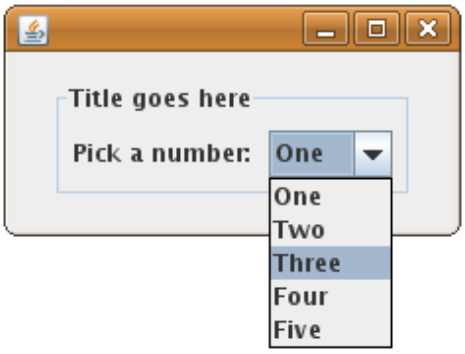

Abbildung 8.6: Beispiel für eine `JComboBox`-Komponente mit Rahmen (`BorderFactory`).

```
        this.add(new JLabel("     "), BorderLayout.WEST);
        this.add(clientServerPanel, BorderLayout.CENTER);

        pack();
    }
}
```

[114] Die Motivation für das obige Beispiel war, einen einfachen Rahmen anzulegen, ohne zu viele neue Eigenschaften auf einmal zu verwenden. Wir haben den Inhalt des Rahmens mit mehreren Beschriftungen aus Leerzeichen „aufgeblasen“, um den Rahmen besser sichtbar zu machen. Dieser „Trick“ scheidet in der Praxis natürlich aus. Statt dessen können Sie einen zusammengesetzten Rahmen (*compound border*) verwenden. Das Ergebnis ähnelt Abbildung 8.6:

```
import java.awt.*;
import javax.swing.*;

public class MyFrame extends JFrame {
    private static final String TITLE = "Title goes here";

    public static void main(String[] args) throws Exception {
        new MyFrame().setVisible(true);
    }

    public MyFrame() throws Exception {
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        String[] items = {"One", "Two", "Three", "Four", "Five"};
        JComboBox choosableItems = new JComboBox(items);

        JPanel clientServerPanel = new JPanel();

        clientServerPanel.setBorder(
            BorderFactory.createCompoundBorder(
                BorderFactory.createEmptyBorder(10, 10, 10, 10),
                BorderFactory.createTitledBorder(TITLE)));
        clientServerPanel.add(new JLabel("Pick a number: "), BorderLayout.EAST);
        clientServerPanel.add(choosableItems, BorderLayout.CENTER);
        this.add(clientServerPanel, BorderLayout.CENTER);

        pack();
    }
}
```

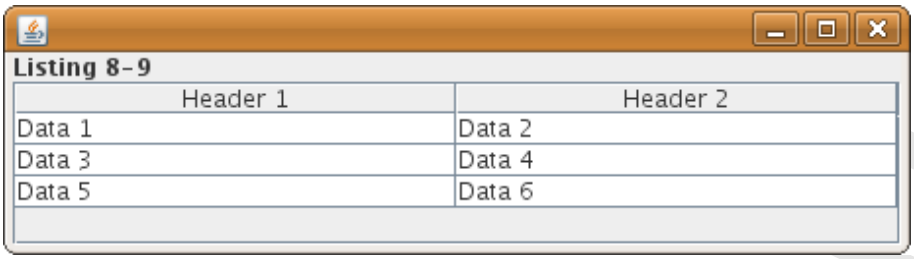

Abbildung 8.7: Beispiel für eine `JTable`-Komponente.

8.3.9 Die Komponente JTable

[115/116] Die Tabelle ist der Standardmodus zur Darstellung entsprechend strukturierter Daten. Das Tabellenschema verbindet die einfache Interpretierbarkeit mit der effizienten Darstellung vieler Informationen auf wenig Raum. Die `javax.swing.JTable`-Komponente stellt Daten in einer Tabelle dar und ist für Entwickler die optimale Kombination aus einfacher Verwendung und Erweiterbarkeit. Sie können eine `JTable`-Komponente erzeugen, indem Sie den Konstruktor der Klasse `JTable` mit zwei Arrays für die Spaltenüberschriften und die Datensätze aufrufen:

```
Object [][] rows = {{"Data 1", "Data 2"},
                    {"Data 3", "Data 4"},
                    {"Data 5", "Data 6"}};
Object [] colNames = {"Header 1", "Header 2"};
JTable table = new JTable (rows, colNames);
```

[117] Abbildung 8.7 zeigt die von diesen Zeilen implementierte Tabelle. Ist das Array mit den Spaltenüberschriften leer (zum Beispiel durch `Object[] colNames = new Object[0];`), so wird die Tabelle nicht angezeigt. Enthält das Array mit den Spaltenüberschriften nur leere Zeichenketten als Überschriften (zum Beispiel `Object[] colNames = {"", ""};`), so wird der Tabellenkörper ohne Spaltenüberschriften angezeigt. Die Spaltengröße wird zwar automatisch bestimmt, kann aber geändert werden. Die Änderungsfähigkeit der Tabellenspalten kann mit Hilfe der folgenden Methoden aktiviert beziehungsweise deaktiviert werden:

- `setSelectionMode(ListSelectionModel.SINGLE_SELECTION)`
- `setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS)`

[118] Die Tabelle im obigen Beispiel reicht aus, wenn es genügt, die Daten lediglich einmal in einer Tabelle anzuzeigen, ist aber keine sehr anpassungsfähige Lösung. Eine gute Tabellenklasse benötigt weitere Methoden und Konstruktoren, um Daten flexibel anzeigen zu können und dem Entwurfsmuster *MVC* zu entsprechen. Die Swing-Klasse `JTable` paßt so gut zu *MVC*, daß sie sogar ein eigenes Tabellenmodell verwendet, nämlich das Interface *`javax.swing.table.TableModel`*.

Bemerkung: Viele Swing-Komponenten, darunter `JTable` und `JTree` implementieren das Entwurfsmuster *MVC* eigenständig.

8.3.10 Das Interface TableModel

[119] Das Interface *`javax.swing.table.TableModel`* ist eine der einfachsten und flexibelsten Möglichkeiten, um einen Datensatz in einer `JTable`-Komponente darzustellen. Die Interaktion zwischen der Datenmodell- und der Präsentationskomponente wurde in diesem Kapitel bereits diskutiert.

Die Präsentationskomponente ist einzig und allein dafür zuständig, die in der Datenmodellkomponente enthaltenen Daten in eine visuelle Darstellungsform umzuwandeln. Diese Konvertierung wird von den Methoden der Klasse `JTable` ausgeführt. Eine Implementierung des Interfaces *`TableModel`* spielt hierbei die Rolle der Datenmodellkomponente und die `JTable`-Komponente die Rolle der Präsentationskomponente. Im wesentlichen wird der `JTable`-Komponenten eine Referenz auf das *`TableModel`*-Objekt übergeben, woraufhin sich die `JTable`-Komponente um die Wiedergabe des Datenmodells in der visuellen Darstellung kümmert. Die Referenz auf das *`TableModel`*-Objekt kann dem Konstruktor der Klasse `JTable` übergeben werden:

```
JTable table = new JTable(TableModel model);
```

Alternativ kann das Tabellenmodell mit Hilfe der Methode `setModel(TableModel model)` eingestellt werden. Nach der Übergabe des Tabellenmodells übernimmt die `JTable`-Komponente die Kontrolle.

[120/121] Die `JTable`-Komponente ist einfach zu verwenden. Die eigentliche Arbeit steckt in der Aufgabe, das Interface *`TableModel`* zu implementieren. Analog zu den Interfaces für die Ereignisbehandler sind nicht immer alle in *`TableModel`* deklarierten Methoden für jedes Datensatzformat erforderlich. Braucht ein Entwickler zum Beispiel nur eine einfache *`TableModel`*-Implementierung, so lohnt es sich nicht, Methoden wie `removeTableModelListener(TableModelListener modListener)` auszuprogrammieren. Aus diesem Grund gibt es eine Adapterklasse, deren Funktionsweise den Adapterklassen für Ereignisbehandler ähnelt. Die abstrakte Klasse `javax.swing.table.AbstractTableModel` implementiert *`TableModel`* in der Weise, daß jede deklarierte Methode eine Standardimplementierung erhält. In der Regel ist das Ableiten von `AbstractTableModel` der beste Ausgangspunkt, um eine eigene Implementierung von *`TableModel`* anzulegen.

[122] In Version 2.0 der Beispielanwendung ist die Klasse `sampleproject.gui.DvdTableModel` von `AbstractTableModel` abgeleitet. Die Methoden `getColumnClass()`, `removeTableModelListener()` und `addTableModelListener()` sind für die Beispielanwendung nicht erforderlich, so daß die Standardimplementierungen nicht überschrieben werden müssen. Alle übrigen in *`TableModel`* deklarierten Methoden werden in `DvdTableModel` überschrieben.

[123] Die Klasse `DvdTableModel` enthält zwei zusätzliche Felder `headerNames` und `dvdRecords`, um die Spaltenüberschriften und Tabellenzeilen speichern zu können:

```
private String[] headerNames = {
    "UPC", "Movie Title", "Director", "Lead Actor",
    "Supporting Actor", "Composer", "Copies in Stock"};

private List<String[]> dvdRecords = new ArrayList<String[]>(5);
```

[124] Ein Objekt der Datenmodellkomponente verhält sich wie ein Datencontainer. Die Präsentationskomponente die den Inhalt der Datenmodellkomponente in einen Präsentationsmodus konvertiert, benötigt keine Informationen darüber, wie die Daten intern strukturiert sind. Abstrakt betrachtet, kapselt eine Tabelle zwei Dinge:

- Die Spaltenüberschriften.
- Die in den einzelnen Zeilen enthaltenen Daten.

[125] Aus diesen beiden Anforderungen folgt, daß das Tabellenmodell intern zwei Kollektionstypen repräsentieren muß: Eine Kollektion für die Spaltenüberschriften sowie eine weitere Kollektion für die Datensätze (je eine Liste von Feldern). Damit bleibt die Entscheidung, welche Kollektionstypen sich am besten eignen, um diese Daten zu speichern.

[126] Die Spaltenüberschriften der Tabelle in unserer Beispielanwendung bleiben während des gesamten Lebenszyklus der Anwendung unverändert. Daher ist ein Array von `String`-Objekten ein

geeigneter Kandidat, um die Spaltenüberschriften zu speichern. Die Anzahl der Spalten in einer Zeile, also die Anzahl der Felder eines Datensatzes, ist konstant, so daß ein `DVD`-Datensatz ebenfalls als Array von `String`-Objekten dargestellt werden kann.

[127] Die Kollektion, welche die einzelnen Datensätze enthält, unterscheidet sich grundlegend von den Datenstrukturen für die Spaltenüberschriften und einzelne Datensätze, da die Anzahl der gespeicherten Datensätze variabel ist. Sucht ein Benutzer nach einer Liste von `DVD`-Datensätzen, so läßt sich die Anzahl der zurückgelieferten Datensätze nicht vorhersagen, es ist also nicht möglich, die Größe der zur Aufnahme der Datensätze erforderlichen Datenstruktur vorweg zu ermitteln. Die beste Lösung ist eine dynamische Kollektion, bei der Datensätze hinzugefügt, entfernt und in einer Schleife durchlaufen werden können. Da die Reihenfolge der Datensätze wichtig ist, scheiden *`Set`*-Kollektionen aus. Die beste Wahl ist eine dynamische Liste, zum Beispiel einer der Typen `Vector`, `LinkedList` und `ArrayList`. `Vector`-Objekte sind vorteilhaft, sofern die Synchronisierung der `Vector`-Methoden tatsächlich benötigt wird, vermitteln aber unter Umständen einen falschen Eindruck von Threadsicherheit (siehe Unterunterabschnitt 4.3.4.1), so daß wir generell davon abraten `Vector`-Objekte zu verwenden. Im vorliegenden Fall können wir auf die Threadsicherheit unseres Tabellenmodells verzichten, da es stets höchstens einen Thread gibt, der die Werte in nur einem Objekt ändern kann. Die Verwendung eines `Vector`-Objektes würde unnötige Unkosten verursachen, aber keinen zusätzlichen Nutzen stiften. Ein `LinkedList`-Objekt hat, verglichen mit einem `ArrayList`-Objekt, einige zusätzliche Methoden, etwa um am Listenanfang und -ende Elemente hinzuzufügen beziehungsweise zu entfernen, die wir allerdings nicht benötigen. Die beste Wahl zum Speichern der `DVD`-Datensätze in unserem Tabellenmodell ist ein `ArrayList`-Objekt.

[128] Unsere *`TableModel`*-Implementierung `DvdTableModel` muß einige Methoden implementieren, die von der `JTable`-Komponente aufgerufen werden, um den Inhalt der Datenmodellkomponente in einen Modus der Präsentationskomponente umzuwandeln. Das *`TableModel`*-Objekt muß der `JTable`-Komponenten beispielsweise mitteilen können, wieviele Spalten die Datensätze haben. Die Methode `getColumnCount()` liefert diese Funktionalität:

```
public int getColumnCount() {
    return this.headerNames.length;
}
```

[129] Im vorliegenden Fall ist die Anzahl der Spalten stets identisch mit der Anzahl der Spaltenüberschriften, so daß die Methode einfach die Länge des Arrays `headerNames` zurückgeben kann. Ein `DvdTableModel`-Objekt muß auch die Überschrift (den Namen) jeder einzelnen Spalte zurückgeben können. Die Methode `getColumnName()` erwartet einen Spaltenindex und gibt einen `String` zurück:

```
public String getColumnName (int column) {
    return headerNames[column];
}
```

[130] Ein `DvdTableModel`-Objekt muß eine Abfrage- und eine Änderungsmethode für den Inhalt eines Feldes in einem bestimmten Datensatz besitzen. Die entsprechenden Methoden `getValueAt()` und `setValueAt()` sind:

```
public Object getValueAt(int row, int column) {
    String[] rowValues = this.dvdRecords.get(row);
    return rowValues[column];
}

public void setValueAt(Object obj, int row, int column) {
    Object[] rowValues = this.dvdRecords.get(row);
    rowValues[column] = obj;
}
```

[131] Beachten Sie die folgende alternative Schreibweise für die Methode `getValueAt()`:

```
public Object getValueAt(int row, int column) {
    return this.dvdRecords.get(row)[column];
}
```

[132] Falls Sie die Versuchung spüren, eine solche Syntaxkonstruktion zu verwenden, so rechtfertigen Sie stets vor sich selbst, welchen Vorteil Sie sich davon versprechen. Ist das Erzeugen beziehungsweise Zerstören der *Referenzvariablen* `rowValues`, die auf das Array verweist, so teuer, daß sich dieser „Kunstgriff“ wirklich auszahlt?

[133] Die Methode `getRowCount()` gibt die Anzahl der im `DvdTableModel`-Objekt gekapselten Datensätze zurück. Die Anzahl der Datensätze ist die Größe (Länge) des `ArrayList<DVD>`-Objektes:

```
public int getRowCount() {
    return this.dvdRecords.size();
}
```

[134] Die Methode `isCellEditable()` unserer `DvdTableModel`-Klasse gibt an, ob der Inhalt einer Zelle geändert werden kann oder nicht. Bei unserem Tabellenmodell ist keine Zelle editierbar, so das die Methode stets `false` zurückgibt:

```
public boolean isCellEditable(int row, int column) {
    return false;
}
```

Hätte unserer Tabellenmodell editierbare Zellen, so würde `isCellEditable()` die übergebenen Zeilen- und Spaltenindizes auswerten, um festzustellen ob, die entsprechende Zelle editierbar ist oder nicht.

Bemerkung: Im Gegensatz zu den anderen Methoden, die wir bis jetzt besprochen haben, ist `isCellEditable()` bereits in `AbstractTableModel` implementiert. Die Version aus der Klasse `AbstractTableModel` liefert dieselbe Funktionalität wie unsere überschriebene Methode. Wir hätten die Methode also nicht überschreiben müssen, haben uns aber für die Implementierung entschieden, um Ihnen das Experimentieren mit dieser Methode zu erleichtern.

---

[135] Schließlich verfügt unsere `DvdTableModel`-Klasse der Bequemlichkeit halber noch über zwei Methoden, die nicht im Interface *`TableModel`* deklariert sind. `DVD`-Objekte werden zwar überall in der Beispielanwendung eingesetzt, haben aber nicht das von der Präsentationskomponente erwartete Format. In der Beispielanwendung entspricht die `JTable`-Komponente der Präsentations- und das `DvdTableModel`-Objekt der Datenmodellkomponente. Die Klasse `GuiController` ist dafür zuständig, ein `DVD`-Objekt (oder eine Kollektion von `DVD`-Objekten) in ein `DvdTableModel`-Objekt zu konvertieren. Die Klasse `DvdTableModel` besitzt die beiden folgenden `addDvdRecord()`-Methoden mit unterschiedlichen Signaturen:

```
public void addDvdRecord(String upc, String name, String director,
                         String leadActor, String supportingActor,
                         String composer, int numberOfCopies) {
    String[] temp = {upc, name, director, leadActor, supportingActor,
                     composer, Integer.toString(numberOfCopies)};
    this.dvdRecords.add(temp);
}

public void addDvdRecord(DVD dvd) {
    addDvdRecord(dvd.getUPC(), dvd.getName(), dvd.getDirector(),
                 dvd.getLeadActor(), dvd.getSupportingActor(),
                 dvd.getComposer(), dvd.getCopy());
}
```

[136] Beide `addDvdRecord()`-Methoden erwarten entweder ein `DVD`-Objekt oder die äquivalenten Einzeldaten und fügen einen neuen Datensatz an die entsprechende Strukur in `DvdTableModel` an (`dvdRecords`, Seite 233). Die erste Methode erwartet die Felder eines `DVD`-Objektes als Einzeldaten, die zweite Methode dagegen ein `DVD`-Objekt. Das `GuiController`-Objekt ruft lediglich eine der beiden `addDvdRecord()`-Methoden auf, um dem `DvdTableModel`-Objekt ein weiteres `DVD`-Objekt hinzuzufügen und umgeht die Umwandlung im `GuiController`-Objekt selbst.

Tipp: Die hier vorgestellte Version der Klasse `DvdTableModel` kann per `addDvdRecord()` stets ein `DVD`-Objekt auf einmal konvertieren. Hin und wieder ist es bequemer, eine Methode zu haben, die eine ganze Kollektion von `DVD`-Objekten umwandelt. Dies würde einer Suchmethode gestatten, alle Suchergebnisse mit Hilfe eines einzigen Methodenaufrufs zu konvertieren.

8.3.11 Die Verbindung zwischen Tabellenmodell und Tabellenkomponente

[137/138] Ist das Interface *`TableModel`* einmal implementiert, so ist die Verknüpfung eines Objektes dieser Implementierung mit einer `JTable`-Komponente nur noch ein kleiner Schritt. Das Tabellenmodell kann beispielsweise dem Konstruktor der Klasse `JTable` übergeben werden. Der folgende Konstruktoraufruf erzeugt eine `JTable`-Komponente zusammen mit dem Tabellenmodell `DvdTableModel` aus dem vorigen Unterabschnitt:

```
JTable table = new JTable(new DvdTableModel());
```

Die Tabelle ist noch leer, da das `DvdTableModel`-Objekt noch keine Datensätze enthält.

Bemerkung: Das Tabellenmodell einer `JTable`-Komponente kann nach der Objekterzeugung modifiziert und sogar ausgetauscht werden. Die `JTable`-Methode `setModel()` weist ihrem `JTable`-Objekt ein neues Tabellenmodell zu und bewirkt, daß die Anzeige der `JTable`-Komponente aktualisiert wird. Die `JTable`-Methode `getModel()` gibt eine Referenz auf das aktuelle Tabellenmodell zurück, in unserem Fall also auf ein `DvdTableModel`-Objekt. Sie können dem Tabellenmodell mittels dieser Referenz weitere Datensätze hinzufügen, in unserem Fall mit Hilfe der Methode `addDvdRecord()`.

[139/140] Da jede Änderung am Tabellenmodell einer `JTable`-Komponente eine Aktualisierung der Präsentationskomponente bewirkt, muß der Client so konstruiert werden, daß dieser Vorteil genutzt wird. Die Klasse `sampleproject.gui.MainWindow` verfügt über ein privates Feld namens `tableData`, welches stets das Tabellenmodell der `JTable`-Komponente referenziert:

```
private DvdTableModel tableData;
```

Da die Datenübertragung zwischen Präsentations- und Programmsteuerungskomponente durch das `DvdTableModel`-Objekt verläuft, repräsentiert das vom `tableData`-Feld referenzierte Tabellenmodell stets den aktuellen Zustand der Datenbankdatei.

[141] Nach jeder Aktualisierung oder Anfrage an die Datenbankdatei wird eine Referenz auf das aktuelle Tabellenmodell im `tableData`-Feld gespeichert. Nach jedem Aufruf der Programmsteuerungskomponente ruft das `MainWindow`-Objekt seine private `setupTable()`-Methode auf:

```
private void setupTable() {
    // Preserve the previous selection
    int index = mainTable.getSelectedRow();
    String prevSelected = (index >= 0)
                        ? (String) mainTable.getValueAt(index, 0)
```

```
                        : "";

        // Reset the table data
        this.mainTable.setModel(this.tableData);

        // Reselect the previous item if it still exists
        for (int i = 0; i < this.mainTable.getRowCount(); i++) {
            String selectedUpc = (String) mainTable.getValueAt(i, 0);
            if (selectedUpc.equals(prevSelected)) {
                this.mainTable.setRowSelectionInterval(i, i);
                break;
            }
        }
    }
```

[142] Die `MainWindow`-Methode `setupTable()` aktualisiert das Datenmodell der `JTable`-Komponente des `MainWindow`-Objektes durch Aufrufen der `JTable`-Methode `setModel()`. Nachdem das Datenmodell ersetzt wurde, wird die Präsentationskomponente aktualisiert und zeigt den geänderten Datensatz an. Falls die `JTable`-Komponente ihren Inhalt nicht aktualisiert, können Sie die Methode `updateUI()` aufrufen, um die Aktualisierung zu erzwingen.

[143] Beachten Sie, daß `setupTable()` vor dem Aktualisieren des Tabellenmodells die zuletzt selektierte Zeile speichert. Nach der Aktualisierung des Tabellenmodells durchläuft `setupTable()` die neue Liste von Datensätzen und lokalisiert die zuletzt gewählte Zeile anhand des UPC-Wertes. Anschließend wird die `JTable`-Methode `setRowSelectionInterval()` aufgerufen, um die zuletzt selektierte Zeile wiederum als selektierte Zeile zu markieren.

8.3.12 Die Klasse JScrollPane

[144] Gelegentlich müssen mehr Daten auf dem Bildschirm angezeigt werden, als in das entsprechende Fenster passen. Beispielsweise kann die Datenbankdatei der Beispielanwendung soviele DVDs enthalten, daß es nicht möglich ist, sie alle gleichzeitig auf einer Bildschirmseite darzustellen.

[145] In solchen Fällen kann die Komponente, deren Inhalt zu umfangreich ist, in eine `JScrollPane`-Komponente eingebettet werden, die bei Bedarf Bildlaufleisten (*scrollbars*) einblendet, um das Verschieben des sichtbaren Bildaussschnitts zu ermöglichen. Per Voreinstellung werden die Bildlaufleisten nur bei Bedarf angezeigt. Wenn sie nicht erforderlich sind, werden sie dagegen nicht eingeblendet.

[146] Das folgende Beispiel zeigt eine `JTextArea`-Komponente mit zuviel Inhalt. Die linke Hälfte von Abbildung 8.8 zeigt das Ergebnis:

```
import javax.swing.*;

public class MyFrame extends JFrame {
    public static void main(String[] args) throws Exception {
        JFrame theFrame = new JFrame();
        theFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        String outputString = "";
        for (int i = 0; i < 100; i++) {
            outputString += "The quick brown fox jumped over the lazy dog. ";
        }
        JTextArea textDisplay = new JTextArea(20, 60);
        textDisplay.setLineWrap(true);
        theFrame.add(textDisplay);
        theFrame.pack();
        textDisplay.setText(outputString);
```

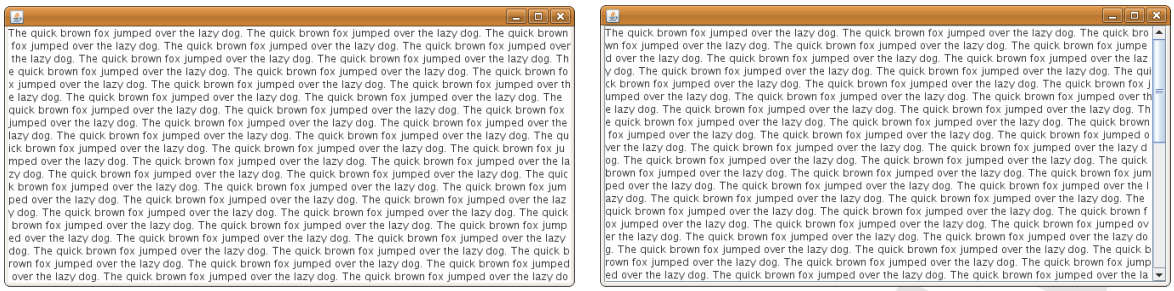

Abbildung 8.8: Links: Beispiel für eine `JTextArea`-Komponente mit zuviel Inhalt. Rechts: Beispiel für eine `JTextArea`-Komponente mit Bildlaufleiste (`JTextArea`-Komponente eingebettet in eine `JScrollPane`-Komponente).

```
            theFrame.setVisible(true);
        }
    }
```

[147] In der Regel genügt es, die Zeile

```
theFrame.add(textDisplay);
```

durch die Zeile

```
theFrame.add(new JScrollPane(textDisplay));
```

zu ersetzen. Die rechte Hälfte von Abbildung 8.8 zeigt das Ergebnis.

8.4 Zusammensetzen der Beispielanwendung

[148–150] Es ist an der Zeit, unsere Beispielanwendung „zusammenzubauen“. Wir bewerkstelligen den Start der Beispielanwendung[1] mittels einer separaten Hilfsklasse (`ApplicationRunner`, siehe unten). Die einzige Aufgabe dieser Klasse besteht darin, den Aufruf der Beispielanwendung vorzubereiten und auszulösen. Eine solche Starterklasse initialisiert in der Regel globale Einstellungen, etwa das Look-and-Feel, und schafft die für den Betrieb der Anwendung erforderlichen Voraussetzungen. Die wichtigste Einstellung für den Client der Beispielanwendung ist das Look-and-Feel. Die Beispielanwendung verwendet das Look-and-Feel des unterliegenden Betriebssystems, damit die Benutzer eine vertraute Umgebung vorfinden. Die Starterklasse prüft außerdem, ob Kommandozeilenschalter vorhanden sind (erlaubt sind `alone`, `server` sowie der Anwendungsstart ohne Schalter), um den Betriebsmodus der Anwendung zu bestimmen.

8.4.1 Die Klasse ApplicationRunner

[151] Die Klasse `sampleproject.gui.ApplicationRunner` ist im wesentlichen ein Applikationslader. Die `main()`-Methode erzeugt lediglich ein Objekt von Typ `ApplicationRunner`. Der Konstruktor legt das Look-and-Feel der Anwendung fest und erzeugt entweder ein `MainWindow`- oder ein `ServerWindow`-Objekt:

```
public ApplicationRunner(String[] args) {
    try {
        UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
```

[1] *Anmerkung des Übersetzers*: Der Client der Beispielanwendung wird im Netzwerkmodus per `java -jar sampleproject.jar` (ohne Kommandozeilenschalter, siehe Seite 279) beziehungsweise per `java -jar sampleproject.jar` ***alone*** im *stand-alone*-Betriebsmodus (siehe Seite 277) gestartet. Der Server wird mittels `java -jar sampleproject.jar server` gestartet (siehe Seite 278).

```
        } catch (UnsupportedLookAndFeelException uex) {
            log.warning("Unsupported look and feel specified");
        } catch (ClassNotFoundException cex) {
            log.warning("Look and feel could not be located");
        } catch (InstantiationException iex) {
            log.warning("Look and feel could not be instantiated");
        } catch (IllegalAccessException iaex) {
            log.warning("Look and feel cannot be used on this platform");
        }

        if (args.length == 0 || "alone".equalsIgnoreCase(args[0])) {
            // Create an instance of the main application window
            new MainWindow(args);
        } else if ("server".equalsIgnoreCase(args[0])) {
            new ServerWindow();
        } else {
            log.info("Invalid parameter passed in startup: " + args[0]);
            // Logging may be turned off, or may be going to a file, so
            // send usage information to the error output (usually the screen).
            System.err.println("Command line options may be one of:");
            System.err.println("\"server\" - starts server application");
            System.err.println("\"alone\"  - starts non-networked client");
            System.err.println("\"\"\"       - (no command line option): "
                                + "networked client will start");
        }
    }
```

[152] Die Klasse `ApplicationRunner` definiert außerdem die statische Methode `handleException()`. Die Methode erwartet ein Argument vom Typ `String` (die Fehlermeldung) und präsentiert dem Benutzer ein Dialogfenster, das die Fehlermeldung anzeigt (siehe Abbildung 8.11, rechts, Seite 246). Die Methode `handleException()` existiert ausschließlich zu dem Zweck, den Benutzer während des Lebenszyklus der Anwendung über Fehler informieren zu können. Alle Ausnahmen in der Klasse `MainWindow` werden abgefangen und in der jeweiligen `catch`-Klausel der `handleException()`-Methode übergeben, um die Fehlermeldung anzuzeigen.

