2 Professionelle Arbeitsumgebung

Das Bearbeiten von Sourcecode ist eine wichtige und elementare Aufgabe bei der Softwareentwicklung. Genau wie Handwerker benötigen auch wir Softwareentwickler eine gut strukturierte Werkbank (Arbeits-/Entwicklungsumgebung) mit nützlichen Werkzeugen (Tools). In diesem Kapitel werden wir damit beginnen, eine Arbeitsumgebung einzurichten, die sowohl das Entwickeln erleichtert als auch Unterstützung beim Testen bietet und dadurch hilft, Fehler zu reduzieren.

Den zentralen Anlaufpunkt, unsere Steuerzentrale, bildet eine sogenannte integrierte Entwicklungsumgebung, kurz IDE. Ich motiviere in Abschnitt 2.1 den Einsatz einer solchen und nenne drei mögliche Kandidaten, wobei in diesem Buch die Wahl auf Eclipse als frei verfügbare IDE fällt. In Abschnitt 2.2 betrachten wir zwei Vorschläge zur Organisation eines Softwareprojekts, also zur Strukturierung von Sourcecode und anderen Dateien (Bilder, Texte, Testcode usw.). Abschnitt 2.3 beschäftigt sich dann mit dem Thema Versionsverwaltungen. Diese helfen dabei, die Dateien eines Projekts sicher zu speichern und verschiedene Versionen davon abrufen zu können. Dazu werde ich einführend auf zentrale und dezentrale Versionsverwaltungen am Beispiel der Open-Source-Tools CVS, Subversion (SVN) sowie Git und Mercurial eingehen. Die professionelle Softwareentwicklung umfasst nicht nur die Implementierung und die spätere Auslieferung, sondern vor allem auch das Testen unserer Programme. In Abschnitt 2.4 werfen wir daher einen kurzen Blick auf das Erstellen von Unit Tests mithilfe von JUnit. Selbst wenn viele Unit Tests existieren und unsere Programme sehr gewissenhaft testen, so wird es doch immer mal wieder zu unerklärlichem Programmverhalten oder gar Fehlern kommen. Für solche Fälle ist es zur Fehlersuche wünschenswert und hilfreich, das Programm schrittweise ausführen zu können, es bei Bedarf an einer bestimmten Stelle zu unterbrechen und dann die Wertebelegungen von Variablen überprüfen zu können. Das Ganze ist mithilfe eines sogenannten Debuggers möglich. Eine Einführung in die Thematik bietet Abschnitt 2.5. Ausgewählte lauffähige Stände wollen wir an Kunden veröffentlichen. Abschnitt 2.6 gibt einen Überblick über die Auslieferung (Packaging und Deployment) von Java-Programmen. Dazu lernen wir JAR-Dateien (Java Archive) kennen. Abschließend stelle ich die Vorteile eines von der IDE unabhängigen Build-Prozesses heraus, mit dem es möglich ist, Programme zu kompilieren, zu testen, zu starten, Auslieferungen zu erzeugen usw. In Abschnitt 2.7 beschreibe ich dazu die Build-Tools Maven und insbesondere Gradle. Letzteres wird zur Ausführung aller im Buch vorgestellten Beispielapplikationen genutzt.

2.1 Vorteile von IDEs am Beispiel von Eclipse

Zum Bearbeiten von Sourcecode empfehle ich den Einsatz einer IDE anstelle von Texteditoren. Zwar kann man für kleinere Änderungen auch mal einen Texteditor nutzen, aber dieser bietet nicht die Annehmlichkeiten einer IDE: Dort können Kompiliervorgänge und Sourcecodeanalysen automatisch und im Hintergrund stattfinden, wodurch gewisse Softwaredefekte direkt noch während des Editierens erkannt und angezeigt werden können, etwa in einer To-do-/Task-Liste. IDEs bereiten zudem vielfältige Informationen auf. Das erlaubt unter anderem die Anzeige von Ableitungshierarchien und das Auffinden von Klassen über deren Namen. Auch das Verknüpfen der JDK-Klassen mit deren Sourcecode und das Anzeigen zugehöriger Dokumentation sind Vorteile von IDEs. Weiterhin werden automatische Transformationen und Änderungen von Sourcecode, sogenannte ***Refactorings***, unterstützt.

Für Java bieten sich verschiedene IDEs an. Sowohl Eclipse als auch NetBeans sind kostenlos. IntelliJ IDEA gibt es als kostenlose Community Edition sowie als kostenpflichtige Ultimate Edition. Alle IDEs haben ihre speziellen Vorzüge, aber auch Nachteile. Entscheiden Sie selbst und besuchen Sie dazu folgende Internetadressen:

- *www.eclipse.org*
- *www.jetbrains.com/idea*
- *www.netbeans.org*

Die genannten IDEs lassen sich durch die Integration von Tools, etwa Sourcecode-Checkern, Versionsverwaltungen, XML-Editoren, Datenbank-Tools, Profiling-Tools usw., erweitern. Im folgenden Text werde ich speziell auf Eclipse und entsprechende Tools eingehen. Nach der Lektüre dieses Kapitels haben Sie dann bereits eine Arbeitsumgebung, die professionelles Entwickeln ermöglicht. Im Verlauf des Buchs werden thematisch passende, arbeitserleichternde Erweiterungen vorgestellt.

Basiskonfiguration für Eclipse

Zur Fehlervermeidung und Qualitätssicherung empfiehlt es sich, den Sourcecode regelmäßig zu analysieren und dabei die Einhaltung gewisser Regeln und Standards zu forcieren. In einem ersten Schritt kann man dazu auf die in Eclipse integrierte Sourcecode-Prüfung zurückgreifen. Umfangreichere Tests bieten Tools wie Checkstyle, FindBugs und PMD (vgl. Abschnitt 19.4).

Die in Eclipse integrierten Sourcecode-Prüfungen lassen sich im Einstellungsdialog WINDOW –> PREFERENCES konfigurieren. Dort wählen wir im Baum den Eintrag JAVA –> COMPILER –> ERRORS/WARNINGS. Im zugehörigen Dialog nehmen wir Anpassungen in den Bereichen CODE STYLE, POTENTIAL PROGRAMMING PROBLEMS, UNNECESSARY CODE sowie NULL ANALYSIS vor. Abbildung 2-1 zeigt eine mögliche, sinnvolle Einstellung der Werte im Abschnitt CODE STYLE. Auch in den anderen Sektionen können Sie bei Interesse strengere Auswertungen wählen. Experimentieren Sie ruhig ein wenig mit den Einstellungen.

Abbildung 2-1 *Konfiguration von Java –> Compiler –> Errors/Warnings –> Code style*

Code style Methoden, die genauso heißen wie die Klasse selbst, also dadurch sehr leicht mit einem Konstruktor verwechselt werden können, werten wir als `Error`. Ein unqualifizierter Zugriff auf Attribute wird ignoriert, genauso wie nicht-externalisierte Strings und die Möglichkeit, Methoden `static` zu machen. Alle anderen Werte setzen wir auf `Warning`.

Potential programming problems Konvertierungen mithilfe von Auto-Boxing und -Unboxing stellen wir auf `Warning`. Eine versehentliche Zuweisung in einer booleschen Bedingung betrachten wir als `Error`. Alles andere stellen wir auf `Warning`.

Unnecessary code Unnötige `else`-Anweisungen sollen ignoriert werden, da es lediglich eine Stilfrage ist, ob man bei Auswertung einer Bedingung durch das `else` die Behandlung des anderen Zweiges darstellen möchte oder nicht. Einen ungenutzten Exception-Parameter setzen wir auf `ignore`. Die restlichen Prüfungen stellen wir auf `Warning`.

Null analysis Um uns vor Problemen mit unerwarteten `null`-Werten und Zugriffen darauf zu bewahren, stellen wir in dieser Sektion alles mindestens auf `Warning`.

2.2 Projektorganisation

Im Sourcecode erleichtert eine einheitliche Formatierung und eine sinnvolle Namensgebung die Verständlichkeit und gute Lesbarkeit. Wenn man das Ganze auf die Dateien eines Projekts überträgt, erleichtert eine einheitliche Projektstruktur die Orientierung in fremden (und auch in eigenen) Projekten. Daher stelle ich nun zwei bewährte Varianten vor, die Sie als Vorschlag für die Verzeichnisstruktur eigener Projekte nutzen können.

Als Erstes gehe ich auf die Projektstruktur ein, die entsteht, wenn man mit Eclipse ein Java-Projekt anlegt. Als Zweites beschreibe ich eine Projektstruktur, die sich durch die Verbreitung von Maven als Build-Tool als De-facto-Standard etabliert hat. Die Beispiele dieses Buchs folgen dieser zweiten Konvention.

2.2.1 Projektstruktur in Eclipse

Wenn man Projekte mit Eclipse anlegt und deren Inhalt verwaltet, so verwendet man ein Hauptverzeichnis mit dem Namen oder Synonym des Projekts. Nahezu alle anderen Daten werden in Unterordnern abgelegt. Lediglich einige (in Eclipse zunächst ausgeblendete) Metadateien (`.project` und `.classpath`) findet man im Hauptverzeichnis. Die Projektstruktur wird zunächst grafisch dargestellt und danach kurz beschrieben:

```
+ <project-root>
|
+---src                          - Sourcecode unserer Applikation
|   +---ui                       - Grafische Oberfläche
|       +---FileDialog.java      - Die Klasse FileDialog
|
+---test                         - Sourcecode der Unit Tests
|   +---ui                       - Tests für die grafische Oberfläche
|       +---FileDialogTest.java  - Tests für die Klasse FileDialog
|
+---lib                          - Externe Klassenbibliotheken (JARs)
|   +---junit_4.11               - Ordner zur Strukturierung
|       +---junit.jar            - Wichtig zum Übersetzen der Unit Tests
|       +---hamcrest-core.jar    - Von JUnit benötigt
|
+---config                       - Konfigurationsdateien
|   +---images                   - Bilder
|   +---texts                    - Sprachressourcen
|
+---docs                         - Dokumentation
+---generated-reports            - Erzeugte Berichte (JUnit, Checkstyle usw.)
+---bin                          - Kompilierte Klassen und JAR der Applikation
```

Sourcecode und Tests Den Sourcecode legen wir im Verzeichnis `src` ab. Darunter erfolgt die Aufteilung in Form von Packages. Parallel dazu werden Tests in einem Ordner `test` abgelegt. Dabei verwenden wir hier eine Spiegelung der Package-Struktur des `src`-Ordners. Das bietet zweierlei Vorteile: Erstens trennt man so Tests und Applikationscode im Dateisystem, wodurch sich ungewünschte Abhängigkeiten leicht erkennen lassen und später bei Auslieferungen die Testklassen nicht Bestandteil des Programms sein müssen. Zweitens liegen die Tests dadurch logisch in gleichen

Packages wie der korrespondierende Sourcecode,[1] was den Zugriff auf alle Elemente des Packages (außer den privaten) möglich macht.

Weitere Dateien In der Regel nutzt man zur Realisierung von Projekten verschiedene Klassenbibliotheken. Selbst in diesem einfachen Beispiel besitzen wir durch die Testklasse eine Abhängigkeit zur JUnit-Bibliothek. JUnit wird durch Eclipse bereits mitgeliefert und automatisch verwaltet. Andere Bibliotheken (oder eine aktuellere Version von JUnit) werden in einem Ordner `lib` gesammelt. Dabei erleichtert eine gut gewählte Verzeichnishierarchie die Übersicht über die verwendeten Bibliotheken und deren Versionen. Häufig sind für ein Projekt auch verschiedene Konfigurationsdateien zu verwalten. Dazu bietet sich ein Ordner namens `config` an. Dort können z. B. Textressourcen für Sprachvarianten und Konfigurationen für Logging usw. abgelegt werden. Verschiedene Arten von Dokumentation speichert man im Verzeichnis `docs`.

Tipp: Bibliotheken (JARs) in Eclipse einbinden

Wenn Sie Klassen aus Bibliotheken nutzen wollen, so liegen diese in Form von JARs (Java Archive) vor. Um diese in einem Eclipse-Projekt zu nutzen, gehen Sie wie folgt vor: Klicken Sie auf Ihr Projekt und wählen Sie im Kontextmenü BUILD PATH –> CONFIGURE BUILD PATH und in dem erscheinenden Dialog wählen Sie links den Eintrag JAVA BUILD PATH und dort den Tab LIBRARIES. Mithilfe der Buttons ADD JAR... und ADD EXTERNAL JAR... können Sie die gewünschten Bibliotheken einbinden. Ersteres wählen Sie, wenn sich die JAR-Dateien innerhalb Ihres Projekts (z. B. im Verzeichnis `lib`) befinden. Mithilfe von ADD EXTERNAL JAR... kann man JARs auch aus externen Verzeichnissen einbinden, etwa einem dedizierten Installationsverzeichnis einer Datenbank o. Ä.

Beim professionellen Programmieren sollte die Verwaltung von Abhängigkeiten und das Einbinden von Fremdbibliotheken über ein Build-Tool wie Maven oder Gradle erfolgen. Details dazu beschreibt Abschnitt 2.7.

Generierte Dateien Gemäß der Faustregel »***Trenne generierte Dateien von regulärem Sourcecode***« werden generierte Dateien in separaten Verzeichnissen abgelegt. Vom Compiler generierte `.class`-Dateien liegen im Ordner `bin`. Durch Tests und andere Sourcecode-Prüfungen entstehende Berichte werden in einem Verzeichnis `generated-reports` gesammelt. Diese Aufteilung erleichtert die Übersicht und Trennung, was wiederum die später beschriebene Versionsverwaltung sowie die Automatisierung von Build-Läufen vereinfacht.

[1] Wenn man in der IDE den Ordner `test` als zusätzliches Sourcecode-Verzeichnis wählt.

Auslieferungen und Releases

Normalerweise wird eine Applikation nicht nur innerhalb der IDE laufen, sondern vor allem als eigenständige Applikation. Häufig sollen davon auch verschiedene Versionen, sogenannte ***Releases***, erzeugt werden. Einige davon stellen stabile Softwarestände dar und können als offizielle Version oder ***Auslieferung*** an Kunden übergeben werden.

Für die aktuelle Programmversion bietet sich die Speicherung in einem Ordner `release` an. Damit die Applikation tatsächlich unabhängig von der IDE ausgeführt werden kann, müssen im Ordner `release` alle benötigten externen Bibliotheken, Konfigurationsdateien usw. bereitgestellt werden (z. B. durch Kopie) oder zugreifbar sein. Deren Pfade müssen in den sogenannten `CLASSPATH` aufgenommen werden. Dies ist die Menge von Verzeichnissen und Dateien, in der die JVM nach Klassen und anderen Ressourcen, etwa Bildern, sucht und die als Startparameter der JVM gesetzt werden kann. Auf das Classloading gehe ich kurz in Anhang A ein.

```
+ <project-root>
|
+---release                  - Release-Ordner
    +---lib                  - Kopie des lib-Verzeichnisses
    +---config               - Kopie des config-Verzeichnisses
    +---app.jar              - Applikation als JAR
```

Schwachpunkte der gezeigten Projektstruktur

Die dargestellte Verzeichnisstruktur eignet sich für viele Projekte recht gut, besitzt aber Schwachstellen. Zunächst einmal muss die Verzeichnisstruktur (bzw. Teile davon) für jedes Projekt erneut von Hand angelegt und auch gepflegt werden. Dabei besteht die Gefahr, dass sich Inkonsistenzen einschleichen: Heißt der Ordner mit den entstehenden Klassen *bin* oder *build*? Und wie derjenige mit den Fremdbibliotheken? In einem Projekt etwa *lib*, im anderen *libs*. Das setzt sich bei der Vergabe der Namen von Unterordnern fort: Wird ein solcher *junit4* genannt oder *junit4.11* oder aber *junit_4.11*?

2.2.2 Projektstruktur für Maven und Gradle

Nachfolgend schauen wir uns die von Maven und Gradle genutzte und weitverbreitete Verzeichnisstruktur als Alternative zu der von Eclipse standardmäßig verwendeten an.

Einheitliches Projektlayout

Das Problem möglicher Inkonsistenzen bezüglich der genutzten Verzeichnisse für kompilierte Klassen, Fremdbibliotheken oder Reports usw. wird von den beiden Build-Tools Maven und Gradle adressiert, indem diese eine einheitliche Verzeichnisstruktur für alle Projekte fordern. Dadurch ist immer klar, wo sich Dateien gewünschten Inhalts befinden, etwa für Sourcen und Tests in den Verzeichnissen `src/main/java` bzw. `src/test/java`. Darüber hinaus benötigte Ressourcendateien werden wiederum in getrennten Verzeichnissen hinterlegt, nämlich wie folgt:

```
+ <project-root>
|
+---src
    +---main
    ¦   +---java            - Java-Klassen
    ¦   +---resources       - Konfigurationsdateien
    ¦
    +---test
        +---java            - Testklassen
        +---resources       - Konfigurationsdateien für Testklassen
```

Projektlayout am Beispiel Nutzen wir die obige Projektstruktur für eine einfache Hello-World-Applikation, so ergibt sich folgendes Verzeichnislayout, unter der Annahme, dass wir die Applikation mit einer Klasse `App.java` und die dazugehörige Testklasse `AppTest.java` im Package `de.javaprofi.helloworld` realisieren:

```
+---src
    +---main
    ¦   +---java
    ¦       +---de
    ¦           +---javaprofi
    ¦               +---helloworld
    ¦                       App.java
    ¦
    +---test
        +---java
            +---de
                +---javaprofi
                    +---helloworld
                            AppTest.java
```

Wenn man das Projekt mit Maven kompiliert und paketiert (`mvn package`), so entsteht ein Verzeichnis `target` als Ziel für kompilierte Klassen und weitere erzeugte Dateien wie z. B. Testresultate. Diese Struktur ist nachfolgend zum besseren Verständnis auszugsweise und gekürzt abgebildet:

```
+---target
    ¦   helloworld-1.0-SNAPSHOT.jar
    ¦
    +---classes
    ¦   +---de
    ¦       +---javaprofi
    ¦           +---helloworld
    ¦                   App.class
    ¦
    +---surefire-reports
    ¦       de.javaprofi.helloworld.AppTest.txt
    ¦       TEST-de.javaprofi.helloworld.AppTest.xml
    ¦
    +---test-classes
        +---de
            +---javaprofi
                +---helloworld
                        AppTest.class
```

Abweichungen im Projektlayout mit Gradle Gradle verwendet für Sourcen, Tests und Ressourcen die Maven-Konventionen zur Strukturierung der Verzeichnisse. Allerdings besitzen die erzeugten Verzeichnisse einen leicht abweichenden Standard. Es entsteht ein Verzeichnis `build` mit verschiedenen Unterverzeichnissen wie folgt:

```
+ <project-root>
¦
+---build
¦   +---classes
¦   ¦   +---main
¦   ¦   ¦   +---de
¦   ¦   ¦       +---javaprofi
¦   ¦   ¦           +---helloworld
¦   ¦   ¦                   App.class
¦   ¦   ¦
¦   ¦   +---test
¦   ¦       +---de
¦   ¦           +---javaprofi
¦   ¦               +---helloworld
¦   ¦                       AppTest.class
¦   ¦
¦   +---dependency-cache
¦   +---libs
¦   ¦       helloworld.jar
¦   ¦
¦   +---reports
¦   ¦   +---tests
¦   ¦       ¦   index.html
...
```

Verwaltung externer Abhängigkeiten

Eine Sache könnte uns noch wundern: Wo und wie werden denn die Abhängigkeiten zu externen Bibliotheken beschrieben? Wir finden in der Verzeichnisstruktur keinen Ordner mit Bibliotheken. Exakt! Während wir für das Eclipse-Projekt bzw. bei Builds mit Ant diese Verwaltung noch selbst erledigen mussten, helfen uns hier die Build-Tools Maven und Gradle, indem sie sich um die Auflösung und Bereitstellung externer Bibliotheken kümmern. Details dazu lernen wir später in Abschnitt 2.7 kennen.

2.3 Einsatz von Versionsverwaltungen

Nachdem wir zwei Alternativen zur Strukturierung von Projekten kennengelernt haben, widmen wir uns nun dem Thema Versionsverwaltungen. Diese ermöglichen die Speicherung und Verwaltung aller relevanten Dateien eines Softwareprojekts inklusive deren Historie (vgl. dazu den folgenden Praxistipp). Dazu werden diese Dateien an einem zentralen Ort, dem sogenannten ***Repository***, abgelegt. Der Vorteil davon ist, dass sich hier nicht nur die aktuelle Version einer Datei befindet, sondern deren gespeicherte Historie zur Verfügung steht. Im Repository sind also frühere Stände enthalten, einsehbar und bei Bedarf wiederherstellbar. Das ist insofern von Bedeutung, weil man so

auch einmal Experimente vornehmen und nötigenfalls zu einem älteren, funktionierenden Stand zurückkehren kann.

Darüber hinaus lassen sich verschiedene, voneinander unabhängige Entwicklungslinien, sogenannte ***Branches***, verwalten. Diese unterstützen z. B. Umbaumaßnahmen, spezielle Features oder Experimente. Die Hauptentwicklungslinie wird ***Stamm***, ***Hauptast*** oder auch ***master*** bzw. ***default*** genannt. Der aktuellste Stand einer Datei auf einer Entwicklungslinie heißt ***HEAD***. Änderungen und Erweiterungen, die auf Branches entwickelt wurden, können über einen ***Merge***-Vorgang in den aktuellen Stand integriert werden. Diese Abläufe deutet Abbildung 2-2 an.

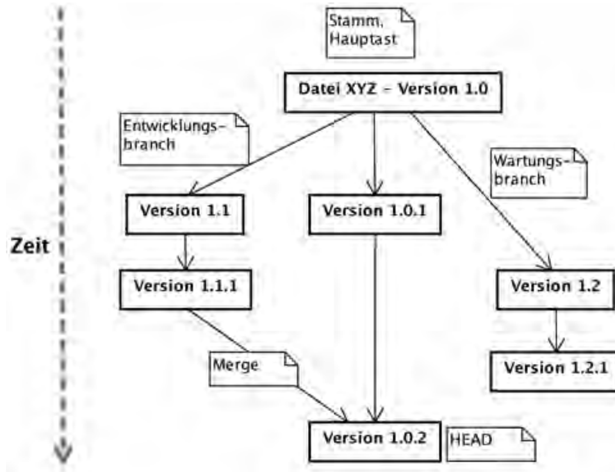

Abbildung 2-2 *Dateiversionsbaum*

Hilfreich ist, dass sich bei Änderungen an Dateien die Unterschiede zu beliebigen anderen Versionen ermitteln lassen. Außerdem können Versionen bei der Speicherung im Repository mit Kommentaren versehen werden, was den Überblick über Änderungen in den jeweiligen Versionen erleichtert.

Tipp: Inhalt des Repository

Anhand der in Abschnitt 2.2 beschriebenen Projektstruktur könnte man auf die Idee kommen, einfach alle Dateien eines Projekts in das Repository aufzunehmen. Dies ist aber nur begrenzt sinnvoll. Abgesehen von wenigen Ausnahmen sollten all diejenigen Dateien, die während des Build-Prozesses entstehen, *nicht* gesichert werden, eben weil sie sich immer wieder problemlos erneut erzeugen lassen. Dazu gehören vor allem die generierten `.class`-Dateien oder Berichte bzw. Dokumentationen. Ausnahmen können spezielle Versionen davon sein, die man historisieren möchte.

Was sollte also gespeichert werden? Die Frage ist einfach zu beantworten: Alle Dateien, die sich nicht automatisch generieren lassen. Dazu gehören der Sourcecode, die Unit Tests, verschiedene Dokumentationen (UML-Diagramme, Textdokumente usw.), Konfigurationsdateien und benötigte Build-Skripte.

Motivation für Versionsverwaltungen

Jede Bank verwendet einen Tresor, um wertvolle Dinge zuverlässig zu schützen. Softwareentwicklung ohne Sicherung ist ähnlich unsicher wie eine Geldanlage in Omas Sparstrumpf. Also sichern Sie Ihren größten Schatz – Ihren Sourcecode – in einem virtuellen Safe! Falls Sie bis jetzt ohne Versionsverwaltung leben, so sollten Sie sich durch die folgenden zwei Situationsbeschreibungen davon überzeugen lassen, schnellstmöglich ein solches System zu nutzen. Ansonsten könnte jeder kleine Fehler fatale Folgen haben, was ich mit zwei Anekdoten verdeutlichen möchte:

- Ich erinnere mich noch gut an die Tage mit Sicherungsdisketten. Es war eine mühselige und fehlerträchtige Arbeit, die aktuellen Daten auf die entsprechende Diskette zu überspielen. Als ich einmal die Arbeit eines Tages sichern wollte, habe ich beim Kopieren Quelle und Ziel vertauscht und damit hatte ich den gesamten Tag umsonst gearbeitet. Außerdem hatte ich danach zwei ältere Softwarestände, von denen ich nicht wusste, welche Änderungen diese enthielten. Um solche Situationen zu vermeiden, sollte häufig ins Repository gesichert werden. Somit bleibt selbst bei Problemen der maximale Datenverlust, d. h. die nicht gesicherten Änderungen, immer überschaubar.
  Diese Anekdote adressiert einen anderen wichtigen Aspekt: ***Datensicherung***. Ein Repository ist zwar ein guter Platz, um Daten zentral abzulegen und gezielt wiederherstellen zu können. Man ist so aber lediglich gegen die »eigene Dummheit« beim Ändern des Sourcecodes geschützt. Für eine weiter gehende Sicherheit der Daten ist es jedoch elementar wichtig, regelmäßig ein Backup des Repository zu erstellen, um das Repository an sich zu sichern, etwa gegen Datenträgerfehler.
- Stellen Sie sich vor, es ist Freitagmorgen und Ihr Chef betritt aufgeregt den Raum und verlangt für den späten Nachmittag ein Release, basierend auf dem letzten Auslieferungsstand des Programms von vor vier Wochen mit genau zwei dedizierten Bugfixes. Ohne Versionsverwaltungssystem haben Sie jetzt ganz schlechte Karten, denn in der Zwischenzeit hat die gesamte Entwicklertruppe kräftig weitergearbeitet. Dies alles ohne Fehler wieder rückgängig zu machen ist nahezu unmöglich. Eine Sicherung des Sourcecodes gibt es zwar irgendwo, aber in der Hektik ist diese momentan nicht auffindbar oder sie ist auf der mobilen Festplatte des Kollegen, der natürlich gerade gestern krank geworden ist.

Die Moral von der Geschichte ist: Lassen Sie den Computer die Arbeit erledigen, die er viel besser kann als Sie: das Verwalten vieler Versionen.

Wie schon eingangs erwähnt, spricht neben den genannten Vorteilen noch ein weiteres wesentliches Argument für den Einsatz einer Versionsverwaltung: Man kann mithilfe von Branches sehr leicht experimentelle Versionen der Software entwickeln, ohne dabei Angst haben zu müssen, Auslieferungen oder die gesamte Weiterentwicklung zu stören. Verläuft ein solches Experiment erfolgreich, so kann man die Änderungen übernehmen. Ansonsten verwirft man sie einfach.

Varianten der Versionsverwaltung

Vor dem Aufkommen von dedizierten Programmen zur Versionsverwaltung wurden häufig die Dateien eines Projekts lediglich in ein anderes Verzeichnis kopiert – zweckmäßigerweise in ein Verzeichnis mit einem Zeitstempel im Namen. Diese händische Versionsverwaltung scheint ganz natürlich und ist auch recht einfach, aber doch ziemlich fehleranfällig. Darüber hinaus besteht ein entscheidender Nachteil: Wenn man herausfinden möchte, was sich zwischen Versionen geändert hat, so ist dies oftmals ziemlich mühselig festzustellen. Erschwerend kommt hinzu, dass mitunter versehentlich die zur Versionierung eigentlich notwendigen Kopien ausbleiben und somit gewisse Zwischenstände nicht gesichert sind. Dann besitzt man nur eine unvollständige Historie.

