

Einführung

Während Sie in den ersten vier Kurseinheiten die Grundlagen der objektorientierten Programmierung in Java kennen gelernt haben, haben Sie sicherlich immer wieder festgestellt, dass Programme beim ersten Entwurf meist fehlerhaft sind. Entwurfs- und Programmierfehler waren schon die Ursache vieler kostspieliger und auch lebensbedrohlicher Vorfälle. So ist z. B. die Explosion der Ariane-Rakete im Juni 1996¹⁷ eindeutig auf Programmierfehler zurückzuführen.

Deshalb werden wir im Laufe dieser Kurseinheit verschiedene Methoden kennen lernen um unsere Programme robuster zu gestalten, Fehler zu vermeiden und die Programme auf Fehler zu untersuchen und diese ggf. zu beheben.

Zunächst lernen wir das Konzept der **Ausnahmen** kennen, die es uns erlauben, unerwartete Situationen zur Laufzeit zu signalisieren und gegebenenfalls auf diese Ausnahmesituationen zu reagieren.

Um Fehler bei der Benutzung anderer Klassen zu vermeiden und um das Verhalten einzelner Klassen und Methoden festzulegen, lernen wir die **Dokumentation** mit **javadoc** kennen.

Auch wenn beim Entwurf und der Implementierung schon versucht wurde, Fehler zu vermeiden, so sollte jede Software trotzdem getestet werden. Wir werden die verschiedenen **Teststufen** und **Testarten** sowie die **Planung** und **Durchführung** von Tests kennen lernen. Zur **Automatisierung** von Tests für Java-Klassen verwenden wir das Framework **JUnit**. Bei der Planung der Tests sollte insbesondere auf die geeignete Auswahl der einzelnen Testfälle geachtet werden.

Die Kurseinheit schließt mit ein paar Hinweisen zu typischen **Fehlern** und wie Fehler lokalisiert werden können.

17 http://www5.in.tum.de/lehre/seminare/semsoft/unterlagen_02/ariane/website/Ariane.Htm

Lernziele

- Das Konzept der Ausnahmebehandlung erklären und Exceptions in Java einsetzen können.
- Wissen, wie man javadoc-Kommentare zu Klassen erstellt.
- Die Notwendigkeiten, Schwierigkeiten und Grenzen des Testens kennen.
- Den Unterschied zwischen Testen und Verifizieren kennen.
- Die verschiedenen Teststufen und Testarten kennen.
- Wissen, wie Tests geplant und durchgeführt werden.
- Gängige Techniken zur Identifikation von Testfällen kennen und anwenden können.
- Testfälle für Java-Klassen mit Hilfe von JUnit implementieren und Ausführen können.
- Techniken zur Vermeidung und Lokalisierung von Fehlern kennen.

28 Ausnahmen

Wir sind bei unseren bisherigen Beispielen auf einige Fälle gestoßen, wo nicht alle Werte als Argumente für Methoden zulässig sind. Wir wollten beispielsweise nicht zulassen, dass ein Rabatt negativ oder größer als 100 % sein darf. Solche Situationen wurden entweder durch Ausgaben oder durch als ungültig interpretierte Rückgabewerte gekennzeichnet. Weitere typische Probleme sind der Zugriff auf ein Attribut oder eine Methode an einer Verweisvariable, der den Wert `null` liefert oder der Zugriff auf eine nicht existierende Datei.

Wir haben gesehen, dass in Programmen immer wieder problematische Situationen auftreten können. Java bietet Sprachkonstrukte, um mit Laufzeitfehlern konstruktiv umzugehen. Sie ermöglichen es,

- das Auftreten von Ausnahmesituationen zu überwachen, Ausnahmesituation
- im Fall einer Ausnahmesituation, bei deren Auftreten der normale Verlauf der Programmabarbeitung unterbrochen wird, vom Laufzeitsystem ein Ausnahme-Objekt erzeugen zu lassen und Ausnahme-Objekt
- die Kontrolle an spezielle Programmteile zu übergeben, die versuchen, den Laufzeitfehler aufzufangen (engl. *catch*) und zu behandeln. Laufzeitfehler

Wir werden in diesem Kapitel die Erzeugung von Ausnahme-Objekten und die Behandlung von Ausnahmen kennen lernen. Als Beispiel verwenden wir die Methode `legeRabattFest()` der Klasse `Rechnung`, die wir dahingehend verbessern wollen, dass sie keinen negativen Rabatt und keinen Rabatt von über 100 Prozent akzeptiert.

28.1 Ausnahmetypen

In Java sind konkrete Ausnahmen [JLS: § 11] Objekte von bestimmten Ausnahme-klassen. Die allgemeinste Klasse ist die Klasse `Throwable`. Die beiden direkten Unterklassen von `Throwable` sind `Error` und `Exception` [JLS: § 11.5]. Exemplare der Klasse `Exception` oder einer Unterklasse sind behandelbare Ausnahmen, wie zum Beispiel Probleme bei der Ein- und Ausgabe (`IOException`). Bei einer solchen Ausnahme soll das Programm nicht einfach abgebrochen werden, sondern es soll wenn möglich eine Lösung für das Problem gefunden werden. Lässt sich das Problem nicht beheben, so sollte eine entsprechende Mitteilung an den Benutzer erfolgen und das Programm ordentlich, d. h. mit einer `return`-Anweisung und nicht mit einer Ausnahme, beendet werden. Die Klasse `RuntimeException` ist eine direkte Unterklasse von `Exception`. Alle Exemplare dieser Klasse oder einer ihrer Unterklassen, zum Beispiel `ArithmeticException` und `NullPointerException` spielen eine besondere Rolle, auf die wir später bei

Ausnahmeklassen

`Throwable`

`Exception`

`RuntimeException`

der Deklaration und Behandlung von Ausnahmen noch einmal zurückkommen werden. Die Klassenhierarchie ist in Abb. 28.1-1 angedeutet.

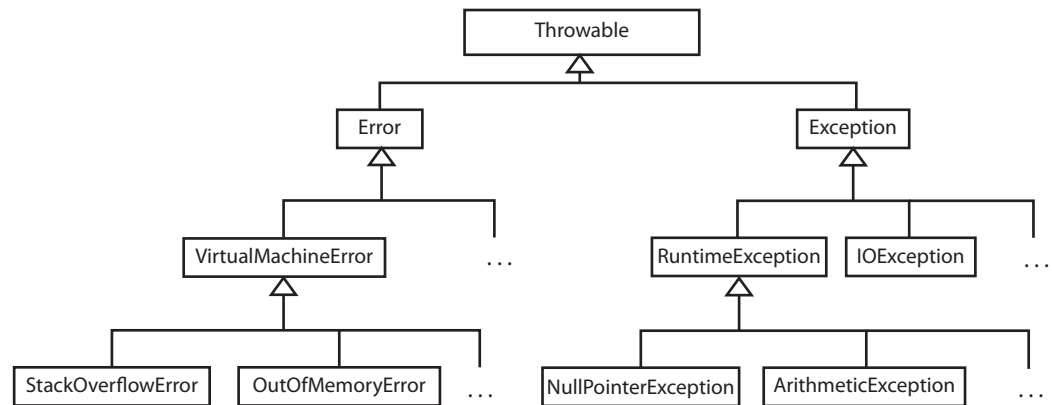


Abb. 28.1-1: Die Klasse Throwable und ihre wichtigsten Unterklassen

Ein Programmierer kann zusätzlich zu den bestehenden Ausnahmeklassen auch eigene Typen definieren. Eigene Ausnahmeklassen sind im Normalfall Unterklassen von `Exception`.

```
public class MeineAusnahme extends Exception {
}
```

Die Klasse `Exception` stellt schon einige hilfreiche Konstruktoren und Methoden zur Verfügung. Jede Ausnahme besitzt zur genaueren Beschreibung des Problems eine Nachricht (`message`). Diese kann, wenn gewünscht, mit Hilfe des Konstruktors der Oberklasse `Exception` festgelegt und mit Hilfe der Methode `getMessage()` erfragt werden. Wird die `toString()`-Methode aufgerufen, zum Beispiel durch eine Ausgabeanweisung, so werden der Name der Ausnahmeklasse und die Nachricht zurückgeliefert. Weitere Methoden der Klasse werden wir noch in den späteren Abschnitten kennen lernen.

`getMessage()`

Eine Ausnahmeklasse für einen ungültigen Rabatt könnte folgendermaßen aussehen:

```
public class UngueltigerRabattAusnahme extends Exception {
    public UngueltigerRabattAusnahme(double rabatt) {
        // setzt die Fehlermeldung
        super("Ein Rabatt von " + rabatt
            + " ist nicht zulaessig.");
    }
}
```

Nachdem wir nun wissen, was für Typen von Ausnahmen es gibt und wie wir eigene Typen definieren können, werden wir uns im nächsten Abschnitt damit beschäftigen, wie solche Ausnahmen in einem Java-Programm verwendet werden können.

28.2 Ausnahmen erzeugen und werfen

Wir wollen nun unsere Methode `legeRabattFest()` so ändern, dass sie keinen negativen Rabatt und keinen Rabatt von über 100 Prozent akzeptiert. Sie soll, falls sie mit einem unzulässigen Wert aufgerufen wird, eine Ausnahme erzeugen und diese werfen. Wird in einer Methode eine Ausnahme geworfen, so wird die Methode sofort mit einem Fehler beendet und dies dem Aufrufer mitgeteilt. Das Werfen einer Ausnahme geschieht mit Hilfe einer `throw`-Anweisung.

Definition 28.2-1: `throw`-Anweisung

Eine `throw`-Anweisung setzt sich aus dem Schlüsselwort `throw` und einer Referenz auf ein Ausnahmeobjekt, also ein Objekt vom Typ `Throwable` zusammen. [JLS: § 14.18]

`throw`-Anweisung

```
throw Ausnahmeobjekt;
```

Häufig wird erst in der Anweisung mit Hilfe eines Konstruktoraufrufs ein neues Ausnahmeobjekt erzeugt.

Kann eine Methode eine Ausnahme werfen, so muss sie dies auch in ihrem Methodenkopf angeben.

Definition 28.2-2: `throws` im Methodenkopf

Alle Ausnahmen, die eine Methode werfen kann, werden mit Hilfe des Schlüsselwortes `throws` nach der Parameterliste im Methodenkopf angegeben. [JLS: § 8.4.6, § 8.8.5]

`throws`

```
Modifikatoren Ergebnistyp Name(Parameterliste)
                               throws Ausnahmetypenliste { }
```

Die `Ausnahmetypenliste` besteht aus mindestens einem `Ausnahmetyp`. Weitere `Ausnahmetypen` können durch Kommata getrennt angegeben werden.

Unsere Methode `legeRabattFest()` würde folglich nun folgendermaßen aussehen:

```
public class Rechnung {
    private double rabatt;
    // ...

    void legeRabattFest(final double neuerRabatt)
        throws UngueltigerRabattAusnahme {
        if(neuerRabatt < 0 || neuerRabatt > 1) {
            throw new UngueltigerRabattAusnahme(neuerRabatt);
        }
        this.rabatt = neuerRabatt;
    }
}
```

nicht zu
berücksichtigende
Ausnahmen
IllegalArgumentException-
Exception

Ausnahmen vom Typ `RuntimeException` müssen nicht explizit im Methodenkopf angegeben werden. Bei ihnen handelt es sich um nicht zu berücksichtigende Ausnahmen (engl. *unchecked exceptions*). [JLS: § 11.2]

Würden wir statt unserer selbst definierten Ausnahme die vordefinierte Ausnahme `IllegalArgumentException` nutzen, die eine Unterklasse von `RuntimeException` ist, so müssten wir folglich keinen `throws`-Teil im Methodenkopf angeben.

```
public class Rechnung {
    private double rabatt;
    // ...

    void legeRabattFest(final double neuerRabatt) {
        if(neuerRabatt < 0 || neuerRabatt > 1) {
            throw new IllegalArgumentException(
                "Dieser Rabatt ist ungueltig: " + neuerRabatt);
        }
        this.rabatt = neuerRabatt;
    }
}
```

Bemerkung 28.2-1:

Kann eine Methode mehrere verschiedene Ausnahmen werfen, so können alle Typen einzeln oder nur entsprechende Oberklassen angegeben werden. Es ist besser, alle Ausnahmen explizit zu erwähnen, da so der Aufrufer auch geeignet auf jeden Typ reagieren kann.