8.4.2 Das Clientfenster

[153] Die Logik des Clients der graphischen Benutzeroberfläche befindet sich zum größten Teil in der Klasse `sampleproject.gui.MainWindow`, die wir in diesem Unterabschnitt vorstellen.

8.4.2.1 Design und Layout

[154] Bereits in Kapitel 2 und nochmals am Anfang dieses Kapitels wurde empfohlen, Ihre graphische Benutzeroberfläche mit Papier und Bleistift zu skizzieren. Abbildung 8-22, Seite 263 (Buch), zeigt unsere Skizze für den Client der graphischen Benutzeroberfläche der Beispielanwendung.

[155] Das Skizzieren eines Prototyps der graphischen Benutzeroberfläche hat, verglichen mit der direkten Entwicklung die folgenden Vorteile:

- Eine Skizze läßt sich schneller anfertigen, als eine graphische Benutzeroberfläche.
- Wenn der Benutzer eine Änderung wünscht, müssen Sie keinen Quelltext verwerfen.
- Eine Skizze auf Papier kann überall präsentiert werden, da kein Computer erforderlich ist.

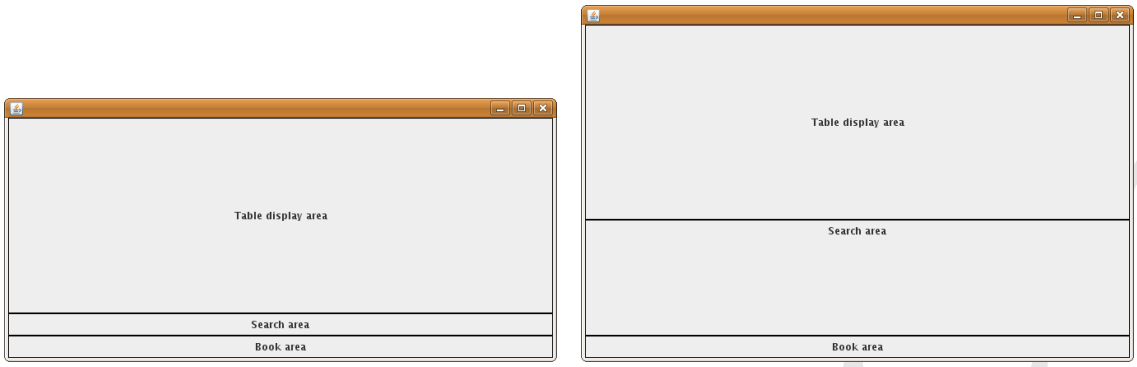

Abbildung 8.9: Links: Die drei Hauptbereiche der graphischen Benutzeroberfläche des Clients (Tabellenanzeige-, Such- und Buchungsbereich.). Rechts: Auswirkung bei Änderung der Fenstergröße. Die Tabelle ist im Bereich `NORTH`, der Suchbereich im Bereich `CENTER` und der Buchungbereich im Bereich `SOUTH` angeordnet.

- Wenn Sie den potentiellen Benutzern die Skizze zeigen, *wissen* sie, daß es sich lediglich um einen Entwurf handelt. Wenn Sie den potentiellen Benutzern dagegen eine Attrappe der graphischen Benutzeroberfläche zeigen, neigen die Benutzer zu dem Irrtum, die Entwicklung der Anwendung sei bereits zu einem großen Teil abgeschlossen (siehe Warnung, Seite 208).
- Eine Vorentscheidung dahingehend, was gut aussieht, wird in der Regel beibehalten. Beim Entwickeln ohne vorherige Skizze besteht die Versuchung, Teile des Designs zu ändern, wenn sich das Programmieren als zu kompliziert herausstellt.
- Existiert eine Vorentscheidung dahingehend, aus welchen Komponenten die graphische Benutzeroberfläche besteht, so ist die Versuchung, beim Programmieren zusätzliche Eigenschaften oder Fähigkeiten zu implementieren, weniger stark.

[156] Abbildung 8-22, Seite 263 (Buch) dokumentiert *unseren Entwurf* für die graphische Benutzeroberfläche des Clients der Beispielanwendung. Eventuell wählen Sie bei Ihrer Prüfungsaufgabe ein völlig anderes Layout. Die graphische Benutzeroberfläche unserer Beispielanwendung implementiert ein schlichtes Layout mit wenig zusätzlichem „Schnickschnack“. Sun Microsystems verlangt in der Prüfung zum *Sun Certified Java Developer* nicht mehr als eine grundlegende Swing-Oberfläche, wobei Sie aber selbst entscheiden können, ob Sie über die Anforderungen hinaus, zusätzliche Eigenschaften und Fähigkeiten implementieren wollen. Beispielsweise könnte eine `JToolBar`-Komponente die Bedienbarkeit der Anwendung verbessern, wird aber für das Bestehen der Prüfung nicht namentlich verlangt. Entscheiden Sie selbst, ob Sie solche Eigenschaften und Fähigkeiten zusätzlich implementieren oder nicht. Einerseits gibt es viele Dinge, durch die Ihre Benutzeroberfläche anwenderfreundlicher und die Bewertung Ihrer Prüfungsaufgabe folglich verbessert wird. Andererseits kann die Anleitung Ihrer Aufgabe ausdrücklich davor warnen, über die Spezifikation hinauszugehen. Entscheiden Sie selbst, wo Sie die Trennlinie ziehen.

[157/158] Die Skizze in Abbildung 8-22 weist drei Bereiche aus (ausgenommen, die Titel- und die Menüleiste), nämlich einen Tabellenanzeige-, einen Such- und einen Buchungsbereich. Abbildung 8.9 (links) zeigt diese drei Bereiche. Die naheliegende Lösung, den Layoutmanager `BorderLayout` für die Anordnung der drei Bereiche zu wählen, scheidet aus. Ändert der Benutzer nämlich in diesem Fall die Fenstergröße, so dehnen sich zwar der Tabellenanzeigebereich (oben, „nördlich“) und der Buchungsbereich (unten, „südlich“) nur horizontal aus. Der im Zentrum platzierte Suchbereich dehnt sich aber auch vertikal, so daß das Fenster unpraktisch verformt wird, siehe Abbildung 8.9 (rechts).

Bemerkung: Das Skizzieren der graphischen Benutzeroberfläche vor der eigentlichen Entwicklung gewährleistet, daß der Benutzer oder Kunde die Oberfläche gemäß seinen Wünschen erhält und ver-

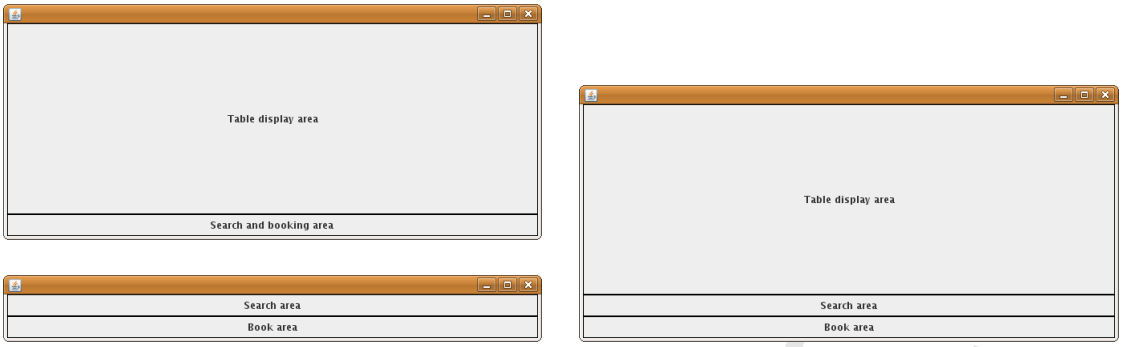

Abbildung 8.10: Oben: Die beiden neuen Bereiche des Hauptfensters. Unten: Der neue Bereich für die Such- und die Buchungskomponente. Rechts: Kombination der neuen Bereiche.

hindert, daß wir die für uns einfachste Lösung wählen. Ohne eine solche Skizze könnten wir versucht sein, das vom Benutzer gewünschte Design zu ändern, um uns die Arbeit zu erleichtern.

---

[159] Wir müssen also die Aufteilung der graphischen Benutzeroberfläche ändern. Da der größte Anteil des Fensterinhaltes in der Tabelle dargestellt wird, platzieren wir die Tabellenkomponente im Zentrum. Wir brauchen einen zusätzlichen Bereich, der die Such- und die Buchungskomponente enthält und unterhalb („südlich“) des Tabellenbereiches liegt, siehe Abbildung 8.10 (oben).

[160] Wir erzeugen eine zweite `JPanel`-Komponente, in der wir die Such- und die Buchungskomponente anlegen. Da sich die Größe dieser beiden Bereiche bei einer Änderung der Fenstergröße nicht zu ändern braucht, platzieren wir die Suchkomponente im oberen („nördlichen“) und die Buchungskomponente im unteren („südlichen“) Bereich, siehe Abbildung 8.10 (unten).

[161] Wir können nun die `JPanel`-Komponente aus Abbildung 8.10 (unten) in Abbildung 8.10 (oben) unterhalb des Tabellenbereiches platzieren, so daß sich die Größe des Tabellenbereiches des Fensters anpassen kann, während der Such- und der Buchungsbereich wie beabsichtigt ihre Abmessungen beibehalten. Das folgende Beispiel demonstriert diese Lösung. Anstelle der tatsächlichen Komponenten wurden Beschriftungen verwendet, um den Quelltext leichter verständlich zu machen:

```
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;

public class MyFrame extends JFrame {
    public static void main(String[] args) throws Exception {
        Border border = BorderFactory.createLineBorder(Color.BLACK);
        JFrame theFrame = new JFrame();
        theFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // This is the panel for the table - that is all it contains
        JPanel tablePanel = new JPanel();
        tablePanel.setBorder(border);
        JLabel table = new JLabel("Table display area", SwingConstants.CENTER);
        table.setPreferredSize(new Dimension(650, 225));
        tablePanel.add(table);

        // The search options panel
        JPanel searchPanel = new JPanel();
        searchPanel.setBorder(border);
        JLabel search = new JLabel("Search area", SwingConstants.CENTER);
        search.setPreferredSize(new Dimension(650, 15));
        searchPanel.add(search);
```

```
            // The booking options panel
            JPanel bookPanel = new JPanel();
            bookPanel.setBorder(border);
            JLabel book = new JLabel("Book area", SwingConstants.CENTER);
            book.setPreferredSize(new Dimension(650, 15));
            bookPanel.add(book);

            // The search & booking options panels are both added to an extra panel
            JPanel optionsPanel = new JPanel(new BorderLayout());
            optionsPanel.add(searchPanel, BorderLayout.NORTH);
            optionsPanel.add(bookPanel, BorderLayout.SOUTH);

            // The tablePanel and optionsPanel are added to the JFrame
            theFrame.add(tablePanel, BorderLayout.CENTER);
            theFrame.add(optionsPanel, BorderLayout.SOUTH);

            theFrame.pack();
            theFrame.setVisible(true);
        }
    }
```

[162] Das obige Beispiel zeigt in groben Zügen den Aufbau des Hauptfensters (Klasse `MainWindow`) des Clients unserer Beispielanwendung. Wir geben im folgenden Unterunterabschnitt den vollständigen Quelltext der Klasse `MainWindow` wieder. Arbeiten Sie das Beispiel dennoch durch, damit Sie mit der Vorgehensweise vertraut sind, bevor Sie weiterlesen. Abbildung 8.10 (rechts) zeigt das mit dem obigen Beispiel erzeugte Fenster.

[163] Der Such- und der Buchungsbereich enthalten jeweils mehrere in einer Reihe angeordnete Komponenten. Diese Anordnung läßt sich mit Hilfe des Layoutmanagers `FlowLayout` bewerkstelligen, wie der Konstruktor der Klasse `DvdScreen` (innere Klasse von `MainWindow`) im nächsten Unterunterabschnitt dokumentiert (siehe Seite 245).

Tipp: Das Entwickeln kleiner Abschnitte, um einen bestimmten Aspekt auszuprobieren, ist sowohl für die Entwicklungspraxis als auch bei der Fehlersuche ein hilfreiches Verfahren. Wenn Sie versuchen, eine Eigenschaft oder Fähigkeit zu implementieren und sich das Programm nicht erwartungsgemäß verhält, ist bei der Fehlersuche häufig ein großer Teil des Quelltextes nicht relevant. Wenn Sie nur ein einfaches Testprogramm schreiben, um diese Eigenschaft oder Fähigkeit auszuprobieren, stehen Ihnen keine irrelevanten und störenden Anweisungen im Weg. Insbesondere haben Sie ein übersichtliches Stück Quelltext, mit dem Sie einen Freund oder Kollegen nötigenfalls um Hilfe bitten können. Wenn Sie bei einer Anwendung mit tausend Zeilen Quelltext um Hilfe bitten, strapazieren Sie die Freundschaft unnötig.

8.4.2.2 Die Klasse MainWindow

[164] Die Klasse `MainWindow` ist von `JFrame` abgeleitet und repräsentiert die Implementierung des Hauptfensters der Beispielanwendung. Der Konstruktor von `MainWindow` konfiguriert die Menüleiste der Beispielanwendung, zeigt die Tabelle an, verknüpft ein `DvdScreen`-Objekt (siehe Seite 244) mit dem Hauptfenster und platziert es in der Bildschirmmitte.

[165] Der Konstruktor der Klasse `MainWindow` erzeugt zunächst ein Objekt der Basisklasse `JFrame`, wobei der Titel des Hauptfensters der Beispielanwendung eingestellt wird. Danach öffnet der Konstruktor ein Dialogfenster, in dem der Benutzer angibt, wo sich die Datenbankdatei befindet (siehe hierzu den folgenden Unterabschnitt 8.4.3). Anschließend werden die Menüleiste, die Menüs und die Menüeinträge erzeugt.

[166] Die Klasse `MainWindow` besitzt eine `JMenuBar`-Komponente (Menüleiste, siehe unten). Mit der `JMenuBar`-Komponente können mehrere `JMenu`-Komponenten (Menüs) verknüpft werden, zum Beispiel eine für das „Datei“- und eine für das „Hilfe“-Menü. Eine `JMenu`-Komponente kann wiederum mehrere `JMenuItem`-Komponenten (Menüpunkte) referenzieren (je eine pro Aktion, die der Benutzer über das entsprechende Menü aufruft).

[167] Jedem Menü und jedem Menüpunkt kann eine mnemonische Tastenkombination und ein Icon zugeordnet werden. Im folgenden Beispiel wird dem „Datei“-Menü der Buchstabe „D“ (im englischen Original „F“ für „File“) und dem Menüpunkt „Beenden“ der Buchstabe „B“ (im Original „Q“ für „Quit“) zugewiesen. Wählt der Benutzer die Tastenkombination „Alt“ + „D“, so öffnet sich das „Datei“-Menü. Wählt der Benutzer anschließend die Tastenkombination „Alt“ + „B“, so wird die Anwendung beendet.

[168] In der Regel wird jeder Menüpunkt (`JMenuItem`-Komponente) mit einem Ereignisbehandler vom Typ *`ActionListener`* verknüpft, um auf Ereignisse reagieren zu können. Das folgende Beispiel verknüpft den Menüpunkt „Beenden“ mit einem Objekt der innneren Klasse `QuitApplication` (siehe Seite 244).

[169] Nach der Konfiguration der Menüs und dem Laden der Daten aus der Datenbankdatei wird ein `DvdScreen`-Objekt mit dem `MainWindow`-Objekt verknüpft. Die innere Klasse `DvdScreen` ist von `JPanel` abgeleitet und enthält die im Unterunterabschnitt 8.4.2.1 beschriebenen Elemente (siehe Seite 244).

[170] Schließlich werden die Startabmessungen des Hauptfensters eingestellt und das Fenster in der Bildschirmmitte platziert:

```
public MainWindow(String[] args) {
    super(”Denny’s DVDs”);
    this.setDefaultCloseOperation(this.EXIT_ON_CLOSE);

    ApplicationMode connectionType = (args.length == 0)
        ? ApplicationMode.NETWORK_CLIENT
        : ApplicationMode.STANDALONE_CLIENT;

    // find out where our database is
    DatabaseLocationDialog dbLocation =
        new DatabaseLocationDialog(this, connectionType);

    if (dbLocation.userCanceled()) {
        System.exit(0);
    }

    try {
        controller = new GuiController(dbLocation.getNetworkType(),
                                       dbLocation.getLocation(),
                                       dbLocation.getPort());
    } catch (GuiControllerException gce) {
        ApplicationRunner.handleException(”Failed to connect to the database”);
    }

    // Add the menu bar
    JMenuBar menuBar = new JMenuBar();
    JMenu fileMenu = new JMenu(”File”);
    JMenuItem quitMenuItem = new JMenuItem(”Quit”);
    quitMenuItem.addActionListener(new QuitApplication());
    quitMenuItem.setMnemonic(KeyEvent.VK_Q);
    fileMenu.add(quitMenuItem);
    fileMenu.setMnemonic(KeyEvent.VK_F);
    menuBar.add(fileMenu);
```

```
        this.setJMenuBar(menuBar);

        // A full data set is returned from an empty search
        try {
            tableData = controller.getDvds();
            setupTable();
        } catch (GuiControllerException gce) {
            ApplicationRunner.handleException("Failed to acquire an initial DVD list."
                                              + "\nPlease check the DB connection.");
        }

        this.add(new DvdScreen());

        this.pack();
        this.setSize(650, 300);

        // Center on screen
        Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
        int x = (int) ((d.getWidth() - this.getWidth())/2);
        int y = (int) ((d.getHeight() - this.getHeight())/2);
        this.setLocation(x, y);
        this.setVisible(true);
    }
```

[171] Wählt der Benutzer den Menüpunkt „Beenden“, um die Beispielanwendung zu beenden, so wird die `actionPerformed()`-Methode eines Objektes der Ereignisbehandlerklasse `QuitApplication` aufgerufen. Die Klasse `QuitApplication` ist denkbar einfach. Die `actionPerformed()`-Methode ruft lediglich `System.exit(0)` auf:

```
    private class QuitApplication implements ActionListener {
        public void actionPerformed(ActionEvent ae) {
            System.exit(0);
        }
    }
```

[172/173] Das nächste Beispiel zeigt den Konstruktor der inneren Klasse `DvdScreen`. Die Hauptkomponenten wurden in den vorigen Abschnitten eingeführt. Der Konstruktor von `DvdScreen` legt zunächst, zentriert im Tabellenbereich, einen Bereich mit Bildlaufleiste (`tableScroll`, `JScrollPane`-Komponente) an, der später die Tabelle mit den DVD-Datensätzen enthalten wird.

[174] Anschließend wird eine `JPanel`-Komponente für die Suchkomponente angelegt (`searchPanel`) und mit dem Texteingabefeld zum Erfassen der Suchparameter aus `MainWindow` (`searchField`, `JTextField`-Komponente) sowie einer Schaltfläche (`searchButton`, `JButton`-Komponente) zum Starten der Suche verknüpft. Die Schaltfläche wird mit einem Ereignisbehandler vom Typ `SearchDVD` verbunden, dessen Implementierung wir im Anschluß an das folgende Beispiel besprechen (siehe Seite 246), da seine Aufgabe etwas komplizierter ist, als beim obigen Ereignisbehandler für den Menüpunkt „Beenden“.

[175/176] Die `JButton`-Komponenten `rentButton` und `returnButton` zum Ausleihen beziehungsweise Zurückgeben einer DVD werden zusammen mit ihren Ereignisbehandlern vom Typ `RentDVD` beziehungsweise `ReturnDVD` sowie die `JPanel`-Komponente `hiringPanel` angelegt. Ein Unterbereich (`bottomPanel`, `JPanel`-Komponente) wird angelegt, der die Such- (oben) und die Buchungskomponente (unten) enthält. Der Unterbereich wird anschließend in den unteren („südlichen“) Teil des Hauptbereiches eingefügt. Schließlich wird die Tabelle konfiguriert und einige Tooltips angelegt:

```
    public DvdScreen() {
        this.setLayout(new BorderLayout());
        JScrollPane tableScroll = new JScrollPane(mainTable);
        tableScroll.setSize(500, 250);
```

```
        this.add(tableScroll, BorderLayout.CENTER);

        // Set up the search pane
        JButton searchButton = new JButton("Search");
        searchButton.addActionListener(new SearchDVD());
        searchButton.setMnemonic(KeyEvent.VK_S);
        // Search panel
        JPanel searchPanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
        searchPanel.add(searchField);
        searchPanel.add(searchButton);

        // Setup rent and return buttons
        JButton rentButton = new JButton("Rent DVD");
        JButton returnButton = new JButton("Return DVD");

        // Add the action listenters to rent and return buttons
        rentButton.addActionListener(new RentDVD());
        returnButton.addActionListener(new ReturnDVD());
        // Set the rent and return buttons to refuse focus
        rentButton.setRequestFocusEnabled(false);
        returnButton.setRequestFocusEnabled(false);
        // Add the keystroke mnemonics
        rentButton.setMnemonic(KeyEvent.VK_R);
        returnButton.setMnemonic(KeyEvent.VK_U);
        // Create a panel to add the rental a remove buttons
        JPanel hiringPanel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
        hiringPanel.add(rentButton);
        hiringPanel.add(returnButton);

        // bottom panel
        JPanel bottomPanel = new JPanel(new BorderLayout());
        bottomPanel.add(searchPanel, BorderLayout.NORTH);
        bottomPanel.add(hiringPanel, BorderLayout.SOUTH);

        // Add the bottom panel to the main window
        this.add(bottomPanel, BorderLayout.SOUTH);

        // Set table properties
        mainTable.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        mainTable.setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS);
        mainTable.setToolTipText("Select a DVD record to rent or return.");

        // Add Tool Tips
        returnButton.setToolTipText("Return the DVD item selected in the above table.");
        rentButton.setToolTipText("Rent the DVD item selected in the above table.");
        searchField.setToolTipText("Enter infromation about a DVD you want to locate.");
        searchButton.setToolTipText("Submit the DVD search.");
    }
```

[177] Abbildung 8.11 (links) zeigt die von `MainWindow` und `DvdScreen` erzeugte graphische Benutzeroberfläche.

[178] Die Methode `actionPerformed()` der inneren Ereignisbehandlerklasse `SearchDVD` muß, wie alle Ereignisbehandlerklassen der Beispielanwendung, die mit `GuiController` kommunizieren, in der Lage sein, eine vom diesem ausgeworfene Ausnahme (`GuiControllerException`) abzufangen und zu behandeln. Die `SearchDVD`-Methode `actionPerformed()` untersucht die abfangene Ausnahme aber etwas genauer. Eine von der `GuiController`-Methode `find(String query)` ausgeworfene `GuiControllerException` kann von einer (in ihr enthaltenen) `java.util.regex.PatternSyntaxException` verursacht worden sein. Eine Ausnahme vom Typ `PatternSyntaxException` wird ausgeworfen, wenn der Benutzer keinen syntaktisch korrekten regulären Ausdruck eingegeben hat. Die in

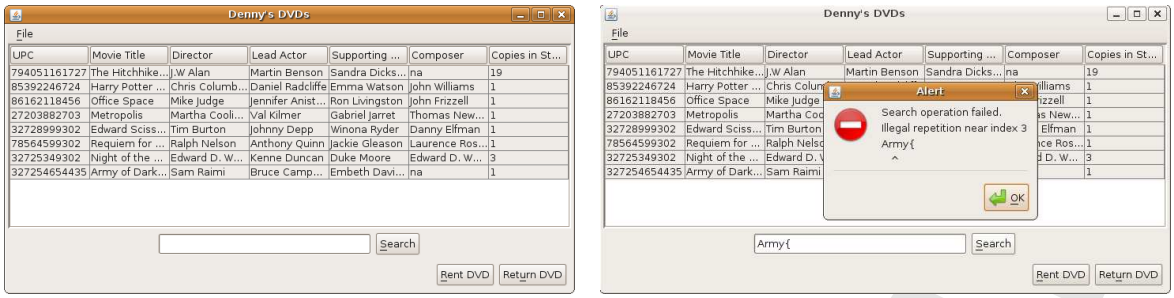

Abbildung 8.11: Links: Das von den Klassen `MainWindow` und `DvdScreen` erzeugte Hauptfenster der Beispielanwendung. Rechts: Dialogfenster mit Fehlermeldung über einen fehlerhaften regulären Ausdruck.

dem Ausnahmeobjekt enthaltene Nachricht kann in diesem Fall für den Benutzer wichtige Informationen über den Syntaxfehler des regulären Ausdrucks enthalten. Wir verketten die Ausnahmen, um diese Informationen an den Benutzer weiterzugeben. Fängt die `actionPerformed()`-Methode eine Ausnahme ab, so prüft sie, ob eine Ausnahme vom Typ `PatternSyntaxException` die abgefangene Ausnahme verursacht hat. Trifft dies zu, so wird die Nachricht aus dem `PatternSyntaxException`-Objekt an die Zeichenkette angefügt, mit der anschließend die Methode `handleException()` aufgerufen wird. Das Ergebnis ist ein hilfreiches Dialogfenster, siehe Abbildung 8.11 (rechts):

```
private class SearchDVD implements ActionListener {
    public void actionPerformed(ActionEvent ae) {
        previousSearchString = searchField.getText();
        try {
            tableData = controller.find(previousSearchString);
            setupTable();
        } catch (GuiControllerException gce) {
            // Inspect the exception chain
            Throwable rootException = gce.getCause();
            String msg = "Search operation failed.";
            // If a syntax error occurred, get the message
            if (rootException instanceof PatternSyntaxException) {
                msg += ("\n" + rootException.getMessage());
            }
            ApplicationRunner.handleException(msg);
            previousSearchString = "";
        }
        searchField.setText("");
    }
}
```

8.4.3 Die wiederverwendbare Konfigurationsvorlage

[179] Der Benutzer muß beim Starten der Beispielanwendung den Pfad zur Datenbankdatei angeben. Je nach Ausgangssituation sind die folgenden Informationen erforderlich:

- Beim Starten des Clients im *stand-alone*-Betriebsmodus muß der Benutzer angeben, wo sich die physikalische Datei befindet.
- Beim Starten des Clients im Netzwerkmodus muß der Benutzer die URL oder IP-Adresse des Servers, den Servertyp (RMI oder Sockets) sowie die vom Server verwendete Portadresse angeben.
- Beim Starten des Servers muß der Benutzer den Pfad der physikalischen Datei, den Servertyp

Abbildung 8.12: Links: Dialogfenster zur Eingabe des Pfades zur Datenbankdatei. Rechts: Die „Konfigurationsvorlage“ (ConfigOptions-Objekt) zum Erfassen der für Client und Server erforderlichen Parameter Pfad zur Datenbankdatei, Serverport und Servertyp (RMI oder Sockets).

(RMI oder Sockets) sowie die vom Server verwendete Portadresse angeben.

Bemerkung: Die Anforderungen an Ihre Prüfungsaufgabe sind weniger kompliziert. Beispielsweise ist die Wahl des Servertyps in der Beispielanwendung nur vorhanden, weil in diesem Buch sowohl eine RMI- als auch eine socketbasierte Netzwerkschnittstelle vorgestellt wird.