Um diesen Nachteilen zu begegnen und Änderungen an den Dateien eines Projekts feingranular speichern und später nachvollziehen zu können, wurden Versionsverwaltungssysteme erfunden. Man unterscheidet zwischen ***zentralen*** und ***dezentralen Versionsverwaltungen***. Schon vor einigen Jahrzehnten entstanden zentrale Versionsverwaltungen (Version Control System, VCS). Dort erfolgt die Versionsverwaltung auf einem zentralen Server und erfordert von den Nutzern (Clients) somit (Netzwerk-)Zugriff darauf. Dezentrale Versionsverwaltungen (Distributed Version Control System, DVCS) kombinieren die Vorteile der beiden gerade beschriebenen Varianten: Sie sind nahezu so einfach in der Handhabung wie des Erstellen lokaler Kopien und bieten darüber hinaus leistungsstarke Versionsverwaltungsfunktionalität.

2.3.1 Arbeiten mit zentralen Versionsverwaltungen

Während die lokale Versionsverwaltung durch Kopieren per Hand selbst schon für eine Person recht schnell an Grenzen stößt, so gilt dies umso mehr bei der Zusammenarbeit und Verwaltung von Änderungen im Team, weil hier deutlich mehr (eventuell auch konkurrierende) Änderungen und damit auch Abstimmungsbedarf entstehen. Als Abhilfe wurden zentrale Versionsverwaltungen entwickelt. Bekannte zentrale Versionsverwaltungen sind das Concurrent Versions System (CVS) und Subversion (SVN). CVS ist älter und besitzt einige Einschränkungen, etwa bei Namensänderungen. SVN wurde als Nachfolger neu entwickelt und unterstützt diverse Dinge besser als CVS.

Repository

Wie bereits erwähnt, werden die Dateien eines Projekts mit ihrer gesamten Historie an einer zentralen Stelle, dem ***Repository***, gespeichert. Gewöhnlich liegt dieses Repository auf einem dedizierten Server und dort wird auch der Serveranteil zur Versionsverwaltung ausgeführt. Zum Zugriff darauf benötigt man als Nutzer eine spezielle Clientsoftware, mit der alle Aktionen zur Versionsverwaltung erfolgen, z. B. das Hinzufügen, Ändern oder Aufbereiten der Versionshistorie. Diese Aktionen erfordern jeweils Zugriffe auf das Repository. Der dezentrale Client-Server-Ansatz erlaubt es, an verschiedenen Orten verteilt an einem Projekt zu arbeiten, setzt aber Netzwerkzugriff voraus.

Arbeitsablauf

Zum Bearbeiten eines Versionsstands überträgt ein Entwickler den gewünschten Stand aller Dateien eines Projekts aus dem Repository in ein Arbeitsverzeichnis auf seinem Rechner. Diesen Vorgang bezeichnet man als »***Auschecken***« (Check-out). Anschließend arbeitet man auf lokalen Kopien der Dateien, der sogenannten ***Working Copy***, und nimmt dort Änderungen vor. Sind diese abgeschlossen oder haben diese einen stabilen Zwischenstand erreicht, so sollte eine Integration in das Repository erfolgen. Dadurch werden die Änderungen allen anderen Kollegen zugänglich gemacht. Dieser Vorgang wird »***Einchecken***« (Check-in) oder auch ***Commit*** genannt. Damit die neu eingespielten Änderungen allerdings für die Kollegen in ihrem Arbeitsbereich tatsächlich sichtbar werden, müssen diese zunächst einen Abgleich mit dem Repository durchführen. Diesen Abgleich nennt man ***Update*** oder ***Synchronize***. Die beschriebenen Arbeitsabläufe visualisiert Abbildung 2-3.

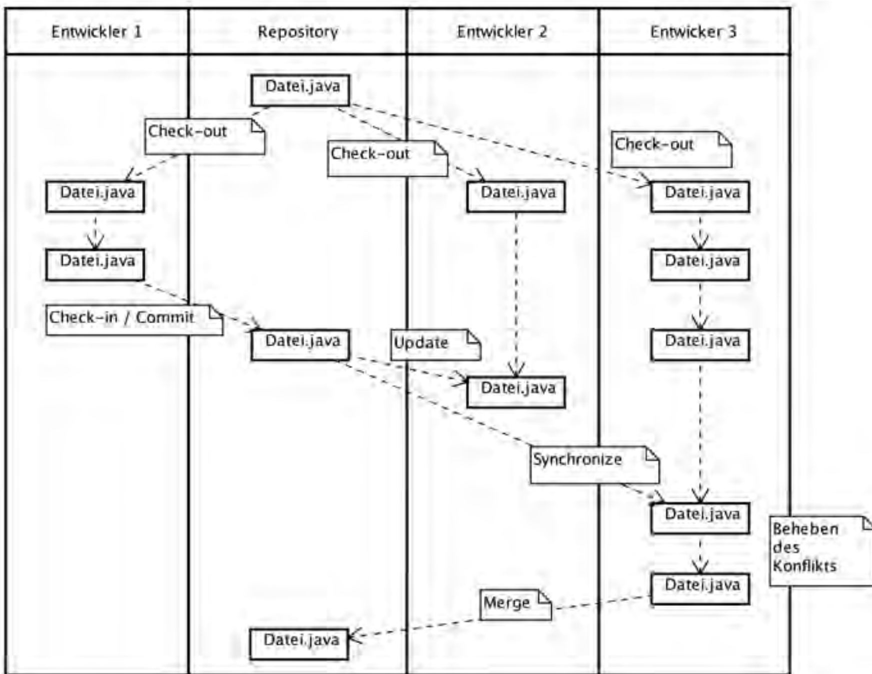

Abbildung 2-3 *Arbeitsablauf mit zentralen Versionsverwaltungen*

Besonderheiten beim Abgleich Da die eigenen Änderungen nur auf lokalen Kopien der Originale erfolgen, kann es zu Problemen beim Abgleich kommen, wenn mehrere Entwickler die gleiche Datei verändert haben. In CVS werden bei einem ***Update*** die lokalen Änderungen mit der aktuellen Version aus dem Repository überschrieben. Ein ***Synchronize*** versucht die Änderungen der lokalen Version mit denen aus dem Repository abzugleichen. Dies funktioniert gut, wenn unterschiedliche Teile einer Datei von Änderungen betroffen sind. SVN integriert Änderungen bei einem Update automatisch, wenn aufgrund der Differenzanalyse keine Probleme festgestellt wurden. Widersprechen sich Änderungen an der lokalen Datei und deren Version im Repository, so muss man steuernd eingreifen. Eine solche Situation wird ***Konflikt*** genannt. Eine betroffene Datei lässt sich erst ins Repository integrieren, wenn alle dort gefundenen Konflikte behoben sind. Die Zusammenführung verschiedener Änderungen wird als ***Merge*** bezeichnet.

Tagging und Branching

Gehen wir noch einmal zu dem Beispiel zurück, in dem eine Auslieferung des Systems, basierend auf dem letzten Release sowie ein paar dringend benötigten Fehlerbehebungen, gewünscht wird. Sie brauchen nun Zugriff auf den Stand des Projekts genau so, wie es ausgeliefert wurde. Beispielsweise lassen sich Dateien anhand ihrer Versionsnummer zurückholen. Dieser Vorgang ist für CVS aufwendig, da die Versionsnummer von Datei zu Datei variiert.[2] SVN bietet mehr Komfort, da die Versionsnummer für alle Dateien des Repository einheitlich behandelt wird (siehe folgenden Praxistipp).

Alternativ können Dateien anhand eines speziellen Datums zurückgeholt werden. Dies ist nützlich, wenn wenige Dateien betroffen sind oder nur der zeitliche Verlauf von Interesse ist. Normalerweise soll ein Projekt jedoch wieder in einen Zustand gebracht werden, in dem es zu einem bestimmten Zeitpunkt oder Ereignis war, beispielsweise einer Auslieferung oder des Abschlusses größerer Änderungen. Das führt uns zum sogenannten Tagging, was wir uns nun anschauen.

Tipp: Versionsnummern in SVN

Die Versionsnummer einer Datei und von Verzeichnissen entspricht immer der Versionsnummer des Projekts zum Zeitpunkt der Änderung. Für Verzeichnisse ist dies die höchste Versionsnummer der enthaltenen Dateien und Verzeichnisse. Dadurch kann die Folge der Versionsnummern einzelner Dateien oder Verzeichnisse auch Lücken haben. Beim Auschecken wird jeweils die größte Versionsnummer aus dem Repository geholt, die kleiner oder gleich der angeforderten Versionsnummer beim Auschecken ist.

[2] Es müssen dann die gesamten Dateien eines Projekts überprüft werden, um herauszufinden, welche Version jede Datei zum Zeitpunkt des gewünschten Stands hatte. Anschließend muss dann jede Datei einzeln anhand der ermittelten Versionsnummer wiederhergestellt werden.

Tagging Wenn man für Änderungen einen ganz speziellen Stand eines Projekts bearbeiten möchte, so wäre es umständlich und fehleranfällig, eine Menge von Dateien über deren Versionsnummer oder ein Datum zusammensuchen zu müssen. Als Abhilfe kann man zu einem beliebigen Zeitpunkt eine sogenannte ***Markierung*** (engl. ***Tag***) setzen. Vor größeren Änderungen und Auslieferungen sollte man dies in jedem Fall tun. Mithilfe von Markierungen lässt sich später ein Zwischenstand exakt wiederherstellen, um basierend darauf Fehlerkorrekturen durchführen zu können.

Man kann sich Markierungen wie eine Verbindungsschnur bzw. einen Zusammenschluss zwischen Versionen verschiedener Dateien vorstellen (vgl. Abbildung 2-4).

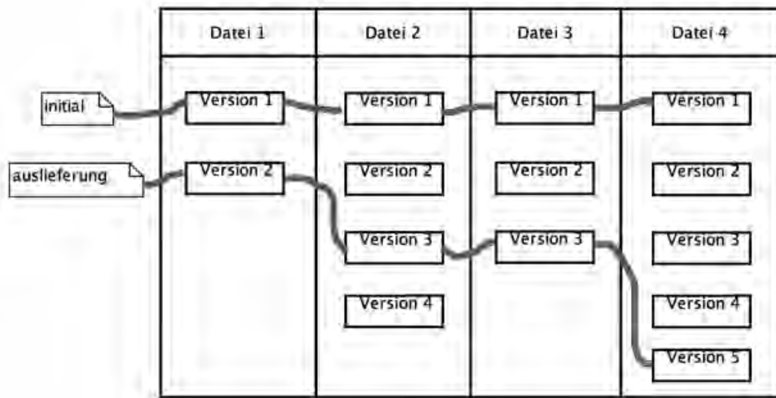

Abbildung 2-4 *Zwei Tags (miteinander verbundene Versionen von Dateien)*

Branching Unter einer Verzweigung oder einem ***Branch*** versteht man eine eigene Entwicklungslinie. Ein solcher Abzweig lässt sich von jeder beliebigen anderen Entwicklungslinie abspalten, oft geschieht dies aber vom Stamm, auch ***Main-Branch*** oder Trunk genannt. In einer solchen Entwicklungslinie entsteht eine eigene Historie, sodass parallel zu anderen Zweigen weitergearbeitet werden kann: Änderungen in einem Zweig haben keinen Einfluss auf andere Verzweigungen. Dadurch sind Branches hilfreich, um potenziell gefährliche oder umfangreiche Änderungen zu isolieren, bis sich diese stabilisiert haben.

Zur Motivation blicken wir auch hier wieder auf das vorherige Beispiel zurück. Der ältere Sourcecode-Stand konnte aufgrund einer gesetzten Markierung leicht wiederhergestellt werden. Auf dieser lokalen Kopie können Fehlerbehebungen erfolgen. Natürlich wäre es wünschenswert, die durchgeführten Änderungen im Repository zu sichern, damit man später auf diesem speziellen Stand weiterarbeiten kann. Dies ist aber nicht sinnvoll möglich, wenn kein Branch erzeugt wurde, weil man ansonsten neuere Versionen des Hauptasts mit einem älteren Stand überspielen würde. Das ist jedoch in der Regel nicht gewünscht. Als Abhilfe dienen Branches, die eine eigene, separate Historie bieten. Dadurch lassen sich Änderungen wieder sinnvoll ins Repository reintegrieren.

Tipps für die Arbeit mit Branches

Beim Erstellen von Branches sollte man etwas Vorsicht walten lassen. Unbedacht eingesetzt können zu viele Branches ein Projekt ins Chaos stürzen, da man schnell den Überblick verliert, was in welchem Branch erweitert oder korrigiert wurde. Als Folge divergiert die Sourcecode-Basis, sodass es mit der Zeit immer schwieriger wird, die Änderungen auf den Branches wieder korrekt zusammenzuführen.

Mit CVS bzw. SVN lassen sich Verbesserungen und Fehlerbehebungen eines Branches oftmals nur mit Mühe auf andere Branches übertragen. Allerdings ist nicht unbedingt die absolute Anzahl an Branches klein zu halten, sondern lediglich die Anzahl »aktiver« Branches, d. h., an denen gleichzeitig gearbeitet wird. Darüber hinaus sollten die Verzweigungstiefe und Komplexität der Branches überschaubar sein. Nur in wenigen Fällen ist es sinnvoll, von einer Verzweigung wiederum Verzweigungen zu erzeugen. Ein möglicher Grund dafür sind Wartungsaufgaben auf älteren Ästen.

Hält man die Anzahl parallel laufender Entwicklungen gering, so sinkt die Wahrscheinlichkeit für Konflikte beim Integrieren. Weniger Konflikte erreicht man auch dadurch, ***dass man Branches möglichst häufig (z. B. täglich oder sobald die Arbeiten einen stabilen Zwischenstand erreicht haben) mit dem Ursprungsbranch synchronisiert***, also die dortigen Änderungen in den eigenen Branch reintegriert.

Tipp: Namensgebung für Markierungen und Verzweigungen

Die Namen von Branches und Markierungen sollten einem einheitlichen Schema folgen und deren Bedeutung sollte offensichtlich sein. Der Name sollte alle wichtigen Informationen über den Softwarestand enthalten und einer Konvention, etwa `ZWECK_KUNDE_VERSION_DATUM`, folgen. Versionsnummern für Releases sollten die Versionsangaben `Major.Minor.Patchlevel` enthalten. Damit ergibt sich als Beispiel folgender Name: `auslieferung_Meyer_V2.7.13_20080612`. Für den Zweck sind zumindest folgende Kürzel sinnvoll:

- `entwicklung` für Branches für Entwicklungen
- `auslieferung` für Auslieferungen an Kunden
- `release` für funktionsfähige Zwischenstände der Software

Bitte beachten Sie, dass die kritischen Aussagen zum Branching insbesondere für VCS mit zentralem Repository gelten und in wesentlich geringerem Maße für DVCS, da diese speziell dafür ausgelegt sind, mit vielen Branches und Versionsständen umgehen zu können.

2.3.2 Dezentrale Versionsverwaltungen

Während die zuvor vorgestellten Systeme CVS und SVN beide eine zentrale Versionsverwaltung repräsentieren, sind heutzutage dezentrale Versionsverwaltungen, etwa Git[3] und Mercurial[4], immer beliebter. Git ist derzeit wohl die populärste dezentrale Versionsverwaltung und wurde von Linus Torvalds kurzerhand selbst geschrieben, da keine der herkömmlichen Versionsverwaltungen ausreichend Flexibilität und Funktionalität für die verteilte Entwicklung des Linux-Kernels bot. Insbesondere fehlte es an gutem Support für viele unabhängige Branches und das offline Arbeiten sowie für das Mergen oder Rücknehmen von verschiedenen Änderungen. Ein zum Einstieg lehrreiches interaktives Tutorial zu Git steht unter `https://try.github.io/` bereit. Dort lernt man das Anlegen eines Repository und das Arbeiten damit in einem geführten Ablauf.

Mercurial ist eine valide und von mir bevorzugte Alternative zu Git. Wieso? Für mich lässt sich Mercurial teilweise einfacher handhaben, wie es nachfolgend im Meinungskasten dargestellt wird.

Meinung: Git vs. Mercurial

Im Prinzip sind Git und Mercurial funktional in etwa gleich mächtig: Bei der Integration in IDEs gibt es aber Unterschiede: Oft wird Git direkt unterstützt und Mercurial mitunter nur durch Plugins. Auf der Kommandozeile hinkt Git aber meiner Meinung nach in der Benutzbarkeit etwas hinterher, weil hier zu viele Implementierungsdetails durchscheinen. Unter Windows empfinde ich auch die grafische Integration von Mercurial (TortoiseHG[a]) in den Explorer ein wenig übersichtlicher als die von TortoiseGit[b]. Das zeigt sich in Kleinigkeiten: Commit-Kommentare werden standardmäßig bei Git über den vi eingegeben. Mercurial öffnet dagegen den vom System voreingestellten Texteditor. Entscheidend ist für mich aber die Art und Weise der Verwaltung und Angabe von Revisionsnummern. In Mercurial sind für den Benutzer Revisionsnummern natürliche Zahlen und es werden nicht die internen hexadezimalen Keys wie bei Git nach außen exponiert. Ich referenziere einfach lieber Revision 4711 als 00eda2a680893a20ea11d48ddd884812dc97a718. Basierend auf den Ausführungen wird klar, dass ich persönlich Mercurial bevorzuge, da es sich für mich natürlicher in der Handhabung anfühlt. Ich empfehle Ihnen, einfach mal beide Versionsverwaltungen zu installieren und damit ein wenig herumzuspielen, um die eigene Präferenz zu finden.

[a] *https://tortoisehg.bitbucket.io/*
[b] *https://tortoisegit.org/*

[3] *http://git-scm.com/downloads*
[4] *https://www.mercurial-scm.org/*

Repositories

Für die Arbeitsweise mit CVS und SVN haben wir nach der Lektüre der vorangegangenen Abschnitte ein erstes Verständnis aufgebaut. Dezentrale Versionsverwaltungen arbeiten gar nicht so verschieden dazu. Insbesondere fügen sie jedoch optional eine weitere Hierarchieebene ein. Schauen wir uns das im Folgenden genauer an.

Während bei zentralen Versionsverwaltungen jeweils Zwischenstände (Arbeitskopien) auf die Rechner der jeweiligen Entwickler übertragen werden, wird bei dezentralen Versionsverwaltungen lokal auf dem Rechner des Entwicklers im Arbeitsverzeichnis das gesamte Repository gespeichert.[5] Innerhalb des ***Arbeitsverzeichnisses*** existiert demnach ein vollständiger Stand des Projekts mitsamt seiner Historie. Mit diesem ***lokalen Repository*** kann man nahezu wie mit CVS oder SVN gewohnt arbeiten, allerdings mit dem großen Vorteil, dass alle Aktionen nicht über das Netzwerk, sondern lokal im Dateisystem und damit extrem performant ausgeführt werden können. ***Git und Mercurial ermöglichen also das Arbeiten mit lokalen Repositories unabhängig von einem zentralen Repository und Netzwerkzugang***.[6] Zunächst einmal erfolgt die Versionskontrolle dezentral und jedes Repository existiert für sich eigenständig.

Es besteht aber die Möglichkeit, ein Repository im Netz zu veröffentlichen und so auch mit anderen teilen zu können. Dadurch gibt es bei den Vertretern von DVCS im Unterschied zu den zentralen Versionsverwaltungen konzeptionell noch die Ebene über dem lokalen Repository, nämlich alle möglichen Repositories im Netz (***Remote Repository***). Für die Zusammenarbeit an einem größeren Projekt empfiehlt es sich, neben vielen verteilten Repositories auch ein ausgezeichnetes Projekt-Repository auf einem zentralen Server zu nutzen, ähnlich zu dem zentralen Server bei CVS/SVN. Dadurch gestaltet sich der Umstieg von CVS/SVN leichter, da man sich die lokalen Repositories als praktische lokale Zwischenspeicher vorstellen kann: Ein Feature, das man sich häufig für CVS/SVN gewünscht hat – vor allem dann, wenn die Zugriffe auf das zentrale Repository langsam oder aufgrund von Netzwerkproblemen nicht bzw. nur eingeschränkt möglich waren.

Tipp: Sicherungskopien

Weil jeder Benutzer ein vollständiges Repository auf seinem Rechner besitzt, ist es ganz einfach durch Kopie des Arbeitsverzeichnisses möglich, ein Repository neu zu erzeugen. Somit können auf einfache Art und Weise Sicherungskopien erstellt werden. Selbst bei Datenträgerdefekten auf einem Rechner kann man somit einen (nahezu) vollständigen Stand einfach durch Kopie eines anderen Repository wiederherstellen.

[5] Mercurial und Git legen dazu versteckte Ordner und Dateien an.

[6] Insbesondere muss kein zentraler Server mit Repository existieren – für die Arbeit in (verteilten) Teams ist dies aber hilfreich.

Begriffe und Arbeitsablauf im Überblick

Die Ideen und Abläufe sind bei der Arbeit mit einem lokalen Repository recht ähnlich zu der von zentralen VCS gewohnten Arbeitsweise. In der Abbildung sehen wir einige Arbeitsschritte, die einen ersten Eindruck von der Arbeit mit einem DVCS vermitteln.

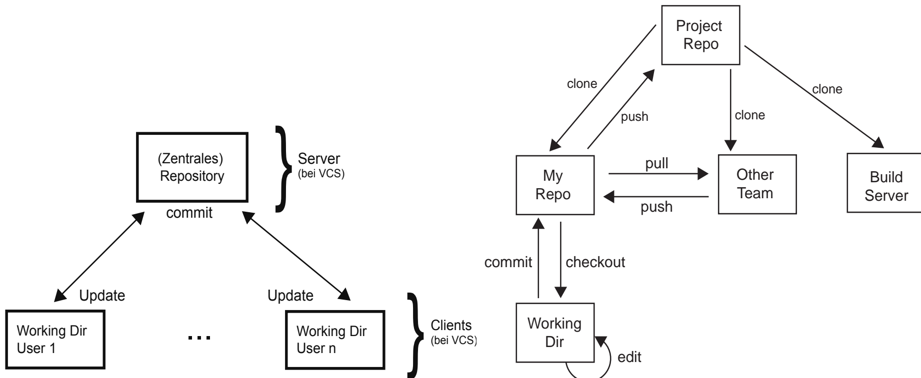

Abbildung 2-5 *Repository-Workflow für SVN (links) und DVCS (rechts)*

Für CVS und SVN haben wir bereits den typischen Arbeitsablauf und die entsprechenden Begriffe wie Check-out und Commit kennengelernt. Nachfolgend betrachten wir ein paar typische und grundlegende Arbeitsschritte beim Einsatz von DVCS, die aufgrund ihrer abweichenden Arbeitsweise auch eine leicht andere Begriffswelt besitzen.

Die folgenden Ausführungen zeigen exemplarisch immer die Befehle sowohl für Mercurial als auch für Git. Das Kommando für Mercurial ist `hg`[7] und für Git `git`.

Repository erstellen Sollen die Dateien in einem Verzeichnis versioniert werden, so kann man sehr einfach dort ein neues Repository erstellen: Man wechselt in das gewünschte Verzeichnis und gibt dafür Folgendes ein:

```
git init     // für Git
// ---------------------------------------------------------------
hg init      // für Mercurial
```

Das ist alles, um ein neues Repository anzulegen. Nun können Sie die zu versionierenden Dateien dem Repository hinzufügen. Bevor ich darauf eingehe, beschreibe ich den Fall, dass das Repository als Kopie eines anderen Repository erstellt wird.

[7] Mercurial bedeutet Quecksilber und hg ist das chemische Kürzel dafür.

Repository clonen Zum Erstellen einer Kopie eines Repository nutzt man den `clone`-Befehl, der folgende Syntax besitzt – als Abkürzung verwende ich nachfolgend mitunter `git/hg`, was ausdrückt, dass die Syntax identisch ist:

```
git/hg clone /pfad/zum/repository ziel               // lokales Repository
git/hg clone benutzername@host:/pfad/zum/repository  // Remote Repository
```

Nachfolgend zeige ich ein Beispiel, wie man auf lokale Repositories zugreifen kann, und deute für Mercurial den Zugriff auf ein Remote Repository unter Nutzung von HTTP, hier vereinfachend des eigenen Rechners (`localhost`), an:

```
git clone file:///C:/Users/Micha/Desktop/DVCS DVCS-GIT
// -------------------------------------------------------------
hg clone file:///C:/Users/Micha/Desktop/DVCS DVCS-CLONE
hg clone http:/localhost:8000/DVCS DVCS-CLONE-BY-HTTP
```

Das erstmalige Initialisieren eines lokalen Repository kann durch eine Kopie eines anderen lokalen oder Remote Repository erfolgen. Dabei wird eine vollständige Kopie erzeugt, d. h. inklusive sämtlicher Branches und der gesamten Versionshistorie. Zudem erfolgt ein Check-out des Hauptasts, den man für Git ***master*** bzw. für Mercurial ***default*** nennt, statt ***trunk*** bei SVN.

Check-out Nach dem Klonen eines Repository befindet man sich zunächst auf dem *master* bzw. *default* und dem aktuellen Stand (HEAD). Um andere Versionsstände (eventuell aus anderen Branches) bearbeiten zu können, dazu dient ein `checkout`, der den gewünschten Versionsstand einer speziellen Revision oder eines Branches in das Arbeitsverzeichnis überträgt. Wollten wir die ältere Revision 123 auschecken, so schreibt man Folgendes, wobei für `git` ein frei erfundener Hex-Key dargestellt ist:

```
git checkout 22a16bb5cdd8a8d72bda4276b720e0a86a2bad72
// -------------------------------------------------------------
hg checkout -r 123
```

Um wieder auf die aktuellste Revision HEAD zurückzuwechseln, schreibt man einfach:

```
git checkout master
// -------------------------------------------------------------
hg checkout default
```

Änderungen und Staging Area Wenn Sie Änderungen an den lokalen Dateien vornehmen, werden diese nicht direkt ins lokale Repository übertragen. Git und Mercurial bieten eine weitere Zwischenebene: die sogenannte ***Staging Area***. Alle veränderten Dateien, die (später) ins Repository committet werden sollen, müssen zunächst der Staging Area hinzugefügt werden. Dazu dient der `add`-Befehl:

```
git/hg add <dateiname>
git/hg add *
```

Dies kann man beliebig für verschiedene Dateien wiederholen oder zur Vereinfachung die Wildcard `*` nutzen. Falls man eine oder mehrere Dateien einmal versehentlich der Staging Area hinzugefügt hat, so kann man diese bei Bedarf wieder daraus entfernen. Meistens wird man sich zunächst einen Überblick verschaffen wollen. Dazu dient der `status`-Befehl:[8]

```
git/hg status
```

Bei der Ausgabe symbolisiert ein `?`, dass die Datei nicht unter Versionskontrolle steht. Das `A` (Added) bedeutet, dass die Datei in der Staging Area hinzugefügt ist:

```
A ToBeRemovedFromStaging.txt
? ToBeAdded.txt
```

Anhand der Konsolenausgabe kann man diejenigen Dateien ermitteln, die man wieder entfernen möchte, etwa die Datei `ToBeRemovedFromStaging.txt`. Der dazu benötigte `remove`-Befehl besitzt folgende Syntax:

```
git/hg remove <dateiname>
git/hg remove *
```

Änderungen ins lokale Repository übertragen Durch Aufruf des `commit`-Befehls werden Änderungen aus der Staging Area ins lokale Repository übertragen:

```
git/hg commit -m "Commit-Nachricht"
```

Der Parameter `-m` erlaubt es, beim Commit eine (möglichst) aussagekräftige Nachricht zu übergeben. Die Änderungen sind dann im lokalen Repository historisiert. Bei der Arbeit im Team wird man diese jedoch immer mal wieder mit dem Projekt-Repository abgleichen wollen. Dazu dient der später beschriebene `push`-Befehl.

Branching Bekanntermaßen lassen sich durch Einsatz von Branches unabhängige Entwicklungen separat und isoliert voneinander mit unterschiedlichen Historien entwickeln. Wenn ein Repository neu erstellt wird, so existiert dort standardmäßig immer der Branch *master* (Git) bzw. *default* (Mercurial). Um Merge-Probleme möglichst zu vermeiden, war es bei CVS/SVN nicht ungewöhnlich, auf dem dazu korrespondierenden *trunk* zu entwickeln. Ein analoges Vorgehen ist für Git und Mercurial zwar auch möglich, aufgrund ihrer deutlich besseren Unterstützung von Branching und Merging bietet sich aber ein anderes Vorgehen an: Entwicklungen sollten auf eigenständigen Branches durchgeführt werden, wobei hier – wie schon für CVS/SVN erwähnt – eine

[8] Bitte beachten Sie, dass dieser Befehl auch alle Unterverzeichnisse durchsucht, wodurch die Liste recht umfangreich werden kann. Möchte man die Treffermenge etwa auf PDF-Dateien begrenzen, so kann man Folgendes eingeben: `git/hg status *.pdf`. Es gibt weitere Möglichkeiten zur Einschränkung. Ergänzende Hinweise gibt `git/hg status --help`.