Würden wir zwei Spezialisierungen von `UngueltigerRabattAusnahme` definieren, um zwischen negativen und zu hohen Rabatten unterscheiden zu können, so könnten im Methodenkopf entweder `UngueltigerRabattAusnahme` oder die beiden Spezialisierungen, mit Komma getrennt, aufgeführt werden.

```
public class NegativerRabattAusnahme
    extends UngueltigerRabattAusnahme {
    public NegativerRabattAusnahme(double rabatt) {
        super(rabatt);
    }
}

public class ZuHoherRabattAusnahme
    extends UngueltigerRabattAusnahme {
    public ZuHoherRabattAusnahme(double rabatt) {
        super(rabatt);
    }
}

public class Rechnung {
    private double rabatt;
    // ...
}
```

```

void legeRabattFest(final double neuerRabatt) throws
    NegativerRabattAusnahme, ZuHoherRabattAusnahme {
    if(neuerRabatt < 0) {
        throw new NegativerRabattAusnahme(neuerRabatt);
    }
    if (neuerRabatt > 1) {
        throw new ZuHoherRabattAusnahme(neuerRabatt);
    }
    this.rabatt = neuerRabatt;
}

```

Ausnahmen bieten somit die Möglichkeit, Methoden auf eine andere Art als mit einer `return`-Anweisung zu beenden und zusätzlich dem Aufrufer den Grund und weitere Informationen mitzuteilen.

Nachdem wir nun gelernt haben, Ausnahmen in Methoden zu werfen, werden wir uns im nächsten Abschnitt damit beschäftigen, wie verfahren werden muss, wenn eine Methode, die eine Ausnahme werfen kann, aufgerufen wird.

Selbsttestaufgabe 28.2-1:

Ergänzen Sie die Methode `legeMehrwertsteuerFest()` so, dass im Falle eines negativen Werts eine `IllegalArgumentException` mit einer passenden Fehlermeldung geworfen wird.

```

void legeMehrwertsteuerFest(double neueMwSt) {
    this.mehrwertsteuer = neueMwSt;
}

```



28.3 Ausnahmen behandeln und weiterreichen

Im Folgenden wollen wir uns nun damit beschäftigen, wie mit geworfenen Ausnahmen umgegangen wird. Nehmen wir an, wir erzeugen eine neue Rechnung und wollen anschließend einen Rabatt festlegen. Da die Methode eine Ausnahme werfen kann, müssen wir diese behandeln. Dies geschieht mit Hilfe der `try`-Anweisung.

Definition 28.3-1: `try`-Anweisung

Eine `try`-Anweisung besteht aus drei verschiedenen Bestandteilen. Die kritischen Anweisungen, wie zum Beispiele Aufrufe an Methoden, die Ausnahmen erzeugen können, werden in einen `try`-Block eingeschlossen. Nach diesem muss entweder mindestens ein `catch`-Abschnitt oder ein `finally`-Block folgen.

`try`-Anweisung

`catch`
`finally`

Mit Hilfe eines `catch`-Abschnitts können im `try`-Block aufgetretene Ausnahmen gefangen und behandelt werden. Im `catch`-Abschnitt wird angegeben, für welche

Ausnahmetypen dieser Abschnitt zuständig ist. Sollen mehrere verschiedene Ausnahmetypen unterschiedliche behandelt werden, so können mehrere catch-Abschnitte folgen. Die Anweisungen im finally-Block werden immer am Ende der gesamten try-Anweisung ausgeführt. Hier werden in der Regel diverse Aufräumarbeiten erledigt. [JLS: § 14.20]

```
try {
    // Kritische Anweisungen
} catch (Ausnahmetyp1 name1) {
    // Diese Anweisungen werden ausgeführt, wenn
    // eine Ausnahme vom Typ Ausnahmetyp1 aufgetreten ist.
    // Die konkrete Ausnahme wird von der Variablen
    // name1 referenziert.
} catch (Ausnahmetyp2 name2) {
    // analog zum vorhergehenden Abschnitt
} finally {
    // Diese Anweisungen werden immer ausgeführt
}
```

Passen mehrere catch-Abschnitte zu der geworfenen Ausnahme, so wird immer nur der erste passende ausgeführt.

Abb. 28.3-1 veranschaulicht den Kontrollfluss in einer try-Anweisung.

Wir führen in unserem Beispiel für die beiden Ausnahmen jeweils eine separate Behandlung durch. Einen finally-Block benötigen wir nicht:

```
public class AusnahmeBeispiel {
    public static void main(String[] args) {
        Rechnung r = new Rechnung();
        double rabatt = 0.2;
        try {
            r.legeRabattFest(rabatt);
        } catch (NegativerRabattAusnahme nra) {
            System.out.println(nra);
        } catch (ZuHoherRabattAusnahme zhra) {
            System.out.println(zhra);
        }
    }
}
```

Bemerkung 28.3-1: Die Klasse `Error`

Error *Ein Objekt vom Typ `Error` ist im Normalfall ein sehr schwerwiegender Fehler, bei dem das Programm nur noch beendet werden kann. Beispiele dafür sind der `StackOverflowError` und der `OutOfMemoryError`, beides Unterklassen der Klasse `VirtualMachineError`. Exemplare der Klasse `Error` oder einer ihrer Unterklassen sollten somit nicht behandelt werden. Bei Ausnahmen vom Typ `Error` handelt es sich wie bei Ausnahmen vom Typ `RuntimeException` um nicht zu berücksichtigende Ausnahmen, so dass diese ebenfalls nicht mit `throws` im Methodenkopf angegeben werden müssen. [JLS: § 11.2.4]*

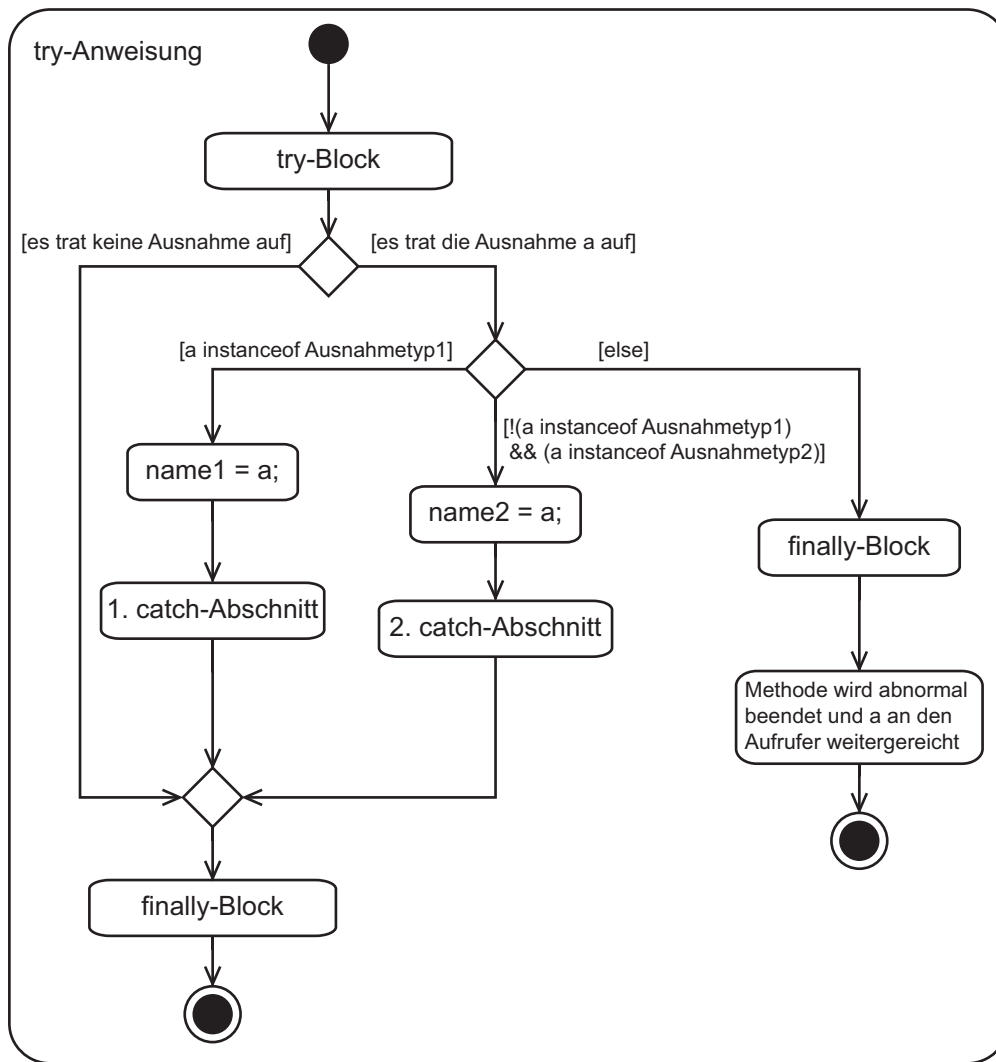


Abb. 28.3-1: Kontrollfluss in einer `try`-Anweisung

Selbsttestaufgabe 28.3-1:

Finden Sie heraus, welche Ausgaben erscheinen, wenn Sie für `rabatt` negative oder zu hohe Werte festlegen.



Selbsttestaufgabe 28.3-2:

Gegeben seien die folgenden Klassen:

```

public class MyException extends Exception {}

public class AnException extends MyException {}

public class AnotherException extends MyException {}

public class JustAnotherException extends MyException {}
  
```

Welcher der `catch`-Abschnitte wird ausgeführt, wenn der Methodenaufruf `foo()` die folgenden Ausnahmen wirft:

- a) `AnException`
- b) `JustAnotherException`
- c) `java.io.IOException`
- d) `AnotherException`
- e) `java.lang.NullPointerException`
- f) `MyException`
- g) `java.lang.ArrayIndexOutOfBoundsException`

```

try {
    foo();
} catch (AnException e) {
    // 1
} catch (JustAnotherException e) {
    // 2
} catch (MyException e) {
    // 3
} catch (RuntimeException e) {
    // 4
} catch (Exception e) {
    // 5
}

```

◇

Kann oder will eine aufrufende Methode die Ausnahme nicht behandeln, so kann sie diese auch an ihren Aufrufer weiterleiten. Dazu muss die Methode lediglich die Ausnahme ebenfalls in ihrem Methodenkopf mit Hilfe von `throws` angeben. Eine Ausnahme wird solange an die Aufrufer weitergereicht, bis eine Methode diese behandelt oder bis die `main()`-Methode erreicht wurde. In diese Falle bekommt der Benutzer die Ausnahme direkt angezeigt. Dass eine Ausnahme bis zum Benutzer gelangt, sollte in der Regel durch geeignete Behandlungen vermieden werden, da so dem Benutzer der Fehler besser erläutert werden kann.

```

public class Blumenladen {
    public void ausnahmeWeiterreichen() throws
        NegativerRabattAusnahme, ZuHoherRabattAusnahme {
        Rechnung r = new Rechnung();
        double rabatt = 0.2;
        r.setRabatt(rabatt);
    }
}

```

Ausnahmen bieten noch weitere hilfreiche Informationen. So kann mit Hilfe der Methode `getStackTrace()` an einem Ausnahmeobjekt erfragt werden, in welcher Methode es geworfen und durch welche Methoden es weitergereicht wurde. Alternativ kann auch die Methode `printStackTrace()` genutzt werden, die den gesamten Methodenstapel auf der Fehlerausgabe ausgibt.

Ausnahmen bieten die Möglichkeit, Programme robuster zu machen und auf fehlerhafte Eingaben oder andere Probleme geeignet zu reagieren. Wir können unabhängig vom Ergebnistyp Fehler anzeigen, entsprechende Informationen mitgeben und auf die Ausnahmen geeignet reagieren.

Selbsttestaufgabe 28.3-3:

Finden Sie die Fehler in der Klasse `ExceptionTest`. Warum treten diese auf und wie können Sie diese beheben?

```
public class MyException extends Exception {}

public class AnException extends Exception {}

public class AnotherException extends RuntimeException {}

public class ExceptionTest {
    public void a(int x) {
        if (x < 0) {
            throw new MyException();
        }
        if (x == 0) {
            throw AnException();
        }
    }

    public void b(String y) {
        if (y == null) {
            throw new AnotherException();
        }
    }

    public void callA() {
        a(-2);
    }

    public void callB() {
        b(null);
    }
}
```



Selbsttestaufgabe 28.3-4:

Welche Ausgaben erzeugen die Aufrufe `z(-2)`, `z(0)` und `z(7)`?

```
class AnotherExceptionTest {
    public void z(int x) {
        System.out.println(a(x));
    }
}
```

```
public int a(int x) {
    try {
        return b(x);
    } catch (RuntimeException e) {
        return -2;
    }
}

public int b(int x) {
    try {
        return c(x);
    } catch (NegativeValueException e) {
        return -1;
    }
}

public int c(int x) throws NegativeValueException,
                    IllegalArgumentException {
    if (x < 0) {
        throw new NegativeValueException();
    } else if (x > 0) {
        return x * x;
    } else {
        throw new IllegalArgumentException(
            "0 is not a valid value");
    }
}

class NegativeValueException extends Exception {
}
```



29 Dokumentation

Programmierer dokumentieren ihr Programm – wenn überhaupt – meist erst, wenn die Anwendung befriedigend läuft. Oft bleibt die Dokumentation aus einem Mangel an Zeit und Interesse äußerst rudimentär, was das Verstehen und Verändern von Programmen unnötig erschwert. Viele Details, implizite Annahmen und Entwurfsentscheidungen, die während der Programmentwicklung getroffen wurden, werden so unterschlagen.