---

[180] Wir erfassen diese Einstellungen clientseitig mit Hilfe des in Abbildung 8.12 (links) gezeigten Dialogfensters. Der größte Teil dieses Dialogfensters ist die wiederverwendbare „Konfigurationsvorlage“ in Abbildung 8.12 (rechts).

[181] Der Konstruktor der Klasse `sampleproject.gui.ConfigOptions` (siehe Seite 251) entscheidet anhand seines `applicationMode`-Parameters, welche Optionen und Tooltips angezeigt werden. Dadurch können wir für den Client im *stand-alone*-Betriebsmodus, den Client im Netzwerkmodus und den Server dieselbe Konfigurationsvorlage verwenden.

[182] Für das Hauptfenster des Clients verwenden wir die einfachen Layoutmanager `BorderLayout` und `FlowLayout`. Beide Layoutmanager reichen in den meisten Fällen aus. Es gibt aber Situationen, in denen sich das Layout eines Bereiches nicht ohne weiteres mit diesen beiden Layoutmanagern bewerkstelligen läßt, beispielsweise die Konfigurationsvorlage. Wir brauchen vier Zeilen für die einzelnen Komponenten, wobei jede Komponente eine Beschriftung hat, die bezüglich ihrer rechten Seitenkante vertikal (rechtsbündig) ausgerichtet werden soll.

[183] Im folgenden Unterunterabschnitt stellen wir einen der mächtigsten Layoutmanager vor, die es gibt: `java.awt.GridBagLayout`. Wir haben die Diskussion von `GridBagLayout` absichtlich bis hierhin aufgehoben, da dieser Layoutmanager komplizierter ist, als die beiden anderen (`BorderLayout` und `FlowLayout`).

Tipp: Der Layoutmanager `GridBagLayout` ist sehr mächtig und eignet sich hervorragend für die Beispielanwendung. Eventuell brauchen Sie `GridBagLayout` aber nicht für Ihre eigene Prüfungsaufgabe. Wir empfehlen die einfachste funktionstüchtige Lösung. Sie erhalten keine Zusatzpunkt dafür, wenn Sie beweisen, daß Sie mit sämtlichen Layoutmanagern umgehen können.

8.4.3.1 Die Layoutmanager GridLayout und GridBagLayout

[184] Java verfügt über zwei gitterbasierte Layoutmanager: `java.awt.GridLayout` und `java.awt.GridBagLayout`. `GridLayout` ist der einfachere von beiden, erzwingt aber, daß alle Zellen des Gitters gleichgroß sind, siehe Abbildung 8.13 (links). Das Fenster wurde mit dem folgenden Beispiel implementiert:

```
import java.awt.*;
import javax.swing.*;
```

Abbildung 8.13: Links: Anwendungsbeispiel für den Layoutmanager `GridLayout`. Alle Zellen sind gleichgroß. Mitte: Erstes Anwendungsbeispiel für den Layoutmanager `GridBagLayout`. Die Zellen haben unterschiedliche Abmessungen. Rechts: Zweites Anwendungsbeispiel für den Layoutmanager `GridBagLayout`. Die Komponenten „Deep“ und „Wide“ überdecken mehr als eine Zelle.

```
public class MyFrame extends JFrame {
    public static void main(String[] args) throws Exception {
        JFrame theFrame = new JFrame();
        theFrame.setLayout(new GridLayout(2, 2));
        theFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        theFrame.add(new JButton("One"));
        theFrame.add(new JLabel("A very long component"));
        theFrame.add(new JTextField("Three"));
        theFrame.add(new JTextArea("Four\nFive\nSix"));
        theFrame.pack();
        theFrame.setVisible(true);
    }
}
```

[185] Wie Sie in Abbildung 8.13 (links) erkennen können, entspricht die Breite der einzelnen Komponenten der Breite der längsten Komponente, nämlich der Beschriftung „A very long component“. Die Höhe der einzelnen Komponenten entspricht der Höhe der größten Komponente, hier der Textbereich mit mehrzeiligem Inhalt.

[186] Dieses Layout ist praktisch, wenn die meisten Komponenten etwa gleichgroß sind, nicht aber wenn sich die Abmessungen der einzelnen Komponenten deutlich unterscheiden, wie bei unserer Konfigurationsvorlage. Würden wir den Layoutmanager `GridLayout` verwenden, so würde beispielsweise für jede einzelne Beschriftung ebenso viel Raum beansprucht werden, als für die größte Komponente, nämlich das Texteingabefeld mit der Beschriftung „Database location“.

[187] Der Layoutmanager `GridBagLayout` arbeitet ebenfalls mit einem Gitter von Zellen. Allerdings kann eine Komponenten hier mehr als eine Zelle besetzen. Die Spaltenbreite ist die Breite der bereitesten Komponente, die nicht mehr als eine Spalte überdeckt. Analog ist die Zeilenhöhe die Höhe der höchsten Komponente, die nicht mehr als eine Zeile überdeckt.

[188] Wir beginnen mit einem einfachen Beispiel. Es verwendet die Komponenten aus Abbildung 8.13 (links), diesesmal aber mit dem Layoutmanager `GridBagLayout` angeordnet. Abbildung 8.13 (Mitte) zeigt das implementierte Fenster:

```
import java.awt.*;
import javax.swing.*;

public class MyFrame extends JFrame {
    static GridBagLayout grid = new GridBagLayout();
    static GridBagConstraints constraints = new GridBagConstraints();
    public static void main(String[] args) throws Exception {
        JFrame theFrame = new JFrame();
        theFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        theFrame.setLayout(grid);
        constraints.anchor = GridBagConstraints.WEST;
        JButton one = new JButton("One");
        grid.setConstraints(one, constraints);
```

```
        theFrame.add(one);
        constraints.gridwidth = GridBagConstraints.REMAINDER;
        theFrame.add(constrain(new JLabel("A very long component")));
        constraints.weightx = 0.0;
        constraints.gridwidth = 1;
        theFrame.add(constrain(new JTextField("Three")));
        theFrame.add(constrain(new JTextArea("Four\nFive\nSix")));
        theFrame.pack();
        theFrame.setVisible(true);
    }
    private static Component constrain(Component c) {
        grid.setConstraints(c, constraints);
        return c;
    }
}
```

[189] Wie Sie an Abbildung 8.13 (Mitte) sehen können, richtet sich die Größe der einzelnen Komponenten nicht nach den Abmessungen der höchsten beziehungsweise breitesten Komponente. Die Höhe der ersten Zeile steht in keiner Beziehung zur Höhe der zweiten Zeile. Die Breite der ersten Spalte steht in keiner Beziehung zur Breite der zweiten Spalte.

[190] Die grundlegende Vorgehensweise beim Verwenden des Layoutmanagers `GridBagLayout` besteht darin, die Formatierungseigenschaften (*contraints*) für eine Komponente zu konfigurieren, den Layoutmanager über die für die jeweilige Komponente festgelegten Eigenschaften zu informieren und die Komponente schließlich im entsprechenden Swing-Container anzulegen. Dieses Drei-Schritt-Verfahren läßt sich am besten am folgenden Auszug aus dem vorigen Beispiel erläutern:

```
constraints.anchor = GridBadConstraints.WEST;

JButton one = new JButton("One");
grid.setConstraints(one, constraints);
theFrame.add(one);
```

[191] Vor diesen Zeilen waren die einzelnen Felder des von `constraints` referenzierten `GridBagConstraints`-Objektes mit Standardwerten initialisiert (beispielsweise Anfang in Zeile 1 und Spalte 1, Zelle im Gitter zentriert, Komponente besetzt eine Zelle im Gitter). Die erste Zeile ändert die Ausrichtung der Komponente (die Komponente wird nun links angeschlagen). Danach erzeugen wir unserer `JButton`-Komponente und benachrichtigen den Layoutmanager, daß für diese Komponente die zuvor festgelegten Formatierungseigenschaften gelten sollen. Schließlich fügen wir die `JButton`-Komponente dem von `theFrame` referenzierten Swing-Container hinzu.

Bemerkung: Im obigen Beispiel wird nur ein einziger Satz von Formatierungseigenschaften für die gesamte „Anwendung“ verwendet. Beim Aufrufen der Methode `setConstraints()` erzeugt der Layoutmanager eine Kopie der Formatierungseigenschaften. Das erleichtert die Programmierarbeit: Sie können zu Beginn des Layoutabschnitts in Ihrem Programm einen Satz allgemein gültiger Formatierungseigenschaften festlegen und diese Einstellungen überall im Layoutabschnitt verwenden.

---

[192] Da die Komponenten bei `GridBagLayout` mehr als eine Zeile und mehr als eine Spalte überdecken können, kann der Layoutmanager nicht feststellen, welches die letzte Komponente einer Zeile oder Spalte ist. Vor dem Hinzufügen der letzten Komponente einer Zeile müssen wir mit Hilfe der Formatierungseigenschaft `gridwith` definieren, daß dies die letzte Komponente in dieser Zeile ist:

```
constraints.gridwidth = GridBadConstraints.REMAINDER;
```

[193] Wie bereits bemerkt gelten im obigen Beispiel für alle Komponenten dieselben Formatierungseigenschaften. Gemäß unserer letzten Definition ist daher *jede* Komponente die letzte Komponente

in ihrer Reihe (die Einstellung wird nirgends zurückgesetzt). Die folgende Zuweisung setzt die Formatierungseigenschaft `gridwith` wieder auf ihren voreingestellten Wert 1 zurück:

```
constraints.gridwidth = 1;
```

[194] Zum Abschluß zeigen wir noch, wie Komponenten mehr als eine Zeile beziehungsweise mehr als eine Spalte überdecken können:

```
import java.awt.*;
import javax.swing.*;

public class MyFrame extends JFrame {
    static GridBagLayout grid = new GridBagLayout();
    static GridBagConstraints constraints = new GridBagConstraints();
    public static void main(String[] args) throws Exception {
        JFrame theFrame = new JFrame();
        theFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        theFrame.setLayout(grid);
        constraints.fill = GridBagConstraints.BOTH;
        constraints.gridheight = 2;
        theFrame.add(constrain(new JButton("Deep")));
        constraints.gridheight = 1;
        constraints.gridwidth = GridBagConstraints.REMAINDER;
        theFrame.add(constrain(new JButton("Wide")));
        constraints.gridwidth = 1;
        theFrame.add(constrain(new JButton("One")));
        theFrame.add(constrain(new JButton("Two")));
        theFrame.pack();
        theFrame.setVisible(true);
    }
    private static Component constrain(Component c) {
        grid.setConstraints(c, constraints);
        return c;
    }
}
```

Abbildung 8.13 (rechts) zeigt das Ergebnis.

Bemerkung: `GridBagLayout` ist einer der nützlichsten Layoutmanager und hat viel mehr Eigenschaften und Fähigkeiten, als sich einem Unterunterabschnitt eines Kapitels vorführen lassen. Eine vollständige Dokumentation der Eigenschaften und Fähigkeiten von `GridBagLayout` geht über den Rahmen dieses Buches hinaus.

8.4.3.2 Die Klasse ConfigOptions

[195] Nachdem wir die Grundlagen des Layoutmanagers `GridBagLayout` besprochen haben, können wir uns dem Quelltext der Konfigurationsvorlage widmen, über die der Benutzer beim Starten der Beispielanwendung den Pfad zur Datenbankdatei eingibt.

[196] Der Konstruktor der Klasse `ConfigOptions` speichert das Argument mit dem er aufgerufen wurde im `ApplicationMode`-Feld `this.applicationMode`, erzeugt ein `GridBagLayout`-Objekt und legt als Formatierungseigenschaft fest, daß zwischen zwei benachbarten Komponenten stets 2 Pixel Abstand eingehalten werden sollen.

[197] Unabhängig vom Betriebsmodus befindet sich neben dem Texteingabefeld für den Pfad zur

Datenbankdatei eine Beschriftung. Die entsprechende `JLabel`-Komponente wird im nächsten Schritt erzeugt.

[198] Die Konfiguration der übrigen Eigenschaften hängt davon ab, in welchem Betriebsmodus die Beispielanwendung gestartet wird. Ist eine Komponente oder Einstellung im gewählten Modus nicht sinnvoll, so wird sie dem Container für die Konfigurationsvorlage nicht hinzugefügt. Ist eine Komponente oder Eigenschaft sinnvoll, so wird sie angelegt und ein dem Betriebsmodus entsprechender Tooltip-Text hinterlegt. Der folgende Quelltext des Konstruktors der Klasse `ConfigOptions` zeigt wie die Komponenten je nach Betriebsmodus mit Hilfe des Layoutmanagers `GridBagLayout` angeordnet werden:

```
public ConfigOptions(ApplicationMode applicationMode) {
    super();
    this.applicationMode = applicationMode;

    GridBagLayout gridbag = new GridBagLayout();
    GridBagConstraints constraints = new GridBagConstraints();
    this.setLayout(gridbag);

    // Standard options
    // ensure there is always a gap between components
    constraints.insets = new Insets(2, 2, 2, 2);

    // Build the Data file location row
    JLabel dbLocationLabel = new JLabel(DB_LOCATION_LABEL);
    gridbag.setConstraints(dbLocationLabel, constraints);
    this.add(dbLocationLabel);

    if (applicationMode == ApplicationMode.NETWORK_CLIENT) {
        locationField.setToolTipText(DB_IP_LOCATION_TOOL_TIP);
        constraints.gridwidth = GridBagConstraints.REMAINDER; //end row
    } else {
        locationField.setToolTipText(DB_HD_LOCATION_TOOL_TIP);
        //next-to-last in row
        constraints.gridwidth = GridBagConstraints.RELATIVE;
    }
    locationField.addFocusListener(new ActionHandler());
    locationField.setName(DB_LOCATION_LABEL);
    gridbag.setConstraints(locationField, constraints);
    this.add(locationField);

    if ((applicationMode == ApplicationMode.SERVER)
        || (applicationMode == ApplicationMode.STANDALONE_CLIENT)) {
        browseButton.addActionListener(new BrowseForDatabase());
        constraints.gridwidth = GridBagConstraints.REMAINDER; //end row
        gridbag.setConstraints(browseButton, constraints);
        this.add(browseButton);
    }

    if ((applicationMode == ApplicationMode.SERVER)
        || (applicationMode == ApplicationMode.NETWORK_CLIENT)) {
        // Build the Server port row if applicable
        constraints.weightx = 0.0;

        JLabel serverPortLabel = new JLabel(SERVER_PORT_LABEL);
        constraints.gridwidth = 1;
        constraints.anchor = GridBagConstraints.EAST;
        gridbag.setConstraints(serverPortLabel, constraints);
        this.add(serverPortLabel);

        portNumber.addFocusListener(new ActionHandler());
```

```
        portNumber.setToolTipText(SERVER_PORT_TOOL_TIP);
        portNumber.setName(SERVER_PORT_LABEL);
        constraints.gridwidth = GridBagConstraints.REMAINDER; //end row
        constraints.anchor = GridBagConstraints.WEST;
        gridbag.setConstraints(portNumber, constraints);
        this.add(portNumber);

        // Build the Server type option row 1 if applicable
        constraints.weightx = 0.0;

        JLabel serverTypeLabel = new JLabel("Server Type: ");
        constraints.gridwidth = 1;
        constraints.anchor = GridBagConstraints.EAST;
        gridbag.setConstraints(serverTypeLabel, constraints);
        this.add(serverTypeLabel);

        constraints.gridwidth = GridBagConstraints.REMAINDER; //end row
        constraints.anchor = GridBagConstraints.WEST;
        gridbag.setConstraints(socketOption, constraints);
        socketOption.setActionCommand(SOCKET_SERVER_TEXT);
        socketOption.addActionListener(new ActionHandler());
        this.add(socketOption);

        // Build the Server type option row 2 if applicable
        constraints.weightx = 0.0;

        constraints.gridwidth = GridBagConstraints.REMAINDER; //end row
        constraints.anchor = GridBagConstraints.WEST;
        constraints.gridx = 1;
        gridbag.setConstraints(rmiOption, constraints);
        rmiOption.addActionListener(new ActionHandler());
        rmiOption.setActionCommand(RMI_SERVER_TEXT);
        this.add(rmiOption);

        ButtonGroup serverTypesGroup = new ButtonGroup();
        serverTypesGroup.add(socketOption);
        serverTypesGroup.add(rmiOption);
    }
}
```

[199/200] Wird die Wahlmöglichkeit zwischen RMI und Sockets angezeigt, so erscheinen zwei untereinander angeordnete Radiobuttons, mit einem links davon angebrachten gemeinsamen Beschreibungstext. Es gibt verschiedene Möglichkeiten diese Anordnung zu realisieren. Wir hätten die Beschriftungskomponente zwei Spalten breit wählen, oder einen eigenen Container für die Radiobuttons anlegen können. Beide Ansätze wurden aber bereits vorgeführt, so daß wir statt dessen die absolute Position des `rmiOption`-Radiobuttons im Zellengitter angeben wollen:

```
constraints.gridx = 1;
```

[201/202] Diese Zuweisung teilt dem Layoutmanager mit, daß die entsprechende Komponente in der Spalte 1 platziert werden soll (die Spaltennummerierung beginnt bei 0). Abbildung 8.14 (links, Seite 257) zeigt die Konfigurationsvorlage im Serverfenster. Kehren Sie anschließend hierher zurück, um zu lernen, wie wir die Informationsübertragung zwischen der Konfigurationsvorlage und ihrem Container beziehungsweise dem Dialogfenster implementiert haben.

8.4.3.3 Das Observer-Entwurfsmuster

[203] Es sind eine Reihe von Eigenschaften zu konfigurieren, wobei weder Client noch Server gestartet werden können, bevor alle benötigten Einstellungen erfolgt sind. Der Client setzt die Konfigurations-

vorlage in ein Dialogfenster ein, der Server in das Serverfenster. Wir brauchen einen Lösungsansatz, so daß Client und Server ein Objekt der Konfigurationsvorlage erzeugen und bei Änderung eines Feldinhaltes benachrichtigt werden können.

[204] In dieser Situation bietet sich das Entwurfsmuster *Observer* an. Die Klasse des beobachteten Objektes wird von `java.util.Observable` abgeleitet. Die Klasse, deren Objekte bei Änderungen eines `Observable`-Objektes benachrichtigt werden sollen, implementiert das Interface *`java.util.Observer`*. Nachdem sich ein *`Observer`*-Objekt („beobach*tendes* Objekt") beim `Observable`-Objekt („beobach*tetes* Objekt") registriert hat, wird es über jede Zustandsänderung des `Observable`-Objektes informiert.

Bemerkung: Es ist nicht üblich, das *Observer*-Entwurfsmuster dafür zu verwenden, daß ein Teil einer graphischen Benutzeroberfläche Benachrichtigungen von einem anderen Teil der Benutzeroberfläche empfangen kann. Das *Observer*-Entwurfsmuster tritt dagegen häufig zusammen mit *Model-View-Controller* auf, wobei es einen oder mehrere beobachtende(n) Präsentationskomponenten/-modi gibt. Ein weiterer Anwendungsfall ist ein `Observable`-Server, der alle *`Observer`*-Objekte benachrichtigt, wenn sich der Zustand des Servers ändert.

---

[205] In der Regel wird die Klasse des beobachteten Objektes von `Observable` abgeleitet. Da unsere `ConfigOptions`-Klasse aber schon von `JPanel` abgeleitet ist, scheidet dieser Ansatz aus. Wir haben statt dessen in `ConfigOptions` eine innere Klasse namens `ConfigObservable` angelegt und von `Observable` abgeleitet. Die innere Klasse `ConfigObservable` stellt eine Methode namens `getObservable()` zur Verfügung, mit deren Hilfe sich Client und Server als Beobachter registrieren können (`getObservable().addObserver(this)`, siehe Klassen `DatabaseLocationDialog` und `ServerWindow`).

[206] Wenn Sie die in Java eingebaute Implementierung des *Observer*-Entwurfsmusters verwenden (*`Observer`*-/`Observable`-Objekte), können Sie den beobachtenden Objekten ein Objekt als Argument übergeben. Die Klasse `OptionUpdate` implementiert das Entwurfsmuster *Value-Object* und enthält das geänderte Feld mit seinem neuen Inhalt:

```
package sampleproject.gui;

public class OptionUpdate implements Serializable {
    public enum Updates {
        NETWORK_CHOICE_MADE,
        DB_LOCATION_CHANGED,
        PORT_CHANGED;
    }
    private Updates updateType = null;
    private Object payload = null;
    public OptionUpdate(Updates updateType, Object payload) {
        this.updateType = updateType;
        this.payload = payload;
    }
    public Updates getUpdateType() {
        return this.updateType;
    }
    public void getPayload(Object payload) {
        this.payload = payload;
    }
    public Object getPayload() {
        return payload;
    }
}
```

Sie finden weitere Informationen über das Entwurfsmuster *Transfer-Object* im Die Klasse DVD und das Entwurfsmuster Value-Objekt von Kapitel 5.

[207] Setzt oder ändert der Benutzer eine Einstellung in der Konfigurationsvorlage, also dem Objekt, welches eventuell von mehreren *`Observer`*-Objekten beobachtet wird, so wird ein Ereignisbehandler aufgerufen. Handelt es sich dabei um ein für die *`Observer`*-Objekte interessantes Ereignis, so wird ein `OptionUpdate`-Objekt erzeugt, bewertet und an alle registrierten *`Observer`*-Objekte versendet.

[208] Verläßt der Fokus beispielsweise das `locationField`- oder das `portNumber`-Feld, so wird die `ConfigOptions`-Methode `focusLost()` aufgerufen:

```
public void focusLost(FocusEvent e) {
    if (DB_LOCATION_LABEL.equals(e.getComponent().getName())
            && (!locationField.getText().equals(location))) {
        location = locationField.getText();
        updateObservers(OptionUpdate.Updates.DB_LOCATION_CHANGED,
                        location.trim());
    }
    if (SERVER_PORT_LABEL.equals(e.getComponent().getName())
            && (!portNumber.getText().equals(port))) {
        port = portNumber.getText();
        updateObservers(OptionUpdate.Updates.PORT_CHANGED, port.trim());
    }
}
```

[209] Angenommen, der Benutzer hat den Inhalt des Pfad-, Port- oder Servertyp-Feldes geändert. Dann wird die `ConfigOptions`-Methode `updateObservers()` aufgerufen:

```
private void updateObservers(OptionUpdate.Updates updateType, Object payLoad) {
    OptionUpdate update = new OptionUpdate(updateType, payLoad);
    observerConfigOptions.setChanged();
    observerConfigOptions.notifyObservers(update);
}
```

Bemerkung: Beachten Sie, daß die `setChanged()`-Methode des `Observable`-Objektes *vor* der `notifyObservers()`-Methode aufgerufen wird. Wenn Sie `notifyObservers()` aufrufen, ohne zuvor `setChanged()` aufzurufen, bleibt die Benachrichtigung der *`Observer`*-Objekte aus.

---

[210] Das genügt, um beliebig viele Beobachterobjekte zu benachrichtigen. Zwar mag der Eindruck entstehen, wir hätten viel Aufwand getrieben, um eine Information an das Client- beziehungsweise Serverfenster zu übergeben, aber der Vorteil besteht darin, daß wir die Konfigurationsvorlage und ihren Benutzer entkoppelt haben. Jede Klasse kann die Konfigurationsvorlage benutzen. Die einfache Registrierung als Beobachter der Konfigurationsvorlage genügt, um über Änderungen ihres Zustandes informiert zu werden.

[211] Die beiden folgenden Unterabschnitte 8.4.4 und 8.4.5 beschreiben die „gegenüberliegende Seite“ des *Observer*-Entwurfsmusters. Beide Klassen implementieren Beobachter des hier beschriebenen `Observable`-Objektes.

8.4.4 Die Klasse DatabaseLocationDialog

[212/213] Das nächste Beispiel (Konstruktor der Klasse `DatabaseLocationDialog`) zeigt die Integration der Konfigurationsvorlage in das Dialogfenster. Das Dialogfenster verhindert das Starten

der Anwendung bis die erforderlichen Informationen eingegeben sind. Die benötigten Daten hängen vom Betriebsmodus des Clients ab. Im *stand-alone*-Betrieb genügt die Angabe des Pfades zur Datenbankdatei. Im Netzwerkmodus benötigen wir dagegen die URL des Servers, die Portadresse und den Servertyp (RMI oder Sockets). Die Anweisungen im Konstruktor beginnen mit der Annahme, daß Portadresse und Servertyp bei einem Dialogfenster im *stand-alone*-Betriebsmodus nicht benötigt werden. Danach erzeugt der Konstruktor ein `ConfigOptions`-Objekt und registriert sein `DatabaseLocationDialog`-Objekt als Beobachter.

[214] Der Konstruktor erzeugt eine `JOptionPane`-Komponente mit zwei Schaltflächen (`JOptionPane.OK_CANCEL_OPTION`) deren Vaterkomponente des Clients das Hauptfenster (`MainWindow`-Objekt) ist. Die beiden Schaltflächen des Dialogfensters werden anschließend so überschrieben, daß die „Connect"-Schaltfläche solange ausgegraut bleibt, bis der Benutzer die verlangten Informationen einsetzt.

[215] Das Schließen des Dialogfensters, ohne eine der Schaltflächen „Connect" oder „Exit" anzuklicken, wird wie das Anklicken von „Exit" interpretiert. Wir legen fest, daß das Dialogfenster beim Beenden nichts tun soll und verknüpfen es anschließend noch mit einen Ereignisbehandler:

```
public DatabaseLocationDialog(Frame parent, ApplicationMode connectionMode) {
    configOptions = (new ConfigOptions(connectionMode));
    configOptions.getObservable().addObserver(this);

    // load saved configuration
    SavedConfiguration config = SavedConfiguration.getSavedConfiguration();

    // the port and connection type are irrelevant in standalone mode
    if (connectionMode == ApplicationMode.STANDALONE_CLIENT) {
        validPort = true;
        validCnx = true;
        networkType = ConnectionType.DIRECT;
        location = config.getParameter(SavedConfiguration.DATABASE_LOCATION);
    } else {
        // there may not be a network connectivity type defined and, if
        // not, we do not set a default - force the user to make a choice
        // the at least for the first time they run this.
        String tmp = config.getParameter(SavedConfiguration.NETWORK_TYPE);
        if (tmp != null) {
            try {
                networkType = ConnectionType.valueOf(tmp);
                configOptions.setNetworkConnection(networkType);
                validCnx = true;
            } catch (IllegalArgumentException e) {
                log.warning("Unknown connection type: " + networkType);
            }
        }

        // there is always at least a default port number, so we don't have
        // to validate this.
        port = config.getParameter(SavedConfiguration.SERVER_PORT);
        configOptions.setPortNumberText(port);
        validPort = true;

        location = config.getParameter(SavedConfiguration.SERVER_ADDRESS);
    }

    // there may not be a default database location, so we had better
    // validate before using the returned value.
    if (location != null) {
        configOptions.setLocationFieldText(location);
```

```
            validDb = true;
        }

        options = new JOptionPane(configOptions,
                                  JOptionPane.QUESTION_MESSAGE,
                                  JOptionPane.OK_CANCEL_OPTION);

        connectButton.setActionCommand(CONNECT);
        connectButton.addActionListener(this);

        boolean allValid = validDb && validPort && validCnx;
        connectButton.setEnabled(allValid);

        exitButton.setActionCommand(EXIT);
        exitButton.addActionListener(this);

        options.setOptions(new Object[] {connectButton, exitButton});

        dialog = options.createDialog(parent, TITLE);
        dialog.setDefaultCloseOperation(JDialog.DO_NOTHING_ON_CLOSE);
        dialog.addWindowListener(this);
        dialog.setVisible(true);
    }
```

[216] Ändert sich eine Einstellung in der Konfigurationsvorlage, so werden alle registrierten *`Observer`*-Objekte benachrichtigt. Die Benachrichtigung erfolgt durch die *`Observer`*-Methode `update(Observable o, Object arg)`. Im Körper der `update()`-Methode wird zunächst geprüft, ob das übergebene Objekt vom Typ `OptionUpdate` ist. Andernfalls wird die Benachrichtigung verworfen. Die verifizierte Referenz wird explizit in den Typ `OptionUpdate` umgewandelt, um anschließend die Informationen über die Zustandsänderung abfragen zu können:

```
    if (! (arg instanceof OptionUpdate)) {
        log.log(Level.WARNING, "DatabaseLocationDialog received update type: "
                + arg, new IllegalArgumentException();)
            return;
    }

    OptionUpdate optionUpdate = (OptionUpdate) arg;
```

Warnung: Prüfen Sie vor einer Typumwandlung *grundsätzlich*, ob die fragliche Referenz auf `null` verweist beziehungsweise dem gewünschten Typ angehört. Wenn Sie den Objekttyp prüfen, kann die explizite Prüfung auf `null` entfallen, da der `instanceof`-Operator `false` zurückgibt, wenn sein linker Operand `null` referenziert. Prüfen Sie bei jeder öffentlichen Methode, ob eines ihrer Argument `null` referenziert. Obwohl wir wissen, daß das gegenwärtige `Observable`-Objekt nur Objekte vom Typ `OptionUpdate` sendet, können wir nicht garantieren, daß sich diese Eigenschaft in Zukunft nicht ändert. Sollte es zu einer Änderung kommen, wird eine Warnung protokolliert, die Auskunft darüber erteilt, welches Argument übergeben wurde, eventuell sogar woher das Argument stammt.

Tipp: Wir erzeugen in der Argumentliste der obigen `log()`-Methode ein Ausnahmeobjekt vom Typ `IllegalArgumentException` um den Stackinhalt protokollieren zu können. Da das Ausnahmeobjekt nicht ausgeworfen wird, läuft die Anwendung weiter. Das Erzeugen eines Ausnahmeobjektes, um seinen Informationsinhalt zur Verfügung zu haben, ist bei der Fehlersuche eine nützliche Vorgehensweise.

Bemerkung: Es ist nicht immer sinnvoll, alle Aktualisierungen zu protokollieren, die für Ihr Ob-

jekt nicht interessant sind. In der ersten veröffentlichten AWT-Version wurde jedes Ereignis an jedes Objekt gesendet, das sich für ein beliebiges Ereignis registriert hatte. Ein Objekt, das sich beispielsweise dafür „interessierte“, ob der Mauszeiger über einen bestimmten Bereich bewegt wurde, konnte unzählige Meldungen erhalten, wenn die Maus über andere Komponenten der Oberfläche bewegt wurde. In einem solchen Fall werden Sie nicht alle „uninteressanten“ Benachrichtigungen protokollieren wollen. Im vorliegenden Fall, da wir alle zur Zeit möglichen Ereignistypen kennen, ist es sinnvoll, eine Warnung zu protokollieren, wenn ein unerwartetes Ereignis eintritt.

[217] Anschließend validieren wir die Benutzereingabe für den Namen der Datenbankdatei und setzen ein boolesches Flag, wenn die Eingabe gültig ist:

```
location = (String) optionUpdate.getPayload();
if (configOptions.getApplication() == ApplicationMode.STANDALONE_CLIENT) {
    File f = new File(location);
    if (f.exists() && f.canRead() && f.canWrite()) {
        validDb = true;
        log.info("File chosen " + location);
    } else {
        log.warning("Invalid file " + location);
    }
}
```

[218] Schließlich prüfen wir, ob alle benötigten Eingaben vorhanden sind und aktivieren die „Connect“-Schaltfläche:

```
boolean allValid = validDb && validPort && validCnx;
connectButton.setEnabled(allValid);
```

8.4.5 Das Serverfenster

[219/220] Abbildung 8.14 (links) zeigt die graphische Benutzeroberfläche des Servers (das „Serverfenster“). Das Serverfenster besteht größtenteils aus der im vorigen Unterabschnitt entwickelten Konfigurationsvorlage (`ConfigOptions`-Objekt). Das folgende Beispiel zeigt den Konstruktor der Klasse `sampleproject.gui.ServerWindow`. Wir legen den Inhalt der Titelleiste des Fensters fest, konfigurieren den Server so, daß die Anwendung beim Anklicken des „Schließen“-Icons in der Fensterdekoration beendet wird und sorgen dafür, daß die Fenstergröße nicht verändert werden kann.

[221] Analog zum Client legen wir auch für den Server eine Menüleiste an und platzieren die Konfigurationsvorlage im Serverfenster. Wir legen Schaltflächen an, um den Server zu starten (deaktiviert, bis alle erforderlichen Einstellungen eingetragen sind) beziehungsweise zu beenden und laden die in der Datei *dennys.properties* gespeicherten Konfigurationseinträge. Abschließend zentrieren wir das Serverfenster in der Bildschirmmitte und setzen seine Sichtbarkeit auf `true`:

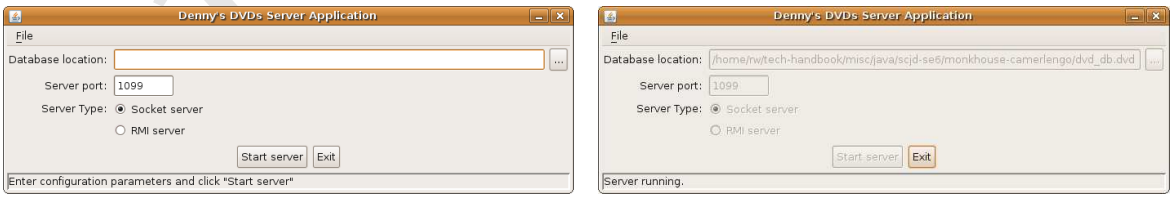

Abbildung 8.14: Links: Graphische Benutzeroberfläche der Serveranwendung („Serverfenster“). Rechts: Serverfenster bei laufendem Server.

```
public ServerWindow() {
    super("Denny's DVDs Server Application");
    this.setDefaultCloseOperation(this.EXIT_ON_CLOSE);
    this.setResizable(false);

    // Add the menu bar
    JMenuBar menuBar = new JMenuBar();
    JMenu fileMenu = new JMenu("File");
    JMenuItem quitMenuItem = new JMenuItem("Quit");
    quitMenuItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                System.exit(0);
            }
        });
    quitMenuItem.setMnemonic(KeyEvent.VK_Q);
    fileMenu.add(quitMenuItem);

    fileMenu.setMnemonic(KeyEvent.VK_F);
    menuBar.add(fileMenu);

    this.setJMenuBar(menuBar);

    configOptionsPanel.getObservable().addObserver(this);
    this.add(configOptionsPanel, BorderLayout.NORTH);

    this.add(commandOptionsPanel(), BorderLayout.CENTER);

    status.setBorder(BorderFactory.createBevelBorder(BevelBorder.LOWERED));
    JPanel statusPanel = new JPanel(new BorderLayout());
    statusPanel.add(status, BorderLayout.CENTER);
    this.add(statusPanel, BorderLayout.SOUTH);

    // load saved configuration
    SavedConfiguration config = SavedConfiguration.getSavedConfiguration();

    // there may not be a default database location, so we had better
    // validate before using the returned value.
    String databaseLocation =
        config.getParameter(SavedConfiguration.DATABASE_LOCATION);
    configOptionsPanel.setLocationFieldText(
        (databaseLocation == null) ? "" : databaseLocation);

    // there may not be a network connectivity type defined and, if not, we
    // do not set a default - force the user to make a choice the at least
    // for the first time they run this.
    String networkType
        = config.getParameter(SavedConfiguration.NETWORK_TYPE);
    if (networkType != null) {
        try {
            ConnectionType connectionType
                = ConnectionType.valueOf(networkType);
            configOptionsPanel.setNetworkConnection(connectionType);
            startServerButton.setEnabled(true);
        } catch (IllegalArgumentException e) {
            log.warning("Unknown connection type: " + networkType);
        }
    }

    // there is always at least a default port number, so we don't have to
    // validate this.
    configOptionsPanel.setPortNumberText(
        config.getParameter(SavedConfiguration.SERVER_PORT));
```

```
        status.setText(INITIAL_STATUS);

        this.pack();

        // Center on screen
        Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
        int x = (int) ((d.getWidth() - this.getWidth()) / 2);
        int y = (int) ((d.getHeight() - this.getHeight()) / 2);
        this.setLocation(x, y);

        this.setVisible(true);
```

[222/223] Nachdem der Benutzer alle Daten in gültiger Form eingegeben und die „Start Server“-Schaltfläche angeklickt hat, sind alle Eingabekomponenten und Schaltflächen mit Ausnahme von „Exit“ deaktiviert („ausgegraut“), siehe Abbildung 8.14 (rechts). Folgende Anweisungen im Ereignisbehandler der „Start Server“-Schaltfläche sind für diese Deaktivierung verantwortlich:

```
configOptionsPanel.setLocationFieldEnabled(false);
configOptionsPanel.setPortNumberEnabled(false);
configOptionsPanel.setBrowseButtonEnabled(false);
configOptionsPanel.setSocketOptionEnabled(false);
configOptionsPanel.setRmiOptionEnabled(false);

startServerButton.setEnabled(false);
```

[224] Wir legen einen Shutdown-Hook an, um „Server beenden“-Ereignisse behandeln zu können:

```
Runtime.getRuntime().addShutdownHook(new CleanExit());
```

[225/226] Schließlich starten wir den entsprechenden Server, je nachdem, welchen Servertyp der Benutzer gewählt hat. Die Implementierung des Shutdown-Hooks ist einfach. Die Klasse definiert lediglich einen Thread, der beim Beenden der Beispielanwendung aufgerufen wird. Der Thread sperrt die Datenbankdatei, damit kein anderer Thread einen Schreibversuch unternehmen kann, während die Anwendung herunterfährt und beendet sich anschließend. Das folgende Beispiel zeigt den Quelltext des Shutdown-Hooks (ohne Dokumentationskommentare):

```
package sampleproject.gui;

import java.io.IOException;
import java.util.logging.*;
import sampleproject.db.*;

public class CleanExit extends Thread {
    private Logger log = Logger.getLogger(”sampleproject.gui”);

    private String dbLocation = null;

    public CleanExit(String dbLocation) {
        this.dbLocation = dbLocation;
    }

    public void run() {
        log.info(”Ensuring a clean shutdown”);
        try {
            DvdDatabase database = new DvdDatabase(dbLocation);
            database.setDatabaseLocked(true);
        } catch (IOException e) {
            log.log(Level.SEVERE, ”Failed to lock database before exiting”, e);
        }
    }
}
```

Tipp: Das Implementieren eines Shutdown-Hooks ist ein guter Programmierstil für Anwendungen, die auf einem Server laufen. Unabhängig davon, ob Sie die `System.exit()`-Methode aufrufen, ob der Benutzer die Anwendung beendet oder ob die Anwendung vom Betriebssystem beendet wird, wird stets ein und derselbe Shutdown-Hook aufgerufen Das betriebssystemseitige Beenden der Anwendung ist eine besonders wertvolle Anwendungssituation. Stellen Sich sich vor, daß der Rechner auf dem die Anwendung läuft, an eine unterbrechungsfreie Stromversorgung (USV) angeschlossen ist und der Strom ausfällt. Üblicherweise läuft die USV eine Zeit lang über Batterien. Bevor die Batterien leer sind, benachrichtigt die USV das Betriebsystem, um den Rechner herunterzufahren. Das Betriebssystem fordert daraufhin alle laufenden Anwendungen auf, sich zu beenden. Der Shutdown-Hook gestattet Ihrer Anwendung diese Aufforderung wahrzunehmen und sich sauber zu beenden.

8.5 Änderungen an Swing in J2SE 5

[227] Anläßlich der Veröffentlichung der J2SE 5 hat Sun Microsystems einige Änderungen an der Swing-Bibliothek vorgenommen. Sie können einige dieser Verbesserungen nutzen, um Ihre Prüfungsaufgabe aufzupolieren. Wir fassen diese Änderungen in diesem Abschnitt kurz zusammen.

8.5.1 Verbesserung des „Metal“-Look-and-Feels

[228–230] Vor J2SE 5 hatten Sie keine andere Wahl, als das „Metal“-Look-and-Feel, um plattformübergreifendes Look-and-Feel zu gewährleisten, siehe Abbildung 8.15 (links). Seit J2SE 5 gibt es eine modifizierte Version des „Metal“-Look-and-Feels, siehe Abbildung 8.15 (rechts). Sun Microsystems bezeichnet die beiden Versionen des „Metal“-Look-and-Feels als „Steel“- beziehungsweise „Ocean“-Thema. Wenn Sie das traditionelle Thema („Steel“) verwenden möchten, bewerten Sie beim Starten der Anwendung die Eigenschaft `swing.metalTheme` mit `steel`:

```
-Dswing.metalTheme=steel
```

8.5.2 Konfigurierbare Look-and-Feels (Themen)

[231] Ein „Thema“ (gleichbedeutend: „Skin“) gestattet die Anpassung eines Look-and-Feels einer Anwendung oder Website, ohne den Quelltext ändern zu müssen. In der Regel wird hierfür eine Konfigurationsdatei verwendet.

[232] Das von Sun Microsystems mitgelieferte `javax.swing.plaf.synth.SynthLookAndFeel` ist ein Beispiel für ein per Konfigurationsdatei anpassungsfähiges Look-and-Feel. Wenn Sie dieses Look-

Abbildung 8.15: Links: Beispiel für das „Metal“-Look-and-Feel mit „Steel“-Thema. Rechts: Beispiel für das „Metal“-Look-and-Feel mit „Ocean“-Thema.

and-Feel verwenden, können Ihre Benutzer anhand der Konfigurationsdatei Ihre Anwendung an ihre persönliches gewohntes Look-and-Feel anpassen, ohne Ihren Quelltext ändern zu müssen.

8.5.3 Erleichteres Anlegen von Swing-Komponenten in Containern

[233] Vor Version 5 des Java Development Kits war es nicht möglich, Komponenten direkt in einem Container vom Typ *javax.swing.RootPaneContainer* (namentlich `JApplet`, `JDialog`, `JFrame`, `JInternalFrame` und `JWindow`, alle im Package `javax.swing`) anzulegen. Statt dessen mußten diese Komponenten in der Inhaltsebene (*content pane*) angelegt werden, so daß zunächst eine Referenz auf die Inhaltsebene angefordert werden mußte (siehe auch Bemerkung auf Seite 219):

```
JFrame theFrame = new JFrame();
theFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
Container thePane = theFrame.getContentPane();
thePane.add(new JButton("Exit"));
theFrame.pack();
theFrame.setVisible(true);
```

[234] Viele Klassen, die `JRootPane` implementieren do not need to use the various panes that exist – they only need to add content to the content pane. Für Version 5 des Java Development Kits wurden die `add()`-Methoden dieser Klassen do umgeschrieben, daß ihr Verhalten den Erwartungen der Entwickler entspricht. Das obige Beispiel läßt sich damit wie folgt umschreiben:

```
JFrame theFrame = new JFrame();
theFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
theFrame.add(new JButton("Exit"));
theFrame.pack();
theFrame.setVisible(true);
```

[235] This reduces confusion as to when we should be dealing with the frame itself or the panel, and makes the code a little more readable.

8.6 Zusammenfassung

[236] Eine gut durchdachte graphische Benutzeroberfläche überbrückt die Lücke zwischen Benutzer und Anwendung. Solide Entwurfsmuster wie *Model-View-Controller* (*MVC*) gewährleisten, daß sich Änderungen an der Datenvisualisierung nur minimal auf die restliche Anwendung auswirken. Der Benutzer beurteilt die Qualität einer Anwendung nach der Funktionalität der graphischen Benutzeroberfläche. Daher ist es wichtig, die graphische Benutzeroberfläche mit Sorgfalt zu planen und zu entwickeln.

[237] In diesem Kapitel haben wir die Grundzüge des Designs und der Entwicklung graphischer Benutzeroberflächen behandelt und anhand einiger Beispiele vorgeführt, wie Sie Layoutmanager zur Anordnung von Swing-Komponenten verwenden können, um Ihr Design zu realisieren. Es wichtig, zu verstehen, daß es keinen „einzig wahren Weg“ zur Entwicklung einer graphischen Benutzeroberfläche gibt. Verwenden Sie die in diesem Kapitel vorgestellten Ansätze, um eine nach Ihrem Ermessen sinnvolle graphische Benutzeroberfläche für Ihre Prüfungsaufgabe zu entwickeln.

8.7 Häufige Fragen

- *Frage*: Die Anleitung verlangt, daß nur Swing-Komponenten verwendet werden. Keiner der Layoutmanager liegt im Package `javax.swing`. Kann ich deswegen durchfallen?

  Antwort: Sie dürfen alle Layoutmanager verwenden. Sie dürfen aber keine AWT-Komponente verwenden, wenn es in der Swing-Bibliothek einen entsprechenden Ersatz gibt. Beispielsweise dürfen Sie keine `java.awt.Button`-Komponente verwenden, da es `javax.swing.JButton` gibt.

- *Frage*: Ersetzt Swing das Abstract Window Toolkit (AWT)?

  Antwort: Swing ist *kein* Ersatz für AWT. Swing baut auf dem Fundament und den Vorlagen (*patterns*) von AWT auf. Swing- und AWT-Komponenten können nebeneinander und anstelle des jeweils anderen in graphischen Benutzeroberflächen verwendet werden. (Beachten Sie aber, daß Sie in den Prüfungsaufgaben nur Swing verwenden dürfen; siehe oben.)

  Swing bietet, verglichen mit AWT, erhebliche Verbesserungen, sowohl im Hinblick auf Performanz als auch auf Funktionalität. Daher gewichtet Sun Microsystems Swing höher als AWT. Swing ist ein Pflichtbestandteil der Prüfung zum *Sun Certified Java Developer*.

- *Frage*: Wie läßt sich ein Fenster auf dem Bildschirm zentrieren?

  Antwort: Um ein Fenster auf dem Bildschirm zu zentrieren, fragen Sie zunächst mit Hilfe der Methode `java.awt.Toolkit.getDefaultToolkit().getScreenSize()` seine Abmessungen ab. Werten Sie das zurückgelieferte `java.awt.Dimension`-Objekt aus, um aus der Bildschirmgröße und den Abmessungen des zu zentrierenden Fensters die Koordinaten der linken oberen Fensterecke auszurechnen. Platzieren Sie das Fenster schließlich an den berechneten Eckkoordinaten. Das folgende Beispiel zentriert ein Dialogfenster (`connectionDialog`):

  ```
  // Center on screen
  Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
  int x = (int) ((d.getWidth() - connectionDialog.getWidth())/2);
  int y = (int) ((d.getHeight() - connectionDialog.getHeight())/2);
  dialog.setLocation(x, y);
  ```

- *Frage*: Wie läßt sich eine mnemonische Tastenkombination für eine Komponente in einer graphischen Benutzeroberfläche definieren?

  Antwort: Die Methode `setMnemonic(int keyvalue)` gestattet für nahezu jede Komponente die Definition einer mnemonischen Tastenkombination. Die Klasse `KeyEvent` enthält vordefinierte Konstanten.

- *Frage*: Kann ich das Macintosh Look-and-Feel auf einem Windowsrechner verwenden, oder umgekehrt?

  Antwort: Sun Microsystems stellt mehrere plattformübergreifende Look-and-Feels zur Verfügung. Vor Version 5 des Java Development Kits waren „Metal“ und „Motif“ die beiden einzigen garantiert vorinstallierten Look-and-Feels. Seit Version 5 des Java Development Kits gehören „Ocean“ (ein konfigurierbares „Metal“-Thema) und „Synth“ (ein Look-and-Feel mit mehreren Themen) zum Standardumfang der Java-Distribution. Die meisten übrigen Bibliotheken sind betriebsystemabhängig. Obwohl es technisch möglich ist, das Windows Look-and-Feel auf einem Mac zu verwenden oder umgekehrt, ist es in der Regel nicht erwünscht, eine Anwendung auszuliefern, die das Look-and-Feel eines anderen Betriebssystems verwendet.

  Die Mächtigkeit von Swing besteht unter anderem in der Erweiterbarkeit und der Möglichkeit, eigene Look-and-Feels zu entwickeln. Es gibt einige plattformübergreifende individuelle Look-

and-Feels, die aber nicht zur Standard-Java-Distribution gehören. Es ist nicht ratsam, sich bei der Entwicklung einer graphischen Benutzeroberfläche auf eine solche Bibliothek zu verlassen.

- *Frage*: Ich habe im Internet ein Look-and-Feel entdeckt, daß mir besser gefällt, als die Look-and-Feels von Sun Microsystems. Kann ich es anstelle der Look-and-Feels von Sun Microsystems verwenden?

  Antwort: Lesen Sie sorgfältig in Ihrer Anleitung nach. Die meisten Anleitungen betonen, daß sämtlicher eingereichter Quelltext von Ihnen selbst stammen muß. Wenn Sie ein Look-and-Feel verwenden, das Sie nicht selbst geschrieben haben und das nicht zum JDK gehört, verletzen Sie diese Regel.

- *Frage*: Wo finde ich Richtlinien über Look-and-Feels für Java?

  Antwort: Unter der Internetadresse *java.sun.com/products/jlf* finden Sie ausführliche Informationen über das Design von Look-and-Feels und die in diesem Kontext üblichen Verfahren.

- *Frage*: Wo finde ich Informationen über das Design graphischer Benutzeroberflächen und das Testen der Anwenderfreundlichkeit?

  Antwort: Die beiden folgenden Internetadressen sind gute Startpunkte für Recherchen über Anwenderfreundlichkeit und Testen: *http://www.useit.com*, *http://www.asktog.com*.

Teil III

Projektabschluß und Einreichen der Prüfungsaufgabe

Kapitel 9

Projektabschluß

[0] Herzlichen Glückwunsch! Sie haben es geschafft, viele schwierige Themen und ihre komplizierten Einzelheiten durchzuarbeiten. Wir haben eine Menge Tips und Informationen zusammengetragen, die Ihnen dabei helfen sollen, die Prüfung zum *Sun Certified Java Developer* zu meistern und einige Eigenschaften und Fähigkeiten von Version 5 des Java Development Kits vorgestellt. Dieses Kapitel faßt einige Entscheidungen zur Architektur der Beispielanwendung zusammen, die wir während der Planung und Entwicklung des Projektes getroffen haben und beantwortet die letzten offenen Fragen über die Packagestruktur und das Starten der Beispielanwendung. Wir besprechen in diesem Kapitel:

- Die Begründungen der Entscheidungen beim Entwurf der Beispielanwendung.
- Das Herunterladen der Beispielanwendung aus dem „Source Code"-Abschnitt der Apress-Website (*http://www.apress.com/book/sourcecode*).
- Das Übersetzen und die Packagestruktur der Beispielanwendung.
- Anlegen einer Manifest-Datei (*MANIFEST.MF*).
- Starten der Beispielanwendung im *stand-alone*-Betriebsmodus.
- Starten der Beispielanwendung im Netzwerkmodus.
- Das Testen der Threadsicherheit der Beispielanwendung mit Hilfe einer Klasse, die mehrere Threads simuliert.
- Verpacken der Beispielanwendung in einer *.jar* Datei.

Abbildung 9-1, Seite 296 (Buch), zeigt Version 2.0 der Beispielanwendung im Überblick.

Bemerkung: In der Regel wird ein Übersichtsdiagramm für eine Anwendung von links nach rechts oder von oben nach unten gezeichnet. Der Client steht dabei am linken oder oberen Rand des Diagramms. Wir haben das Diagramm von unten nach oben gezeichnet, um die Abfolge der einzelnen Schritte bei der Entwicklung der Beispielanwendung durch die Kapitel dieses Buches wiederzugeben. Wir haben in Kapitel 5 mit der Klasse `sampleproject.db.DvdDatabase` begonnen, in den Kapitel 6 und 7 Remote Method Invocation (RMI) beziehungsweise Sockets als Netzwerkschnittstelle beschrieben und in Kapitel 8 die graphische Benutzeroberfläche entwickelt.

---

[1] Die Design-Entscheidungen sind durch den dreischichtigen Aufbau der Beispielanwendung motiviert, einen zentralen Baustein in der Prüfung zum *Sun Certified Java Developer*. Die Kernthemen

sind der Sperr-/Reservierungsmechanismus für die logischen Datensätze im Package `sampleproject.db`, die Wahl zwischen RMI und Sockets als Netzwerkschnittstelle (Packages `sampleproject.remote` und `sampleproject.sockets`) und die Anwendung des Entwurfsmusters *Model-View-Controller* (*MVC*) im Package `sampleproject.gui`.

[2] Außerdem besprechen wir die Verteilung der Beispielanwendung auf die verschiedenen Packages sowie die Installation, das Aufrufen und Testen der Beispielanwendung. Sie können den Quelltext der Beispielanwendung aus dem „Source Code"-Abschnitt der Apress-Website (*http://www.apress.com/book/sourcecode*) herunterladen.

9.1 Threadsicherheit und Sperr-/Reservierungsverfahren

[3] Das Thema „Threadsicherheit" wird in Kapitel 4 behandelt. Kapitel 4 enthält eine Einführung in die Threadprogrammierung und verwandte Gebiete wie Synchronisierung, Sperren, *concurrency* sowie das neue `java.util.concurrent`-Package in Version 5 des Java Development Kits. *Threads im Wartezustand wurden ebenso detailliert besprochen wie weitere Gesichtspunkte der gemeinsamen Verwendung einer Resource durch mehrere Clientthreads*.

[4] In Kapitel 5 wurde ein Reservierungsmechanismus für die logischen Datensätze in der Beispielanwendung implementiert, der diese technischen Konzepte nutzt. Beispielsweise erzeugt die RMI-Netzwerkschnittstelle pro Client ein separates `DvdDatabase`-Objekt, so daß der Besitzer der Reservierung eines Datensatzes identifiziert werden kann (siehe Unterunterabschnitt 5.4.1.3).

9.2 Die Wahl zwischen RMI und Sockets als Netzwerkschnittstelle

[5] Die Wahl zwischen RMI und Sockets (plus Objektserialisierung) als Netzwerkschnittstelle Ihrer Prüfungsaufgabe ist nicht leicht. Sun Microsystems akzeptiert beide Varianten, wobei für die Wahl alleine weder Punkte vergeben noch abgezogen werden. Viele Kandidaten entscheiden sich daher einfach für die Variante, mit der sie weniger Erfahrung haben, um soviel wie möglich lernen zu können. Wenn Sie mit beiden Varianten nicht vertraut sind oder beide beherrschen, lohnt es sich, die jeweiligen Vorteile zu betrachten.

Bemerkung: Beachten Sie beim Lesen der folgenden Seiten, daß einem Vorteil einer Variante ein Nachteil der anderen Variante gegenüberstehen kann. Hin und wieder lassen sich mehrere Vorteile in einem Argument zusammenfassen. Wir geben nach Möglichkeit bei der Diskussion eines Vorteiles an Ort und Stelle auch mögliche Gegenargumente an.

9.2.1 Die Vor- und Nachteile von Sockets plus Serialisierung

[6/7] Bei „richtigen" Anwendungen fällt die Entscheidung zwischen RMI und Sockets häufig anhand der Anforderungen an die Skalierbarkeit und Performanz der Anwendung. Sockets sind die ideale Lösung, wenn es auf Performanz ankommt, da Sie das Ausmaß an Unkosten beschränken können Sockets eignen sich zur Datenübertragung (auch in komprimierter Form), wenn kein aufwendiges Kommunikationsprotokoll erforderlich ist. Je umfangreicher und komplizierter das Protokoll wird, umso attraktiver wird RMI. Eine sorgfältig durchdachte socketbasierte Netzwerkschnittstelle kann eine RMI-basierte Netzwerkschnittstelle allerdings in ihrer Performanz übertreffen. Wenn Sie viele Anfragen effizient verarbeiten müssen, sind Sockets unter Umständen die bessere Wahl.

[8] Selbstverständlich können Sie Ihre Daten auch bei einer RMI-basierten Netzwerkschnittstelle komprimieren oder verschlüsseln. Sie können eine RMI-Netzwerkschnittstelle mit einer anwendungsspezifischen Socket-Fabrikklasse kombinieren, um spezielle Funktionalität zu implementieren (weitere Informationen unter der Internetadresse *http://java.sun.com/j2se/1.5.0/docs/guide/rmi/socketfactory*). Wenn Sie sich aber für Sockets entscheiden, haben Sie die Möglichkeit, sich auf die tatsächlich benötigte Funktionalität zu beschränken. Beispielsweise können Sie die bei RMI stets vorhandene „Heartbeat-Funktionalität" („Keep-Alive-Funktionalität") fortlassen. Diese Funktionalität bewirkt, daß ein RMI-Server in regelmäßigen Zeitabständen ein Signal (einen „Herzschlag") an die RMI-Registratur sendet, damit die Registratur „weiß", daß der Server noch „lebt". Die RMI-Clients senden ebenfalls solche Signale.

[9] Ein weiterer Vorteil von Sockets besteht darin, daß Systemadministratoren in der Regel wissen, wie ein Server implementiert wird, der an einem bestimmten Port lauscht, selbst wenn die Verbindung durch eine Firewall geht. Nur wenige Administratoren können dagegen eine RMI-Registratur in Betrieb nehmen und noch weniger wissen, wie eine RMI-Registratur hinter einer Firewall konfiguriert wird. RMI kann durch eine Firewall betrieben werden, aber die Konfiguration ist komplizierter als eine Socketverbindung durch eine Firewall.

Bemerkung: Sockets waren für lange Zeit das Standardverfahren für Netzwerkverbindungen, während RMI eine vergleichsweise neue Variante ist. Dies alleine rechtfertigt natürlich nicht, Sockets den Vorzug vor RMI zu geben. Die Entscheidung, serialisierte Objekte über eine Socketverbindung zu senden, garantiert *praktisch*, daß sich nur ein Java-Objekt mit dem Socketserver verbinden kann. (Die „Garantie" ist durch den Zusatz „praktisch" absichtlich eingeschränkt, da die Java-Spezifikationen das Format serialisierter Objekte explizit dokumentiert. Es ist also prinzipiell möglich, wenn auch sehr kompliziert, einen Client in einer anderen Sprache zu schreiben.) Im Gegensatz dazu gestattet RMI-over-IIOP (das von CORBA verwendete Internet Inter-ORB Protocol) jedem CORBA-kompatiblen Client, sich mit Ihrem Server zu verbinden (die Verwendung von RMI-over-IIOP ist gegenwärtig in den Prüfungsaufgaben nicht erlaubt). Weitere Informationen über IIOP und CORBA finden Sie auf den entsprechenden Wikipedia-Seiten *http://en.wikipedia.org/wiki/IIOP* beziehungsweise *http://en.wikipedia.org/wiki/Corba*.

---

[10] Änderungen an entfernten Methoden wirken sich bei einer socketbasierten Netzwerkschnittstelle weniger stark aus, als bei RMI. Solange Sie keinen tieferen Eingriff vornehmen, etwa eine von den Clients benötigte Methode entfernen oder den Port ändern auf dem der Server lauscht, können Sie in der Regel einen socketbasierten Server austauschen, ohne daß sich die Clients dieser Änderung bewußt sein müssen (sie erfahren einfach nichts von der neuen Funktionalität). Bei einer RMI-Netzwerkschnittstelle mit älteren Clients (vor Version 5 des Java Development Kits) oder Clients, die Stubklassen nicht dynamisch erzeugen dürfen, müssen die clientseitigen Stubklassen neu übersetzt und verteilt werden. Ist das dynamische Herunterladen von Stubklassen erlaubt, können sich Sicherheitsprobleme im Zusammenhang mit der `codebase`-Eigenschaft ergeben.

[11] Eine socketbasierte Netzwerkschnittstelle benötigt in der Regel weniger Sockets und weniger Netzwerkverkehr zwischen Client und Server. Die socketbasierte Netzwerkschnittstelle der Beispielanwendung kommt mit nur einen Socket pro Client aus und das übertragene Datenvolumen hängt davon ab, wie aufwendig das Kommunikationsprotokoll gestaltet wird. Eine RMI-basierte Netzwerkschnittstelle beansprucht stets einen lauschenden Port für die RMI-Registratur, einen lauschenden Port für Ihren RMI-Server, einen verbundenen Port zwischen Server und Registratur sowie je einen Socket pro Client zwischen Client und Server. Eine RMI-basierte Netzwerkschnittstelle verursacht außerdem mehr Netzwerkverkehr, sowohl während des Verbindungsaufbaus (Anfrage an die RMI-Registratur) als auch im Leerlauf (durch die verteilte automatische Speicherbereinigung und das Senden von Heartbeat-Signalen).

[12] Bei einer RMI-Lösung läuft zudem stets ein zusätzlicher Prozeß (der RMI-Server). Die meisten Rechner werden durch diesen einen Prozeß nicht belastet. Beansprucht der Prozeß aber einen erheblichen Anteil an Rechenzeit und/oder Arbeitsspeicher, weil der Server einen langsamen Prozessor und wenig Arbeitsspeicher hat oder der RMI-Server eine aufwendige Verarbeitung durchführen muß, so kann dieser Aspekt durchaus in den Vordergrund treten.

[13] Eine *sorgfältig durchdachte* socketbasierte Netzwerkschnittstelle *kann* für einen unerfahrenen Entwickler leichter zu verstehen und zu pflegen sein, als eine RMI-Netzwerkschnittstelle. Das widerspricht der allgemein akzeptierten Auffassung, daß RMI die leichtere Lösung ist. Hier sind verschiedene Punkte zu berücksichtigen. Bei einer sorgfältig entwickelten socketbasierten Netzwerkschnittstelle ist die tatsächliche Implementierung der Netzwerkschnittstelle vor der Implementierung des Clients und des Server verborgen. Client und Server rufen lediglich Schnittstellenmethoden auf. Der unerfahrene Entwickler braucht sich allerdings bei einer socketbasierten Netzwerkschnittstelle nicht um die RMI-Registratur, die Verwendung des `rmic`-Compilers und den Umgang mit den Stubklassen zu kümmern. Die Lernkurve ist also kürzer. Andererseits besteht nur wenig Zweifel daran, daß eine RMI-Netzwerkschnittstelle nach dem Durchlaufen der zugehörigen Lernkurve leichter zu entwickeln, zu verstehen und zu pflegen ist. Außerdem ist eine socketbasierte Netzwerkschnittstelle nur dann klarer zu verstehen als eine RMI-Netzwerkschnittstelle, wenn sie tatsächlich sehr sorgfältig durchdacht wurde (und die RMI-Netzwerkschnittstelle zugleich schlecht geschrieben ist).

[14] Das Identifizieren des Clients ist bei einer socketbasierten Netzwerkschnittstelle einfacher: Sie können den beim Verbindungsaufbau erzeugten Thread verwenden, um den Client zu identifizieren. Falls Ihre Aufgabe allerdings verlangt, daß Sie Cookies zur Reserierung/Aufhebung von Reservierungen und beim Aufrufen von Änderungsmethoden verwenden, können Sie den Cookie auswerten, um den Client zu identifizieren. Wenn Sie sich aufgrund von Skalierbarkeitsüberlegungen für einen Threadpool entscheiden, können Sie Threads allerdings nicht mehr zum Identifizieren eines Clients gebrauchen. Gleichgültig, ob Sie sich für RMI oder Sockets entscheiden, bietet eine Fabrikklasse für den Verbindungsaufbau einen einfachen Mechanismus zur Client-Identifizierung.

[15] Die Gefahr, daß jemand versehentlich Ihren Server außer Betrieb nimmt, ist gering. Ruft bei einer RMI-Netzwerkschnittstelle ein anderer Server die *`java.rmi.registry.Registry`*-Methode `rebind()` mit derselben entfernten Referenz auf, so ersetzt dieser Server den vorherigen. Ist der Server bei einer socketbasierten Netzwerkschnittstelle einmal an einen bestimmten Port gebunden, so kann kein anderer Server diesen Port verwenden. Aber auch dies kann sich zu Ihrem Nachteil auswirken. Installiert ein anderer Benutzer einen anderen Server mit derselben Portadresse und startet den Rechner neu, so kann nicht vorausgesagt werden, welcher Server zuerst an den Port gebunden wird und Sie wissen nicht, welcher Server Ihre Anfragen entgegennimmt. Dies ist eine Wettlaufsituation (*race condition*). Wettlaufsituationen gehören thematisch zu Threads und werden in Kapitel 4 behandelt.

9.2.2 Die Vor- und Nachteile von RMI

[16] Beim Entwurf der Lösung einer Prüfungsaufgabe für die Zertifizierung zum *Sun Certified Java Developer* brauchen Skalierbarkeit und Performanz nicht berücksichtigt zu werden. Wie wir bereits im vorigen Unterabschnitt diskutiert haben, beinhaltet keine der beiden Lösungsvarianten eine Schnittstelle, über die sich ein nicht in Java geschriebener Client mit der Anwendung verbinden kann.

[17] Eine sorgfältig durchdachte socketbasierte Netzwerkschnittstelle *kann* für einen unerfahrenen Entwickler leichter zu verstehen und zu pflegen sein als eine RMI-Netzwerkschnittstelle. Als Entwickler wird allerdings von Ihnen erwartet, daß Sie sowohl mit Sockets als auch mit RMI vertraut sind, um eine fundierte Entscheidung zwischen beiden treffen zu können (zu Ihrer Unterstützung ist

RMI in Kapitel 6 und Sockets in Kapitel 7 beschrieben). Wenn Sie beide Verfahren verstanden haben, stellen Sie vermutlich fest, daß Sie Ihre Arbeit mit RMI einfacher gewesen wäre. Ihr Gutachter beherrscht RMI, das heißt Sie können sich in dieser Hinsicht bedenkenlos für RMI entscheiden. Wir fassen die Vorteile von RMI in diesem Unterschnitt kurz zusammen.

[18] Die Implementierung einer socketbasierten Netzwerkschnittstelle beinhaltet stets die Entwicklung eines Kommunikationsprotokolls. Bei einer Socketverbindung werden serialisierte Objekte über ein Netzwerk transportiert und müssen auf der Empfängerseite zur Laufzeit ausgewertet werden. Das Entwickeln einer socketbasierten Netzwerkschnittstelle kann daher mühsamer und schwieriger sein, als bei RMI. Die letzte Aussage läßt sich ein Stück weit umkehren, wenn Sie einen gut faktorisierten Quelltext entwickeln, das heißt wenn jede Klasse nur eine einzige Aufgabe erfüllt und die Methoden auf dieselbe Weise faktorisiert sind. Andererseits macht das *Command*-Entwurfsmuster die Konstruktion des Kommunikationsprotokolls zu einer trivialen Aufgabe.

[19] Die Einzelheiten der Objektserialisierung und Netzwerkkommunikation sind bei RMI verborgen, während Sie sie bei einer socketbasierten Netzwerkschnittstelle selbst implementieren müssen. Sie erfinden gewissermaßen ein Verfahren neu, das bereits existiert.

[20] Unabhängig davon, wieviel Mühe sie sich beim Entwickeln Ihres Quelltextes geben, ist eine socketbasierte Netzwerkschnittstelle stets umfangreicher als eine RMI-Netzwerkschnittstelle. Je länger ein Quelltext ist, umso wahrscheinlicher sind Fehler vorhanden. Es ist unter Unterständen sinnvoller, die von den RMI-Entwicklern geschriebenen Bibliotheken zu nutzen, die über viele Jahre hinweg getestet und verwendet worden sind (ein Teil des RMI-Quelltextes geht auf Version 1.1 des Java Development Kits zurück, die im September 1997 herausgegeben wurde).

[21] RMI macht das Netzwerk transparent, das heißt aus der Perspektive des Clients verhält sich ein entferntes Objekt wie ein lokales Objekt. Es ist daher nicht notwendig, ein Handshake-Protokoll zu implementieren oder sich mit systemnahen Einzelheiten wie dem Öffnen und Schließen von Socketverbindungen auseinanderzusetzen.

[22] Durch die Verwendung von Interfaces und die Tatsache, daß sich entfernte Methoden wie lokale Methoden verhalten, sind Aufrufe entfernter Methoden in der Regel typsicher. Dasselbe gilt grundsätzlich auch für Sockets, wobei im Gegensatz zu RMI keine Interfaces verlangt werden. Dementsprechend ist es nicht schwer, ein Programm zu schreiben, das zur Übersetzungszeit (und sogar zur Laufzeit) keine Fehler durch Verletzung der Typsicherheit meldet.

[23] RMI befreit Sie auch von der Aufgabe, einen multithreadfähigen Server zu schreiben, eine unter Umständen komplizierte Aufgabe. You are still required to write thread-safe code in either protocol, but the actual server does not have to spawn thread or manage thread pools.

[24] Threadpools können die Performanz einer für viele gleichzeitige Benutzerzugriffe ausgelegten Anwendung deutlich verbessern. RMI beinhaltet Threadpools sozusagen „ohne Aufpreis“. Dieser Luxus hat aber einen Nachteil: Die Identifizierung des Clients ist etwas komplizierter, wenn keine Cookies erlaubt sind.

[25] RMI ist eine der tragenden Säulen der Enterprise JavaBeans (EJBs). Das Erlernen und die Verwendung von RMI in Ihrer Prüfungsaufgabe wird Ihnen bei EJB-Projekten helfen.

[26] Die RMI-Registratur unterstützt schnelles dynamisches Deployment des Servers. Die `Registry`-Methode `rebind()` gestattet Ihnen, den neuen Server mit minimaler Ausfallzeit (*downtime*) in Betrieb zu nehmen. Bei einer Socket-Lösung müssen Sie entweder den alten Server herunter- und den neuen hochfahren (längere Ausfallzeit als bei einem Aufruf der RMI-Methode `rebind()`) oder einen anderen Port wählen (wobei Sie Ihre Clients umkonfigurieren oder ersetzen müssen).

[27] Wenn Sie sich für RMI entscheiden, entfällt die Wahl des Ports an dem Ihr Dienst Anfragen

erwartet (lauscht). Bei einer socketbasierten Netzwerkschnittstelle müssen Sie festlegen, an welchem Port Ihr Server lauscht und die Clients müssen eine Verbindung zu diesem Port aufbauen. Wenn Sie die Portadresse nicht konfigurierbar machen und eine andere Anwendung denselben Port beansprucht (und die Portadresse ebenfalls nicht konfigurierbar ist), muß eine der beiden Anwendungen mit geänderter Portadresse neu übersetzt werden. Auch wenn die Portadresse konfigurierbar ist müssen Sie alle Clients neu konfigurieren, um den geänderten Serverport zu verwenden, was schwierig oder in der Praxis sogar undurchbar sein kann. RMI verwendet per Voreinstellung eine von der RMI-Registratur zufällig gewählte serverseitige Portadresse. Die Clients müssen nur die Portadresse der Registratur kennen, um anfragen zu können, wie sie den für sie interessanten Server erreichen können.

[28] RMI gestattet das Herunterladen von Klassen, wobei der Gebrauch dieses Mechanismus' in den SCJD-Prüfungsaufgaben nicht erlaubt ist. Sie können diesen Mechanismus beispielsweise nutzen, um einen Algorithmus herunterzuladen, der sämtliche zwischen Client und Server transportierten Daten verschlüsselt. Da der Algorithmus dynamisch heruntergeladen wird, erfährt der Entwickler auf der Clientseite nicht, wie Sie den Algorithmus implementiert haben (Sie können sogar verbergen, daß der Algorithmus überhaupt verwendet wird), und entsprechend gering ist die Gefahr, daß Ihr Algorithmus geknackt wird, da ihn niemand zu Gesicht bekommt.

9.2.2.1 Entscheidung für eines der beiden Verfahren

[29] Es gibt sowohl für Sockets also auch für RMI gute Argumente aber keine absolute Trennlinie zwischen den beiden Lösungsvarianten. Wir glauben, daß Sun Microsystems diese Situation absichtlich herbeigeführt hat, um zu sehen wie Sie sich entscheiden und wie Sie Ihre Entscheidung begründen.

[30] Entscheiden Sie selbst, welche der obigen Argumente (oder Begründungen aus anderen Quellen) Ihnen am sinnvollsten erscheinen und welche Sie verteidigen wollen/können, wenn Sie aufgefordert werden, Ihre Entscheidung zu erläutern.

9.3 MVC in der graphischen Benutzeroberfläche

[31] Kapitel 8 dokumentiert einige Design-Entscheidungen im Hinblick auf die graphische Benutzeroberfläche der Beispielanwendung. Die Darstellung beginnt mit allgemein gültigen Richtlinien für den Entwurf von Anwendungsschnittstellen zu menschlichen Benutzern und erstreckt sich bis hin zur Diskussion einzelner Swing-Komponenten wie `JTable`. Im Gegensatz zur Wahlmöglichkeit zwischen RMI und Sockets (siehe Kapitel 6 und 7 sowie Abschnitt 9.2) verlangt Sun Microsystems für das Bestehen der Prüfung zum *Sun Certified Java Developer* ausdrücklich, daß eine `JTable`-Komponente verwendet wird.

[32] Die Design-Entscheidung betrifft daher nicht die Frage, welche Swing-Komponente sich für die Datenvisualisierung in der Beispielanwendung eignet, sondern wie sich eine `JTable`-Komponente zweckmäßig in die Swing-basierte graphische Benutzeroberfläche der Beispielanwendung einfügen läßt. Die in Kapitel 8 vorgeführte Lösung ist eine anwendungsübergreifende Implementierung des *MVC*-Entwurfsmusters. Sie haben in Kapitel 8 gelernt, daß Swing-Komponenten wie `JTree` oder `JTable` das *MVC*-Entwurfsmuster intern implementieren. Für eine *MVC*-Implementierung über den Rahmen der einzelnen Komponenten hinaus ist der Entwickler dagegen selbst zuständig. Der wichtigste Vorteil eine anwendungsübergreifenden *MVC*-Architektur ist die deutlichere Abstraktion zwischen der Datenvisualisierung und dem eigentlichen Datenbestand. Die `JTable`-Komponente und

ihre interne *MVC*-Struktur spielen in der Datenvisualisierung eine Nebenrolle und arbeiten mit der anwendungsübergreifenden *MVC*-Implementierung zusammen.

9.4 Herunterladen der Beispielanwendung

[33] Sie finden die Quelltextbeispiele zu diesem Buch im „Source Code"-Abschnitt der Apress-Website (*http://www.apress.com/book/sourcecode*) zum Herunterladen. Dort befindet sich eine *.zip* Datei mit dem gesamten Quelltext von Version 2.0 der Beispielanwendung. In diesem Kapitel beschreiben wir die Version 2.0 dieser *.zip* Datei.

[34] Dekomprimieren Sie die *.zip* Datei auf Ihrem Rechner in einem Verzeichnis Ihrer Wahl. Sie erhalten ein neues Verzeichnis namens *sampleproject*, das wiederum vier Unterverzeichnisse *db*, *remote*, *sockets* und *gui* enthält. Die Unterverzeichnisse entsprechen den vier Packages

- `sampleproject.db`
- `sampleproject.remote`
- `sampleproject.sockets`
- `sampleproject.gui`

[35] Sofern Sie das Buch nicht nur durchgeblättert und mit Kapitel 9 begonnen haben, sind Ihnen diese Packages vertraut. Nachdem Sie den gesamten Java-Quelltext zur oben beschriebenen Verzeichnisstruktur dekomprimiert haben, können Sie die Beispielanwendung übersetzen.

9.5 Übersetzen und Packageaufteilung der Beispielanwendung

[36] Öffnen Sie ein Konsolenfenster und geben Sie `java -version` ein. Achten Sie darauf, daß Sie die J2SE 5 oder eine spätere Version verwenden:

```
~$ java -version

java version "1.6.0_03"
Java(TM) SE Runtime Environment (build 1.6.0_03-b05)
Java HotSpot(TM) Server VM (build 1.6.0_03-b05, mixed mode)

~$
```

Falls Sie nur eine ältere Java-Version installiert haben, laden Sie sich eine aktuelle Version herunter (*http://www.sun.com/software/javaseforbusiness/getit_download.jsp*) und installieren Sie sie.

Bemerkung: Die Endung `b05` der obigen Versionsnummer gibt an, daß es sich um „Build Nr. 5" handelt, nicht etwa, daß es eine Beta-Version ist.

---

[37] Wechseln Sie in das Wurzelverzeichnis der Beispielanwendung, das heißt in das Verzeichnis, in dem Sie die *.zip* Datei dekomprimiert haben. Wir verwenden hierfür das Verzeichnis *dennysDVDs-2.0*.

Tipp: Sie können *.zip* Dateien in der Regel mit dem `jar`-Kommando dekomprimieren (siehe unten), da *.zip* Dateien und *.jar* Dateien nach demselben Kompressionsverfahren erzeugt werden.

---

[38] Wir übersetzen nun die *.java* Dateien in ein Zielverzeichnis. Per Voreinstellung platziert `javac` die *.class* Dateien im selben Verzeichnis wie die *.java* Dateien. Der Schalter `-d` gestattet Ihnen, die *.class* Dateien in einem separaten Verzeichnisbaum anzulegen. Die Trennung von *.java* und *.class* Dateien erleichtert die Organisation Ihres Projektes.

[39] Überlegen Sie sich, wo Ihre *.class* Dateien deponiert werden sollen. Wir installieren unsere *.class* Dateien der Einfachheit halber in einem Verzeichnis namens *classes*, direkt unterhalb des Wurzelverzeichnisses *dennysDVDs2.0*, das wir per `mkdir classes` anlegen. Alternativ können Sie das Verzeichnis auch mit Hilfe Ihres Dateibrowsers (zum Beispiel dem Windows-Explorer) anlegen.

[40] Übersetzen Sie nun jedes Package separat per `javac`, wobei Sie mit dem Schalter `-d` das Zielverzeichnis angeben. Die benötigten Kommandos, um die Beispielanwendung erfolgreich zu übersetzen, sind:

```
~$ jar -xf SCJD_Exam_with_J2SE_5_Second_Edition-2563.zip
~$ mkdir classes
~$ cd src
~$ javac -d ../classes sampleproject/db/*.java
~$ javac -d ../classes sampleproject/remote/*.java
~$ javac -d ../classes sampleproject/sockets/*.java
~$ javac -d ../classes sampleproject/gui/*.java
~$
```

Achten Sie darauf, die Packages in der Reihenfolge `db`, `remote`, `sockets` und schließlich `gui` zu übersetzen. Diese Reihenfolge berücksichtigt die Abhängigkeiten innerhalb der Beispielanwendung. In Kapitel 5 existiert nur das `db`-Package. Die anderen Packages kommen in der Reihenfolge der Kapitel 6 (`remote`), 7 (`sockets`) und 8 (`gui`) hinzu.

9.6 Die Manifest-Datei

[41] Wir legen nun eine Manifest-Datei an. Diese Datei wird zusammen mit der Beispielanwendung verpackt und von der Laufzeitumgebung ausgewertet, um „die richtige Klasse“ zu laden und die Anwendung zu starten. Der entsprechende Eintrag in der Manifest-Datei ist der Name der Klasse, welche die zur Ausführung der Anwendung aufzurufende `main()`-Methode enthält. Bei unserer Beispielanwendung ist dies die Klasse `sampleproject.gui.ApplicationRunner`.

[42] Die Manifest-Datei enthält nur wenige Zeilen. Tragen Sie beiden folgenden Zeilen ein und speichern Sie die Datei unter dem Namen *MANIFEST.MF*. Wir deponieren unsere Manifest-Datei direkt im Wurzelverzeichnis *dennysDVDs2.0*.

```
Manifest-Version: 1.0
Main-Class: sampleproject.gui.ApplicationRunner
```

Bemerkung: Wenn Sie das `jar`-Kommando verwenden, um die *.jar* Datei („Java Archive“) zu erzeugen, können Sie den Namen der Manifest-Datei frei wählen. Im Abschnitt 9.8 gehen wir im Rahmen der Erläuterungen zum `jar`-Kommando genauer auf diesen Punkt ein. Falls Sie das `jar`-Kommando von Sun Microsystems allerdings *nicht* verwenden (es gibt aber eigentlich keinen Grund, `jar` nicht zu verwenden), so *müssen* Sie die Manifest-Datei *MANIFEST.MF* nennen und in einem Unterverzeichnis namens *META-INF* direkt im Wurzelverzeichnis der *.jar* Datei deponieren.

[43] Der Pfad zur Manifest-Datei innerhalb einer *.jar* Datei kommt ins Spiel, wenn wir eine *.jar* Datei aus den *.class* Dateien des Beispielprojektes erzeugen. Jede *.jar* Datei enthält eine Manifest-Datei.

Eigenschaft	Beschreibung
`Manifest-Version`	Specifies which version of the *MANIFEST.MF* definition you are conforming to. At present only version 1.0 has been defined.
`Created-By`	Gibt den Erzeuger des `jar`-Kommandos und dessen Versionsnummer an. Die Eigenschaft wird von `jar` automatisch angelegt und bewertet. Sie sollten diese Eigenschaft nicht manuell anlegen oder ihren Wert ändern.
`Class-Path`	Optionale Eigenschaft. Gibt zur Laufzeit benötigte Bibliotheken an. Die Angabe muß relativ zur aktuellen *.jar* Datei erfolgen, das heißt die Angabe `lib/another.jar` ist zulässig, nicht aber ein absoluter Pfad wie `/home/../libs/another.jar`. Mehrere *.jar* Dateien werden durch ein Leerzeichen voneinander getrennt.
`Main-Class`	Teilt der Laufzeitumgebung mit, welche Klasse aufgerufen werden soll, wenn die Laufzeitumgebung mit dem Schalter `-jar` aufgerufen wurde.

Tabelle 9.1: Die wichtigsten Eigenschaften (Schlüsselwörter) von *MANIFEST.MF*.

Die Manifest-Datei dient dazu, bestimmte Eigenschaften der *.jar* Datei zu bewerten. Tabelle 9.1 zeigt einige häufige Eigenschaften.

[44] Eine Manifest-Datei kann noch viele weitere Eigenschaften haben, deren Beschreibung aber über den Rahmen dieses Buches hinausgeht. Wenn Sie sich für diese Eigenschaften interessieren, empfehlen wir die Dokumentation für *.jar* Dateien von Sun Microsystems unter der Internetadresse *http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html#JAR Manifest*.

[45] Wenn Sie die Eigenschaften einer Manifest-Datei selbst bewerten möchten, müssen Sie dem `jar`-Kommando per Kommandozeilenoption (`-m`) eine entsprechende Datei übergeben. Andernfalls erzeugt `jar` eine Manifest-Datei mit Standardwerten und ohne `Main-Class`-Eigenschaft.

9.7 Der rmic-Compiler und das remote-Package

[46] Zu den Vorteilen von Version 5 des Java Development Kits gehört, daß die Verwendung des `rmic`-Compilers zum Erzeugen von Stubklassen nicht mehr nötig ist, da die Stubklassen nun dynamisch generiert werden können. Das manuelle Erzeugen von Stubklassen ist allerdings noch immer erforderlich, wenn Sie ältere Clients (vor Version 5 des Java Development Kits) verwenden oder das dynamische Generieren von Stubklassen nicht erlaubt ist (siehe Kapitel 6). Dieser Abschnitt soll Entwicklern helfen, die Stubklassen manuell erzeugen müssen.

Warnung: Zum Zeitpunkt der Drucklegung dieses Buches verbieten die Anleitungen der Prüfungsaufgaben das dynamische Herunterladen von Stubklassen. Ihre ausführbare *.jar* Datei muß alle Stubklassen in vorkompilierter Form enthalten. Da das dynamische Erzeugen von Stubklassen in das dynamische Herunterladen dieser Klassen mündet, dürfen Sie die Fähigkeit des JDK 5, Stubklassen dynamisch zu generieren, nicht nutzen. Sehen Sie aber die Anleitung zu Ihrer Prüfungsaufgabe in diesem Punkt sorgfältig durch. Eventuell wird das Verbot bei zukünftigen Aufgaben aufgehoben.

---

[47] Die Entscheidung für RMI als Netzwerkschnittstelle schließt das Erzeugen von Stubklassen mit Hilfe des `rmic`-Compilers ein. Die Stubklassen müssen in der *.jar* Datei enthalten sein, damit das `sampleproject.remote`-Package der Beispielanwendung bei Wahl des Servertyps „RMI“ funktioniert.

[48] Der `rmic`-Compiler muß mit den Klassen der entfernten Objekte aufgerufen werden, in unserem

Beispielprojekt also mit der Klasse `sampleproject.remote.DvdDatabaseImpl`. Achten Sie darauf, `rmic` im Zielverzeichnis aufzurufen, da `rmic` die *.class* Dateien benötigt. Der `rmic`-Compiler wird folgendermaßen mit der Klasse `DvdDatabaseImpl` aufgerufen:

```
~$ cd classes/
~$ rmic sampleproject.remote.DvdDatabaseImpl
~$ ls -l sampleproject/remote/

total 36K
-rw-r--r-- 1 rw rw 1.4K 2009-09-15 16:24 DvdConnector.class
-rw-r--r-- 1 rw rw  279 2009-09-15 16:24 DvdDatabaseFactory.class
-rw-r--r-- 1 rw rw  734 2009-09-15 16:24 DvdDatabaseFactoryImpl.class
-rw-r--r-- 1 rw rw 2.2K 2009-09-15 16:24 DvdDatabaseImpl.class
-rw-r--r-- 1 rw rw 3.8K 2009-09-15 16:25 DvdDatabaseImpl_Stub.class
-rw-r--r-- 1 rw rw  192 2009-09-15 16:24 DvdDatabaseRemote.class
-rw-r--r-- 1 rw rw  851 2009-09-15 16:24 RegDvdDatabase.class
-rw-r--r-- 1 rw rw 1.7K 2009-09-15 16:24 RmiFactoryExample.class
-rw-r--r-- 1 rw rw 1.7K 2009-09-15 16:24 RmiNoFactoryExample.class

~$
```

Der `rmic`-Compiler muß noch einmal aufgerufen werden, nämlich mit der Klasse `sampleproject.remote.DvdDatabaseFactoryImpl`.

[49] Wenn Sie den `rmic`-Compiler mit dem Schalter `-verbose` aufrufen, erhalten Sie detailliertere Informationen als bei der obigen Ausgabe. Mit dem Schalter `-help` aufgerufen, zeigt `rmic` alle verfügbaren Optionen an. Die Datei *DvdDatabaseImpl_Stub.class* ist das Ergebnis des Aufrufs des `rmic`-Compilers.

9.8 Das jar-Kommando

[50] Wir sind soweit, daß wir unsere Beispielanwendung als *.jar* Datei verpacken können. Das `jar`-Kommando archiviert Dateien unterschiedlichen Typs in Form einer einzigen zip-komprimierten Datei. Das `jar`-Kommando hat die folgende Syntax:

```
jar [options] [manifest] destination input-file [input-files]
```

[51] Um eine *.jar* Datei zu erzeugen, verwenden Sie die in Tabelle 9.2 beschriebenen vier Kommandozeilenschalter beziehungsweise -optionen `c`, `v`, `f` und `m`. Sie müssen außerdem den Pfad zur Vorlage Ihrer Manifest-Datei angeben, im Falle unserer Beispielanwendung also das Verzeichnis *dennysDVDs2.0*.

Schalter	Beschreibung
`c`	Erzeugt („create“) ein neues Archiv. Ist der Schalter `-f` nicht vorhanden, so wird das Archiv über die Standardausgabe ausgegeben.
`v`	Gibt während der Arbeit ausführliche („verbose“) Meldungen aus.
`f`	Gibt an, daß das Archiv nicht über die Standardausgabe ausgegeben, sondern in eine Datei („file“) geschrieben werden soll. Der Dateiname ist das nächste Argument auf der Kommandozeile.
`m`	Inkludiert eine Manifest-Datei, die als nächstes Argument auf der Kommandozeile übergeben wird. Das `jar`-Kommando wertet den Dateiinhalt aus und speichert ihn in einer Datei namens *MANIFEST.MF* in einem Unterverzeichnis namens *META-INF*.

Tabelle 9.2: Beim Erzeugen von *sampleproject.jar* verwendete `jar`-Schalter beziehungsweise Optionen.

Bemerkung: Enthält der Aufruf des `jar`-Kommandos mehr als eine Option, hat das Kommando also mehr als ein Argument, so müssen die Argumente in der Reihenfolge der Optionen übergeben werden. Lautet die Liste der Kommandozeilenschalter und -optionen beispielsweise `-cvfm`, so muß der Name der *.jar* Datei vor dem Namen der Manifest-Datei übergeben werden, da die Option `f` vor `m` steht. Wären die Schalter und Optionen dagegen in der Reihenfolge `-cvmf` notiert, so müßte die Manifest-Datei vor der *.jar* Datei angegeben werden, da nun `m` vor `f` steht.

[52] Das `jar`-Kommando besitzt noch weitere Schalter und Optionen. Rufen Sie `jar` mit dem Schalter `-help` auf, um eine kurze Dokumentation der verschiedenen Schalter, Optionen und Verwendungsmöglichkeiten zu erhalten. Das folgende Kommando erzeugt die Datei *sampleproject.jar*:

```
jar -cfm sampleproject.jar Manifest.mf -C classes .
```

Warnung: Dem Verzeichnisnamen „classes“ im obigen `jar`-Aufruf folgt ein einzelner Punkt. In einem Buch ist der Punkt leicht zu übersehen, vor allem am Satzende.

[53] Mit dem Schalter `-v` zeigt das `jar`-Kommando an, welche Dateien archiviert und welche Kompressionsrate dabei erzielt wurde. Verwenden Sie `-cvfm` anstelle von `-cfm`, um ausführliche Informationen anzuzeigen.

9.9 Starten der Beispielanwendung

[54] Die Beispielanwendung hat drei verschiedene Betriebsmodi („Client im *stand-alone*-Betriebsmodus“, „Client im Netzwerkmodus“ sowie „Server“), je nach dem beim Programmaufruf auf der Kommandozeile übergebenen Argument. Die folgenden drei Unterabschnitte beschreiben diese Modi.

9.9.1 Der Client im stand-alone-Betriebsmodus

[55] Wechseln Sie in das Verzeichnis, das die Datei *sampleproject.jar* enthält, um die Anwendung im *stand-alone*-Betriebsmodus aufzurufen. Die Kommandos `java` (Unix, Linux) beziehungsweise `javaw` (Windows) gestatten mit Hilfe der Option `-jar`, eine Anwendung direkt aus einer *.jar* Datei heraus aufzurufen. Geben Sie das folgende Kommando ein:

```
javaw -jar sampleproject.jar alone
```

[56] Die Datei *sampleproject.jar* kann in einem beliebigen Verzeichnis deponiert werden, da sie die gesamte Beispielanwendung enthält. Das obige Kommando öffnet zunächst das Dialogfenster zum Erfassen des Pfades zur Datenbankdatei (im Wurzelverzeichnis *dennysDVDs2.0*), siehe Abbildung

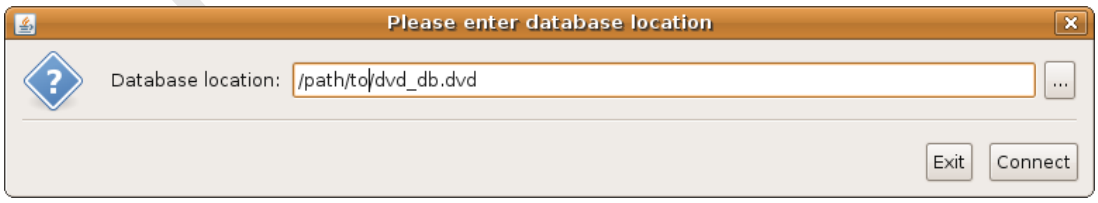

Abbildung 9.1: Starten des Clients der Beispielanwendung im *stand-alone*-Betriebsmodus.

Abbildung 9.2: Die Beispielanwendung im *stand-alone*-Betriebsmodus.

9.1. Das Hauptfenster wird geöffnet, nachdem Sie auf die „Connect"-Schaltfläche geklickt haben, siehe Abbildung 9.2.

9.9.2 Der Server

[57] Beim Serverstart müssen der Pfad zur Datenbankdatei, die gewünschte Portadresse und der Servertyp (RMI oder Sockets) angegeben werden. Geben Sie das folgende Kommando ein, um den Server zu starten:

```
javaw -jar sampleproject.jar server
```

[58] Das Kommando öffnet das Konfigurationsfenster des Servers, siehe Abbildung 9.3 (links). Geben Sie den Pfad zur Datenbankdatei ein (Wurzelverzeichnis *dennysDVDs2.0*), ändern Sie erforderlichenfalls die Portadresse, wählen Sie den Servertyp und klicken Sie auf die Schaltfläche „Start Server". Das Konfigurationsfenster zeigt nun an, daß der Server läuft und deaktiviert die meisten Komponenten, siehe Abbildung 9.3 (rechts).

[59] Wir haben im Konfigurationsfenster festgelegt, daß der Server Port 1099 verwenden soll, Abbildung 9.3 (links). Das bedeutet, daß die RMI-Registratur an Port 1099 lauscht (bei einer socket-basierten Netzwerkschnittstelle würde der Server selbst an Port 1099 lauschen). Wir können diese Aussage mittels `netstat -a` bestätigen. Das Kommando gibt alle verbundenen oder lauschenden Ports an:

```
netstat -a | find /i "listening" | find /i "1099"
```

[60] Die `/i` Zeile dokumentiert, daß ein Dienst (die RMI-Registratur) an Port 1099 lauscht. Der Port auf dem der RMI-Server selbst lauscht ist nicht bekannt und wird von der RMI-Registratur automatisch konfiguriert.

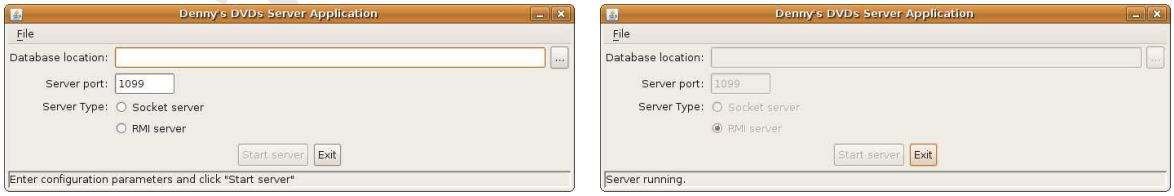

Abbildung 9.3: Starten des Servers der Beispielanwendung. Links: Konfigurationsfenster des Servers. Rechts: Server im Betrieb.

Abbildung 9.4: Starten des Clients der Beispielanwendung im Netzwerkmodus.

9.9.3 Der Client im Netzwerkmodus

[61] Beim Starten der Beispielanwendung im Netzwerkmodus sind die IP-Adresse oder der Name des Rechners, der den Server und die Datenbankdatei enthält, die Portadresse an der die RMI-Registratur beziehungsweise der Socketserver lauschen sowie der Servertyp erforderlich. Geben Sie das folgende Kommando ein, um die Beispielanwendung im Netzwerkmodus zu starten:

```
java -jar sampleproject.jar server
```

[62] Geben Sie den Pfad zur Datenbankdatei ein (entweder mit dem Namen des Rechners, zum Beispiel `localhost`, oder seiner IP-Adresse, zum Beispiel `127.0.0.1`), ändern Sie die Portadresse, falls Sie die Voreinstellung beim Serverstart geändert haben und wählen Sie denselben Servertyp wie beim Serverstart. Klicken Sie auf die „Connect“-Schaltfläche und Sie erhalten das Hauptfenster, wie in Abbildung 9.2.

9.10 Testen der Beispielanwendung

[63] Version 2.0 der Beispielanwendung enthält ein Package namens `sampleproject.test`, dessen Dateien nicht in der Verzeichnisstruktur der Beispielanwendung selbst liegen. Die *.java* Dateien der Beispielanwendung liegen im Verzeichnis *src*, während das `sampleproject.test`-Package im Verzeichnis *test* liegt. Die Verteilung des Quelltextes auf zwei Verzeichnisse gewährleistet, daß wir keine Dateien einreichen, die wir nicht einreichen wollen (siehe hierzu Abschnitt 9.11).

[64] Die folgende Abbildung zeigt die Verzeichnisstruktur unserer Beispielanwendung:

```
+-- dennysDVDs2.0
      |
      +-- classes
      |     |
      |     +-- sampleproject
      |           |
      |           +-- db
      |           |
      |           +-- direct
      |           |
      |           +-- gui
      |           |
      |           +-- remote
      |           |
      |           +-- sockets
      |
      +-- src
```

```
|      |
|      +-- sampleproject
|            |
|            +-- db
|            |
|            +-- direct
|            |
|            +-- gui
|            |
|            +-- remote
|            |
|            +-- sockets
|
+-- src - examples
|
+-- test
       |
       +-- sampleproject
             |
             +-- test
```

Wenn Sie dem Kapitel bis zu dieser Stelle gefolgt sind, haben Sie ein Wurzelverzeichnis names *dennysDVDs2.0* angelegt. (Sie können den Verzeichnisnamen und, sofern Sie unter Windows arbeiten, auch den Laufwerksbuchstaben selbstverständlich frei wählen.) Das Wurzelverzeichnis enthält zum gegenwärtigen Zeitpunkt die Unterverzeichnisse *classes* (dort hat `javac` die übersetzten *.class* Dateien deponiert), *src* (enthält die *.java* Dateien der Beispielanwendung) und *test* (enthält die Testklassen; sowohl *.java* als auch *.class* Dateien). Die Packagestruktur im Unterverzeichnis *src* wiederholt sich im Unterverzeichnis *classes*, beginnend mit *sampleproject*.

Bemerkung: Das zusätzliche Unterverzeichnis *src - examples* im Wurzelzeichnis *dennysDVDs2.0* wurde bis jetzt noch nicht angesprochen. Es enthält den Quelltext vieler im Buch vorkommender Beispiele, die nicht unmittelbar mit der Beispielanwendung zusammenhängen, etwa das Eisverkäuferbeispiel aus Kapitel 4.

---

[65] Das `sampleproject.test`-Package enthält das Testsystem unserer Beispielanwendung. Es ist unbedingt erforderlich, die Beispielanwendung im Betrieb mit mehreren Clients zu testen. Das Testsystem simuliert eine Gruppe von Benutzern beim gleichzeitigen Zugriff auf die Datenbankdatei.

[66] Das `sampleproject.test`-Package enthält zwei Klassen. Die Klasse `DBTester` ist von `Thread` abgeleitet und simuliert einen Client, der DVDs ausleiht beziehungsweise zurückgibt. Die Klasse `DBTestRunner` erzeugt mehrere `DBTester`-Threads, wartet bis sie verarbeitet sind und dokumentiert deren Erfolg (oder Scheitern).

[67] Da sich das `sampleproject.test`-Package in einem von der Beispielanwendung getrennten Verzeichnis befindet, müssen wir die beiden *.java* Dateien noch separat übersetzen. Geben Sie das folgende Kommando ein:

```
javac -cp ../classes -d ../classes sampleproject/test/*.java
```

[68] Das folgende Beispiel zeigt den Kopfabschnitt des Testclients `DBTester` bis zum Konstruktor. Im Gegensatz zum Quelltext der Beispielanwendung verbindet sich der Testclient direkt über das `sampleproject.remote.DvdConnector`- (RMI) beziehungsweise das `sampleproject.sockets.DvdConnector`-Objekt (Sockets). Die gewünschte Netzwerkschnittstelle wird durch Auskommentieren der `import`-Anweisung für die andere Variante gewählt:

```
package sampleproject.test;

import sampleproject.db.*;
import java.util.Date;

// rather than going through a factory, we are directly calling the connector
// Uncomment the DVDConnector of the protocol you want to use.
import sampleproject.remote.DvdConnector;
//import sampleproject.sockets.DVDConnector;

/**
 * A DBTester is the test equivalent of a client who is trying to book one or
 * more DVDs. However we know exactly how the DBTester is going to behave,
 * therefore we can predict the results of this testing.
 */
public class DBTester extends Thread {
    // various status for what can happen when we try to book over the network
    public enum Status {
        SUCCESS, OUT_OF_STOCK, TIMEOUT
    }

    private String dvdUpc; // the DVD we are supposed to rent
    private int numberOfRentals; // number of times to rent it
    private DBClient db; // connection to the remote database

    private int successfulRentals = 0;  // number of times we rented the DVD
    private int outOfStock = 0; // number of times we failed due to no copies left
    private int timeouts = 0; // number of times timed out trying to reserve DVD

    // To make the screen output easier to read, we are using a pretend logger
    // If we chose to convert to the real JDK logger, we could just change this
    private PretendLogger log = new PretendLogger();

    /**
     * Create a test client to book a certain DVD a certain number of times.
     * There is no exception handling here - it is all thrown back to the test
     * harness.
     *
     * @param title the name of this client - the thread name
     * @param numberOfRentals how many times we will attempt to rent the DVD
     * @param dvdUpc the unique identifier for the DVD we will attempt to rent
     * @throws Exception if we cannot connect to the database
     */
    public DBTester(String title, int numberOfRentals, String dvdUpc)
            throws Exception {
        super(title);
        this.numberOfRentals = numberOfRentals;
        this.dvdUpc = dvdUpc;

        db = DvdConnector.getRemote("localhost", "1099");
    }
```

[69] Die Arbeit wird zum größten Teil in der `run()`-Methode verrichtet: Der Testclient tritt in eine Schleife ein, die sooft durchlaufen wird, als die Anzahl der auszuleihenden Exemplare (`numberOf-Rentals`) angibt. In dieser Schleife versucht der Testclient die DVD auszuleihen. Bei Erfolg behält der Testclient die DVD für zwei Sekunden und gibt sie anschließend zurück. Ist kein Exemplar der gewünschten DVD mehr vorrätig, so beschwert sich der Testclient für eine Sekunde beim Filialleiter. Kann der Testclient die DVD nicht ausleihen, weil das Warten auf die Sperre des entsprechenden Datensatzes länger als fünf Sekunden dauert, nimmt der Testclient diesen Umstand lediglich zur Kenntnis. Unabhängig davon, welche dieser drei Situationen eintritt, wartet der Testclient für weitere

zwei Sekunden bevor er erneut versucht die DVD auszuleihen (Schleife).

Bemerkung: Wir haben jedes Ereignis mit einer bestimmten Wartezeit verknüpft, um zu einem gewissen Grad die Reproduzierbarkeit des Testergebnisses zu gewährleisten. Es kann zwar nicht garantiert werden, daß wir jedesmal dasselbe Ergebnis erhalten, da sich kleine Schwankungen in der Übertragungsgeschwindigkeit bei vielen Buchungen bemerkbar machen können. Bei wenigen Buchungen über ein LAN mit wenig Netzwerkverkehr oder wenn die Anwendung auf einem einzelnen Rechner im *stand-alone*-Betriebsmodus läuft, sollte sich das Ergebnis aber voraussagen lassen.

```
public void run() {
    int secondsForWatchingDvd = 2;
    int secondsForComplaining = 1;
    int secondsForBrowsingStore = 2;

    try {
        for (int i = 0; i < this.numberOfRentals; i++) {
            switch (rentDvd(dvdUpc)) {
                case SUCCESS:
                    successfulRentals++;
                    // watch the DVD
                    Thread.sleep(secondsForWatchingDvd * 1000);
                    // then return it so somebody else can rent it
                    returnDvd(dvdUpc);
                    break;
                case OUT_OF_STOCK:
                    outOfStock++;
                    // complain that it is not in stock
                    Thread.sleep(secondsForComplaining * 1000);
                    break;
                case TIMEOUT:
                    // just track that we had the problem, and continue
                    timeouts++;
                    break;
            }
            // wander around the DVD store looking at DVDs.
            Thread.sleep(secondsForBrowsingStore * 1000);
        }
    } catch (Exception e) {
        // This should never ever go into production code, but for testing
        // we are simply catching *every* exception and displaying it
        System.err.println("Exception thrown by " + getName());
        e.printStackTrace(System.err);
        System.err.println();
    }
}
```

[70] Die Methode `rentDvd()` dupliziert einen Teil der Geschäftslogik unserer Beispielanwendung. Die Methode reserviert die DVD, so daß der zugehörige Datensatz von keinem anderen Testclient geändert werden kann (vorausgesetzt, daß auch der andere Testclient dem Protokoll gehorcht, eine DVD also erst reserviert, bevor er den zugehörigen Datensatz ändert), fordert eine Referenz auf das entsprechende `DVD`-Objekt an (um zu gewährleisten, daß wir have the latest copy), prüft ob noch genügend Exemplare vorhanden sind (entfernt eines davon) und speichert den aktualisierten Zustand des Datensatzes in der Datenbankdatei:

```
private Status rentDvd(String upc) throws Exception {
    if (db.reserveDVD(upc)) {
```

```
            try {
                DVD dvd = db.getDVD(upc);
                int copiesInStock = dvd.getCopy();
                if (copiesInStock > 0) {
                    copiesInStock--;
                    log.info(getName() +
                             ”       -> (Rent)     ” +
                             ”Copies in stock = ” + copiesInStock );
                    dvd.setCopy(copiesInStock);
                    db.modifyDVD(dvd);
                    return Status.SUCCESS;
                } else {
                    log.info(getName() + ”    00    (No stock)”);
                    return Status.OUT_OF_STOCK;
                }
            } finally {
                db.releaseDVD(upc);
            }
        } else {
            log.info(getName() + ”    XX    (Timeout)”);
            return Status.TIMEOUT;
        }
    }
```

[71] Die Methode `returnDvd()` reserviert die DVD ebenfalls, so daß der zugehörige Datensatz von anderen Testclients nicht geändert werden kann, fordert eine Referenz auf das entsprechende `DVD`-Objekt an (um zu gewährleisten, daß wir have the latest copy), erhöht die Anzahl der verfügbaren Exemplare um 1 und sichert den aktualisierten Datensatz in der Datenbankdatei:

```
    private void returnDvd(String upc) throws Exception {
        if (db.reserveDVD(upc)) {
            try {
                DVD dvd = db.getDVD(upc);
                int copiesInStock = dvd.getCopy() + 1;
                dvd.setCopy(copiesInStock);
                log.info(getName() +
                         ” <-       (Return)   Copies in stock = ” +
                         copiesInStock );
                db.modifyDVD(dvd);
            } finally {
                db.releaseDVD(upc);
            }
        }
    }
```

[72] Nachdem der Testclient-Thread vollständig verarbeitet worden ist, ermittelt das Testsystem, wieviele erfolgreiche beziehungsweise gescheiterte Buchungen durchgeführt worden sind. Diese Informationen können über die folgenden Abfragemethoden ausgewertet werden:

```
    public int getSuccessfulRentals() {
        return successfulRentals;
    }

    public int getOutOfStock() {
        return outOfStock;
    }

    public int getTimeouts() {
        return timeouts;
```

```
}
```

[73] Schließlich wollen wir während des Testlaufs Informationen über das Geschehen ausgeben. Ein Standard-`Logger` ist hierfür ungeeignet, da die resultierende Bildschirmausgabe eventuell schwierig zu entziffern ist. Bei einem umfangreicheren Testsystem könnten wir eine eigene Formatierungsklasse von der abstrakten Klasse `java.util.logging.Formatter` ableiten. Für dieses Kapitel wäre eine eigene Formatierungsklasse aber ein unangemessener Aufwand, so daß wir uns für eine Protokollklassenattrappe (*pretend logger*) entschieden haben:

```
private class PretendLogger {
    /**
     * Displays the time and the information provided on screen.
     *
     * @param logInformation the information to display on screen.
     */
    void info(String logInformation) {
        System.out.format("%tT %s\n", new Date(), logInformation);
    }
}
```

Die innere Klasse `PretendLogger` gestattet einfache Protokollaufzeichnungen und falls wir uns zu einem späteren Zeitpunkt entscheiden sollten, auf der Protokollmechanismus von Version 5 des Java Development Kits umzusteigen, müßten wir lediglich die Deklaration des `log`-Feldes ändern.

[74] Die zweite Klasse des Testsystems (`DBTestRunner`) ist leicht zu verstehen. Sie erzeugt lediglich mehrere `DBTester`-Threads (Testclients), ruft deren `run()`-Methoden auf, wartet bis ihre Verarbeitung abgeschlossen ist und zeigt eine Statistik an:

```
package sampleproject.test;

import java.util.Calendar;

public class DBTestRunner {
    private int numberOfClients = 4; // how many test clients we will start
    private int rentalsPerClient = 2; // number of rentals each client will make
    private String dvdUpc = "32725349302"; // the DVD they will rent

    private DBTester[] clients = null; // an array of the test clients

    public static void main(String[] args) throws Exception {
        new DBTestRunner();
    }

    DBTestRunner() throws Exception {
        clients = new DBTester[numberOfClients];

        startClients();
        waitForClientsToDie();
        displayStatistics();
    }

    private void startClients() throws Exception {
       for (int i = 0; i < numberOfClients; i++) {
           String clientName = "Client " + i;
           clients[i] =  new DBTester(clientName, rentalsPerClient, dvdUpc);
           clients[i].start();
       }
   }

    private void waitForClientsToDie() throws Exception {
       // wait for them all to finish
       for (DBTester client : clients) {
```

```
            client.join();
        }
    }
```

[75] Beachten Sie, daß die `DBTester`-Objekte noch immer existieren und ihre öffentlichen Methoden aufgerufen werden können, um die Statistik zusammenzustellen, nachdem alle Testclient-Threads in den Zustand `TERMINATED` übergegangen sind:

```
private void displayStatistics() {
   // display some statistics
   System.out.println();
   formatLine("=======", "========", "========", "========", "========");
   formatLine("Client #", "Rented", "No stock", "Timeout", "Total");
   formatLine("--------", "--------", "--------", "--------", "--------");
   for (DBTester client : clients) {
       formatLine(client.getName(),
                  client.getSuccessfulRentals(),
                  client.getOutOfStock(),
                  client.getTimeouts());
   }
   formatLine("=======", "========", "========", "========", "========");
}

private void formatLine(String name, int rentals, int noStock, int timeout) {
formatLine(name,
               "" + rentals,
               "" + noStock,
               "" + timeout,
               "" + (rentals + noStock + timeout));
}

private void formatLine(String name, String rentals, String noStock,
                        String timeout, String total) {
    System.out.format("%tT %8s %8s %8s %8s %8s%n",
                      Calendar.getInstance(),
                      name,
                      rentals,
                      noStock,
                      timeout,
                      total);
}
```

[76] Die Ausgabe der Statistik und die Ausgabe in der Klasse `PretendLogger` verwenden beide die seit Version 5 des Java Development Kits vorhandene `PrintStream`-Methode `format()`. Das statische `System`-Feld `out` ist vom Typ `PrintStream`, so daß wir die `format()`-Methode zur Formatierung unserer Ausgabe verwenden können.

[77] Die `format()`-Methode erwartet ein `String`-Argument, welches die Formatierung der Ausgabe beschreibt („Formatdefinition“) und eine entsprechende Anzahl von Argumenten (ein Anwendungsbeispiel für eine Parameterliste variabler Länge). This means that no matter how many arguments you provide, one definition of the format() method can handle them all.

[78] Das `String`-Argument, das die Formatierung der Ausgabe beschreibt, hat viel mehr Einstellungsmöglichkeiten, als an dieser Stelle erklärt werden können. Tabelle 9.3 beschreibt die Formatierungszeichen, die wir in unserem Testsystem verwendet haben. (Siehe auch die API-Dokumention der Klassen `PrintStream` (Methode `format()`) und `java.text.Format` unter der Internetadresse (*http://java.sun.com/j2se/1.5.0/docs/api/index.html.*)

Zeichen	Beschreibung
`%tT`	Gibt an, daß das entsprechende Argument nach der Formatdefinition wie ein Datum formatiert werden soll (kleines `t`), wobei die Zeit im 24 Stundenmodus formatiert werden soll (großes `T`).
`%8s`	Gibt an, daß das entsprechende Argument nach der Formatdefinition rechtsbündig und mindest 8-stellig formatiert werden soll. Das Argument muß ein `String`-Objekt sein.
`%n`	Gibt das beziehungsweise die bei Ihrem Betriebssystem erforderliche(n) Zeichen für das Ende einer Zeile aus (Zeilenumbruch).

Tabelle 9.3: Formatierungzeichen in der Formatdefinition der Testumgebung.

[79] Wie Sie anhand von Abbildung 9.5 bestätigen können, enthält die Datenbankdatei drei Exemplare des Films „Night of the Ghouls“. Wir haben in unserem Testsystem definiert, daß vier Testclients versuchen sollen, diesen Film auszuleihen, wissen also, daß ein Testclient leer ausgeht. Außerdem haben wir festgelegt, daß jeder Testclient zweimal versuchen soll, diesen Film auszuleihen. Durch die Zeitvorgaben ist es möglich, daß der Testclient, der beim ersten Versuch leer ausgegangen ist, den Film beim zweiten Versuch erhält und ein anderer Testclient nichts bekommt.

[80] Die Auswertung eines Aufrufs des Testsystems lautet (Beispiel):

```
~$ java sampleproject.test.DBTestRunner

19:16:42 Client 2       -> (Rent)     Copies in stock = 2
19:16:42 Client 3       -> (Rent)     Copies in stock = 1
19:16:42 Client 1       -> (Rent)     Copies in stock = 0
19:16:42 Client 0    00    (No stock)
19:16:44 Client 3 <-       (Return)   Copies in stock = 1
19:16:44 Client 1 <-       (Return)   Copies in stock = 2
19:16:44 Client 2 <-       (Return)   Copies in stock = 3
19:16:45 Client 0       -> (Rent)     Copies in stock = 2
19:16:46 Client 3       -> (Rent)     Copies in stock = 1
19:16:46 Client 1       -> (Rent)     Copies in stock = 0
19:16:46 Client 2    00    (No stock)
19:16:47 Client 0 <-       (Return)   Copies in stock = 1
19:16:48 Client 3 <-       (Return)   Copies in stock = 2
19:16:48 Client 1 <-       (Return)   Copies in stock = 3

19:16:50 ======== ======== ======== ======== ========
19:16:50 Client #   Rented No stock  Timeout    Total
19:16:50 -------- -------- -------- -------- --------
19:16:50 Client 0        1        1        0        2
19:16:50 Client 1        2        0        0        2
19:16:50 Client 2        1        1        0        2
19:16:50 Client 3        2        0        0        2
19:16:50 ======== ======== ======== ======== ========

~$
```

Erwartungsgemäß geht ein Testclient (Nr. 0) beim ersten Versuch den Film auszuleihen leer aus und ein anderer Testclient beim zweiten Versuch (Nr. 2).

Warnung: Trotz aller Bemühungen, ein Testsystem zu entwickeln, das reproduzierbare Ergebnisse liefert, sollten Sie sich der Tatsache bewußt sein, daß der Threadscheduler der Laufzeitumgebung und die netzwerkbedingte Latenzzeit geringfügig abweichende Ergebnisse bewirken können.

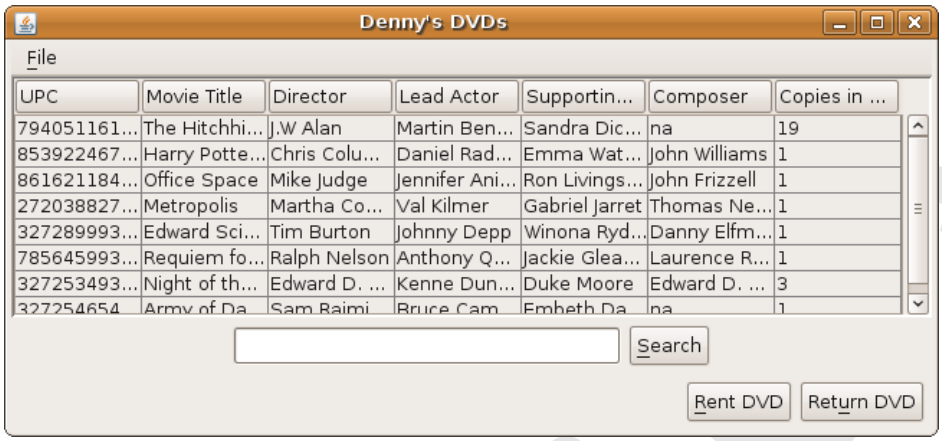

Abbildung 9.5: Die „Datenbank“ vor und nach einem Lauf der Testumgebung.

9.11 Verpacken der Beispielanwendung

[81] Vergleichen Sie diesen Abschnitt mit der Anleitung zu Ihrer Prüfungsaufgabe, um herauszufinden, was Sie verpacken und zur Begutachtung Ihrer Arbeit einreichen müssen. Da es keine Anleitung für unsere Beispielanwendung gibt, lehnen wir uns an die Anleitungen der „richtigen“ Prüfungsaufgaben an, das heißt wir erzeugen eine *.jar* Datei, die die ausführbare *.jar* Datei mit der eigentlichen Beispielanwendung, die Datenbankdatei, den Quelltext (in einem Verzeichnis namens *src*) sowie die API-Dokumentation (in einem Verzeichnis namens *doc*) enthält. Schließlich erhalten wir eine *.jar* Gesamtdatei, die wiederum eine *.jar* Datei, einige Verzeichnisse und die Datenbankdatei enthält, siehe ~~Abbildung 9-15, Seite 318 (Buch)~~.

Warnung: Die Anleitung Ihrer Prüfungsaufgabe verlangt vermutlich noch weitere Bestandteile (zum Beispiel die Dokumentation Ihrer Design-Entscheidungen) und eventuell eine andere Verzeichnisstruktur. Sie *müssen* die Anleitung von Sun Microsystems sorgfältig lesen und buchstäblich befolgen. Wenn Sie einen Teil Ihrer Lösung falsch verpacken, kann Sie der Gutachter direkt durchfallen lassen.

---

[82] Wir haben unsere ausführbare *.jar* Datei bereits vor dem Übersetzen unserer Testklassen gepackt und wissen daher, daß sie keine Bestandteile enthält, die wir nicht einreichen wollen. Falls wir die dazu erforderlichen Schritte in diesem Kapitel wiederholen müssen, genügt es, den Inhalt des Verzeichnisses *classes* zu löschen und den Inhalt des Verzeichnisses *src* nochmals zu übersetzen, um zu gewährleisten, daß die ausführbare *.jar* Datei „sauber“ ist. Hier zahlt es sich aus, die eigentliche Beispielanwendung und die Testklassen in separaten Verzeichnissen unterzubringen.

[83] Der Quelltext unserer Beispielanwendung befindet sich im Verzeichnis *src*, läßt sich also mühelos in die ausführbare *.jar* Datei übernehmen. Die Datenbankdatei befindet sich im Wurzelverzeichnis *dennysDVDs2.0* und läßt sich genauso einfach einfügen. Bleibt die Javadoc API-Dokumentation.

[84] Zunächst legen wir mit dem folgenden Kommando das Verzeichnis an, das sie API-Dokumentation enthalten wird:

```
mkdir doc/api
```

[85] Das Kommando zum Erzeugen der API-Dokumentation lautet:

```
javadoc -quiet -link http://java.sun.com/j2se/1.5.0/docs/api -d doc/api \
        -sourcepath src -public sampleproject.db sampleproject.gui \
```

```
                                        sampleproject.remote sampleproject.sockets
```

Bemerkung: Beachten Sie, daß das `javadoc`-Kommando einzeilig ist. Die Zeile wurde lediglich der besseren Lesbarkeit halber mehrmals umgebrochen.

[86] Die Ausgabe des obigen `javadoc`-Kommandos ohne den Schalter `-quiet` lautet:

```
~$ javadoc -link http://java.sun.com/j2se/1.5.0/docs/api -d doc/api \
        -sourcepath src -public sampleproject.db sampleproject.gui \
                                sampleproject.remote sampleproject.sockets

Loading source files for package sampleproject.db...
Loading source files for package sampleproject.gui...
Loading source files for package sampleproject.remote...
Loading source files for package sampleproject.sockets...
Constructing Javadoc information...
Standard Doclet version 1.6.0_03
Building tree for all the packages and classes...
Generating doc/api/sampleproject/db//DBClient.html...
Generating doc/api/sampleproject/db//DosClient.html...

    ⋮   // Ausgabe gekürzt

Generating doc/api/sampleproject/sockets//package-summary.html...
Generating doc/api/sampleproject/sockets//package-tree.html...
Generating doc/api/constant-values.html...
Generating doc/api/serialized-form.html...
Building index for all the packages and classes...
Generating doc/api/overview-tree.html...
Generating doc/api/index-all.html...
Generating doc/api/deprecated-list.html...
Building index for all classes...
Generating doc/api/allclasses-frame.html...
Generating doc/api/allclasses-noframe.html...
Generating doc/api/index.html...
Generating doc/api/overview-summary.html...
Generating doc/api/help-doc.html...
Generating doc/api/stylesheet.css...

~$
```

[87] Im Unterunterabschnitt 2.5.1.6 finden Sie ausführliche Erläuterungen zu den obigen Kommandozeilenschaltern und -optionen. Die Erläuterungen schließen Optionen und Schalter ein, die oben nicht verwendet wurden, für Sie aber nützlich sein können.

[88] Wenn Sie die API-Dokumentation häufig oder regelmäßig neu generieren, können Sie die obigen Optionen in einer Optionsdatei (*option file*) speichern. Ein Optionsdatei ist eine schlichte Textdatei, in der die einzelnen Optionen und Schalter durch Leerzeichen oder Zeilenumbrüche voneinander getrennt werden (siehe auch Kapitel 2, Seite 43), zum Beispiel:

```
/opt/usdateien/auch/unter/Linux/
```

[89] Nun können wir eine einzelne *.jar* Gesamtdatei anlegen, die alle benötigten Teile enthält. Rufen Sie dazu das folgende Kommando auf:

```
jar -cf submission.jar sampleproject.jar dvd_db.dvd src doc
```

Bemerkung: Beachten Sie, daß wir bei der *.jar* Gesamtdatei keine eigene Manifest-Datei angegeben haben. Diese *.jar* Datei dient lediglich als Container zum Einreichen Ihrer Lösung, nicht aber als ausführbare *.jar* Datei. Die vom `jar`-Kommando automatisch erzeugte Manifest-Datei reicht hierfür aus.

[91] Wir sind nun bereit, um unsere Beispielanwendung einzureichen. Denken Sie nochmals daran, Ihre individuelle Anleitung sorgfältig zu lesen. Falls Ihre Anleitung von diesem Kapitel abweicht, müssen Sie sich an Ihrer Anleitung orientieren!

[92] Zum Zeitpunkt der Drucklegung dieses Buches laden Sie die *.jar* Datei mit Ihrer Lösung auf dieselbe Website hoch, von der Sie zuvor Ihre Aufgabe, bestehend aus Anleitung und Datenbankdatei, heruntergeladen haben. Sie erhalten beim Hochladen Ihrer *.jar* Datei ausdrückliche Anweisungen, wie Sie Ihre *.jar* Gesamtdatei nennen sollen (die *.jar* Datei, die alles enthält, insbesondere Ihre ausführbare *.jar* Datei).

Tipp: Gegenwärtig müssen Sie die Berechtigung zum Hochladen beantragen, bevor Sie Ihre Lösung einreichen können. Leider können wir nicht garantieren, daß sich dieser Umstand in Zukunft ändern wird. Es gibt nur eine einzige Möglichkeit, um es herauszufinden: Versuchen Sie, Ihre Lösung hochzuladen. (Selbstverständlich nicht, wenn Ihre Lösung noch nicht vollständig ist. Andererseits können Sie Ihr Browserfenster einfach schließen, wenn Sie aufgefordert werden, die *.jar* Gesamtdatei mit Ihrer Lösung auszuwählen.) Sie sollten Ihre Lösung einreichen bevor Sie die schriftliche Prüfung absolvieren. Planen Sie vor dem Einreichen Ihrer Lösung einige Werktage als Puffer ein, falls Sie die Berechtigung zum Hochladen beantragen müssen.

9.12 Zusammenfassung

[93] Wir hoffen, daß Sie an diesem Buch Freude gefunden haben. Wir hoffen selbstverständlich vor allem, daß Sie nun gut vorbereitet sind, um Ihr Zertifizierungsprojekt für die Prüfung zum *Sun Certified Java Developer* zu meistern. Sie haben viel gelernt und es ist eine umfangreiche Aufgabe, alles Gelernte in den Kontext Ihrer Prüfungsaufgabe einzuordenen. Zweifellos beschäftigt Sie die eine oder andere schleichende Frage, etwa „Habe ich meine Anwendung ausreichend getestet?“ oder „Wie würde ich meine Anwendung in dieser oder jener Situation verhalten?“

[94] Es gibt scheinbar immer eine Möglichkeit, um Ihre Anwendung zu verbessern und es ist wichtig, im Laufe der Zeit zur Überzeugung zu kommen, daß Ihre Anwendung gut genug ist, um dem forschenden Blick des Gutachters von Sun Microsystems standzuhalten. Schließlich geht es in erster Linie darum, daß Sie die Prüfung bestehen und nicht darum ein kommerzielles Produkt zu entwickeln. Sun Microsystems schätzt die Entwicklungszeit für eine funktionstüchtige Lösung auf etwa 100 Stunden.

[95] Die folgende Liste zählt einige Punkte auf, die für das Bestehen Ihrer Prüfung wichtig sind:

- Lesen Sie die Anleitung zu Ihrer Prüfungsaufgabe sehr sorgfältig. Obwohl viele Prüfungsaufgaben unter demselben Namen herausgegeben werden, gibt es Unterschiede in Details.
- Lesen Sie dieses Buch. Machen Sie sich zumindest mit der Beispielanwendung *Denny's DVDs* vertraut, um zu verstehen, wie wir die grundsätzlichen Probleme gelöst haben, beispielsweise die Reservierung logischer Datensätze, die beiden Netzwerkschnittstellen (RMI und Sockets) und die graphische Benutzeroberfläche, insbesondere den Umgang mit der `JTable`-Komponente. Wir sind davon überzeugt, daß Sie für Ihre eigene Prüfungsaufgabe gerüstet

sind, wenn Sie den Quelltext der Beispielanwendung verstanden haben.

- Greifen Sie während der Arbeit an Ihrer Prüfungsaufgabe auf dieses Buch zurück.
- Beteiligen Sie sich am SCJD-Forum auf der Java-Ranch (siehe Abschnitt 9.13).
- Testen Sie Ihre Anwendung. Entwickeln Sie nach der Vorlage in diesem Kapitel ein Testsystem. Testen Sie Ihre Anwendung nach Möglichkeit in einem echten Netzwerk. Testen Sie jeden einzelnen Anwendungsfall. Testen Sie Ihre Anwendung mit mehreren Clients und verwenden Sie dazu eine Testklasse wie `sampleproject.test.DBTestRunner`.
- Verpacken Sie Ihre *readme.txt*-Datei und Design-Entscheidungen zusammen mit Ihrer Anwendung und reichen Sie das Projekt unter Beachtung der Prüfungsrichtlinien bei Sun Microsystems ein.

Viel Erfolg!

9.13 Häufige Fragen

- *Frage*: Woher bekomme ich die Datenbankdatei für die Beispielanwendung?

  Antwort: Die Datenbankdatei gehört zu dem ZIP-Archiv, das Sie aus dem „Source Code"-Abschnitt der Apress-Website (*http://www.apress.com/book/sourcecode*) herunterladen können.

- *Frage*: Muß ich meine Anwendung als *.jar* Datei verpacken?

  Antwort: Die Antwort zu dieser Frage hängt von den spezifischen Anforderungen Ihrer Prüfungsaufgabe ab. Nach unserer Erfahrung wird meistens verlangt, daß die Prüfungskandidaten ihre Lösungen als *.jar* Datei einreichen. Es wird empfohlen, separate *.jar* Dateien in einer *.jar* Gesamtdatei zusammenzupacken.

- *Frage*: Welche Teile muß ich zusammen mit meiner Lösung einreichen?

  Antwort: Die grundlegenden Bestandteile Ihrer Lösung sind:

  - Die Quelltextdateien.
  - Eine ausführbare *.jar* Datei.
  - Die Datenbankdatei(en).
  - Ein Dokument in dem Sie einige der wichtigsten Design-Entscheidungen beschreiben.
  - Eine Datei, in der Sie Ihre Entwicklungsumgebung beschreiben.
  - Die API-Dokumentation (Javadoc).
  - Die Betriebsanleitung für die Benutzer.

  Sie *müssen* allerdings die von Sun Microsystems ausgehändigte Anleitung sorgfältig lesen, um garantiert alle Anforderungen erfüllen zu können. Falls Sie einen Unterschied zwischen unserer Beschreibung und der Anleitung von Sun Microsystems entdecken, müssen Sie sich nach der Anleitung von Sun Microsystems richten.

- *Frage*: Unter welchem Betriebssystem soll ich meine Lösung testen?

  Antwort: Da Java plattformunabhängig ist, sollte Ihre Lösung sowohl unter Windows als auch unter Unix lauffähig sein. Wir empfehlen aber nachdrücklich, daß Sie ihre Lösung unter

Windows, Unix, Linux und Mac OSX testen, da es subtile Unterschiede zwischen den einzelnen Plattformen gibt, die Schwierigkeiten bereiten können. Aus Konsistenzgründen sowie um der Einfachheit Willen, wurden alle Beispiele und Screenshots in diesem Buch unter Windows 2000 getestet beziehungsweise aufgenommen. Es gibt keine Garantie dafür, daß Ihre Anwendung auf einer bestimmten Plattform ausgeführt wird, nachdem Sie sie eingereicht haben. Achten Sie dennoch darauf, Ihre Entwicklungsplattform in der *readme.txt*-Datei zu dokumentieren.

- *Frage*: Was soll ich nach der bestandenen Prüfung tun?

  Antwort: Feiern Sie. Anschließend schicken Sie uns ein E-Mail an die Adresse *scjd@apress.com*.

- *Frage*: Ich habe Fragen oder Kommentare zu diesem Buch. An wen soll ich mich wenden?

  Antwort: Sie erreichen die Autoren unter der E-Mail-Adresse *scjd@apress.com*.

- *Frage*: Welche Online-Resourcen sind nützlich, um sich auf die Prüfung vorzubereiten?

  Antwort: Es gibt viele ausgezeichnete Online-Resourcen. Hier eine Auswahl der Resourcen, die aus unserer Sicht besonders hilfreich sind:

  - Die Website der Java-Ranch (*http://www.javaranch.com*).
  - Die Diskussionsgruppen zum Thema *Sun Certified Java Developer* bei Yahoo! (*http://-groups.yahoo.com/*)
  - Die Dokumentationseite zu Version 5 des Java Development Kits (*http://java.sun.com/j2-se/1.5.0/docs/*).
  - Das „Java-Tutorial“, Abschnitt „RMI“ (*http://java.sun.com/docs/books/tutorial/rmi*).
  - Das „JFC Swing-Tutorial“ (*http://java.sun.com/docs/books/tutorial/uiswing*)
  - Javaworld-Artikel *Sockets programming in Java: A tutorial* unter der Internetadresse *http://www.javaworld.com/jw-12-1996/jw-12-sockets.html*.
  - Das „Portland Pattern Repository“ (*http://c2.com/ppr*)
  - Die „Java Coding Conventions“ (*http://java.sun.com/docs/codeconv/html/CodeConv-TOC.doc.html*)
  - Der „Javadoc Style Guide“ von Sun Microsystems (*http://java.sun.com/j2se/javadoc/wri-tingdoccomments/*)
  - Die „Java Look and Feel Guidelines“ (*http://java.sun.com/products/jlf*)
  - Zwei Websites zu den Themen „Schnittstellenentwurf“ und „Softwaretest“. (*http://www.use-it.com* und *http://www.asktog.de*).

Anhang A

Die Klassen und Interfaces der Beispielanwendung

A.1 Package sampleproject.direct

Bemerkung: Das Package `sampleproject.direct` *enthält keine inneren Klassen.*

- `DvdConnector` (44 Zeilen): Die `getLocal()`-Methode erzeugt ein `DvdDatabase`-Objekt und gibt eine Referenz darauf zurück:

```
public static DBClient getLocal(String dbLocation)
        throws IOException, ClassNotFoundException {
    return new DvdDatabase(dbLocation);
}
```

A.2 Package sampleproject.db

- *`DBClient`* (98 Zeilen): Vorgegebenes Interface. Es gibt genau zwei implementierende Klassen: `sampleproject.db.DvdDatabase` und `sampleproject.sockets.DvdSocketClient`. Deklariert acht Methoden (beachte, daß `DVD` stets in Großbuchstaben geschrieben ist):
  - `public boolean addDVD(DVD dvd) throws IOException` registriert eine neue DVD in der Datenbankdatei.
  - `public DVD getDVD(String UPC) throws IOException` „verleiht“ ein Exemplar einer DVD. Die DVD wird durch ihren eindeutigen UPC beschrieben.

    Frage: Das Interface *`DBClient`* deklariert *keine* Methode, um eine entliehene DVD wieder zurückzugeben. Wie wurde dieses Problem gelöst?
  - `public boolean modifyDVD(DVD dvd) throws IOException` ändert den Datensatz einer in der Datenbankdatei registrierten DVD.
  - `public boolean removeDVD(String UPC) throws IOException` löscht eine registrierte DVD aus der Datenbankdatei. Die DVD wird durch ihren eindeutigen UPC beschrieben.
  - `public` *`List<DVD>`* `getDVDs() throws IOException` gibt eine Liste aller in der Datenbankdatei registrierten DVDs zurück.

  - `public Collection<DVD> findDVD(String query) throws IOException, PatternSyntaxException` gibt eine Kollektion aus den DVDs zurück, die zum Suchkriterium `query` passen. Das Suchkriterium ist ein regulärer Ausdruck.
  - `boolean reserveDVD(String UPC) throws IOException, InterruptedException` reserviert (sperrt) einen DVD-Datensatz *logisch*. Die DVD wird durch ihren eindeutigen UPC beschrieben.
  - `void releaseDVD(String UPC) throws IOException` hebt die *logische* Reservierung eines DVD-Datensatzes auf. Die DVD wird durch ihren eindeutigen UPC beschrieben.

- `DosClient` (260 Zeilen, *wird in der Beispielanwendung nicht gebraucht*): Gehört zur ersten Version der Beispielanwendung und wird nicht mehr verwendet. Siehe Tipp auf Seite 60.
- *`DvdDataAccess`* (91 Zeilen, *wird in der Beispielanwendung nicht gebraucht*): Die Suche nach dem Interface in allen vier Packages der Beispielanwendung per `cgrep DvdDataAccess sampleproject/*/*.java` liefert keinen Treffer. Das Interface *`DvdDataAccess`* hat keine Funktion und wurde vermutlich beim „Aufräumen“ übersehen. `:-(`
- `DvdDatabase implements` *`DBClient`* (180 Zeilen): Schnittstelle zu den Klassen `DvdFileAccess` (Lese- und Schreiboperationen auf der physikalischen Datenbankdatei) und `ReservationsManager` (Logisches Reservieren von zwischengespeicherten Datensätzen). Alle acht im Interface *`DBClient`* deklarierten Methoden sowie `rent()` und `returnRental()` sind nach einem gemeinsamen Schema implementiert (vergleiche `DvdSocketClient`):
  - Die *`DBClient`*-Methoden `addDVD()`, `getDVD()`, `removeDVD()`, `modifyDVD()`, `getDVDs()` und `findDVD()` rufen die entsprechenden `DvdFileAccess`-Methoden auf: `addDvd()`, `getDvd()`, `removeDvd()`, `modifyDvd()`, `getDvds()` und `find()` (ohne „`DVD`“-Endung).
  - Die *`DBClient`*-Methoden `reserveDVD()` und `releaseDVD()` rufen die entsprechenden `ReservationsManager`-Methode `reserveDvd()` und `releaseDvd()` auf.

  Die Methode `setDatabaseLocked()` wird vom Shutdown-Hook (`CleanExit`) aufgerufen, um das `DvdFileAccess`-Objekt zu sperren, welches die Datenbankdatei repräsentiert (siehe Seite 298).

  Frage: Wieviele `DvdFileAccess`-Objekt gibt es im normalen Betrieb der Beispielanwendung?
- `DvdFileAccess` (441 Zeilen):

  `DvdFileAccess` hat zwei lokale innere Klassen:
  - `RecordFieldReader` in der Methode `retrieveDvd()`:
  - `RecordFieldWriter` in der Methode `persistDvd()`:
- `DVD implements` *`Serializable`* (476 Zeilen): JavaBean.
- `ReservationsManager` (102 Zeilen):

A.3 Package sampleproject.remote

Bemerkung: Das Package `sampleproject.remote` *enthält keine inneren Klassen.*

- `DvdConnector` (52 Zeilen): Die `getRemote()`-Methode erzeugt ein `DvdDatabaseImpl`-Objekt und gibt eine Referenz darauf zurück:

```
public static DBClient getRemote(String hostname, String port)
        throws RemoteException {
    String url = "rmi://" + hostname + ":" + port + "/DvdMediator";

    try {
        DvdDatabaseFactory factory
                = (DvdDatabaseFactory) Naming.lookup(url);
        return (DBClient) factory.getClient();
    } catch (NotBoundException e) {
        System.err.println("Dvd Mediator not registered: "
                + e.getMessage());
        throw new RemoteException("Dvd Mediator not registered: ", e);
    } catch (java.net.MalformedURLException e) {
        System.err.println(hostname + " not valid: " + e.getMessage());
        throw new RemoteException("cannot connect to " + hostname, e);
    }
}
```

- `DvdDatabaseFactoryImpl` `extends` `UnicastRemoteObject` `implements` *`DvdDatabaseFactory`* (50 Zeilen):
- *`DvdDatabaseFactory`* `extends` *`Remote`* (20 Zeilen):
- `DvdDatabaseImpl` `extends` `UnicastRemoteObject` `implements` *`DvdDatabaseRemote`* (180 Zeilen):
- *`DvdDatabaseRemote`* `extends` *`Remote`*, *`DBClient`* (14 Zeilen):
- `RegDvdDatabase` (60 Zeilen):
- `RmiFactoryExample` (67 Zeilen):
- `RmiNoFactoryExample` (73 Zeilen):

A.4 Package sampleproject.sockets

Bemerkung: Das Package `sampleproject.sockets` *enthält keine inneren Klassen.*

- `DbSocketRequest` `extends` `Thread` (153 Zeilen): Thread. Empfängt das von einem `DvdSocketClient`-Objekt gesendete serialisierte Kommandoobjekt (`DvdCommand`-Objekt) und transformiert die darin enthaltene Anweisung in einen Methodenaufruf in der „Datenbankschicht“. Die `run()`-Methode des `DbSocketRequest`-Threads öffnet den Ein- und den Ausgabedatenstrom des dem `DbSocketRequest`-Konstruktor übergebenen Sockets. Eine Endlosschleife in `run()` deserialisiert die nacheinander eintreffenden Kommandoobjekte[1], ruft die `execCmdObject()`-Methode auf und sendet das von dieser erhaltene Ergebnisobjekt (`DvdResult`-Objekt) an den clientseitigen Socket zurück. Die `execCmdObject()`-Methode besteht aus einer großen `switch`-Anweisung mit je einem `case`-Zweig pro Element des Aufzählungstyps `SocketCommand`. In jedem `case`-Zweig wird über das im Konstruktor der Klasse `DbSocketRequest` erzeugte `DvdDatabase`-Objekt (*Façade*) die dem Kommandoobjekt entsprechende Methode in der „Datenbankschicht“ aufgerufen, ihr Rückgabewert in einem Ergebnisobjekt verpackt und an die `run()`-Methode zurückgegeben.

  Frage:

[1] **Notiz** (Folgerung): Es gibt eine Socketverbindung *pro Client*, nicht aber eine Socketverbindung pro Anfrage.

  - Die `case`-Zweige für `RENT` und `RETURN` sind auskommentiert. Wie werden „Rent DVD“- beziehungsweise „Return DVD“-Anfragen verarbeitet?
  - Das Hauptfenster gestattet nur zwei Operationen, nämlich „Rent DVD“ und „Return DVD“. Wie rufe ich die übrigen Anfragetypen auf, zum Beispiel `addDVD()`, `removeDVD()` oder `modifyDVD()` („*`DBClient`*-Schreibweise“)?

- `DvdCommand implements` *`Serializable`* (123 Zeilen):
- `DvdConnector` (39 Zeilen): Die `getRemote()`-Methode erzeugt ein `DvdSocketClient`-Objekt und gibt eine Referenz darauf zurück:

```
public static DBClient getRemote(String hostname, String port)
        throws UnknownHostException, IOException {
    return new DvdSocketClient(hostname, port);
}
```

  `DvdSocketClient` baut eine Socketverbindung mit `DvdSocketServer` auf und repräsentiert den clientseitigen Socket der Verbindung. Ein `DvdConnector`-Objekt stellt also eine Verbindung zwischen dem Client (graphische Benutzeroberfläche) und der „Datenbankschicht“ her.

- `DvdResult implements` *`Serializable`* (171 Zeilen):
- `DvdSocketClient implements` *`DBClient`* (317 Zeilen): Fordert via `DvdSocketServer` eine Socketverbindung an und repräsentieren deren clientseitigen Socket. Sendet das von einem `DvdSocketServer`-Objekt empfangene und von einem `DbSocketRequest`-Thread ausgewertete serialisierte Kommandoobjekt (`DvdCommand`-Objekt). Der Konstruktor ruft eine private Hilfsmethode namens `initialize()` auf, die ein `Socket`-Objekt erzeugt und die beiden Felder `oos` und `ois` der Klasse `DvdSocketClient` mit Referenzen auf den Aus- beziehungsweise Eingabedatenstrom des eben erzeugten Sockets bewertet. Alle acht im Interface *`DBClient`* deklarierten Methoden sowie `rent()` und `returnRental()` sind nach einem gemeinsamen Schema implementiert (vergleiche `DvdDatabase`):
  - Die Methode erwartet ein `DVD`-Objekt als Argument (`addDVD()` und `modifyDVD()`) oder sie erzeugt ein `DVD`-Objekt und bewertet es mit einem UPC (`getDVD()`, `rent()`, `returnRental()`, `removeDVD()`, `reserveDVD()` und `releaseDVD()`).

    Zwei Ausnahmen: Bei `getDVDs()` (Liste aller in der Datenbankdatei registrierten DVDs) und `findDVD()` (Liste aller in der Datenbankdatei registrierten DVDs, die zu einem Suchkriterium passen) kommt kein `DVD`-Objekt vor.
  - Die Methode erzeugt ein Kommandoobjekt, wobei der Konstruktor der Klasse `DvdCommand` zwei Argumente erhält: Ein Element aus dem Aufzählungstyp `SocketCommand` (siehe unten) und das im vorigen Schritt erzeugte `DVD`-Objekt, zum Beispiel:

```
DvdCommand cmdObj = new DvdCommand(SocketCommand.ADD, dvd);
```

  - Die Methode übergibt das Kommandoobjekt der privaten Methode `getResultFor()`, die das Kommandoobjekt serialisiert und über die Socketverbindung zum serverseitigen `DvdSocketServer`-Objekt sendet und das von dort zurückerhaltene Ergebnisobjekt (`DvdResult`-Objekt) an ihren Aufrufer zurück gibt:

```
oos.writeObject(command);
DvdResult result = (DvdResult) ois.readObject();
```

  - Die Methode wandelt das von `getResultFor()` erhaltene Ergebnisobjekt in `boolean`, `DVD` (bei `getDVD()`), *`List<DVD>`* (bei `getDVDs()`) beziehungsweise *`Collection<DVD>`* (bei `findDVD()`) um.

Bemerkung: `DvdSocketClient` ist die einzige Klasse in der Beispielanwendung, die eine `finalize()`-Methode enthält. `finalize()` ruft die Hilfsmethode `closeConnections()` auf, welche die Datenströme vom und zum Socket sowie den Socket selbst schließt.

- `DvdSocketServer extends Thread` (88 Zeilen): Thread. Repräsentiert den Serversocket. Der Konstruktor von `DvdSocketServer` wird *nicht* direkt aufgerufen. Die statische Methode `register()` wird (in der Klasse `sampleproject.gui.NetworkStarterSockets`; genau ein Vorkommen in der Beispielanwendung) mit dem Pfad zur Datenbankdatei sowie einer Portnummer aufgerufen und erzeugt und startet einen `DvdSocketServer`-Thread, dessen `run()`-Methode lediglich die Methode `listenForConnections()` aufruft. Die `listenForConnections()`-Methode erzeugt ein `ServerSocket`-Objekt, verknüpft es mit dem gewählten Port und definiert per `setSoTimeout()`, daß die `accept()`-Methode des `ServerSocket`-Objektes höchstens eine Minute lang auf Verbindungsanfragen warten darf. Verstreicht die Wartezeit, ohne daß eine Verbindungsanfrage eingeht, so wird eine Ausnahme vom Typ `SocketTimeoutException` ausgeworfen und der `DvdSocketServer`-Thread abgebrochen. Akzeptiert das `ServerSocket`-Objekt vor Ablauf der Wartezeit eine Verbindungsanfrage, so gibt seine `accept()`-Methode eine Referenz auf ein `Socket`-Objekt zurück, das den serverseitigen Endpunkt der Socketverbindung repräsentiert. Die Verarbeitung der clientseitigen Anfrage geschieht mittels eines Threads vom Typ `DbSocketRequest` (siehe oben).

  Frage: Wirkungsweise der `while`-Schleife in `listenForConnections()` nicht klar. Ein-/Auskommentieren verändert das Verhalten des Programms nicht.

- `SocketCommand` (33 Zeilen): Aufzählungstyp; definiert elf Werte:
  - `UNSPECIFIED`,
  - `FIND`,
  - `RENT`,
  - `RETURN`,
  - `MODIFY`,
  - `ADD`,
  - `REMOVE`,
  - `GET_DVD`,
  - `GET_DVDs`,
  - `RESERVE` und
  - `RELEASE`.

A.5 Package sampleproject.gui

- `ApplicationMode` (16 Zeilen): Aufzählungstyp; definiert drei Werte:
  - `STANDALONE_CLIENT`,
  - `NETWORK_CLIENT` und
  - `SERVER`.

- `ApplicationRunner` (88 Zeilen): Anwendungsspezifischer Klassenlader (*application loader*). Wertet die beim Aufruf per Kommandozeile übergebenen Argumente aus und ruft die zum Starten der Anwendung benötigten Klassen im entsprechenden Modus auf.

  Der Aufruf `java -jar sampleproject.jar` wird per *MANIFEST.MF* auf `ApplicationRunner` abgebildet. Drei Fälle:

  – Aufruf ohne Argument (`java ApplicationRunner`) bewirkt, daß die Beispielanwendung im *stand-alone*-Betriebsmodus gestartet wird.

  – Aufruf mit Argument `alone` (`java ApplicationRunner alone`); wie „Aufruf ohne Argument".

  – Aufruf mit Argument `server` (`java ApplicationRunner server`) bewirkt, daß der Server der Beispielanwendung gestartet wird.

  Andernfalls gibt der `ApplicationRunner`-Konstruktor eine Fehlermeldung aus, die den ungültigen Parameter enthält:

  ```
  INFO: Invalid parameter passed in startup: invalid
  Command line options may be one of:
  ”server” - starts server application
  ”alone”  - starts non-networked client
  ””       - (no command line option): networked client will start
  ```

  Die statische Hilfsmethode `handleException()` wird von verschiedenen Methoden in verschiedenen Klassen aufgerufen und präsentiert den Meldungstext einer Ausnahme mit Hilfe eines Dialogfensters siehe Abbildung 8.11 (rechts) auf Seite 246.

  Frage: Implementiert die Klasse `ApplicationRunner` das *Façade*-Entwurfsmuster? (Aus der API-Dokumentation der Klasse `ApplicationRunner`: „The Denny's DVD application loader — a *facade* to the three modes the application can run in.")

- `CleanExit extends Thread` (66 Zeilen): Shutdown-Hook der Beispielanwendung.

  ```
  public void run() {
      log.info(”Ensuring a clean shutdown”);
      try {
          DvdDatabase database = new DvdDatabase(dbLocation);
          database.setDatabaseLocked(true);
      } catch (IOException e) {
          log.log(Level.SEVERE, ”Failed to lock database before exiting”, e);
      }
  }
  ```

  Ruft die Sperrt das *`ReadWriteLock`*-Objekt

- `ConfigOptions extends JPanel` (420 Zeilen): Die „wiederverwendbare Konfigurationsvorlage" (siehe Unterabschnitt 8.4.3). Zwei Klassen erzeugen in ihrem jeweiligen Konstruktor ein `ConfigOptions`-Objekt: `DatabaseLocationDialog` und `ServerWindow`. Der Konstruktor von `ConfigOptions` erzeugt die zum Erfassen des Pfades zur Datenbankdatei, der Portnummer und des Servertyps erforderlichen Komponenten und legt beim `applicationMode` `SERVER` (siehe Aufzählungstyp `ApplicationMode`) eine Schaltfläche für ein „Datei öffnen"-Dialogfenster an, so daß der Benutzer die Datenbankdatei per Maus auswählen kann. Der Konstruktor verknüpft mit jeder Komponente einen Ereignisbehandler vom Typ `ActionHandler` beziehungsweise `BrowseForDatabase` (beim „Datei öffnen"-Dialogfenster).

  `ConfigOptions` hat drei innere Klassen:

  - `ActionHandler implements` *`ActionListener`*, *`FocusListener`*: Ereignisbehandler „an“ den Texteingabefeldern für den Pfad zur Datenbankdatei und die Portnummer (*`FocusListener`*) sowie an den Radiobuttons für den Servertyp (*`ActionListener`*). Wählt der Benutzer per Mausklick den Servertyp (entweder RMI oder Sockets), so wertet die `actionPerformed()`-Methode den gewählten Servertyp aus und ruft die `updateObservers()`-Methode des `ConfigOptions`-Objektes auf, die letztendlich die Methoden `setChanged()` und `notifyObservers()` des `ConfigObservable`-Objektes aufruft, wodurch die Beobachter benachrichtigt werden. Bewegt der Benutzer den Mauszeiger entweder von außen über eine der beiden `JTextField`-Komponenten oder von dieser fort nach außen, so wird die `focusGained()`- beziehungsweise `focusLost()`-Methode aufgerufen. (Der Methodenkörper von `focusGained()` ist bei `ActionHandler` leer.) Unterscheidet sich die Texteingabe vom Inhalt der `JTextField`-Komponente vor dem Ein- beziehungsweise Austritt des Fokus, so ruft `focusLost()` die `updateObservers()`-Methode des `ConfigOptions`-Objektes auf (die Beobachter werden benachrichtigt).

  - `BrowseForDatabase implements` *`ActionListener`*: Implementiert ein „Datei öffnen“-Dialogfenster, so daß die Datenbankdatei per Maus ausgewählt werden kann. (Die innere Klasse `BrowseForDatabase` enthält wiederum eine anonyme innere Klasse vom Typ `javax.swing.filechooser.FileFilter`.) Nach Auswahl einer Datei ruft die `actionPerformed()`-Methode des `BrowseForDatabase`-Objektes die `updateObservers()`-Methode des `ConfigOptions`-Objektes auf (die Beobachter werden benachrichtigt).

  - `ConfigObservable extends Observable`: Implementiert die Beobachtbarkeit der äußeren Klasse (`ConfigOptions`). Die `ConfigOptions`-Methode `getObservable()` gibt eine Referenz auf das in der Klasse `ConfigOptions` enthaltene `ConfigObservable`-Objekt zurück. Interessierte Beobachter, hier die Objekte der Klassen `DatabaseLocationDialog` und `ServerWindow`, können sich per `getObservable().addObserver(this)` beim `ConfigObservable`-Objekt registrieren.

    Die ausschlaggebende Änderung am Zustand des beobachtenden Objektes wird programmatisch definiert, in dem die `ConfigObservable`-Methode `setChanged()` aufgerufen wird. Anschließend wird, wiederum programmatisch (also *nicht automatisch*), die `ConfigObservable`-Methode `notifyObservers()` mit einem Argument vom Typ `OptionUpdate` aufgerufen, um alle registrierten Beobachter zu benachrichtigen.

- `ConnectionType` (11 Zeilen): Aufzählungstyp; definiert drei Werte:

  - `SOCKET`,

  - `RMI` und

  - `DIRECT`.

- `DatabaseLocationDialog extends WindowAdapter implements` *`ActionListener`*, *`Observer`* (350 Zeilen): Dialogfenster zum Erfassen des Pfades zur Datenbankdatei, des Serverports und des Servertyps (RMI oder Sockets); siehe Abbildung 8.12 (links) auf Seite 247. Die Klasse `DatabaseLocationDialog` kommt nur einmal vor: Der Konstruktor der Klasse `MainWindow` erzeugt ein `DatabaseLocationDialog`, also ein Dialogfenster, um die zuvor aufgezählten Informationen abzufragen. Der Konstruktor von `DatabaseLocationDialog` erzeugt ein `ConfigOptions`-Objekt (wiederverwendbare Konfigurationsvorlage, siehe Abbildung 8.12 (rechts)). Die oben aufgezählten Informationen können über die Methoden `getNetworkType()`, `getLocation()` beziehungsweise `getPort()` abgefragt werden.

- `DvdTableModel extends AbstractTableModel` (137 Zeilen):

- `GuiController` (223 Zeilen): Steuerungskomponente. Führt sämtliche Interaktionen zwischen der Präsentationskomponente (graphische Benutzeroberfläche, `MainWindow`) und der Datenmodellkomponente aus. Die Methoden `find()`, `findDvd()` und `getDvds()` geben eine Referenz auf ein `DvdTableModel`-Objekt zurück. Die Methode `rent()` und `returnRental()` buchen eine DVD entweder aus oder ein und de- beziehungsweise inkrementieren die Anzahl der vorrätigen Exemplare. Der Konstruktor vom `GuiController` und die fünf Methoden können eine Ausnahme von Typ `GuiControllerException` auswerfen, in der stets eine andere Ausnahme (`ClassNotFoundException`, `Exception`, `IOException`, `InterruptedException`, `PatternSyntaxException` oder `RemoteException`) verpackt ist.

  `GuiController` referenziert ein *`DBClient`*-Feld `connection`, welches vom `GuiController`-Konstruktor je nach `ConnectionType` (`RMI`, `SOCKET` oder `DIRECT`) mit einer Referenz auf ein „Datenbank-Objekt“ bewertet wird:

```
// Aus "sampleproject.direct.DvdConnector"
public static DBClient getLocal(String dbLocation)
        throws IOException, ClassNotFoundException {
    return new DvdDatabase(dbLocation);
}

// Aus "sampleproject.remote.DvdConnector"
public static DBClient getRemote(String hostname, String port)
        throws RemoteException {
    String url = "rmi://" + hostname + ":" + port + "/DvdMediator";

    try {
        DvdDatabaseFactory factory = (DvdDatabaseFactory) Naming.lookup(url);
        return (DBClient) factory.getClient();
    } catch (NotBoundException e) {
        System.err.println("Dvd Mediator not registered: " + e.getMessage());
        throw new RemoteException("Dvd Mediator not registered: ", e);
    } catch (java.net.MalformedURLException e) {
        System.err.println(hostname + " not valid: " + e.getMessage());
        throw new RemoteException("cannot connect to " + hostname, e);
    }
}

// Aus "sampleproject.socket.DvdConnector"
public static DBClient getRemote(String hostname, String port)
        throws UnknownHostException, IOException {
    return new DvdSocketClient(hostname, port);
}
```

  Bemerkung: Die Klassen `sampleproject.db.DvdDatabase` und `sampleproject.sockets.DvdSocketClient` implementieren das vorgegebene Interface *`DBClient`*. Die *`DvdDatabaseFactory`*-Methode `getClient()` gibt eine Referenz vom Typ *`DvdDatabaseRemote`* (ein von *`DBClient`* abgeleitetes Interface) zurück.

- `GuiControllerException extends Exception` (36 Zeilen): Siehe `GuiController`.

- `MainWindow extends JFrame` (347 Zeilen): Hauptfenster der Beispielanwendung. Das `MainWindow`-Objekt referenziert ein `GuiController`- (siehe oben), ein `JTable`- (die Tabellenkomponente) und ein `DvdTableModel`-Objekt (in der Tabellenkomponente angezeigte Datensätze; siehe oben). Abbildung 8.11 (links) auf Seite 246 zeigt das von `MainWindow` und `DvdScreen` erzeugte Hauptfenster. Der Konstruktor von `MainWindow` erzeugt ein `DatabaseLocationDialog`-Objekt, um den Pfad zur Datenbankdatei, den Serverport und -typ (RMI oder Sockets) abzufragen.

`MainWindow` hat fünf innere Klassen:

- `DvdScreen extends JPanel`: Hilfsklasse. Baut den Inhalt des Hauptfensters auf und verkürzt dadurch die Anzahl der Anweisungen im Konstruktor von `MainWindow`. Setzt die Tabellenkomponente in eine `JScrollPane`-Komponente (Container mit Bildlaufleiste) ein. Legt die Schaltflächen „Search“ mit zugehörigem Texteingabefeld (zusammengefaßt im `searchPanel`), „Rent DVD“ und „Return DVD“ (zusammengefaßt im `hiringPanel`) an, verknüpft sie mit den gleichnamigen Ereignisbehandlern und weißt jeder Komponente einen Tooltip zu. Faßt die `JPanel`s `searchPanel` und `hiringPanel` zum `JPanel` `bottomPanel` zusammen. Konfiguriert die Tabellenkomponente so, daß stets höchstens eine Zeile ausgewählt werden kann (`ListSelectionModel.SINGLE_SELECTION`) und daß die Spaltenbreite automatisch bestimmt wird (`JTable.AUTO_RESIZE_ALL_COLUMNS`).
- `RentDVD implements` *`java.awt.event.ActionListener`*:
- `ReturnDVD implements` *`ActionListener`*:
- `SearchDVD implements` *`ActionListener`*:
- `QuitApplication implements` *`ActionListener`*: Ereignisbehandler „am“ Menüpunkt „File|Quit“ (einziges Vorkommen in der Beispielanwendung). Die `actionPerformed()`-Methode beendet die Beispielanwendung per `System.exit(0)`.

Siehe auch `ServerWindow`.

Bemerkung: Das `JFrame`-Feld `mainWindow` wird nicht initialisiert. Die Suche nach dem `mainWindow`-Feld im gesamten Package `sampleproject.gui` per `cgrep mainWindow sampleproject/gui/*.java` liefert genau einen Treffer. Das Feld `mainWindow` hat keine Funktion und wurde vermutlich durch `mainTable` ersetzt und beim „Aufräumen“ übersehen. `:-(`

- `NetworkStarterRmi` (103 Zeilen): Startet die RMI-Registratur (ruft die `RegDvdDatabase`-Methode `register()` mit dem Pfad zur Datenbankdatei und dem Port auf, über den die RMI-Registratur erreicht werden kann). Sichert die Benutzereingaben via `SavedConfiguration` in der Properties-Datei *dennys.properties* (siehe unten).
- `NetworkStarterSockets` (82 Zeilen): Startet den Serversocket (ruft die `DvdSocketServer`-Methode `register()` mit dem Pfad zur Datenbankdatei und dem Port auf, über den der Serversocket erreicht werden kann). Sichert die Benutzereingaben via `SavedConfiguration` in der Properties-Datei *dennys.properties* (siehe unten).
- `OptionUpdate implements` *`Serializable`* (105 Zeilen): Hilfsklasse (*Value-Object*) zur Datenübertragung zwischen dem beobachteten Objekt (`ConfigOptions`) und den Beobachtern (`DatabaseLocationDialog` und `ServerWindow`) in der dortigen Implementierung des *Observer*-Entwurfsmusters beim Aufruf der `Observable`-Methode `notifyObservers()`. Das zwischen dem beobachteten Objekt und seinem Beobachter übertragene `OptionUpdate`-Objekt enthält zwei Informationen: den Namen der Eigenschaft deren Wert sich geändert hat (einen Wert des Aufzählungstyps `Updates`) und den neuen Eigenschaftswert (`payload`-Feld vom Typ `Object`; *payload*: Ladung, Nutzlast).
- `PositiveIntegerField extends JTextField` (116 Zeilen): Textfeld zum Erfassen der Portnummer (erlaubt nur positive ganzzahlige Eingaben). Einziges Vorkommen in `ConfigOptions` (wiederverwendbare Konfigurationsvorlage).

  `PositiveIntegerField` hat eine innere Klasse:

  - `NumericDocument extends PlainDocument`:

Frage: Funktion/Wirkungsweise von „Dokumenten“ (von `PlainDocument` abgeleitete Klassen) bei Texteingabefeldern unklar.

- `SavedConfiguration` (175 Zeilen): Sichert beziehungsweise liest den vom Benutzer übergebenen Pfad zur Datenbankdatei, Servertyp (RMI oder Sockets), Serverport und die IP-Adresse des Rechners auf dem sich die RMI-Registratur befindet in beziehungsweise aus der Properties-Datei *dennys.properties*. Die im `Properties`-Objekt beziehungsweise in der Datei verwendeten Schlüssel sind durch die folgenden vier Konstanten definiert: `DATABASE_LOCATION`, `NETWORK_TYPE`, `SERVER_ADDRESS` und `SERVER_PORT`. Die Klasse `SavedConfiguration` verfügt über je eine Abfrage- und eine Änderungsmethode für einzelne „Properties“ (`getParameter()`, `setParameter()`), eine statische Methode `getSavedConfiguration()`, die eine Referenz auf das statische `SavedConfiguration`-Objekt (*Singleton*, siehe Bemerkung unten) zurückgibt sowie über die Hilfsmethoden `loadParametersFromFile()` und `saveParametersToFile()`, um die Properties-Datei einzulesen beziehungsweise zu schreiben.

  Bemerkung: Die Klasse `SavedConfiguration` implementiert in trivialer Weise das *Singleton*-Entwurfsmuster. Das *statische* Feld `savedConfiguration` referenziert stets ein `SavedConfiguration`-Objekt, das heißt es existiert stets genau ein Objekt dieser Klasse. Die statische Methode `getSavedConfiguration()` gibt eine Referenz auf dieses Objekt zurück.

- `ServerWindow extends JFrame implements` *`Observer`* (241 Zeilen): Dialogfenster („Serverfenster“) zum Erfassen des Pfades zur Datenbankdatei, des Serverports und des Servertyps (RMI oder Sockets); siehe Abbildung 8.14 (links) auf Seite 257. Bei laufendem Server ist das Serverfenster deaktiviert („ausgegraut“), siehe Abbildung 8.14 (rechts) Der Konstruktor von `ServerWindow` erzeugt ein `ConfigOptions`-Objekt (wiederverwendbare Konfigurationsvorlage, siehe Abbildung 8.12 (rechts) auf Seite 247).

  `ServerWindow` hat eine innere Klasse:

  - `StartServer implements` *`ActionListener`*: Ereignisbehandler „an“ der „Start server“-Schaltfläche des Serverfensters. Die `actionPerformed()`-Methode deaktiviert alle Komponenten mit Ausnahme der „Exit“-Schaltfläche, registiert den Shutdown-Hook (`CleanExit`, siehe Seite 259):

    ```
    Thread exitRoutine = new CleanExit(databaseLocation);
    Runtime.getRuntime().addShutdownHook(exitRoutine);
    ```

    und erzeugt je nach Servertyp entweder ein `NetworkStarterRmi`- oder `NetworkStarterSockets`-Objekt.

  Siehe auch `MainWindow`.

- `Updates` (30 Zeilen): Aufzählungstyp in `OptionUpdate`; definiert drei Werte:
  - `NETWORK_CHOICE_MADE`,
  - `DB_LOCATION_CHANGED` und
  - `PORT_CHANGED`.

Name	Typ	Package unter sampleproject	Abstammung	Quelltext auf Seite	Erläuterungen auf Seite
ActionHandler	Innere Klasse in ConfigOptions.	gui	implements *ActionListener*, *FocusListener*	??	299
ApplicationMode	Aufzählungstyp. Drei Werte: STANDALONE_CLIENT, NETWORK_CLIENT und SERVER.	gui		??	297
ApplicationRunner	Klasse	gui		??	298
BrowseForDatabase	Innere Klasse in ConfigOptions (Enthält eine anonyme innere Klasse vom Typ FileFilter).	gui	implements *ActionListener*	??	299
CleanExit	Klasse (Shutdown-Hook)	gui	extends Thread	??	298
ConfigObservable	Innere Klasse in ConfigOptions.	gui	extends Observable	??	299
ConfigOptions	Klasse (hat drei innere Klassen)	gui	extends JPanel	??	298
ConnectionType	Aufzählungstyp. Drei Werte: SOCKET, RMI und DIRECT.	gui		??	299
DatabaseLocationDialog	Klasse	gui	extends WindowAdapter implements *ActionListener*, *Observer*	??	299
DBClient	Interface	db		??	293
DbSocketRequest	Klasse	sockets	extends Thread	??	295
DosClient, gehört nicht zur Beispielanwendung.	Klasse	db		??	294
DvdCommand	Klasse	sockets	implements *Serializable*	??	296
DvdConnector	Klasse	direct		??	293
DvdConnector	Klasse	remote		??	294
DvdConnector	Klasse	sockets		??	296
DvdDataAccess, gehört nicht zur Beispielanwendung.	Interface	db		??	294

DvdDatabaseFactoryImpl	Klasse	remote	extends UnicastRemoteObject implements *DvdDatabaseRemote*	??	295
DvdDatabaseFactory	Interface	remote	extends *Remote*	??	295
DvdDatabaseImpl	Klasse	remote	extends UnicastRemoteObject implements *DvdDatabaseRemote*	??	295
DvdDatabaseRemote	Interface	remote	extends *Remote*, *DBClient*	??	295
DvdDatabase	Klasse	db	implements *DBClient*	??	294
DvdFileAccess	Klasse (hat zwei lokale innere Klassen)	db		??	294
DvdResult	Klasse	sockets	implements *Serializable*	??	296
DvdScreen	Innere Klasse in MainWindow.	gui	extends JPanel	??	301
DvdSocketClient	Klasse	sockets	implements *DBClient*	190–195,	296
DvdSocketServer	Klasse	sockets	extends Thread	??	297
DvdTableModel	Klasse	gui	extends AbstractTableModel	??	299
DVD	Klasse (JavaBean)	db	implements *Serializable*	??	294
GuiControllerException	Klasse (Ausnahme)	gui	extends Thread	??	300
GuiController	Klasse	gui		??	300
MainWindow	Klasse (hat fünf innere Klassen)	gui	extends JFrame	??	300
NetworkStarterRmi	Klasse	gui		??	301
NetworkStarterSockets	Klasse	gui		??	301
NumericDocument	Innere Klasse in PositiveIntegerField.	gui	extends PlainDocument	??	301
OptionUpdate	Klasse (definiert einen Aufzählungstyp)	gui	implements *Serializable*	??	301
PositiveIntegerField	Klasse (hat eine innere Klasse)	gui	extends JTextField	??	301
QuitApplication	Innere Klasse in MainWindow.	gui	implements *ActionListener*	??	301
RecordFieldReader	Lokale innere Klasse in retrieveDvd() (DvdFileAccess).	db		??	294

RecordFieldWriter	Lokale innere Klasse in persistDvd() (DvdFileAccess).	db		??	294
RegDvdDatabase	Klasse	remote		??	295
RentDVD	Innere Klasse in MainWindow.	gui	implements *ActionListener*	??	301
ReservationsManager	Klasse	db		??	294
ReturnDVD	Innere Klasse in MainWindow.	gui	implements *ActionListener*	??	301
RmiFactoryExample, gehört nicht zur Beispielanwendung.	Klasse	remote	extends Thread	??	295
RmiNoFactoryExample, gehört nicht zur Beispielanwendung.	Klasse	remote	extends Thread	??	295
SavedConfiguration	Klasse	gui		??	302
SearchDVD	Innere Klasse in MainWindow.	gui	implements *ActionListener*	??	301
ServerWindow	Klasse (hat eine innere Klasse)	gui	extends JFrame implements *Observer*	??	302
SocketCommand	Aufzählungstyp. Elf Werte: UNSPECIFIED, FIND, RENT, RETURN, MODIFY, ADD, REMOVE, GET_DVD, GET_DVDs, RESERVE und RELEASE.	sockets		??	297
StartServer	Innere Klasse in ServerWindow.	gui	implements *ActionListener*	??	302
Updates	Aufzählungstyp in OptionUpdate.	gui		??	302

Tabelle 1.1: Tabellarische Übersicht über die Klassen und Interfaces der Beispielanwendung