Beschränkung auf eine überschaubare Anzahl an Branches aus purem Selbstschutz und zur besseren Übersicht eingehalten werden sollte.

Möchte man eine Funktionalität auf einem Branch entwickeln, so führt man Folgendes zu dessen Erstellung aus:

```
git branch mybranch        // Branch erstellen
git checkout mybranch      // Stand mybranch in Working Copy holen
git checkout -b mybranch   // Kurzform
// ------------------------------------------------------------
hg branch mybranch
```

Nun kann man Änderungen an den Dateien der Arbeitskopie vornehmen, die den Branch repräsentiert.

Merge Irgendwann sind die Arbeiten an dem besonderen Feature auf dem Branch *mybranch* erfolgreich fertiggestellt und sollen wieder mit dem *master* bzw. *default* abgeglichen werden. Gerade im Zusammenführen von Änderungen und Branches liegt eine Stärke von DVCS.

Um einen Merge-Vorgang zu starten, muss man sich in dem Branch befinden, in den die Änderungen eingespielt werden sollen. Da wir bislang auf dem Branch `mybranch` gearbeitet haben, müssen wir mit dem Arbeitsverzeichnis zunächst wieder auf den *master* bzw. *default* wechseln. Dazu existieren folgende Kommandos:

```
git checkout master
// ------------------------------------------------------------
hg checkout default
```

Um dann einen Abgleich zwischen *mybranch* und *master* bzw. *default* vorzunehmen, gibt man folgendes Kommando ein:

```
git merge mybranch   // Merge vom Branch mybranch
// ------------------------------------------------------------
hg merge mybranch    // Merge vom Branch mybranch
hg commit -m "Merged Branch mybranch"
```

Im Vergleich zu zentralen Versionsverwaltungen benötigen obige Aktionen wenig Zeit, da keine aufwendigen Zugriffe über das Netzwerk erfolgen müssen. Darüber hinaus kann in der Regel deutlich mehr an konkurrierenden Änderungen sinnvoll miteinander kombiniert werden, ohne dass dies Eingriffe des Entwicklers erfordert. Beim Mergen können natürlich dennoch ab und zu Konflikte auftreten, die manuell gelöst werden müssen.

Push und Pull Nachdem nun die Arbeiten abgeschlossen sind oder einen sinnvollen Zwischenstand erreicht haben, können und sollten diese mit dem Projekt-Repository abgeglichen werden.[9] Dazu dient der `push`-Befehl. Diesen kann man jedoch nicht ausführen, wenn bereits Änderungen vorliegen.

Das ist bei der Arbeit in Teams aber eher die Regel: Andere Teammitglieder haben oftmals ebenfalls bereits Änderungen durchgeführt und diese schon an das Projekt-Repository übertragen. Um das eigene, lokale Repository mit demjenigen auf dem Projektserver abzugleichen, dient der `pull`-Befehl.

Wie bei einem Update in SVN können bei einem `pull` mehrere Änderungen innerhalb einer Datei existieren und zu behandeln sein. Oft lassen sich diese automatisch mergen. Bei Widersprüchen müssen diese allerdings von Hand aufgelöst werden. Danach ist dann schließlich ein `push` möglich und erlaubt die Veröffentlichung der eigenen Änderungen.

2.3.3 VCS und DVCS im Vergleich

In diesem Abschnitt stelle ich nochmals zusammenfassend die Eigenschaften zentraler Versionsverwaltungen, insbesondere deren Nachteile, und die Vorteile dezentraler Versionsverwaltungen gegenüber und schließe mit einem Fazit.

Nachteile zentraler Versionsverwaltungen

Zentrale Versionsverwaltungen erleichtern zwar unbestritten die Verwaltung der Historie von Dateien enorm, jedoch besitzen sie auch ganz entscheidende Nachteile:

- **Single Point of Failure** – Durch ihre Ausrichtung auf das zentrale Repository besitzen zentrale Versionsverwaltungen einen ***Single Point of Failure***: Sobald das Repository einmal nicht mehr per Netz erreichbar ist, können nahezu keine Interaktionen mehr ausgeführt werden, die im Zusammenhang mit der Versionsverwaltung stehen. Schnell werden dann ganze Teams einige Stunden ausgebremst.
- **Zeitpunkt des Commits** – Es ist nicht so leicht, den richtigen Zeitpunkt für einen Commit zu finden: Mitunter ist es sogar recht schwierig zu entscheiden, wann und vor allem auch welche Änderungen eingecheckt und welche Teile besser noch nicht im Repository veröffentlicht werden sollten.

Wahl des richtigen Zeitpunkts zum Commit bei VCS Intuitiv scheint klar, dass man mit einem Commit so lange warten sollte, bis die gemachten Änderungen einen sinnvollen Zwischenstand erreicht haben, und dass auch nur lauffähiger, möglichst gut getesteter Sourcecode ins Repository integriert werden sollte. Das Ganze hat

[9] Damit ein Abgleich mit Remote Repositories erfolgen kann, müssen diese entsprechend konfiguriert werden. Ich setze voraus, dass dies initial geschehen ist.

Rückgabewerte vs. Exceptions

Nach dieser Einführung in die Fehlerbehandlung mithilfe von Exceptions möchte ich auf einen weiteren wichtigen Aspekt dabei eingehen: Die Fehlerbehandlung sollte möglichst sauber vom »Nutzcode« trennbar sein. Meistens lässt sich dies durch die Verwendung von Exceptions einfacher als durch den Einsatz von Rückgabewerten erreichen. Letzteres verursacht recht schnell eine Vermischung von Anwendung und Fehlerbehandlung und führt oftmals zu einer schlechteren Trennung von Zuständigkeiten oder endet gar im Chaos. Exceptions bieten mit ihrer Verarbeitung in `catch`-Blöcken eine bessere Trennung. Allerdings kann es durch mehrere zu behandelnde Exceptions schnell zu vielen `catch`-Blöcken kommen.

Ein Vorteil von Exceptions ist, dass sich diese an Aufrufer propagieren lassen, wenn man meint, an anderer Stelle besser und angemessener auf Fehlersituationen reagieren zu können. Ein Nachteil ist, dass durch Exceptions ein Overhead zur Laufzeit für das Erzeugen (Aufbereitung des Stacktrace), Durchreichen usw. entsteht.[18] Für Rückgabewerte gilt das nicht – allerdings lassen sich diese auch nicht so gut weiter propagieren.

Fallstricke bei der Fehlerbehandlung

Erfahrenen Entwicklern ist bewusst, dass zum Teil mehr Mühe und Gehirnschmalz in die Behandlung möglicher Fehlersituationen einfließen muss als in die normale Programmlogik. Dies gilt vor allem dann, wenn man mit anderen Systemen, Komponenten, Ein- und Ausgabegeräten und im Besonderen mit dem Benutzer interagiert. Für unerfahrene Programmierer ist Fehlerbehandlung oftmals eines der Dinge, um die man sich eher ungern oder halbherzig kümmert, meistens gerade so weit, dass das Programm nicht abstürzt. Zwei Beispiele für misslungene Fehlerbehandlungen sind folgende:

- **»Anstands«-Null-Prüfungen** – Solche Prüfungen dienen lediglich zur Vermeidung von `NullPointerException`s:

```
public static void updateSystemState()
{
    final SystemState state = calculateSystemState();
    if (state != null)
    {
        systemStateMap.put(KEY_SYSTEM, state);
    }
}
```

  Im Falle eines `null`-Werts für die Variable `state` geschieht einfach nichts. Problematisch daran ist, dass auf diese Weise ein spezieller Zustand (`state == null`), der möglicherweise sogar ein Fehlerfall ist, kaschiert wird und zudem keine Behandlung oder Warnmeldung erfolgt. Abschnitt 16.3.7 diskutiert dies als BAD SMELL: SONDERBEHANDLUNG VON RANDFÄLLEN.

[18] Da meistens eine Ausnahmesituation vorliegt, ist der Mehraufwand oftmals irrelevant.

- **»Delayed Exception«** – Ein Beispiel für diese »Fehlerbehandlungsstrategie« sind öffentliche Methoden, die keine Konsistenzprüfung ihrer Parameter durchführen und so die Eingabe ungültiger Daten ermöglichen. Aufgrund der fehlenden Prüfung führen derartige Eingaben später potenziell zu fehlerhaften Berechnungen oder gar zu (unerwarteten) Exceptions. Im nachfolgenden Beispiel können `null`-Werte in einer `systemStateMap` gespeichert werden, obwohl diese keine gültige Eingabe darstellen. Die Konsistenzprüfung erfolgt erst bei Lesezugriffen (und damit zu spät) in der `getSystemState()`-Methode:

```
public static void setSystemState(final SystemState state)
{
    systemStateMap.put(KEY_SYSTEM, state);
}

public static SystemState getSystemState()
{
    final SystemState state = systemStateMap.get(KEY_SYSTEM);
    if (state == null) // Beispiel für schlechte Fehlerbehandlung
        throw new IllegalStateException("No entry for system state");

    return state;
}
```

  Das Problem fällt zunächst nicht auf. Erst wenn später ein Aufruf von `getSystemState()` erfolgt, wird die ungültige Eingabe sichtbar und löst eine Exception aus: Eine mögliche Fehleingabe bleibt dadurch unter Umständen sehr lange unerkannt und ein solcher Fehler wird viel zu spät erkannt.

5.6.2 Checked Exceptions und Unchecked Exceptions

In Java unterscheidet man zur Behandlung von Fehlersituationen zwischen Checked Exceptions und Unchecked Exceptions. Nachfolgende Abbildung 5-3 zeigt die Ableitungshierarchien der beiden Arten von Exceptions.

Checked Exceptions sind Bestandteil des »Vertrags« zwischen Aufrufer und Bereitsteller einer Methode und zeigen mögliche, durch Aufrufer zu erwartende Fehlersituationen an. Sie müssen daher in der Methodensignatur mit dem Schlüsselwort `throws` angegeben werden. Dies sichert ab, dass Aufrufer entweder selbst aktiv mit einem `catch`-Block darauf reagieren oder ansonsten automatisch eine Propagation an weitere aufrufende Methoden erfolgt.

Eine Behandlung ist für ***Unchecked Exceptions*** mit dem Basistyp `RuntimeException` nicht zwingend erforderlich – aber möglich, und man kann diese normal mit einem `catch`-Block bearbeiten. Weil aber Unchecked Exceptions normalerweise schwerwiegende Programmierprobleme oder unerwartete Situationen ausdrücken, ist das Verarbeiten mit einem `catch`-Block eher ungewöhnlich: Ein Aufrufer kann darauf oft nicht sinnvoll reagieren. Daher sind Unchecked Exceptions in der Regel auch nicht Bestandteil der Methodensignatur (obwohl man sie dort explizit aufführen kann).

Man kann Unchecked Exceptions etwa zur Signalisierung ungültiger Parameterwerte in Form einer `IllegalArgumentException` nutzen.

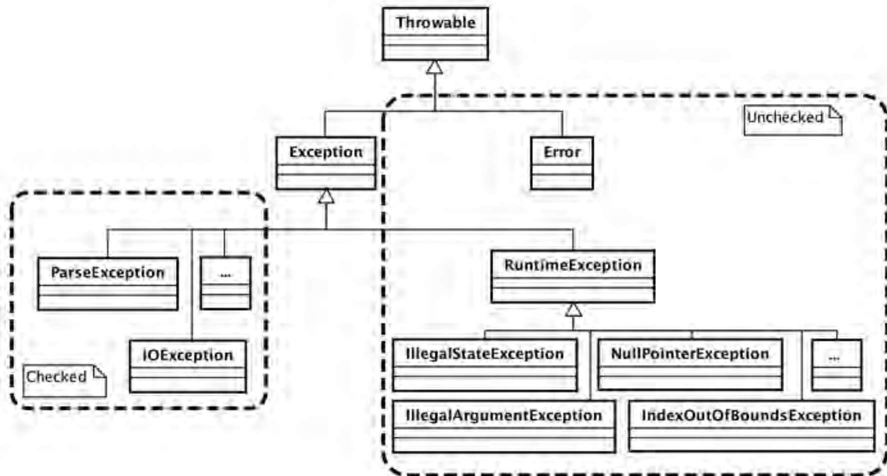

Abbildung 5-3 *Exception-Hierarchie*

Einige Programmierer geraten in die Versuchung, nur noch Unchecked Exceptions einzusetzen und eigene Exception-Klassen von `java.lang.RuntimeException` abzuleiten. Obwohl dies vom »lästigen« Bearbeiten einer möglicherweise auftretenden Exception befreit und auch keine Fehler beim Kompilieren provoziert, erschwert ein derartiges Vorgehen es manchmal doch, angemessen auf Fehlersituationen reagieren zu können: Einerseits sieht man in der Methodensignatur nicht unbedingt, dass eine Exception ausgelöst wird, weshalb dies in der Methodendokumentation aufgeführt werden sollte. Andererseits kann die Exception einfach ignoriert werden. Sie führt dann zur Laufzeit zu einem unerwarteten Programmfehler. Im Extremfall wird eine solche Exception unbehandelt bis zur `main()`-Methode bzw. `run()`-Methode des gerade aktiven Threads und damit zur JVM propagiert, wodurch dieser Thread terminiert.

Aus dieser Diskussion leiten wir folgenden Hinweis ab: ***Wenn ein Aufrufer eine außergewöhnliche Situation behandeln kann, so kann eine Checked Exception bevorzugt werden. Ist nicht davon auszugehen, dass ein Aufrufer die Fehlersituation korrigieren soll, oder ist ein Aufrufer dazu höchstwahrscheinlich nicht in der Lage, so ist eine Unchecked Exception*** die richtige Wahl.

5.6.3 Besonderheiten beim Exception Handling

Nachfolgend beschäftigen wir uns mit einigen Neuerungen aus JDK 7, die das Exception Handling vereinfachen und insbesondere auch das Resource Handling adressieren.

Multi Catch

Manchmal ist es wünschenswert, mehrere Exception-Typen gleichartig zu behandeln. Idealerweise möchte man dann nur einen gemeinsamen `catch`-Block definieren und nicht für jeden zu verarbeitenden Exception-Typ jeweils einen Block.

Betrachten wir zunächst die Situation vor JDK 7 und hier zur Demonstration eine `main()`-Methode, die eine weitere Methode `exceptionThrowingMethod()` aufruft, die wiederum zwei verschiedene Exceptions in ihrer Signatur definiert. Die klassische Umsetzung einer Behandlung sieht folgendermaßen aus:

```
public static void main(final String[] args)
{
    try
    {
        exceptionThrowingMethod();
    }
    // Gleichartige Behandlung durch zwei catch-Blöcke
    catch (final RemoteException ex)
    {
        reportException(ex);
    }
    catch (final FileNotFoundException ex)
    {
        reportException(ex);
    }
}

private static void exceptionThrowingMethod() throws RemoteException,
                                                     FileNotFoundException
{
    // ...
}
```

Mehrere identische `catch`-Block stellen eine Form der Sourcecode-Duplikation dar und verletzen das DRY-Prinzip (Don't Repeat Yourself). Für dieses Beispiel sollte man die identische Behandlung nur einmalig durchführen. Mit JDK 7 ist das Fangen mehrerer Exceptions mit nur einem `catch`-Block – ***Multi Catch*** genannt – wie folgt möglich:

```
public static void main(final String[] args)
{
    try
    {
        exceptionThrowingMethod();
    }
    // Mehrfachangabe unterschiedlicher Exceptions
    catch (final RemoteException | FileNotFoundException ex)
    {
        reportException(ex);
    }
}
```

Das Verhalten von Multi Catch ist dem mehrerer hintereinander liegender `catch`-Blöcke ähnlich. Es gibt jedoch kleinere Unterschiede – insbesondere gilt dies für die Abarbeitung bzw. die Spezialisierung und Überdeckung von Exceptions. Bei mehreren `catch`-Blöcken kann man eine explizite Behandlung auch für spezielle (Sub-)Typen realisieren, indem der speziellere Typ von Exception im Sourcecode zuerst wie folgt angegeben wird:

```
catch (final FileNotFoundException ex)
{
    reportFileException(ex);
}
catch (final IOException ex)
{
    reportIOException(ex);
}
```

Dieses Verhalten ist mithilfe von Multi Catch so nicht möglich, denn es gilt, dass im Multi Catch nur Exceptions mit unterschiedlichem Basistyp erlaubt sind. Schreiben wir trotzdem einmal Folgendes:

```
// Führt zu einem Compile-Error
catch (final FileNotFoundException | IOException ex)
{
   // ...
}
```

Für die gefangene `IOException` sowie deren Subtyp `FileNotFoundException` kommt es beim Kompilieren zu folgender Fehlermeldung – egal in welcher Reihenfolge Sie die beiden Exceptions angeben: »`The exception FileNotFoundException is already caught by the alternative IOException`«.

Final Rethrow

Das Sprachfeature Final Rethrow ermöglicht es, mehrere Exception-Typen aus einem einzigen `catch`-Block weiter zu propagieren. Somit ist es möglich, dass verschiedene Typen von Exceptions per `catch (Exception ex)` gefangen werden, diese dann aber spezifisch weiter propagiert werden können, ohne dies explizit ausprogrammieren zu müssen. Dadurch kann der `catch`-Block übersichtlich gehalten und um gewünschte Funktionalität ergänzt werden.

Betrachten wir zum Verständnis die Methode `performCalculation(String)`, die laut Signatur zwei verschiedene Typen von Exceptions werfen kann:

```
private void performCalculation(final String fileName) throws IOException,
                                                              RemoteException
```

Nehmen wir an, wir wollten diese Methode derart erweitern, dass zur besseren Wartbarkeit eine gegebenenfalls auftretende Exception in eine Log-Datei geschrieben und anschließend weiter propagiert wird. Folgende Realisierungsidee (ohne den Einsatz von Multi Catch) wäre denkbar: Durch die Notation `catch (Exception)` werden einfach

alle Typen von Exceptions gefangen, dann erfolgt ein Logging der gefangenen Exception und über `throw ex` deren Propagierung:

```
private void finalRethrowV2(final String fileName) throws IOException,
                                                           RemoteException
{
    try
    {
        performCalculation(fileName);
    }
    // Das final verhindert für erste Versionen von JDK 7 einen Compile-Error
    catch (final Exception ex)
    {
        log.error("exception occurred", ex);
        throw ex; // Compile-Error vor JDK 7
    }
}
```

Tatsächlich kompiliert diese Umsetzung vor JDK 7 nicht, da durch die Anweisung `throw ex` lediglich der allgemeine Typ `Exception`, nicht aber die in der Signatur definierten Exceptions propagiert werden. Zudem ist unklar, welche Typen von Exceptions propagiert werden können. Eine (wenn auch schlechte) Idee könnte darin bestehen, den Typ `Exception` in die Signatur aufzunehmen:

```
// ACHTUNG, nicht machen, ganz schlecht
private void finalRethrowBad(final String fileName) throws IOException,
                                                            RemoteException,
                                                            Exception
```

Mit einer solchen Angabe verliert die Signatur allerdings sämtliche Aussagekraft bezüglich der von der Methode ausgelösten Exceptions. Es wird lediglich deutlich, dass überhaupt Exceptions auftreten können.

Praktischerweise ist es seit JDK 7 möglich, mehrere Typen von Exceptions allgemein zu fangen und diese spezialisiert weiterzuleiten. Während es in früheren Versionen von JDK 7 noch notwendig war, die im `catch` angegebene Exception `final` zu definieren – woher auch der Name Final Rethrow herrührt –, kann in den aktuellen Java-Versionen auf die Angabe von `final` verzichtet werden. Die moderneren Compiler können die Unveränderlichkeit einer Variablen selbst erkennen und erfordern keine explizite Kennzeichnung mit `final` mehr. Man spricht in diesem Zusammenhang auch von »effectively final«.

Automatic Resource Management (ARM)

Bis hierher haben wir gesehen, dass Aufräumarbeiten bei I/O-Operationen durch die dazu notwendigen `try`-`catch`-Blöcke recht umfangreich und unleserlich werden können. Mit JDK 7 wurde genau für diese Aufgaben ein automatisches Ressourcenmanagement in Java integriert. Dieses entlastet den Entwickler von manuellen Aufräumarbeiten und es hilft dabei, weniger Fehler zu machen. Betrachten wir den Aufwand der manuellen Aufräumarbeiten ohne ARM beim Einsatz eines `BufferedReader`s:

```
// I/O ohne ARM
public static String readFirstLine(final String path) // throws IOException
{
    BufferedReader br = null;
    try
    {
        br = new BufferedReader(new FileReader(path));
        return br.readLine();
    }
    catch (final IOException ex)
    {
        // handle or rethrow
    }
    finally
    {
        // Diese manuellen Aufräumarbeiten werden durch ARM überflüssig
        try
        {
            if (br != null)
            {
                br.close();
            }
        }
        catch(final IOException ioe)
        {
            // ignore
        }
    }
    return "";
}
```

Zur Aktivierung des ARM muss die Verarbeitung in einem speziellen `try`-Block und mit der Angabe der später freizugebenden Ressourcenvariablen erfolgen:

```
public static String readFirstLine(final String path)
{
    // Spezielle Angabe der Ressourcenvariablen
    try (final FileReader fr = new FileReader(path);
         final BufferedReader br = new BufferedReader(fr))
    {
        return br.readLine();
    }
    catch (final IOException ex)
    {
        // handle or rethrow
    }
    return "";
}
```

Für jede in den runden Klammern des `try`-Blocks angegebene Variable wird beim Verlassen des `try`-Blocks *automatisch* die Methode `close()` aufgerufen. Voraussetzung dafür ist, dass die dort definierten Referenzvariablen das Interface `java.lang.AutoCloseable` erfüllen. Dann werden die Aufrufe vom Compiler an den korrespondierenden Stellen in den Bytecode generiert.

Wie im Listing gezeigt, sollte man Verkettungen von Konstruktoraufrufen vermeiden, da dann möglicherweise nicht alle Ressourcen freigegeben werden. Vielmehr ist eine separate Definition wie oben zu empfehlen.

5.6.4 Exception Handling und Ressourcenfreigabe

Nachdem wir gerade ARM kennengelernt haben, möchte ich das Thema Exceptions und Ressourcenfreigabe nun ausführlicher behandeln. Zur Demonstration wähle ich eine Netzwerkkommunikation über Sockets – zur Einführung finden Sie einige Informationen zu Sockets im nachfolgenden Hinweis »Netzwerkkommunikation mit Sockets im Kurzüberblick«. Das Beispiel zeigt eine unzureichende Fehlerbehandlung. Das Ganze wird schrittweise verbessert. In der letzten Ausbaustufe ist dann sichergestellt, dass belegte Systemressourcen auch wieder freigegeben werden.

Ausgangsbeispiel

Im nachfolgenden Listing wird ein `ServerSocket` erzeugt und dann durch Aufruf einer Methode `handleIncomingConnections()` auf eingehende Daten gewartet.

```
// Beispiel für schlechtes EXCEPTION HANDLING
public static void openAndProcess()
{
    try
    {
        final int AUTO_ALLOCATE_PORT = 0;
        final ServerSocket serverSocket = new ServerSocket(AUTO_ALLOCATE_PORT);

        // Weitere Verarbeitung - hier jedoch uninteressant
        handleIncomingConnections(serverSocket);

        serverSocket.close();
    }
    catch (final IOException ex)
    {
        ex.printStackTrace();
    }
}
```

Wie bereits angedeutet ist die Implementierung der Fehlerbehandlung problematisch: Solange alle Aufrufe erfolgreich sind bzw. ohne Exceptions ablaufen, wird nach Abarbeitung der durch einen Kommentar angedeuteten Aktionen das Socket durch einen Aufruf der Methode `close()` wieder geschlossen. Wird jedoch eine Exception ausgelöst, so wird die Ausführung innerhalb des `try`-Blocks unterbrochen und es werden die Anweisungen des `catch`-Blocks ausgeführt – im Beispiel kommt es aber *nicht* zum Aufruf von `close()`. Für diesen Fall bleibt demnach die vom Betriebssystem bereitgestellte Netzwerkressource alloziert (zumindest eine gewisse Zeit).

Hinweis: Netzwerkkommunikation mit Sockets im Kurzüberblick

Sockets bilden logische Endpunkte einer Verbindung zwischen Computern und erlauben das Senden und Empfangen von Nachrichten. Die Kommunikation kann z. B. über TCP erfolgen. ***TCP*** steht für ***Transmission Control Protocol*** und stellt eine sichere und fehlerfreie Verbindung zwischen zwei Kommunikationsendpunkten im Netz dar. Ähnlich wie in einem Telefonnetz einer Firma können Kommunikationspartner über Durchwahlnummern, sogenannte ***Ports***, erreicht werden.

Sockets bieten eine streambasierte Schnittstelle zur Kommunikation. Bei dieser ***Client-Server-Kommunikation*** stellt ein Client den aktiven Part der Kommunikation dar: Er schickt Anfragen an einen Server, der wiederum Antworten an den Client sendet. Dazu wartet ein Server mithilfe eines `ServerSocket`s auf eingehende Verbindungen. Die Kommunikation zwischen Client und Server wird dann mit der Klasse `Socket` realisiert, die Zugriff auf jeweils eine `InputStream`- und eine `OutputStream`-Instanz bietet.

Schritt 1: Ressourcenfreigabe im Fehlerfall

Ein erster Reparaturschritt besteht darin, im `catch`-Block einen Aufruf der Methode `close()` zu ergänzen. Das erfordert aber einige Umbauarbeiten: Weil wir Zugriff auf die Variable `serverSocket` im `catch`-Block benötigen, muss diese vor dem `try-catch`-Block definiert werden. Zudem muss der Aufruf von `close()` im `catch`-Block durch einen weiteren `try-catch`-Block ummantelt werden, da in der Signatur der `close()`-Methode eine `IOException` definiert ist. Auf diese Exception kann man eher selten sinnvoll und oftmals nur durch einen leeren `catch`-Block reagieren.

```
// Beispiel für schlechtes EXCEPTION HANDLING
public static void openAndProcess()
{
    ServerSocket serverSocket = null;
    try
    {
        final int AUTO_ALLOCATE_PORT = 0;
        serverSocket = new ServerSocket(AUTO_ALLOCATE_PORT);

        handleIncomingConnections(serverSocket);

        serverSocket.close();
    }
    catch (final IOException ex)
    {
        if (serverSocket != null)
        {
            try
            {
                serverSocket.close();
            }
            catch (final IOException e)
            {
                // Sollte close() eine IOException werfen? Ja und Nein!
                // Häufig ist das eher unpraktisch. Es ist aber hilfreich,
                // wenn sich das Programm merken will, dass eine Ressource
                // nicht korrekt geschlossen werden konnte und der weiter-
                // gehende Zugriff erst einmal unterbunden werden soll.
            }
        }
    }
}
```

Der Konstruktor der Klasse `ServerSocket` kann eine `IOException` auslösen. Daher darf man im `catch`-Block nicht davon ausgehen, dass der Variablen `serverSocket`

ein Wert zugewiesen wurde und diese ungleich `null` ist. Vor dem Aufruf von `close()` muss daher eine Prüfung auf `null` erfolgen. Solche Spezialbehandlungen machen den Sourcecode unübersichtlich. Sowohl die bedingte Anweisung als auch die Behandlung der `IOException` erzeugen weitere Komplexität und reduzieren die Lesbarkeit. Diese »hässliche« Form der Fehlerbehandlung sieht man – bedingt durch das Java-API bei Sockets und beim Stream-I/O – leider häufiger. Darüber hinaus sind die mehrfachen Aufrufe von `close()` eine Form der Duplikation, was gegen das DRY-Prinzip verstößt.