Dokumentation

In Java-Programmen kann erläuternder Kommentar, wie in Tab. 29-1 angegeben, auf drei Weisen eingefügt werden.

Kommentar

Tab. 29-1: Kommentare

Typ	Gebrauch
// Kommentar	Alle Zeichen nach dem // bis zum Zeilenende werden ignoriert.
/* Kommentar */	Alle Zeichen zwischen /* und */ werden ignoriert.
/** Kommentar */	Vergleichbar mit /* *//, bis auf die Ausnahme, dass der Kommentar mit dem javadoc-Werkzeug verwendet werden kann, um eine automatische Dokumentation zu erstellen. Dieser Kommentar wird als Dokumentationskommentar bezeichnet.

 Kommentare sollten über das ganze Programm verteilt sein und Information enthalten, die für das Lesen und Verstehen eines Programms von Bedeutung sind.

Die Java-Umgebung enthält das Dienstprogramm `javadoc`¹⁸, das dazu dient, Referenzdokumentationen im HTML-Format aus den Java-Quelltexten zu erzeugen. Die Java-API-Referenz¹⁹, die wir im Zusammenhang mit Zeichenketten (Kapitel 26) kennen gelernt haben, ist eine auf diese Weise generierte Referenzdokumentation der Java-Standardbibliothek.

javadoc

Das `javadoc`-Dienstprogramm extrahiert die Dokumentationskommentare, die direkt vor einer Klassen-, Schnittstellen-, Konstruktor-, Methoden- oder Attributdefinition stehen. Die Dokumentationskommentare können spezielle `javadoc`-Schlüsselwörter enthalten, die alle mit dem Zeichen `@` beginnen. Tab. 29-2 enthält die wichtigsten Schlüsselwörter, die in `javadoc`-Kommentaren verwendet werden können, und eine kurze Beschreibung ihrer Bedeutung.

javadoc-Schlüsselwörter


¹⁸ <http://java.sun.com/j2se/javadoc/>

¹⁹ <http://java.sun.com/j2se/1.5.0/docs/api/>

Tab. 29-2: javadoc-Schlüsselwörter

Schlüsselwort	Beschreibung
<code>@author</code>	Wird bei Klassendefinitionen verwendet. Dieser Kommentar benennt die Autorin oder den Autor der Klasse. Für eine Klasse sind mehrere <code>@author</code> -Kommentare erlaubt.
<code>@deprecated</code>	Kann in jeder Kommentarart verwendet werden, um Programmstellen zu kennzeichnen, die zwar (wegen Rückwärtskompatibilität) existieren, jedoch nicht mehr verwendet werden sollen. I. d. R. in Kombination mit einer Empfehlung für eine Alternative.
<code>@param</code>	Wird bei Methoden- oder Konstruktordefinitionen verwendet. Dieser Kommentar benennt einen Parameter und erläutert dessen Funktion. Es sollten so viele dieser Kommentare vorhanden sein, wie die Methode oder der Konstruktor Parameter hat.
<code>@return</code>	Wird bei Methodendefinitionen verwendet, um die Art des Ergebnisses der Methode zu beschreiben.
<code>@see</code>	Wird in jeder Kommentarart verwendet. Dieser Kommentar erlaubt es der Programmiererin, eine Programmstelle zu anderen Klassen, Attributen, Methoden oder Dokumentationen mit einem Hypertext-Verweis in Beziehung zu setzen.
<code>@since</code>	Gibt an, seit welcher Versionsnummer eine Programmstelle existiert. Kann in jeder Kommentarart verwendet werden.
<code>@throws</code>	Wird bei Methoden- und Konstruktordefinitionen verwendet. Benennt Ausnahmefälle (Exceptions), die durch diese Methode erzeugt werden können.
<code>@version</code>	Wird bei Klassen verwendet, um die Versionsnummer oder ein Erstellungsdatum der Klasse anzugeben.

Viele Entwicklungsumgebungen können automatisch Kommentarzeilen und javadoc-Schlüsselwörter erstellen, wenn eine neue Klasse erzeugt wird.

 Die Dokumentation einer Klasse sollte alle nicht-privaten Elemente dieser Klasse auflisten.

Selbsttestaufgabe 29-1:

Ergänzen Sie die folgende Klasse Punkt um aussagekräftige javadoc-Kommentare. Verwenden Sie javadoc und betrachten Sie das Ergebnis in Ihrer Entwicklungsumgebung oder mit einem Webbrowser.

```
public class Punkt {
    private double x;
    private double y;
```

```

public Punkt() {
    x = 0.0;
    y = 0.0;
}

public Punkt(double xPosition, double yPosition) {
    x = xPosition;
    y = yPosition;
}

public double distanzZu(Punkt p) {
    // Math ist eine in Java eingebaute Klasse mit statischen
    // Methoden für grundlegende mathematische Operationen
    return Math.sqrt((x - p.getX()) * (x - p.getX())
        + (y - p.getY()) * (y - p.getY()));
}

public double getX() {
    return x;
}

public double getY() {
    return y;
}
}

```



Selbsttestaufgabe 29-2:

Versehen Sie die Methode `legeRabattFest()` mit einem aussagekräftigen javadoc-Kommentar:

```

public void legeRabattFest(final double neuerRabatt) throws
    NegativerRabattAusnahme, ZuHoherRabattAusnahme {
    if(neuerRabatt < 0) {
        throw new NegativerRabattAusnahme(neuerRabatt);
    }
    if (neuerRabatt > 1) {
        throw new ZuHoherRabattAusnahme(neuerRabatt);
    }
    this.rabatt = neuerRabatt;
}

```



30 Testen

Objektorientierte Programmier Techniken mögen dazu beitragen, einen besseren Programmaufbau zu erzeugen als mit herkömmlichen Programmieransätzen, sowie die Erweiterbarkeit von Programmen zu fördern. Sie können aber auch keine korrekten Programme garantieren.

Fehler
Fehlverhalten

Neben vielen anderen Qualitätskriterien für Software ist Korrektheit eine fundamentale Qualität von Programmen. Korrektheit (engl. *correctness*) bedeutet, dass es keine Widersprüche zwischen den vorgegebenen Anforderungen und dem Programmsystem gibt, das die Anforderung zu erfüllen vorgibt, und dass alle Anforderungen erfüllt sind. Umgangssprachlich wird Korrektheit auch mit der Abwesenheit von Fehlern gleichgesetzt, wobei ein Fehler (engl. *fault*) eine im Quelltext vorhandene Ursache eines Fehlverhaltens (engl. *failure*) ist. Man kann einen Fehler auch als Abweichung des Ist- vom Sollverhalten bezeichnen.

Entwurfs- und Programmfehler beruhen auf menschlichen Irrtümern. Wirkungszusammenhänge werden nicht beherrscht, Randbedingungen der Einsatzumgebung von Programmen werden übersehen, und Aufträge werden falsch in Daten- und Programmstrukturen umgesetzt.

Im harmlosesten Fall verärgern fehlerhafte Programme die Kundinnen, im schlimmsten Fall können sie Menschenleben oder hohe Vermögenswerte beeinträchtigen.

Wie andere Produkte müssen auch Programme vor ihrer Freigabe eingehend geprüft werden, um möglichst viele Fehler zu entdecken und zu beseitigen. Für die Überprüfung der Software gibt es verschiedene Techniken. Eine davon ist das Testen, womit wir uns im Rahmen dieses Kapitels beschäftigen wollen.

statischer Test

Es gibt sowohl statische als auch dynamische Tests. Bei einem statischen Test wird das Programm nicht ausgeführt, sondern einer Analyse unterzogen. Auch hier gibt es wieder eine Vielzahl von Techniken und Verfahren, eine davon ist auch das Übersetzen des Programms, wodurch es auf syntaktische Fehler untersucht wird. Alternativ kann der Quelltext auch inspiziert werden.

dynamischer Test

Bei einem dynamischen Test wird das Programm mit vorher festgelegten Eingabedaten ausgeführt und anschließend das tatsächliche Verhalten mit dem erwarteten Verhalten verglichen.

Definition 30-1: Dynamisches Testen

Dynamisches Testen bedeutet, ein Programm(-teil) stichprobenartig auszuführen, um Unterschiede zwischen seinem erwarteten und tatsächlichen Verhalten zu finden.

Bei der Verifikation wird im Gegensatz zum Testen mit mathematischen Verfahren bewiesen, dass der Quelltext mit dem formalen Modell oder der formalen Spezifikation übereinstimmt.

Verifikation

Bemerkung 30-1: Testen gegen Verifizieren

Ein Test kann im Allgemeinen keine vollständigen Aussagen über die Korrektheit eines Programms liefern. Eine Ausnahme bildet das erschöpfende Testen, bei dem alle möglichen Testfälle durchgespielt werden. Dies ist aber in den wenigsten Anwendungsfällen möglich. In der Regel kann Testen nur als Stichprobenverfahren eingesetzt werden. Ein Test kann nur die Anwesenheit von Fehler zeigen und nicht ihre Abwesenheit.

Ein Test ist demnach erfolgreich, wenn Fehler gefunden werden. Ein Test, der keinen Unterschied zwischen erwartetem und tatsächlichem Verhalten aufdeckt, bietet keine neuen Informationen.

Verifizierende Verfahren können allgemeine Aussagen über Korrektheit und andere Programmeigenschaften erzeugen. Sie sind für realistische Anwendungen aber meist nicht anwendbar, weil die formale Spezifikation der Anforderungen zu zeitaufwändig wäre oder weil die Komplexität der mathematischen Verfahren Grenzen setzt.

└

Testtechniken haben für die IT-Industrie eine hohe praktische Bedeutung. Aufgrund mangelnder Disziplin werden Tests aber oft nur gelegentlich und nicht gründlich genug durchgeführt. Viel zu häufig wird erst kurz vor der Auslieferung der Software an die Auftraggeber getestet, und bei Terminproblemen wird meist beim Testen gespart. Qualitätssicherung wird so zu einem nachgeordneten Prozess. Mangelnde Softwarequalität, unzufriedene Kundinnen und entsprechender Nachbesserungsaufwand sind die Folge. Dies verringert nicht nur die Produktivität und Rentabilität der Programmentwicklung, sondern erhöht auch das Risiko von Gewährleistungs- und Haftungsansprüchen.

Wir werden im Folgenden zunächst auf verschiedene Arten des Testens und allgemeine Testverfahren eingehen. Wir verengen dann die Perspektive auf das Testen von Klassen und zeigen dann, wie man solche Tests mit Hilfe des Testwerkzeugs JUnit automatisieren und wiederholbar machen kann.

30.1 Teststufen und Testarten

Neben der Unterscheidung in statische und dynamische Test können wir auch zwischen verschiedenen Teststufen wie Komponententest, Integrationstest, Systemtest, Abnahmetest und Regressionstest differenzieren.

Teststufen

Beim Komponententest wird eine kleine, abgeschlossene Einheit, zum Beispiel eine Klasse, getestet. Deshalb wird dieser Test auch Modul- oder Klassentest genannt.

Komponententest
Modultest
Klassentest

Mit dieser Art von Test werden wir uns im weiteren Verlauf des Kapitels noch intensiver beschäftigen.

Integrationstest Das Zusammenspiel der vorher getesteten Komponenten wird mit Hilfe des Integrationstests überprüft. Hierbei können zum Beispiel Fehler in Schnittstellen gefunden werden.

Systemtest Ob wirklich das gesamte System die Anforderungen erfüllt und alle Elemente richtig zusammenarbeiten, wird mit Hilfe des Systemtests untersucht.

Abnahmetest Die bisherigen Tests werden in der Regel vom Hersteller durchgeführt. Der Abnahmetest hingegen wird im Beisein des Kunden oder von ihm selbst durchgeführt. Hierbei wird auch oft die Akzeptanz der Benutzer analysiert.

Regressionstest In der Regel wird Software nicht nur einmal entwickelt, sondern später erweitert, angepasst oder es werden später aufgedeckte Fehler behoben. Da sich bei der Überarbeitung und Veränderung des Quelltextes auch wieder neue Fehler einschleichen können, sollten nach jeder Änderung sogenannte Regressionstests durchgeführt werden.

funktionale Anforderungen
nicht funktionale Anforderungen Ergänzend zu den Teststufen können auch noch verschiedene Arten von Tests zum Einsatz kommen. So kann man sowohl funktionale als auch nicht funktionale Anforderungen testen. Dabei beschreiben funktionale Anforderungen das erwünschte sichtbare Verhalten des Systems. Nicht funktionale Anforderungen beschreiben, wie gut und mit welcher Qualität dieses Verhalten erreicht werden soll. Zu nicht funktionalen Anforderungen zählen beispielsweise Effizienz, Zuverlässigkeit und Benutzbarkeit. Im Rahmen des Kurses werden wir uns auf den funktionalen Test konzentrieren.

struktureller Test
Whitebox-Verfahren
Blackbox-Verfahren Außerdem kann beim Testen auch dahingehend unterschieden werden, auf welcher Basis die Testfälle ausgewählt werden. So kann die Struktur des Quelltextes berücksichtigt werden, in diesem Fall spricht man von einem strukturellen Test oder Whitebox-Verfahren. Wird die Struktur nicht berücksichtigt, sondern lediglich die Spezifikation oder Anforderungen, so spricht man von Blackbox-Verfahren. Auf diese beiden Arten werden wir später im Kapitel noch zurückkommen.