Schritt 2: Duplikation entfernen

Die Sourcecode-Duplikation lässt sich leicht korrigieren, indem man eine Methode `closeServerSocket()` herausfaktoriert, die aus den Anweisungen des `catch`-Blocks besteht: Die Methode ist tolerant gegenüber der Übergabe von `null`-Werten und zudem wird die in der Signatur der Methode `close()` angegebene `IOException` ignoriert, da man diese kaum sinnvoll behandeln kann.

```
private static void closeServerSocket(final ServerSocket serverSocket)
{
    if (serverSocket != null)
    {
        try
        {
            serverSocket.close();
        }
        catch (final IOException e)
        {
            // oftmals keine sinnvolle Behandlung möglich (s.o.)
        }
    }
}
```

Derartige Hilfsmethoden machen den Sourcecode oftmals deutlich übersichtlicher, kürzer und besser lesbar. Dadurch erhalten wir folgende Methode, die den gemeinsamen Aufruf von `closeServerSocket()` hinter den `catch`-Block verlagert:

```
public static void openAndProcess()
{
    ServerSocket serverSocket = null;
    try
    {
        final int AUTO_ALLOCATE_PORT = 0;
        serverSocket = new ServerSocket(AUTO_ALLOCATE_PORT);

        handleIncomingConnections(serverSocket);

        // nach unten verlagert: closeServerSocket(serverSocket);
    }
    catch (final IOException ex)
    {
        // nach unten verlagert: closeServerSocket(serverSocket);
    }
    // DRY: keine Duplikation
    closeServerSocket(serverSocket);
}
```

Das war bereits ein guter Schritt in die richtige Richtung: Die Methode ist nun deutlich besser lesbar und behandelt Fehler relativ gut. Wieso nur relativ gut? Wir haben noch ein »Schlupfloch« vergessen. Es existiert ein Problem, wenn `return`-Anweisungen im `try`- oder im `catch`-Block verwendet werden oder dort Exceptions ausgelöst oder weiter propagiert werden. Wird ein `try`- oder `catch`-Block auf diese Weise verlassen, so wird der dem `catch`-Block nachfolgende Sourcecode, in diesem Fall die `closeServerSocket()`-Methode, nicht ausgeführt. Dadurch kann sich das Programmverhalten in Fehlersituationen auf unbestimmte Weise verändern. Möglicherweise werden belegte Systemressourcen nicht wieder freigegeben. Betrachten wir mögliche Abhilfen.

Schritt 3: Variante A: Nutzung von `finally`

Mithilfe des Schlüsselworts `finally` kann das Exception Handling elegant und robust gestaltet werden: Laut JLS (Java Language Specification) wird der `finally`-Block nämlich immer ausgeführt,[19] und zwar unabhängig davon, ob

- der `try`-Block normal beendet,
- eine Exception im `try`-Block geworfen oder
- ein `return` im `try`- und `catch`-Block ausgeführt wird.

Durch dieses Verhalten lassen sich Systemressourcen freigeben – unabhängig davon, wie eine Methode beendet wird, sogar bei einem `return` im `try`- und `catch`-Block:

```
public static boolean openAndProcess()
{
   ServerSocket serverSocket = null;
   try
   {
      final int AUTO_ALLOCATE_PORT = 0;
      serverSocket = new ServerSocket(AUTO_ALLOCATE_PORT);

      handleIncomingConnections(serverSocket);

      return true; // Ausstieg zur Demonstration
   }
   catch (final IOException e)
   {
      // Die Freigabe der Ressource erfolgt hier in finally. Exception darf
      // nicht propagiert werden, weil sie nicht in der Signatur definiert ist.

      return false;
   }
   finally
   {
      closeServerSocket(serverSocket);
   }
}
```

Trotz der ganzen Korrekturen sind wir immer noch gezwungen, eine `IOException` abzufangen und einen `catch`-Block zu schreiben. Das liegt daran, dass die `IOException`

[19] Außer beim Aufruf von `System.exit()`.

eine sogenannte Checked Exception ist: Wie schon in Abschnitt 5.6.2 beschrieben, muss diese entweder per `throws` in der Methodensignatur definiert sein oder aber durch einen `catch`-Block behandelt werden. Des Weiteren definiert die Methode `openAndProcess()` in ihrer Signatur keine Exception, sodass wir dies aus Kompatibilitätsgründen für diese öffentliche Methode auch nicht ändern dürfen. Natürlich sollten wir aber belegte Systemressourcen freigeben. In unserem Beispiel geschieht dies durch den Methodenaufruf von `closeServerSocket()` im `finally`-Block in jedem Fall.

Schritt 3: Variante B: Nutzung von ARM

Das mit JDK 7 eingeführte und in Abschnitt 5.6.3 beschriebene ***Automatic Resource Management*** (***ARM***) erlaubt es, das Ganze eleganter und kürzer zu implementieren. Praktischerweise muss man sich dann nicht mehr selbst um die Freigabe von Ressourcen kümmern, sondern nur die Syntax mit den runden Klammer wie folgt nutzen:

```
public static void openAndProcess()
{
   final int AUTO_ALLOCATE_PORT = 0;
   try (final ServerSocket serverSocket = new ServerSocket(AUTO_ALLOCATE_PORT))
   {
      handleIncomingConnections(serverSocket);
      return true;
   }
   catch (final IOException ex)
   {
      // Die Freigabe der Ressource erfolgt hier durch ARM. Die Exception darf
      // nicht propagiert werden, weil sie nicht in der Signatur definiert ist.

      return false;
   }
}
```

5.6.5 Assertions

Nachdem wir uns nun recht intensiv mit Exceptions, vor allem im Zusammenhang mit I/O und Ressourcenmanagement, beschäftigt haben, möchte ich nachfolgend auf Assertions (Zusicherungen) eingehen. Diese dienen dazu, erwartete Zustände abzusichern. Zur Formulierung einer solchen Zusicherung wird das Schlüsselwort `assert` sowie eine boolesche Bedingung angegeben. Wird diese zu `false` evaluiert, so wird ein `java.lang.AssertionError` ausgelöst. Das ist ein Subtyp von `java.lang.Throwable`. Demzufolge können `AssertionError`s zwar durch einen `try`-`catch`-Block bearbeitet werden, allerdings ist dieses Vorgehen selten sinnvoll, da man einen Programmfehler aufdecken und nicht verschweigen möchte. Dabei sollte man zusätzlich wissen, dass die ***Verarbeitung von Assertions für die JVM standardmäßig deaktiviert ist und explizit aktiviert werden muss***, damit die Zusicherungen tatsächlich ausgewertet werden.

Assertions am Beispiel

Aus einem Eingabestring `versions` sollen die zwei Versionsinformationen `majorVersion` und `minorVersion` extrahiert werden. Jeder Bestandteil der Versionsinformation muss mindestens aus einem Zeichen bestehen. Dazu wird die Länge mithilfe von Assertions geprüft. Die aus der Eingabe korrespondierenden Teilstrings ermitteln wir mithilfe eines `StringTokenizer`s und dessen Methode `nextToken()`. Um die spätere Verarbeitung und gegebenenfalls eine Umwandlung in Zahlen zu vereinfachen, werden Leerzeichen durch einen Aufruf von `trim()` abgeschnitten.

```
public static void main(final String[] args)
{
    // ACHTUNG: fehlendes Token Minor-Version
    final String versions = "12. ";
    final StringTokenizer tokenizer = new StringTokenizer(versions, ".");

    final int tokenCount = tokenizer.countTokens();
    if (tokenCount > 1)
    {
        // Versionen auslesen
        final String majorVersion = tokenizer.nextToken().trim();
        final String minorVersion = tokenizer.nextToken().trim();

        // Sicherstellen, dass Tokens einen Wert enthalten
        assert !majorVersion.isEmpty();
        assert !minorVersion.isEmpty();

        System.out.println("Major: '" + majorVersion + "'");
        System.out.println("Minor: '" + minorVersion + "'");
        System.out.println("#Tokens: '" + tokenCount + "'");
    }
    else
    {
       System.err.println("Unexpected version number format => no '.' found");
    }
}
```

Listing 5.36 *Ausführbar als* '**ASSERTEXAMPLE**'

Im Beispiel enthält der zu verarbeitende Eingabewert bewusst einen Leerstring als zweite Versionsinformation, um die Funktionsweise von Assertions zu demonstrieren. Deshalb sollte der String `minorVersion` in diesem speziellen Fall eine Länge von 0 besitzen und damit die Zusicherung `!minorVersion.isEmpty()` verletzen. Demnach erwartet man bei der Eingabe von "`12. `" ein Fehlschlagen der Assertion und als Folge einen Programmabbruch mit einem `AssertionError`. Allerdings läuft das Programm ASSERTEXAMPLE ohne Fehler. Zunächst ist das möglicherweise verwunderlich, doch die folgende Konsolenausgabe bringt uns auf die richtige Spur:

```
Major: '12'
Minor: ''
#Tokens: '2'
```

Somit ist es möglich, dass andere Threads zu einem beliebigen Zeitpunkt der Ausführung der obigen Anweisungsfolge aktiviert werden. Dadurch ist keine Atomarität und keine Thread-Sicherheit mehr gegeben. Daher muss jede Folge von Anweisungen, die exklusiv durch einen Thread ausgeführt werden soll, als kritischer Abschnitt geschützt werden.

Tipp: Besonderheit Postinkrement

Dass das Postinkrement `counter++` aus drei Einzelschritten wie zuvor gezeigt besteht, können Sie am Rückgabewert der folgenden Methode nachvollziehen, die den Wert 10 und nicht den Wert 11 liefert, wie man es eventuell zunächst erwarten würde.

```
public static int postIncrementSurprise()
{
    int counter = 10;
    return counter++;
}
```

Atomare Variablen als Lösung?

Die Klassen `AtomicInteger` und `AtomicLong` ermöglichen atomare Read-Modify-Write-Sequenzen, wie `counter++`.[7] Die Atomic-Klassen nutzen eine sogenannte ***Compare-and-Swap-Operation*** (CAS). Die Besonderheit ist, dass diese zunächst einen Wert aus dem Speicher lesen und mit einem erwarteten Wert vergleichen, bevor sie eine Zuweisung mit einem übergebenen Wert durchführen. Dies geschieht allerdings nur, wenn ein erwarteter Wert gespeichert ist. Übertragen auf das vorherige Beispiel des Zählers würde dies unter Verwendung der Klasse `AtomicLong` wie folgt aussehen:

```
public long incrementAndGetUsingCAS()
{
    long oldValue = counter.get();
    // Unterbrechung möglich
    while (!atomicLongCounter.compareAndSet(oldValue, oldValue + 1))
    {
        // Unterbrechung möglich
        oldValue = atomicLongCounter.get();
        // Unterbrechung möglich
    }
    return oldValue + 1;
}
```

Wie man sieht, ist selbst das einfache Hochzählen eines Werts deutlich komplizierter als ein Einsatz von `synchronized`. Das liegt vor allem daran, dass in einer Schleife geprüft werden muss, ob das Setzen des neuen Werts korrekt erfolgt ist. Diese Komplexität entsteht dadurch, dass zwischen dem Lesen des Werts und dem Setzen ein anderer Thread den Wert verändert haben könnte. In einem solchen Fall wird die Schleife so lange wiederholt, bis eine Inkrementierung erfolgreich ist.

[7] Für Gleitkommatypen gibt es keine Unterstützung.

Auf einer solch niedrigen Abstraktionsebene möchte man in der Regel nicht arbeiten. Zur Vereinfachung werden die Details durch verschiedene Methoden der atomaren Klassen gekapselt: Ein sicheres Inkrementieren erfolgt beispielsweise mit der Methode `incrementAndGet()`. Weitere Details zu den atomaren Klassen finden Sie in den Büchern »Java Concurrency in Practice« von Brian Goetz [24] und »Java Threads« von Scott Oaks und Henry Wong [57].

Wie bei `volatile` gilt, dass sobald mehrere Attribute konsistent zueinander geändert werden sollen, eine Definition eines kritischen Abschnitts über `synchronized` oder Locks zwingend notwendig wird.

10.4.3 Reorderings

Die JVM darf gemäß der JLS zur Optimierung beliebige Anweisungen in ihrer Ausführungsreihenfolge umordnen, wenn dabei die Semantik des Programms nicht verändert wird. Bei der Entwicklung von Singlethreading-Anwendungen muss man den sogenannten ***Reorderings*** keine Beachtung schenken. Für Multithreading gibt es jedoch einige Dinge zu berücksichtigen. Betrachten wir zur Verdeutlichung eine Klasse `ReorderingExample` mit den Attributen `x1` und `x2` und folgenden Anweisungen:

```
public class ReorderingExample
{
    int x1 = 0;
    int x2 = 0;

    void method()
    {
        // Thread 1
        x1 = 1;                                                    // #1
        x2 = 2;                                                    // #2
        System.out.println("x1 = " + x1 + " / x2 = " + x2 );      // #3
    }
}
```

Werden die obigen Anweisungen ausgeführt, so kann der Compiler die Anweisungen #1 und #2 der Methode `method()` vertauschen, ohne dass dies Auswirkungen auf die nachfolgende Anweisung #3, hier die Ausgabe, hat. Anweisung #3 kann jedoch nicht mit Anweisung #1 oder #2 getauscht werden, da die Ausgabe lesend auf die zuvor geschriebenen Variablen zugreift. Reorderings sind demnach nur dann erlaubt, wenn sichergestellt werden kann, dass es keine Änderungen an den Wertzuweisungen der sequenziellen Abarbeitung der Befehle gibt. Es kommt also immer zu der folgenden Ausgabe: `x1 = 1 / x2 = 2`.

Für Multithreading wird die Situation komplizierter: Würde parallel zu den obigen Anweisungen in einem separaten Thread etwa folgende Methode `otherMethod()` dieser Klasse abgearbeitet, so kann es zu unerwarteten Ergebnissen kommen.

```
void otherMethod()
{
    // Thread 2
    int y1 = x2;                                          // #1
    int y2 = x1;                                          // #2
    System.out.println("x1 = " + x1 + " / x2 = " + x2 );  // #3
    System.out.println("y1 = " + y1 + " / y2 = " + y2 );  // #4
}
```

Betrachten wir zunächst den einfachen Fall ohne Reorderings. Wird Thread 1 vor Thread 2 ausgeführt, also `method()` komplett vor `otherMethod()` abgearbeitet, dann gilt offensichtlich $x1 = 1 = y2$ und $x2 = 2 = y1$. Dies ergibt sich daraus, dass alle Schreiboperationen in Thread 1 bereits ausgeführt wurden, bevor die Leseoperationen in Thread 2 durchgeführt werden. Werden die beiden Threads allerdings abwechselnd, beliebig ineinander verwoben, ausgeführt, so kann es zu merkwürdigen Ausgaben kommen. Denkbar ist etwa eine Situation, in der zwar wie vermutet $x1 = 1$ und $x2 = 2$ gilt, aber auch $y1 = 0$ und $y2 = 1$. Dies ist der Fall, wenn zunächst die Ausführung von Thread 1 (`method()`) begonnen und nach der Zuweisung $x1 = 1$ unterbrochen wird und dann Thread 2 ausgeführt wird, wie dies im Folgenden beispielhaft dargestellt ist:

```
int x1 = 0;
int x2 = 0;

// Thread 1
                                  // Thread 2
x1 = 1;
                                  y1 = x2 = 0;
                                  y2 = x1 = 1;
                                  // System.out: x1 = 1 / x2 = 0
                                  // System.out: y1 = 0 / y2 = 1
x2 = 2;
// System.out: x1 = 1 / x2 = 2
```

Wie man leicht sieht, kann es bereits beim abwechselnden Ausführen (***Interleaving***) von Threads zu Inkonsistenzen kommen. War obige Ausgabe noch mit etwas Nachdenken intuitiv verständlich, ist jedoch auch folgende, unerwartete Ausführungsreihenfolge möglich, wenn die Anweisungen umgeordnet werden:

```
int x1 = 0;
int x2 = 0;

// Thread 1
                                  // Thread 2
                                  y1 = x2 = 0;
x2 = 2;
                                  y2 = x1 = 0;
                                  // System.out: x1 = 0 / x2 = 2
                                  // System.out: y1 = 0 / y2 = 0
x1 = 1;
// System.out: x1 = 1 / x2 = 2
```

Man erkennt einige Besonderheiten beim Zusammenspiel von Threads und gemeinsamen Variablen: Es kann zu Inkonsistenzen durch Reorderings und Interleaving kommen. Für Singlethreading analysiert und ordnet die JVM die Folge von Schreib- und Lesezugriffen automatisch, sodass diese Probleme gar nicht erst entstehen können. Bei Multithreading ist die zeitliche Reihenfolge der Abarbeitung nicht im Vorhinein bekannt, sodass keine definierte Reihenfolge bezüglich des Schreibens und Lesens durch verschiedene Threads gegeben ist und die JVM keine Konsistenz sicherstellen kann. ***Die Folge ist, dass ein Programm zufällig (häufig sogar korrekte) Resultate liefert, aber im schlimmsten Fall unbrauchbar wird***. Bei Multithreading ist es daher Aufgabe des Entwicklers, der JVM Hinweise zu geben, in welcher Reihenfolge die Abarbeitungen erfolgen sollen bzw. welche Bereiche kritisch sind und zu welchen Zeitpunkten keine Reorderings stattfinden dürfen.

Für Multithreading ist dazu eine spezielle Ordnung ***Happens-before*** (hb) definiert, die für zwei Anweisungen A und B beliebiger Threads besagt, dass für B alle Änderungen von Variablen einer Anweisung A sichtbar sind, wenn $hb(A, B)$ gilt. Durch diese Ordnung wird indirekt festgelegt, in welchem Rahmen Reorderings stattfinden dürfen, da durch $hb(A, B)$ »Abstimmungspunkte« oder »Synchronisationspunkte« von Multithreading-Applikationen definiert werden. Zwischen diesen Abstimmungspunkten sind Reorderings allerdings möglich. An solchen Abstimmungspunkten im Programm weiß man dann jedoch sicher, dass alle Änderungen stattgefunden haben und diese für andere Threads sichtbar sind. Beispielsweise kann über Synchronisation eine gewisse Abarbeitungsreihenfolge erreicht werden: Zwei über dasselbe Lock synchronisierte kritische Abschnitte schließen sich gegenseitig aus. Laut $hb(A, B)$ gilt, dass nachdem einer von beiden abgearbeitet wurde, alle Modifikationen für den nachfolgenden sichtbar sind. Über die Reihenfolge der Ausführung innerhalb eines `synchronized`-Blocks kann allerdings keine Aussage getroffen werden. ***Besteht die Happens-before-Ordnung zwischen zwei Anweisungen jedoch nicht, so darf die JVM den Ablauf von Befehlen beliebig umordnen***.

Hintergrundinformationen zur Ordnung hb

Zum besseren Verständnis des Ablaufs bei Multithreading ist es hilfreich zu wissen, für welche Anweisungen eine Happens-before-Ordnung gilt:

- Zwei Anweisungen A und B erfüllen $hb(A, B)$, wenn B *nach* A im selben Thread in der sogenannten Program Order steht. Vereinfacht bedeutet dies, dass jeder Schreibzugriff für folgende Lesezugriffe auf ein Attribut sichtbar ist. Erfolgen Lese- und Schreibzugriffe nicht auf gleiche Attribute, stehen Anweisungen also nicht in einer Lese-Schreib-Beziehung, dürfen diese vom Compiler beliebig umgeordnet werden. Bei Singlethreading macht sich dies nicht bemerkbar, da trotz Reorderings die Bedeutung nicht verändert wird.

- Es gilt $hb(A, B)$, wenn zwei Anweisungen A und B über denselben Lock synchronisiert sind.
- Für einen Schreibzugriff A auf ein `volatile`-Attribut und einen späteren Lesezugriff B gilt $hb(A, B)$.
- Für den Aufruf von `start()` (A) eines Threads und alle Anweisungen B, die in diesem Thread ausgeführt werden, gilt $hb(A, B)$.
- Für alle Anweisungen A eines Threads vor dessen Ende B gilt $hb(A, B)$, d. h., alle nachfolgenden Threads können diese Änderungen sehen – aber nur, wenn diese über den gleichen Lock synchronisiert sind oder `volatile`-Attribute verwenden.

In Multithreading-Programmen muss man daher als Programmierer dafür sorgen, eine Happens-before-Ordnung herzustellen, um mögliche Fehler durch Reorderings zu verhindern. Auch ohne sämtliche Details dieser Ordnung zu verstehen, kann man Thread-sichere Programme schreiben, wenn man sich an folgende Grundregeln hält:

1. **Korrekte, minimale, aber vollständige Synchronisierung** – Greifen mehrere Threads auf ein gemeinsam benutztes Attribut zu, so müssen immer *alle* Zugriffe über *dasselbe* Lock-Objekt synchronisiert werden. Für semantische Einheiten von Attributen kann auch ein und dasselbe Lock-Objekt verwendet werden.
2. **Beachtung von Atomarität** – Zuweisungen erfolgen in der Regel atomar. Für die 64-Bit-Datentypen muss dies explizit über das Schlüsselwort `volatile` sichergestellt werden. Mehrschrittoperationen (beispielsweise `i++`) können so nicht geschützt werden und müssen zwingend synchronisiert werden.

Tipp: Eigenschaften der Ordnung hb

Die Ordnung hb besitzt für beliebige Anweisungen A und B folgende Eigenschaften:

- **Irreflexiv** – $!hb(A, A)$ – Keine Anweisung A »sieht« ihre eigenen Änderungen.
- **Transitiv** – Wenn $hb(A, B)$ und $hb(B, C) \Rightarrow hb(A, C)$ – Die Transitivität sorgt für die Sichtbarkeit gemäß der Reihenfolge der Anweisungen im Sourcecode.
- **Antisymmetrisch** – Wenn $hb(A, B)$ und $hb(B, A) \Rightarrow A = B$ oder hier intuitiver: Wenn $hb(A, B)$ und $A \neq B \Rightarrow !hb(B, A)$ – Die Antisymmetrie stellt sicher, dass eine Anweisung A nicht die Modifikationen einer Nachfolgeanweisung B »sieht«. Tatsächlich ist dies schon durch die Irreflexivität ausgeschlossen.

10.5 Besonderheiten bei Threads

Nachdem wir recht ausführlich den Lebenszyklus und die Zusammenarbeit von Threads kennengelernt haben, geht dieser Abschnitt auf Besonderheiten ein. Dazu stelle ich zunächst verschiedene Arten von Threads vor. Danach betrachten wir, wie sich Exceptions in Threads auswirken. Anschließend greife ich das Beenden von Threads erneut auf und zeige, wie dies sicher realisiert werden kann. Abschließend stelle ich Möglichkeiten zur zeitgesteuerten Ausführung von Aufgaben vor.

10.5.1 Verschiedene Arten von Threads

Bei der Arbeit mit Threads gibt es noch ein bisher nur am Rande erwähntes Detail zu beachten: In Java unterscheidet man zwischen User- und Daemon-Threads.

`main`-Thread und User-Threads

Wie bereits bekannt, erzeugt die JVM beim Start einen speziellen Thread, den man `main`-Thread nennt, weil dieser statt der `run()`-Methode die `main()`-Methode des Programms ausführt. Nehmen wir an, eine Applikation würde mehrere Threads aus dem `main`-Thread erzeugen und starten. Diese Threads nennt man ***User-Threads***. Die JVM bleibt selbst nach Ausführung des `main`-Threads bzw. der letzten Anweisung der `main()`-Methode aktiv, solange noch vom `main`-Thread abgespaltene User-Threads existieren und deren `run()`-Methode ausgeführt wird.

Daemon-Threads

Manchmal sollen Aufgaben im Hintergrund ablaufen und die Terminierung der JVM von solchen Threads unabhängig sein. Werden aber lediglich User-Threads gestartet, so wird die JVM nicht beendet, solange noch einer von diesen läuft. Als Abhilfe gibt es sogenannte ***Daemon-Threads***. Einen solchen erhält man, wenn man einen beliebigen Thread, d. h. ein `Thread`-Objekt, durch Aufruf der Methode `setDaemon(boolean)` *vor* seiner Ausführung zu einem Daemon-Thread umwandelt. Daemon-Threads sind praktisch, wenn die Hintergrundaufgaben nicht wirklich zur eigentlichen Programmfunktionalität beitragen. Das gängigste Beispiel ist der Garbage Collector, der im Hintergrund Speicher aufräumt (vgl. Abschnitt 12.4).

Den Unterschied zwischen User- und Daemon-Threads erkennt man, wenn ein Programm bzw. die zugehörige JVM terminieren soll. Bekanntermaßen geschieht dies nur, nachdem alle User-Threads beendet sind. Dann noch aktive Daemon-Threads werden abrupt beendet, d. h. irgendwo in der Abarbeitung ihrer `run()`-Methode. Daher muss man sorgsam sein, wenn Daemon-Threads Ressourcen belegen. Zu deren Freigabe kann man in einer eigenen, von `Thread`-abgeleiteten Klasse die Methode `finalize()` implementieren. Weil deren Abarbeitung laut JLS jedoch nicht garantiert ist, bietet sich ein Shut-down-Hook (vgl. Abschnitt 16.1.12) an.

10.5.2 Exceptions in Threads

In diesem Abschnitt wollen wir uns damit beschäftigen, was passiert, wenn während der Abarbeitung eines Threads eine Exceptions ausgelöst wird. Nehmen wir dazu an, eine Applikation würde mehrere Threads aus dem `main`-Thread starten und in irgendeinem dieser Threads würde eine Ausnahmesituation auftreten, die zu einer Exception führt. Aus Abschnitt 5.6.1 wissen wir, dass eine Exception mit einem `catch`-Block behandelt werden muss oder automatisch entlang der Methodenaufrufkette weitergereicht wird, bis ein passender `catch`-Block gefunden wird oder man schließlich die `run()`- bzw. `main()`-Methode erreicht.

Findet sich auch dort kein passender `catch`-Block, so bricht die Ausführung ab und dies führt zu einer Beendigung des ausführenden Threads. Abschließend wird die vom Programm unbehandelte Exception über den Error-Stream `System.err` inklusive des kompletten Stacktrace ausgegeben. Zum Nachvollziehen des zuvor beschriebenen Verhaltens können Sie folgendes Programm EXCEPTIONINTHREADSEXAMPLE ausführen:

```
// Achtung: Nur zur Demonstration des Exception Handlings
public static void main(final String[] args) throws InterruptedException
{
    exceptionInMethod();
}

static void exceptionInMethod() throws InterruptedException
{
    final Thread exceptional = new Thread()
    {
        public void run()
        {
            throw new IllegalStateException("run() failed");
        }
    };

    exceptional.start();
    Thread.sleep(1000);
}
```

Listing 10.9 *Ausführbar als* **'EXCEPTIONINTHREADSEXAMPLE'**

Es erscheint folgende Ausgabe (gekürzt) auf der Konsole:

```
Exception in thread "Thread-0" java.lang.IllegalStateException: run() failed
```

Wird das abrupte Ende eines Threads lediglich auf der Konsole ausgegeben, erschwert dies eine spätere Fehlersuche. Zum Nachvollziehen ist es nützlicher, Exceptions in eine Log-Datei zu schreiben. Dazu wollen wir eine kleine Utility-Klasse erstellen und greifen auf das mit Java 5 eingeführte innere Interface `UncaughtExceptionHandler` der Klasse `Thread` zurück. Dessen Implementierung erlaubt es, von `catch`-Blöcken unbehandelte Exceptions behandeln zu können, indem man in der Methode `uncaughtException(Thread, Throwable)` das gewünschte Verhalten realisiert.

Die folgende Klasse `LoggingUncaughtExceptionHandler` implementiert beispielsweise eine Ausgabe in eine Log-Datei. Im Listing ist eine `main()`-Methode gezeigt, die zu Demonstrationszwecken Exceptions provoziert:

```
public final class LoggingUncaughtExceptionHandler implements Thread.
      UncaughtExceptionHandler
{
   private static final Logger log = LogManager.getLogger("UncaughtExceptions");

   @Override
   public void uncaughtException(final Thread thread, final Throwable throwable)
   {
       log.error("Unexpected exception occured: ", throwable);
   }
}
```

In diesem Beispiel wird ein mögliches Problem zwar nicht weiter behandelt, aber zumindest in einer Log-Datei protokolliert, was eine spätere Analyse erleichtert.

Ein solcher `UncaughtExceptionHandler` kann für alle Threads durch Aufruf von `Thread.setDefaultUncaughtExceptionHandler(Thread.Uncaught-ExceptionHandler)` global gesetzt werden. Bei Bedarf kann dies für jeden Thread einzeln durch Aufruf von `setUncaughtExceptionHandler(Thread.Uncaught-ExceptionHandler)` erfolgen.

10.5.3 Sicheres Beenden von Threads

Wie bereits erwähnt, lassen sich Threads leider nicht so einfach beenden wie starten. Die Methode `stop()` der Klasse `Thread` ist als `@deprecated` markiert. In verschiedenen Quellen wird als Grund dafür genannt, dass beim Beenden eines Threads nicht alle Locks freigegeben werden. Das ist ein verbreiteter Irrtum. Definitiv werden beim Auftreten von Exceptions alle gehaltenen Locks durch die JVM zurückgegeben. Wäre dem nicht so, wäre kein sinnvolles Programmverhalten mehr möglich: Alle Aufrufe von `synchronized`-Methoden, die über diese Locks geschützt werden, wären für die restliche Laufzeit der JVM blockiert.

Der Grund für die Markierung als `@deprecated` ist vielmehr folgender: Wenn ein Thread durch Aufruf von `stop()` beendet wird, können dadurch die Daten, auf denen der Thread gerade gearbeitet hat, in einen inkonsistenten Zustand gebracht werden. Insbesondere gilt dies, wenn der Thread innerhalb einer eigentlich atomaren Anweisungsfolge unterbrochen wird: Wird `stop()` durch einen anderen Thread mitten während der Ausführung eines über `synchronized` definierten kritischen Bereichs aufgerufen, so wird dieser irgendwo unterbrochen und nicht mehr atomar ausgeführt. Eine mögliche Inkonsistenz im Objektzustand ist die Folge.

Um Konsistenz zu wahren, müssen wir andere Wege finden, einen Thread korrekt zu beenden. Zwei Möglichkeiten, dies sauber zu lösen, sind die folgenden:

1. Einführen einer Hilfsklasse, die die Funktionalität bereitstellt
2. Beenden durch Aufruf der Methode `interrupt()`

Hilfsklasse zum Beenden

Eine mögliche Lösung zum Beenden von Threads besteht darin, eine Hilfsklasse mit einem Flag-Attribut `shouldStop` sowie korrespondierende Zugriffsmethoden zu implementieren und periodisch das Flag-Attribut abzufragen.

Betrachten wir zunächst eine denkbare, aber falsche Umsetzung, wie man sie in der Praxis und (leider auch) in manchem Buch findet:

```
// Achtung: Nicht Thread-sicher
public class BaseStoppableThread extends Thread
{
    private boolean shouldStop = false;

    public void requestStop()
    {
        shouldStop = true;
    }

    public void shouldStop()
    {
        return shouldStop;
    }

    public void run()
    {
        while (!shouldStop())
        {
            // Kein Aufruf von requestStop() und
            // keine Schreibzugriffe auf shouldStop
        }
    }
}
```

In der Praxis wird das Flag häufig `stopped` genannt. Zudem heißen die Zugriffsmethoden auf das Flag etwa `setStopped(boolean)` und `isStopped()`. Dies entspricht zwar der Intention des Beendens, allerdings wird hier eher ein Stoppwunsch geäußert. Daher werden die Methoden von mir `requestStop()` und `shouldStop()` genannt.

Auf den ersten Blick ist im Listing – abgesehen von der ungeschickten Ableitung von der Utility-Klasse `Thread` (vgl. folgenden Hinweis »Ableitung von der Klasse Thread«) – kein Fehler zu erkennen. Wieso ist diese Umsetzung trotzdem problematisch? Nach Lektüre von Abschnitt 10.4 über das Java-Memory-Modell sind wir bereits etwas sensibilisiert: Die JVM darf zur Optimierung Reorderings durchführen, sofern die Happens-before-Ordnung eingehalten wird. Ohne diese Ordnung beachtet der Compiler bei der Optimierung keine Multithreading-Aspekte: Er kann beispielsweise wiederholte Lesezugriffe auf sich nicht ändernde Variablen zu vermeiden versuchen.

Innerhalb der `run()`-Methode finden wir keinen Aufruf von `requestStop()` oder einen sonstigen Schreibzugriff auf das Attribut `shouldStop`. Für Singlethreading ergibt sich daraus, dass sich das Attribut `shouldStop` in der Schleife nicht mehr ändert. Damit ist das Ergebnis der Bedingung konstant. Um den Methodenaufruf und die wiederholte Auswertung der Bedingung `!shouldStop()` einzusparen, kann der Sourcecode durch den Compiler und die JVM wie folgt optimiert und umgewandelt werden:

```
public void run()
{
    if (!shouldStop())
    {
        while (true)
        {
            // ...
```

Beim Einsatz von Multithreading und Reorderings ist zum korrekten Ablauf dieses Beispiels die Happens-before-Ordnung sicherzustellen. Das kann in diesem Fall entweder über die Deklaration des Attributs `shouldStop` als `volatile` oder einen synchronisierten Zugriff geschehen. Bei der nachfolgenden Korrektur vermeiden wir außerdem, von der Klasse `Thread` abzuleiten, und implementieren eine Lösung, die auf dem Interface `Runnable` basiert und die Happens-before-Ordnung sicherstellt, wodurch keine Reorderings und keine Optimierung der Schleifenabfrage erfolgen:

```
abstract class AbstractStoppableRunnable implements Runnable
{
    private volatile boolean shouldStop = false;

    public void requestStop()
    {
        shouldStop = true;
    }

    public boolean shouldStop()
    {
        return shouldStop;
    }

    public void run()
    {
        while (!shouldStop())
        {
            // ...
        }
    }
}
```

Hinweis: Ableitung von der Klasse `Thread`

Ableitungen von Utility-Klassen sind in der Regel ungünstig. ***Leider sieht man aber genau dies häufig beim Einsatz der Klasse `Thread`.*** Schauen wir uns an, wieso es zu Problemen kommen kann. Nehmen wir dazu an, eine Klasse `AbstractStoppableThread` sei durch Vererbung realisiert und die restliche Realisierung entspräche der zuvor vorgestellten Klasse `AbstractStoppableRunnable`:

```
abstract class AbstractStoppableThread extends Thread
{
    private volatile boolean shouldStop = false;

    // ...
}
```

Durch diese Implementierung wird das Substitutionsprinzip und demzufolge auch die »is-a«-Eigenschaft (vgl. Kapitel 3) verletzt. Der Grund ist folgender: Mit dieser von `Thread` abgeleiteten Klasse `AbstractStoppableThread` können keine Implementierungen von `Runnable` ausgeführt werden. Es entsteht eine Inkompatibilität: Die `run()`-Methode der Basisklasse `Thread` wird durch eine eigene Implementierung überschrieben, die keine Ausführung von `Runnable`s erlaubt. Die obige Umsetzung ist somit OO-technisch unsauber, weil sich die eigene Klasse nicht so verwenden lässt wie die Klasse `Thread`.

Beenden mit `interrupt()`

Statt einen Thread durch die Verwendung eigener Mechanismen, etwa den Einsatz eines Stop-Flags, zu beenden, kann man über die Methode `interrupt()` der Klasse `Thread` die Beendigung der Ausführung eines anderen Threads anregen. Wie bereits bekannt, ist ein Aufruf der Methode `interrupt()` jedoch nur als Aufforderung zu sehen, sie besitzt keine unterbrechende Wirkung: Es wird lediglich ein Flag gesetzt. Die Bearbeitung und Auswertung dieses Flags mithilfe der Methode `isInterrupted()` ist Aufgabe des Entwicklers der `run()`-Methode und kann in etwa so geschehen:

```
public void run()
{
    while (!Thread.currentThread().isInterrupted())
    {
        // ...
    }
}
```

Achtung: `Thread.interrupted()` vs. `isInterrupted()`

Bei der Abfrage des Flags muss man etwas Vorsicht walten lassen. Die Klasse `Thread` bietet die statische Methode `interrupted()`, die zwar das Flag prüft, dieses allerdings auch zurücksetzt. In der Regel soll eine Prüfung ohne Seiteneffekt erfolgen. Dazu ist immer die Objektmethode `isInterrupted()` zu verwenden.

Fazit

Beide Lösungen zum Beenden von Threads sind funktional nahezu gleichwertig. Damit länger andauernde Aktionen in der `run()`-Methode tatsächlich abgebrochen werden können, müssen dort gegebenenfalls weitere Abfragen und Aufrufe von `shouldStop()` bzw. `isInterrupted()` erfolgen, um auch zwischen den ausgeführten Arbeitsschritten eine Reaktion auf Stoppwünsche zu ermöglichen. Das haben wir bereits in Abschnitt 10.1.4 diskutiert.

10.6 Weiterführende Literatur

Da die Themen Multithreading und Nebenläufigkeit recht komplex und vielschichtig sind, empfiehlt es sich, weitere Quellen zu konsultieren. Weiterführende Informationen finden Sie in folgenden Büchern:

- »**Parallele und verteilte Anwendungen in Java**« von Rainer Oechsle [58]
  Dieses Buch gibt einen sehr lesbaren und verständlichen Einstieg in das Thema Multithreading mit Java 5.
- »**Java Threads**« von Scott Oaks und Henry Wong [57]
  Ebenso wie das Buch von Rainer Oechsle bietet dieses Buch einen sehr lesbaren und verständlichen Einstieg in das Thema Multithreading mit Java 5. Durch den Fokus auf Threads erklärt es einige Dinge noch gründlicher.
- »**Concurrent Programming in Java**« von Doug Lea [48]
  Dieser Klassiker, der bereits einige Jahre auf dem Buckel hat, stammt vom Entwickler der Concurrency Utilities selbst. Viele der im Buch beschriebenen Klassen haben später Einzug in die Java-Klassenbibliothek gehalten. Es ist allerdings keine leichte Lektüre.
- »**Taming Java Threads**« von Allen Holub [30]
  Dieses Buch gibt einen fundierten Einstieg und beschreibt sehr genau die Details von Multithreading inklusive diverser Fallstricke. Es werden auch fortgeschrittenere Themen wie Multithreading und Swing, Thread-Pools und blockierende Warteschlangen behandelt. Da dieses Buch noch auf Java 1.4 basiert, werden hier ähnliche Ideen und Klassen entwickelt wie bei Doug Lea.

11 Modern Concurrency

Die bis einschließlich JDK 1.4 vorhandenen Sprachmittel, wie die Schlüsselwörter `volatile` und `synchronized` sowie die Methoden `wait()`, `notify()` und `notify-All()`, erlauben es zwar, ein Programm auf mehrere Threads aufzuteilen und zu synchronisieren, allerdings lassen sich viele Aufgabenstellungen so nur relativ umständlich realisieren. Außerdem möchte kaum jemand heute noch auf Low-Level-Konzepten mit `Thread` und `Runnable` direkt programmieren. Im vorherigen Kapitel wurde deutlich, welche Fallstricke lauern und dass es anspruchsvoll ist, das korrekt umzusetzen. Zur besseren Auslastung von Multicore-Prozessoren ist es für Concurrency erstrebenswert, sich von den feingranularen und recht primitiven Ansätzen mit Threads zu lösen. Vielmehr bieten sich High-Level-Frameworks an. Doug Lea hat hier einen enormen Beitrag geleistet: Nicht nur das Executor- und das Fork-Join-Framework, sondern auch die von ihren Möglichkeiten her großartige Klasse `CompletableFuture<T>` sowie die mit JDK 9 bereitgestellten Reactive Streams gehen auf sein Konto.

Das gilt ebenfalls für die mit JDK 5 eingeführten Concurrency Utilities. Diese erleichtern die Entwicklung von Multithreading-Anwendungen, da sie für viele zuvor nur mühsam zu lösende Probleme bereits fertige Bausteine bereitstellen. Durch deren Einsatz wird Komplexität aus der Anwendung in das Framework verlagert, was die Lesbarkeit und Verständlichkeit des Applikationscodes deutlich erhöht: Für Locks und atomare Variablen haben wir bereits gesehen, dass sich damit Ideen und Konzepte klarer ausdrücken lassen.

Einführend werden wir in Abschnitt 11.1 die Concurrent Collections anschauen und ihre Vorteile in Multithreading-Umgebungen gegenüber normalen Collections beleuchten. Danach folgt in Abschnitt 11.2 eine Einführung in das Executor-Framework, das die Ausführung von Tasks mithilfe von Thread-Pools vereinfacht. Im Anschluss daran schauen wir in Abschnitt 11.3 kurz auf das Fork-Join-Framework, mit dessen Hilfe man rekursive Berechnungen nach dem Teile-und-herrsche-Prinzip parallel abarbeiten kann. Darüber hinaus wird die mit JDK 8 eingeführte Klasse `CompletableFuture<T>` in Abschnitt 11.4 vorgestellt. Damit lassen sich Verarbeitungsabläufe sehr elegant beschreiben und parallelisieren. Diese Klasse sollte jedem ambitionierten Java-Entwickler geläufig sein. Den Abschluss des Kapitels bildet eine Beschreibung zu Reactive Streams in Abschnitt 11.5. Reactive Streams ermöglichen Reactive Programming und die nicht blockierende Verarbeitung von Daten.

11.1 Concurrent Collections

Die Concurrent Collections enthalten auf Parallelität spezialisierte Implementierungen der Interfaces List<E>, Set<E>, Map<K,V>, Queue<E> und Deque<E>, wodurch ein einfaches Ersetzen »normaler« Datenstrukturen durch deren Concurrent-Pendants wesentlich erleichtert wird, falls Nebenläufigkeit erforderlich ist.

Wenn mehrere Threads parallel verändernd auf eine Datenstruktur zugreifen, kann es leicht zu Inkonsistenzen kommen. Dies gilt insbesondere, da die meisten Containerklassen des Collections-Frameworks nicht Thread-sicher sind. Die Zusammenarbeit und Kommunikation von Threads haben wir recht ausführlich in den Abschnitten 10.2 und 10.3 kennengelernt. Wir rekapitulieren in diesem Abschnitt zunächst kurz typische Probleme in Multithreading-Anwendungen beim Einsatz »normaler« Collections. Anschließend werden wir zur Lösung dieser Probleme die Concurrent Collections nutzen. Diese können bei Bedarf nach Parallelität stellvertretend für die »Original«-Container eingesetzt werden. Die Grundlage für diese Austauschbarkeit bilden gemeinsame Interfaces (z. B. List<E>, Set<E> und Map<K,V>).

Wenn tatsächlich Nebenläufigkeit und viele parallele Zugriffe unterstützt werden müssen, können die Concurrent Collections ihre Stärken ausspielen. In Singlethreading-Umgebungen oder bei sehr wenigen konkurrierenden Zugriffen ist ihr Einsatz gut abzuwägen, da in den Containern selbst einiges an Aufwand betrieben wird, um sowohl Thread-Sicherheit als auch Parallelität zu gewährleisten.

11.1.1 Thread-Sicherheit und Parallelität mit »normalen« Collections

Die synchronized-Wrapper des Collections-Frameworks ermöglichen einen Thread-sicheren Zugriff durch Synchronisierung aller Methoden (vgl. Abschnitt 6.3.2). Betrachten wir folgende synchronisierte Map als Ausgangsbasis unserer Diskussion:

```
final Map<String, Person> syncMap = Collections.synchronizedMap(personsMap);
```

Eine derartige Ummantelung kann zu einer (stark) eingeschränkten Parallelität führen, weil die Synchronisierung die Zugriffe serialisiert. Zudem schützt die Ummantelung nicht vor möglichen Inkonsistenzen, wenn man mehrere für sich Thread-sichere Methoden hintereinander aufruft. Das habe ich bereits bei der Beschreibung der synchronized-Wrapper in Abschnitt 6.3.2 diskutiert und greife es hier kurz auf, um die Arbeitsweise und Vorteile der Concurrent Collections zu motivieren.

Datenzugriff Jeder einzelne Methodenaufruf einer synchronisierten Collection ist für sich gesehen Thread-sicher. Für eine benutzende Komponente sind solche feingranularen Sperren aber häufig uninteressant. Vielmehr sollen Operationen mit mehreren Schritten atomar und Thread-sicher ausgeführt werden. Solche Mehrschrittoperationen sind etwa »testAndGet()« oder »putIfAbsent()«, die zunächst prüfen, ob

ein gewisses Element enthalten ist, und nur dann einen Zugriff bzw. eine Modifikation ausführen. Funktional würde man Folgendes schreiben:

```
// ACHTUNG: nicht Thread-sicher
public Person putIfAbsent(final String key, final Person newPerson)
{
    if (!syncMap.containsKey(key))
    {
        return syncMap.put(key, newPerson);
    }
    return syncMap.get(key);
}
```

Abschnitt 10.2.2 beschreibt, dass für sich Thread-sichere Methoden in ihrer Kombination nicht Thread-sicher sind. Abhilfe schafft eine Synchronisierung der gesamten Aktionen. Mithilfe eines Synchronisationsobjekts implementiert man eine korrekt synchronisierte Version der `putIfAbsent(String, Person)`-Methode etwa wie folgt:

```
public Person putIfAbsent(final String key, final Person newPerson)
{
    synchronized (syncMap)  // Kritischer Bereich für Mehrschrittoperationen
    {
        if (!syncMap.containsKey(key))
        {
            return syncMap.put(key, newPerson);
        }
        return syncMap.get(key);
    }
}
```

Durch eine solche Umsetzung wird allerdings keine gute Nebenläufigkeit erreicht, da andere Zugriffe blockiert werden.

Iteration Eine weitere sehr gebräuchliche Mehrschrittoperation ist das Iterieren über eine Datenstruktur. Die Iteratoren der »normalen« Container sind »fail-fast«, d. h., sie prüfen sehr streng, ob möglicherweise während einer Iteration eine Veränderung an der Datenstruktur vorgenommen wurde, und reagieren darauf mit einer `ConcurrentModificationException` (vgl. Abschnitt 6.1.4).

Zur Thread-sicheren Iteration über die Einträge einer synchronisierten Liste ist dieser Vorgang zusätzlich durch einen `synchronized`-Block zu schützen:

```
// Blockiert andere Zugriffe auf syncPersons während der Iteration
synchronized (syncPersons)
{
    for (final Person person : syncPersons)
    {
        person.doSomething();
    }
}
```

Auf diese Weise ist zwar eine korrekt synchronisierte Iteration möglich, allerdings auf Kosten von Nebenläufigkeit: Durch den `synchronized`-Block wird ein kritischer Bereich definiert und ***die Liste `syncPersons` ist während des Iterierens für mögliche andere Zugriffe blockiert. Nebenläufigkeit wird dadurch stark behindert***.

11.1.2 Parallelität mit den Concurrent Collections

Wenn mehrere Threads gemeinsam lesend auf einer Collection arbeiten, muss nicht immer die gesamte Datenstruktur gesperrt und dadurch der Zugriff für andere Threads blockiert werden. Mehr noch: Lesezugriffe sollten sich gegenseitig nicht beeinflussen und nicht blockieren, wohingegen Lese- und Schreibzugriffe aufeinander abgestimmt werden sollten. Dazu existieren verschiedene Verfahren. In den Concurrent Collections werden die zwei aufgelisteten Techniken eingesetzt, um neben Thread-Sicherheit auch für mehr Parallelität als bei den `synchronized`-Wrappern zu sorgen:

- **Kopieren beim Schreiben** – Die dahinterliegende Idee ist, vor jedem Schreibzugriff die Datenstruktur zu kopieren und dann das Element hinzuzufügen, zu verändern oder zu entfernen. Andere Threads können dadurch lesend zugreifen, ohne durch die Modifikation gestört zu werden. Diese Variante realisieren die Klassen `CopyOnWriteArrayList<E>` und `CopyOnWriteArraySet<E>` für Listen sowie Sets.
- **Lock-Striping / Lock-Splitting** – Hierbei werden verschiedene Teile eines Objekts oder einer Datenstruktur mithilfe mehrerer Locks geschützt. Dadurch sinkt die Wahrscheinlichkeit für gleichzeitige Zugriffe auf jeden einzelnen Lock und Threads werden seltener durch das Warten auf Locks am Weiterarbeiten gehindert. Als Folge davon steigt die Möglichkeit zur Parallelisierung. Zur Realisierung der parallelen `ConcurrentHashMap<K,V>` wird genau dieses Verfahren angewendet, das auf die spezielle Arbeitsweise von Hashcontainern mit Buckets abgestimmt ist und statt der gesamten `HashMap<K,V>` nur jeweils Teile davon schützt.

Datenzugriff in den Klassen `CopyOnWriteArrayList<E>/-Set<E>` Die Implementierung der Klasse `CopyOnWriteArraySet<E>` nutzt die Klasse `CopyOnWriteArrayList<E>`. Diese wiederum verwendet ein Array zur Speicherung von Elementen und passt dieses bei Schreibvorgängen ähnlich der in Abschnitt 6.1.2 beschriebenen Größenänderung von Arrays an. Jede Änderung der Daten erzeugt eine Kopie des zugrunde liegenden Arrays. Dadurch können Threads ungestört parallel lesen, sehen eventuell jedoch nicht die aktuellen Änderungen durch andere Threads.

Allerdings sollte man folgende zwei Dinge beim Einsatz bedenken: Für kleinere Datenmengen (< 1.000 Elemente) ist das Kopieren und die mehrfache Datenhaltung in der Regel vernachlässigbar. Der negative Einfluss steigt mit der Anzahl zu speichernder Elemente linear an. Aufgrund der gewählten Strategie des Kopierens bieten sich diese Datenstrukturen daher vor allem dann an, wenn deutlich mehr Lese- als Schreibzugriffe erfolgen und die zu speichernden Datenvolumina nicht zu groß sind.

Datenzugriff in der Klasse `ConcurrentHashMap<K,V>` Betrachten wir die Klasse `ConcurrentHashMap<K,V>`, die eine Realisierung einer `Map<K,V>` ist und mehreren Threads paralleles Lesen und Schreiben ermöglicht. Die Realisierung garantiert, dass sich Lesezugriffe nicht gegenseitig blockieren. Für Schreibzugriffe kann man zum Konstruktionszeitpunkt bestimmen, wie viel Nebenläufigkeit unterstützt werden soll, indem man die Anzahl der Schreibsperren festlegt.

Streng genommen ist die Klasse `ConcurrentHashMap<K,V>` lediglich ein Ersatz für die Klasse `Hashtable<K,V>` und nicht für die Klasse `HashMap<K,V>`, wie es der Name andeutet. Das liegt daran, dass die Klasse `ConcurrentHashMap<K,V>` im Gegensatz zur Klasse `HashMap<K,V>` keine `null`-Werte für Schlüssel und Werte unterstützt. Der Wert `null` drückt stattdessen aus, dass ein gesuchter Eintrag fehlt. In vielen Fällen wird diese Unterstützung für `null` nicht benötigt. Dann kann man die Klasse `ConcurrentHashMap<K,V>` problemlos anstelle einer `HashMap<K,V>` verwenden.

Das Interface `ConcurrentMap<K,V>` deklariert vier Mehrschrittoperationen. Das Besondere daran ist, dass diese von den konkreten Realisierungen `ConcurrentHashMap<K,V>` und `ConcurrentSkipListMap<K,V>` atomar ausgeführt werden müssen. Die folgende Aufzählung nennt die Methoden und zeigt zur Verdeutlichung der Funktionalität eine schematische Pseudoimplementierung. Die tatsächliche Implementierung ist wesentlich komplizierter, da sie Parallelität ohne Blockierung gewährleistet.

- `V putIfAbsent(K key, V value)` – Erzeugt einen neuen Eintrag zu diesem Schlüssel und dem übergebenen Wert, sofern kein Eintrag zu dem Schlüssel existiert. Liefert den zuvor gespeicherten Wert zurück, falls bereits ein Eintrag zu dem Schlüssel vorhanden ist, oder `null`, wenn dies nicht der Fall ist.

```
if (!map.containsKey(key))
{
    return map.put(key, value);
}
return map.get(key);
```

- `boolean remove(Object key, Object value)` – Entfernt den Eintrag zu diesem Schlüssel, falls der gespeicherte Wert mit dem übergebenen Wert übereinstimmt. Liefert `true`, falls es einen solchen Eintrag gab, ansonsten `false`.

```
if (map.containsKey(key))
{
    if (map.get(key).equals(value))
    {
        map.remove(key);
        return true;
    }
}
return false;
```

Tipp: Wieso der Typ `Object` statt generischer Typparameter?

Vielleicht fragen Sie sich, wieso hier `Object` statt der generischen Typen `K` und `V` genutzt wird. Das liegt zum einen daran, dass die Methoden in den Concurrent Collections diejenigen aus den Originalcontainerklassen überschreiben und somit die gleichen Typparameter besitzen müssen. Im Collections-Framework sind die `remove()`-Methoden mit `Object` definiert, weil die Prüfungen mit `equals()` auch mit `Object` arbeiten und man somit mehr Flexibilität beim Löschen hat.

- `V replace(K key, V value)` – Ersetzt zu diesem Schlüssel den gespeicherten durch den übergebenen Wert. Liefert den zuvor mit dem Schlüssel assoziierten Wert zurück oder `null`, wenn es keinen Eintrag zu dem Schlüssel gab:

```
if (map.containsKey(key))
{
    return map.put(key, value);
}
return null;
```

- `boolean replace(K key, V oldValue, V newValue)` – Ersetzt den Eintrag zu diesem Schlüssel mit dem neuen Wert `newValue`, falls der gespeicherte Wert mit dem alten Wert `oldValue` übereinstimmt. Liefert `true`, falls dem so ist, ansonsten `false`.

```
if (map.containsKey(key))
{
    if (map.get(key).equals(oldValue))
    {
        map.put(key, newValue);
        return true;
    }
}
return false;
```

Die gezeigten Aufrufe von `map.containsKey(key)` sind notwendig, um zu verhindern, dass bei einem fehlenden Eintrag eine `NullPointerException` ausgelöst wird.

Iteration Um eine parallele Verarbeitung zu unterstützen, wurde das Verhalten der Iteratoren so angepasst, dass diese weder `ConcurrentModificationExceptions` auslösen noch eine Synchronisierung benötigen. Allerdings sind die Iteratoren auch nur »schwach« konsistent oder »weakly consistent«, d. h., eine Iteration liefert nicht immer die aktuelle Zusammensetzung der Datenstruktur, aber zumindest den Zustand zum Zeitpunkt der Erzeugung des Iterators. Nachfolgende Änderungen an der Zusammensetzung können bei der Iteration berücksichtigt werden, müssen es aber nicht. Das erlaubt es, Iterationsvorgänge parallel zu Veränderungen durchzuführen.

11.1.3 Blockierende Warteschlangen und das Interface `BlockingQueue<E>`

Zum Austausch von Daten zwischen Programmkomponenten können Implementierungen des Interface `Queue<E>` (vgl. Abschnitt 6.6) verwendet werden. Das Interface `BlockingQueue<E>` erweitert das Basisinterface `Queue<E>` um folgende Methoden:

- `boolean offer(E element, long time, TimeUnit unit)` – Fügt ein Element in die Queue ein, falls keine Größenbeschränkung existiert oder die Queue nicht voll ist. Ansonsten wartet der Aufruf blockierend maximal die angegebene Zeitspanne, bis eine andere Programmkomponente ein Element entfernt, sodass als Folge ein Einfügen möglich wird. Liefert `true`, wenn das Einfügen erfolgreich war, ansonsten `false`.
- `E poll(long time, TimeUnit unit)` – Gibt das erste Element zurück und entfernt es aus der Queue. Wenn die Queue leer ist, wird maximal die angegebene Zeitspanne auf das Einfügen eines Elements durch eine andere Programmkomponente gewartet und nach einem erfolglosen Warten `null` zurückgegeben.
- `void put(E element)` – Fügt ein Element in die Queue ein. Dieser Aufruf erfolgt blockierend. Das bedeutet, dass gewartet werden muss, wenn die Queue eine Größenbeschränkung besitzt und zum Zeitpunkt des Einfügens voll ist.
- `E take()` – Gibt das erste Element zurück und entfernt es aus der Queue. Der Aufruf blockiert, solange die Queue leer ist.

Wie schon angedeutet, blockieren diese Methoden beim Schreiben in eine volle Queue bzw. beim Lesen aus einer leeren Queue.[1] Das erleichtert die Kommunikation zwischen Threads, da auf eine fehleranfällige Synchronisierung und Benachrichtigung über die Methoden `wait()` und `notify()` bzw. `notifyAll()` verzichtet werden kann.