30.2 Testplanung und -durchführung

Ebenso wie beim Entwurf und der Entwicklung von Programmen sollte auch beim Testen systematisch vorgegangen werden. Wichtig ist, dass Tests wiederholbar sind, so dass, wenn ein gefundener Fehler behoben wurde, der Test erneut durchgeführt werden kann.

Testplan
Teststrategie Vor Beginn des Tests muss deshalb ein Testplan inklusive Teststrategie erstellt werden. Da wir nur stichprobenartig testen können, muss in der Teststrategie festgelegt werden, welche Bestandteile der Software wie umfangreich und mit welchen Methoden getestet werden sollen. Ein paar Methoden werden wir später im Kapitel an

Hand des Klassentests kennen lernen. Ein weiterer wichtiger Bestandteil des Testplans ist die Testspezifikation, die die Informationen zu den einzelnen Testfällen beinhaltet. Dabei machen wir für jeden Testfall die folgenden Angaben:

Testspezifikation
Testfall

- Eingabedaten,
- Vorbedingungen und
- das erwartete Verhalten.

Eingabedaten dokumentieren die von uns benutzten Eingabewerte. Die Vorbedingungen dokumentieren den Zustand des an der Durchführung dieses Testfalls beteiligten Systems oder der Klasse. Das erwartete Verhalten beschreibt die Ausgaben und sonstiges Verhalten, das wir in Kenntnis der Spezifikation der Methode erwarten.

Bei Testfällen unterscheiden wir zwischen positiven und negativen Tests. Beim positiven Test überprüfen wir, ob das Programm seine Spezifikation erfüllt. Positive Tests werden mit gültigen Eingaben durchgeführt. Negative Tests benutzen ungültige Eingabedaten, um herauszufinden, wie sich das Programm bei solchen Eingaben verhält, oder um schwierige Fälle im Programm zu identifizieren.


positiver Test
negativer Test

Die Testdurchführung wird in Form eines Testprotokolls dokumentiert. Es enthält Informationen darüber, welcher Testfall mit welchen Ergebnissen durchgeführt wurde. Nach der Ausführung muss überprüft werden, ob es Abweichungen vom erwarteten Verhalten gibt. Im Falle einer Abweichung muss dann die Ursache für den Fehler gefunden und später behoben werden.

Testdurchführung
Testprotokoll

30.3 Klassen dynamisch testen

Beim dynamischen Testen von Klassen wird jede Methode der Klasse systematisch im Bezug auf ihre Spezifikation stichprobenartig untersucht. Die Spezifikation kann, wie wir an vielen Programmbeispielen und Selbsttestaufgaben sehen konnten, informell vorgegeben sein; sie kann aber auch mit mathematischer Präzision formuliert sein.

 Wichtig beim Testen ist, dass gegen die Spezifikation und nicht gegen die Implementierung getestet wird.

Das Testen von Klassen sollte folgende Schritte umfassen:

- Neue Objekte mit Hilfe der verfügbaren Konstruktoren erzeugen und initialisieren,
- überprüfen, ob die erzeugten Objekte den erwarteten Zustand aufweisen,
- jede Methode mit geeigneten Argumenten sowie
- kritische Kombinationen von Objektzustand und Methodenaufruf testen und
- mögliche Ausnahmesituationen überprüfen.

Da es im Allgemeinen nicht möglich ist, alle denkbaren Kombinationen von Objektzuständen und Methodenaufrufen zu testen, müssen wir repräsentative Testfälle zur Überprüfung von Objektzuständen aussuchen.

Die Methoden und Konstruktoren einer Klasse sollten sowohl positiv als auch negativ getestet werden.

Wir wollen nun unsere Klasse `Rechnung` so anpassen, dass der Rabatt automatisch bestimmt wird. Dabei soll bei Premiumkundinnen immer ein Rabatt von 5 Prozent gewährt werden. Ab einem Rechnungsbetrag von 100 Euro wird allen Kunden ein Rabatt von 5 Prozent gewährt. Ab 200 Euro erhalten Premiumkundinnen 10 Prozent Rabatt. Die Klasse `Rechnung` benötigt somit kein Attribut `rabatt` mehr. Dafür kommt eine neue Methode `bestimmeRabatt()` hinzu. Da wir nur ganze Prozentwerte zulassen, besitzt die Methode den Ergebnistyp `int`. Zudem fügen wir die Klasse `Premiumkunde` als Unterklasse von `Kunde` hinzu. Die Klasse `Kunde` wird außerdem um eine Methode `istPremiumkunde()` ergänzt.

Im Folgenden wollen wir uns nun mit dem Test der Klasse `Rechnung` beschäftigen. Dabei gehen wir davon aus, dass die Klasse `Kunde` anderweitig getestet wird.

Gegeben seien die folgenden Auszüge aus den Implementierungen der Klassen `Rechnung`, `Kunde` und `Premiumkunde`. Die Methodenrumpfe sind zur Übersicht leer gelassen. Zur Vereinfachung gibt es auch die Methode `legeBetragFest()`, um direkt den Betrag festlegen zu können.

```
public class Kunde {

    /**
     * erzeugt einen neuen Kunden mit gegebenen Namen
     * @param name der Name des Kunden
     */
    public Kunde(String name) {
        // ...
    }

    /**
     * liefert Informationen, ob es sich um einen Premiumkunden
     * handelt
     * @return true, wenn es sich um einen Premiumkunden handelt,
     * ansonsten false
     */
    public boolean istPremiumkunde() {
        // ...
    }

    // ...
}
```

```
public class Premiumkunde extends Kunde {

    /**
     * erzeugt einen neuen Premiumkunden mit gegebenem Namen
     * @param name der Name des Premiumkunden
     */
    public Premiumkunde(String name) {
        // ...
    }

    // ...
}

public class Rechnung {

    /**
     * erzeugt eine neue Rechnung für den Kunden
     * @param k der rechnungsempfänger
     */
    public Rechnung(Kunde k) {
        // ...
    }

    /**
     * legt den Rechnungsbetrag fest
     * @param betrag der Rechnungsbetrag
     */
    public void legeBetragFest(double betrag) {
        // ...
    }

    /**
     * berechnet an Hand des Kunden und des Rechnungsbetrages den
     * zu gewaehrenden Rabatt
     * @return liefert den Rabatt in Prozent zurueck
     */
    public int bestimmeRabatt() {
        // ...
    }

    // ...
}
```

Um unsere Klasse Rechnung zu testen, könnten wir ein kleines Java-Programm schreiben, dass verschiedene Objekte erzeugt und daran Methoden aufruft und entsprechende Ausgaben mit dem erwarteten und tatsächlichen Verhalten erzeugt. Dabei stellt sich schnell heraus, dass solche Tests aufwändig und schwer handhabbar sind. Deshalb werden wir im folgenden Abschnitt JUnit kennen lernen.

Selbsttestaufgabe 30.3-1:

Überlegen Sie sich mögliche Testfälle, insbesondere auch negative Tests, für die Klasse Rechnung.



Selbsttestaufgabe 30.3-2:

Versuchen Sie das folgende Testprogramm für die Klasse Rechnung weiterzuentwickeln. Implementieren Sie mindestens zwei positive und zwei negative Tests. Gehen Sie davon aus, dass bei einem negativen Test eine Ausnahme auftreten sollte.

```
public class EinfacherRechnungstest {
    public static void main(String[] args) {
        Kunde k = new Kunde("Anna Meier");
        Rechnung r = new Rechnung(k);
        r.legeBetragFest(120);
        int rabatt = r.bestimmeRabatt();
        if (rabatt != 5) {
            System.out.println("Der Rabatt bei einer Rechnung "
                + "von 120 Euro wird falsch berechnet!");
            System.out.println("Der Rabatt sollte 5 Prozent "
                + "betragen, betraegt aber: " + rabatt);
        }
        // ...
    }
}
```



30.4 Testen mit JUnit

JUnit JUnit ist ein Rahmenwerk und Testwerkzeug, das die Erstellung von Testfällen, das Ausführen und Wiederholen von Tests sowie die Auswertung von Testfällen systematisch unterstützt. JUnit wurde von Kent Beck und Erich Gamma 1998 auf der Grundlage von SUnit²⁰, einer Testumgebung für Smalltalk, entwickelt. Die Grundidee, die auch in JUnit einging, stammt von Kent Beck, der 1994 das Konzept der Testmuster und die zugehörige Testumgebung entwickelte²¹.

JUnit besteht aus einer Reihe von Klassen, die es erlauben, Werte und Bedingungen zu testen, die jeweils erfüllt sein müssen, damit ein Test erfolgreich ist. JUnit wird ergänzt um programmtechnische Hilfsmittel zur Organisation von Testfällen in Testfolgen (engl. *test suite*), und es enthält Werkzeuge, um Tests ablaufen zu lassen und zu protokollieren. Das Ziel von JUnit ist es, Testabläufe soweit wie möglich zu automatisieren und sie möglichst ohne manuellen Eingriff wiederholen zu können.

²⁰ <http://sunit.sourceforge.net/>

²¹ <http://www.xprogramming.com/testfram.htm>

Alle so erfassten Tests können in einem Testlauf ausgeführt werden. JUnit unterstützt den Ansatz, die Entwicklung einer Testumgebung eng mit der Programmentwicklung zu verzahnen.

Die meisten Entwicklungsumgebungen unterstützen heutzutage die Ausführung von JUnit-Tests. Aber auch ohne Entwicklungsumgebung lässt sich JUnit nutzen. Erläuterungen, wie Sie JUnit in verschiedenen Umgebungen nutzen können, finden Sie auf der Webseite zum Kurs.

In JUnit werden in der Regel zu jeder Klasse eine oder mehrere öffentliche Testfallklassen entwickelt. Diese Klassen sind Unterklassen von `junit.framework.TestCase`.

`junit.framework.
TestCase`

Eine Testklasse für die Klasse Rechnung könnte folgendermaßen aussehen:

```
public class Rechnungstest extends junit.framework.TestCase {
    // ...
}
```

Ein konkreter Testfall wird in der Regel in Form einer einzelnen Methode implementiert. Eine solche Methode muss vier Eigenschaften aufweisen. Die Methode

Eigenschaften einer
Testmethode

- muss öffentlich (`public`) sein,
- darf kein Ergebnis liefern (`void`),
- darf keine Parameter erwarten und
- der Methodenname muss mit `test` beginnen.

Um nun zu überprüfen, ob bei einer neu erzeugten Rechnung für eine normale Kundin mit einem Betrag von 0 Euro auch ein Rabatt von 0 Prozent bestimmt wird, könnten wir die Klasse `Rechnungstest` um die Methode `testeRabattBeiNeuerRechnung()` ergänzen.

```
public class Rechnungstest extends junit.framework.TestCase {
    public void testeRabattBeiNeuerRechnung() {
        Kunde k = new Kunde("Anna Meier");
        Rechnung r = new Rechnung(k);
        // TODO: Rabatt überprüfen
    }
}
```

JUnit bietet für die Überprüfung von Bedingungen sogenannte `assert`-Methoden an. Diese können beispielsweise prüfen, ob ein tatsächlicher Wert mit einem erwarteten übereinstimmt. So prüft die Methode `assertEquals(int erwartet, int tatsaechlich)`, ob die beiden Werte übereinstimmen. Ist dies nicht der Fall, so schlägt die umgebende Testmethode fehl.

`assert`-Methoden

`assertEquals()`

```

public class Rechnungstest extends junit.framework.TestCase {
    public void testeRabattBeiNeuerRechnung() {
        Kunde k = new Kunde("Anna Meier");
        Rechnung r = new Rechnung(k);
        assertEquals(0, r.bestimmeRabatt());
    }
}

```

Es gibt noch eine Reihe von weiteren `assert`-Methoden. Eine Übersicht dazu finden Sie in Tab. 30.4-1.