Die durch das Interface `BlockingQueue<E>` beschriebene Schnittstelle wurde bereits in Abschnitt 10.3 zur Abstimmung von Producer und Consumer genutzt. Ähnliche Funktionalität wird durch folgende Klassen der Concurrent Collections realisiert:

- `ArrayBlockingQueue<E>` – Diese Realisierung bietet einen FIFO-Zugriff, besitzt eine Größenbeschränkung und verwendet ein Array zur Speicherung.
- `LinkedBlockingQueue<E>` – Diese Realisierung bietet einen FIFO-Zugriff und nutzt eine `LinkedList<E>`, wodurch keine Größenbeschränkung gegeben ist.
- `PriorityBlockingQueue<E>` – Diese Realisierung nutzt ein Sortierkriterium, um die Reihenfolge der Elemente innerhalb der Queue zu bestimmen. Elemente mit der höchsten Priorität stehen am Anfang und werden somit bei Lesezugriffen zuerst zurückgeliefert.
- `SynchronousQueue<E>` – Diese Queue ist ein Spezialfall einer größenbeschränkten Queue, allerdings mit der Größe 0. Zunächst klingt dies unsinnig, ist aber für solche Anwendungsfälle geeignet, die erfordern, dass zwei Threads unmittelbar

[1] Durch das Interface kann dies nur gefordert, nicht aber sichergestellt werden.

aufeinander warten. Bezogen auf das Producer-Consumer-Beispiel bedeutet dies, dass ein Producer erst sein Produkt »speichern« kann, wenn ein Consumer dieses direkt abholt. Andersherum gilt: Möchte ein Consumer Daten aus der Queue entnehmen, muss er warten, bis ein Producer Daten ablegt.[2]

- `DelayQueue<E>` – Bei dieser Art von Queue beschreiben zu speichernde Elemente ihre Sortierung, indem sie das Interface `java.util.concurrent.Delayed` implementieren. Das Interface `Delayed` erweitert `Comparable<Delayed>` und beschreibt somit eine Ordnung: In einer solchen Queue sind die Elemente nach ihrem »Verfallsdatum« geordnet. Dazu kann ein Ablaufzeitpunkt über die Methode `getDelay(java.util.concurrent.TimeUnit unit)` angegeben werden.

Beispiel

Entwickeln wir ein minimales Beispiel mit einer auf vier Einträge größenbeschränkten `ArrayBlockingQueue<Integer>`. Zunächst produzieren wir ein paar mehr als die dort vorgesehenen Einträge, wodurch der Producer nach Erreichen des Limits ein klein wenig warten muss, bis der Consumer die Daten abholt. Danach verlangsamt der Producer seine Arbeit und der Consumer muss dann immer 2 Sekunden warten, bis der nächste Wert produziert ist und abgeholt werden kann. Die Erleichterung im Vergleich zu den Beispielen aus dem vorherigen Kapitel besteht darin, dass einfacher auf die Queue zugegriffen werden kann, ohne dies speziell synchronisieren zu müssen:

```
public class BlockingQueueExample
{
    public static void main(final String[] args) throws Exception
    {
        final BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(4);

        new Thread(new Producer(queue)).start();
        final Thread consumerThread = new Thread(new Consumer(queue));
        consumerThread.start();

        Thread.sleep(7000);
        consumerThread.interrupt();
    }

    static class Producer implements Runnable
    {
        private final BlockingQueue<Integer> queue = null;

        public Producer(final BlockingQueue<Integer> queue)
        {
            this.queue = queue;
        }

        public void run()
        {
            // erste 4 gehen direkt, dann warten auf Consumer
            for (int i = 0; i < 8; i++)
            {
```

[2] Man spricht auch von Rendezvous, was Treffen in Französisch bedeutet. Für Multithreading meint dies, dass sich zwei Threads zum Datenaustausch »treffen«.

```
                try
                {
                    System.out.println("putting " + i);
                    queue.add(i);
                }
                catch (Exception e1)
                {
                    e1.printStackTrace();
                }
            }
            // Consumer muss immer 2 Sekunden warten, bis Erzeugung fertig
            try
            {
                queue.put(1111);
                Thread.sleep(2000);
                queue.put(2222);
                Thread.sleep(2000);
                queue.put(3333);
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }

    static class Consumer implements Runnable
    {
        private final BlockingQueue<Integer> queue;

        public Consumer(final BlockingQueue<Integer> queue)
        {
            this.queue = queue;
        }

        public void run()
        {
            while (!Thread.currentThread().isInterrupted())
            {
                try
                {
                    System.out.println("Consuming " + queue.take());
                    Thread.sleep(100);
                }
                catch (InterruptedException e)
                {
                    Thread.currentThread().interrupt();
                }
            }
        }
    }
}
```

Nutzt man im Producer `add()` statt `put()`, so kann man die Größenbeschränkung sichtbar machen: Bei Überschreiten der Größe wird eine Exception ausgelöst:

```
putting 0
putting 1
putting 2
putting 3
putting 4
java.lang.IllegalStateException: Queue full
```

11.2 Das Executor-Framework

Das Executor-Framework unterstützt bei der Ausführung asynchroner Aufgaben durch Bereitstellung von Thread-Pools und ermöglicht es, verschiedene Abarbeitungsstrategien zu verwenden sowie die Bearbeitung abzubrechen und Ergebnisse abzufragen.

Das Executor-Framework vereinfacht den Umgang mit Threads und die Verarbeitung von Ergebnissen asynchroner Aufgaben, sogenannter ***Tasks***. Es erfolgt eine Trennung der Beschreibung eines Tasks und dessen tatsächlicher Ausführung. Dabei wird von den Details abstrahiert, etwa wie und wann eine Aufgabe von welchem Thread abgearbeitet wird. Die Kernidee ist, Tasks an sogenannte Executoren zur (späteren) Ausführung zu übergeben.

11.2.1 Einführung

Ohne das Executor-Framework können Tasks (Aufgaben bzw. Arbeitsblöcke), die das Interface `Runnable` implementieren, mithilfe der Klasse `Thread` ausgeführt werden. Bekanntermaßen schreibt man dazu Folgendes:

```
new Thread(runnableTask).start();
```

Das Interface `Executor`

Statt explizit mit Threads zu arbeiten, kann man alternativ eine Realisierung des Interface `java.util.concurrent.Executor` nutzen, um einen Task auszuführen. Dadurch wird von den konkreten Aktionen »Erzeugen« und »Starten« abstrahiert. Das Interface `Executor` ist im JDK wie folgt definiert:

```
public interface Executor
{
    void execute(final Runnable runnableTask);
}
```

Damit lässt sich die obige Ausführung folgendermaßen schreiben:

```
executor.execute(runnableTask);
```

Sollen nur einige wenige Tasks abgearbeitet werden, so ergibt sich kaum ein Vorteil durch den Einsatz eines Executors gegenüber dem Einsatz von Threads. Für Multithreading-Anwendungen, die aus diversen (unabhängigen) Tasks bestehen, kann eine derartige Abstraktion allerdings sehr vorteilhaft sein, wie wir dies nun beleuchten.

Implementierung des Interface `Executor` Die zuvor gezeigte Abstraktion von Threads und Implementierungsdetails wirkt zunächst wenig spektakulär. Warum das Ganze dennoch hilfreich ist, werden wir sukzessive kennenlernen. Ein erster Vorteil besteht darin, dass das Interface `Executor` keine konkrete Art der Ausführung vorgibt.

Die Details werden durch die jeweilige `Executor`-Implementierung spezifiziert, beispielsweise wie viele Aufgaben parallel ausgeführt werden und wann bzw. in welcher Reihenfolge dies erfolgt.

Betrachten wir zunächst zwei extreme Arten der Ausführung: alle Tasks synchron hintereinander und alle Tasks vollständig asynchron. Für Ersteres könnte man eine Klasse `SynchronousExecutor` wie folgt implementieren:

```
public class SynchronousExecutor implements Executor
{
    public void execute(final Runnable runnable)
    {
        runnable.run();
    }
}
```

In der Regel sollen Tasks nicht synchron im aufrufenden Thread, sondern parallel dazu ausgeführt werden. Eine Klasse `AsyncExecutor`, die für jeden Task einen eigenen Thread startet, könnte folgendermaßen realisiert werden:

```
public class AsyncExecutor implements Executor
{
    public void execute(final Runnable runnableTask)
    {
        new Thread(runnable).start();
    }
}
```

Hinweis: `Executor` selbst implementieren?

Wie bereits gesehen, ist es zwar möglich, eigene Implementierungen des Interface `Executor` zu schreiben, normalerweise empfiehlt es sich jedoch, die vordefinierten Klassen des Executor-Frameworks zu verwenden, die wir im Verlauf dieses Abschnitts kennenlernen werden.

Motivation für Thread-Pools

Nehmen wir an, es seien viele (möglichst voneinander unabhängige Aufgaben zu erledigen (konsultieren Sie bitte den folgenden Praxistipp für Hintergrundinformationen dazu). Die zuvor vorgestellten Extreme bei der Umsetzung, d. h. die streng sequenzielle Ausführung aller Aufgaben durch einen einzigen Thread bzw. die maximale Parallelisierung durch Abspalten eines eigenen Threads pro Aufgabe, besitzen beide unterschiedliche Nachteile. Der erste Ansatz führt durch die Hintereinanderausführung zu langen Wartezeiten und schlechten Antwortzeiten. Es kommt zu einem geringeren Durchsatz. Der zweite Ansatz parallelisiert die Ausführung und erreicht auf diese Weise einen höherem Durchsatz, sodass man etwas naiv auf die Idee kommen könnte, einfach so viele Threads zu erzeugen, wie Aufgaben existieren, um dadurch die Parallelität der Abarbeitung zu maximieren und für gute Antwortzeiten zu sorgen. Allerdings skaliert dieses Vorgehen nur begrenzt: Zunächst einmal gibt es doch mitunter Abhängigkeiten

15.2.6 Arten von Modulen

Im Verlauf dieses Kapitels haben wir bereits einige Arten von Modulen kennengelernt, aber nicht explizit benannt. Das möchte ich an dieser Stelle nachholen:

- **Named Platform Modules** – Bekanntermaßen wurde im Rahmen von Project Jigsaw auch das JDK in Module unterteilt. Dort findet man rund 80 Module, die Sie sich mit dem Kommando `java --list-modules` auflisten lassen können. Diese speziellen Module des JDKs haben keinen Zugriff auf den Module-Path.
- **Named Application Modules** – Auch Named Application Modules haben wir schon in verschiedenen Beispielen selbst erstellt. Darunter werden Module verstanden, die Anwendungen oder Bibliotheken bündeln. Im Speziellen sind dies modulare JARs, also solche, die in ihrem JAR einen Moduldeskriptor in Form der Datei `module-info.class` enthalten. Diese können auf alle über den Module-Path erreichbaren Module zugreifen, nicht jedoch auf Klassen aus dem `CLASSPATH`.
- **Open Modules** – Wie die beiden ersten Module, allerdings geben Open Modules alle Packages für Reflection nach außen frei. Mit dem Schlüsselwort `opens` im Moduldeskriptor bzw. dem Kommandozeilenparameter `--add-opens` kann dies auch auf spezifische Packages eingeschränkt erfolgen. Für Details verweise ich Sie an mein Buch »Java - die Neuerungen in Version 9 bis 14: Modularisierung, Syntax- und API-Erweiterungen« [41].
- **Automatic Modules** – Für die Migration von Anwendungen ist es von Vorteil, dass man auch gewöhnliche JAR-Dateien, also ohne `module-info.class`, im Module-Path angeben kann. Diese JAR-Dateien werden zu sogenannten Automatic Modules: Dabei wird der JAR-Dateiname ohne Versionsnummer und Endung als Modulname genutzt. Named Application Modules können auf Automatic Modules per `requires` zugreifen. Insbesondere exportieren Automatic Modules alle ihre Packages und können auf sämtliche Module aus dem Module-Path sowie JARs aus dem `CLASSPATH` zugreifen.
- **Unnamed Modules** – Ergänzend zum Module-Path kann man sowohl beim Kompilieren als auch beim Programmstart einen `CLASSPATH` angeben. Alle dort vorhandenen Typen werden zu dem sogenannten Unnamed Module zusammengefasst. Enthält der `CLASSPATH` modulare JARs, so werden diese wie normale JARs ohne Modularisierung behandelt: Sie exportieren alle enthaltenen öffentlichen Typen. Somit können alle öffentlichen Typen aus einem Unnamed Module direkt aufeinander zugreifen, also ohne Sichtbarkeitsschutz. Zudem gilt Folgendes: Automatic Modules können auf das Unnamed Module zugreifen, aber Named Application Modules können dies nicht. Dieser Sachverhalt ist für die schrittweise Migration wichtig. So kann man ein benötigtes JAR als Automatic Module einbinden und dessen Abhängigkeiten aus dem `CLASSPATH` erfüllen.

Nach diesem einführenden Blick auf die verschiedenen Arten von Modulen schauen wir uns später in Abschnitt 15.5 mögliche Migrationsszenarien an und lernen dabei weitere Details zu den Modularten kennen.

15.3 Sichtbarkeiten und Zugriffsschutz

Die Themen Sichtbarkeit und Zugriffsschutz wurden zuvor nicht explizit thematisiert, sondern eher am Rande besprochen. Beiden Themen wollen wir uns nun widmen.

15.3.1 Sichtbarkeiten

Eingangs nannte ich als eines der Ziele bei der Modularisierung eine bessere Steuerung von Abhängigkeiten und Sichtbarkeiten. Die bisherigen Beispiele haben gezeigt, dass seit Java 9 die Sichtbarkeiten strenger als in Java 8 geprüft werden. Rekapitulieren wir zunächst den Status quo in Java 8, bevor ich die Erweiterungen von Java 9 vorstelle.

Sichtbarkeiten bis JDK 8

Bis Java 8 besaßen Typen eine der folgenden vier Sichtbarkeiten:

- `private` – Nur in der eigenen Klasse sichtbar
- *default* / *package private* (kein Schlüsselwort) – Nur im eigenen Package sichtbar
- `protected` – Wie *package private*, aber auch in abgeleiteten Klassen sichtbar
- `public` – Aus allen Packages zugreifbar

Tatsächlich bedeutet `public`, dass ein solcher Typ im `CLASSPATH` für alle anderen Typen zugänglich ist. Demnach kann man – zumindest für `public` – nicht wirklich von einer Sichtbarkeitssteuerung sprechen.

Sichtbarkeiten seit JDK 9

Mit der Einführung der Modularisierung lässt sich die Sichtbarkeit von Typen genauer spezifizieren. Relevant ist dies in der Regel nur für die als `public` definierten Typen. Diese untergliedern sich seit JDK 9 wie folgt:

- **Global** – `public` für alle Module – Wenn das entsprechende Package einer `public` definierten Klasse mit `exports` zum Zugriff freigegeben wurde, ist diese von allen anderen Modulen zugreifbar, die auf dieses Modul per `requires` verweisen.
- **Eingeschränkt** auch ***Qualified Export*** genannt – `public` für angegebene Module – Mithilfe von `exports to` kann eine Liste von Modulen spezifiziert werden, die Zugriff erhalten sollen. Das erfordert zudem, dass andere Module auf dieses Modul per `requires` verweisen.
- **Modul intern** – `public` im Modul selbst – Sofern Klassen in Packages liegen, die nicht in `exports` aufgeführt werden, sind diese Klassen nur aus den Packages des eigenen Moduls zugreifbar.

Den ersten und dritten Fall haben wir schon in verschiedenen Beispielen kennengelernt: Auf eine `public` definierte Klasse konnte nicht von anderen Modulen aus zugegriffen

werden. Dies war erst möglich, nachdem mit `requires` die Abhängigkeit und damit der Zugriffswunsch sowie mit `exports` die öffentliche Bereitstellung beschrieben wurden. Der zweite Fall ist ein Spezialfall des ersten. Allgemein geht es darum, dass man das Exportieren von Klassen auf eine vordefinierte Menge an Modulen beschränken kann. Das benötigt man z. B., wenn man innerhalb von Modulen eines Frameworks Zugriff erlauben möchte, diese Klassen aber von Nutzern des Frameworks nicht zugreifbar sein sollen. Auf diese Weise wurden im JDK spezielle, nicht für das öffentliche API vorgesehene Funktionalitäten nur intern bereitgestellt.

Ein Bild sagt mehr als tausend Worte. Somit sollte sich Ihr Verständnis nach einem Blick auf die folgende Abbildung 15-12 deutlich verbessern.

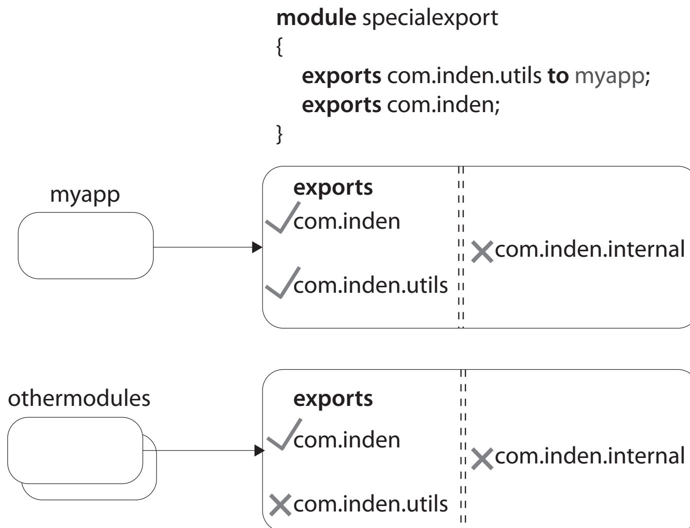

***Abbildung 15-12** Auswirkungen der drei Sichtbarkeiten bei* `public`

Einsatz im JDK Zuvor hatte ich angedeutet, dass es manchmal wünschenswert ist, bestimmte Typen bzw. Packages dediziert lediglich für spezielle Module zugreifbar zu machen. Im JDK selbst gilt dies auch für das Package `sun.reflect`. Dieses soll nur für diejenigen Module des JDKs verfügbar sein, die explizit in `exports` hinter dem Schlüsselwort `to` aufgeführt sind:

```
module java.base
{
     exports sun.reflect to java.corba,
                            java.logging,
                            java.sql,
                            java.sql.rowset,
                            jdk.scripting.nashorn;
}
```

15.3.2 Zugriffsschutz und Reflection

Bei den Sichtbarkeiten gilt, dass Packages explizit mit `exports` nach außen veröffentlicht und mit `requires` Module referenziert werden müssen. Nachfolgend schauen wir uns zur Vertiefung und zur Ergänzung das Thema Zugriffsschutz und Reflection an. Für das Beispiel sei das Verzeichnis wie folgt aufgebaut:

```
ch15_3_2_accessibility
`-- src
    |-- timeclient
    |   |-- com
    |   |   `-- client
    |   |       `-- CurrentTimeExample.java
    |   `-- module-info.java
    `-- timeserver
        |-- com
        |   `-- server
        |       |-- TimeInfo.java
        |       `-- internal
        |           `-- InternalUtil.java
        `-- module-info.java
```

Zugriffsschutz für interne Packages

Betrachten wir das Thema Zugriffsschutz: In der obigen Applikation soll im Modul `timeserver` die Klasse `InternalUtil` aus dem Package `com.server.internal` nach außen nicht sichtbar sein, sondern lediglich das Package `com.server`. Deswegen wird das interne Package in `exports` nicht aufgeführt:

```
module services
{
    exports com.server;
}
```

Zugriffsschutz und Reflection

Bis einschließlich Java 8 lässt sich mit Reflection so ziemlich jedes Sicherheitsmerkmal von Java umgehen. Beispielsweise konnte man sich mit `setAccessible(true)` selbst auf private Attribute und Methoden Zugriff verschaffen. Es war sogar möglich, wenn auch sicher nicht ratsam, als `final` definierte Attribute nachträglich noch mit einem anderen Wert zu versehen. Reflection war also ein wenig wie ein Zauberstab, mit dem man viel Magie betreiben konnte, insbesondere weil sich die sonst üblichen Regeln außer Kraft setzen lassen.

Deswegen sollten wir uns fragen, welche Auswirkungen die Modularisierung auf die Zugriffsmöglichkeiten per Reflection hat. Also konkret, ob sich mit Reflection auch die durch die Modularisierung eingeschränkten Zugriffe umgehen lassen.

Die Klasse `CurrentTimeExample` ändern wir so ab, dass dort mit Reflection ein Zugriff auf die Klasse `InternalUtil` und deren Methode `getCurrentTime()` erfolgt:

```
public class CurrentTimeExample
{
   public static void main(final String[] args) throws Exception
   {
      System.out.println("Trying to access getCurrentTime() via reflection");

      final ClassLoader classLoader = CurrentTimeExample.class.getClassLoader();

      final Class<?> internalClass = classLoader.
            loadClass("com.server.internal.InternalUtil");

      final Method method = internalClass.getMethod("getCurrentTime");
      final Object result = method.invoke(null, new Object[0]);

      System.out.println("getCurrentTime() returned: " + result);
   }
}
```

Das Programm CURRENTTIMEEXAMPLE lässt sich mit folgenden Kommandos kompilieren und starten:

```
javac -d build --module-source-path src $(find src -name '*.java')
java -p build -m timeclient/com.client.CurrentTimeExample
```

Dadurch kommt es in etwa zur folgenden Konsolenausgabe:

```
Trying to access getCurrentTime() via reflection
Exception in thread "main" java.lang.IllegalAccessException: class com.client.
      CurrentTimeExample (in module timeclient)
      cannot access class com.server.internal.InternalUtil (in module timeserver)
       because module timeserver does not export com.server.internal to
      module timeclient
```

Wir sehen, dass trotz des erlaubten Zugriffs auf die Methode `getCurrentTime()` per `getMethod()` und des zurückgelieferten validen `Method`-Objekts ein Aufruf per `invoke()` durch eine `IllegalAccessException` verhindert wird.

Hinweis: Zugriffsschutz mit `--add-exports` aushebeln

Mitunter benötigt man Zugriff auf Interna. In begründeten Fällen kann durch den Kommandozeilenparameter `--add-exports` eine explizite Freigabe erfolgen – das sollte aber wirklich nur in Ausnahmefällen eingesetzt werden. Durch Folgendes ist dann auch der Aufruf per Reflection erlaubt und es wird die Methode `getCurrentTime()` ausgeführt:

```
java -p build --add-exports timeserver/com.server.internal=timeclient
     -m timeclient/com.client.CurrentTimeExample
```

15.4 Empfehlenswertes Verzeichnislayout für Module

Bereits bei der Einführung zur Modularisierung habe ich in Abschnitt 15.2.1 angedeutet, dass das von Oracle für Module empfohlene Verzeichnislayout für die meisten eigenen Applikationen eher ungünstig ist.

Zur Erinnerung sei hier nochmals das Verzeichnislayout dargestellt: Unterhalb eines gemeinsamen `src`-Ordners sind die jeweiligen Module mit gleichnamigem Unterverzeichnis beheimatet – hier exemplarisch für die Module `personprocessor` und `utilities` gezeigt:

```
person_processor_application
`-- src
    |-- personprocessor
    |   |-- com
    |   |   |-- domain
    |   |   |   `-- Person.java
    |   |   `-- application
    |   |       `-- PersonProcessingApplication.java
    |   `-- module-info.java
    `-- utilities
        |-- com
        |   `-- utils
        |       `-- SomeUtil.java
        `-- module-info.java
```

Während dieses Verzeichnislayout für ganz spezifische Anwendungsfälle, etwa zum Kompilieren des JDKs, hilfreich ist, empfiehlt sich für eigene Module aus verschiedenen Gründen ein Verzeichnislayout, das sich an dem Standard von Maven und Gradle orientiert. Diese Thematik schauen wir uns nun genauer an.

Schwachstellen des Oracle-Verzeichnislayouts

Warum ist dieses Oracle-Verzeichnislayout nicht immer optimal? Das hat unter anderem folgende Gründe:

- Durch das gemeinsame Source-Verzeichnis mit jeweils Unterverzeichnissen pro Modul gestaltet es sich recht schwierig, Module unabhängig voneinander zu entwickeln, zu kompilieren und in JARs zu verpacken.
- Beim Oracle-Verzeichnislayout lässt sich durch das Unterverzeichnis pro Modul kein normaler Maven- oder Gradle-Build nutzen, da die Sourcen dann nicht wie standardmäßig erwartet in `src/main/java` liegen. Dadurch müssen die Source- und Test-Verzeichnisse für jedes Projekt bzw. Modul angepasst werden. Ähnliches gilt für IDEs. Insgesamt wird das Handling dadurch unkomfortabel.
- Ebenfalls erschwert sich die Zuordnung der Abhängigkeiten von Fremdbibliotheken zu den entsprechenden Modulen.

Für die Praxis empfohlenes Verzeichnislayout

In diversen kleineren eigenen Projekten kam ich im Laufe der letzten Jahre zur Erkenntnis, dass es sich anbietet, für jedes Modul ein separates Verzeichnis mit eigenem Source-Verzeichnis zu nutzen. IntelliJ, Maven und Gradle erlauben dazu Multi-Module-Projekte, die ein übergeordnetes Hauptverzeichnis vorsehen. In Eclipse muss man dagegen mehrere eigenständige Projekte nutzen, die aber idealerweise auf einem Multi-Module Build von Maven basieren.

Nachfolgend gezeigtes Verzeichnislayout verdeutlicht das zuvor Gesagte: Module sind in eigenen Verzeichnissen untergebracht, hier für die Module `personprocessor` und `utilities` gezeigt. Darunter existiert jeweils ein Source-Verzeichnis pro Modul. Zudem wird statt `src` der Standard `src/main/java` genutzt:

```
person_processor_application
|-- personprocessor
|   `-- src
|       `-- main
|           `-- java
|               |-- com
|               |   |-- domain
|               |   |   `-- Person.java
|               |   `-- application
|               |       `-- PersonProcessingApplication.java
|               `-- module-info.java
`-- utilities
    `-- src
        `-- main
            `-- java
                |-- com
                |   `-- utils
                |       `-- SomeUtil.java
                `-- module-info.java
```

Empfohlenes Verzeichnislayout für die nachfolgenden Beispiele

In diesem Kapitel habe ich in den Beispielen bislang ein Verzeichnislayout genutzt, das sich an die Vorgaben von Oracle hält.

Oben habe ich auf dessen Schwachstellen hingewiesen und motiviert, wie eine Alternative dazu aussehen kann. Diese werde ich für die nachfolgenden Beispiele verwenden. Daraus ergeben sich die Vorteile, dass sich dieses Verzeichnislayout einfacher und ohne Konfigurationsaufwand mit Build-Tools und IDEs verarbeiten lässt. Zudem wird dadurch die Unabhängigkeit von Modulen gefördert.

15.5 Kompatibilität und Migration

Bislang haben wir Module selbst definiert oder auf Module des JDKs zugegriffen. Es gibt dabei jedoch zwei Punkte zu bedenken:

1. **Rückwärtskompatibilität** – Applikationen, die ohne das Modulsystem erstellt wurden, sollten weiterhin lauffähig sein, um einen möglichst reibungslosen Übergang von Java 8 auf 9, 11 oder gar 15 zu ermöglichen.
2. **Einbinden von Fremdbibliotheken** – Jede etwas größere Java-Applikation verweist immer auch auf Fremdbibliotheken, die eingebunden werden müssen, die aber vermutlich noch nicht modularisiert wurden.

Den ersten Punkt behandeln wir direkt im Anschluss. Auf das Einbinden von Fremdbibliotheken gehe ich dann in verschiedenen Abschnitten ein und betrachte mehrere Migrationsszenarien. Dazu schauen wir uns die bereits kurz vorgestellten Konzepte Unnamed Module und Automatic Modules genauer an.

15.5.1 Kompatibilitätsmodus

Früher wurde bei jeder neuen Java-Version dem Erhalten der Rückwärtskompatibilität große Aufmerksamkeit geschenkt. Die Modularisierung ist erstmalig ein Feature, das zu Inkompatibilitäten führt. Zur schrittweisen Umstellung und zur Rückwärtskompatibilität bietet das Modulsystem jedoch einen ***Kompatibilitätsmodus***, bei dem Applikationen wie bisher aus JARs zusammengesetzt sein können und trotzdem die Module des JDKs nutzbar sind.

Wie bereits beschrieben, werden Typen durch das Modulsystem über den Module-Path aufgelöst. Doch was passiert, wenn dies nicht erfolgreich ist? Dann kommt es zu einem Fehler beim Applikationsstart. Jedoch erlaubt es das Modulsystem, ergänzend zum Module-Path einen `CLASSPATH` anzugeben, in dem die JVM zusätzlich nach Typen suchen kann. Durch die reine Angabe von Abhängigkeiten im `CLASSPATH` lassen sich Applikationen exakt so starten wie zuvor mit JDK 8 auch.

Wie schon aus den bisherigen Beispielen bekannt, finden die Aktionen diesmal im Verzeichnis namens `ch15_5_1_migration_compatibility_mode` statt – nachfolgend markiere ich das jeweilige Verzeichnis fett und verzichte in der Regel auf eine explizite Nennung im Text.

Beispiel: Applikation nur im `CLASSPATH`

Oftmals werden Sie neben Klassen des JDKs auch externe Bibliotheken einbinden, etwa die verbreitete Bibliothek Google Guava in Form der Datei `guava-29.0-jre.jar`, die unter `https://mvnrepository.com/artifact/com.google.guava/guava` heruntergeladen werden kann.

Exemplarisch betrachten wir eine Klasse `UseGuavaFromClassPathExample`, die die Klasse `com.google.common.base.Joiner` aus Google Guava zum Verketten von Strings nutzt:

```
package com.inden.javaprofi;

import com.google.common.base.Joiner;

public class UseGuavaFromClassPathExample
{
    public static void main(final String[] args)
    {
        joinWithGuava("Guava", null, "From", "ClassPath");
    }

    public static void joinWithGuava(final String... strings)
    {
        final Joiner joiner = Joiner.on(", " ).skipNulls();

        System.out.println(joiner.join(strings));
    }
}
```

Die obige Klasse ist in nachfolgend gezeigter Verzeichnishierarchie definiert, wobei wir auch erkennen, dass die Datei `guava-29.0-jre.jar` in einem Verzeichnis `externallibs` parallel zum Hauptverzeichnis des Moduls abgelegt ist:

```
jigsaw_ch15
|-- externallibs
|    `-- guava-29.0-jre.jar
|
`-- ch15_5_1_migration_compatibility_mode
    `-- src
        `-- main
            `-- java
                `-- com
                    `-- inden
                        `-- javaprofi
                            `-- UseGuavaFromClassPathExample.java
```

Kompilieren und Starten Versuchen wir das Ganze zu kompilieren und geben dabei über `-cp` einen `CLASSPATH` an:

```
javac -d build -cp ../externallibs/guava-29.0-jre.jar \
      $(find src -name '*.java')
```

Es entsteht die bekannte Spiegelung des `src`-Ordners im `build`-Ordner:

```
ch15_5_1_migration_compatibility_mode/
`-- build
    `-- com
        `-- inden
            `-- javaprofi
                `-- UseGuavaFromClassPathExample.class
```

Die beim Kompilieren entstandene Datei `UseGuavaFromClassPathExample` lässt sich durch Angabe der Abhängigkeit auf `guava-29.0-jre.jar` im `CLASSPATH` wie folgt starten:

```
java -cp ../externallibs/guava-29.0-jre.jar:build \
     com.inden.javaprofi.UseGuavaFromClassPathExample
```

Beachten Sie dabei, dass wir unsere kompilierten Sourcen über den `CLASSPATH` und das Verzeichnis `build` einbinden. Der obige Aufruf produziert folgende Konsolenausgabe:

```
Guava, From, ClassPath
```

Packaging Schließlich erzeugen wir ein JAR namens `myjarneedsguava.jar`:

```
mkdir lib

jar --create --file lib/myjarneedsguava.jar -C build .
```

Das ergibt folgenden Verzeichnisinhalt:

```
ch15_5_1_migration_compatibility_mode
`-- lib
    `-- myjarneedsguava.jar
```

Dieses JAR lässt sich dann mit folgendem Kommando starten:

```
java -cp ../externallibs/guava-29.0-jre.jar:lib/myjarneedsguava.jar \
     com.inden.javaprofi.UseGuavaFromClassPathExample
```

Auch hier nutzen wir den `CLASSPATH` sowohl für Guava als auch für unsere Klasse aus dem JAR namens `myjarneedsguava.jar`.

Fazit

Wir haben gesehen, dass wir Programme wie bisher gewohnt kompilieren und paketieren können, obwohl das JDK bereits modularisiert ist. Dieser exklusive Einsatz des `CLASSPATH` für die Applikationsbestandteile ist eine Variante, Rückwärtskompatibilität zu erhalten. Doch was geschieht eigentlich, wenn wir unser Programm modularisieren und Module verwenden wollen?

15.5.2 Migrationsszenarien

Wenn wir unsere Programme in mehrere Module untergliedern, dann gibt es dabei mitunter den einen oder anderen Stolperstein, wenn Klassen aus externen Bibliotheken eingebunden werden müssen. Wie wir schrittweise vorgehen können und welche Migrationsszenarien existieren, betrachten wir nun. Dabei lernen wir sowohl das Unnamed Module als auch Automatic Modules und deren Eigenschaften genauer kennen.

Unnamed Module

Das sogenannte Unnamed Module ist ein künstliches Modul: Jeder im `CLASSPATH` gefundene Typ wird dem ***Unnamed Module*** zugeordnet. Dieses besitzt spezielle Eigenschaften, um Migrationen zu erleichtern: Zunächst einmal liest das Unnamed Module alle anderen Module. Dadurch kann aus Typen des `CLASSPATH` auf alle in beliebigen Modulen enthaltenen Typen zugegriffen werden. Das ist insbesondere für die vom JDK exportierten Packages und Typen wichtig. Auf diese Weise sind alle Applikationen, die auf dem `CLASSPATH`-Mechanismus basieren, wie bisher lauffähig.

Eine Migration wird außerdem dadurch erleichtert, dass das Unnamed Module alle seine Packages exportiert. Um trotz des `CLASSPATH` ein Mindestmaß einer verlässlichen Konfiguration (***Reliable Configuration***) sicherzustellen, ist aus Named Modules kein Zugriff auf das Unnamed Module, also den `CLASSPATH`, gestattet. Mehr noch: Named Modules können keine Abhängigkeiten auf das Unnamed Module definieren und somit auch nicht besitzen.[5]

Automatic Modules

Bei einer Umwandlung einer bestehenden Applikation in ein modularisiertes Programm benötigt man ab und an aber doch Zugriff aus Modulen auf den `CLASSPATH`. Für diesen Fall existieren die sogenannten ***Automatic Modules***. Diese erlauben es einem konventionellen JAR, wie ein Named Module zu erscheinen, jedoch mit der Möglichkeit, auf Typen aus dem Unnamed Module und damit dem `CLASSPATH` zugreifen zu können.

Wie entsteht ein solches Automatic Module? Ganz einfach dadurch, dass wir das JAR im Module-Path aufführen. Das Modulsystem erkennt dies und erzeugt anhand des JAR-Namens automatisch ein korrespondierendes implizites Named Module. Bitte beachten Sie die Hinweise im folgenden Praxistipp.

Weil Automatic Modules keinen Moduldeskriptor enthalten, kann man im Vorhinein nicht wissen, von welchen Modulen ein Automatic Module abhängig ist. Zur leichteren Migration wird automatisch durch den Compiler bzw. die JVM ein künstlicher Moduldeskriptor mit `requires` »all« und `exports` »all« generiert. Dadurch kann ein Automatic Module alle Named Modules lesen. Darüber hinaus ist nicht vorhersehbar, welche Packages exportiert werden sollen. Daher werden der obigen Argumentation

[5] Das ist logisch, da ja die Referenzierung über den Namen erfolgt. Demnach gilt: Kein Name, keine Referenzierbarkeit!

folgend von einem Automatic Module alle Packages exportiert.[6] Schließlich bietet ein Automatic Module allen anderen Automatic Modules eine Implied Readability (transitive Propagation von Abhängigkeiten) – auch diese erleichtert die Migration, sie sollte bei einer möglichen späteren Konvertierung in ein Named Module aber genauer geprüft und vermutlich eingeschränkt werden.

Tipp: Wissenswertes zu Automatic Modules

Die Referenzierung bestehender JARs als Automatic Modules im Moduldeskriptor funktioniert nur dann, wenn die Namen in ein erwartetes Schema passen. Beispielsweise sind Minuszeichen innerhalb des Namens eines JARs problematisch, sofern sie nicht direkt vor der Versionsnummer stehen. Das betrifft beispielsweise das JAR von MongoDB (`mongo-java-driver-3.12.7.jar`). Dafür ist es nicht erlaubt, Folgendes im Moduldeskriptor zu schreiben:

```
// Achtung: Leider ist dies so syntaktisch nicht erlaubt
requires mongo-java-driver;
```

Zwar kann man für diese Fälle die Namen der JAR-Dateien von Hand anpassen und die Minuszeichen entfernen. Das führt aber zu recht langen und unleserlichen Namen und bereitet Schwierigkeiten bei Versionswechseln. Empfehlenswert ist es, für die Minuszeichen im Dateinamen einen Punkt (.) in dem per `requires` angegebenen Namen zu nutzen, weil dies vom Modulsystem unterstützt wird:

```
requires mongo.java.driver;
```

Es wäre wohl deutlich einfacher gewesen, wenn man die Namen von Modulen im Moduldeskriptor in Anführungszeichen angeben könnte, aber es gibt eine elegantere Abhilfe, die wir jetzt betrachten.

Abhilfe durch `Automatic-Module-Name` in `MANIFEST.MF` Für die gerade beschriebenen möglichen Schwierigkeiten bezüglich der automatischen Ableitung des Namens eines Moduls aus demjenigen des JARs gibt es folgende Abhilfe:[a] Der Name eines Automatic Module lässt sich mithilfe des Attributs `Automatic-Module-Name` in der Datei `MANIFEST.MF` des JARs festlegen. Das vermeidet Probleme, falls der Namen des JARs vom erwarteten Schema abweicht. Eine Beschreibung zu den Mechanismen beim Auflösen der Namen eines Automatic Module finden Sie in der Dokumentation zum Interface `ModuleFinder`: `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/module/ModuleFinder.html#of(java.nio.file.Path...)`.

[a] Diese habe ich zufällig beim Wechseln auf eine neuere Guava-Version entdeckt: Für Guava Version 24 hat sich der Name des JARs auf `guava-24.0-jre-jar` geändert. Damit schlägt der Automatismus und ein Verweis per `requires guava` fehl.

[6] Das erleichtert zwar eine Migration einer bestehenden in eine modularisierte Applikation, jedoch sollte man sich auch bewusst sein, dass so durchaus als intern gedachte Packages nach außen sichtbar und zugänglich werden.

Bottom-up-Migration und Migrationsszenarien

Um die nachfolgend vorgestellten Migrationsszenarien leichter nachvollziehen zu können, ist es wichtig, zu wissen, welche Zugriffe aus welchen Arten von Modulen erlaubt sind. Abbildung 15-13 zeigt dies, wobei die jeweils relevante Modulart leicht dunkler eingefärbt ist.

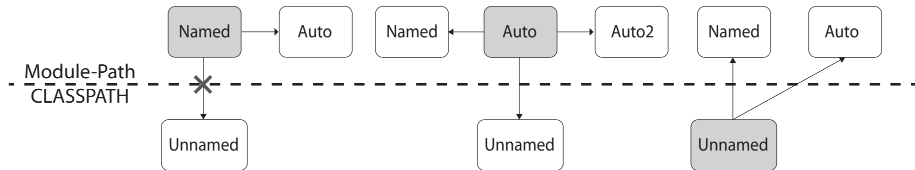

Abbildung 15-13 *Zugriffsmöglichkeiten aus unterschiedlichen Modularten*

Bei einer sogenannten Bottom-up-Migration werden JARs schrittweise in modulare JARs umgewandelt. Dabei kann man folgende zwei Szenarien unterscheiden:

1. **Bottom-up-Migration und Named Modules** – Wenn wir den Sourcecode eines JARs im Zugriff haben, können wir ein JAR in ein modulares JAR überführen, indem wir einen geeigneten Moduldeskriptor hinzufügen und das Modul neu kompilieren und paketieren.
2. **Bottom-up-Migration und Automatic Modules** – Besteht kein Zugriff auf den Sourcecode des JARs, so lässt sich das JAR nicht auf die zuvor beschriebene Weise in ein modulares JAR konvertieren. Als gute Alternative bleibt die Nutzung als Automatic Module, gegebenenfalls in Kombination mit dem Unnamed Module.

Stellen wir uns für eine Migration vor, eine Applikation bestehe aus verschiedenen JARs, die im `CLASSPATH` aufgeführt werden, etwa aus den JARs `app.jar`, `A.jar`, `B.jar` und `C.jar`.

Bottom-up-Migration und Named Modules Zunächst scheint die Bottom-up-Migration mit Named Modules recht einfach, weil man lediglich einen Moduldeskriptor erstellen muss, damit ein modulares JAR entsteht. Dabei gibt es allerdings zwei Hürden: Zunächst einmal gestaltet es sich einigermaßen aufwendig, herauszufinden, welche Packages exportiert werden sollen. Zudem können wir ein JAR nur dann in ein modulares JAR umwandeln, wenn dieses keine Abhängigkeiten auf andere, nicht modulare JARs des `CLASSPATH` besitzt, weil sich diese nicht im Moduldeskriptor referenzieren lassen. Das zeigt Abbildung 15-14.

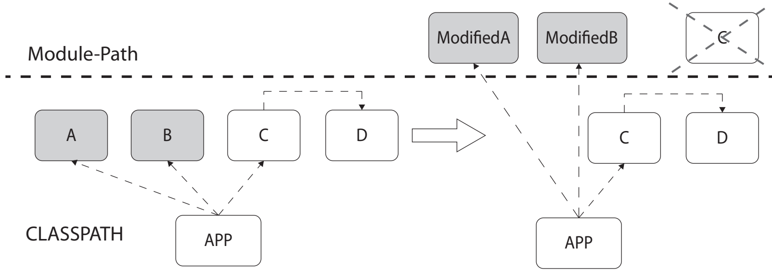

Abbildung 15-14 *Bottom-up-Migration und Named Modules*

Die gerade geschilderte Bottom-up-Migration mit der Umwandlung einzelner JARs in Named Modules ist vielfach nicht möglich, weil es Abhängigkeiten von einem Named Module auf eine Klasse vom `CLASSPATH` (also das Unnamed Module) oder aber ein Automatic Module gibt. Außerdem kann man – wie in unserem Beispiel – nur diejenigen JARs, die man unter eigener Kontrolle hat, in Named Modules konvertieren.

Zur Erinnerung sei nochmals Folgendes erwähnt: Interessanterweise und migrationserleichternd dürfen JARs aus dem `CLASSPATH` durchaus Abhängigkeiten auf Module besitzen: Das Unnamed Module liest alle Module.

Bottom-up-Migration und Automatic Modules Für viele Anwendungsfälle ist die Bereitstellung von JARs als Automatic Modules leichter und besser geeignet als der zuvor beschriebene Ansatz. Das gilt vor allem für externe Bibliotheken ohne Zugriff auf die Sourcen.

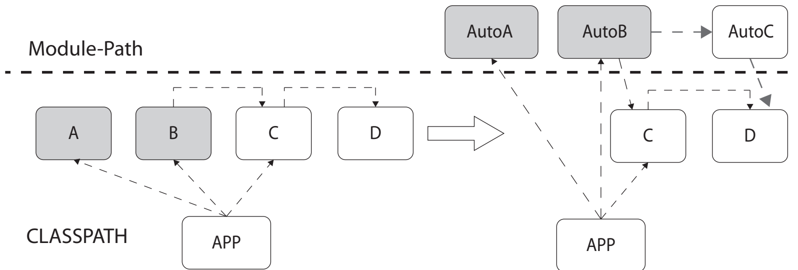

Abbildung 15-15 *Bottom-up-Migration und Automatic Modules*

Abbildung 15-15 verdeutlicht, dass sich nicht nur die beiden JARs `A.jar` und `B.jar` in Automatic Modules überführen lassen, sondern auch `C.jar`. Tatsächlich ist dies bei Wunsch sogar auch für `D.jar` möglich.

15.5.3 Fallstrick bei der Bottom-up-Migration

Bei der Bottom-up-Migration gibt es noch einen Punkt zu bedenken: Man sollte mit den JARs starten, die keine oder kaum Abhängigkeiten besitzen. Allerdings ist man intuitiv versucht, die Umwandlung mit dem eigenen Applikations-JAR zu beginnen. Das ist aber ein Irrweg. Um das zu verstehen, nutzen wir das vorangegangene Beispiel als Basis. Für die Transformation unserer Applikation in ein Modul benötigen wir einen Moduldeskriptor sowie eine leicht abweichende Verzeichnisstruktur wie folgt – für die Beispiele soll das `guava-29.0-jre.jar` im Verzeichnis `externallibs` liegen:

```
jigsaw_ch15
|-- externallibs
|   `-- guava-29.0-jre.jar
|
`-- ch15_5_3_pitfall_bottomup_migration
    `-- src
        `-- main
            `-- java
                |-- com
                |   `-- inden
                |       `-- javaprofi
                |           `-- UseGuavaFromClassPathInNamedModuleExample.java
                `-- module-info.java
```

Schritt 1: Definition des Moduldeskriptors

Bereits beim Erstellen des Moduldeskriptors würden wir stutzen und uns fragen, wie wir die Abhängigkeit zu Guava angeben sollen. Das wollen wir kurz außer Acht lassen. Weil hier die Applikation beschrieben wird, exportieren wir keine Packages und der Moduldeskriptor enthält somit lediglich die Moduldefinition:

```
module bottomup_unnamed_example
{
}
```

Schritt 2: Implementierung

Exemplarisch betrachten wir die aus dem letzten Beispiel leicht abgewandelte und in `UseGuavaFromClassPathInNamedModuleExample` umbenannte Klasse, die wiederum die Klasse `Joiner` aus Guava nutzt:

```
public class UseGuavaFromClassPathInNamedModuleExample
{
    public static void main(final String[] args)
    {
        final Joiner joiner = Joiner.on(", ").skipNulls();

        System.out.println(joiner.join("Guava", null, "As", null, "Unnamed"));
    }
}
```

Schritt 3: Kompilieren

Zum Kompilieren geben wir über `-cp` einen `CLASSPATH` für Guava an:

```
javac -d build -cp ../externallibs $(find src -name '*.java')
```

Im Gegensatz zu der reinen `CLASSPATH`-Variante erhält man folgende Kompilierfehler, die verdeutlichen, dass aus unserem Modul kein Zugriff auf die Klasse `Joiner` besteht:

```
src/main/java/com/inden/javaprofi/UseGuavaFromClassPathInNamedModuleExample.java
      :3: error: package com.google.common.base does not exist
import com.google.common.base.Joiner;
                             ^
src/main/java/com/inden/javaprofi/UseGuavaFromClassPathInNamedModuleExample.java
      :22: error: cannot find symbol
        final Joiner joiner = Joiner.on(", ").skipNulls();
              ^
  symbol:   class Joiner
  location: class UseGuavaFromClassPathInNamedModuleExample
src/main/java/com/inden/javaprofi/UseGuavaFromClassPathInNamedModuleExample.java
      :22: error: cannot find symbol
        final Joiner joiner = Joiner.on(", ").skipNulls();
                              ^
  symbol:   variable Joiner
  location: class UseGuavaFromClassPathInNamedModuleExample
3 errors
```

Fazit

Wie man sieht, scheitert der Versuch, aus einem Named Module auf Typen aus dem `CLASSPATH` zuzugreifen. Dieser vermeintliche Migrationspfad schlägt fehl, weil, wie schon einmal kurz erwähnt, aus einem Named Module nicht auf das Unnamed Module (`CLASSPATH`) zugegriffen werden kann!

15.5.4 Beispiel: Migration mit Automatic Modules

Wir lernen nun, wie man den gerade beschriebenen Fehler bei Migrationen vermeiden kann. Dazu wird die eigene Applikation als Named Module realisiert und Fremdbibliotheken werden als Automatic Modules eingebunden, hier für das JAR `guava-29.0-jre.jar` gezeigt. Das Verzeichnislayout sieht wie folgt aus:

```
jigsaw_ch15
|-- externallibs
|   `-- guava-29.0-jre.jar
|
`-- ch15_5_4_migration_automatic_modules
    `-- src
        `-- main
            `-- java
                |-- com
                |   `-- inden
                |       `-- javaprofi
                |           `-- AutomaticModuleExample.java
                `-- module-info.java
```

Schritt 1: Definition des Moduldeskriptors

Zur Referenzierung von Guava in unserem Moduldeskriptor muss man wissen, dass bis einschließlich Guava-Version 23 die Automatik des Modulsystems greift, die aus dem Namen des JARs den Namen des Automatic Module herleitet. Ab Version 24 wird der Modulname aus der Angabe des Attributs `Automatic-Module-Name` in der Datei `MANIFEST.MF` des JARs ermittelt. Somit ergeben sich folgende Varianten:

```
module automatic_module_example
{
    // requires guava;            // Bis guava-23.0.jar
    requires com.google.common;   // Ab guava-24.0-jre.jar
}
```

Schritt 2: Implementierung der nutzenden Applikation

Unsere Beispielapplikation besteht erneut nur aus einer einzigen Klasse, die zur Konkatenation einiger Strings die Klasse `Joiner` aus Guava nutzt:

```
package com.inden.javaprofi;

import com.google.common.base.Joiner;

public class AutomaticModuleExample
{
    public static void main(final String[] args)
    {
        final Joiner joiner = Joiner.on(", ").skipNulls();

        System.out.println(joiner.join("Guava", null, "As", null, "AUTOMATIC"));
    }
}
```

Schritt 3: Kompilieren

Versuchen wir das übliche Kommando zum Kompilieren:

```
javac -d build $(find src -name '*.java')
```

Wie aufgrund der per `requires` angegebenen, aber im Module-Path nicht aufgeführten Abhängigkeit zu erwarten war, kommt es zu folgender Fehlermeldung:

```
src/automatic_module_example/module-info.java:3:
      error: module not found: com.google.common
    requires com.google.common; // Ab guava-24.0-jre.jar
```

Zur Behebung müssen wir das Verzeichnis `externallibs`, in dem Guava liegt, folgendermaßen beim Kompilieren im Module-Path angeben:

```
javac -d build --module-path ../externallibs $(find src -name '*.java')
```

Schritt 4: Starten

Schließlich können wir das Programm wie folgt starten:

```
java --module-path ../externallibs:build \
     -m automatic_module_example/com.inden.javaprofi.AutomaticModuleExample
```

Das Programm erzeugt erwartungskonform folgende Konsolenausgabe:

```
Guava, As, AUTOMATIC
```

15.5.5 Beispiel: Automatic und Unnamed Module

Bisher haben wir nur den einfachsten Fall betrachtet, dass Automatic Modules in sich abgeschlossen waren und keine Typen anderer JARs bzw. Module außer denjenigen des JDKs benötigte. Wie gehen wir aber vor, wenn Querabhängigkeiten aus dem Automatic Module auf andere Typen aus dem `CLASSPATH` bestehen?

Nachfolgend nutzen wir JARs und Module in verschiedenen Modularten und der Aufrufhierarchie »Application Module → Automatic Module → Unnamed Module«:

- Wir erstellen eine einfache Applikation als Named Application Module.
- Diese soll das JAR `myjarneedsguava.jar`, das in Abschnitt 15.5.1 zum Kompatibilitätsmodus erstellt wurde, als Automatic Module einbinden.
- Innerhalb von `myjarneedsguava.jar` wird Funktionalität aus Google Guava verwendet. Deshalb muss diese Bibliothek zur Laufzeit im Zugriff sein. Wir wollen hier den `CLASSPATH`, also das Unnamed Module, nutzen.

Dazu kopieren wir ausgehend vom fett markierten Hauptverzeichnis das zuvor in Abschnitt 15.5.1 erstellte JAR `myjarneedsguava.jar` mit folgendem Kommando:

```
cp ../ch15_5_1_migration_compatibility_mode/lib/* ../ownlibs/
```

Dadurch sieht die Verzeichnisstruktur folgendermaßen aus:

```
jigsaw_ch15
|-- externallibs
|   `-- guava-29.0-jre.jar
|--ownllibs
|   `-- myjarneedsguava.jar
|
`-- ch15_5_5_migration_automatic_and_unnamed_modules
    `-- src
        `-- main
             `-- java
                 |-- com
                 |   `-- inden
                 |       `-- javaprofi
                 |           `-- auto_uses_unnamed
                 |               `-- AutomaticAndUnnamedModuleExample.java
                 `-- module-info.java
```

Schritt 1: Erstellen des Moduldeskriptors

Nun wollen wir das JAR `myjarneedsguava.jar` als Automatic Module einbinden. Dazu notieren wir die Abhängigkeit im Moduldeskriptor:

```
module auto_unnamed_modules_example
{
    requires myjarneedsguava;
}
```

Schritt 2: Implementierung der Applikation

Die folgende Klasse `AutomaticAndUnnamedModuleExample` nutzt Funktionalität aus unserem JAR:

```
package com.inden.javaprofi.auto_uses_unnamed;

import com.inden.javaprofi.UseGuavaFromClassPathExample;

public class AutomaticAndUnnamedModuleExample
{
    public static void main(final String[] args) throws Exception
    {
        System.out.println("AutomaticAndUnnamedModuleExample"));

        UseGuavaFromClassPathExample.joinWithGuava("Guava", "From ClassPath",
                                                   "Referenced By",
                                                   "Automatic Module");
    }
}
```

Hinweis: Notwendigkeit eindeutiger Packages

Bitte beachten Sie, dass wir unsere Klasse in einem Package mit einem eindeutigen Namen – hier das Subpackage `auto_uses_unnamed` – definieren müssen, wenn wir das bereits erstellte JAR einbinden wollen. Der Grund ist folgender: Das JAR enthält bereits das Package `com.inden.javaprofi` und das Modulsystem erlaubt keine Split Packages, also gleichnamige Packages in verschiedenen Modulen.

Schritt 3: Kompilieren

Das Kompilieren geschieht mit

```
javac -d build --module-path ../ownlibs  $(find src -name '*.java')
```

wodurch ein Ordner `build` mit den kompilierten Sourcen entsteht.

Schritt 4a: Starten

Wir wissen, dass die Applikation eine transitive Abhängigkeit besitzt: Das als Modul eingebundene JAR `myjarneedsguava.jar` benötigt eine Abhängigkeit auf `guava-29.0-jre.jar`. Wir starten das Programm trotzdem mal wie folgt:

```
java -p build:../ownlibs  \
     -m auto_unnamed_modules_example/com.inden.javaprofi.auto_uses_unnamed.
           AutomaticAndUnnamedModuleExample
```

Das führt zu folgender Ausgabe:

```
AutomaticAndUnnamedModuleExample
Exception in thread "main" java.lang.NoClassDefFoundError:
      com/google/common/base/Joiner
    at myjarneedsguava/com.inden.javaprofi.UseGuavaFromClassPathExample.
          joinWithGuava(UseGuavaFromClassPathExample.java:22)
    at auto_unnamed_modules_example/com.inden.javaprofi.auto_uses_unnamed.
          AutomaticAndUnnamedModuleExample.main(AutomaticAndUnnamedModuleExample.
          java:18)
Caused by: java.lang.ClassNotFoundException: com.google.common.base.Joiner
    at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(
          BuiltinClassLoader.java:583)
    at java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(
          ClassLoaders.java:178)
    at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:521)
    ... 2 more
```

Zunächst sieht man den Text `AutomaticAndUnnamedModuleExample`. Das zeigt, dass unser Named Application Module geladen und ausgeführt wird. Jedoch kommt es beim Abarbeiten des als Automatic Module eingebundenen JARs (das wiederum eine indirekte Abhängigkeit auf Google Guava besitzt) zu einem Fehler, da die Klasse `Joiner` nicht geladen werden kann, was wiederum zum `NoClassDefFoundError` führt.

Schritt 4b: Starten mit zusätzlichem CLASSPATH

Demnach ist der Start nicht erfolgreich, weil die externe Bibliothek Google Guava nicht gefunden wird. Deren Verfügbarkeit war beim Kompilieren nicht wichtig, da hierbei die Abhängigkeit nur aus dem als Bytecode vorliegenden JAR besteht. Zur Laufzeit wird die Klasse `Joiner` und somit das korrespondierende JAR aber benötigt.