Tab. 30.4-1: `assert`-Methoden


Methode	Verwendung
<code>assertTrue(boolescherAusdruck)</code>	sichert zu, dass der Ausdruck <code>true</code> ist
<code>assertFalse(boolescherAusdruck)</code>	sichert zu, dass der Ausdruck <code>false</code> ist
<code>assertEquals(erwarteter, tatsaechlicherWert)</code>	sichert zu, dass zwei Werte oder Objekte gleich sind
<code>assertEquals(erwartet, tatsaechlich, toleranz)</code>	sichert zu, dass zwei Gleitkommazahlen im Rahmen der angegebenen Toleranz gleich sind: $ \text{erwartet} - \text{tatsaechlich} < \text{toleranz}$
<code>assertNull(objektVerweis)</code>	sichert zu, dass <code>objektVerweis</code> <code>null</code> ist
<code>assertNotNull(objektVerweis)</code>	sichert zu, dass <code>objektVerweis</code> nicht <code>null</code> ist
<code>assertSame(objekt1, objekt2)</code>	sichert zu, dass zwei Objektverweise auf dasselbe Objekt verweisen
<code>assertNotSame(objekt1, objekt2)</code>	sichert zu, dass die beiden Objektverweise auf verschiedene Objekte verweisen

☞ Beachten Sie, dass bei der Berechnung von Gleitkommawerten Rundungsfehler entstehen können. Deshalb ist ein Vergleich mit `assertEquals()` in diesem Fall häufig nicht hilfreich. Benutzen Sie in solchen Fällen die entsprechende `assertEquals()`-Methode für Gleitkommazahlen, die noch zusätzlich eine Toleranz berücksichtigt.

Allen `assert`-Methoden kann zusätzlich als erstes Argument ein `String`-Objekt mitgegeben werden, um im Falle eines Fehlschlages noch zusätzliche Informationen zu übermitteln.


```
public class Rechnungstest extends junit.framework.TestCase {
    public void testeRabattBeiNeuerRechnung() {
        Kunde k = new Kunde("Anna Meier");
        Rechnung r = new Rechnung(k);
        assertEquals("Rabatt bei 0 Euro und normaler Kundin "
            + "fehlerhaft.", 0, r.bestimmeRabatt());
    }
}
```

Wird eine Testklasse als JUnit-Test ausgeführt, so werden alle Testmethoden, die die vier oben genannten Eigenschaften aufweisen, ausgeführt und alle `assert`-Aufrufe ausgewertet. Schlägt eine `assert`-Methode fehl, so wird die Ausführung der Testmethode sofort beendet und diese als fehlgeschlagen markiert. Tritt bei der Ausführung der Methode eine Ausnahme auf, so wird vermerkt, dass dort eine Ausnahme auftrat. Es wird zwischen fehlgeschlagenen `assert`-Methoden und Ausnahmen unterschieden. Tritt keines von beidem ein, so wird die Testmethode als bestanden vermerkt.

 Beachten Sie, dass Ihre Testmethoden nicht zu umfangreich werden und immer eine Facette testen. Teilen Sie längere Testmethoden lieber in mehrere einzelne auf.

 Kommentieren Sie Ihre Tests, so dass diese auch für andere verständlich sind.

Zusätzlich zu den Testmethoden kann eine Testklasse noch weitere besondere Methoden aufweisen. Oft ist es nötig, vor der Ausführung einer Testmethode bestimmte Objekte zu erzeugen und zu initialisieren oder nach einem Test bestimmte Aufräumarbeiten auszuführen. Für solche Aufgaben bietet die Klasse `TestCase` die Methoden `setUp()` und `tearDown()`, die in der konkreten Testklasse überschrieben werden können. Wenn wir das Erzeugen des Rechnungsobjekts in die `setUp()`-Methode auslagern, sähe die Testklasse folgendermaßen aus.

`setUp()`
`tearDown()`

```
public class Rechnungstest extends junit.framework.TestCase {
    Rechnung r;

    public void setUp() {
        Kunde k = new Kunde("Anna Meier");
        r = new Rechnung(k);
    }

    public void testeRabattBeiNeuerRechnung() {
        assertEquals("Rabatt bei 0 Euro und normaler Kundin "
            + "fehlerhaft.", 0, r.bestimmeRabatt());
    }
}
```

Die JUnit-Laufzeitumgebung sorgt automatisch dafür, dass die `setUp()`-Methode direkt vor jeder Testmethode und die `tearDown()`-Methode direkt im Anschluss ausgeführt wird.

Bemerkung 30.4-1: Typisches Vorgehen beim Testen von Klassen

Jede Methode einer Klasse sollte zunächst unabhängig von anderen getestet werden. Nachdem alle Methoden einzeln überprüft wurden, sollten kritische Pfade über mehrere Methoden hinweg getestet werden.

Beim Testen dürfen wir nicht vergessen, daran zu denken, dass Objekte Zustände haben. Wir müssen deshalb auch testen, ob kritische Zustände von Attributen richtig behandelt werden.

┘

Nachdem wir nun die technischen Hilfsmittel zur Erstellung und Durchführung von Tests kennen gelernt haben, fehlen uns noch Verfahren um unsere Klassen systematisch testen zu können. Verfahren und Techniken zur Erzeugung von Testfällen lernen wir im folgenden Abschnitt kennen.

Selbsttestaufgabe 30.4-1:

Ergänzen Sie die Klasse `Rechnungstest` um eine Methode, die den Rabatt bei einer neuen Rechnung für eine Premiumkundin überprüft.

◇

Selbsttestaufgabe 30.4-2:

Gegeben sei die folgende Klasse und die dazugehörige Testklasse. Versuchen Sie herauszufinden, welche Testmethoden aus welchen Gründen fehlschlagen.

```
public class MeinRechner {

    public int addiere(int x, int y) {
        return x + y;
    }

    public int subtrahiere(int x, int y) {
        return x - y;
    }

    public int multipliziere(int x, int y) {
        return x * y;
    }

    public int dividiere(int x, int y) {
        return x / y;
    }

    public int berechneBetrag(int x) {
        if (x < 0) {
            return -x;
        }
        return x;
    }
}
```

```
public class MeinRechnerTest extends junit.framework.TestCase {

    MeinRechner mr;

    public void setUp() {
        mr = new MeinRechner();
    }

    public void testAddition() {
        assertEquals(4, mr.addiere(2,2));
        assertTrue(3 > mr.addiere(1,2));
    }

    public void testSubtraktion() {
        assertFalse(1 < mr.subtrahiere(2,2));
        assertTrue(4 == mr.subtrahiere(6,2));
    }

    public void testMultiplikation() {
        assertEquals(8, mr.multipliziere(6,2));
    }

    public void testDivision() {
        assertEquals(10, mr.dividiere(30,3));
        assertEquals(0, mr.dividiere(5,0));
    }

    public void testBetrag() {
        assertEquals(2, mr.berechneBetrag(-2));
        assertTrue(0 == mr.berechneBetrag(0));
    }

}
```



30.5 Testfälle identifizieren

Testfälle können entweder aus der Spezifikation eines Programms oder aus der Struktur der Implementierung abgeleitet werden. Zunächst werden wir uns mit dem Ableiten von Testfällen aus der Spezifikation beschäftigen, dem sogenannten Blackbox-Testen.

Beim Blackbox-Testen werden die möglichen Werte für die verschiedenen Parameter häufig in Äquivalenzklassen eingeteilt. Das Verhalten der Methode oder des Konstruktors muss für alle Werte einer Äquivalenzklasse identisch sein. So könnten wir den Rechnungsbetrag und die Kunden für die Rabattberechnung in die folgenden Äquivalenzklassen einteilen:

Blackbox-Testen
Äquivalenzklassen

Betrag:	negativ	$0 \leq \text{betrag} < 100$	$100 \leq \text{betrag} < 200$
Kunde:	Premiumkunde	kein Premiumkunde	null

Basierend auf den ungültigen Werten können negative Tests festgelegt werden. In diesem Fall ein negativer Rechnungsbetrag oder ein null-Verweis als Kunde.

Grenzwertanalyse

Nach der Einteilung in Äquivalenzklassen müssen passende Repräsentanten ermittelt werden. Hierbei wird häufig noch eine Grenzwertanalyse durchgeführt. Fehlerzustände treten häufig an den Grenzen zwischen Äquivalenzklassen auf, so dass gerade diese Bereiche möglichst gut getestet werden sollten. Solche Fehler entstehen beispielsweise durch falsche Vergleichsoperatoren. Beim Rechnungsbetrag wären -1, 0, 1, 99, 100, 101 sowie 199, 200 und 201 typische Grenzwerte. Ergänzend dazu sollten aber auch Repräsentanten aus den mittleren Bereichen der Äquivalenzklassen ausgewählt werden.

Selbsttestaufgabe 30.5-1:

Implementieren Sie die positiven Testfälle für die Rabattberechnung, die sie durch Äquivalenzklassenbildung und Grenzwertanalyse identifiziert haben, in JUnit. ◇

Selbsttestaufgabe 30.5-2:

Bestimmen Sie für die folgende Beschreibung die Äquivalenzklassen, wählen Sie mit Hilfe der Grenzwertanalyse Repräsentanten aus und ergänzen Sie diese mit Werten aus den mittleren Bereichen.

Bei einem Autoverleih gelten die folgenden Preise: Bei einer Mietdauer von weniger als 14 Tagen werden pro Tag 20 Euro berechnet. Zusätzlich wird bis einschließlich 200 km eine Pauschale von 40 Euro fällig. Bei mehr als 200 km werden zusätzlich zur Pauschale pro weiterem Kilometer 15 Cent berechnet. Ab 14 Tagen Mietdauer wird eine Pauschale von 300 Euro fällig und 12 Cent pro gefahrenem Kilometer. ◇

Das Gegenteil zum Blackbox-Test ist der sogenannte Whitebox-Test. Bei diesem wird die Struktur des Quelltextes betrachtet. Wenn wir die Implementierung unserer Rabattberechnung betrachten, können wir daraus ein zugehöriges Aktivitätsdiagramm erstellen und somit den Kontrollfluss veranschaulichen.

```
public class Rechnung {
    private Kunde empfaenger;
    private double betrag;

    // ...

    public int bestimmeRabatt() {
        if (empfaenger.istPremiumkunde()) {
            if (betrag >= 200) {
```

```

        return 10;
    } else {
        return 5;
    }
} else {
    if (betrag >= 100) {
        return 5;
    } else {
        return 0;
    }
}
}
}

```

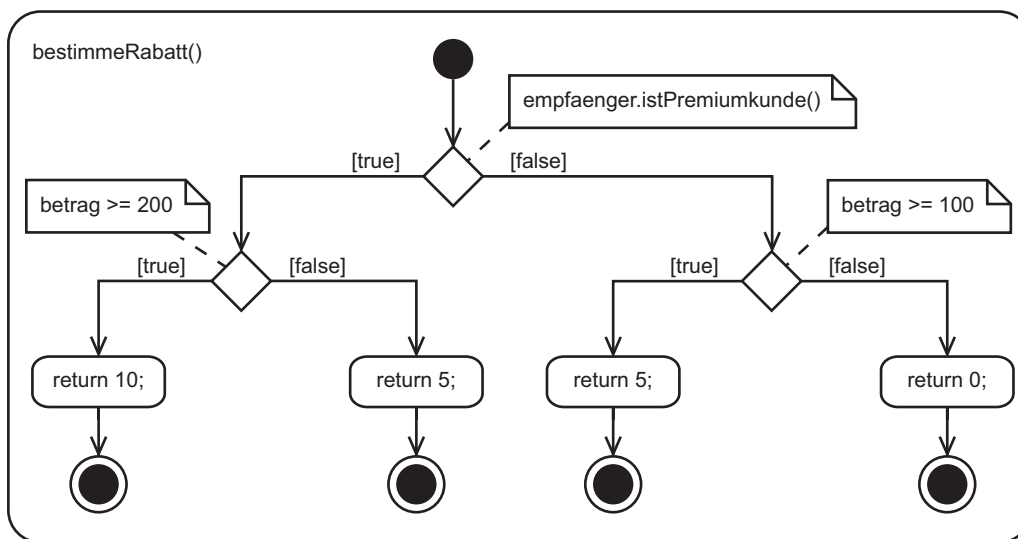


Abb. 30.5-1: Aktivitätsdiagramm Rabattberechnung

Basierend auf dem Kontrollfluss können verschiedene Überdeckungsmaße definiert werden um zu messen, wie viel des Quelltextes durch die Tests abgedeckt werden.

Relativ einfach zu erreichen ist eine vollständige Anweisungsüberdeckung, bei der lediglich gefordert wird, dass jede Anweisung mindestens einmal ausgeführt wird. Natürlich kann auch eine Anweisungsüberdeckung prozentual gefordert werden.

Anweisungs-
überdeckung

Für unsere Methode würden die folgenden Testfälle eine vollständige Anweisungsüberdeckung liefern.

- Premiumkunde und ein Betrag von 300
- Premiumkunde und ein Betrag von 20
- kein Premiumkunde und ein Betrag von 30
- kein Premiumkunde und ein Betrag von 150

Selbsttestaufgabe 30.5-3:

Erstellen Sie ein Aktivitätsdiagramm für die folgende Methode und versuchen Sie Eingabewerte zu bestimmen, so dass Sie eine vollständige Anweisungsüberdeckung erreichen. Implementieren Sie anschließend einen passenden JUnit-Test.

```
public class Rechner {
    public int berechne(int x, int y) {
        int erg = 0;
        if (x < 0) {
            x *= -1;
        }
        if (y < 0) {
            y *= -1;
        }
        erg = x * y;
        return erg;
    }
}
```



Zweigüberdeckung Bei der Zweigüberdeckung wird gemessen, wie viele der Kanten des Aktivitätsdiagramms ausgeführt wurden.

Selbsttestaufgabe 30.5-4:

Versuchen Sie zu der Methode aus Selbsttestaufgabe 30.5-3 Testfälle zu identifizieren, so dass Sie eine vollständige Zweigüberdeckung erhalten und ergänzen Sie den JUnit-Test um eine geeignete Methode.



Pfadüberdeckung Die Pfadüberdeckung hingegen misst, wie viele der möglichen Pfade durch das Aktivitätsdiagramm abgedeckt sind. Problematisch wird eine vollständige Pfadüberdeckung, wenn der Quelltext Schleifen enthält, da dann sehr viele, manchmal sogar unendlich viele Pfade existieren.

Selbsttestaufgabe 30.5-5:

Versuchen Sie, zu der Methode aus Selbsttestaufgabe 30.5-3 Testfälle zu identifizieren, so dass Sie eine vollständige Pfadüberdeckung erhalten und ergänzen Sie den JUnit-Test um eine geeignete Methode.



Alternativ zu den Überdeckungsmaßen, die auf dem Kontrollfluss beruhen, können auch Überdeckungsmaße auf Bedingungen definiert werden. So kann gefordert werden, dass jede Bedingung mindestens einmal zu `true` und einmal zu `false` ausgewertet wird. Gerade bei komplexeren Bedingungen können auch noch Teilausdrücke mitberücksichtigt werden.

Selbsttestaufgabe 30.5-6:

Gegeben sei die folgende Methode zur Berechnung eines Durchschnitts von Feldelementen.

```
public class Durchschnittsberechner {  
  
    public double berechneDurchschnitt(int[] feld) {  
        long erg = 0;  
        for (int i : feld ) {  
            erg += i;  
        }  
        return erg / feld.length;  
    }  
}
```

Spezifizieren Sie mindestens fünf Testfälle und implementieren Sie diese in Form von JUnit-Testfällen. Decken Ihre Tests Fehler auf und wenn ja welche?



31 Umgang mit Fehlern

Das Testen von Programmen hilft uns zu bemerken, dass Fehler existieren. Es verrät uns in der Regel nicht, warum und an welcher Stelle Fehler entstanden sind. In diesem Kapitel betrachten wir einige besonders häufige Fehler, befassen uns mit der systematischen Lokalisierung von Fehlern und geben Hinweise zur Vermeidung von Fehlern.

31.1 Häufige Programmierfehler

häufige Fehler Im Folgenden listen wir eine Reihe häufig auftretender Programmierfehler auf. Falls Ihr Programm nicht übersetzt wird, nicht arbeitet, nicht endet oder andere als die erwarteten Ergebnisse liefert, können Sie Ihren Quelltext überprüfen und herausfinden, ob:

- Sie vergessen haben, eine Klasse oder ein Paket zu importieren.
- Sie Variablen-, Klassen- oder Methodennamen falsch geschrieben haben.
- Sie vergessen haben, Anweisungsblöcke mit geschweiften Klammern zu umschließen.
- Sie vergessen haben, Anweisungen mit einem Semikolon abzuschließen.
- Sie versehentlich ein zusätzliches Semikolon gesetzt haben und damit eine Anweisung vorzeitig beenden wie in

```
for (int i = 0; i < LOOPS; i++); {
    doSomething();
}
```

- Sie den Zuweisungsoperator = verwendet haben, obwohl Sie eigentlich den Vergleichsoperator == benutzen wollten, wie in

```
if (b = true) {...} // wird vom Übersetzer akzeptiert
```

Der Übersetzer akzeptiert diese Zuweisung, so dass der Fehler zunächst unbemerkt bleiben kann. Wenn Sie sich angewöhnen, bei Vergleichen die Konstante voranzustellen, werden Sie bereits beim ersten Übersetzungsversuch auf Ihren Fehler aufmerksam:

```
if (true = b) {...} // wird vom Übersetzer nicht
// akzeptiert
```

- Sie Gleitkommazahlen vergleichen, indem Sie == oder != nutzen; da Gleitkommazahlen Annäherungen sind, ist es sicherer, in Bedingungen die Vergleichsoperatoren > und < zu verwenden.
- Sie Zeichenketten vergleichen, indem Sie == nutzen; für einen sicheren Vergleich von Zeichenketten sollten Sie die equals()-Methode verwenden.
- Sie Klammern in arithmetischen Ausdrücken vergessen haben.

- Sie versehentlich eine Ganzzahl-Division durchführen wie in

```
int a = 1;
int b = 2;
double durchschnitt = (a + b) / 2; // ergibt 1.0
```

- Sie vergessen haben, innerhalb einer Schleife die Schleifenvariable zu verändern wie in

```
while (zaehler < 100) {
    doSomething();
}
```

- Sie die Schleifenvariable einer anderen (z. B. umgebenden) Schleife ungewollt verändern, wie in

```
for (int i = 0; i < LOOPS; i++) {
    for (int j = 0; j < LOOPS; i++) {
        ...
    }
}
```

- Sie eine do-while-Schleife verwenden und nicht daran gedacht haben, dass eine do-while-Schleife mindestens einmal ausgeführt wird.
- Sie vergessen haben, break nach einer bestimmten Anweisung zu gebrauchen.
- Sie die Bedeutung von break und continue verwechselt haben.
- Sie vergessen haben, ein Feld zu initialisieren wie in

```
String[] namen;
namen[3] = "Kraemer";
```

- Sie versehentlich nicht existierende Feldelemente adressieren wie in

```
String[] namen = new String[3];
namen[3] = "Kraemer";
```

- Sie unbeabsichtigt eine Methode einer Oberklasse überschreiben.
- Sie, statt eine Methode einer Oberklasse zu überschreiben, eine neue Methode mit abweichender Signatur deklarieren.

☞ Um zu kennzeichnen, dass eine Methode in der Unterklasse eine Methode der Oberklasse überschreibt, kann eine @Override-Annotation an die Methode geheftet werden.

```
@Override
public boolean equals(Object obj) {
    // ...
}
```

Wenn der Übersetzer diese Annotation vorfindet, überprüft er automatisch, ob die Methode tatsächlich eine andere überschreibt. Ist dies nicht der Fall, wird ein Übersetzungsfehler erzeugt. Dies hilft zum Beispiel, Inkonsistenzen bei der Änderung von Methodennamen oder -signaturen aufzudecken.

31.2 Fehler lokalisieren

Beim Testen eines Programms gibt es häufig Situationen, in denen wir zwar aufdecken, dass das Programm fehlerhaft arbeitet, aber allein durch die Auswertung der gescheiterten Testfälle nicht auf die Ursache eines Fehlers schließen können. In solchen Fällen benötigen wir Strategien und Techniken, um die Fehlerquelle aufzuspüren.

Techniken zur
Fehlersuche

Betrachten wir zunächst drei Techniken, die uns bei der Fehlersuche helfen können.

- Rückmeldung erzeugen. Wir können unseren Quelltext mit Ausgabeanweisungen anreichern, die während des Programmlaufs die Veränderung von Werten oder Objektzuständen dokumentieren.

```
for (int i = 0; i < LOOPS; i++) {
    ...
    a = ...
    System.out.println("i hat den Wert " + i + ", a hat "
        + "den Wert " + a);
}
```

- Programmausführung begrenzen. Bisweilen ist es hilfreich, ein Programm nicht vollständig ablaufen zu lassen, sondern Programmteile vorübergehend von der Ausführung auszuschließen:

```
public void methodeA() {
    ...
    this.methodeB();
//      this.methodeC(); // tritt das Problem auch auf,
//                        // wenn methodeC() nicht
//                        // ausgeführt wird?
}
```

Das vorübergehende Ausschalten von Programmteilen mit Hilfe von Kommentarzeichen wird „Auskommentieren“ genannt.

Debugger

- Wir können auf spezielle Hilfsprogramme zur interaktiven Fehlersuche zurückgreifen, sogenannte Debugger. Debugger, die Bestandteil der meisten Entwicklungsumgebungen sind, gewähren Einblick in das Verhalten eines Programms. Folgende Aufgaben werden typischerweise von Debuggern unterstützt:
 - setzen von Programmstopps, um die Ausführung eines Programms an einem bestimmten Punkt des Codes zu unterbrechen.
 - schrittweiser Durchlauf durch den Code und
 - Untersuchen von Attributen, Klassenvariablen und lokalen Variablen während des Programmlaufs.

Obwohl der Name anderes suggeriert, entfernt ein Debugger keine Fehler, er spürt nicht einmal Fehler auf. Alle drei genannten Techniken können uns lediglich helfen zu verstehen, was ein Programm schrittweise tut. Das Aufspüren und Beheben

konkreter Fehler bleibt uns überlassen. Eine bewährte Strategie, um Fehler aufzuspüren, ist folgendes schrittweises Vorgehen: Suchstrategie

1. Fehler reproduzieren. Zunächst müssen Sie wissen, bei welcher Vorbedingung oder Eingabe der Fehler auftritt. Möglicherweise haben Sie diese Information durch Tests bereits erhalten. Reproduzieren Sie den Fehler beim Durchlaufen der folgenden Schritte.
2. Fehlerstelle lokalisieren. Grenzen Sie, ausgehend von der `main()`-Methode, den fehlerhaften Programmteil ein. Benutzen Sie den Debugger und/oder kommentieren Sie Programmteile aus, um die Fehlerstelle nach und nach einzugrenzen:

```
public static void main(String[] args) {  
    X meinX = new X(1234); // 1234 ist eine Eingabe, bei  
                           // der der Fehler auftritt  
    meinX.methodeA();  
    //    meinX.methodeB(); // tritt das Problem auch auf,  
                           // wenn methodeB() nicht  
                           // ausgeführt wird?  
}
```

3. Fehler erkennen. Wenn Sie die Fehlerstelle soweit wie möglich eingegrenzt haben, sollten Sie die betreffende Stelle Ihres Programms Schritt für Schritt analysieren. Fragen Sie sich *vor* jedem Ausführungsschritt, welche Werte die beteiligten Variablen annehmen *sollten*. Benutzen Sie Ausgabeanweisungen oder den Debugger um festzustellen, welche Werte sie *tatsächlich* annehmen und vergleichen Sie.
4. Fehler verstehen. Haben Sie die fehlerhaft arbeitende Anweisung gefunden, so ist es unverzichtbar, dass Sie verstehen, warum die Anweisung zu dem Fehler führt. Erst wenn Sie sich die Auswirkung der Anweisung erklären können, sind Sie in der Lage, den Fehler zuverlässig zu korrigieren. Häufige Fehler haben wir in Abschnitt 31.1 aufgelistet.

31.3 Fehler vermeiden

In erster Linie sollten Sie natürlich bestrebt sein, so zu programmieren, dass Ihnen möglichst wenige Fehler unterlaufen und Ihre Programme robust und zuverlässig arbeiten. Beherzigen Sie deshalb die folgenden praktischen Ratschläge.

Quelltext wird häufiger gelesen als geschrieben. Gut lesbarer Quelltext erleichtert die Kommunikation unter Entwicklern, die Änderung oder Ergänzung von Programmteilen und die Fehlersuche. Deshalb:

- Wählen Sie aussagekräftige Variablennamen.
- Halten Sie Ausdrücke übersichtlich. Vereinfachen oder zerlegen Sie lange Ausdrücke.

- Sparen Sie nicht an Programmzeilen, indem Sie Anweisungen aneinanderreihen.

Das Geheimnisprinzip schützt vor unsachgemäßer Verwendung einer Klasse. Beachten Sie deshalb das Geheimnisprinzip beim Entwurf von Klassen:

- Machen Sie außerhalb der Klasse nur diejenigen Elemente sichtbar, die tatsächlich benötigt werden.
- Exemplarvariablen sollten außerhalb der Klasse grundsätzlich unsichtbar (`private`) sein.

Der Entwurf von Methoden erfordert besondere Aufmerksamkeit. In Methoden mit hoher Komplexität stecken die meisten Fehlerquellen. Deshalb:

- Verteilen Sie die Funktionalität komplexer Methoden auf mehrere übersichtliche Methoden.
- Vermeiden Sie Quelltext-Wiederholungen, indem Sie redundante Teile in eine separate Methode auslagern.
- Rechnen Sie damit, dass Methoden mit unsinnigen Parametern aufgerufen werden können, und sorgen Sie für diesen Fall vor.
- Denken Sie beim Entwurf an die Möglichkeit, Ausnahmen zu werfen.
- Fragen Sie sich, ob Sie die Methode verstünden, wenn sie nicht die Person wären, die sie geschrieben hat.
- Dokumentieren Sie Vor- und Nachbedingungen verständlich.

Wir beenden diese Auflistung mit zwei sehr allgemeinen Ratschlägen:

- Scheuen Sie sich nicht, von vorne zu beginnen. Änderungen misslungener Lösungen können langwieriger und fehleranfälliger sein als ein frischer Anlauf.
- Beginnen Sie nicht von vorne, ohne nach bewährten Lösungen Ausschau gehalten zu haben. Objektorientierte Programmierung bedeutet Wiederverwendung. Existiert bereits eine Klasse, die das Benötigte leistet oder entsprechend spezialisiert werden kann? Konsultieren Sie auch die Java-API-Referenz.

31.4 EXKURS: Weiterführende Konzepte

Auf den systematischen Entwurf von Klassen und Klassengeflechten, die Verteilung von Funktionalität und die methodische Entwicklung zuverlässiger Softwaresysteme können wir im Rahmen dieses Kurses nicht weiter eingehen. Wir wollen zum Abschluss dieses Kapitels ein paar Stichworte zu Themen und Techniken des Software-Engineering nennen und Sie ermuntern, weiterführende Literatur zu konsultieren.

[Som07] Ian Sommerville:

Software Engineering

Pearson Studium, 8. (aktualisierte) Auflage, 2007

Standardlehrwerk mit einer breit angelegten Übersicht über die wichtigsten Aspekte des Software-Engineering in deutscher Übersetzung. Die Programmbeispiele sind in Java geschrieben, die Objektmodelle in der UML-Notation dargestellt.

[GoF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:

Design Patterns. Elements of Reusable Object-Oriented Software

Addison Wesley, 1995

Entwurfsmuster (engl. *design patterns*) sind bewährte Lösungs-Schemata für wiederkehrende Entwurfsprobleme in der Softwareentwicklung. Sie stellen Vorlagen zur Problemlösung dar, die in einem bestimmten Zusammenhang einsetzbar sind.

[Fowler99] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts:

Refactoring. Improving the Design of Existing Code

Addison-Wesley, 1999

Refactoring bezeichnet die Strukturverbesserung von Quelltext unter Beibehaltung des beobachtbaren Programmverhaltens. Durch Refactoring sollen die Lesbarkeit, Verständlichkeit, Wartbarkeit und Erweiterbarkeit verbessert werden, mit dem Ziel, den Aufwand für Fehleranalyse und funktionale Erweiterungen zu senken.

[Beck02] Kent Beck:

Test Driven Development by Example

Addison-Wesley, 2002

Testgetriebene Entwicklung (engl. *test first development*) ist eine Methode, bei der Software-Tests konsequent vor den zu testenden Komponenten entwickelt werden. Grob gesagt werden die Anforderungen an eine zu entwickelnde Software zunächst in Form von Tests formuliert, die sich mit Hilfe eines Test-Frameworks wie JUnit jederzeit automatisch ausführen lassen. Natürlich scheitern diese Tests zunächst, da die geforderte Funktionalität noch nicht implementiert ist, und werden erst mit fortschreitender Realisierung der Software nach und nach bestanden.

32 Zusammenfassung

Mit Hilfe von **Ausnahmen** können in Java Laufzeitfehler angezeigt und auch behandelt werden. Die Oberklasse aller Ausnahmen ist `Throwable`. Es existieren darunter zum einen die Klasse `Exception`, die im Regelfall behandelbare Fehler darstellt, und die Klasse `Error`, die für schwere Laufzeitfehler, die im Normalfall nicht behandelt werden können, steht. Ausnahmen werden mit Hilfe von `throw` **geworfen** und können an den Aufrufer **weitergereicht** oder mit Hilfe der `try`-Anweisung **behandelt** werden. Eine Methode, die eine Ausnahme erzeugen kann, muss dies mit Hilfe von `throws` in ihrem Kopf anzeigen, falls es sich bei der Ausnahme nicht um eine `RuntimeException` oder einen `Error` handelt.

Mit Hilfe von **javadoc**-Kommentaren und den zugehörigen Schlüsselwörtern können **Dokumentationen** für Java-Klassen generiert werden.

Tests sollten zu jeder Programmentwicklung dazugehören. Dabei sollten Tests systematisch und wiederholbar sein.

Das Framework **JUnit** unterstützt bei der Entwicklung, Durchführung und Auswertung von Tests. Durch diese Unterstützung können die Tests auch ohne weiteres nach jeder Programmänderung erneut ausgeführt werden.

Eine JUnit-Testklasse ist eine Unterklasse von `junit.framework.TestCase`. Jede Testmethode muss vier Eigenschaften aufweisen. Vor der Ausführung jeder Testmethode werden zusätzlich die `setUp()` und anschließend die `tearDown()` aufgerufen.

Im Rahmen des Kurses haben wir uns mit **Klassentests** beschäftigt. Es gibt jedoch noch weitere Teststufen wie den **Integrations**-, **System**-, **Abnahme**- und **Regressionstest**.

Bevor die Tests durchgeführt werden, müssen sie geplant werden und für jeden Testfall die Eingabedaten, Vorbedingungen sowie das erwartete Verhalten bzw. die erwartete Ausgabe festgelegt werden. Dabei muss beachtet werden, dass die Anforderungen geprüft werden und nicht die Implementierung als Spezifikation hergenommen wird.

Bei der Identifikation von konkreten Testfällen unterscheiden wir zwischen **Whitebox**- und **Blackbox-Verfahren**. Für letzteres haben wir die Bildung von Äquivalenzklassen sowie die Grenzwertanalyse kennen gelernt. Bei **Whitebox**-Verfahren können Aussagen über **Anweisungs**-, **Zweig**- und **Pfadüberdeckung** getroffen werden.

Um **Fehler** zu lokalisieren, können wir Ausgaben in unsere Programme einbauen oder Programmteile auskommentieren, jedoch können diese Techniken schnell unübersichtlich werden. Alternativ können Debugger verwendet werden, die es ermöglichen, Programme schrittweise zu durchlaufen und jederzeit die Werte von Variablen zu inspizieren.

Lösungen zu Selbsttestaufgaben der Kurseinheit

Lösung zu Selbsttestaufgabe 28.2-1:

```
void legeMehrwertsteuerFest(double neueMwSt) {  
    if (neueMwSt < 0) {  
        throw new IllegalArgumentException("Eine negative "  
            + "Mehrwertsteuer ist nicht zulaessig");  
    }  
    this.mehrwertsteuer = neueMwSt;  
}
```

Die Ausnahme muss nicht im Methodenkopf angegeben werden.

Lösung zu Selbsttestaufgabe 28.3-1:

Bei einem Rabatt von -0.2 wird die folgende Ausgabe erzeugt:

```
NegativerRabattAusnahme: Ein Rabatt von -0.2 ist nicht zulaessig.
```

Bei einem Rabatt von 1.2 wird die folgende Ausgabe erzeugt:

```
ZuHoherRabattAusnahme: Ein Rabatt von 1.2 ist nicht zulaessig.
```

Lösung zu Selbsttestaufgabe 28.3-2:

- a) 1
- b) 2
- c) 5
- d) 3
- e) 4
- f) 3
- g) 4

Lösung zu Selbsttestaufgabe 28.3-3:

Bei der Anweisung `throw AnException();` in Methode `a` muss ein `new` ergänzt werden. Bei Methode `a` muss außerdem im Kopf `throws MyException, AnException` ergänzt werden. Es wäre auch möglich, stattdessen `throws Exception` zu ergänzen, jedoch sollten die Ausnahmetypen im Normalfall so genau wie möglich angegeben werden. Ein `try-catch`-Konstrukt ist in diesem Fall keine sinnvolle Lösung.

Bei Methode `b` kann ein `throws AnotherException` ergänzt werden. Dies ist aber nicht nötig, da es sich bei `AnotherException` um eine `RuntimeException` handelt.

In der Methode `callA()` müssen die Ausnahmen entweder gefangen werden, oder der Methodenkopf muss auch um eine `throws`-Anweisung ergänzt werden. Entscheidet man sich für `try-catch`, so sollte man zwei `catch`-Blöcke verwenden, statt nur einen für die gemeinsame Oberklasse `Exception`, um so jede Ausnahme gezielt behandeln zu können:

```
public void callA() {
    try {
        a(-2);
    } catch (MyException e) {
        System.out.println("MyException caught");
    } catch (AnException e) {
        System.out.println("AnException caught");
    }
}
```

Ließe man einen der beiden Blöcke weg, so würde dies einen Übersetzungsfehler verursachen.

Bei `callB()` kann man auch ein `try-catch` ergänzen oder eine `throws`-Anweisung im Kopf der Methode, muss dies jedoch nicht tun.

Lösung zu Selbsttestaufgabe 28.3-4:

Der Aufruf `z(-2)` erzeugt die Ausgabe `-1`, `z(0)` erzeugt die Ausgabe `-2`, und `z(7)` erzeugt die Ausgabe `49`.

Lösung zu Selbsttestaufgabe 29-1:

Ihre Kommentare sollten mindestens die folgenden Angaben enthalten:

```
/**
 * Ein Punkt in einem zweidimensionalen Koordinatensystem.
 */
public class Punkt {
    private double x;
    private double y;

    /**
     * Erzeugt einen Punkt mit den Koordinaten (0.0, 0.0).
     */
    public Punkt() {
        x = 0.0;
        y = 0.0;
    }

    /**
     * Erzeugt einen Punkt mit den gegebenen Koordinaten.
     * @param xPosition die horizontale Position
     * @param yPosition die vertikale Position
     */
    public Punkt(double xPosition, double yPosition) {
        x = xPosition;
        y = yPosition;
    }

    /**
     * Berechnet die Distanz zu einem anderen Punkt.
     * @param p der andere Punkt
     * @return die Distanz
     */
    public double distanzZu(Punkt p) {
        return Math.sqrt((x - p.getX()) * (x - p.getX())
            + (y - p.getY()) * (y - p.getY()));
    }

    /**
     * @return die horizontale Position
     */
    public double getX() {
        return x;
    }

    /**
     * @return die vertikale Position
     */
    public double getY() {
        return y;
    }
}
```

Lösung zu Selbsttestaufgabe 29-2:

```

/**
 * legt einen neuen Rabatt für die Rechnung fest
 * @param neuerRabatt der neue Rabatt
 * @throws NegativerRabattAusnahme falls der neue Rabatt
 * negativ ist
 * @throws ZuHoherRabattAusnahme falls der Rabatt groesser
 * als 100 Prozent ist
 */
public void legeRabattFest(final double neuerRabatt) throws
    NegativerRabattAusnahme, ZuHoherRabattAusnahme {
    // ...
}

```

Lösung zu Selbsttestaufgabe 30.3-1:

Negative Tests:

- Beim Erzeugen der Rechnung wird ein null-Verweis übergeben und anschließend die Rabattbestimmung getestet.
- Es wird ein negativer Betrag festgelegt und anschließend die Rabattbestimmung getestet.

Positive Tests:

- Die Rabattbestimmung für eine Premiumkundin bei einem Betrag von 250 Euro wird getestet.
- Die Rabattbestimmung für eine normale Kundin bei einem Betrag von 10 Euro wird getestet.

Lösung zu Selbsttestaufgabe 30.3-2:

```

public class EinfacherRechnungstest {
    public static void main(String[] args) {
        Kunde k = new Kunde("Anna Meier");
        Rechnung r = new Rechnung(k);
        r.legeBetragFest(120);
        int rabatt = r.bestimmeRabatt();
        if (rabatt != 5) {
            System.out.println("Der Rabatt bei einer Rechnung "
                + "von 120 Euro wird falsch berechnet!");
            System.out.println("Der Rabatt sollte 5 Prozent "
                + "betragen, betraegt aber: " + rabatt);
        }
        k = new Premiumkunde("Hans Mayer");
        r = new Rechnung(k);
        r.legeBetragFest(20);
        rabatt = r.bestimmeRabatt();
    }
}

```

```

    if (rabatt != 5) {
        System.out.println("Der Rabatt bei einer Rechnung "
            + "von 20 Euro für einen Premiumkunden wird "
            + "falsch berechnet!");
        System.out.println("Der Rabatt sollte 5 Prozent "
            + "betragen, betraegt aber: " + rabatt);
    }
    k = new Premiumkunde("Ina Kunz");
    r = new Rechnung(k);
    r.legeBetragFest(230);
    rabatt = r.bestimmeRabatt();
    if (rabatt != 10) {
        System.out.println("Der Rabatt bei einer Rechnung "
            + "von 230 Euro für einen Premiumkunden wird "
            + "falsch berechnet!");
        System.out.println("Der Rabatt sollte 10 Prozent "
            + "betragen, betraegt aber: " + rabatt);
    }
    r = new Rechnung(null);
    try {
        r.legeBetragFest(10);
        rabatt = r.bestimmeRabatt();
        System.out.println("Die Rabattbestimmung bei einer "
            + "Rechnung ohne Kunden ist unerwarteter "
            + "Weise erfolgreich verlaufen");
    } catch (Exception e) {
        // Wir erwarten eine Ausnahme
    }
    k = new Premiumkunde("Fritz Hinz");
    r = new Rechnung(k);
    try {
        r.legeBetragFest(-10);
        rabatt = r.bestimmeRabatt();
        System.out.println("Die Rabattbestimmung bei einem "
            + "negativen ist unerwarteter Weise "
            + "erfolgreich verlaufen");
    } catch (Exception e) {
        // Wir erwarten eine Ausnahme
    }
}
}

```

Lösung zu Selbsttestaufgabe 30.4-1:

```

public class Rechnungstest extends junit.framework.TestCase {
    Rechnung r;
    Rechnung rPremium;

    public void setUp() {
        Kunde k = new Kunde("Anna Meier");
        Premiumkunde pk = new Premiumkunde("Ina Kunze");
    }
}

```

```
        r = new Rechnung(k);
        rPremium = new Rechnung(pk);
    }

    public void testeRabattBeiNeuerRechnung() {
        assertEquals("Rabatt bei 0 Euro und normaler Kundin "
            + "fehlerhaft.", 0, r.bestimmeRabatt());
    }

    public void testeRabattBeiNeuerRechnungPremium() {
        assertEquals("Rabatt bei 0 Euro und Premiumkundin "
            + "fehlerhaft.", 5, rPremium.bestimmeRabatt());
    }
}
```

Lösung zu Selbsttestaufgabe 30.4-2:

testAddition() schlägt fehl, da `3 > mr.addiere(1,2)` zu false evaluiert.

testSubtraktion() schlägt nicht fehl.

testMultiplikation() schlägt fehl, da `mr.multipliziere(6,2)` 12 liefert und nicht 8.

testDivision() schlägt fehl, da bei der zweiten Anweisung eine Division durch 0 auftritt und daher eine Ausnahme geworfen wird; das Auftreten von nicht behandelten Ausnahmen führt immer zum Fehlschlagen einer Testmethode.

testBetrag() schlägt nicht fehl.

Lösung zu Selbsttestaufgabe 30.5-1:

```
public class Rechnungstest extends junit.framework.TestCase {
    Rechnung r;
    Rechnung rPremium;

    public void setUp() {
        Kunde k = new Kunde("Anna Meier");
        Premiumkunde pk = new Premiumkunde("Ina Kunze");
        r = new Rechnung(k);
        rPremium = new Rechnung(pk);
    }

    public void testeRabattBeiNeuerRechnung() {
        assertEquals("Rabatt bei 0 Euro und normaler Kundin "
            + "fehlerhaft.", 0, r.bestimmeRabatt());
    }
}
```

```

public void testeRabattBeiNeuerRechnungPremium() {
    assertEquals("Rabatt bei 0 Euro und Premiumkundin "
        + "fehlerhaft.", 5, rPremium.bestimmeRabatt());
}

public void testeRabattBeiNiedrigemBetrag() {
    r.legeBetragFest(1.5);
    assertEquals("Rabatt bei 1.5 Euro und normaler Kundin "
        + "fehlerhaft.", 0, r.bestimmeRabatt());
}

public void testeRabattBeiNiedrigemBetragPremium() {
    rPremium.legeBetragFest(1.5);
    assertEquals("Rabatt bei 1.5 Euro und Premiumkundin "
        + "fehlerhaft.", 5, rPremium.bestimmeRabatt());
}

public void testeRabattBeiBetragsgrenze100() {
    r.legeBetragFest(99.0);
    assertEquals("Rabatt bei 99 Euro und normaler Kundin "
        + "fehlerhaft.", 0, r.bestimmeRabatt());
    r.legeBetragFest(99.5);
    assertEquals("Rabatt bei 99.5 Euro und normaler Kundin "
        + "fehlerhaft.", 0, r.bestimmeRabatt());
    r.legeBetragFest(99.9);
    assertEquals("Rabatt bei 99.9 Euro und normaler Kundin "
        + "fehlerhaft.", 0, r.bestimmeRabatt());
    r.legeBetragFest(100);
    assertEquals("Rabatt bei 100 Euro und normaler Kundin "
        + "fehlerhaft.", 5, r.bestimmeRabatt());
    r.legeBetragFest(100.5);
    assertEquals("Rabatt bei 100.5 Euro und normaler Kundin "
        + "fehlerhaft.", 5, r.bestimmeRabatt());
    r.legeBetragFest(101);
    assertEquals("Rabatt bei 101 Euro und normaler Kundin "
        + "fehlerhaft.", 5, r.bestimmeRabatt());
}

public void testeRabattBeiBetragsgrenze100Premium() {
    rPremium.legeBetragFest(99.0);
    assertEquals("Rabatt bei 99 Euro und Premiumkundin "
        + "fehlerhaft.", 5, rPremium.bestimmeRabatt());
    rPremium.legeBetragFest(99.5);
    assertEquals("Rabatt bei 99.5 Euro und Premiumkundin "
        + "fehlerhaft.", 5, rPremium.bestimmeRabatt());
    rPremium.legeBetragFest(99.9);
    assertEquals("Rabatt bei 99.9 Euro und Premiumkundin "
        + "fehlerhaft.", 5, rPremium.bestimmeRabatt());
    rPremium.legeBetragFest(100);
    assertEquals("Rabatt bei 100 Euro und Premiumkundin "
        + "fehlerhaft.", 5, rPremium.bestimmeRabatt());
    rPremium.legeBetragFest(100.5);
}

```

```

        assertEquals("Rabatt bei 100.5 Euro und Premiumkundin "
            + "fehlerhaft.", 5, rPremium.bestimmeRabatt());
        rPremium.legeBetragFest(101);
        assertEquals("Rabatt bei 101 Euro und Premiumkundin "
            + "fehlerhaft.", 5, rPremium.bestimmeRabatt());
    }

    public void testeRabattBeiBetragsgrenze200() {
        r.legeBetragFest(199.0);
        assertEquals("Rabatt bei 199 Euro und normaler Kundin "
            + "fehlerhaft.", 5, r.bestimmeRabatt());
        r.legeBetragFest(199.5);
        assertEquals("Rabatt bei 199.5 Euro und normaler Kundin "
            + "fehlerhaft.", 5, r.bestimmeRabatt());
        r.legeBetragFest(199.9);
        assertEquals("Rabatt bei 199.9 Euro und normaler Kundin "
            + "fehlerhaft.", 5, r.bestimmeRabatt());
        r.legeBetragFest(200);
        assertEquals("Rabatt bei 200 Euro und normaler Kundin "
            + "fehlerhaft.", 5, r.bestimmeRabatt());
        r.legeBetragFest(200.5);
        assertEquals("Rabatt bei 200.5 Euro und normaler Kundin "
            + "fehlerhaft.", 5, r.bestimmeRabatt());
        r.legeBetragFest(201);
        assertEquals("Rabatt bei 201 Euro und normaler Kundin "
            + "fehlerhaft.", 5, r.bestimmeRabatt());
    }

    public void testeRabattBeiBetragsgrenze200Premium() {
        rPremium.legeBetragFest(199.0);
        assertEquals("Rabatt bei 199 Euro und Premiumkundin "
            + "fehlerhaft.", 5, rPremium.bestimmeRabatt());
        rPremium.legeBetragFest(199.5);
        assertEquals("Rabatt bei 199.5 Euro und Premiumkundin "
            + "fehlerhaft.", 5, rPremium.bestimmeRabatt());
        rPremium.legeBetragFest(199.9);
        assertEquals("Rabatt bei 199.9 Euro und Premiumkundin "
            + "fehlerhaft.", 5, rPremium.bestimmeRabatt());
        rPremium.legeBetragFest(200);
        assertEquals("Rabatt bei 200 Euro und Premiumkundin "
            + "fehlerhaft.", 10, rPremium.bestimmeRabatt());
        rPremium.legeBetragFest(200.5);
        assertEquals("Rabatt bei 200.5 Euro und Premiumkundin "
            + "fehlerhaft.", 10, rPremium.bestimmeRabatt());
        rPremium.legeBetragFest(201);
        assertEquals("Rabatt bei 201 Euro und Premiumkundin "
            + "fehlerhaft.", 10, rPremium.bestimmeRabatt());
    }
}

```

Lösung zu Selbsttestaufgabe 30.5-2:

Es gibt die folgenden Äquivalenzklassen:

- $0 < \text{tage} < 14 \ \&\& \ 0 \leq \text{km} \leq 200$
- $0 < \text{tage} < 14 \ \&\& \ \text{km} > 200$
- $\text{tage} \geq 14$
- $\text{tage} \leq 0 \ || \ \text{km} < 0$

Mögliche Testwerte sind folglich:

Tage	Kilometer
-1	20
0	20
1	20
1	199
1	200
1	201
1	400

Tage	Kilometer
4	50
5	199
5	200
6	201
8	500
13	20
13	199

Tage	Kilometer
13	200
13	201
13	450
14	5
14	200
14	300
20	1000

Lösung zu Selbsttestaufgabe 30.5-3:

Abb. ML 18 zeigt das Aktivitätsdiagramm für die Methode `berechne()`.

Mit Hilfe der Werte $x = -2$ und $y = -3$ erhalten wir eine vollständige Anweisungsüberdeckung.

```
public class RechnerTest extends junit.framework.TestCase {
    public void testBerechneAnweisung() {
        assertEquals("Berechnung bei x = -2 und y = -3 "
            + "fehlerhaft", 6, new Rechner().berechne(-2, -3));
    }
}
```

Lösung zu Selbsttestaufgabe 30.5-4:

```
public class RechnerTest extends junit.framework.TestCase {
    // ...

    public void testBerechneZweig() {
        assertEquals("Berechnung bei x = 4 und y = 5 "
            + "fehlerhaft", 20, new Rechner().berechne(4, 5));
        assertEquals("Berechnung bei x = -2 und y = -3 "
            + "fehlerhaft", 6, new Rechner().berechne(-2, -3));
    }
}
```

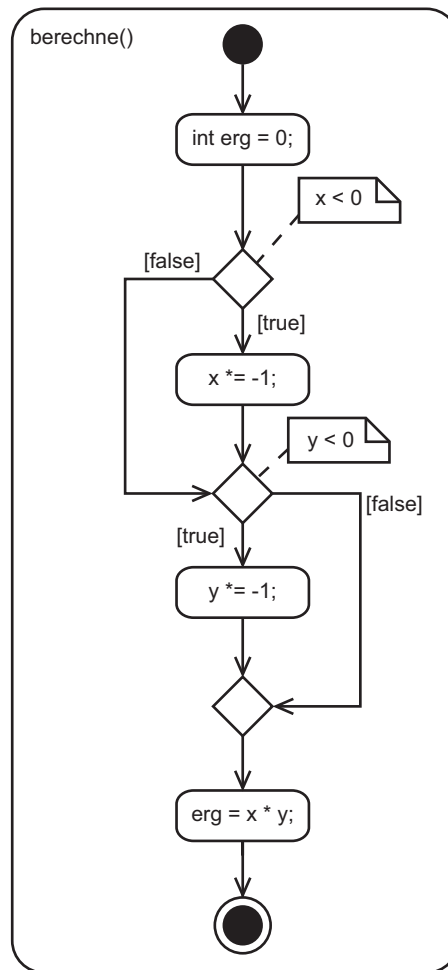


Abb. ML 18: Aktivitätsdiagramm `berechne()`

Lösung zu Selbsttestaufgabe 30.5-5:

```

public class RechnerTest extends junit.framework.TestCase {
    // ...

    public void testBerechnePfad() {
        assertEquals("Berechnung bei x = 4 und y = 5 "
            + "fehlerhaft", 20, new Rechner().berechne(4, 5));
        assertEquals("Berechnung bei x = -2 und y = -3 "
            + "fehlerhaft", 6, new Rechner().berechne(-2, -3));
        assertEquals("Berechnung bei x = -3 und y = 7 "
            + "fehlerhaft", 21, new Rechner().berechne(-3, 7));
        assertEquals("Berechnung bei x = 5 und y = -2 "
            + "fehlerhaft", 10, new Rechner().berechne(5, -2));
    }
}

```


Lösung zu Selbsttestaufgabe 30.5-6:

Mögliche Tests sind `feld = null` und `feld = new int[0]`, `feld = new int[] {1, 2}` und `feld = new int[] {1, 8, 123, 8}` und `feld = new int[] {1}`. Dabei decken die ersten vier jeweils verschiedene Fehler auf.