Fügen wir beim Start der Applikation den `CLASSPATH` hinzu:

```
java -p build:../ownlibs -cp ../externallibs/guava-29.0-jre.jar \
     -m auto_unnamed_modules_example/com.inden.javaprofi.auto_uses_unnamed.
           AutomaticAndUnnamedModuleExample
```

Dann kommt es zu folgenden Ausgaben:

```
AutomaticAndUnnamedModuleExample
Guava, From ClassPath, Referenced By, Automatic Module
```

Fazit

Selbst an diesem einfachen Beispiel erkennt man, wie fragil das Arbeiten mit dem `CLASSPATH` ist. Bei einer vollständig modularisierten Applikation würde das Problem einer fehlenden Abhängigkeit direkt schon beim Kompilieren und nicht erst zur Laufzeit ersichtlich werden.

Aus dem bisher Gesagten wird aber auch deutlich, dass Automatic Modules in Kombination mit dem Unnamed Module einen sinnvollen, gangbaren Migrationspfad vorgeben: JARs aus Fremdbibliotheken können wir zumindest so lange als Automatic Modules einbinden, bis diese in einer modularen Variante vorliegen.

15.5.6 Beispiel: Abwandlung mit zwei Automatic Modules

Im vorherigen Beispiel wurde ein externes JAR noch in Form des Unnamed Module genutzt. Weil dies keine zuverlässige Konfiguration erlaubt, wäre es wünschenswert, wenn wir stattdessen ein Automatic Module nutzen könnten. Betrachten wir das Ganze in folgendem Verzeichnislayout:

```
jigsaw_ch15
|-- externallibs
|   `--  guava-29.0-jre.jar
|-- ownllibs
|   `-- myjarneedsguava.jar
|
`-- ch15_5_6_migration_two_automatic_modules
    `-- src
        `-- main
            `-- java
                |-- com
                |   `-- inden
                |       `-- javaprofi
                |           `-- onlyauto
                |               `-- OnlyAutomaticModulesExample.java
                `-- module-info.java
```

Schritt 1: Erstellen des Moduldeskriptors

Unser eigenes Modul sowie Guava binden wir wie folgt als Automatic Module ein:

```
module two_automatic_modules_example
{
    requires myjarneedsguava;
    requires com.google.common;
}
```

Schritt 2: Implementierung

Damit wäre bereits alles Wesentliche umgestellt, im Speziellen sind keine Modifikationen an den Klassen nötig. Weil wir aber einen anderen Text ausgeben wollen, ändern wir die zuvor genutzte Klasse `AutomaticAndUnnamedModuleExample` minimal ab und nennen sie in `OnlyAutomaticModulesExample` um:

```
package com.inden.javaprofi.onlyauto;

import com.inden.javaprofi.UseGuavaFromClassPathExample;

public class OnlyAutomaticModulesExample
{
    public static void main(final String[] args) throws Exception
    {
        System.out.println("OnlyAutomaticModulesExample");

        UseGuavaFromClassPathExample.joinWithGuava("Both", "As",
                                                   "Automatic Module");
    }
}
```

Schritt 3: Kompilieren und Starten

Beim Kompilieren wird lediglich der Module-Path verwendet. Das Ganze geschieht mit folgendem Kommando:

```
javac -d build --module-path ../ownlibs:../externallibs \
       $(find src -name '*.java')
```

Schritt 4: Starten

Der Start des Programms ist nun nur unter Angabe des Module-Path möglich:

```
java -p build:../ownlibs:../externallibs \
     -m two_automatic_modules_example/com.inden.javaprofi.onlyauto.
           OnlyAutomaticModulesExample
```

Dabei kommt es zu folgender Ausgabe:

```
OnlyAutomaticModulesExample
Both, As, Automatic Module
```

Fazit

Wir haben gesehen, wie einfach sich bestehende JARs in Automatic Modules überführen lassen. Das kann Migrationen erleichtern. Demzufolge bieten Automatic Modules oftmals eine Möglichkeit zu einer flexiblen Migration.

15.5.7 Fazit

Wir haben verschiedene Migrationsszenarien sowie mögliche Probleme betrachtet. Zusammenfassend lässt sich sagen, dass eine Migration einer bestehenden Applikation, die sich aus verschiedenen JARs zusammensetzt, gut überlegt werden sollte. Zwar erreicht man eine bessere technische Struktur, jedoch schafft man dadurch oftmals kaum oder keinen Business-Nutzen. Wenn man sich dennoch für eine Migration einer bestehenden Applikation entscheiden sollte, dann erfolgt diese am besten schrittweise JAR für JAR, um sofort auf mögliche Probleme reagieren zu können.

Für eine Migration bieten sich folgende Schritte an:

1. **Keine Abhängigkeiten** – Man beginnt mit JARs ohne externe Abhängigkeiten. Sofern deren Sourcecode zur Verfügung steht, lassen sich diese per Bottom-up-Migration in Named Modules überführen – bei Bedarf können diese immer noch als Automatic Modules eingebunden werden.
2. **Mit Abhängigkeiten** – Vorhandene JARs ohne Zugriff auf deren Sourcecode lassen sich problemlos als Automatic Modules nutzen.
3. **Eigene Applikation** – Nachdem die JARs entweder in Module überführt wurden oder aber im Unnamed Module verbleiben, kann man versuchen, die eigene Applikation in ein Named Application Module zu transformieren.

Wenn man bezüglich der Migration noch ein wenig unerfahren ist – was auf Sie nach der Lektüre dieses Kapitels nicht mehr zutreffen sollte ;-) –, kann man notfalls immer noch auf den Kompatibilitätsmodus zurückgreifen.

Hinweis: Mögliche Schwierigkeiten bei Migrationen

Bei der Migration einer bestehenden Applikation existieren mitunter (wenn auch nicht wünschenswert) zyklische Abhängigkeiten zwischen den Typen zweier JARs. Wollte man diese JARs in Module überführen und würde man dazu in den Moduldeskriptoren `A requires B` und `B requires A` vermerken, käme es dann direkt beim Kompilieren zu einem Fehler:

```
src/timeclient/module-info.java:3: error:
      cyclic dependence involving timeserver
    requires timeserver;
             ^
```

Für das Aufbrechen von Zyklen haben wir Services kennengelernt. Alternativ kann man ein Modul erstellen, das lediglich die benötigten Interfaces enthält. Etwas Ähnliches wurde in Abschnitt 15.2.2 besprochen.

15.6 Zusammenfassung

Dieses Kapitel hat einen Einstieg in das Thema Modularisierung gegeben. Durch den vorgestellten Themenmix sollten Sie einen guten ersten Eindruck gewonnen haben und fit für erste eigene Experimente sein.

Allerdings gibt es noch diverse weitere Dinge zu entdecken und auszuprobieren. Da wären z. B. Services oder die Besonderheiten und neuen Möglichkeiten mit Reflection. Diese leicht fortgeschrittenen Themen behandle ich in meinem Buch »Java - die Neuerungen in Version 9 bis 14: Modularisierung, Syntax- und API-Erweiterungen« [41].

16 Bad Smells

In diesem Kapitel werfen wir einen Blick auf verbreitete Fallstricke und Probleme, die einem Entwickler immer mal wieder begegnen. Man spricht in diesem Zusammenhang auch von »Bad Smells«. Das bezeichnet Abschnitte des Sourcecodes, die im übertragenen Sinne einen schlechten Geruch verbreiten – an denen potenziell etwas »faul« ist. Dieser eingängige Begriff wurde von Kent Beck und Martin Fowler im Buch »Refactoring« [20] zur Beschreibung derartiger Programmabschnitte eingeführt.

In diesem Kapitel werden Sie erfahren, welche Tücken und Fehler sich leicht in den Sourcecode einschleichen. Wenn wir diese Bad Smells erkennen, lokalisieren und verstehen können, sind wir auch in der Lage, Gegenmaßnahmen zu ergreifen. In den folgenden Abschnitten präsentiere ich dazu einen Katalog von Bad Smells, der sich in die Abschnitte 16.1 »Programmdesign«, 16.2 »Klassendesign« und 16.3 »Fehlerbehandlung und Exception Handling« gliedert und diverse Sourcecode-Auszüge unterschiedlicher Qualität zeigt. Wir analysieren jeweils die darin enthaltenen Probleme und schulen damit Ihr Auge. Das Erkennen eines Problems ist gut, allerdings sollten wir auch in der Lage sein, es zu beheben. Daher werden zu jedem Bad Smell passende Tipps zur Vermeidung gegeben und kleinere Abhilfemaßnahmen sofort vorgestellt. Allgemeingültige Umbaumaßnahmen stelle ich separat in Kapitel 17 »Refactorings« vor. Den Abschluss dieses Kapitels bilden in Abschnitt 16.4 »Häufige Fallstricke« einige Programmiertücken, die nicht die Schwere eines Bad Smells besitzen, aber dennoch hier erwähnt werden sollen, um diese in eigenen Programmen zu vermeiden.

Bad Smells am Beispiel

Bekanntermaßen beschreiben Bad Smells Sourcecode-Abschnitte, an denen möglicherweise etwas nicht in Ordnung ist. Diese Programmteile vergrößern die Gefahr für Fehlfunktionen oder enthalten bereits Fehler – manchmal sind solche Sourcecode-Stellen aber auch akzeptabel. In letzterem Fall sollte man dies durch einen Kommentar beschreiben.

Mit ein wenig Übung springen beim Analysieren von Sourcecode potenziell gefährliche Stellen schnell ins Auge. Außerdem ist es in der Regel so, dass Bad Smells gehäuft auftreten. Zum besseren Verständnis betrachten wir ein Beispiel mit der folgenden Methode `setDataState(int, OperationState)`:

```
public void setDataState(final int department, final OperationState state)
{
    if (this.dataState == null)
    {
        this.dataState = new HashMap<Integer, OperationState>();
    }
    this.dataState.put(new Integer(department), state);

    if (log.isInfoEnabled())
    {
        String msg = "device data state for department " + department + ": ";
        switch (state)
        {
            case RUNNING:
                msg += "running";
                break;
            case GOING_DOWN:
                msg += "going down";
                break;
            case DOWN:
                msg += "down";
                break;
            case GOING_UP:
                msg += "going up";
                break;
            default:
                msg += "unknown";
        }
        log.info(msg);
    }
}
```

Diese Methode scheint zunächst durchaus in Ordnung. Allerdings offenbaren sich dem geübten Auge hier unter anderem folgende Schwachstellen:

1. **Fehlende Plausibilitätsprüfungen der Parameter** – Es lauern Probleme durch unerwartete Parameterwerte: Nur ein kleiner Teil des `int`-Wertebereichs des Parameters `department` entspricht gültigen Werten. Außerdem ist für den Parameter `state` der Wert `null` ungültig und sollte zurückgewiesen werden.
2. **Unpassender Einsatz von Lazy Initialization** – Die Gefahr für mögliche Multithreading-Probleme wird durch den unsynchronisierten Einsatz von Lazy Initialization[1] für das Attribut `dataState` erhöht. Tatsächlich traten diese Probleme in der Praxis auf und führten zu Inkonsistenzen in den gespeicherten Werten.
3. **Unausgewogenheit beim Logging** – Der Logging-Code dominiert die Funktionalität: Nur die ersten Programmzeilen tragen zur Anwendungsfunktionalität bei. Die restlichen Zeilen bereiten Log-Informationen auf und sollten in eine Methode herausfaktoriert werden, um die Struktur klarer erkennbar zu machen. Ergänzend bietet es sich an, das `switch`-`case` durch eine Lookup-Map (vgl. Abschnitt 6.1.10) zu ersetzen.

[1] Lazy Initialization beschreibt die Technik, ein Attribut nicht direkt zu initialisieren, sondern zunächst mit `null` zu belegen, um die Konstruktion komplexer Objekte zu vermeiden. Erst beim ersten Zugriff erfolgt dann eine Initialisierung. Kapitel 22 beschreibt diese Technik im Detail.

16.1 Programmdesign

Nach diesem einführenden Beispiel stelle ich nun einen Katalog von Bad Smells vor und beginne dabei mit möglichen Problemen im Programmdesign.

16.1.1 Bad Smell: Verwenden von Magic Numbers

Oftmals kommen an beliebigen Stellen im Sourcecode verschiedene Zahlenwerte vor, die explizit als Zahlenliteral – als sogenannte ***Magic Number*** – angegeben werden. Zum Teil existieren funktionale und semantische Abhängigkeiten zwischen diesen Werten, die jedoch nicht im Sourcecode, sondern bestenfalls durch Kommentare ausgedrückt werden können.

Wir betrachten hier eine Klasse `CommandExecutor`, die im Konstruktor einen `int`-Wert erhält, der steuert, wie Kommandos verwaltet und einer internen Datenstruktur hinzugefügt werden. Für diesen Zweck gibt es mehrere Konstanten, unter anderem die Konstante `ADD_AS_LAST` mit dem Wert `2`. Im folgenden Beispiel werden zwei Instanzen der Klasse `CommandExecutor` erzeugt, sowohl unter Verwendung der Konstantendefinition als auch durch Angabe des korrespondierenden Zahlenwerts:

```
final CommandExecutor executor1 = new CommandExecutor(ADD_AS_LAST);

final CommandExecutor executor2 = new CommandExecutor(2);  // Magic Number 2
```

Warum ist das ein Bad Smell? Bereits anhand dieses einfachen Beispiels sieht man, dass die Lesbarkeit unter dem Einsatz von Magic Numbers leidet. Das liegt daran, dass ein Zahlenwert wenig semantische Aussagekraft besitzt. Erschwerend kommt hinzu, dass Änderungen an den Werten – etwa wenn obige Konstante z. B. später den Wert 7 erhält – überall im nutzenden Sourcecode nachgezogen werden müssen. Das kann sich aufwendig und fehleranfällig gestalten, alle betroffenen Vorkommen einer Magic Number lediglich anhand ihres Werts aufzuspüren: Logischerweise repräsentiert nicht jedes Vorkommen der Zahl `2` die Konstante `ADD_AS_LAST`. Wird bei einer solchen Korrektur eine Stelle übersehen, so kann dies zu unerwarteten Resultaten und schwer zu findenden Fehlern führen: Wird irgendwann einmal der Wert der Konstante auf `4711` verändert und eine andere Konstante erhält den Wert `2`, so besagt diese `2` nun beispielsweise das Einfügen an erster Stelle und verändert den Programmablauf damit komplett. Wird dagegen eine andere Programmstelle versehentlich geändert, so kommt es dort zu Fehlern im Programmverhalten, weil der neue Wert (die `4711`) dort keinen Sinn ergibt und möglicherweise sogar außerhalb des zulässigen Wertebereichs liegt.

Tipps und Refactorings Mit sprechenden Konstantennamen kann man leicht ausdrücken, was bewirkt werden soll, hier das Einfügen an letzter Position. Zudem stellen nachträgliche Änderungen der Werte kein Problem im nutzenden Sourcecode dar. Ein weiterer Vorteil der Verwendung von Konstanten ist, dass diese im Nachhinein, wie

in diesem Beispiel sicher sinnvoll, leicht und ohne größere Änderungen in eine `enum`-Aufzählung umgewandelt werden können. Wenn jedoch Magic Numbers verwendet wurden, dann sollte man jedes Vorkommen überprüfen und gegebenenfalls anpassen.

16.1.2 Bad Smell: Konstanten in Interfaces definieren

Java erlaubt es, Konstanten in einem Interface zu definieren. Dabei lassen sich zwei Varianten unterscheiden. In der ersten wird ein Interface ausschließlich zur Definition von Konstanten verwendet, hier durch das Interface `FigureConstants` gezeigt:

```
//ACHTUNG: Interfaces (möglichst) so nicht verwenden
public interface FigureConstants
{
    // Randbreite und -höhe
    int BORDER_WIDTH  = 5;
    int BORDER_HEIGHT = 5;

    // Abstand der Punkte im Raster in X- und Y-Richtung
    int GRID_SIZE_X   = 20;
    int GRID_SIZE_Y   = 20;
}
```

In der zweiten Variante – hier am Beispiel des Interface `FigureIF` – werden sowohl Konstanten als auch Methoden definiert:

```
//ACHTUNG: Interfaces (möglichst) so nicht verwenden
public interface FigureIF
{
    // Rückgabewerte für hitTest
    int BORDER_HIT   = 0;
    int TITLE_HIT    = 1;
    int SIZE_BOX_HIT = 2;

    int hitTest(final int x, final int y);
    void draw(final Graphics g);
}
```

Warum ist das ein Bad Smell? Die Definition von Konstanten in Interfaces ist in vielerlei Hinsicht ungünstig. Zunächst ist durch die JLS jede dort definierte Konstante implizit `public static final` und jede Methode `public abstract`. Daher kann die Sichtbarkeit nicht eingeschränkt werden. Die Situation hat sich mit Java 8 und 9 leider nicht zum Guten geändert: Seit Java 8 sind neben Defaultmethoden auch statische Methoden in Interfaces erlaubt (vgl. Abschnitt 4.2). Mit Java 9 lassen sich sogar private Methoden in Interfaces definieren, was allerdings kein schönes Design ist.

Zudem ist es möglich, ein Konstanten-Interface durch eine Klasse zu implementieren. Dadurch werden die Konstantennamen in den Namensraum der jeweiligen Klasse aufgenommen, und somit geht der Verweis auf den tatsächlichen Definitionsort der Konstante verloren. Für Konstanten als Bestandteile eines Methoden deklarierenden Interface besteht das Problem gleichermaßen.

In der folgenden Klasse `BadFigure`, die zu Demonstrationszwecken das Interface `FigureConstants` implementiert, ist dies in der Methode `isInside(int, int)` mit der Verwendung der Konstanten `BORDER_WIDTH` und `BORDER_HEIGHT` gezeigt, wobei zum einen eine Referenzierung über `FigureConstants` und zum anderen über `BadFigure` erfolgt:

```
// ACHTUNG: Interfaces (möglichst) so nicht verwenden
public class BadFigure implements FigureConstants
{
    final Rectangle boundingRect;

    BadFigure(final Rectangle boundingRect)
    {
        this.boundingRect = boundingRect;
    }

    public boolean isInside(final int x, final int y)
    {
        final Rectangle innerRect = new Rectangle(boundingRect);

        // Definitionsort der Konstanten BORDER_WIDTH und BORDER_HEIGHT
        // ist bei Referenzierung über BadFigure unklar
        innerRect.grow(-FigureConstants.BORDER_WIDTH,
                       -BadFigure.BORDER_HEIGHT);

        return innerRect.contains(x, y);
    }
}
```

Die beschriebene und im Listing gezeigte »Namensraumverschmutzung« scheint zunächst ein etwas hergeholtes Argument zu sein. Doch in der Praxis erschwert eine solche Vorgehensweise mögliche Erweiterungen, Refactorings und die Nachvollziehbarkeit. Betrachten wir dazu die Problematik einmal genauer: Nehmen wir an, eine Klasse `BaseFigure` implementiert versehentlich ein Konstanten-Interface `Constants` mit der Konstante `OK`. Eine Erweiterung der Klassenhierarchie um eine Subklasse `ProcessingFigure` führt zu Kompilierfehlern, wenn diese Subklasse ein Interface `ProcessingResults` implementiert, in dem ebenfalls eine Konstante `OK` definiert ist: Die Konstante `OK` ist dann mehrdeutig. Das zeigt Abbildung 16-1.

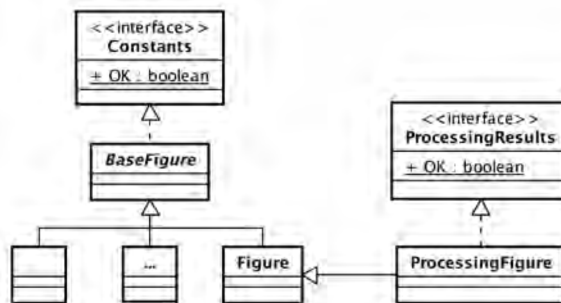

***Abbildung 16-1** Doppeldeutigkeit der Konstante* `OK`

Tipps und Refactorings Benutzen Sie Interfaces nur dazu, wozu sie aus OO-Sicht dienen sollen, nämlich, um eine Schnittstelle mit angebotenen Methoden zu definieren.

Reine Konstanten-Interfaces können in der Regel in finale Konstantensammlungsklassen mit privaten Konstruktoren oder `enum`-Aufzählungen umgewandelt werden. Wie man dabei vorgeht, beschreibt das Refactoring WANDLE KONSTANTENSAMMLUNG IN `ENUM` UM in Abschnitt 17.4.12 als Schritt-für-Schritt-Anleitung.

Wurden solche Interfaces allerdings bereits (versehentlich) implementiert, dann muss mühselig jedes Vorkommen einer Konstante mit einer Referenz auf die neue Konstantenklasse versehen werden. Dabei kann es zu Problemen kommen, wenn man nicht alle Klassen im Zugriff hat oder verändern kann, die das Interface implementiert haben. Daher ist es sinnvoll, alle bekannten Nutzer vor einer solchen Maßnahme über den Umbau zu informieren. Ist die Nutzerbasis groß (z. B. für das Java-API), so ist ein solches Vorgehen nicht ohne Schwierigkeiten möglich und eine solche Designsünde lässt sich kaum mehr korrigieren, sondern oftmals nur als veraltet markieren. Dazu nutzt man das Javadoc-Tag `@deprecated` oder die Annotation `@Deprecated`. ***An diesem Beispiel erkennt man, wie wichtig es ist, gleich von Anfang an mit großer Sorgfalt zu arbeiten***. Dies gilt insbesondere bei der Implementierung aller nach außen sichtbaren Klassen und Methoden.

Info: Ähnliche Probleme durch statischen Import (`import static`)

Statische Imports ermöglichen es, Attribute oder Methoden anderer Klassen ohne Angabe des qualifizierenden Klassennamens zu nutzen. Es werden dadurch weitere Namen in den Namensraum der eigenen Klasse aufgenommen.

Beim Import statischer Attribute kommt es bei gleichnamigen, eigenen Klassenattributen zu Namenskonflikten und Kompilierproblemen, wie dies z. B. beim Implementieren verschiedener Interfaces mit gleichnamigen Konstanten der Fall ist.

Bei Mehrdeutigkeiten in Bezug auf Methoden treten keine Fehler beim Kompilieren auf, stattdessen überdeckt eine Objektmethode aus der eigenen Klasse eine statisch importierte Methode. Dadurch kommt es gegebenenfalls allerdings zu folgendem Problem: ***Wird bei einer Erweiterung eine Objektmethode eingeführt, so wird das Verhalten unerwartet verändert, da als Folge nicht mehr die statisch importierte Methode aufgerufen wird, sondern die neu eingeführte Methode der eigenen Klasse.*** Dies gilt übrigens auch dann, wenn die aufrufende Methode selbst statisch ist!

16.1.3 Bad Smell: Zusammengehörende Konstanten nicht als Typ definiert

Müssen einige Werte als Konstanten definiert werden, wie etwa die Einfügestrategien für Kommandos aus dem Beispiel der Klasse `CommandExecutor`, sieht man häufiger Definitionen zusammengehörender Konstanten als `int`-Werte:

```
/** altes Kommando durch neues ersetzen */
public static final int REPLACE_OLD_OR_ADD_AS_LAST = 0;

/** Kommando als erstes Kommando neu erzeugen */
public static final int ADD_AS_FIRST               = 1;

/** Kommando als letztes Kommando neu erzeugen */
public static final int ADD_AS_LAST                = 2;
```

Warum ist das ein Bad Smell? Werden Konstanten als `int` definiert, so sind als Werte grundsätzlich alle aus dem Wertebereich des Typs erlaubt. Dadurch können auch unsinnige Werte Anwendung finden, wie folgendes Beispiel zeigt:

```
// Was mag die 4711 bedeuten?
final CommandExecutor executor = new CommandExecutor(4711);
```

Wünschenswert ist es, sicherzustellen, dass die übergebenen Werte im Bereich der definierten Konstanten liegen. Das erfordert eine Bereichsprüfung und wird schnell unübersichtlich. Dieser Effekt verstärkt sich, wenn die Prüfungen an verschiedenen Stellen im Sourcecode notwendig sind, die Werte keinen zusammenhängenden Bereich abbilden oder sich Werte nachträglich ändern bzw. neue Konstanten hinzukommen. Machen wir es konkret für den folgenden Konstruktor:

```
public CommandExecutor(final int strategy)
{
    if (strategy < REPLACE_OLD_OR_ADD_AS_LAST || strategy > ADD_AS_LAST)
    {
        throw new IllegalArgumentException("parameter 'strategy' is invalid: " +
                          "value='" + strategy + "' not in range [" +
                          REPLACE_OLD_OR_ADD_AS_LAST + " -- " + ADD_AS_LAST + "]");
    }
    this.registrationStrategy = strategy;
}
```

Prinzipiell gut an der Parameterprüfung ist, dass sie überhaupt vorhanden ist und der Benutzer bei einem Fehler im Aufruf detailliert sowohl über die Wertebelegung als auch über gültige Parameterwerte informiert wird.

Allerdings stehen diesen Vorteilen einige Nachteile gegenüber. Die obige Wertebereichsprüfung und Fehlerbehandlung ist aus folgenden Gründen ziemlich fragil:

1. Es werden Annahmen über die konkreten Werte der Konstanten `REPLACE_OLD_OR_ADD_AS_LAST` und `ADD_AS_LAST` gemacht, insbesondere, dass der Wert von `ADD_AS_LAST` den größeren der beiden Werte repräsentiert. Wird der Wert der Konstanten `REPLACE_OLD_OR_ADD_AS_LAST` bzw. `ADD_AS_LAST` derart verändert, dass diese Annahme nicht mehr gilt, scheitert die Wertebereichsprüfung.
2. Der Vergleich setzt voraus, dass durch die Konstanten ein zusammenhängender Wertebereich abgedeckt wird. Diese Forderung erschwert unter Umständen eine sinnvolle Vergabe der Parameterwerte.

3. Wird im Nachhinein eine zusätzliche Kommandoart hinzugefügt, erfordert dies nicht nur die Definition einer neuen Konstanten, sondern es müssen überall im verwendenden Sourcecode Anpassungen in den Wertebereichsprüfungen erfolgen.

Tipps und Refactorings In der Regel möchte man Aufrufern lediglich erlauben, symbolische Werte, d. h. Konstantennamen, zu verwenden. Das lässt sich elegant und einfach durch das Sprachmittel `enum` erreichen, mit dem man Konstanten zu einem neuen Typ zusammenfassen kann:

```
enum RegistrationStrategyType
{
    REMOVE_OLD_AND_ADD_AS_LAST(0, "altes Kommando durch neues ersetzen"),
    ADD_AS_FIRST(1, "Kommando als erstes Kommando neu erzeugen"),
    ADD_AS_LAST(2, "Kommando als letztes Kommando neu erzeugen");

    final int    value;
    final String description;

    RegistrationStrategyType(final int value, final String description)
    {
        this.value = value;
        this.description = description;
    }
}
```

Die durch Einsatz eines `enum` gewonnene Typsicherheit löst sowohl die Probleme der Bereichsprüfung – diese ist nun überflüssig, da sie implizit durch den Compiler basierend auf der Typangabe sichergestellt wird – als auch die der nicht zusammenhängenden Wertebereiche oder sich ändernder Konstantenwerte. Eine Anleitung liefert das Refactoring WANDLE KONSTANTENSAMMLUNG IN `ENUM` UM in Abschnitt 17.4.12.

16.1.4 Bad Smell: Casts auf unbekannte Subtypen

Casts werden in Programmen verwendet, wenn eine Typumwandlung gewünscht wird, die vom Compiler nicht garantiert werden kann. In wenigen Fällen ist ein Einsatz unumgänglich, beispielsweise wenn man ein Zahlenliteral vom Typ `int` an eine Methode mit `short`-Parametern übergeben möchte. In der Regel weist der Einsatz eines Casts allerdings auf eine potenzielle Fehlerquelle hin, und man sollte einen genaueren Blick auf die entsprechende Sourcecode-Stelle werfen.

Betrachten wir die Problematik am Beispiel einer Methode `getPersons()`, die in einem Interface `IDataAccess` definiert ist:

```
interface IDataAccess
{
    List<Person> getPersons();
}
```

Der folgende Sourcecode-Ausschnitt nutzt eine Realisierung dieses Interface zum Aufruf von `getPersons()`. Vor der Zuweisung an die Variable `persons` findet ein Cast auf eine `LinkedList<Person>` statt: