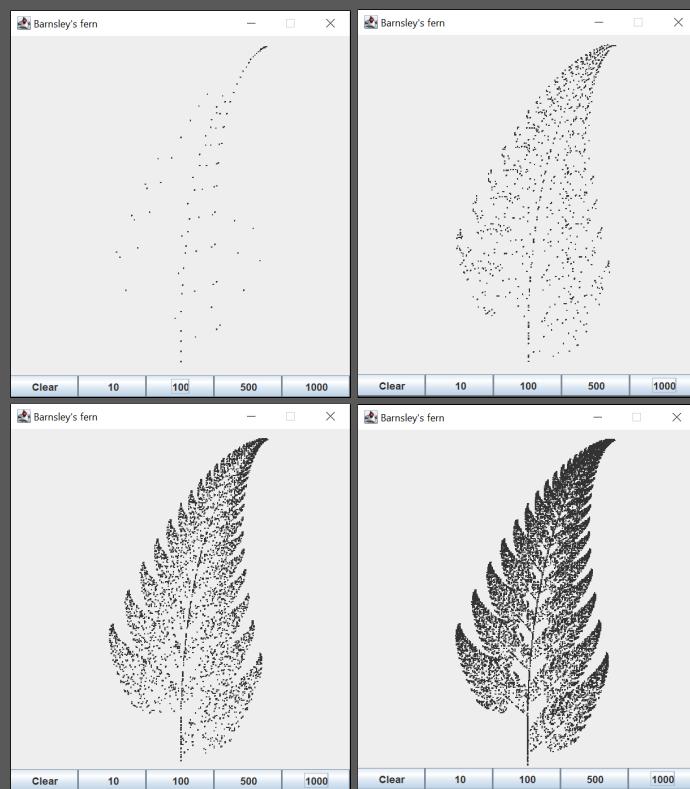


Marc Hensel

Objektorientierte Programmierung

Übungsaufgaben, Fragen und ausführliche Praktika



The author has made effort to ensure the provided information is correct. However, the information is provided without any warranty. Neither the author, nor any other person or organization will be held liable for any damages caused or alleged to have been caused directly or indirectly by this document.

Arbeitsheft zur Lehrveranstaltung *Objektorientierte Programmierung*
Stand: 19.09.2024



© Prof. Dr. Marc Hensel
<http://www.haw-hamburg.de/marc-hensel>

Published under Creative Commons license CC BY-NC-ND 3.0
Attribution, non-commercial, no derivatives
<https://creativecommons.org/licenses/by-nc-nd/3.0/deed.en>

Vorwort

Dieses Arbeitsheft richtet sich an Studierende des Bachelorstudiengangs *Elektrotechnik und Informationstechnik* der Hochschule für Angewandte Wissenschaften Hamburg. Ziel ist es vor allem, das Gelernte anzuwenden, anzuwenden und anzuwenden. Denn letztendlich sollen und müssen Sie als Vorbereitung für Ihr späteres Berufsleben Kompetenzen erwerben – also etwas *können*.

Aufgaben und Fragen Die 121 Übungsaufgaben und 92 Verständnisfragen sind auf die Struktur der begleitenden Vorlesung und insbesondere auch auf die Lernziele ausgerichtet. Sie sollten daher die Inhalte der Vorlesung und dieses Arbeitsheft „parallel“ durcharbeiten. Seien Sie zudem stets neugierig und setzen Sie in Software um, was Sie persönlich interessiert. Schwierigkeit und Umfang der Übungsaufgaben steigen im Verlauf der Kapitel bewusst teils erheblich an. Vergleichsweise einfache Aufgaben in den ersten Kapiteln sollen Ihnen den Einstieg in die Softwareentwicklung mit Java erleichtern. Schwierigere und komplexere Aufgaben sollen Sie hingegen gut auf die praktische Prüfung vorbereiten.

Praktika Den Aufgaben und Fragen schließen sich 15 etwas umfangreichere Praktikumsaufgaben an. Diese beinhalten ab Kapitel 14 insbesondere jeweils aufeinander aufbauende Aufgaben rund um ein Casino, geografische Koordinaten und das Spiel des Lebens. In den Aufgabenkapiteln wird an entsprechenden Stellen darauf verwiesen, wenn es inhaltlich Zeit für die nächste Praktikumsaufgabe wäre. Dies soll es Ihnen unabhängig von der konkreten Semesterplanung erleichtern, die Materialien sinnvoll aufeinander aufbauend zu bearbeiten. Beachten Sie zudem folgende Hinweise:

- ▶ *Vorbereitung:* Ich empfehle ganz ausdrücklich, die Aufgaben bereits *vor* dem Labortermin *selbstständig* zu bearbeiten. Hierdurch gewinnen Sie in den Laboren Zeit, die Aufgaben ausgiebig innerhalb Ihrer Gruppe sowie mit mir oder dem Betreuer bzw. der Betreuerin zu besprechen und Fragen zu klären. Tauschen Sie sich bei Problemen in der Vorbereitung mit Ihrer Gruppe und/oder anderen Kommilitonen aus. Erzeugen Sie im Labor *eine* gemeinsame Lösung, die Ihre Gruppe zur Abnahme präsentiert.
- ▶ *Erfolgreiche Teilnahme:* Es besteht Anwesenheitspflicht bei *allen* Laborterminen. Die erfolgreiche Abnahme der Praktika ist darüber hinaus Voraussetzung zur Teilnahme an der Laborprüfung. Hierfür *muss* der Code den Programmierrichtlinien gemäß Anhang C auf Seite 153 entsprechen. Weiterhin müssen zur erfolgreichen Abnahme die theoretischen Grundlagen und Hintergründe der im Praktikum behandelten Konzepte erläutert werden können.

Arbeitshilfen Zur Erleichterung Ihrer Arbeit bzw. Ihres Einstiegs in bestimmte Themenbereiche beinhaltet dieses Arbeitsheft einige Anhänge. Diese beinhalten insbesondere Hilfen zur verwendeten Entwicklungsumgebung *IntelliJ IDEA* sowie eine Übersicht der wichtigsten Programmierrichtlinien (*Coding style*).

Beispiel-Lösungen Ich habe zu allen Aufgaben Beispiel-Lösungen in GitHub abgelegt. Bitte schauen Sie im zur Verfügung gestellten IntelliJ IDEA-Projekt in die jeweils zu Beginn einer Aufgabe angegebene Klasse.

Downloads

https://github.com/MarcOnTheMoon/coding_learners_java



Und nun los – viel Spaß!

Marc Hensel

Inhaltsverzeichnis

Vorwort	3
Inhaltsverzeichnis	4
I Aufgaben und Fragen	9
1 Einführung	10
1.1 Aufgaben	10
1.2 Fragen	10
2 Imperative Konzepte	11
2.1 Datentypen und Operatoren	11
2.2 Programmfluss	14
2.3 Fragen	17
3 Klassen und Objekte	18
3.1 Variablen	18
3.2 Methoden und Konstruktoren	18
3.3 Klassenvariablen und Klassenmethoden	22
3.4 Fragen	24
4 Ausgewählte Klassen	26
4.1 Zeichenketten	26
4.2 Felder (Arrays)	27
4.3 Mehrdimensionale Felder (Arrays)	29
4.4 Listen	32
4.5 Fragen	33
5 Anwendungsbeispiel: Bildverarbeitung	35
5.1 Bilddaten	35
5.2 Bilddateien lesen und schreiben	35
5.3 Kodierung	37
5.4 Punktoperationen	39
5.5 Kantendetektion	40
6 Vererbung	44
6.1 Grundlegende Klassen	44
6.2 Vererbung	46
6.3 Fragen	47
7 Abstrakte Elemente	48
7.1 Nullstellen mathematischer Funktionen	48
7.2 Sortieren und Iterieren	49
7.3 Benachrichtigen von Beobachtern	50
7.4 Fragen	53
8 Grafische Benutzeroberflächen (GUI)	55
8.1 Würfel	55
8.2 Mehrwertsteuer	56
8.3 Graphen mathematischer Funktionen	57
8.4 Grünfpflanzen und weitere grafische Elemente	58
8.5 Fragen	59

9 Anwendungsbeispiel: Audioverarbeitung	61
9.1 Überblick	61
9.2 Audio-Signale im Zeitbereich	62
9.3 Diskrete Fourier-Transformation (DFT)	63
9.4 Filterung im Frequenzbereich	66
10 Ausnahmen	68
10.1 Ausnahmen fangen	68
10.2 Eigene Ausnahme-Klassen	68
10.3 Fragen	69
11 Eingabe und Ausgabe	71
11.1 Tastatureingabe und Textdateien schreiben	71
11.2 Text-Dateien lesen	71
11.3 Fragen	71
12 Anwendungsbeispiel: Audiodateien erzeugen	73
12.1 WAVE-PCM-Dateiformat	73
12.2 Töne und Akkorde	75
13 Parallelverarbeitung mit Threads	77
13.1 Primzahlen	77
13.2 Stoppuhr	78
13.3 Fragen	78
II Praktika: Casino	81
14 Praktikum: Einführung	82
14.1 Aktivierung von IntelliJ IDEA (PC-Pool)	82
14.2 Projektstruktur	82
14.3 Debugging	83
14.4 JUnit-Tests	84
15 Praktikum 1: Banditen	85
15.1 Übersicht	85
15.2 Einarmige Banditen	85
15.3 Mehrarmige Banditen	88
15.4 Ausblick	90
16 Praktikum 2: Spielhalle und Gewinn-Strategie	91
16.1 Übersicht	91
16.2 Spielhalle (Glücksräder)	92
16.3 Spiel mit mehrarmigen Banditen	93
17 Praktikum 3: GUI für mehrarmige Banditen	97
17.1 Übersicht	97
17.2 Anforderungen	99
17.3 Lösungsstrategie	100
III Praktika: Geografische Koordinaten	101
18 Praktikum 1: Geografische Koordinaten	102
18.1 Theorie: Geografische Koordinaten und Entfernung	102

6 ► INHALTSVERZEICHNIS

18.2 Klasse <i>GeoPosition</i>	103
18.3 Abstände	105
18.4 Einhaltung der Programmierrichtlinien (Qualität)	105
18.5 Theoretische Fragen	106
19 Praktikum 2: Wegstrecken (Routen)	107
19.1 Geografische Wegstrecken	107
19.2 Funktionalität	108
19.3 HAW-Laftreff	109
19.4 Flugrouten	109
19.5 Einhaltung der Programmierrichtlinien (Qualität)	109
19.6 Theoretische Fragen	110
20 Praktikum 3: GUI für Wegstrecken	111
20.1 Überblick	111
20.2 Aufgabe	111
20.3 Lösungsstrategie	113
20.4 Einhaltung der Programmierrichtlinien (Qualität)	113
21 Praktikum 4: Wegstrecken (Tracking)	114
21.1 Überblick	114
21.2 Aufgabe	115
IV Praktika: Spiel des Lebens (Game of Life)	117
22 Praktikum 3: GUI (Teil 1)	118
22.1 Überblick	118
22.2 Aufgabe	119
22.3 Lösungsstrategie	120
22.4 Einhaltung der Programmierrichtlinien (Qualität)	120
23 Praktikum 4: Logik & Threads (Teil 2)	121
23.1 Einleitung	121
23.2 Aufgabe	122
23.3 Hinweise	122
V Praktika: Sonstige	125
24 Praktikum 1: Komplexe Zahlen und Schwingkreis	126
24.1 Klasse <i>Complex</i>	126
24.2 Schwingkreis	128
24.3 Einhaltung der Programmierrichtlinien (Qualität)	129
25 Praktikum 1: 2D-Vektoren und Flugplan	130
25.1 Flugplan	130
25.2 Klasse <i>Vector2D</i>	130
25.3 Flugplan „Reloaded“	132
25.4 Einhaltung der Programmierrichtlinien (Qualität)	133
25.5 Theoretische Fragen	133

26 Praktikum 2: Kalender	134
26.1 Überblick	134
26.2 Funktionalität	135
26.3 Anwendung	137
27 Praktikum 3: Alarmanlage	138
27.1 Überblick	138
27.2 Aufgabe	139
27.3 Lösungsstrategie	139
28 Praktikum 4: Alarmanlagen-GUI	140
28.1 Überblick	140
28.2 Aufgabe	140
28.3 Lösungsstrategie	141
VI Appendix	143
A IntelliJ IDEA	144
A.1 Tipps & Tricks	144
A.2 Probleme und Lösungen	144
A.3 Unit-Tests	145
A.4 Debugging	147
A.5 Wichtige Tastenkombinationen	147
B Eclipse	149
B.1 Unit-Tests	149
B.2 Debugging	150
C Checkliste Softwarequalität	153
Index	154

Teil I

Aufgaben und Fragen

Kapitel 1

Einführung

1.1 Aufgaben

■ **Aufgabe 1.** (Klasse: `FirstApp`) Einrichten der Entwicklungsumgebung und erstes Programm:

1. Installieren und starten Sie *IntelliJ IDEA Community*.
2. Erstellen Sie ein Verzeichnis, das ein Projekt mit Ihren Programmen enthalten soll.
3. Erzeugen Sie über *File / New Project* ein Java-Projekt im zuvor erstellten Verzeichnis.
4. Erzeugen Sie im Ordner `src` in der Projektübersicht im linken Bereich der grafischen Oberfläche ein Paket.
5. Erstellen Sie innerhalb des Paketes eine ausführbare Klasse `FirstApp`.
6. Fügen Sie eine `main()`-Methode mit einer Codezeile hinzu, sodass bei Ausführung des Programms eine Zeichenkette ausgegeben wird, und führen Sie das Programm aus.

Beispielausgabe:

```
Let's have some fun!
```

■ **Aufgabe 2.** (Klasse: `PrintLines`) Erstellen Sie innerhalb des in Aufgabe 1 erstellten Paketes ein weiteres Programm, indem Sie eine zusätzliche ausführbare Klasse `PrintLines` erzeugen.

1. Fügen Sie zur `main()`-Methode mehrere aufeinanderfolgende Textausgaben über `println()` hinzu. Die auszugebenden Zeichenketten können Sie frei wählen.
2. Es befinden sich zwei ausführbare Klassen im Paket. Führen Sie mal die eine, mal die andere Klasse aus.
3. Ändern Sie die Methodennamen `println()` in `print()`. Wie ändert sich hierdurch die Ausgabe?

1.2 Fragen

Frage 1. „Quelltexte in Java werden in Dateien übersetzt, die auf jedem üblichen System (z. B. Windows, Linux) ausführbar sind.“ Wahr oder unwahr?

Frage 2. Was ist die Aufgabe des Compilers?

Frage 3. Was ist die Aufgabe der JVM?

Frage 4. Erläutern Sie die Unterschiede zwischen Quelldatei und Bytecode.

Kapitel 2

Imperative Konzepte

2.1 Datentypen und Operatoren

■ **Aufgabe 3.** (Klasse: Seconds) Erstellen Sie ein Programm, das die Anzahl der Sekunden pro Stunde, pro Tag, pro Woche und im Monat April berechnet und in Variablen speichert. Erzeugen Sie mittels der berechneten Werte folgende Konsolenausgabe:

```
Sekunden pro Stunde : 3600
Sekunden pro Tag   : 86400
Sekunden pro Woche : 604800
Sekunden im April  : 2592000
```

■ **Aufgabe 4.** (Klasse: Diode) Der Vorwiderstand $R = 150 \Omega$ einer Diode werde von einem Strom $I = 20 \text{ mA}$ durchflossen. Erstellen Sie ein Programm, das

- den Spannungsabfall U über R sowie
- die dem Widerstand zugeführte Leistung P

berechnet und ausgibt.

■ **Aufgabe 5.** (Klasse: ParallelResistors) Schreiben Sie ein Programm, das den Gesamtwiderstand R mit

$$\frac{1}{R} = \frac{1}{R_1} + \frac{1}{R_2} = \frac{R_1 + R_2}{R_1 R_2} \quad (2.1)$$

zweier parallel geschalteter Widerstände $R_1 = 150 \Omega$ und $R_2 = 220 \Omega$ berechnet und ausgibt.

■ **Aufgabe 6.** (Klasse: BinaryWeights) Schreiben Sie ein Programm, das die Gewichte 2^n der Ziffern im binären Zahlensystem für $n \in \mathbb{N}_0$, $n < 8$, berechnet und ausgibt. Beispielausgabe:

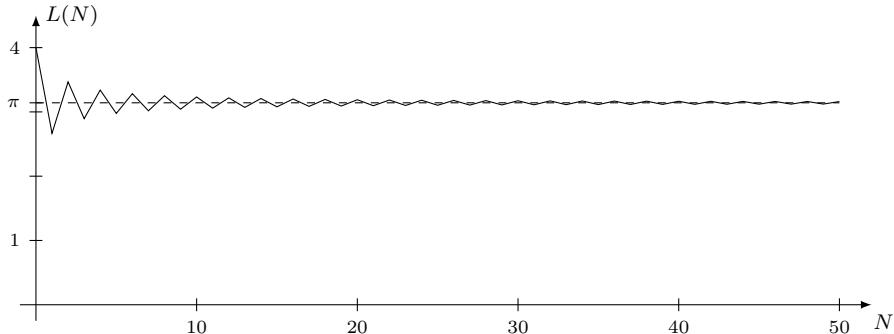
```
2^0 = 1
2^1 = 2
2^2 = 4
2^3 = 8
2^4 = 16
2^5 = 32
2^6 = 64
2^7 = 128
```

■ **Aufgabe 7.** (Klasse: TeamPlayers) Die Schüler und Schülerinnen aus vier Klassen sollen für ein Sportturnier auf acht Teams aufgeteilt werden. Bestimmen Sie mittels eines Programms,

- wie viele Personen jedes Team umfasst und
- wie viele Personen bei gleich großen Teams übrig bleiben

für den Fall, dass zwei Klassen aus jeweils 24 Kindern und die übrigen beiden Klassen aus 25 bzw. 26 Kindern bestehen. Beispielausgabe:

```
99 players distributed to 8 teams result in
12 players per team with 3 player(s) remaining.
```

Abbildung 2.1: Reihenentwicklung $L(N)$ der Kreiszahl π nach Leibniz (Aufgabe 8)

■ **Aufgabe 8.** (Klasse: LeibnizSeries) Nach Gottfried Wilhelm Leibniz lässt sich die Kreiszahl π über folgende Reihenentwicklung mit $N \rightarrow \infty$ bestimmen (Abb. 2.1):

$$\pi \approx 4 \cdot \sum_{n=0}^N \frac{(-1)^n}{2n+1} = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \cdots + \frac{4 \cdot (-1)^N}{2N+1} \quad (2.2)$$

Schreiben Sie ein Programm, das alle Näherungen $N = 0$ bis $N = 6$ berechnet und ausgibt. Beispieldaten:

```
n = 0: pi = 4.0
n = 1: pi = 2.6666666666666667
n = 2: pi = 3.4666666666666667
n = 3: pi = 2.8952380952380956
n = 4: pi = 3.3396825396825403
n = 5: pi = 2.9760461760461765
n = 6: pi = 3.2837384837384844
```

Formatierte Ausgabe

Über die Methode `System.out.printf()` lassen sich formatierte Konsolenausgaben erzeugen. Die Verwendung ist (fast) identisch mit der `printf()`-Funktion der Programmiersprache C.

■ **Aufgabe 9.** (Klasse: PeriodSeconds) Schreiben Sie ein Programm, das eine in einer Variablen gegebene Zeitdauer in Sekunden im Format *hh:mm:ss* (*h*: *hours*, *m*: *minutes*, *s*: *seconds*) auf Konsole ausgibt. Hinweis: Der Formatspezifikator `%02d` formatiert eine Ganzzahl derart, dass diese aus mindestens zwei Ziffern besteht und gegebenenfalls eine Null vorangestellt wird.

Beispieldaten:

```
A period of 3867 seconds corresponds to 1:04:27 hours.
```

■ **Aufgabe 10.** (Klasse: LogicLevel) Beim Anschluss eines Bluetooth-Moduls HC-06 an einen Arduino Nano R3 muss der Spannungsspeicher der RX-Signalleitung von 5 V auf etwa 3,3 V abgesenkt werden. Der einfachste Ansatz wäre ein Spannungsteiler mit (Abb. 2.3)

$$U_{RX} = \frac{R_2}{R_1 + R_2} \cdot U_0 . \quad (2.3)$$

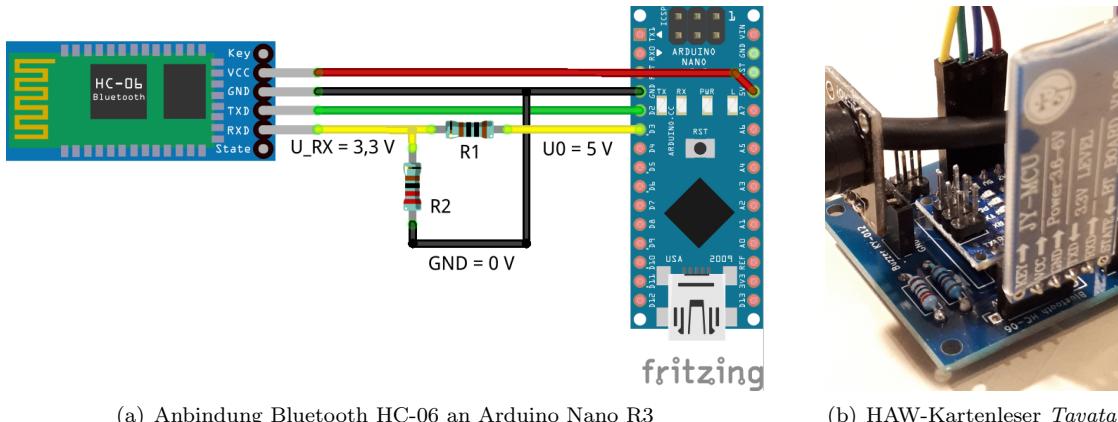
Berücksichtigt man allerdings den Stromfluss I_{RX} in den *RXD*-Pin, ergibt sich

$$U_{RX} = \frac{R_2}{R_1 + R_2} \cdot (U_0 - R_1 I_{RX}) . \quad (2.4)$$



(a) Mobile Spielstandsanzeige mit Zeitnahme (Prototyp)

(b) Android-App

Abbildung 2.2: Beispielanwendung zur Darstellung im Format $hh:mm:ss$ (Aufgabe 9)

(a) Anbindung Bluetooth HC-06 an Arduino Nano R3

(b) HAW-Kartenleser Tavata

Abbildung 2.3: Anpassung des logischen Spannungspiegels von 5 V auf 3,3 V (Aufgabe 10)

Schreiben Sie ein Programm, das für gegebene Werte der Widerstände R_1 und R_2 die Spannung U_{RX} jeweils für die Lastströme $I_{RX} = 0 \text{ mA}$ sowie $I_{RX} = 0,5 \text{ mA}$ ausgibt. Welche der Widerstandsgrößen $1 \text{ k}\Omega$, $1,5 \text{ k}\Omega$, $2,2 \text{ k}\Omega$ und $4,7 \text{ k}\Omega$ sollten Sie für R_1 und R_2 verwenden, wenn der Pegel U_{RX} nicht unter 3 V fallen darf?

Zufallszahlen

Der Aufruf der Methode `random()` der Klasse `Math` erzeugt eine Zufallszahl r vom Typ `double` im Bereich $0 \leq r < 1$.

Aufgabe 11. (Klasse: ThrowDice) Lassen Sie uns spielen! Genauer gesagt wollen wir würfeln. Erzeugen Sie auf nachfolgende Arten zufällige Würfe $n \in \mathbb{N}$, $n \leq 6$, eines Würfels und geben Sie diese auf Konsole aus:

- Erzeugte Zufallszahlen werden zunächst mit dem Faktor 6 skaliert (d. h. $6.0 * \text{Math.random}()$).
- Erzeugte Zufallszahlen werden zunächst mit 6000 skaliert (d. h. $6000.0 * \text{Math.random}()$).

2.2 Programmfluss

■ **Aufgabe 12.** (Klasse: LeapYear) Der 29. Februar existiert nur in Schaltjahren. Ein Jahr ist dann ein Schaltjahr, wenn die Jahreszahl durch 4 teilbar ist, ausgenommen sie besitzt den Teiler 100, aber nicht den Teiler 400. Schreiben Sie ein Programm, das überprüft, ob ein eingegebenes Jahr ein Schaltjahr ist.

■ **Aufgabe 13.** (Klasse: LeapYearLogical) Finden Sie eine Lösung zu Aufgabe 12, bei der zur Bestimmung, ob es sich um ein Schaltjahr handelt, keine *if*-Anweisungen verwendet werden.

■ **Aufgabe 14.** (Klasse: MathCircle) Legen Sie in einem Programm Variablen für den Radius r eines Kreises sowie die Kreiszahl π an. Stellen Sie sicher, dass der Wert der Variable für π nach einer Zuweisung nicht mehr geändert werden kann. Geben Sie den Umfang $U = 2\pi r$ sowie die Fläche $A = \pi r^2$ des Kreises auf Konsole aus. Ist der Radius negativ, soll eine Fehlermeldung ausgegeben werden. Überprüfen Sie die Ausgaben anhand ausgewählter Radien.

■ **Aufgabe 15.** (Klasse: LoopTypes) Bilden Sie die Summe der Zahlen von 1 bis 100 und verwenden Sie hierfür eine a) *for*-Schleife, b) *while*-Schleife und c) *do/while*-Schleife.

■ **Aufgabe 16.** (Klasse: PrintNumbers) Geben Sie jeweils aufsteigend in der ersten Zeile alle Zahlen, in der zweiten Zeile alle ungeraden Zahlen und in der dritten Zeile alle geraden Zahlen im Bereich $n \leq 20$, $n \in \mathbb{N}$, auf Konsole aus.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	3	5	7	9	11	13	15	17	19										
2	4	6	8	10	12	14	16	18	20										

■ **Aufgabe 17.** (Klasse: BinaryWeightsLoop) Implementieren Sie die Ausgabe der ersten acht Faktoren 2^n im binären Zahlensystem gemäß Aufgabe 6 auf Seite 11 unter Verwendung einer Schleife.

Formatierte Ausgabe

Durch Aufruf von `System.out.printf()` mit zusätzlichem ersten Parameter `Locale.US` werden Dezimalpunkte statt Dezimalkommata erzeugt.

■ **Aufgabe 18.** (Klasse: BankAccount) Ein Konto hat ein Guthaben von 1.000,- €. Nach wie vielen Jahren hätte sich das Guthaben bei einer jährlichen Verzinsung von 0,35 % verdoppelt¹ (Abb. 2.4)?

■ **Aufgabe 19.** (Klasse: DiceDistribution) In Aufgabe 11 haben wir zufällige Würfe eines Würfels simuliert. Nun wollen wir überprüfen, ob die Augenzahlen in etwa gleich häufig vorkommen². Schreiben Sie ein Programm, das $k \in \mathbb{N}$ mal würfelt und sowohl k als auch die prozentuale Verteilung der gewürfelten Zahlen auf Konsole ausgibt. Beispielausgabe:

```
Anzahl: 100 Millionen
Augenzahl 1: 16.664603 %
Augenzahl 2: 16.669071 %
Augenzahl 3: 16.661852 %
Augenzahl 4: 16.671382 %
Augenzahl 5: 16.664543 %
Augenzahl 6: 16.668549 %
```

¹Die traurige Wahrheit ist, dass dies erst nach 199 Jahren der Fall ist.

²Ist dies nicht der Fall, überprüfen Sie Ihre Implementierung in Aufgabe 11.

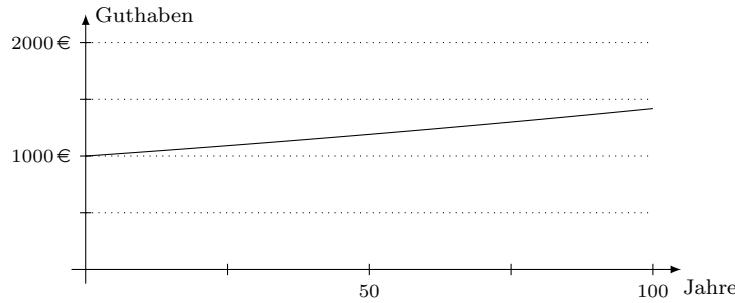


Abbildung 2.4: Entwicklung des Guthabens in den ersten 100 Jahren (Aufgabe 18)

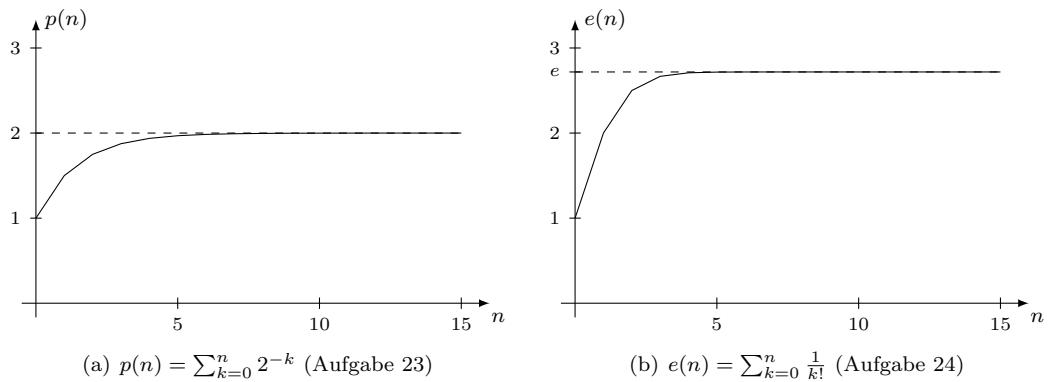


Abbildung 2.5: Mathematische Reihen

Aufgabe 20. (Klasse: ByteRange) Bestimmen Sie durch ein Programm den Wertebereich, d. h. die kleinste und größte darstellbare Zahl, des Datentyps *byte*. Hinweis: Durchlaufen Sie den Zahlenbereich in einer *while*-Schleife. Was passiert sobald der Zahlenbereich überschritten wird?

Aufgabe 21. (Klasse: FloatGap) Bestimmen Sie durch ein Programm die erste positive ganze Zahl, die nicht als *float* gespeichert werden kann.

Aufgabe 22. (Klasse: LeibnizSeriesLoop) Wir kommen zurück auf die Reihenentwicklung der Kreiszahl π nach Leibniz in Gleichung (2.2) auf Seite 12. Geben Sie die Näherungen für $N = 0$ bis $N = 10$ sowie für $N \in \{250, 500, 750, 1000\}$ aus.

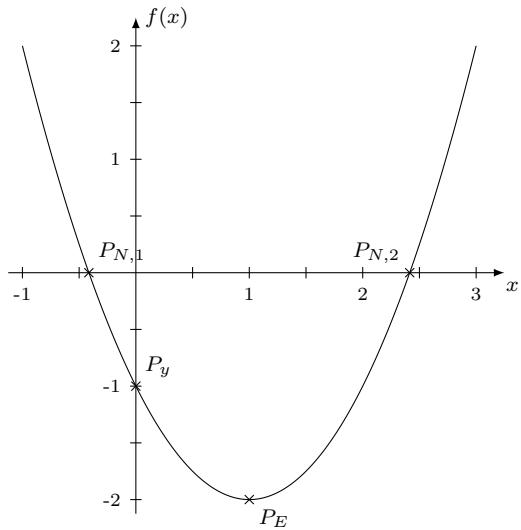
Aufgabe 23. (Klasse: MathSeries) Approximieren Sie den Wert der mathematischen Reihe (Abb. 2.5(a))

$$p = \sum_{k=0}^{\infty} 2^{-k}. \quad (2.5)$$

Nach wie vielen Summanden ändert sich p um weniger als $\Delta p = 10^{-10}$?

Aufgabe 24. (Klasse: EulerNumber) Die Euler-Zahl e kann durch die Reihe (Abb. 2.5(b))

$$e \approx \sum_{k=0}^n \frac{1}{k!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{n!} \quad (2.6)$$

Abbildung 2.6: $f(x) = x^2 - 2x - 1$ (aus M. Hensel: *Kurvendiskussion*, 2. Auflage, 2012)

mit

$$m! = \prod_{p=1}^m p = 1 \cdot 2 \cdot 3 \cdots \cdot m \quad (2.7)$$

für $m \geq 1$ und $0! = 1$ approximiert werden. Implementieren Sie die Approximation und überprüfen Sie diese anhand der Werte für $n \in \mathbb{N}_0$, $n \leq 5$. Beispielausgabe:

```
Approximation of Euler number:
n = 0: 1.000000000
n = 1: 2.000000000
n = 2: 2.500000000
n = 3: 2.666666667
n = 4: 2.708333333
n = 5: 2.716666667
```

■ Aufgabe 25. (Klasse: BisectionMethod) Die Funktion

$$f(x) = x^2 - 2x - 1 \quad (2.8)$$

besitzt eine Nullstelle x_0 mit $f(x_0) = 0$ auf der positiven x -Achse ($P_{N,2}$ in Abb. 2.6). Approximieren Sie x_0 mittels Bisektionsverfahren:

1. Berechnen Sie $f(x)$ für $x_l = 0$ und $x_r = 4$. Da sich die Vorzeichen von $f(x_l)$ und $f(x_r)$ unterscheiden, liegt im Intervall $x \in [x_l, x_r]$ eine Nullstelle.
2. Berechnen Sie $f(x_m)$ in der Mitte von x_l und x_r , d. h. für $x_m = \frac{1}{2}(x_l + x_r)$.
3. Wählen Sie als nächstes, halbiertes Intervall $[x_m, x_r]$ oder $[x_l, x_m]$, sodass die Funktionswerte $f(x)$ an den Intervall-Grenzen weiterhin unterschiedliche Vorzeichen haben.
4. Wiederholen Sie ab Schritt 2. bis die Länge des Intervalls $\Delta x < 10^{-6}$ beträgt.

Beispielausgabe:

```
Zero-crossing for f(x) = x^2 - 2x - 1 by bisection method:
Stopped at interval size: 0.000001000
Approximated x0 : 2.414213657
Approximated f(x0) : 0.000000269
```

2.3 Fragen

Datentypen

Frage 5. Nennen Sie alle ganzzahligen primitiven Datentypen.

Frage 6. In welchem Format sind Zeichen (Datentyp *char*) kodiert? Wie viele Byte Speicher werden pro Zeichen benötigt?

Frage 7. Welchen Datentyp würden Sie verwenden, um 1 Byte-Werte von 0 bis 255 zu speichern?

Frage 8. Welche Ergebnis liefert die Operation $7/2$? Welches Ergebnis liefert $7./2$?

Frage 9. Welchen Datentyp besitzt das Ergebnis der Division eines *int*-Wertes durch einen *double*-Wert?

Frage 10. Welchen Datentyp besitzt das Ergebnis der Division eines *double*-Wertes durch einen *int*-Wert?

Frage 11. Welchen Wert besitzt der folgende Ausdruck? (*int*) $7.1 + 3.5$

Frage 12. Welchen Wert besitzt der folgende Ausdruck? $(3 + 4) / 2 * 3$

Frage 13. Welchen Wert besitzt der folgende Ausdruck? $3 * 3 + 4 / 2$

Frage 14. Welche logischen Werte werden durch folgende Zahlen repräsentiert? 0, 1, 2, 745, -1

Frage 15. Geben Sie drei unterschiedliche Ansätze an, um den Wert der ganzzahligen Variable *a* um 1 zu erhöhen.

Frage 16. Sie verwenden in Kommentaren deutsche Umlaute. Welche Zeichen werden bei einem Freund in Australien dargestellt, wenn er Ihr Software-Projekt auf seinem PC importiert?

Programmfluss

Frage 17. Ist es möglich, eine *switch*-Anweisung über einen *char*-Wert zu steuern? Ist es möglich, diese über einen *double*-Wert zu steuern?

Frage 18. Was bewirkt *break* innerhalb einer *switch*-Anweisung?

Frage 19. Was bewirkt *continue* innerhalb einer *switch*-Anweisung?

Frage 20. Nennen Sie alle Arten von Schleifen.

Frage 21. Wann verwendet man am besten eine *for*-Schleife, wann eine *while*-Schleife?

Frage 22. Erläutern Sie den wesentlichen Unterschied zwischen einer *while*-Schleife und einer *do/while*-Schleife.

Frage 23. Was bewirkt die *break*-Anweisung in Schleifen? Was bewirkt die *continue*-Anweisung?

Frage 24. Eine *while*-Schleife ist derart programmiert, dass Sie terminiert, falls eine ganzzahlige Variable *k* den Wert 100 überschreitet. Wird die Schleife jemals abbrechen, falls der Schleifenkörper eine *continue*-Anweisung enthält?

Praktikumsaufgaben (Einführung)

Bearbeiten Sie nun das Praktikum auf Seite 82. Stellen Sie sicher, dass Sie das Praktikum lösen können, bevor Sie fortfahren.

Kapitel 3

Klassen und Objekte

3.1 Variablen

■ **Aufgabe 26.** (Klasse: PersonApp) Definieren Sie eine Klasse *Person* zur Repräsentation einer Person. Die Klasse enthält als Attribute den Vornamen und Nachnamen. Erstellen Sie zudem ein Programm, das zwei Objekte vom Typ *Person* erzeugt, initialisiert und die Namen auf Konsole ausgibt. Beispielausgabe:

```
There are objects for these two people:  
Marc Hensel  
Dido Armstrong
```

■ **Aufgabe 27.** (Klasse: BookApp) Definieren Sie eine Klasse *Book* zur Repräsentation eines Buches. Die Klasse enthält als Attribute den Titel sowie den Vornamen und Nachnamen des Autors bzw. der Autorin. Erstellen Sie zudem ein Programm, das ein Objekt der Klasse *Book* erzeugt, initialisiert und die Attribute auf Konsole ausgibt. Beispielausgabe:

```
There is an object for following book:  
Horst Evers: "Wäre ich du, würde ich mich lieben"
```

■ **Aufgabe 28.** (Klasse: SongApp) Definieren Sie eine Klasse *Song* zur Repräsentation eines Musiktitels. Wie schon die Klasse *Book* aus Aufgabe 27 beinhaltet sie einen vollständigen Namen (Interpret bzw. Interpretin) sowie einen Titel. Abweichend von *Book* soll der Name jedoch nicht in zwei Variablen vom Typ *String*, sondern in einem Objekt vom Typ *Person* aus Aufgabe 26 gespeichert werden. Erstellen Sie ein Programm, das Objekte der Klasse *Song* erzeugt, initialisiert und die Attribute auf Konsole ausgibt. Beispielausgabe:

```
Playlist:  
Isobel (Dido Armstrong)  
I shall believe (Sheryl Crow)
```

3.2 Methoden und Konstruktoren

■ **Aufgabe 29.** (Klasse: BookApp) Erweitern Sie die Klasse *Book* aus Aufgabe 27 um Getter und Setter für die Attribute (Abb. 3.1). Passen Sie das ausführbare Programm derart an, dass ausschließlich über Getter und Setter auf die *Book*-Attribute zugegriffen wird.

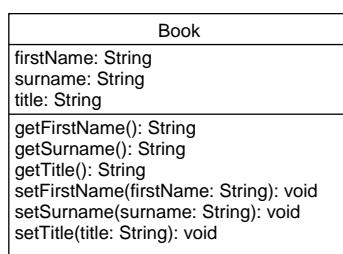
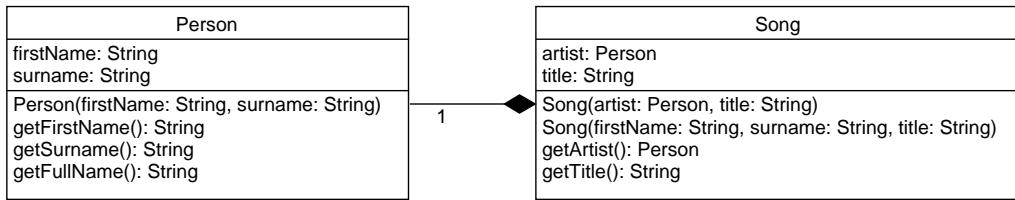


Abbildung 3.1: Klassendiagramm der Klasse *Book* (Aufgabe 29)

Abbildung 3.2: Klassendiagramme für *Person* (Aufgabe 30) und *Song* (Aufgabe 31)

■ **Aufgabe 30.** (Klasse: PersonApp) Erweitern Sie die Klasse *Person* aus Aufgabe 26 um folgende Methoden (Abb. 3.2):

- ▶ Konstruktor, dem der Vorname und der Nachname übergeben werden
- ▶ Getter für die Attribute
- ▶ Methode *getFullName()*, die Vor- und Nachnamen durch Leerzeichen trennt zurückgibt

Passen Sie das ausführbare Programm derart an, dass ausschließlich über den Konstruktor und die Methoden auf die Attribute zugegriffen wird. Verwenden Sie für die Konsolenausgabe für das erste Objekt die Getter und für das zweite Objekt die Methode *getFullName()*.

■ **Aufgabe 31.** (Klasse: SongApp) Erweitern Sie die Klasse *Song* aus Aufgabe 28 um folgende Methoden (Abb. 3.2):

- ▶ Konstruktor, dem ein *Person*-Objekt gemäß Aufgabe 30, sowie der Titel übergeben wird
- ▶ Konstruktor, dem Vorname und Nachname sowie der Titel übergeben wird
- ▶ Getter für die Attribute vom Typ *Person* und *String*

Passen Sie das ausführbare Programm derart an, dass beide Konstruktoren zur Instantiierung von Objekten verwendet werden und ausschließlich über die Methoden *getTitle()* und *getFullName()* lesend auf die Attribute zugegriffen wird.

■ **Aufgabe 32.** (Klasse: SongApp) Passen Sie den zweiten Konstruktor der Klasse *Song* aus Aufgabe 31 derart an, dass er zur Instantiierung eines Objektes den ersten Konstruktor aufruft.

■ **Aufgabe 33.** (Klasse: TimePeriodApp) In Aufgabe 9 auf Seite 12 haben wir Zeitdauern im Format *hh:mm:ss* betrachtet. Definieren Sie eine Klasse *TimePeriod* gemäß Abbildung 3.3 zur Repräsentation einer Dauer. Erstellen Sie zudem ein Programm, das *TimePeriod* exemplarisch verwendet. Beispieldausgabe:

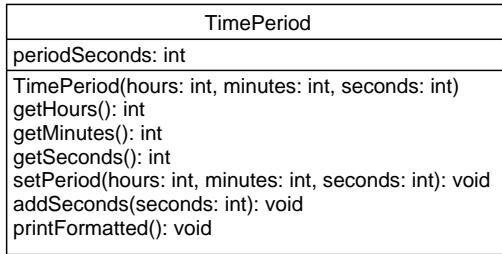
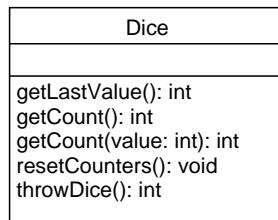
```

TimePeriod(1, 18, 6)      : 1:18:06
addSeconds(61)           : 1:19:07
setPeriod(0, 61, 3606)   : 2:01:06

```

■ **Aufgabe 34.** (Klasse: DiceApp) In Aufgabe 11 auf Seite 13 haben wir kennengelernt, wie sich zufällige Ergebnisse beim Würfeln simulieren lassen. In dieser Aufgabe sollen Sie eine entsprechende Klasse erstellen, die zudem über ein „Gedächtnis“ verfügt, wie oft jede Zahl bislang gewürfelt wurde. Erstellen Sie hierfür eine Klasse *Dice* gemäß folgenden Anforderungen (Abb. 3.4):

- ▶ Es existiert eine Variable zum Speichern des Ergebnisses des letzten Wurfes.
- ▶ Es existieren Variablen zum Zählen wie oft die Zahlen von 1 bis 6 geworfen wurden.

Abbildung 3.3: Klassendiagramm für die Klasse *TimePeriod* (Aufgabe 33)Abbildung 3.4: Klassendiagramm für *Dice* ohne Angabe der Attribute (Aufgabe 34)

- Die Methode *throwDice()* führt einen Wurf aus und gibt das Ergebnis zurück.
- Die Methode *getLastValue()* gibt die zuletzt gewürfelte Zahl zurück.
- Die Methode *resetCounters()* setzt alle Zähler auf null zurück.
- Die Methode *getCount()* gibt zurück, wie oft die als Argument übergebene Zahl seit Instantiierung des Objektes bzw. dem letzten Reset gewürfelt wurde.
- Die parameterlose Methode *getCount()* gibt zurück wie oft seit Instantiierung des Objektes bzw. dem letzten Reset gewürfelt wurde.

Erstellen Sie zudem ein Programm, das ein Objekt der Klasse *Dice* erzeugt und die Ergebnisse einiger Würfe inklusive der Zählerstände auf Konsole ausgibt. Beispieldaten bei Reset der Zähler nach dem zweiten Wurf:

Dice	#1	#2	#3	#4	#5	#6	Overall
5	0	0	0	0	1	0	1
6	0	0	0	0	1	1	2
- Reset counters -							
3	0	0	1	0	0	0	1
1	1	0	1	0	0	0	2
6	1	0	1	0	0	1	3
1	2	0	1	0	0	1	4

■ **Aufgabe 35.** (Klasse: ParabolaApp) Deklarieren Sie eine Klasse *Parabola* zur Repräsentation eines mathematischen Polynoms 2. Grades (Abb. 3.5). Die Klasse stellt die Methoden *f()* und *f1()* zur Bestimmung des Funktionswertes

$$f(x) = a_2x^2 + a_1x + a_0 \quad (3.1)$$

bzw. des Wertes

$$f'(x) = 2a_2x + a_1 \quad (3.2)$$

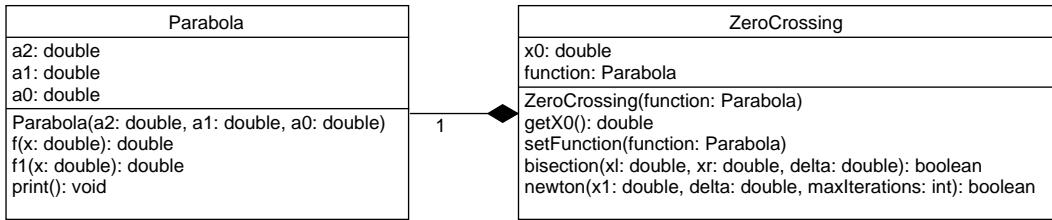


Abbildung 3.5: Klassendiagramme der Klassen *Parabola* (Aufgabe 35) und *ZeroCrossing* (Aufgaben 36 und 37)

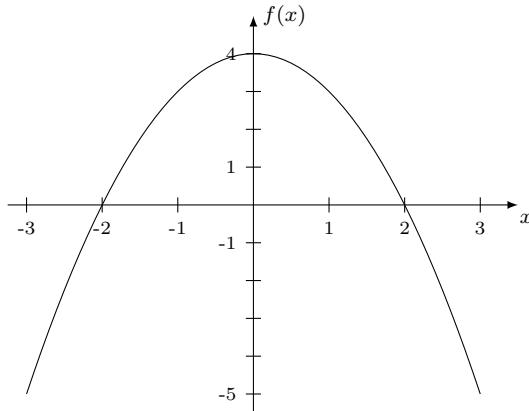


Abbildung 3.6: $f(x) = -x^2 + 4$ (Aufgaben 35, 36 und 37)

der ersten Ableitung zur Verfügung, während *print()* die Formel $f(x)$ auf Konsole ausgibt.

Erstellen Sie zudem ein Programm, das mittels eines Objektes vom Typ *Parabola* eine Wertetabelle auf Konsole ausgibt. Beispieldarstellung für $f(x) = -x^2 + 4$ (Abb. 3.6):

Selected values for $f(x) = -1 * x^2 + 0 * x + 4$:

x	-4	-3	-2	-1	0	1	2	3	4
$f(x)$	-12	-5	0	3	4	3	0	-5	-12

■ **Aufgabe 36.** (Klasse: MathBisectionApp) In Aufgabe 25 auf Seite 16 haben wir die Approximation von Nullstellen x_0 einer Funktion $f(x)$ mittels Bisektionsverfahrens kennengelernt. Im Folgenden kombinieren wir das Verfahren mit der Klasse *Parabola* aus Aufgabe 35. Hierdurch kann die zu analysierenden Funktion $f(x) = a_2x^2 + a_1x + a_0$ über die Parameter a_2 , a_1 und a_0 festgelegt werden.

Deklarieren Sie eine Klasse *ZeroCrossing* zur Bestimmung der Nullstellen von Polynomen 2. Grades (Abb. 3.5, ausgenommen der Methode *newton()*):

- ▶ Die Klasse besitzt ein Attribut zum Speichern der ermittelten Nullstelle x_0 sowie ein Objekt vom Typ *Parabola*, das die zu analysierenden Funktion $f(x)$ repräsentiert.
- ▶ Der Konstruktor initialisiert die zu analysierende Funktion $f(x)$.
- ▶ Die Methode *getX0()* gibt die ermittelte Nullstelle x_0 zurück.
- ▶ Über die Methode *setFunction()* kann die zu analysierende Funktion $f(x)$ geändert werden.
- ▶ Die Methode *bisection()* führt das Bisektionsverfahren gemäß Aufgabe 25 durch. Sie gibt *true* zurück, falls das Verfahren erfolgreich war, andernfalls *false*.

Beispielausgabe für die Analyse der Funktion $f(x) = -x^2 + 4$ (Abb. 3.6) mit dem Start-Intervall $x_l = 0$ und $x_r = 4$ sowie Abbruch der Methode bei $\Delta x < 10^{-6}$:

```
Zero-crossing for f(x) = -1 * x^2 +0 * x +4:  
Approximated x0 : 2.000000  
Approximated f(x0): 1.907e-06
```

■ **Aufgabe 37.** (Klasse: MathNewtonApp) In Aufgabe 36 haben wir Nullstellen mathematischer Funktionen mittels Bisektionsverfahren angenähert. Ein weiteres verbreitetes Verfahren ist die Methode nach Newton¹. Hierbei wird die Nullstelle ausgehend von einem Startwert x_0 iterativ wie folgt genähert:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad (3.3)$$

Erweitern Sie die Klasse *ZeroCrossing* um das Newtonverfahren. Hierfür erhält die Methode Methode *newton()* als Argumente den Startwert x_0 , das Stoppkriterium Δx sowie die maximale Anzahl an Iterationen. Das Verfahren wird beendet, wenn sich der aktuelle x -Wert in einem Schritt um weniger als Δx verändert hat (d. h. $|x_{n+1} - x_n| < \Delta x$) oder die maximale Anzahl Iterationen erreicht wurde. Beispielausgabe für $f(x) = -x^2 + 4$ (Abb. 3.6) mit dem Startwert $x_0 = 1$:

```
Zero-crossing for f(x) = -1 * x^2 +0 * x +4:  
Approximated x0 : 2.000000  
Approximated f(x0): -8.882e-15
```

3.3 Klassenvariablen und Klassenmethoden

■ **Aufgabe 38.** (Klasse: MathFunctionsApp) In Aufgabe 24 auf Seite 15 haben wir die Fakultät

$$n! = \prod_{p=1}^n p = 1 \cdot 2 \cdot 3 \cdots \cdot n \quad (3.4)$$

mit $n \geq 1$ und $0! = 1$ betrachtet. Deklarieren Sie eine Klasse *MathFunctions*, die eine Methode *factorial()* zur Bestimmung der Fakultät einer Zahl $n \in \mathbb{N}_0$ enthält. Die Methode soll sich ohne Instantiierung eines Objektes vom Typ *MathFunctions* aufrufen lassen. Erstellen Sie zudem ein Programm, das die Fakultäten ausgewählter Werte auf Konsole ausgibt. Beispielausgabe:

```
0! = 1  
1! = 1  
2! = 2  
3! = 6  
4! = 24  
5! = 120
```

■ **Aufgabe 39.** (Klasse: MemberApp) Definieren Sie eine Klasse *Member* zur Repräsentation von Mitgliedern beispielsweise eines Vereins (Abb. 3.7):

- Der Name wird in einem Objekt vom Typ *Person* gemäß Aufgabe 30 gespeichert.
- Die Klasse besitzt ein Attribut für die Mitgliedsnummer sowie eine Klassenvariable, welche die nächste zuzuweisende Mitgliedsnummer enthält.
- Die Klasse besitzt einen Konstruktor, dem der Vor- und Nachname übergeben wird.
- Die Klasse besitzt Methoden zur Rückgabe von Vorname, Nachname und Mitgliedsnummer.
- Es existiert eine Klassenmethode, die die Anzahl der bisherigen Mitglieder zurückgibt.

¹<https://de.wikipedia.org/wiki/Newtonverfahren> (Besucht am 20.02.2021)

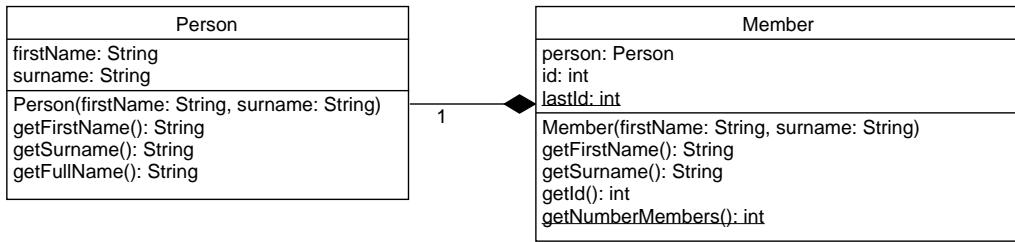
Abbildung 3.7: Klassendiagramm der Klasse *Member* (Aufgabe 39)

Tabelle 3.1: Frequenzen der nullten Oktave der C-Dur-Tonleiter (Aufgabe 40)

Ton	C_0	D_0	E_0	F_0	G_0	A_0	B_0
Frequenz [Hz]	16,3516	18,3540	20,6017	21,8268	24,4997	27,5000	30,8677

Erstellen Sie zudem ein Programm, das Mitglieder instantiiert und mit Namen und Mitgliedsnummer auf Konsole ausgibt. Beispielausgabe:

```
We have 1 member(s):
Hensel, Marc (Member 1)
```

```
We have 2 member(s):
Hensel, Marc (Member 1)
Musterfrau, Jana (Member 2)
```

■ **Aufgabe 40.** (Klasse: MusicNotesApp) Töne eines Musikinstrumentes bestehen aus sinusförmigen Wellen, wobei die Tonhöhe von der Frequenz abhängig ist. Je schneller die Schwingung, desto höher nehmen wir den Ton wahr. Zur künstlichen Erzeugung von Musik zum Beispiel mittels eines Synthesizers muss die Frequenz des jeweiligen Tones bekannt sein.

In der *C-Dur-Tonleiter* erhalten die Grundtöne (*Tonika*) mit zunehmender Höhe die alphabetischen Bezeichnungen *C*, *D*, *E*, *F*, *G*, *A* und *B* (im Deutschen meist *H* anstatt *B*). Um einen weiten Tonbereich abzudecken, wird für den sogenannten *Oktavraum* eine mit der Höhe ansteigende Ziffer 0, 1, 2, ... angefügt:

$$C_0 \rightarrow D_0 \rightarrow E_0 \rightarrow F_0 \rightarrow G_0 \rightarrow A_0 \rightarrow B_0 \rightarrow C_1 \rightarrow \dots \rightarrow B_1 \rightarrow C_2 \rightarrow \dots \rightarrow B_2 \rightarrow \dots$$

Tabelle 3.1 gibt die Frequenzen der Grundtöne an². Die Frequenz eines Tons in der jeweils nächsten Oktave erfolgt durch Verdopplung, zum Beispiel:

$$C_2 = 2 \cdot C_1 = 2 \cdot 2 \cdot C_0 = 4 \cdot 16,3516 \text{ Hz} = 65,4064 \text{ Hz}$$

Definieren Sie eine Klasse *MusicNotes* mit einer Klassenmethode `tone2FrequencyHz()`, die zu einem als Argument übergebenen Ton ('a' bis 'g' bzw. 'A' bis 'G') sowie der Oktave (0, 1, 2, ...) die Frequenz in Herz zurückgibt. Erstellen Sie zudem ein Programm, das mittels `tone2FrequencyHz()` die Frequenztabelle für die ersten sechs Oktaven auf Konsole ausgibt. Beispielausgabe:

```
Frequencies of musical notes in Hz:
```

	0	1	2	3	4	5
C	16.3516	32.7032	65.4064	130.8128	261.6256	523.2512
D	18.3540	36.7080	73.4160	146.8320	293.6640	587.3280
E	20.6017	41.2034	82.4068	164.8136	329.6272	659.2544
F	21.8268	43.6536	87.3072	174.6144	349.2288	698.4576
G	24.4997	48.9994	97.9988	195.9976	391.9952	783.9904
A	27.5000	55.0000	110.0000	220.0000	440.0000	880.0000
B	30.8677	61.7354	123.4708	246.9416	493.8832	987.7664

²https://de.wikipedia.org/wiki/Frequenzen_der_gleichstufigen_Stimmung (Besucht am 31.10.2020)

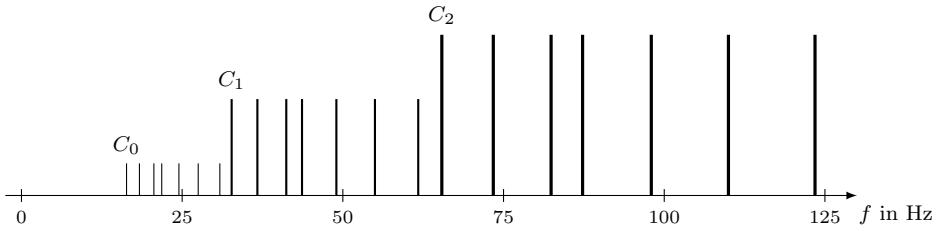


Abbildung 3.8: Frequenzen der ersten drei Oktaven der C-Dur-Tonleiter (Aufgabe 40)

3.4 Fragen

Klassen und Objekte

Frage 25. Was ist der Unterschied zwischen einer Klasse und einem Objekt?

Frage 26. Was ist der Unterschied zwischen einem Objekt und einer Instanz?

Frage 27. Wie nennt man den Mechanismus, der von Objekten allozierten Speicher wieder freigibt? Welche Bedingung muss für die Freigabe des Speichers erfüllt sein?

Konstruktoren

Frage 28. Was ist ein Standardkonstruktor?

Frage 29. Wann besitzt eine Klasse einen Standardkonstruktor?

Frage 30. Wie viele Konstruktoren hat eine Klasse, wenn im Quelltext der Klasse kein Konstruktor angegeben wurde?

Frage 31. Kann eine Klasse keinen Konstruktor besitzen?

Frage 32. Wie kann man in einem Konstruktor einen anderen Konstruktor der eigenen Klasse aufrufen?

Variablen und Methoden

Frage 33. Was ist der wesentliche Unterschied zwischen einer Referenzvariable und einer Variable eines primitiven Datentyps?

Frage 34. Nennen Sie drei beliebige im Java SDK enthaltene Referenzdatentypen.

Frage 35. Was muss erfüllt sein, damit eine Variable automatisch initialisiert wird?

Frage 36. Durch welches Schlüsselwort werden Klassenvariablen gekennzeichnet?

Frage 37. Wie unterscheiden sich Klassenvariablen von Instanzvariablen?

Frage 38. Nehmen Sie Stellung zu folgender Aussage: „Die `toString()`-Methode einer Klasse gibt immer die Werte der Instanzvariablen auf Konsole aus.“

Frage 39. Kann in einer statischen Methode die `this`-Referenz verwendet werden?

Frage 40. Was ist der Unterschied zwischen `this` und `this()`?

Frage 41. Gegeben seien nachfolgende Definitionen. Referenzieren die Variablen `a` und `b` dasselbe Objekt?

```
int a = 15;
int b = a;
```

Praktikumsaufgaben (Casino, Aufgabenteil „Einarmige Banditen“)

Bearbeiten Sie nun Praktikum 1 auf Seite 85.

- ▶ Stellen Sie sicher, dass Sie das Praktikum lösen können, bevor Sie fortfahren.
- ▶ Es ist sinnvoll auch die weiteren Versuche zum Praktikum 1 zu bearbeiten.

Praktikumsaufgaben (Geografische Koordinaten)

Bearbeiten Sie nun Praktikum 1 auf Seite 102.

- ▶ Stellen Sie sicher, dass Sie das Praktikum lösen können, bevor Sie fortfahren.
- ▶ Es ist sinnvoll auch die weiteren Versuche zum Praktikum 1 zu bearbeiten.

Kapitel 4

Ausgewählte Klassen

4.1 Zeichenketten

■ **Aufgabe 41.** (Klasse: PersonApp) Erweitern Sie die Klasse *Person* aus Aufgabe 30 um eine *toString()*-Methode, die den vollen Namen der jeweiligen Person im Format "Vorname Nachname" zurückgibt. Erstellen Sie zudem ein Programm, das ein Objekt der Klasse *Person* erzeugt und über eine Variable *person* referenziert.

- ▶ Welche der folgenden Code-Zeilen kompilieren erfolgreich und was wird jeweils ausgegeben?
- ▶ Welche Fehlermeldung wird für den Fall, dass eine Zeile nicht ausführbar ist, erzeugt?

```
System.out.println(person);
System.out.println(person.toString());
System.out.println(person + 1);
System.out.println(person.toString() + 1);
```

■ **Aufgabe 42.** (Klasse: SongApp) Erweitern Sie die Klasse *Song* aus Aufgabe 31 um eine *toString()*-Methode, die Interpret bzw. Interpretin und den Titel wie in der nachfolgenden Beispieldausgabe formatiert zurückgibt. Verwenden Sie zur Formatierung der Rückgabe *nicht* die Verkettung über den Plus-Operator. Beispieldausgabe:

```
Playlist:
Heather Nova: Walk this world
James Reyne: Reckless
```

■ **Aufgabe 43.** (Klasse: CountLetterApp) Schreiben Sie ein Programm, das für eine Zeichenkette ausgibt, wie oft diese den Buchstaben *a* enthält. Beispieldausgabe:

```
"Es war eine Mutter, die hatte 4 Kinder:
den Frühling , den Sommer, den Herbst und Klaus-Günter ."

The text contains 'a' 3 times .
```

Texteingabe

Die statische Methode *showInputDialog()* der Klasse *JOptionPane* ermöglicht die Eingabe von Zeichenketten über die Tastatur.

■ **Aufgabe 44.** (Klasse: StringAnalysisApp) Erstellen Sie ein Programm, das zu einer Zeichenkette die Anzahl der enthaltenen Vokale sowie die Anzahl der enthaltenen Konsonanten ausgibt. Beispieldausgabe:

```
Text      : Erstellen Sie ein Programm .
Vocals   : 9
Consonants : 14
```

■ **Aufgabe 45.** (Klasse: StringAnalysisApp) Wandeln Sie Ihre Lösung zu Aufgabe 44 derart ab, dass Sie die Wrapper-Klasse *Character* verwenden, um zu überprüfen, ob es sich bei einem Zeichen um einen Buchstaben handelt.

■ **Aufgabe 46.** (Klasse: ParseKeyValueApp) Erstellen Sie ein Programm, das von der Tastatur Daten im Format *key=value* einliest (z. B. "university =HAW Hamburg"). Geben Sie die Eingaben für *key* und *value* jeweils in einer eigenen Zeile auf Konsole aus. Beispieldausgabe:

```
Input : Type=Fender Telecaster
Key   : Type
Value : Fender Telecaster
```

■ **Aufgabe 47.** (Klasse: SubstringApp) Aus einer Zeichenkette soll derjenige Teilstring extrahiert werden, der mit einem bestimmten Wort anfängt und mit einem bestimmten Wort endet. Implementieren Sie diese Funktionalität in einer Methode `getSubstring()`, welcher die Zeichenkette sowie das Start- und Endwort übergeben werden. Überprüfen Sie die Methode anhand eines Programms. Beispielausgabe:

```
Text      : Und wer baggert da so spät noch am Baggerloch?
Start    : wer
End      : da
Extracted : wer baggert da
```

4.2 Felder (Arrays)

■ **Aufgabe 48.** (Klasse: InitArray) Erstellen Sie ein Programm, das ein `int`-Feld mit absteigenden Werten 50, 49, 48, ..., 1 erzeugt und die Werte in Zeilen je zehn Zahlen auf Konsole ausgibt. Beispielausgabe:

```
50 49 48 47 46 45 44 43 42 41
40 39 38 37 36 35 34 33 32 31
30 29 28 27 26 25 24 23 22 21
20 19 18 17 16 15 14 13 12 11
10  9  8  7  6  5  4  3  2  1
```

■ **Aufgabe 49.** (Klasse: CompareArrays) Erstellen Sie eine Methode `equalContents()` zum Vergleich zweier `int`-Felder. Diese gibt `true` dann und nur dann zurück, wenn die übergebenen Felder die gleiche Größe besitzen und alle korrespondierenden Positionen identische Zahlenwerte beinhalten. Überprüfen Sie die Methode anhand ausgewählter Felder. Beispielausgabe:

```
Compare array 1 2 3 4 5 6 to:
a) Exact copy : 1 2 3 4 5 6 -> true
b) Other values: 1 2 3 4 6 7 -> false
c) Other size  : 1 2 3           -> false
```

Aufbau von Zeichenketten

String-Objekte sind unveränderbar (*immutable*), was nachteilig ist, falls Zeichenketten nach und nach erweitert werden sollen. Es bietet sich daher an, Zeichenketten in einem Objekt der Klasse *StringBuilder* aufzubauen und über `toString()` zurückzugeben.

■ **Aufgabe 50.** (Klasse: InvertArray) Erstellen Sie ein Programm, das die Reihenfolge der Werte eines Feldes umkehrt, ohne jedoch ein zweites Feld zu erzeugen. Beispielausgabe:

```
Source   : 1  2  5  9 11 20 26
Reordered: 26 20 11  9  5  2  1
```

■ **Aufgabe 51.** (Klasse: RandomArrayApp1) Erstellen Sie eine Klasse *RandomArray*, deren Objekte bei der Instanzierung ein `int`-Feld erzeugen und mit Zufallszahlen initialisieren. Es gelte zudem:

- ▶ Alle Elemente werden unabhängig mit einer zufälligen Zahl $n \in \mathbb{N}$, $n \leq 25$, initialisiert.
- ▶ Die Größe des Arrays wird dem Konstruktor übergeben, darf aber nicht in einer Instanzvariablen gespeichert werden.

- Implementieren Sie `toString()` derart, dass sie die Zahlen in einer Zeile formatiert zurückgibt.
- Überprüfen Sie die Klasse anhand eines ausführbaren Programms.

Beispielausgabe:

```
Random array: 9, 20, 4, 17, 22, 11, 11, 14, 7, 10
```

■ **Aufgabe 52.** (Klasse: RandomArrayApp2) Erweitern Sie die Klasse `RandomArray` aus Aufgabe 51 um folgende Methoden:

- Die Methode `contains()` gibt dann und nur dann `true` zurück, falls der als Argument übergebene Wert im Array enthalten ist.
- Die Methode `sort()` sortiert die Elemente im Array in aufsteigender Reihenfolge. Versuchen Sie, die Methode ohne Hilfe (z. B. Skript oder Internet) zu implementieren.

Beispielausgabe:

```
Random array: 2, 4, 10, 10, 12, 16, 17, 18, 23, 25
Contains 1: false
Contains 2: true
Contains 3: false
Contains 4: true
Contains 5: false
```

■ **Aufgabe 53.** (Klasse: LotteryApp1) Lassen Sie uns nun `RandomArray` aus Aufgabe 52 verwenden, um das klassische Lotto „6-aus-49“ umzusetzen. Definieren Sie eine Klasse `Lottery` zum Ziehen von m Zahlen (hier: $m = 6$) aus einer Menge $\mathbb{L} = [1, n] \subset \mathbb{N}$ (hier: $n = 49$). Diese soll folgende Anforderungen erfüllen:

- Dem Konstruktor werden die Anzahl m zu ziehender Zahlen sowie die Anzahl n vorhandener Zahlen übergeben.
- Die Methode `drawNumbers()` führt die Ziehung der Zahlen durch, indem ein internes Array mit zufällig bestimmten Zahlen aus \mathbb{L} gefüllt wird.
- Zahlen können nicht mehrfach gezogen werden, d. h. alle gezogenen Werte unterscheiden sich.
- Die gezogenen Zahlen werden nach der Ziehung in aufsteigender Reihenfolge sortiert.
- Durch erneuten Aufruf von `drawNumbers()` wird eine neue Ziehung durchgeführt.
- Die `toString()`-Methode gibt die gezogenen Zahlen in einer Zeile formatiert zurück.

Überprüfen Sie die Klasse anhand eines Programms. Beispielausgabe:

```
1. drawing: 8, 14, 19, 24, 26, 42
2. drawing: 7, 8, 28, 30, 38, 45
3. drawing: 6, 26, 36, 39, 43, 49
4. drawing: 2, 4, 10, 24, 25, 35
5. drawing: 3, 6, 17, 22, 45, 49
```

■ **Aufgabe 54.** (Klasse: LotteryApp2) Nun spielen wir Lotto – und zwar *sehr, sehr* oft!

- Ändern Sie die Klasse `Lottery` aus Aufgabe 53 derart, dass die Methode `drawNumbers()` eine Referenz auf das interne Feld der gezogenen Zahlen zurückgibt.
- Erstellen Sie ein Programm, das über die Klasse `Lottery` zunächst einen Tippschein mit sechs zufälligen Zahlen erzeugt, anschließend viele Male Lottozahlen zieht und auf Konsole ausgibt, bei wie vielen Ziehungen der Tippschein „6 Richtige“ enthalten hat. Verwenden Sie die Klasse `CompareArrays` aus Aufgabe 49 zum Vergleich der Zahlen.

Beispielausgabe:

```
Number games : 50,000,000 (50 millions)
You win      :      2
You lose     : 49,999,998
```

■ **Aufgabe 55.** (Klasse: DiceDistribution) Na gut, Lottospielen scheint sich nach dem Ergebnis aus Aufgabe 54 nicht zu lohnen. Also spielen wir lieber ohne finanziellen Einsatz mit Freunden Würfelspiele. In Aufgabe 19 auf Seite 14 haben wir untersucht, ob die erzeugten Zufallszahlen von 1 bis 6 in etwa mit gleicher Wahrscheinlichkeit auftreten. Verbessern Sie Ihre Lösung durch die Verwendung eines Feldes zum Zählen der Häufigkeiten. Beispielausgabe:

```
Distribution throwing dice 100 million times:
Roll 1: 16,669,450 (16.67 %)
Roll 2: 16,667,592 (16.67 %)
Roll 3: 16,665,813 (16.67 %)
Roll 4: 16,665,337 (16.67 %)
Roll 5: 16,667,321 (16.67 %)
Roll 6: 16,664,487 (16.67 %)
```

4.3 Mehrdimensionale Felder (Arrays)

■ **Aufgabe 56.** (Klasse: –) Wandeln Sie nachfolgenden Quelltext derart um, dass er aus nur einer Anweisungszeile besteht, aber dieselbe Funktionalität erfüllt.

```
int [][] x;
x = new int [2][];
for (int i = 0; i < 2; i++) {
    x[i] = new int [3];
}
```

■ **Aufgabe 57.** (Klasse: DiceDistribution2D) Simulieren Sie ausgehend von Aufgabe 55 Würfe mit zwei Würfeln. Schreiben Sie ein Programm, das die Anzahl an geworfenen Kombinationen in einer Matrix zählt und die relativen Häufigkeiten auf Konsole ausgibt. Beispielausgabe:

```
Frequency in % when rolling two dices 1,000,000 times:
```

	1	2	3	4	5	6
1	2.79	2.77	2.79	2.78	2.77	2.78
2	2.76	2.78	2.77	2.75	2.81	2.78
3	2.76	2.78	2.77	2.78	2.77	2.80
4	2.76	2.76	2.79	2.81	2.77	2.78
5	2.77	2.78	2.78	2.81	2.77	2.74
6	2.79	2.78	2.77	2.80	2.78	2.75

■ **Aufgabe 58.** (Klasse: SegmentDisplayApp) Viele technische Geräte wie beispielsweise die mobile Spielstandsanzeige in Abbildung 2.2 auf Seite 13 enthalten zur Darstellung von Ziffern 7-Segment-Anzeigen. Die Segmente werden üblicherweise mit Buchstaben *a* bis *g* bezeichnet und durch jeweils eigene Steuerleitungen ein- bzw. ausgeschaltet (Abb. 4.1 und 4.2). Definieren Sie eine Klasse *SegmentDisplay* für 7-Segment-Anzeigen:

- ▶ Die Steuerdaten aus Abbildung 4.1 sind als Matrix mit konstanten Werten ablegt.
- ▶ Die Klassenmethode *digitToSignals()* gibt für eine übergebene Ziffer ein Feld mit den Werten *a* bis *g* zurück. Hierfür wird kein neues Feld erzeugt, sondern eine Referenz auf die entsprechende Zeile der Matrix zurückgegeben.
- ▶ Die Klassenmethode *print()* erhält ein Feld mit Werten *a* bis *g* und simuliert die 7-Segment-Anzeige durch entsprechende Konsolenausgaben.

Ziffer	a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1

Abbildung 4.1: Steuersignale einer 7-Segment-Anzeige

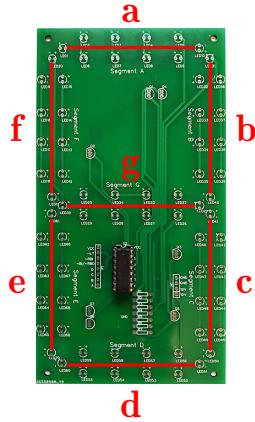


Abbildung 4.2: Platine der mobilen Anzeige in Abb. 2.2 auf Seite 13

Erstellen Sie ein Programm, in dem Anwender bzw. Anwenderinnen zunächst über die Klassenmethode `showInputDialog()` der Klasse `JOptionPane` eine Ziffer als Zeichenkette eingeben. Die Eingabe wird in eine Ganzzahl gewandelt und sowohl die zugehörigen Steuersignale als auch die simulierte Anzeige auf Konsole ausgegeben. Beispieldausgabe:

```
Please enter a digit: 6
Control for 7 segment display:

a | b | c | d | e | f | g
---+---+---+---+---+---+---
1 | 0 | 1 | 1 | 1 | 1 | 1

Display:
  _____
  |
  |_____|_
  |    |
```

■ **Aufgabe 59.** (Klasse: MatrixVector) In der Mathematik ist die Multiplikation einer Matrix $\mathbf{A} \in \mathbb{R}^{3 \times 3}$ mit einem Vektor $\mathbf{x} \in \mathbb{R}^3$ wie folgt definiert:

$$\mathbf{Ax} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} x_1 a_{11} + x_2 a_{12} + x_3 a_{13} \\ x_1 a_{21} + x_2 a_{22} + x_3 a_{23} \\ x_1 a_{31} + x_2 a_{32} + x_3 a_{33} \end{pmatrix} \quad (4.1)$$

Schreiben Sie ein Programm, das eine Matrix-Vektor-Multiplikation ausführt und das Ergebnis auf Konsole ausgibt. Die Koeffizienten von \mathbf{A} und \mathbf{x} sind vom Datentyp `int` und dürfen frei gewählt werden. Beispieldausgabe:

```
y = Ax = |1 2 0| * |2| = |4|
          |0 2 1|   |1|   |5|
          |3 0 1|   |3|   |9|
```

■ **Aufgabe 60.** (Klasse: RotateVector) In Aufgabe 59 haben wir die Multiplikation eines Vektors $\mathbf{x} \in \mathbb{R}^3$ mit einer Matrix $\mathbf{A} \in \mathbb{R}^{3 \times 3}$ betrachtet. In der Ebene \mathbb{R}^2 beschreibt folgende Multiplikation eine Drehung des Vektors \mathbf{x} um den Winkel α :

$$\mathbf{Ax} = \begin{pmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} x_1 \cos \alpha - x_2 \sin \alpha \\ x_1 \sin \alpha + x_2 \cos \alpha \end{pmatrix} \quad (4.2)$$

Schreiben Sie eine Methode `printRotatedVector()`, die einen Vektor und einen Winkel in Grad übergeben bekommt und den rotierten Vektor auf Konsole ausgibt. Verwenden Sie für die trigonometrischen Funktionen die Klasse `Math` und beachten Sie, dass diese Winkel in Bogenmaß erwarten. Beispielausgabe:

```
Input vector : (2.0 3.0)^T
Rotate by     : 90.0 degree
Rotated vector: (-3.0 2.0)^T
```

■ **Aufgabe 61.** (Klasse: MatrixApp1) Definieren Sie eine Klasse `Matrix` zur Repräsentation mathematischer Matrizen $\mathbf{A} \in \mathbb{R}^{M \times N}$:

- ▶ Die Koeffizienten sind im Sinne einer übersichtlicheren Konsolenausgabe vom Datentyp `int`.
- ▶ Die Klasse verfügt über einen Konstruktor, dem die Koeffizienten als zweidimensionales Array übergeben werden.
- ▶ Die Methode `sum()` gibt die Summe aller Koeffizienten zurück.
- ▶ Die Methode `print()` gibt die Koeffizienten in einem zweidimensionalen Raster auf Konsole aus.

Erstellen Sie zudem ein Programm, das die Methoden für folgende Matrix beispielhaft anwendet:

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 1 \\ 1 & 3 & -4 \\ 2 & 4 & -2 \end{pmatrix}$$

Beispielausgabe:

```
Matrix A with coefficient sum = 6:
 2 -1  1
 1  3 -4
 2  4 -2
```

■ **Aufgabe 62.** (Klasse: MatrixApp2) Erweitern Sie die Klasse `Matrix` aus Aufgabe 61 um die Matrizenmultiplikation. Ergänzen Sie hierfür eine Klassenmethode `multiply()`, der zwei Matrizen \mathbf{A} und \mathbf{B} als Objekte vom Typ `Matrix` übergeben werden. Die Methode gibt ein intern erzeugtes weiteres Objekt mit dem Produkt $\mathbf{C} = \mathbf{AB}$ zurück. Überprüfen Sie die Methode mittels folgender Multiplikationen:

- a) Einheitsmatrix \mathbf{E} :

$$\mathbf{AE} = \mathbf{EA} = \mathbf{A}$$

- b)

$$\mathbf{AB} = \begin{pmatrix} 2 & -1 & 1 \\ 1 & 3 & -4 \\ 2 & 4 & -2 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 4 \\ 2 & 3 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 6 & 3 \\ -1 & -7 & 11 \\ 6 & 2 & 20 \end{pmatrix}$$

■ **Aufgabe 63.** (Klasse: RowReferences) Erstellen Sie in einem Programm ein `int`-Array der Dimension 3×3 , bei dem die erste und die dritte Zeile nicht nur die gleichen Werte beinhalten, sondern ein und dasselbe Objekt sind. Das bedeutet, dass Änderungen in der ersten Zeile automatisch auch in der dritten Zeile sichtbar werden und umgekehrt. Beispielausgabe:

```
0 1 1
2 4 2
0 1 1
```

```
Assign a[0][1] = 3 and a[2][0] = 5:
```

```
5 3 1
2 4 2
5 3 1
```

4.4 Listen

■ **Aufgabe 64.** (Klasse: BookStackApp) Setzen Sie einen virtuellen Bücherstapel um, bei dem Bücher jeweils nur oben hinzugefügt und wieder entfernt werden können:

- ▶ Erweitern Sie die Klasse *Book* aus Aufgabe 29 auf Seite 18 um einen Konstruktor, dem Vor- und Nachname des Autors bzw. der Autorin sowie der Titel übergeben werden, und überschreiben Sie *toString()* passend zur unten stehenden Beispielausgabe.
- ▶ Definieren Sie eine Klasse *BookStack*, die einen Bücherstapel repräsentiert. Es existieren Methoden *push()* und *pop()*, mit denen ein Buch vom Datentyp *Book* auf den Stapel gelegt bzw. das oberste Buch vom Stapel entfernt werden kann. Die *toString()*-Methode gibt die auf dem Bücherstapel enthaltenen Bücher derart formatiert zurück, dass das neueste Buch „oben liegt“.

Beispielausgabe:

```
Book stack:
Torsten Sträter: Es ist nie zu spät, unpünktlich zu sein
James Hansen: First Man (The Life of Neil Armstrong)
Bruce Springsteen: Born to run

Removing -> Sträter
Removing -> Hansen
Book stack:
Bruce Springsteen: Born to run

Removing -> Springsteen
Removing -> null
```

■ **Aufgabe 65.** (Klasse: PersonQueueApp) Erstellen Sie eine Klasse *PersonQueue* zur Repräsentation einer Warteschlange:

- ▶ Es existieren Methoden *enter()* und *leave()*, mit denen sich eine Person vom Datentyp *Person* gemäß Aufgabe 41 auf Seite 26 am Ende anstellt bzw. die vorderste Person die Warteschlange verlässt.
- ▶ Die *toString()*-Methode gibt die wartenden Personen in der Reihenfolge, in der sie in der Schlange stehen, zurück.

Beispielausgabe:

```
Queue:
Horst-Gudrun Warteschon

Queue:
Horst-Gudrun Warteschon
Marc Hensel

Queue:
Horst-Gudrun Warteschon
Marc Hensel
Janick-Nick Hintermann

Leave -> Horst-Gudrun Warteschon
Leave -> Marc Hensel
Queue:
Janick-Nick Hintermann

Leave -> Janick-Nick Hintermann
Leave -> null
```

■ **Aufgabe 66.** (Klasse: LotteryApp3) In Aufgabe 53 auf Seite 28 war das klassische Lotto „6-aus-49“ umzusetzen. Hierbei war unter anderem zu berücksichtigen, dass gezogene Zahlen nicht doppelt auftreten dürfen. Umgehen Sie dieses Problem durch eine wie nachfolgend beschriebene Klasse *Lottery* zur Ziehung der Lotto-Zahlen:

- Die Klasse besitzt zwei Listen. Die erste repräsentiert die „Lostrommel“ und enthält daher alle Zahlen, die noch gezogen werden können (also initial 1 bis 49). Die zweite Liste ist initial leer und enthält alle gezogenen Zahlen.
- Für jede gezogene Zahl wird ein zufälliges Element aus der ersten Liste entfernt und aufsteigend sortiert in die Liste der gezogenen Zahlen eingefügt.

Beispielausgabe:

```
1. drawing: 8, 10, 20, 24, 25, 33
2. drawing: 12, 19, 29, 39, 40, 43
3. drawing: 4, 5, 15, 21, 43, 48
4. drawing: 16, 19, 27, 29, 39, 49
5. drawing: 6, 13, 27, 41, 44, 46
```

■ **Aufgabe 67.** (Klasse: MathNewtonApp) In Aufgabe 37 auf Seite 22 haben wir das Newtonverfahren zur Approximation von Nullstellen einer mathematischen Funktion kennengelernt. Modifizieren Sie Ihre Lösung derart, dass die Methode *newton()* der Klasse *ZeroCrossing* eine Liste aller Iterationen x_n zurückgibt. Beispielausgabe:

```
Zero-crossing for f(x) = -1 * x^2 +0 * x +4:
Approximated x1 : 1.00000000000000
Approximated x2 : 2.50000000000000
Approximated x3 : 2.05000000000000
Approximated x4 : 2.000609756098
Approximated x5 : 2.000000092922
Approximated x6 : 2.00000000000000

Approximated x0 : 2.00000000000000
Approximated f(x0): -8.882e-15
```

■ **Aufgabe 68.** (Klasse: MathBisectionApp) Modifizieren Sie analog zur Aufgabe 67 die Umsetzung des Bisektionsverfahrens aus Aufgabe 36 auf Seite 21 derart, dass die Intervallgrenzen aller Bisektionsschritte zurückgegeben werden. Beispielausgabe:

```
Zero-crossing for f(x) = -1 * x^2 +0 * x +4:
Bisection interval: [1.800 , 2.300]
Bisection interval: [1.800 , 2.050]
Bisection interval: [1.925 , 2.050]
Bisection interval: [1.987 , 2.050]
Bisection interval: [1.987 , 2.019]
Bisection interval: [1.987 , 2.003]
Bisection interval: [1.995 , 2.003]

Approximated x0 : 1.999219
Approximated f(x0): 3.124e-03
```

4.5 Fragen

Frage 42. Nehmen Sie Stellung zu folgender Aussage: „Ein Variable, die ein Feld referenziert, ist ein Zeiger auf das erste Element des Feldes.“ Ist die Aussage richtig oder falsch?

Frage 43. Gegeben sei ein Feld vom Typ *Point[]*. Nehmen Sie Stellung zu folgender Aussage: „Die Komponenten *x* und *y* jedes Punktes sind direkt im Feld gespeichert.“ Ist die Aussage richtig oder falsch?

Praktikumsaufgaben (Casino, Aufgabenteil „Mehrarmige Banditen“)

Bearbeiten Sie nun Praktikum 1 auf Seite 85.

- ▶ Stellen Sie sicher, dass Sie das Praktikum lösen können, bevor Sie fortfahren.
- ▶ Es ist sinnvoll auch die weiteren Versuche zum Praktikum 1 zu bearbeiten.

Kapitel 5

Anwendungsbeispiel: Bildverarbeitung

In Kapitel 4.3 haben wir Aufgaben zu 2D-Feldern betrachtet. Allgemein eröffnen sich einem durch den Umgang mit mehrdimensionalen Feldern spannende und motivierende Anwendungsgebiete. Insbesondere entsprechen die Bildpunkte (*Pixel*) digitaler Bilder und Videos lediglich Matrizen und damit 2D-Feldern.

Dieses Kapitel gibt eine erste Einführung in die Bildverarbeitung. Wir werden uns dem Thema in nachfolgenden Schritten nähern und es in späteren Kapiteln wieder aufgreifen:

1. Lesen und Speichern von Bilddateien
2. Kodierung von 8-Bit Graustufenbildern
3. Änderung von Bildpunkten (Negativbild)
4. Änderung von Bildpunkten in Abhängigkeit benachbarter Werte (Kantendetektion)

5.1 Bilddaten

Die zur Veranschaulichung verwendeten und in Abbildung 5.1 dargestellten Fotos stelle ich Ihnen für die Bearbeitung der Aufgaben mit dem Vorlesungsmaterial zur Verfügung. Sie können stattdessen aber selbstverständlich eigene Fotos verwenden.

Beachten Sie, dass wir zunächst nur den Umgang mit Graustufenbildern diskutieren. Konvertieren Sie Ihre Bilder daher gegebenenfalls mit einem entsprechenden Bildbearbeitungsprogramm wie *GIMP*¹. In der zum Zeitpunkt des Schreibens verwendeten Version 2.10 wählen Sie den Menüpunkt *Bild / Modus / Graustufen* und speichern das Graustufenbild über *Datei / Exportieren nach*.

5.2 Bilddateien lesen und schreiben

Eine Voraussetzung zur Verarbeitung von Fotos ist, dass wir diese zuvor aus einer Bilddatei in das Programm einlesen. Zudem brauchen wir eine Möglichkeit, bearbeitete Bilder zu betrachten. Da wir grafische Benutzeroberflächen – beispielsweise zur Darstellung von Bildern in einem Programmfenster – erst in einem späteren Kapitel besprechen, speichern wir die Ergebnisse vorerst in Bilddateien auf Festplatte.

Der nachfolgende Quelltext liest die Datei *DocksGray.jpg* von der Festplatte und speichert das Foto unverändert unter einem anderen Dateinamen und Format (*Target.png*). Sie können die entsprechenden Code-Ausschnitte im Weiteren mit entsprechenden Anpassungen übernehmen.

```
1 import java.awt.image.BufferedImage;
2 import java.io.File;
3 import java.io.IOException;
4 import javax.imageio.ImageIO;
5
6 public class ReadWriteImage {
7
8     public static void main(String[] args) throws IOException {
9         // Read image from file
10        File sourceFile = new File("DocksGray.jpg");
11        BufferedImage image = ImageIO.read(sourceFile);
12
13        // Write image to file
14        File destFile = new File("Target.png");
```

¹<https://www.gimp.org/> (Besucht am 08.03.2021)



(a) „Docks“ (24-Bit Farbe)



(b) „Kölner Dom“ (24-Bit Farbe)



(c) „Docks“ (8-Bit Graustufen)



(d) „Kölner Dom“ (8-Bit Graustufen)

Abbildung 5.1: Ausgangsbilder zur Bildverarbeitung

```
15     ImageIO.write(image, "png", destFile);
16 }
17 }
```

- *Klassen:* Wir verwenden die Klasse *BufferedImage* zur Repräsentation von Bildern. Zum Lesen und Schreiben von bzw. in Dateien greifen wir auf statische Methoden der Klasse *ImageIO* zurück.
- *Datei lesen:* Die Methode *read()* liest das Bild aus der Datei *DocksGray.jpg* in ein Objekt vom Typ *BufferedImage* (Zeile 11). Hierfür übergeben wir *read()* ein Objekt der Klasse *File*, das zuvor mit dem Dateinamen instantiiert wurde (Zeile 10).
- *Datei schreiben:* Die Methode *write()* schreibt das Bild in die Datei *Target.png* (Zeile 15). Zusätzlich zum Bild werden das Dateiformat als *String* sowie ein mit dem Namen der Zielfile instantiiertes (Zeile 14) *File*-Objekt übergeben.
- *Exceptions:* Bei Dateizugriffen kann alles Mögliche schiefgehen (z. B. fehlende Schreibrechte). Daher muss der Quelltext definieren, ob und wie mit derartigen *Ausnahmen* bzw. *Exceptions* umzugehen ist. Wir betrachten die entsprechenden Mechanismen in einem späteren Kapitel. Für den Moment genügt es, allen Methodenköpfen, in denen direkt oder indirekt auf Dateien zugegriffen wird, **throws IOException** anzufügen (z. B. Zeile 8).

■ **Aufgabe 69.** (Klasse: ReadWriteImage) Nun ist es an der Zeit, dass Sie selber die ersten Schritte in die Welt der Bildverarbeitung unternehmen:

- Ziehen Sie eine Bilddatei (z. B. *CologneGray.jpg* oder *DocksGray.jpg*) aus dem Explorer in Ihr Java-Projekt innerhalb des IntelliJ IDEA-Fensters. Die Datei erscheint im *Package Explorer* parallel zu den Paketen des Projektes und es kann aus den Paketen heraus über den Dateinamen auf sie zugegriffen werden.
- Erstellen Sie ein Programm, das eine zuvor im zugehörigen Java-Projekt (*nicht* dem Paket!) abgelegte Bilddatei einliest und das entsprechende Foto unter einem anderen Dateinamen speichert.
- Öffnen Sie die erstellte Datei in einem Bildbetrachter (z. B. durch Doppelklick auf den Dateinamen im Explorer).

5.3 Kodierung

Bislang haben wir ein Bild aus einer Datei gelesen und anschließend in einer anderen Datei gespeichert. Nun wollen wir uns anschauen, wie wir auf einzelne Pixel zugreifen und wie die Zahlenwerte zu interpretieren sind. Folgender Quelltext-Ausschnitt überprüft zunächst, ob es sich tatsächlich um ein 8-Bit Graustufenbild handelt. Sofern dies der Fall ist, wird die Variable *pixels* als Referenz auf die Bildpunkte angelegt. Wie Sie sehen, sind diese nicht etwa in einer 2D-Matrix, sondern in einem 1D-Feld vom Typ *byte* gespeichert.

```
if (image.getType() == BufferedImage.TYPE_BYTE_GRAY) {
    // Get pixel data (1D array of type byte [-128, 127])
    byte[] pixels = ((DataBufferByte)image.getRaster().getDataBuffer()).getData();
    // Do something with the pixels ...
}
```

■ **Aufgabe 70.** (Klasse: PixelValues) Bei 8-Bit Graustufenbildern besitzen Pixel Zahlenwerte von 0 (schwarz) bis 255 (weiß). Allerdings ist der Datentyp *byte* in Java vorzeichenbehaftet und besitzt damit Werte von -128 bis 127. Untersuchen Sie, wie die Werte interpretiert werden:

- Lesen Sie ein beliebiges Graustufenbild (z. B. *CologneGray.jpg*) ein.
- Greifen Sie auf das *byte*-Feld der Pixelwerte zu, setzen Sie alle Werte auf 0 und speichern Sie das Bild unter anderem Dateinamen.
- Wiederholen Sie b) für die Werte -128, -1 und 127 und betrachten Sie die erzeugten Bilder. Welche Rückschlüsse ziehen Sie auf die Kodierung der Bilddaten?

Wie sich in Abbildung 5.2 zeigt, wird die Helligkeit wie erwartet mit Werten von 0 (schwarz) bis 255 (weiß) kodiert. Durch das Vorzeichen des Datentyps *byte* werden die Zahlen von 128 bis 255 jedoch mittels Wrap-around in den Bereich -128 bis -1 verschoben (Abb. 5.3).

■ **Aufgabe 71.** (Klasse: ImagingApp) Definieren Sie eine Klasse *Imaging* als Basis für die Verarbeitung von 8-Bit Graustufenbildern (Abb. 5.4). Dies Klasse beinhaltet u. a. eine 2D-Matrix mit Pixelwerten in [0, 255] als Kopie der im 1D-Feld des *BufferedImage*-Objekts enthaltenen Pixelwerte in [-128, 127]:

- ▶ Der Konstruktor erhält als Argument den Pfad der zu öffnenden Datei, liest das Bild in die Instanzvariable vom Datentyp *BufferedImage* und weist den Instanzvariablen für die Breite und Höhe des Bildes die entsprechenden Werte zu.

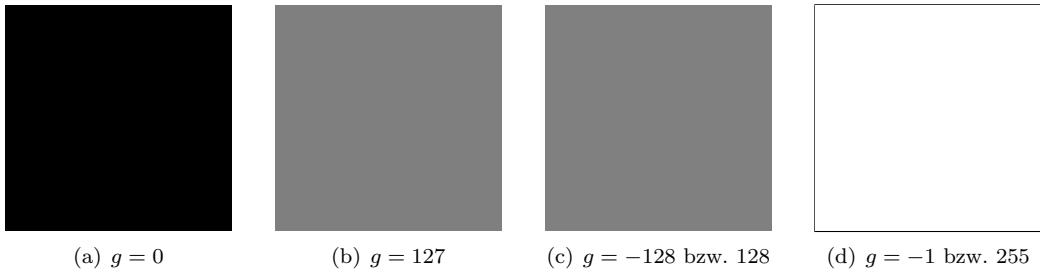


Abbildung 5.2: Kodierung der Pixelwerte g in 8-Bit Graustufenbildern (Aufgabe 70)

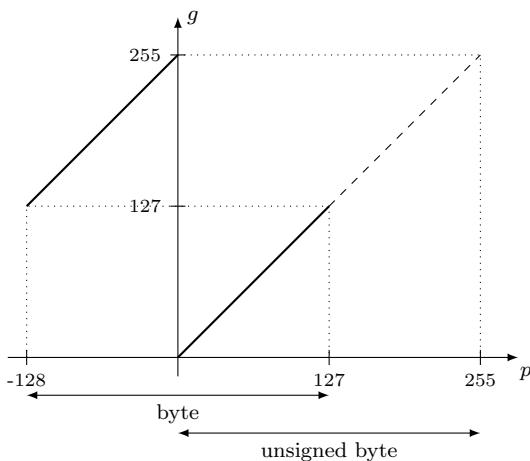


Abbildung 5.3: Zuordnung Pixelwerte p auf Grauwerte $g = 0$ (schwarz) bis $g = 255$ (weiß)

- Zusätzlich erzeugt der Konstruktor eine *short*-Matrix in der Größe des Bildes und weist den Pixeln die zugehörigen Grauwerte $\in [0, 255]$ zu.
- Die Methode *setPixels()* weist allen Pixelwerten der Matrix *pixels* den übergebenen Grauwert $\in [0, 255]$ zu.
- Die Methode *writeFilePNG()* kopiert die Pixelwerte der Matrix *pixels* in das *byte*-Feld des *BufferedImage*-Objekts und speichert das Bild unter dem übergebenen Dateinamen.
- Denken Sie daran, alle Methoden, die Dateioperationen ausführen (also den Konstruktor sowie *writeFilePNG()*), um **throws IOException** zu ergänzen. Gleiches gilt für Methoden, die diese aufrufen (z. B. *main()*).

Imaging
image: BufferedImage
width: int
height: int
pixels: short[][]
Imaging(imagePath: String)
setPixels(pixelValue: int): void
writeFilePNG(filePath: String): void

Abbildung 5.4: Klassendiagramm der Klasse *Imaging* (Aufgabe 71)

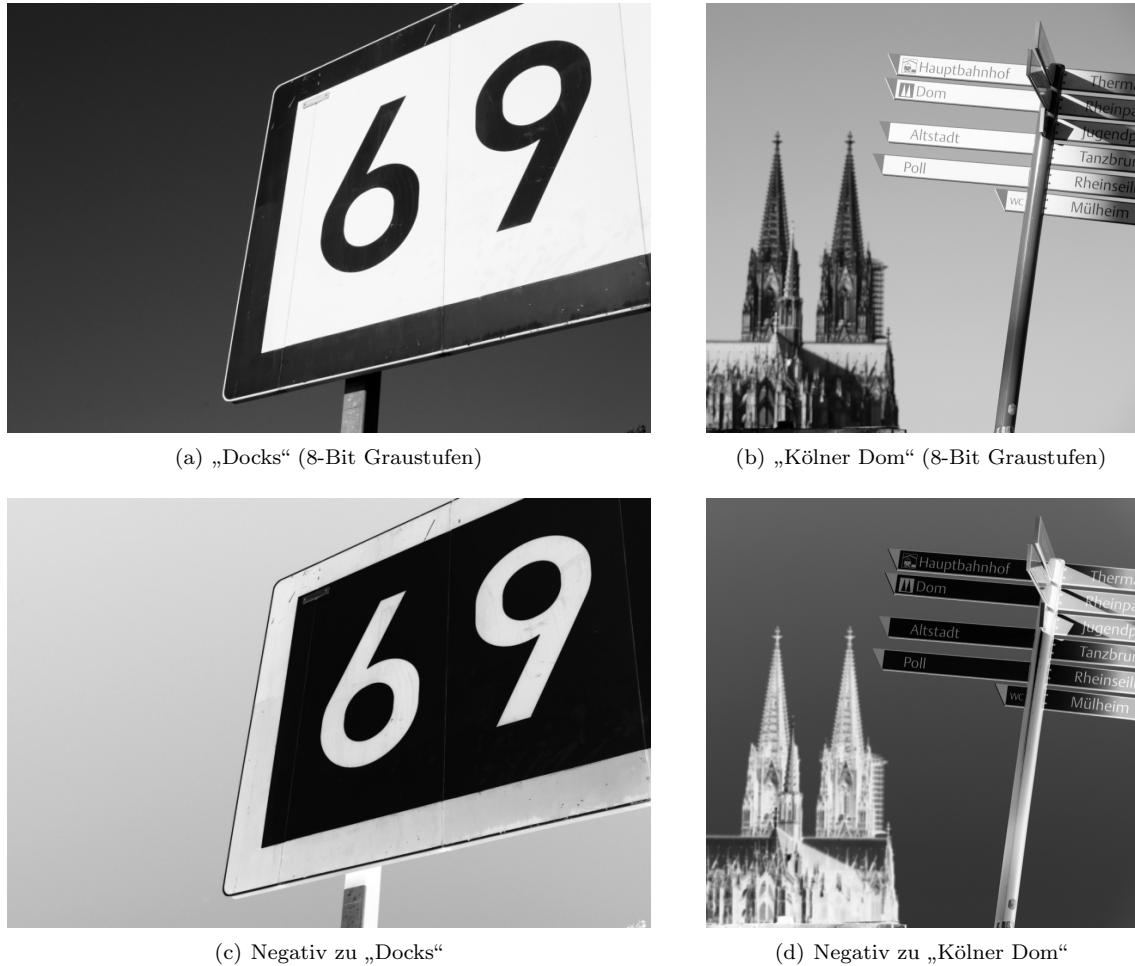


Abbildung 5.5: Negative (Aufgabe 72)

Schreiben Sie ein Programm, das ein *Imaging*-Objekt erzeugt, den Pixeln einen Wert zuweist und das monochrome Bild unter anderem Dateinamen speichert.

5.4 Punktoperationen

Bei sogenannten *Punktoperationen* wird jedem Pixel $g(x, y)$ nur aufgrund seines Grauwertes g ein Wert $f(g)$ zugewiesen. Die Werte benachbarter Bildpunkte sowie der Ort (x, y) werden hierbei nicht berücksichtigt:

$$g \rightarrow f(g) \quad (5.1)$$

Invertieren (Negative) Eine recht bekannte Punktoperation ist das *Invertieren* der Helligkeit. Das entstehende Bild entspricht dem aus der analogen Film-Fotografie bekannten *Negativ* (Abb. 5.5). Durch Invertieren der Helligkeit werden schwarze Pixel zu weißen ($0 \rightarrow 255$), weiße zu schwarzen ($255 \rightarrow 0$) und die Graustufen verlaufen linear von weiß nach schwarz. Es ergibt sich also folgende mathematische Funktion (Abb. 5.6(a)):

$$f(g) = 255 - g \quad (5.2)$$

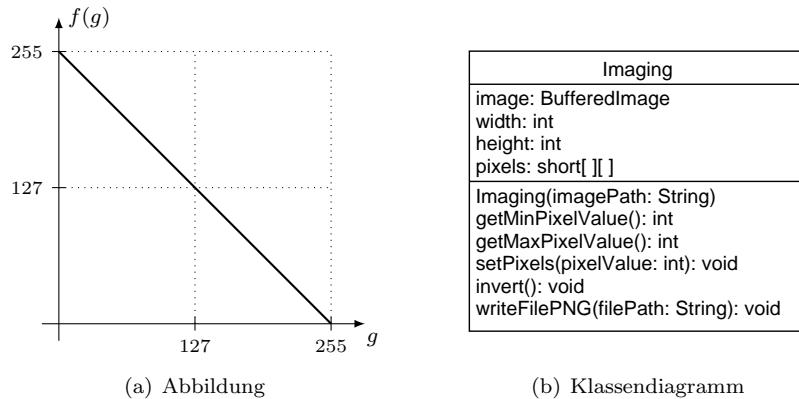


Abbildung 5.6: Invertierten von Bildern (Aufgabe 72)

■ **Aufgabe 72.** (Klasse: *InvertImage*) Erweitern Sie die Klasse *Imaging* aus Aufgabe 71 wie folgt (Abb. 5.6(b)):

- ▶ Die Methoden *getMinPixelValue()* und *getMaxPixelValue()* geben den kleinsten bzw. größten im Bild enthaltenen Pixelwert zurück.
- ▶ Die Methode *invert()* ersetzt jeden Pixelwert g der Pixelmatrix *pixels* durch den invertierten Wert $f(g)$ gemäß Gleichung (5.2).

Erstellen Sie zudem ein Programm, das ein Bild invertiert sowie die Wertebereich der Pixel beider Bilder auf Konsole ausgibt. Beispieldausgabe:

```
File opened : DocksGray.jpg
Input image : [0, 249]
Inverted    : [6, 255]
File written: InvertImage.png
```

5.5 Kantendetektion

Orte, an denen sich die Helligkeit im Bild stark ändert, nehmen wir als „Kanten“ wahr. Es gibt eine Reihe unterschiedlicher Ansätze zur Detektion von Kanten in Bildern. Im Folgenden betrachten wir zunächst die vergleichsweise einfache, mathematisch motivierte *Gradienten*-basierte Kantendetektion. Ein Nachteil dieses Ansatzes ist, dass starkes Rauschen fälschlicher Weise als Kante erkannt wird. Der anschließend vorgestellte *Sobel-Operator* ist diesbezüglich ein klein wenig robuster.

Gradienten Betrachten wir die Grauwerte einer einzelne Zeile als Funktionswerte $g_y(x) = g(x, y)$, so ist die Ableitung $g'_y(x) = \frac{\Delta}{\Delta x} g_y(x)$ nach x bzw. der Gradient $\frac{\partial}{\partial x} g(x, y)$ ein Maß für die Kantenstärke in x -Richtung am entsprechenden Ort im Bild:

- ▶ Eine große Ableitung entspricht einer starken Grauwert-Änderung und daher einer Kante.
- ▶ Ist die Ableitung hingegen klein (oder null), so ändern sich die Grauwerte kaum (bzw. nicht) und wir nehmen keine Kante wahr.

Wir erhalten also ein Bild, dessen Helligkeit ein Maß für die Kantenstärke in horizontaler Richtung ist, indem wir jeden Pixelwert $g(x, y)$ durch den Betrag seines Gradienten in x -Richtung ersetzen:

$$\text{grad}_x(x, y) = \left| \frac{\partial}{\partial x} g(x, y) \right| \quad (5.3)$$

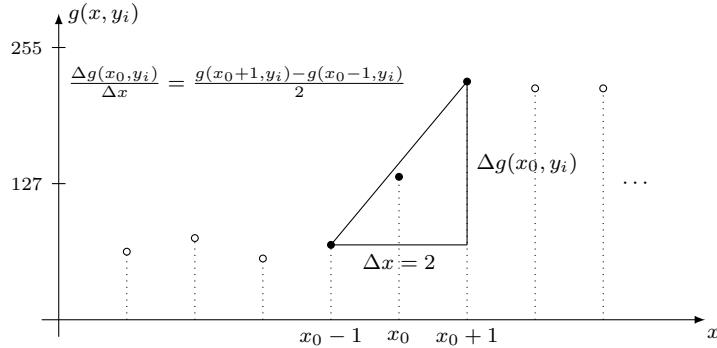


Abbildung 5.7: Kantenstärke entlang einer Zeile $g(x, y_i)$ als Approximation der Steigung

Analog hierzu erhalten wir ein vertikales Kantenbild durch Ersetzen der Pixelwerte $g(x, y)$ durch den Betrag ihrer Gradienten in y -Richtung:

$$\text{grad}_y(x, y) = \left| \frac{\partial}{\partial y} g(x, y) \right| \quad (5.4)$$

Was kompliziert aussieht, wird sich nun sehr stark vereinfachen. Eine Bildmatrix ist nämlich keine stetige und differenzierbare Funktion $\mathbb{R} \times \mathbb{R} \mapsto \mathbb{R}$, sondern besitzt nur diskrete Werte an diskreten Koordinaten $(x, y) \in \mathbb{N}_0^2$. Daher können wir die partiellen Ableitungen nicht bilden. Stattdessen approximieren wir die Steigungen an einem Ort (x, y) durch die Pixelwerte des linken und rechten Nachbarn (horizontale Kante, Abb. 5.7),

$$\frac{\partial}{\partial x} g(x, y) \approx \frac{g(x+1, y) - g(x-1, y)}{\Delta x} = \frac{1}{2} \cdot (g(x+1, y) - g(x-1, y)), \quad (5.5)$$

bzw. des oberen und unteren Nachbarn (vertikale Kante):

$$\frac{\partial}{\partial y} g(x, y) \approx \frac{g(x, y+1) - g(x, y-1)}{\Delta y} = \frac{1}{2} \cdot (g(x, y+1) - g(x, y-1)) \quad (5.6)$$

Zwei Werte von einander abzuziehen und durch zwei zu teilen ist doch machbar! Dies funktioniert lediglich am Bildrand (z. B. dem ersten und letzten Pixel jeder Zeile bei horizontalen Kantenbildern) nicht, da die Randpixel nur einen statt zwei Nachbarn besitzen. Diese Pixel werden daher zu null gesetzt.

Aufgabe 73. (Klasse: Gradients) Erweitern Sie die Klasse *Imaging* aus Aufgabe 72 um Methoden *gradientX()* sowie *gradientY()* zur Erzeugung eines Gradienten-basierten horizontalen bzw. vertikalen Kantenbildes (Abb. 5.8):

- Erzeugen Sie für das Kantenbild eine *short*-Matrix in der Größe der Bildmatrix.
- Laufen Sie durch die Kantenmatrix und weisen Sie die zugehörigen Gradienten der Bildmatrix *pixels* als Kantenwerte zu.
- Weisen Sie der Bildmatrix *pixels* die Kantenmatrix zu.

Erstellen Sie zudem ein Programm, das zu einem Graustufenbild das horizontale und vertikale Kantenbild erzeugt und unter anderem Dateinamen speichert. Abbildungen 5.8(c) und (d) zeigen die Ergebnisse für das Ausgangsbild „Docks“.

Aufgabe 74. (Klasse: Gradients) In Aufgabe 73 haben wir horizontale und vertikale Gradienten-basierte Kantenbilder betrachtet. In dieser Aufgabe sollen Sie diese miteinander verbinden, sodass

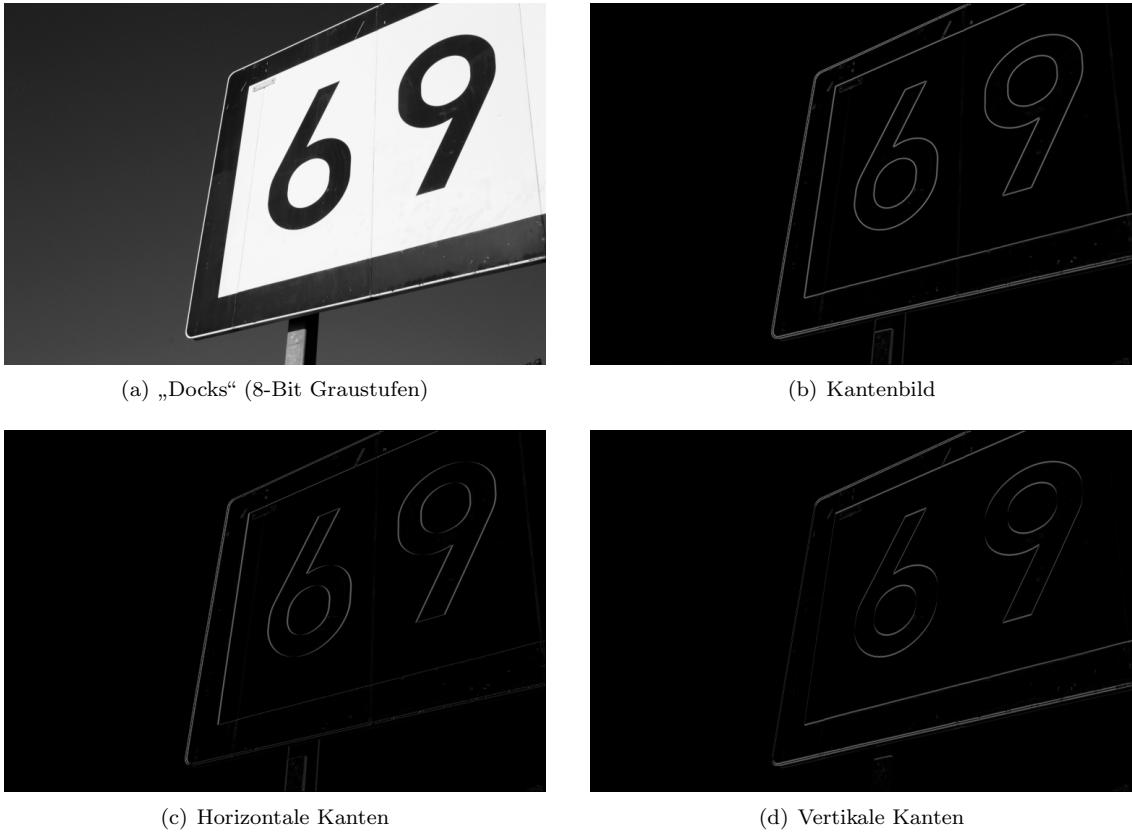


Abbildung 5.8: Gradienten-basierte Kantendetektion (Aufgaben 73 und 74)

ein Kantenbild entsteht, das unabhängig davon ist, in welcher Richtung eine Kante verläuft. Erweitern Sie hierfür die Klasse *Imaging* um eine Methode *gradient()*, die folgende Formel umsetzt:

$$\text{grad}(x, y) = \sqrt{(\text{grad}_x(x, y))^2 + (\text{grad}_y(x, y))^2} \quad (5.7)$$

Das Ergebnis bei Anwendung auf das Ausgangsbild „Docks“ ist in Abbildung 5.8(b) dargestellt.

Sobel-Operator Der sogenannte *Sobel-Operator* ist der Gradienten-basierten Kantendetektion recht ähnlich. Zur Reduktion des Rauschens werden lediglich die für die Approximation der Gradienten verwendeten Nachbarpixel $g(x-1, y)$ und $g(x+1, y)$ (horizontales Kantenbild) bzw. $g(x, y-1)$ und $g(x, y+1)$ (vertikales Kantenbild) durch gewichtete Mittelwerte ersetzt:

$$\bar{g}(x-1, y) = \frac{1}{4} \cdot (g(x-1, y-1) + 2 \cdot g(x-1, y) + g(x-1, y+1)) \quad (5.8)$$

$$\bar{g}(x+1, y) = \frac{1}{4} \cdot (g(x+1, y-1) + 2 \cdot g(x+1, y) + g(x+1, y+1)) \quad (5.9)$$

$$\bar{g}(x, y-1) = \frac{1}{4} \cdot (g(x-1, y-1) + 2 \cdot g(x, y-1) + g(x+1, y-1)) \quad (5.10)$$

$$\bar{g}(x, y+1) = \frac{1}{4} \cdot (g(x-1, y+1) + 2 \cdot g(x, y+1) + g(x+1, y+1)) \quad (5.11)$$

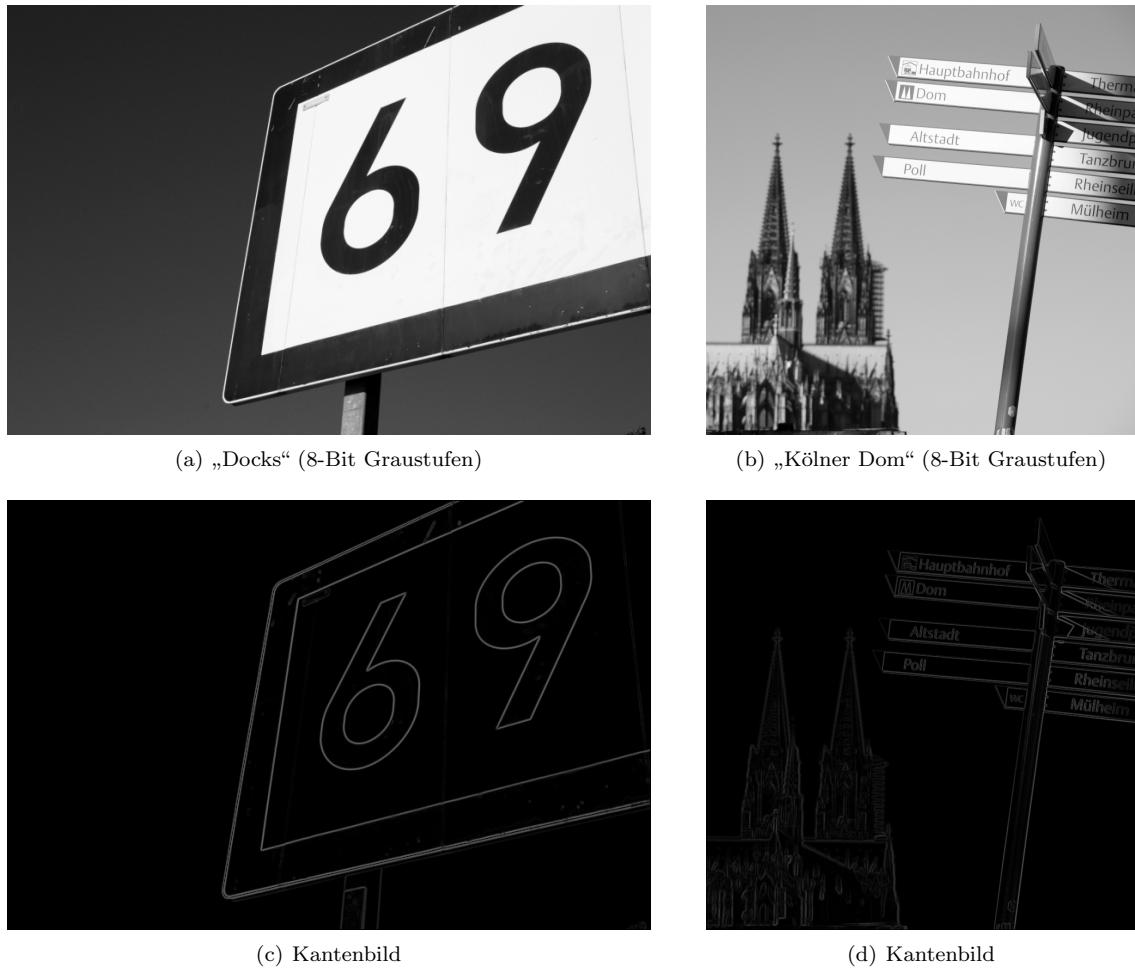


Abbildung 5.9: Kantendetektion mittels Sobel-Operator (Aufgabe 75)

Das horizontale bzw. vertikale Kantenbild ergibt sich hiermit analog zu Gleichung (5.5) bzw. (5.6):

$$\text{sobel}_x(x, y) = \frac{1}{2} \cdot (\bar{g}(x+1, y) - \bar{g}(x-1, y)) \quad (5.12)$$

$$\text{sobel}_y(x, y) = \frac{1}{2} \cdot (\bar{g}(x, y+1) - \bar{g}(x, y-1)) \quad (5.13)$$

Für das richtungsunabhängige Kantenbild gilt analog zu Gleichung (5.7):

$$\text{sobel}(x, y) = \sqrt{(\text{sobel}_x(x, y))^2 + (\text{sobel}_y(x, y))^2} \quad (5.14)$$

Aufgabe 75. (Klasse: Sobel) Erweitern Sie die Klasse *Imaging* um eine Methode *sobel()*, die den Sobel-Operator nach Gleichung (5.14) anwendet. Abbildung 5.9 zeigt die Kanten exemplarisch für die Ausgangsbilder „Docks“ und „Kölner Dom“.

Kapitel 6

Vererbung

In den Aufgaben zu diesem Kapitel sollen Sie eine einfache Bibliothek für Musiktitel und Videos erzeugen. Beachten Sie, dass die Instanzvariablen aller Klassen als *private* deklariert sein müssen.

6.1 Grundlegende Klassen

■ **Aufgabe 76.** (Klasse: MediaApp1) Erstellen Sie zunächst die nachfolgend beschriebenen und in Abbildung 6.1 dargestellten Klassen:

- ▶ Die Klasse *Audio* besitzt Instanzvariablen für den Interpreten bzw. die Interpretin sowie den Titel. Beide Variablen werden über einen Konstruktor gesetzt und können über Getter abgefragt werden. Die *toString()*-Methode wird passend zur Beispielausgabe überlagert.
- ▶ Die Klasse *Video* besitzt eine Instanzvariable für den Titel. Die Variable wird über einen Konstruktor gesetzt und kann über einen Getter abgefragt werden. Die *toString()*-Methode wird passend zur Beispielausgabe überlagert.
- ▶ Die Klasse *MediaLib* repräsentiert eine Medien-Bibliothek und speichert intern beliebig viele Objekte der Klassen *Audio* und *Video*. Die Objekte lassen sich über *add()*-Methoden hinzufügen. Die Methode *print()* gibt alle enthaltenen *Audio*- und *Video*-Objekte auf Konsole aus.
- ▶ Die Klasse *MediaApp1* erzeugt ein Objekt der Medien-Bibliothek, fügt einige Audio-Titel und Videos hinzu und gibt diese über einen Aufruf der *print()*-Methode auf Konsole aus.

Beispielausgabe:

```
Rusk & Bread: Marie
Sheryl Crow: Hard to make a stand
"Was ist mit Bob?"
"Der Marsianer"
```

■ **Aufgabe 77.** (Klasse: MediaApp2) In dieser und den folgenden Aufgaben werden wir die Klassen aus Aufgabe 76 wie in Abbildung 6.2 dargestellt erweitern. Erweitern Sie zunächst die Klasse *Video* um die Größe der einzelnen Videobilder. Erstellen Sie hierzu eine Klasse *ImageSize* mit der Breite und Höhe eines Bildes in Pixeln. Die Breite und Höhe werden über den Konstruktor gesetzt und können über Getter abgefragt werden.

■ **Aufgabe 78.** (Klasse: MediaApp2) Erweitern Sie die Klassen *Audio* und *Video* um die Dauer der entsprechenden Titel. Erstellen Sie hierfür eine Klasse *Duration*, die intern die Zeit in Minuten und Sekunden speichert.

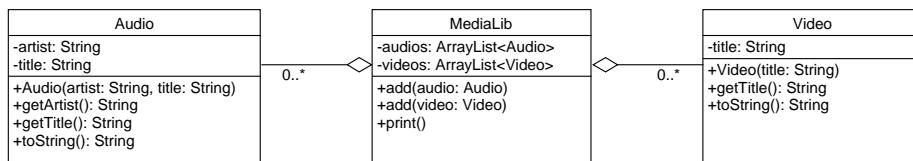


Abbildung 6.1: Initiales Klassendiagramm

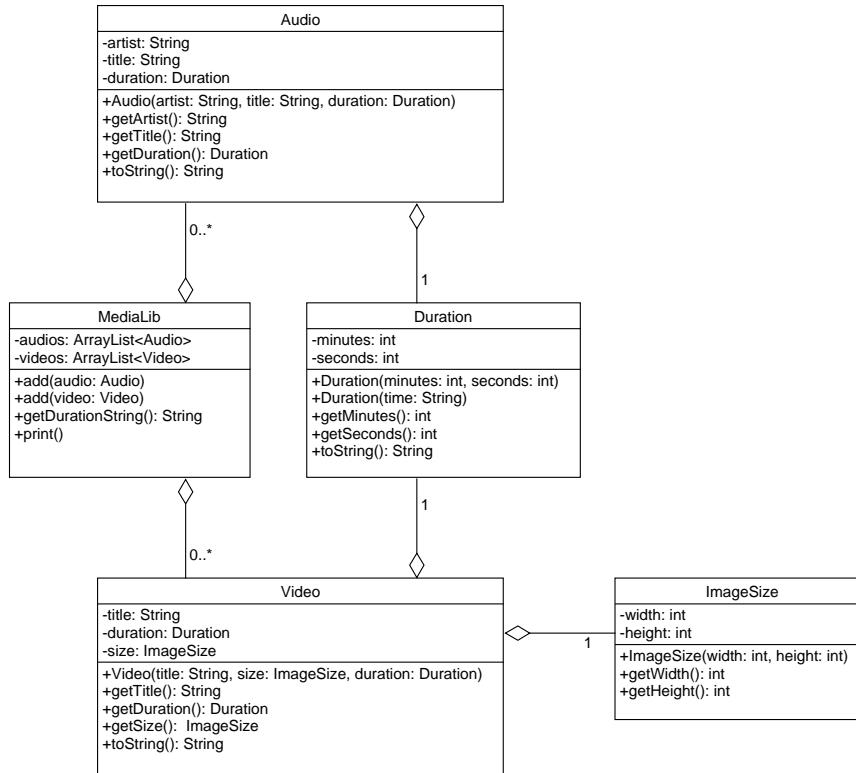


Abbildung 6.2: Erweitertes Klassendiagramm

- ▶ Die Klasse besitzt einen Konstruktor, dem die Minuten und Sekunden übergeben werden. Beispielsweise initialisiert die Anweisung `new Duration(1, 30)` ein Objekt, das eine Dauer von $1\frac{1}{2}$ Minuten repräsentiert.
- ▶ Der Konstruktor behandelt übergebene Sekunden, die über 59 hinausgehen, entsprechend. Beispielsweise initialisiert die Anweisung `new Duration(1, 90)` ein Objekt, das eine Dauer von $2\frac{1}{2}$ Minuten repräsentiert.
- ▶ Die Minuten und Sekunden lassen sich über Getter zurückgeben.
- ▶ Überlagern Sie die `toString()`-Methode derart, dass die Zeit im Format `mm:ss` mit Minuten *m* und Sekunden *s* zurückgegeben wird. Sowohl die Minuten als auch die Sekunden werden mit mindestens zwei Zeichen und bei Werten kleiner 10 mit führender Null dargestellt.

Passen Sie die Klassen *Audio* und *Video* unter Verwendung der `toString()`-Methode der Klasse *Duration* derart an, dass die Beispieldausgabe folgendermaßen formatiert wird:

```
Rusk & Bread: Marie (04:15)
Sheryl Crow: Hard to make a stand (03:08)
"Was ist mit Bob?" (99:00)
"Der Marsianer" (151:00)
```

Aufgabe 79. (Klasse: MediaApp2) Erweitern Sie die Klasse *Duration* um einen zweiten Konstruktor, dem die Dauer als Zeichenkette im Format `mm:ss` mit Minuten *m* und Sekunden *s* übergeben wird. Beispieldaufruf:

```
Duration duration = new Duration("3:08");
```

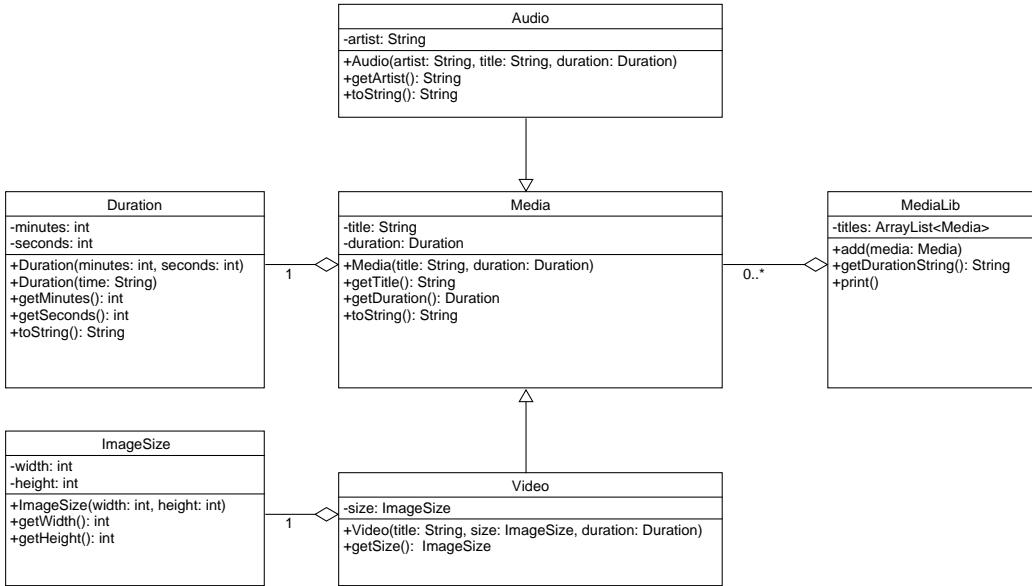


Abbildung 6.3: Klassendiagramm mit Vererbung

Aufgabe 80. (Klasse: MediaApp2) Erweitern Sie die Klasse *MediaLib* um eine Methode *getDurationString()*, welche die gesamte Spieldauer aller in der Bibliothek vorhandener Audio-Titel und Videos als String im Format `h:mm:ss` mit Stunden *h*, Minuten *m* und Sekunden *s* zurückgibt. Verwenden Sie zum Durchlaufen der Listen ausschließlich *foreach*-Schleifen. Beispielausgabe:

```

Rusk & Bread: Marie (04:15)
Sheryl Crow: Hard to make a stand (03:08)
"Was ist mit Bob?" (99:00)
"Der Marsianer" (151:00)

The media collection contains fun for 4:17:23 hours.
  
```

6.2 Vererbung

Eventuell sind Ihnen zwei Dinge aufgefallen. Zunächst einmal haben wir in den bisherigen Aufgaben noch gar keine Vererbung verwendet. Zudem stört in der Klasse *MediaLib* die Unterscheidung bzw. Sonderbehandlung von Objekten der Klassen *Audio* und *Video*. Also lassen Sie uns das Konzept der Vererbung nutzen, um Abhilfe zu schaffen und den Code zu vereinfachen (Abb. 6.3)!

Aufgabe 81. (Klasse: MediaApp3) Analysieren Sie die Klassen *Audio* und *Video* auf gemeinsame Instanzvariablen.

- ▶ Erstellen Sie als gemeinsame Basisklasse eine Klasse *Media*, in welche Sie diese Variablen verschieben. Belassen Sie die Variablen *private* und initialisieren Sie sie nicht über Setter, sondern über einen entsprechenden Konstruktor.
- ▶ Überlagern Sie die *toString()*-Methode der Klasse *Media* derart, dass sie den Titel sowie die Dauer im Format `"Titel" (mm:ss)` mit Minuten *m* und Sekunden *s* zurückgibt, z. B.:

```
"Marie" (04:15)
```

Aufgabe 82. (Klasse: MediaApp3) Überlagern Sie die *toString()*-Methode der Klasse *Audio* derart, dass sie Interpreten, Titel sowie die Dauer im Format `Interpret : "Titel" (mm:ss)` mit Minuten

m und Sekunden *s* zurückgibt. Rufen Sie hierbei explizit die *toString()*-Methode der Basisklasse *Media* auf. Beispielausgabe:

Rusk & Bread: "Marie" (04:15)

■ **Aufgabe 83.** (Klasse: MediaApp3) Ersetzen Sie in *MediaLib* die Listen für Objekte der Klassen *Audio* und *Video* durch eine gemeinsame Liste für Referenzen des Datentyps *Media*. Vereinfachen Sie den weiteren Quelltext derart, dass *MediaLib* nicht mehr zwischen Objekten der Klassen *Audio* und *Video* unterscheidet. Die Beispielausgabe soll sich im Vergleich zu Aufgabe 80 nicht ändern. Welche Methoden der in der *MediaLib* gespeicherten Objekte können Sie über die Referenz vom Typ *Media* nicht aufrufen?

6.3 Fragen

Frage 44. Wie viele direkte Basisklassen besitzt eine Klasse mindestens?

Frage 45. Wie viele direkte Basisklassen kann eine Klasse höchstens besitzen?

Frage 46. Wie viele und gegebenenfalls welche Basisklassen hat eine Klasse, für die keine Basisklasse über das Schlüsselwort *extends* angegeben ist?

Frage 47. Wie lässt sich verhindern, dass von einer Klasse eine Subklasse gebildet wird?

Frage 48. Wie kann man in einem Konstruktor einen Konstruktor der Basisklasse aufrufen?

Frage 49. Warum kann man einer Variablen vom Typ *Object* Referenzen eines beliebigen Typs zuweisen?

Frage 50. Nennen Sie zwei Methoden der Klasse *Object*.

Frage 51. Erläutern Sie den Effekt der Modifier *public*, *private*, *protected* und *final* auf Instanzvariablen.

Frage 52. In welcher Situation kann man auf Elemente einer Klasse, die als *private* markiert sind, trotzdem zugreifen?

Frage 53. Erläutern Sie Überladen (*Overloading*) von Methoden.

Frage 54. Erläutern Sie Überlagern (*OVERRIDING*) von Methoden.

Frage 55. Nehmen Sie Stellung zu folgender Aussage: „Auf eine Klassenvariable können grundsätzlich keine Objekte anderer Klassen zugreifen.“

Frage 56. Erläutern Sie den Unterschied zwischen der Operation *==* und der Methode *equals()*.

Kapitel 7

Abstrakte Elemente

Der Nutzen abstrakter Elemente wie beispielsweise abstrakter Basisklassen und Interfaces lässt sich am besten anhand konkreter Anwendungsbeispiele erkennen. In diesem Zusammenhang werden wir zunächst die Approximation von Nullstellen mathematischer Polynome zweiten Grades wieder aufgreifen und um quasi beliebige stetig differenzierbare Funktionen erweitern. Nachfolgend betrachten wir mit der Iteration durch Daten sowie der Benachrichtigung bei Ereignissen typische Anwendungsfälle.

7.1 Nullstellen mathematischer Funktionen

Aufgabe 84. (Klasse: FunctionApp) In Aufgabe 35 auf Seite 20 haben wir die Klasse *Parabola* zur Repräsentation eines mathematischen Polynoms 2. Grades erstellt. Von Objekten dieser Klasse haben wir insbesondere Nullstellen bestimmt. Erzeugen bzw. erweitern Sie zur Verallgemeinerung auf stetig differenzierbare Funktionen folgende Klassen (Abb. 7.1):

- ▶ Eine Klasse *Function* mit abstrakten Methoden $f()$ und $f1()$ zur Rückgabe des Funktionswertes $f(x)$ bzw. der ersten Ableitung $f'(x) = \frac{d}{dx}f(x)$.
- ▶ Die Klasse *Parabola* erbt von *Function*.
- ▶ Die Klasse *Logarithm* erbt von *Function* und repräsentiert Logarithmen $f(x) = \log_b(x)$ zu einer über den Konstruktor festzulegenden Basis b , $b \in \mathbb{N} \setminus \{1\}$:

$$f(x) = \log_b(x) = \frac{\ln(x)}{\ln(b)} \quad (7.1)$$

$$f'(x) = \frac{1}{\ln(b) \cdot x} \quad (7.2)$$

Erzeugen Sie eine Tabelle mit Funktionswerten von $f(x) = x^2$ und $f(x) = \log_2(x)$. Beispielausgabe:

x	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0
$f(x) = 1.0 * x^2$	1.0	4.0	9.0	16.0	25.0	36.0	49.0	64.0
$f(x) = \log_2(x)$	0.0	1.0	1.6	2.0	2.3	2.6	2.8	3.0

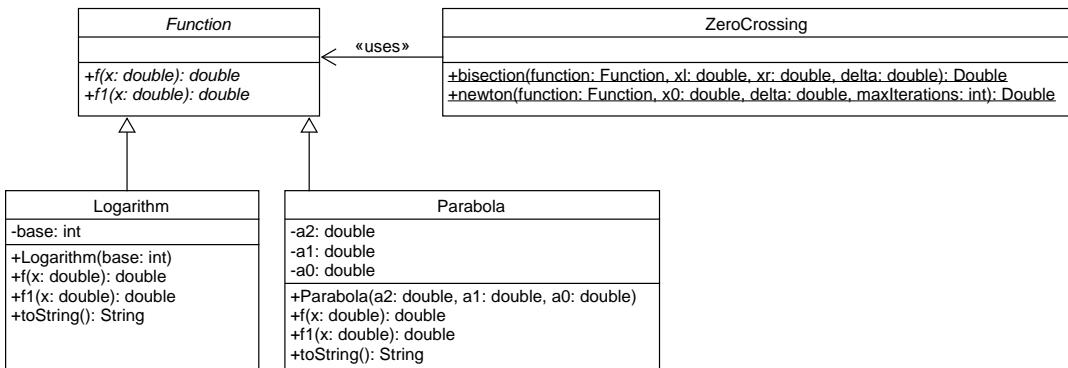


Abbildung 7.1: Klassendiagramm zu mathematischen Funktionen (Aufgabe 84 und 85)

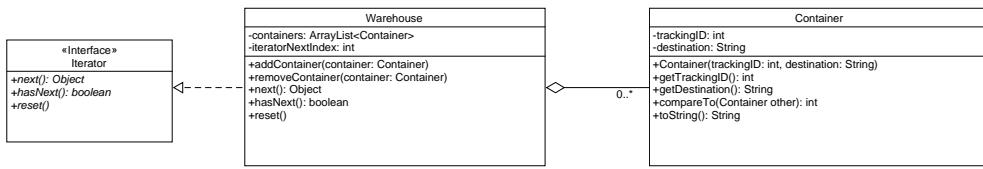


Abbildung 7.2: Klassendiagramm zur Iteration durch ein Lager vom Typ *Warehouse* (Aufgabe 86 und 87)

■ **Aufgabe 85.** (Klasse: ZeroCrossingApp) In Aufgabe 67 und 68 auf Seite 33 waren die Bestimmung von Nullstellen mittels Newton- und Bisektionsverfahren umzusetzen, wobei wir uns auf Polynome 2. Grades beschränkt haben. Modifizieren Sie die Lösungen unter Verwendung der Klasse *Function* aus Aufgabe 84 derart, dass Nullstellen von beliebigen stetig differenzierbaren Funktionen bestimmt werden können (Abb. 7.1):

- ▶ Die Klasse *ZeroCrossing* enthält ausschließlich statische Methoden zur Bestimmung der Nullstellen mittels Newton- bzw. Bisektionsverfahren.
- ▶ Die zu untersuchende Funktion $f(x)$ wird den Methoden als *Function*-Objekt übergeben.
- ▶ Sofern dieser bestimmt werden kann, geben die Methoden den x -Wert der approximierten Nullstelle zurück. Andernfalls wird *null* zurückgegeben.

Beispielausgabe:

```

Zero-crossing for f(x) = -1.0 * x^2 + 4.0:
Newton      : x0 = 2.000000, f(x0) = -8.882e-15
Bisection   : x0 = 2.000001, f(x0) = -3.052e-06

Zero-crossing for f(x) = log6(x):
Newton      : x0 = 1.000000, f(x0) = -2.394e-13
Bisection   : x0 = 0.999999, f(x0) = -5.323e-07

```

7.2 Sortieren und Iterieren

■ **Aufgabe 86.** (Klasse: ContainerApp) Erstellen Sie eine Klasse *Container* mit folgenden Eigenschaften (Abb. 7.2):

- ▶ Container besitzen eine Sendeverfolgungsnummer vom Typ *int* sowie einen Zielort vom Typ *String*. Die Instanzvariablen werden mit dem Konstruktor übergebenen Werten initialisiert und lassen sich über Getter zurückgeben.
- ▶ Die Klasse überlagert die *toString()*-Methode derart, dass die Ausgabe dem Format der nachfolgenden Beispielausgabe entspricht.
- ▶ Die Klasse implementiert das Interface *Comparable* in der Form *Comparable<Container>*. Die Rückgabe der *compareTo()*-Methode führt zu einer alphabetisch aufsteigenden Sortierung von *Container*-Objekten, wobei die Groß- und Kleinschreibung ignoriert wird.

Erstellen Sie ein Programm, das mehrere *Container*-Objekte erzeugt, diese über die Klasse *Collections* sortiert und auf Konsole ausgibt. Beispielausgabe:

```

Initial list:
Container No.72725462 to London
Container No.82647552 to Hamburg
Container No.23753255 to Melbourne
Container No.95252532 to Bristol

```

```
Sorted by destination:
Container No.95252532 to Bristol
Container No.82647552 to Hamburg
Container No.72725462 to London
Container No.23753255 to Melbourne
```

■ **Aufgabe 87.** (Klasse: WarehouseApp) Im Folgenden wollen wir aufbauend auf Aufgabe 86 eine Klasse *Warehouse* zur Repräsentation eines Container-Lagers erstellen (Abb. 7.2). Prinzipiell könnte zur Speicherung der Referenzen auf *Container*-Objekte beispielsweise ein Feld (*Container[]*) oder eine Liste (*ArrayList<Container>*) verwendet werden. *Warehouse* soll jedoch nach außen verstecken, über was für eine Datenstruktur die interne Speicherung umgesetzt ist. Statt dessen soll sie eine Möglichkeit bieten, unabhängig von der tatsächlichen Datenstruktur durch alle eingelagerten Container zu iterieren:

- ▶ Erstellen Sie ein Interface *Iterator* mit den Methoden *next()*, *hasNext()* sowie *reset()*.
- ▶ Erstellen Sie eine Klasse *Warehouse* mit Methoden *addContainer()* und *removeContainer()* zum Einlagern bzw. Auslagern von *Container*-Objekten.
- ▶ Implementieren Sie das Interface *Iterator* in *Warehouse*, um über die eingelagerten Container zu iterieren. Die Methode *next()* gibt das nächste Element zurück und setzt den internen Index um eine Position weiter. Die Methode *hasNext()* gibt *false* zurück, falls kein weiteres Element existiert („Ende der Liste“). Mittels *reset()* wird der interne Index zurück auf den Anfang der Liste gesetzt.

Erstellen Sie ein Programms, das mehrere Container einlagert und anschließend zur Ausgabe des Lagerbestandes auf Konsole durch *Warehouse* iteriert. Beispieldaten:

```
Container No.72725462 to London
Container No.82647552 to Hamburg
Container No.23753255 to Melbourne
Container No.95252532 to Bristol
```

7.3 Benachrichtigen von Beobachtern

In den Aufgaben dieses Abschnitts sollen Sie einen Service für Nachrichten umsetzen. Die Nachrichten sollen insbesondere von unterschiedlichen Empfängern abonniert werden.

■ **Aufgabe 88.** (Klasse: NewsMain) Erstellen Sie zunächst eine Klasse *News* zur Repräsentation einer Nachricht bzw. Meldung (Abb. 7.3) und überprüfen Sie diese mit Hilfe eines ausführbaren Programms:

- ▶ Die Klasse *News* besitzt Instanzvariablen für das Nachrichten-Ressort (z. B. Kultur, Sport oder Politik) sowie den Nachrichtentext. Die Werte dieser Variablen werden dem Konstruktor übergeben und können über Getter zurückgegeben werden.
- ▶ Objekte besitzen eine Identifikationsnummer (*ID*), die über einen Getter abgefragt werden kann. Die IDs sind für alle Objekte unterschiedlich, wobei das erste erzeugte Objekt den Wert 1, das Zweite den Wert 2, das Dritte den Wert 3 usw. erhält.
- ▶ Überlagern Sie die *toString()*-Methode derart, dass bei entsprechenden Werten nachfolgende Beispieldaten erzeugt werden.

^{~1} Panorama: Panzerknacker rauben Dagobert Duck aus!

^{~2} Kultur: Heute mal nichts Neues

■ **Aufgabe 89.** (Klasse: NewsMain) Erstellen Sie eine Klasse *NewsProvider* zur Repräsentation eines Nachrichten-Archivs (Abb. 7.3):

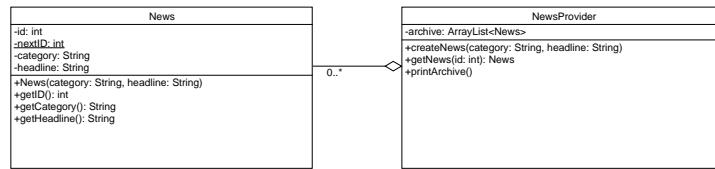


Abbildung 7.3: Initiales Klassendiagramm

- ▶ Über die Methode `createNews()` wird eine neue Nachricht erzeugt und dem Archiv hinzugefügt. Der Methode werden das Ressort sowie der Text der Nachricht übergeben.
- ▶ Die Methode `getNews()` erhält als Parameter eine Nachrichten-ID und gibt die zugehörige Nachricht zurück.
- ▶ Die Methode `printArchive()` gibt alle bisherigen Nachrichten wie in der nachfolgenden Beispielausgabe auf Konsole aus.

Nachrichten-Archiv:

```

~1 Panorama: Panzerknacker rauben Dagobert Duck aus!
~2 Kultur: Heute mal nichts Neues

```

Aufgabe 90. (Klasse: NewsMain) Nun wollen wir Nachrichten-Kanäle umsetzen, die benachrichtigt werden, sobald im `NewsProvider` eine neue Nachricht erstellt wird (Abb. 7.4). Fügen Sie hierfür zunächst eine Klasse `NewsChannel` hinzu:

- ▶ Dem Konstruktor wird der Name des Nachrichten-Kanals sowie das Ressort, aus dem Nachrichten angezeigt werden sollen, übergeben (z. B. `new NewsChannel("Sport-Kanal", "Sport")`).
- ▶ Der Methode `onNews()` werden als Parameter ein Archiv vom Typ `NewsProvider` sowie eine Nachrichten-ID übergeben. Die Methode gibt die zugehörige Nachricht dann und nur dann auf Konsole aus, wenn das Ressort der Nachricht mit dem Ressort des Nachrichten-Kanals übereinstimmt. Die Formatierung muss der am Ende dieser Aufgabe angegebenen Beispielausgabe entsprechen.

Modifizieren Sie zudem die Klasse `NewsProvider` wie folgt:

- ▶ Die Klassen enthält eine Liste von Objekten des Typs `NewsChannel`. Nachrichten-Kanäle lassen sich über die Methoden `registerReceiver()` bzw. `deregisterReceiver()` zur Liste hinzufügen bzw. aus dieser entfernen.
- ▶ Bei der Erzeugung einer neuen Nachricht werden alle in der Liste enthaltenen Objekte vom Typ `NewsChannel` durch Aufruf ihrer Methode `onNews()` über die neue Nachricht in Kenntnis gesetzt.

Für die nachfolgende Beispielausgabe wurde der mittels `new NewsChannel("Sport-Kanal", "Sport")` erzeugte Kanal über alle fünf erzeugten Nachrichten informiert, hat jedoch nur die aus dem Ressort `Sport` als Meldung ausgegeben. Entsprechend hat ein über `new NewsChannel("Wissen-1", "Wissenschaft")` erzeugter Kanal nur die Meldung aus dem Ressort `Wissenschaft` ausgegeben:

```

+++ Sport-Kanal aktuell ***
HSV gewinnt in Kreisliga B

+++ Wissen-1 aktuell ***
Skelett eines Dino-Teenagers mit Sitzsack gefunden!

Nachrichten-Archiv:
~1 Panorama: Panzerknacker rauben Dagobert Duck aus!
~2 Kultur: Heute mal nichts Neues
~3 Sport: HSV gewinnt in Kreisliga B
~4 Panorama: Wahnsinn – Tomate sieht rot!
~5 Wissenschaft: Skelett eines Dino-Teenagers mit Sitzsack gefunden

```

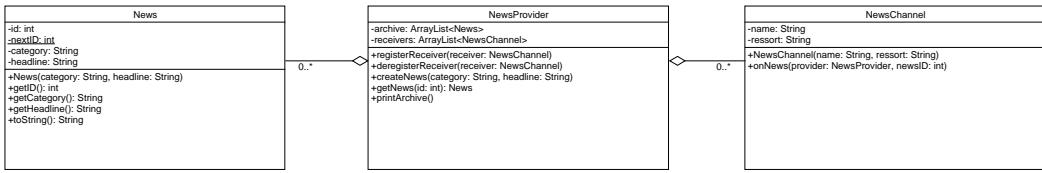


Abbildung 7.4: Klassendiagramm nach Erweiterung um *NewsChannel*

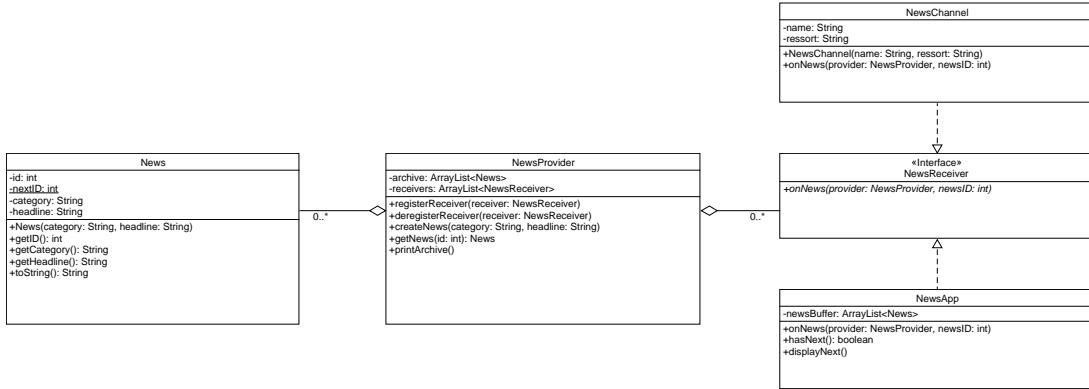


Abbildung 7.5: Die Klassen *NewsChannel* und *NewsApp* sind für Objekte vom Typ *NewsProvider* hinter dem Interface *NewsReceiver* versteckt.

Aufgabe 91. (Klasse: NewsMain) Warum sollten wir unsere wertvollen Nachrichten nur den Kanälen überlassen?¹ Daher fügen wir jetzt eine App hinzu, die über eine Klasse *NewsApp* alle Nachrichten erhält, zunächst puffert (d. h. zwischenspeichert) und bei Bedarf ausgibt. Hierfür muss unser *NewsProvider* lediglich auch registrierte Objekte vom Typ *NewsApp* über neue Nachrichten informieren. Aber was, wenn weitere Arten von Abnehmern hinzukommen (z. B. Live-Stream, Datenbank)? Müssen wir dann für jede der zugehörigen Klassen in *NewsProvider* eine eigene Liste anlegen und jeweils eigene Methoden *registerReceiver()* und *deregisterReceiver()* implementieren? Und müssen wir bei einer neuen Nachricht all diese Listen einzeln durchlaufen, um alle registrierten Objekte zu benachrichtigen? Oje ...

- Denken Sie über das Problem nach und setzen Sie eine Lösung um, in der *NewsProvider* die Klassen der zu benachrichtigen Objekte nicht kennen und nicht unterscheiden braucht. Insbesondere darf es nur eine gemeinsame Liste und nur eine Implementierung der Methoden *registerReceiver()* und *deregisterReceiver()* geben.
- Implementieren Sie die Klasse *NewsApp* derart, dass sie alle von *NewsProvider* eingehenden Nachrichten speichert. Über die Methode *hasNext()* kann abgefragt werden, ob aktuell mindestens eine Nachricht im Speicher ist. Über *displayNext()* wird die älteste Nachricht aus dem Speicher entfernt und auf Konsole ausgegeben.

Für den Fall, dass Sie nicht alleine zum Ziel kommen, enthält Abbildung 7.5 das Klassendiagramm einer möglichen Lösung. Nachfolgend ist zudem ein Beispielausgabe angegeben, bei der zunächst zwei Kanäle und eine App registriert und dann fünf Nachrichten erzeugt wurden. Vor der Ausgabe des Archivs wurden alle in der App gespeicherten Nachrichten abgerufen und ausgegeben.

¹Hah, soweit kommt's noch!

```

+++ Sport-Kanal aktuell ===
HSV gewinnt in Kreisliga B

+++ Wissen-1 aktuell ===
Skelett eines Dino-Teenagers mit Sitzsack gefunden

App display <-> Panzerknacker rauben Dagobert Duck aus!
App display <-> Heute mal nichts Neues
App display <-> HSV gewinnt in Kreisliga B
App display <-> Wahnsinn – Tomate sieht rot!
App display <-> Skelett eines Dino-Teenagers mit Sitzsack gefunden

Nachrichten-Archiv:
~1 Panorama: Panzerknacker rauben Dagobert Duck aus!
~2 Kultur: Heute mal nichts Neues
~3 Sport: HSV gewinnt in Kreisliga B
~4 Panorama: Wahnsinn – Tomate sieht rot!
~5 Wissenschaft: Skelett eines Dino-Teenagers mit Sitzsack gefunden

```

7.4 Fragen

Abstrakte Klassen und Methoden

Frage 57. Was bewirkt der Modifier *abstract* für Methoden?

Frage 58. Warum dürfen abstrakte Methoden nicht die Modifier *private* oder *static* besitzen?

Frage 59. Wann *muss* eine Klasse als abstrakt deklariert werden?

Frage 60. Kann eine abstrakte Klasse Konstruktoren besitzen?

Frage 61. Kann eine Klasse oder Methode als *abstract final* deklariert werden?

Interfaces

Frage 62. Was ist der Unterschied zwischen einem Interface und einer abstrakten Klasse?

Frage 63. Darf ein Interface von einer abstrakten Klasse erben?

Frage 64. Wie viele Konstruktoren kann ein Interface haben?

Frage 65. Welche Elemente, die Klassen besitzen können, dürfen Interfaces nicht besitzen?

Frage 66. Welche Modifier erhalten alle Methoden eines Interfaces – selbst dann, wenn man sie nicht explizit hinschreibt?

Frage 67. Wie wird der Rückgabewert der *compareTo()*-Methode gedeutet?

Praktikumsaufgaben (Casino)

Bearbeiten Sie nun Praktikum 2 auf Seite 91.

- ▶ Stellen Sie sicher, dass Sie das Praktikum lösen können, bevor Sie fortfahren.
- ▶ Es ist sinnvoll auch die weiteren Versuche zum Praktikum 2 zu bearbeiten.

Praktikumsaufgaben (Geografische Koordinaten)

Bearbeiten Sie nun Praktikum 2 auf Seite 107.

- Stellen Sie sicher, dass Sie das Praktikum lösen können, bevor Sie fortfahren.
- Es ist sinnvoll auch die weiteren Versuche zum Praktikum 2 zu bearbeiten.

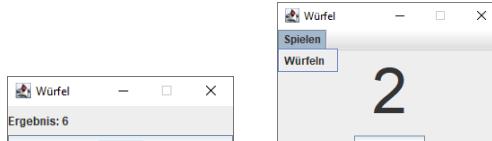
Kapitel 8

Grafische Benutzeroberflächen (GUI)

In den Aufgaben dieses Kapitels sind Programme mit grafischen Benutzeroberflächen zu erstellen. Den Aufbau der Oberflächen dürfen Sie frei gestalten. Die gegebenen Abbildungen sind lediglich Beispiele, von denen Sie abweichen dürfen. Beachten Sie zudem, dass die Instanzvariablen aller Klassen als *private* oder *protected* deklariert sein müssen.

8.1 Würfel

■ **Aufgabe 92.** (Klasse: DiceApp1) Erstellen Sie ein Programm, in dem über einen Button ein Würfel geworfen werden kann. Das Ergebnis des Wurfes wird in Textform in der grafischen Benutzeroberfläche dargestellt (Abb. 8.1a). Stellen Sie sicher, dass sich das Programm-Fenster nicht von Benutzern verkleinern oder vergrößern lässt.



(a) Aufgabe 92

(b) Aufgabe 93

Abbildung 8.1: Beispiel-Ausgaben zum Würfeln

■ **Aufgabe 93.** (Klasse: DiceApp2) Erweitern Sie die Lösung zu Aufgabe 92 derart, dass sowohl über einen Button als auch über das Menü gewürfelt werden kann (Abb. 8.1b).

■ **Aufgabe 94.** (Klasse: DiceImageApp1) Modifizieren Sie die Lösung zu Aufgabe 92 derart, dass das Ergebnis des letzten Wurfes grafisch als Würfel angezeigt wird (Abb. 8.2).

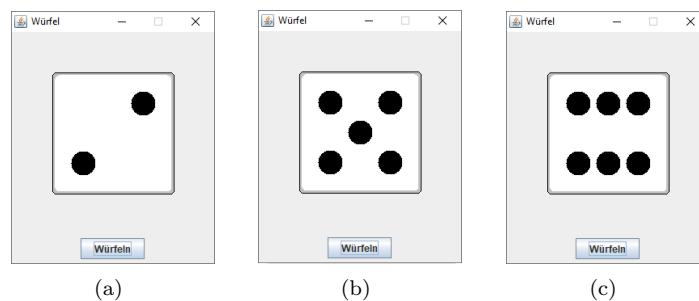


Abbildung 8.2: Beispiel-Ausgaben zu Aufgabe 94

■ **Aufgabe 95.** (Klasse: DiceImageApp2) Erweitern Sie die Anwendung aus Aufgabe 94 derart, dass zwei Würfeln gleichzeitig geworfen werden (Abb. 8.3).

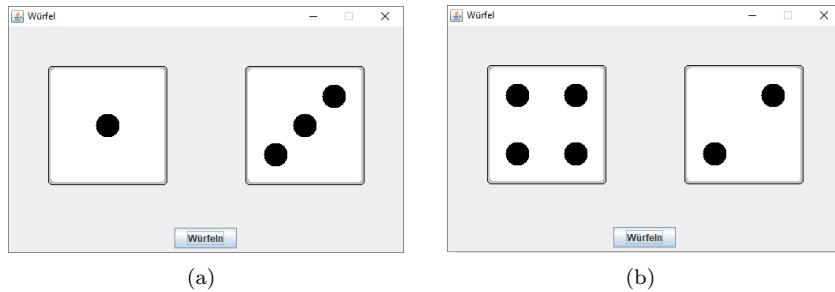


Abbildung 8.3: Beispiel-Ausgaben zu Aufgabe 95

■ **Aufgabe 96.** (Klasse: DiceStatisticsApp) Erweitern Sie die Anwendung aus Aufgabe 94 derart, dass der aktuelle Mittelwert aller Würfe als Zahlenwert sowie der Verlauf des Mittelwertes über die Anzahl der Würfe grafisch ausgegeben werden. Fügen Sie eine Möglichkeit hinzu, den bisherigen Verlauf der Mittelwerte zu löschen (Abb. 8.4).

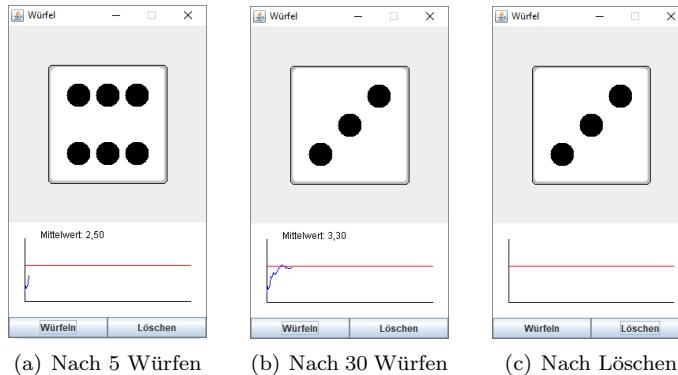


Abbildung 8.4: Beispiel-Ausgaben zu Aufgabe 96

8.2 Mehrwertsteuer

■ **Aufgabe 97.** (Klasse: VATApp1) Erstellen Sie eine Anwendung zur Bestimmung der in einem Betrag enthaltenen Mehrwertsteuer in Höhe von 19 %. Die Anwendung ermöglicht die Eingabe des Brutto-Betrages über ein Element vom Typ *JTextField* und gibt die enthaltene Mehrwertsteuer aus (Abb. 8.5). Hinweise:

- ▶ *JTextField*-Objekte erzeugen bei Beendigung der Eingabe mittels [Eingabe]-Taste ein *ActionEvent*.
- ▶ Gehen Sie vorerst davon aus, dass Benutzer einen Betrag im korrekten Format (d. h. keine Buchstaben oder ähnliches) eingeben.

■ **Aufgabe 98.** (Klasse: VATApp2) Erweitern Sie die Lösung zu Aufgabe 97 derart, dass über einen Auswahlkasten (*Check box*) vom Typ *JCheckBox* der verminderte Mehrwertsteuersatz in Höhe von 7 % gewählt werden kann (Abb. 8.6). Die enthaltene Mehrwertsteuer wird bei Änderung des Steuersatzes ohne weitere Benutzereingabe automatisch aktualisiert.



Abbildung 8.5: Beispiel-Ausgabe zu Aufgabe 97

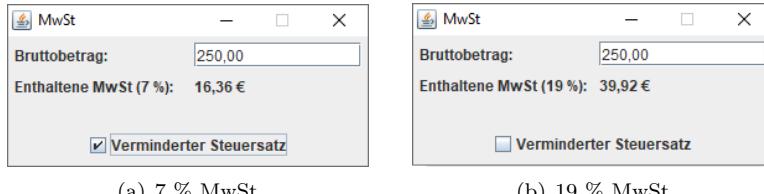


Abbildung 8.6: Beispiel-Ausgaben zu Aufgabe 98

■ **Aufgabe 99.** (Klasse: VATApp3) Erweitern Sie die Lösung zu Aufgabe 98 derart, dass der Steuersatz in Höhe von 7 % bzw. 19 % über Optionsfelder (*Radio buttons*) gewählt werden kann (Abb. 8.7). Die enthaltene Mehrwertsteuer wird bei Änderung des Steuersatzes ohne weitere Benutzereingabe automatisch aktualisiert.

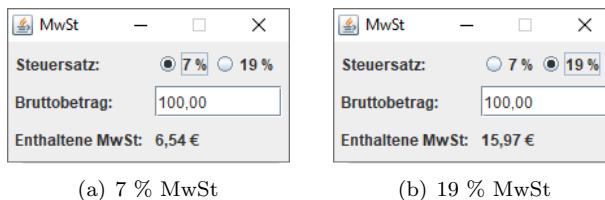


Abbildung 8.7: Beispiel-Ausgaben zu Aufgabe 99

8.3 Graphen mathematischer Funktionen

■ **Aufgabe 100.** (Klasse: PlotApp) Erstellen Sie eine Anwendung zur Darstellung mathematischer Funktionen (Abb. 8.8a):

- ▶ Die Achsen des Koordinatensystems werden mindestens als Geraden dargestellt. Für die minimalen und maximalen Werte der Achsen, d. h. die Größe $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}] \subset \mathbb{R}^2$ des abgebildeten Bereichs, existieren Instanzvariablen mit entsprechenden Settern.
- ▶ Die Funktion $f(x)$ wird durch ein Objekt der Klasse *Parabola* repräsentiert (Aufgabe 84 auf Seite 48) und als Kurvenverlauf dargestellt.
- ▶ Der Hintergrund des Zeichenbereiches ist weiß.

■ **Aufgabe 101.** (Klasse: PlotApp) Erweitern Sie Ihr Programm aus Aufgabe 100 wie folgt (Abb. 8.8b):

- ▶ Die Werte der Parameter a_2 , a_1 und a_0 von $f(x) = a_2x^2 + a_1x + a_0$ lassen sich über die Benutzeroberfläche verändern.
- ▶ Die minimalen und maximalen Werte der Achsen lassen sich über die Benutzeroberfläche verändern.

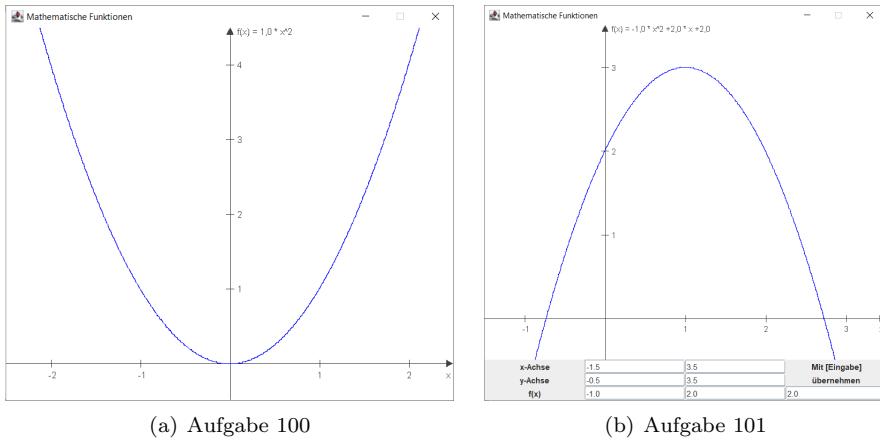


Abbildung 8.8: Darstellung mathematischer Funktionen

■ **Aufgabe 102.** (Klasse: PlotApp) Erweitern Sie Ihr Programm aus Aufgabe 101 wie folgt (Abb. 8.9):

- ▶ Durch Linksklick mit der Maus lässt sich ein Startwert x_s auswählen.
- ▶ Ausgehend vom Punkt $P_s = (x_s, f(x_s))$ wird mittels Newtonverfahren eine Nullstelle $P_0 = (x_0, 0)$ approximiert. Verwenden Sie die Klasse *ZeroCrossing* aus Aufgabe 85 auf Seite 49.
- ▶ Die Punkte P_s und P_0 werden im Graphen markiert.

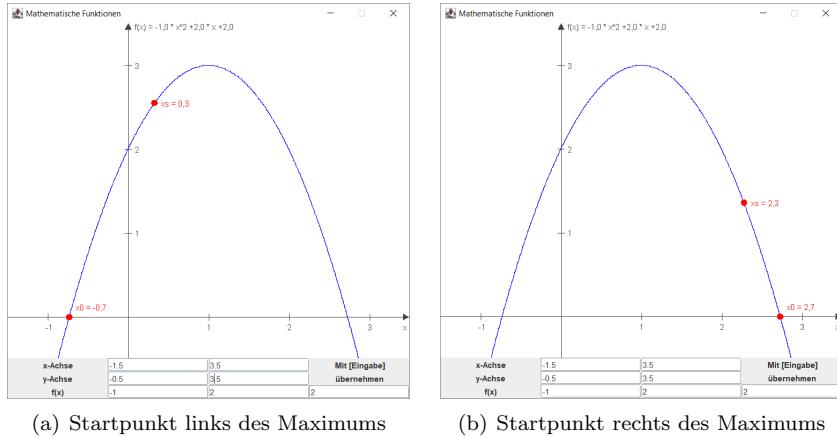


Abbildung 8.9: Approximation von Nullstellen über interaktiv gesetzte Startpunkte (Aufgabe 102)

8.4 Grünpflanzen und weitere grafische Elemente

■ **Aufgabe 103.** (Klasse: FarnApp) Lassen Sie uns etwas Wunderschönes erzeugen: Barnsleys Farn¹. Gegeben sei eine Menge von N durch Ortsvektoren $\mathbf{x}_n = (x_n, y_n)^T$, $n = 1, 2, \dots, N$, definierten Punkten. Ausgehend vom Startpunkt $\mathbf{x}_1 = (0, 0)^T$ ergibt sich der jeweils nächste Punkt

¹Paul Willmott: *Grundkurs Machine Learning*, Rheinwerk Verlag, 2000, Seite 30f

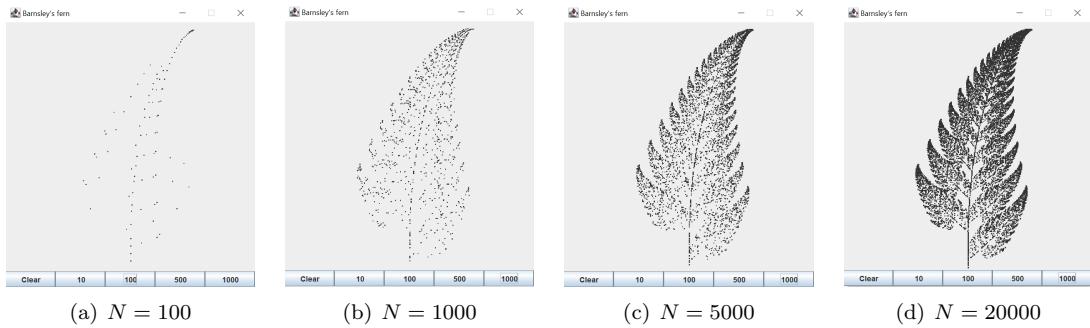


Abbildung 8.10: Beispiel-Ausgaben zu Aufgabe 103

durch Anwendung einer der folgenden linearen Abbildungen:

$$\mathbf{x}_{i+1} = f_1(\mathbf{x}_i) = \begin{pmatrix} 0 & 0 \\ 0 & 0,16 \end{pmatrix} \mathbf{x}_i \quad (8.1)$$

$$\mathbf{x}_{i+1} = f_2(\mathbf{x}_i) = \begin{pmatrix} 0,85 & 0,04 \\ -0,04 & 0,85 \end{pmatrix} \mathbf{x}_i + \begin{pmatrix} 0 \\ 1,6 \end{pmatrix} \quad (8.2)$$

$$\mathbf{x}_{i+1} = f_3(\mathbf{x}_i) = \begin{pmatrix} 0,2 & -0,26 \\ 0,23 & 0,22 \end{pmatrix} \mathbf{x}_i + \begin{pmatrix} 0 \\ 1,6 \end{pmatrix} \quad (8.3)$$

$$\mathbf{x}_{i+1} = f_4(\mathbf{x}_i) = \begin{pmatrix} -0,15 & 0,28 \\ 0,26 & 0,24 \end{pmatrix} \mathbf{x}_i + \begin{pmatrix} 0 \\ 0,44 \end{pmatrix} \quad (8.4)$$

Die Abbildung wird jeweils zufällig gewählt, wobei f_1 mit einer Wahrscheinlichkeit von 1%, f_2 mit 85%, f_3 mit 7% sowie f_4 mit 7% angewendet wird.

Erstellen Sie ein Programm, das die Punktmenge visualisiert. Sehen Sie hierbei die Möglichkeit vor, die Anzahl N an Punkten vorzugeben (Abb. 8.10).

Aufgabe 104. (Klasse: SliderApp) Der Schwerpunkt dieser Aufgabe ist nicht die Umsetzung von (halbwegs) sinnvoller Funktionalität. Stattdessen sollen Sie die Verwendung weiterer grafischer Elemente kennenlernen. Erstellen Sie eine Anwendung, die einen Schieberegler (*Slider*) sowie einen Fortschrittsbalken (*Progress bar*) enthält. Bei Änderung des über den Schieberegler eingestellten Wertes soll der Fortschrittsbalken automatisch auf den identischen Wert eingestellt werden (Abb. 8.11).

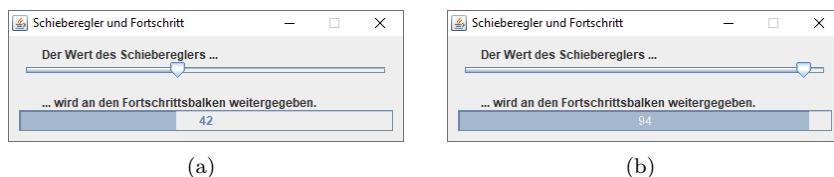


Abbildung 8.11: Beispiel-Ausgaben zu Aufgabe 104

8.5 Fragen

Frage 68. Welche Aufgabe hat der Layout-Manager?

Frage 69. Wann wird typischerweise die Methode `paintComponent()` eines Objektes vom Typ `JPanel` aufgerufen? Welche Aufgabe hat der Parameter vom Typ `Graphics`?

Frage 70. Erläutern Sie den Zweck des Observer Patterns.

Frage 71. Erläutern Sie den zeitlichen Ablauf der Kommunikation zwischen den beteiligten Objekten im Observer Pattern.

Praktikumsaufgaben (Casino, ohne Aufgabenteil zum Multi-Threading)

Bearbeiten Sie nun Praktikum 3 auf Seite 97.

- ▶ Stellen Sie sicher, dass Sie das Praktikum lösen können, bevor Sie fortfahren.
- ▶ Es ist sinnvoll auch die weiteren Versuche zum Praktikum 3 zu bearbeiten.

Praktikumsaufgaben (Spiel des Lebens)

Bearbeiten Sie nun Praktikum 3 auf Seite 118.

- ▶ Stellen Sie sicher, dass Sie das Praktikum lösen können, bevor Sie fortfahren.
- ▶ Es ist sinnvoll auch die weiteren Versuche zum Praktikum 3 zu bearbeiten.

Kapitel 9

Anwendungsbeispiel: Audioverarbeitung

In Kapitel 5 auf Seite 35 haben wir uns als Anwendungsfall mit Grundlagen der Bildverarbeitung beschäftigt. Nun, da Sie Programme mit grafischen Benutzeroberflächen erstellen können, wollen wir ein weiteres Thema aus dem Bereich Signalverarbeitung betrachten.

Die nachfolgenden Darstellungen sind vereinfacht und erheben nicht den Anspruch, Lehrinhalt über Signale und Systeme zu sein. Dennoch hoffe ich, Ihnen hiermit aufzuzeigen, dass sich insbesondere auf der Theorie zu diskreten Signalen und Systemen recht einfach spannende Anwendungen aufbauen lassen.

9.1 Überblick

In den folgenden Abschnitten wollen wir die in Abbildung 9.1 gezeigte Anwendung inklusive der enthaltenen Signalverarbeitung umsetzen. Zunächst werden wir ein Zeit-diskretes Sinus-förmiges Signal erzeugen und als Funktion über der Zeit darstellen (Abb. 9.1, oben links). In einem zweiten Schritt werden wir mittels *Diskreter Fourier-Transformation (DFT)* die zugehörige Zerlegung in die Frequenzanteile erhalten und als Betrags-Spektrum visualisieren (Abb. 9.1, oben rechts).

Aufbauend auf die Darstellungen im Zeit- und Frequenzbereich soll die Unterdrückung von Störfrequenzen umgesetzt werden. Hierfür erzeugen wir zunächst das Störsignal (Abb. 9.1, zweite Zeile) und addieren es zum ursprünglichen Sinus (Abb. 9.1, dritte Zeile). Zur Rauschreduktion

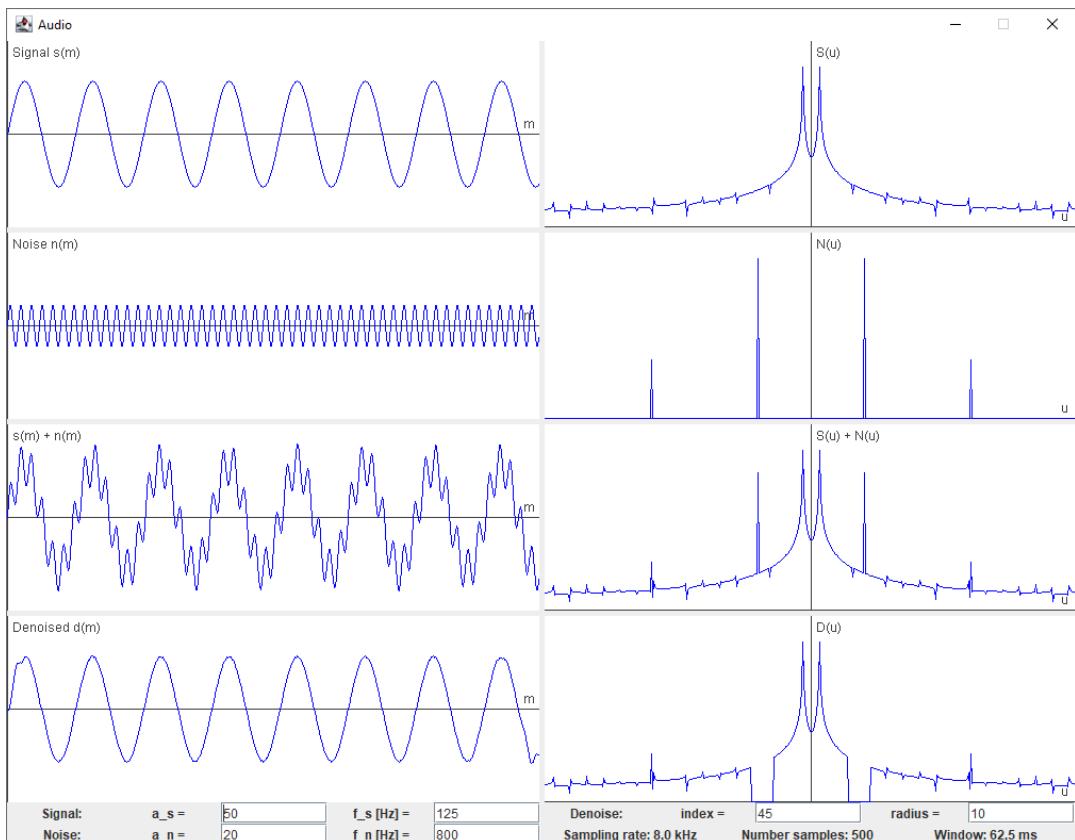


Abbildung 9.1: Beispiel der zu erzeugenden Anwendung

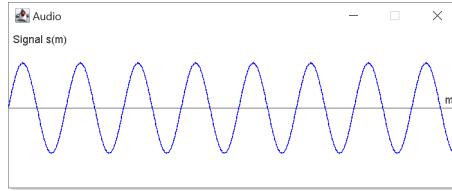


Abbildung 9.2: Signal $s(n)$ mit $a = 50$ und $f = 125$ Hz bei $N = 500$ mit $f_s = 8$ kHz abgetasteten Werten (Aufgabe 105)

werden die beiden ausgeprägten Spitzen des Störsignales im Frequenzbereich entfernt (Abb. 9.1, unten rechts) und aus dem Frequenz- zurück in den Zeitbereich transformiert (Abb. 9.1, unten links). Das resultierende Signal ist zwar nicht identisch mit dem ursprünglichen Sinus-Verlauf, kommt diesem aber sehr nahe.

Letztendlich werden wir Eingabe-Felder zur Anwendung hinzufügen, über die sich die Signale sowie die Parameter der Rauschreduktion verändern lassen. Hierdurch können Sie Signale und Systeme quasi „live“ erleben.

9.2 Audio-Signale im Zeitbereich

Musik, Sprache und andere Geräusche, die wir wahrnehmen, bestehen aus Schwingungen der Luft bzw. des Luftdrucks. Diese lassen sich durch eine Überlagerung von Sinus-förmigen Wellen unterschiedlicher Frequenz f und Amplitude a darstellen bzw. erzeugen¹. Betrachten wir einen Ton, der aus nur einer einzigen Schwingung besteht, so ist der Schalldruck also eine Sinus-Funktion über die Zeit $t \in \mathbb{R}$:

$$s(t) = a \cdot \sin(2\pi f \cdot t) \quad (9.1)$$

Zur Verarbeitung in einem Computer stellen wir Töne als Folge von Zahlenwerten dar. Hierfür diskretisieren wir die Funktion über die Zeit, indem wir in einem festen Zeitabstand $\Delta t = T_s$ die Funktionswerte² $s(t)$ abgreifen. Man spricht hierbei auch von *Abtastung*. Den Zeitabstand T_s zwischen Funktionswerten bezeichnet man als *Abtastintervall* und die zugehörige Frequenz $f_s = \frac{1}{T_s}$ als *Abtastfrequenz*. Bezogen auf unseren Ton in Gleichung (9.1) ergibt sich für $t \geq 0$ eine Folge $s(n)$ mit $n \in \mathbb{N}_0$:

$$s(n) = a \cdot \sin(2\pi f \cdot nT_s) \quad (9.2)$$

$$= a \cdot \sin\left(2\pi \frac{f}{f_s} \cdot n\right) \quad (9.3)$$

Aufgabe 105. (Klasse: AudioApp) Erzeugen Sie ein Programm, das wie in Abbildung 9.2 beispielhaft umgesetzt ein Signal nach Gleichung (9.3) grafisch darstellt. Als Grundlage für die nachfolgenden Aufgaben strukturieren Sie Ihren Quelltext gemäß des Klassendiagramms in Abbildung 9.3:

- Die Klasse *Signal* repräsentiert ein Signal $s(n)$ mit $n = 0, \dots, N - 1$. Über den Konstruktor wird die Anzahl N im Feld *samples* gespeicherter Werte festgelegt.
- Initial sind alle Werte eines *Signal*-Objekts null. Über die Methode *setSamplesSine()* werden die Werte $s(n)$ der abgetasteten Sinus-Funktion nach Gleichung (9.3) in *samples* abgelegt.
- Die Klasse *AudioModel* verwaltet die Audio-Daten der Anwendung. Im Konstruktor wird ein Objekt vom Typ *Signal* erzeugt und mit Sinus-Werten initialisiert.

¹Eine Phasenverschiebung innerhalb der Sinus-Funktion vernachlässigen wir an dieser Stelle.

²Streng genommen werden auch die Werte diskretisiert, was wir im Folgenden jedoch vernachlässigen.

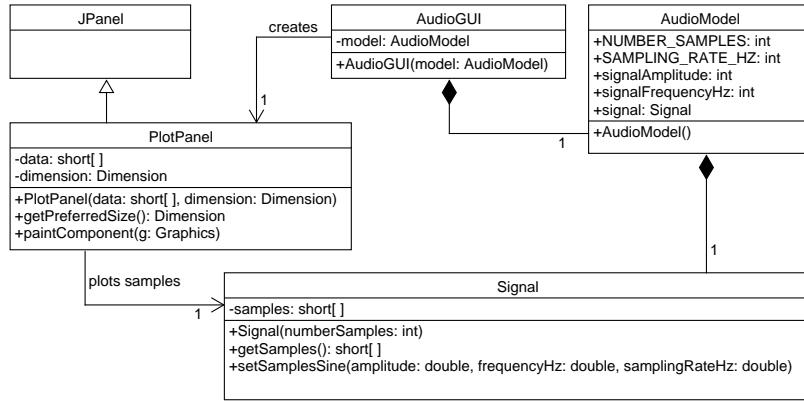


Abbildung 9.3: Klassendiagramm zur Darstellung eines Audio-Signals (Aufgabe 105)

- ▶ Die Klasse *AudioGUI* erzeugt die grafische Oberfläche der Anwendung. Insbesondere beinhaltet diese eine Zeichenfläche vom Typ *PlotPanel*.
- ▶ Die von *JPanel* abgeleitete Klasse *PlotPanel* dient zur grafischen Darstellung eines Signals. Hierfür erhalten Objekte eine Referenz auf das Feld *samples* mit Funktionswerten $s(n)$.
- ▶ Machen Sie sich das Leben nicht unnötig schwer: Wählen Sie die als Breite des Zeichenbereiches die Anzahl N an Abtastwerten.

9.3 Diskrete Fourier-Transformation (DFT)

Signale lassen sich auf unterschiedliche Art und Weise darstellen. Im letzten Abschnitt haben wir den Verlauf über die Zeit betrachtet (*Zeitbereich*). Zugleich sind beispielsweise natürliche Geräusche auch als Überlagerung von Sinus-förmigen Wellen unterschiedlicher Frequenzen darstellbar (*Frequenzbereich*). Wir werden unser Programm daher in diesem Abschnitt derart erweitern, dass wir visualisieren können, wie groß die Anteile der unterschiedlichen Frequenzen in Signalen sind. Hierfür verwenden wir die Fourier-Transformation – genauer: die Diskrete Fourier-Transformation (DFT), da es sich schließlich um Zeit-diskrete Werte handelt.

Transformation Die komplexen Werte bzw. Koeffizienten $S(u) \in \mathbb{C}$ der DFT eines reellen Signals $s(n) \in \mathbb{R}$ sind durch folgende Formel gegeben:

$$s(n) \circlearrowleft S(u) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} s(n) \cdot e^{-2\pi j \cdot \frac{nu}{N}} \quad (9.4)$$

$$= \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} s(n) \cdot \left(\cos \left(2\pi \frac{nu}{N} \right) - j \cdot \sin \left(2\pi \frac{nu}{N} \right) \right) \quad (9.5)$$

Spektrum und Darstellung Da die Koeffizienten $S(u) = \Re\{S(u)\} + j \cdot \Im\{S(u)\}$ komplexe Zahlen sind, bleibt die Frage wie diese grafisch darzustellen sind. Sind lediglich die Stärken der Frequenzanteile von Interesse, so bietet sich das Spektrum in Form der Beträge an:

$$P(u) = |S(u)| = \sqrt{\Re\{S(u)\}^2 + \Im\{S(u)\}^2} \quad (9.6)$$

Bei der Darstellung sind zwei Aspekte zu berücksichtigen:

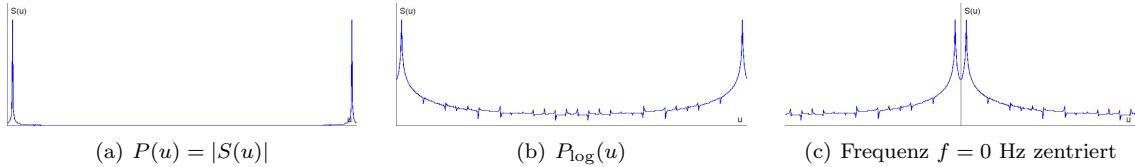


Abbildung 9.4: Darstellung eines Spektrums

- Spektren $P(u)$ weisen in der Regel bei wenigen Frequenzen (oftmals $f = 0 \text{ Hz}$) einen sehr großen Anteil auf (Abb. 9.4a). Zur Erhöhung der Sichtbarkeit niedrigerer Werte wird daher oftmals der duale Logarithmus auf das Spektrum angewendet (Abb. 9.4b). Hierbei ist der mathematisch nicht definierte Ausdruck $\log_2(0)$ zu vermeiden, was durch Addition von Eins geschieht:

$$P_{\log}(u) = c \cdot \log_2(P(u) + 1) \quad \text{mit} \quad c \in \mathbb{R} \quad (9.7)$$

- Spektren sind symmetrisch mit der höchsten Frequenz $f_{\max} = \frac{f_s}{2}$ in der Mitte sowie der Frequenz $f = 0 \text{ Hz}$ bei $u = 0$ und $u = N - 1$. Üblich ist eine Darstellung mit $f = 0 \text{ Hz}$ in der Mitte, das heißt Spektren werden für die Darstellung um die halbe Breite $\frac{N}{2}$ verschoben (Abb. 9.4c):

$$\tilde{P}(u) = c \cdot \log_2 (P(\tilde{u}) + 1) \quad \text{mit} \quad \tilde{u} = \left(u + \frac{N}{2} \right) \bmod N \quad (9.8)$$

■ **Aufgabe 106.** (Klasse: AudioApp) Erweitern Sie Ihre Lösung um die Darstellung des zum Signal gehörenden Spektrums (Abb. 9.5). Strukturieren Sie Ihren Quelltext gemäß des Klassendiagramms in Abbildung 9.6:

- Die Klasse *Complex* repräsentiert die für die DFT benötigten komplexen Zahlen $z = x + jy \in \mathbb{C}$ mit Realteil $\Re\{z\} = x$ und Imaginärteil $\Im\{z\} = y$. Je nach Umsetzung können Ihre Methoden von den abgebildeten abweichen.
- Die Klasse *Fourier* stellt eine statische Methode *dft()* zur Berechnung der DFT-Koeffizienten eines Feldes reeller Zahlen gemäß Gleichung (9.5) zur Verfügung. Die statische Methode *getSpectrumLog2()* bestimmt zu DFT-Koeffizienten das logarithmierte Betrags-Spektrum gemäß Gleichung (9.8). Das Spektrum wird über $c \in \mathbb{R}$ derart skaliert, dass der maximale Wert dem übergebenen Argument entspricht.
- Die Klasse *Signal* wird unter anderem um Felder für die DFT-Koeffizienten sowie das Spektrum erweitert. Die Methode *updateFrequencyDomain()* aktualisiert diese Felder mittels der Werte im Zeitbereich (d. h. in *samples*).

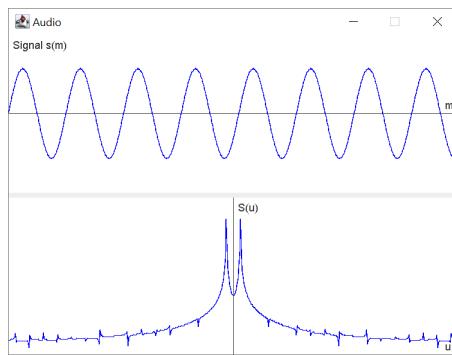


Abbildung 9.5: Darstellung im Zeit- und Frequenzbereich (Aufgabe 106)

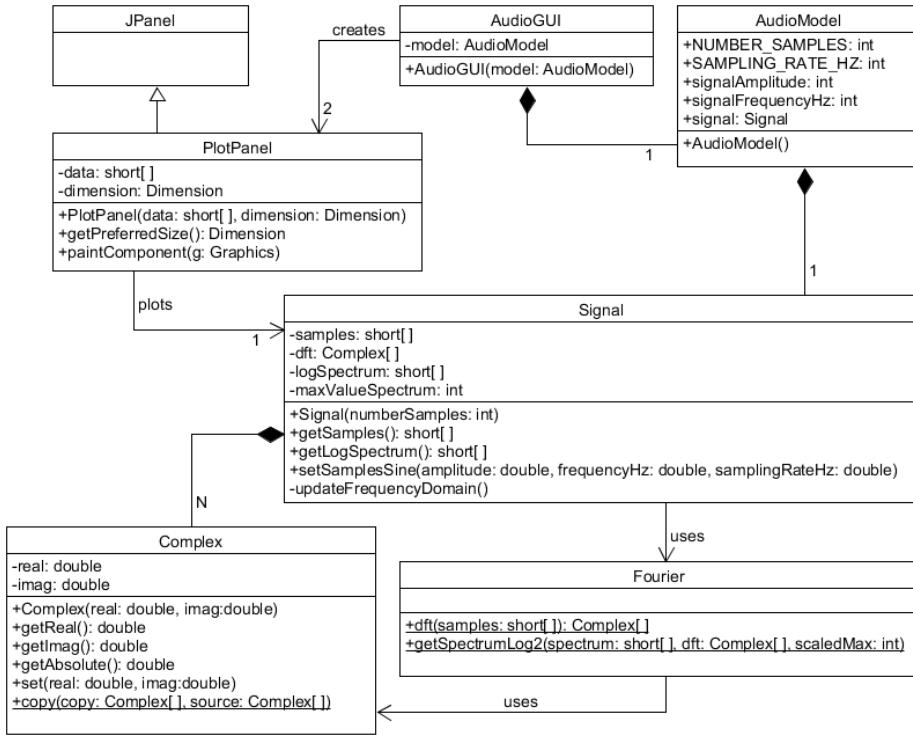


Abbildung 9.6: Klassendiagramm bei Erweiterung um den Frequenzbereich (Aufgabe 106)

- Die Klasse *AudioGUI* erzeugt zur Darstellung des Spektrums ein zweites Objekt vom Typ *PlotPanel*.

Reduktion der Rechenzeit Die DFT ist eine vergleichsweise rechenintensive Abbildung. Zwar existiert mit der *FFT* (*Fast Fourier Transformation*) eine schnelle Variante. Diese ist aber nicht trivial zu verstehen. Daher begnügen wir uns an dieser Stelle mit einer einfacher nachzuvollziehenden Reduktion des Rechenaufwandes. Diese basiert darauf, dass sowohl die Sinus- als auch die Kosinus-Funktion periodisch in 2π sind, d. h. es gilt

$$\cos(\alpha) = \cos(\alpha + 2\pi \cdot k) \quad (9.9)$$

$$\sin(\alpha) = \sin(\alpha + 2\pi \cdot k) \quad (9.10)$$

für beliebige ganze Zahlen $k \in \mathbb{Z}$. In Gleichung (9.5) treten diese Funktionen sehr häufig auf. Bei genauer Betrachtung werden sie jedoch nur für N unterschiedliche Argumente benötigt. Hierbei nutzen wir die Division mit Rest sowie die Periodizität in 2π :

$$\cos\left(2\pi \frac{nu}{N}\right) = \cos\left(2\pi \cdot \underbrace{\left(\left\lfloor \frac{nu}{N} \right\rfloor + \frac{(nu) \bmod N}{N}\right)}_{k \in \mathbb{N}_0 \text{ Rest}/N} \right) = \cos\left(\underbrace{2\pi k}_{\text{Periode}} + 2\pi \cdot \frac{(nu) \bmod N}{N}\right) \quad (9.11)$$

Es folgt also

$$\cos\left(2\pi \frac{nu}{N}\right) = \cos\left(2\pi \frac{(nu) \bmod N}{N}\right) \quad (9.12)$$

$$\sin\left(2\pi \frac{nu}{N}\right) = \sin\left(2\pi \frac{(nu) \bmod N}{N}\right) \quad (9.13)$$

$$(9.14)$$

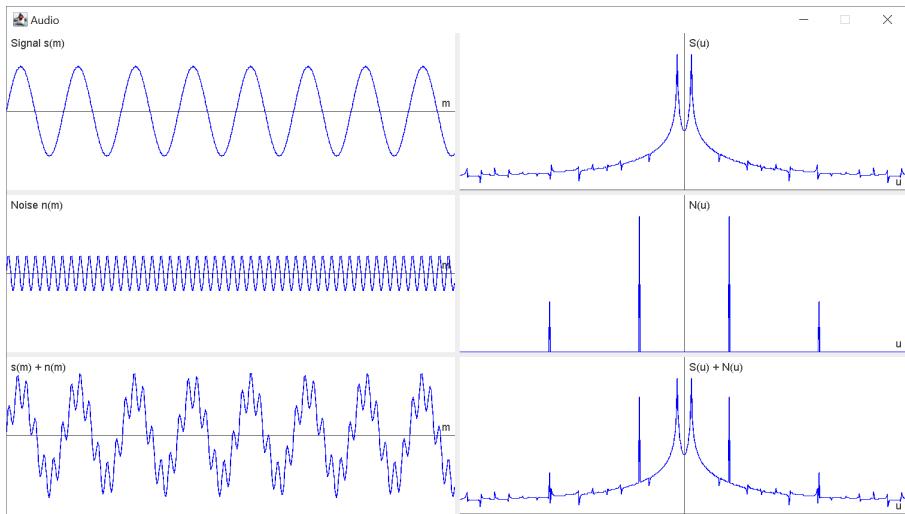


Abbildung 9.7: Hinzufügen eines Störsignals (Aufgabe 108)

und wegen $(nu) \bmod N = 0, 1, 2, \dots, N-1$ sind die trigonometrischen Funktionen in Gleichung (9.5) lediglich für N unterschiedliche Argumente zu berechnen.

■ **Aufgabe 107.** (Klasse: AudioApp) Reduzieren Sie die Rechenzeit Ihres Programms durch Einführung von sogenannten *LUT* (*Lookup table*):

- ▶ Ergänzen Sie die Klasse *Fourier* um statische Felder *cosLUT* und *sinLUT* vom Typ *double*.
- ▶ Berechnen Sie vor Durchführung einer Transformation die Sinus- und Kosinus-Werte von $\frac{2\pi k}{N}$, $k = 0, 1, 2, \dots, N$, und speichern Sie diese in den Feldern *cosLUT* und *sinLUT*.
- ▶ Verwenden Sie in der Berechnung einer Transformation die zuvor berechneten Sinus- und Kosinus-Werte (z. B. mittels $k = (u * n) \% \text{size}$ sowie *cosLUT[k]*).

Störsignal Wenn Sie ein Sinus-Signal im Zeit- und Frequenzbereich erzeugen und darstellen können, klappt das auch mit mehreren. Hierbei hilft uns unsere Architektur, da wir für jedes weitere Signal lediglich ein Objekt vom Typ *Signal* zur Repräsentation der Daten sowie zwei Objekte vom Typ *PlotPanel* zur Darstellung des Zeit-Signals sowie des zugehörigen Spektrums.

■ **Aufgabe 108.** (Klasse: AudioApp) Erweitern Sie Ihr Programm durch folgende Signale (Abb. 9.7):

- ▶ Sinus-förmiges Störsignal $\eta(n)$ („Rauschen“ bzw. *noise*) mit $a = 20$ und $f = 800$ Hz
- ▶ Durch additives Rauschen gestörtes Signal $s_\eta(n) = s(n) + \eta(n)$

9.4 Filterung im Frequenzbereich

Ja, dieses Kapitel ist anspruchsvoll, aber so ist auch das „echte Leben“ – und Sie haben es gleich geschafft. Immerhin haben wir unser Signal s durch das Störsignal η kaputt gemacht und daher müssen wir es auch wieder reparieren³. Hierzu werden wir wie folgt vorgehen:

1. Entfernen von Frequenzen in $S_\eta(u)$, die vor allem durch η hinzugekommen sind
2. Rück-Transformation aus dem Frequenz- in den Zeitbereich

³Ordnung muss sein!

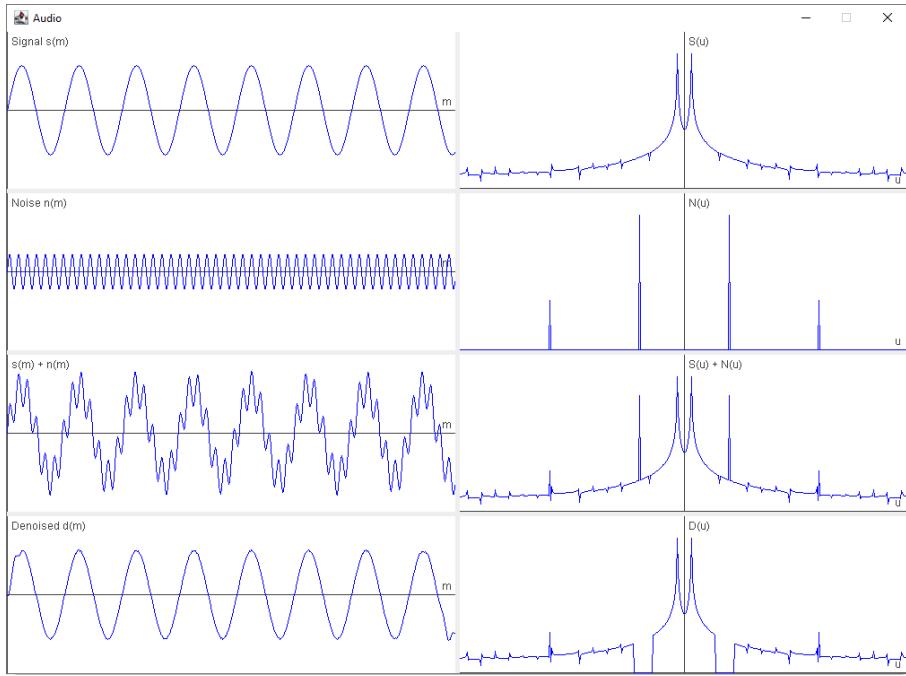


Abbildung 9.8: Hinzufügen eines Störsignals (Aufgabe 109)

j

Filterung Bei einem Filter, das Frequenzanteile innerhalb eines bestimmten Bereiches $[f_{\min}, f_{\max}]$ mit $f_{\min} > 0$ Hz und $f_{\max} < \infty$ entfernt, spricht man auch von einem *Bandsperrfilter* (*Bandreject filter*). „Entfernen“ bedeutet hierbei, dass die zugehörigen DFT-Koeffizienten zu null gesetzt werden.

Rück-Transformation Zu gegebenen komplexen Fourier-Koeffizienten $S(u)$ erhält man das zugehörige Zeitsignal durch die *Inverse DFT* (*IDFT*):

$$S(u) \xrightarrow{\bullet \circ} s(n) = \frac{1}{\sqrt{N}} \sum_{u=0}^{N-1} S(u) \cdot e^{2\pi j \cdot \frac{nu}{N}} \quad (9.15)$$

$$= \frac{1}{\sqrt{N}} \sum_{u=0}^{N-1} S(u) \cdot \left(\cos \left(2\pi \frac{nu}{N} \right) + j \cdot \sin \left(2\pi \frac{nu}{N} \right) \right) \quad (9.16)$$

■ **Aufgabe 109.** (Klasse: AudioApp) Fügen Sie zum dem Programm aus Aufgabe 108 eine Reduktion des Störsignals η im Frequenzbereich hinzu (Abb. 9.8):

1. Für das durch Rauschen überlagerte Signal wird allen Fourier-Koeffizienten $S_\eta(u)$, die von der Frequenz $f = 0$ Hz einen Abstand $\Delta u = 45 \pm 10$ besitzen, der Wert null zugewiesen.
2. Transformieren Sie anschließend das gefilterte Signal aus dem Frequenz- in den Zeitbereich.
3. Stellen Sie das gefilterte Signal sowohl im Frequenzbereich als auch im Zeitbereich dar.

■ **Aufgabe 110.** (Klasse: AudioApp) Erweitern Sie Ihr Programm abschließend um die Möglichkeit, Amplituden und Frequenzen der Sinus-förmigen Signale sowie den Bereich des Bandsperrfilters interaktiv zu verändern (Abb. 9.1 auf Seite 61).

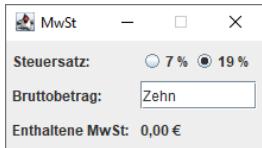
Kapitel 10

Ausnahmen

10.1 Ausnahmen fangen

■ **Aufgabe 111.** (Klasse: VATApp) In Aufgabe 99 auf Seite 57 war eine Anwendung zur Berechnung der in einem Betrag enthaltenen Mehrwertsteuer zu erstellen. Hierbei sind von Benutzern als Zeichenkette eingegebene Zahlen in Fließkommawerte zu wandeln. Bislang haben wir jedoch keine Fehler behandelt, die auftreten, wenn beispielsweise Buchstaben anstatt Zahlen eingegeben werden.

- ▶ Implementieren Sie, sofern noch nicht geschehen, die Umwandlung von *String* nach *double* mittels der Klassenmethode *Double.parseDouble()*.
- ▶ Fangen und behandeln Sie hierbei potentiell auftretende Ausnahmen (Abb. 10.1).



(a) Eingabe ist keine Zahl



(b) Dialog-Fenster im Fehlerfall

Abbildung 10.1: Fangen und Behandeln von fehlerhaften Eingaben (Aufgabe 111)

Dialog-Fenster

Über die Methode *showMessageDialog()* der Klasse *JOptionPane* lassen sich von Anwendern zu bestätigende Dialog-Fenster anzeigen. Diese können unter anderem mit Icons für Hinweise, Warnungen und Fehler versehen werden.

10.2 Eigene Ausnahme-Klassen

■ **Aufgabe 112.** (Klasse: MatrixApp) In Aufgabe 62 auf Seite 31 war die Multiplikation $\mathbf{A} \cdot \mathbf{B}$ zweier Matrizen umzusetzen. Diese ist nur definiert, sofern die Anzahl der Spalten der linken Matrix der Anzahl Zeilen der rechten Matrix entspricht, d. h. $\mathbf{A} \in \mathbb{R}^{m \times n}$ und $\mathbf{B} \in \mathbb{R}^{n \times p}$ für $m, n, p \in \mathbb{N}$. Modifizieren Sie die Lösung zu Aufgabe 62 wie folgt (Abb. 10.2):

- ▶ Die Methode *multiply()* der Klasse *Matrix* wirft eine Ausnahme des zu erstellenden Typs *LinearAlgebraException*, falls die Dimensionen der zu übergebenen Matrizen nicht passen.
- ▶ Ausnahmen des Typs *LinearAlgebraException* müssen nicht zwingend gefangen oder mittels *throws* an die aufrufende Methode weitergereicht werden.
- ▶ Geworfene Ausnahmen des Typs *LinearAlgebraException* werden in *main()* gefangen.

Beispieldausgabe:

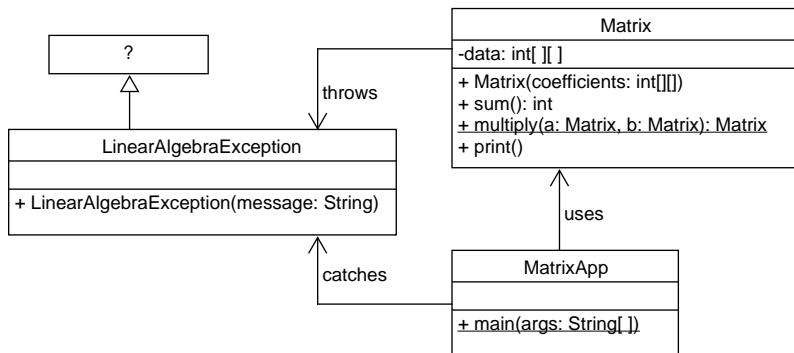


Abbildung 10.2: Klassendiagramm mit eigener Ausnahme-Klasse (Aufgabe 112)

```

Matrix A:
2   3
4   -1

Matrix B:
2   -1   1
3   2   4

Multiplikation AB:
13   4   14
5   -6   0

Multiplikation BA:
WARNING: Dimensions do not match in matrix multiplication: 3 != 2
  
```

10.3 Fragen

Frage 72. Was passiert, wenn eine Ausnahme geworfen wird und es weder in der aktuellen Methode, noch in einer der aufrufenden Methoden einen passenden *catch*-Block gibt?

Frage 73. In welcher Reihenfolge müssen die *try/catch/finally*-Blöcke stehen? Wie oft müssen die Blöcke mindestens vorhanden sein? Wie oft dürfen sie höchstens vorhanden sein?

Wahr oder falsch?

Frage 74. Ausnahmen können einen beliebigen Datentyp haben.

Frage 75. Abgesehen vom Datentyp der Ausnahme können keine weiteren Informationen über die Ausnahme weitergereicht werden.

Frage 76. Der Stack-Trace kann nur ausgegeben werden, wenn sich ein Programm mit einer Ausnahme beendet.

Frage 77. Die Reihenfolge von *catch*-Blöcken und *finally*-Blöcken ist beliebig.

Frage 78. Der *finally*-Block wird auch durchlaufen, wenn keine Ausnahme geworfen wurde.

Frage 79. Innerhalb eines *finally*-Blocks dürfen keine weiteren Ausnahmen geworfen werden.

Frage 80. Mehrere *try/catch/finally*-Blöcke können nicht geschachtelt werden.

Frage 81. In einer Methoden-Deklaration müssen alle möglichen Ausnahmen aufgeführt werden.

Frage 82. In einer Methoden-Deklaration müssen alle Ausnahmen, die innerhalb der Methode gefangen werden, aufgeführt werden.

Praktikumsaufgaben (Geografische Koordinaten)

Bearbeiten Sie nun Praktikum 3 auf Seite 111.

- ▶ Stellen Sie sicher, dass Sie das Praktikum lösen können, bevor Sie fortfahren.
- ▶ Es ist sinnvoll auch die weiteren Versuche zum Praktikum 3 zu bearbeiten.

Kapitel 11

Eingabe und Ausgabe

11.1 Tastatureingabe und Textdateien schreiben

■ **Aufgabe 113.** (Klasse: LogInputApp) Erstellen Sie ein Programm, das über die Tastatur eingegebene Zeilen in einer Log-Datei speichert. Das Programm endet, sobald „Stop“ eingegeben wurde. Beispiel:

```
Enter text to write to file ("stop" to stop input):
No one said, it would be easy.
Indeed ...
stop
Written to file: Written to file: _09_1_IO_Text/LogInput.txt
```

11.2 Text-Dateien lesen

■ **Aufgabe 114.** (Klasse: FindWordApp) Erstellen Sie eine Methode, die eine Textdatei nach einem als Argument übergegebenen Wort durchsucht. Alle Zeilen, in denen das Wort gefunden wurde, sollen als Liste zurückgegeben werden und auf Konsole ausgegeben werden. Als Beispiel sollte die Suche nach dem Wort „mit“ in dem Gedicht „Dunkel war's, der Mond schien helle“¹ zu der nachfolgenden bzw. einer ähnlichen Ausgabe führen:

```
Word "mit" found 2 times.
1: mit kohlenschwarzem Haar
2: die mit Schmalz bestrichen war.
```

■ **Aufgabe 115.** (Klasse: SudokuApp) Erstellen Sie ein Programm, das in einer grafischen Benutzeroberfläche die Werte eines Sudoku-Spiels der Größe 9×9 darstellt. Die initialen Werte werden zu Beginn des Spieles aus einer Textdatei eingelesen, wobei jede Textzeile die durch Kommata getrennten Zahlen eines Blocks der Größe 3×3 beinhaltet. Als Beispiel erzeugt die Initialisierung mit den nachfolgenden Zeilen das in Abbildung 11.1 dargestellte Spiel.

```
,,3,,6,,2,,,
,,1,,,,8,,
,,,,3,,,,9,,,
,,,,4,,,,5,,,
1,,,,2,,,
,,,,,,7
,,6,,7,,,
,,9,,4,,,
,,,,6,,2,,,
```

11.3 Fragen

Frage 83. Was ist der wesentliche Unterschied zwischen Eingabeströmen der Klassen *InputStream* und *Reader*?

Frage 84. Was wird durch ein Objekt vom Typ *File* repräsentiert?

Frage 85. Durch Objekte welcher Klasse werden Verzeichnisse repräsentiert?

Frage 86. Ist folgende Aussage korrekt? „Sowohl beim Lesen (z. B. über *FileReader*) als auch beim Schreiben (z. B. über *FileWriter*) von Daten können diese wahlweise gepuffert werden.“

¹https://de.wikipedia.org/wiki/Dunkel_war's,_der_Mond_schien_helle, abgerufen am 14.05.2021

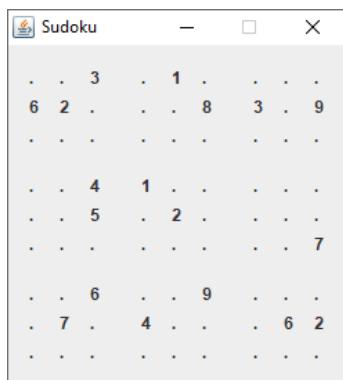


Abbildung 11.1: Beispiel eines initialisierten Sudokus nach Aufgabe 115

Frage 87. Kann man auf die Daten eines Streams wahlfrei zugreifen (z. B. über einen Index für einen *FileStream* beliebig die Position des nächsten Zugriffs setzen)?

Praktikumsaufgaben (Geografische Koordinaten)

Bearbeiten Sie nun Praktikum 4 auf Seite 114.

- ▶ Stellen Sie sicher, dass Sie das Praktikum lösen können, bevor Sie fortfahren.
 - ▶ Es ist sinnvoll auch die weiteren Versuche zum Praktikum 4 zu bearbeiten.

Kapitel 12

Anwendungsbeispiel: Audiodateien erzeugen

In den folgenden Abschnitten werden wir unser bisheriges Wissen über Audiosignale (Kapitel 9 auf Seite 61) und Frequenzen musikalischer Noten (Aufgabe 40 auf Seite 23) mit dem Schreiben von Dateien (Kapitel 11) verknüpfen, um Audiodateien im WAVE-PCM-Format zu erzeugen. Hierzu betrachten wir zunächst den Aufbau des Dateiformats.

12.1 WAVE-PCM-Dateiformat

Es gibt unterschiedliche Ausprägungen des WAVE-Dateiformates¹. Für unsere Zwecke bieten sich Dateien mit einem Kanal („Mono“) an, deren Signalwerte jeweils durch zwei Byte repräsentiert werden. In Java entspricht dies Ganzzahlen vom Datentyp *short* mit Werten in $[-2^{15}, 2^{15} - 1] \subset \mathbb{Z}$ bzw. $[-32.786, 32.767]$. Die Dateien bestehen aus einleitenden Kopfdaten (*Header*) mit Metadaten (z. B. Anzahl Kanäle) gefolgt von den eigentlichen Nutzdaten.

12.1.1 Kopfdaten (Header)

Um Ihnen das Leben nicht unnötig zu erschweren, stelle ich Ihnen an dieser Stelle nachfolgende Beispiel-Implementierung zum Schreiben des Headers zur Verfügung. Der Methode sind ein mit der zu schreibenden Datei verknüpfter Byte-Strom vom Datentyp *OutputStream*, die Abtastfrequenz sowie die Anzahl Bytes der Audiodaten zu übergeben. Denken Sie beim Aufruf der Methode daran, dass jeder Audiowert aus *zwei* Bytes besteht.

```
/** Write wave file header to file.
 *
 * @param fileStream Stream to write header to
 * @param samplingRateHz Sampling rate in Hz
 * @param numberDataBytes Number of bytes of audio data
 * @throws IOException
 */
private static void writeFileHeader(OutputStream fileStream,
    int samplingRateHz, long numberDataBytes) throws IOException {
    int audioFormat = 1;
    int audioChannels = 1;
    int bytesPerSample = 2;    // Samples in audio data are of type short => 2 bytes
    int bitsPerSample = 16;
    int byteRate = bytesPerSample * samplingRateHz;
    long totalDataLength = numberDataBytes + 36;

    // Write 44 byte header
    writeUTF8(fileStream, "RIFF");           // Chunk ID
    writeUInt32(fileStream, totalDataLength);
    writeUTF8(fileStream, "WAVE");            // WAVE format

    // FMT subchunk
    writeUTF8(fileStream, "fmt ");           // Subchunk 1 ID
    writeUInt32(fileStream, 16);              // Subchunk 1 size
    writeUInt16(fileStream, audioFormat);
    writeUInt16(fileStream, audioChannels);
    writeUInt32(fileStream, samplingRateHz);
    writeUInt32(fileStream, byteRate);
    writeUInt16(fileStream, audioChannels * bytesPerSample); // Block align
    writeUInt16(fileStream, bitsPerSample);

    // Data subchunk
```

¹https://de.wikipedia.org/wiki/RIFF_WAVE (Besucht am 19.05.2021)

```

    writeUTF8(fileStream, "data");           // Subchunk 2 ID
    writeUInt32(fileStream, numberDataBytes); // Subchunk 2 size
}

```

Zum Schreiben der Daten werden folgende Hilfsmethoden aufgerufen:

- Zeichenketten (1 Byte pro Zeichen):

```

/** Write a string as 1-byte chars to the file.
 *
 * @param fileStream Stream to write to
 * @param chars Characters to write to the file
 * @throws IOException
 */
private static void writeUTF8(FileOutputStream fileStream, String chars)
    throws IOException {
    fileStream.write(chars.getBytes("UTF-8"));
}

```

- Ganzzahlen (2 Byte bzw. 4 Byte):

```

/** Write a 2-byte unsigned integer to the file.
 *
 * Byte order is reversed, i.e., the least significant byte is written first.
 *
 * @param fileStream Stream to write to
 * @param value Value to write
 * @throws IOException
 */
private static void writeUInt16(FileOutputStream fileStream, int value)
    throws IOException {
    fileStream.write(new byte[] {
        (byte) (value & 0xff),
        (byte) ((value >> 8) & 0xff)
    });
}

/** Write a 4-byte unsigned integer to the file.
 *
 * Byte order is reversed, i.e., the least significant byte is written first.
 *
 * @param fileStream Stream to write to
 * @param value Integer value to write (4-byte, unsigned)
 * @throws IOException
 */
private static void writeUInt32(FileOutputStream fileStream, long value)
    throws IOException {
    fileStream.write(new byte[] {
        (byte) (value & 0xff),
        (byte) ((value >> 8) & 0xff),
        (byte) ((value >> 16) & 0xff),
        (byte) ((value >> 24) & 0xff)
    });
}

```

12.1.2 Nutzdaten

Auf den Header folgen die Nutzdaten im 16-Bit PCM-Format² (*Pulse Code Modulation*). Dies entspricht Rohdaten vom Datentyp *short*, wie wir Sie bereits in der Klasse *Signal* in Aufgabe 105 auf Seite 62 erzeugt und verwendet haben. Sie können die Signalwerte beispielsweise nacheinander über die oben angegebene Methode *writeUInt16()* in die Datei schreiben. Denken Sie daran, den Dateistrom nach dem Schreiben aller Daten wieder zu schließen.

²<https://de.wikipedia.org/wiki/Puls-Code-Modulation> (Besucht am 19.05.2021)

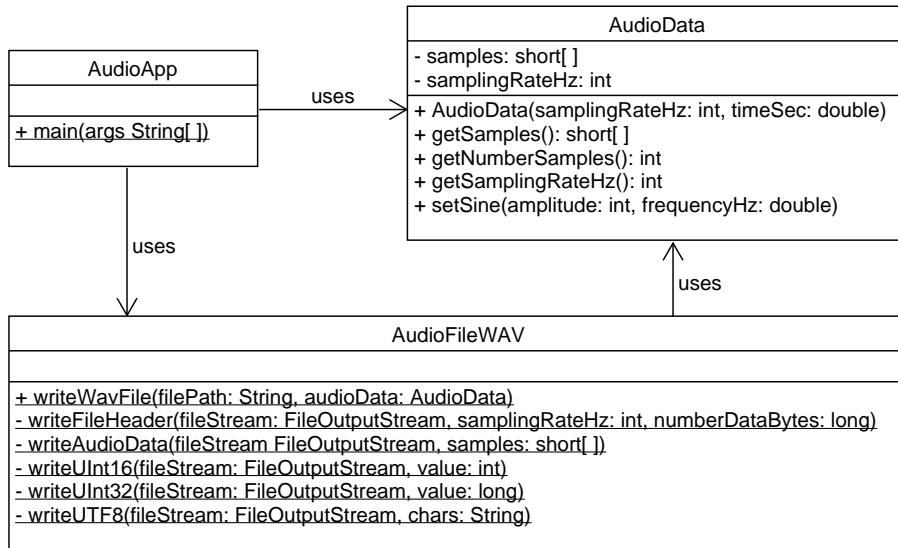


Abbildung 12.1: Klassendiagramm zu Aufgabe 116

12.2 Töne und Akkorde

Nun wollen wir aber etwas hören. Also Boxen an und aufgedreht!

■ **Aufgabe 116.** (Klasse: `AudioApp`) Schreiben Sie ein Programm, das eine Audiodatei erzeugt, die einige Sekunden lang den Kammerton A4 (440 Hz) abspielt. Gehen Sie hierbei gemäß des Klassendiagramms in Abbildung 12.1 vor:

- ▶ Die Klasse `AudioData` entspricht weitestgehend der aus Aufgabe 105 auf Seite 62 bekannten Klasse `Signal`. Sie enthält jedoch lediglich Daten im Zeitbereich, nicht im Frequenzbereich. Zudem wird die Anzahl der Audiowerte nicht explizit übergeben, sondern mittels der Abtastrate sowie der Dauer des Signals in Sekunden bestimmt.
- ▶ Die Klasse `AudioFileWAV` stellt eine statische Methode `writeWavFile()` zum Schreiben einer Audiodatei im WAVE-PCM-Format (*.wav) zur Verfügung. Die privaten Hilfsmethoden sind in Abschnitt 12.1 angegeben bzw. erläutert.

■ **Aufgabe 117.** (Klasse: `ChordApp`) Modifizieren Sie das Programm aus Aufgabe 116 derart, dass ein A-Moll-Akkord bestehend aus den Tönen A4, E4, A5 und C5 erklingt (Abb. 12.2):

- ▶ Erweitern Sie die Klasse `AudioData` um eine Methode `add()`, die zu dem im Objekt enthaltenen Signalwerten die Werte eines zweiten `AudioData`-Objektes addiert.
- ▶ Für den Fall, dass die Längen und/oder Abtastraten von zu addierenden Signalen nicht übereinstimmen, ist eine Ausnahme vom Typ `AudioDataException` zu werfen. Der Ausnahmetyp soll nicht zwingend behandelt oder mittels `throws` deklariert werden.
- ▶ Nutzen Sie die Methode `tone2FrequencyHz()` der Klasse `MusicNote` aus Aufgabe 40 auf Seite 23, um die Frequenzen der jeweiligen Töne zu erhalten.

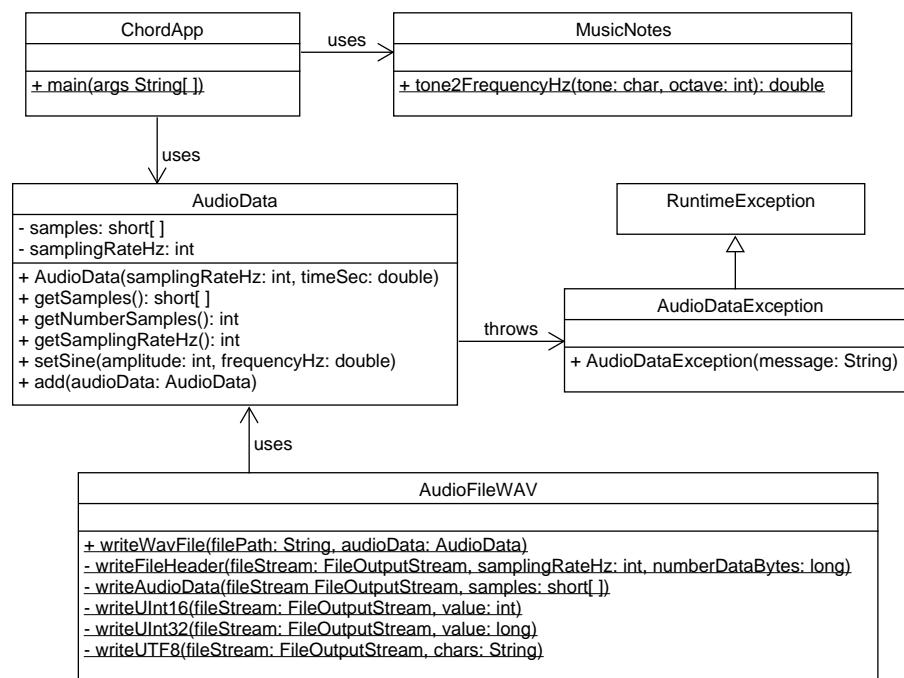


Abbildung 12.2: Klassendiagramm zu Aufgabe 117

Kapitel 13

Parallelverarbeitung mit Threads

13.1 Primzahlen

■ **Aufgabe 118.** (Klasse: PrimeApp) Erstellen Sie als Vorbereitung für nachfolgende Aufgaben eine Klasse *PrimeNumbers* mit einer statischen Methode *isPrime()*. Über die Methode kann wie nachfolgend beschrieben ermittelt werden, ob eine übergebene Zahl n vom Datentyp *long* eine Primzahl ist (Rückgabe *true*) oder nicht (Rückgabe *false*):

1. Kleine Zahlen: Für $n \leq 1$ handelt es sich nicht um eine Primzahl.
2. Gerade Zahlen: Ist n durch 2 teilbar, so handelt es sich für $n = 2$ um eine Primzahl, für $n \geq 4$ jedoch nicht.
3. Ungerade Teiler: Durchlaufen Sie die alle ungeraden Zahlen von 3 bis $\lfloor \frac{n}{2} \rfloor$. Teilt eine der Zahlen n ganzzahlig, so handelt es sich bei n nicht um eine Primzahl.
4. Wurde die Methode zuvor noch nicht verlassen, so ist n nur durch 1 und sich selber teilbar und es handelt sich um eine Primzahl.

Beispielausgabe:

```
0: not a prime number
1: not a prime number
2: prime number
3: prime number
4: not a prime number
21: not a prime number
29: prime number
```

■ **Aufgabe 119.** (Klasse: PrimeRangeApp) Erweitern Sie die Klasse *PrimeNumbers* aus Aufgabe 118 um einem Konstruktor, dem eine Zahl N vom Datentyp *long* übergeben wird. Der Konstruktor bestimmt alle Primzahlen im Intervall $[1, N]$. Implementieren Sie eine Getter-Methode *getPrimes()*, über die die ermittelten Primzahlen abgefragt werden können und überprüfen Sie die Ergebnisse durch ein ausführbares Programm. Beispielausgabe für $N = 250$:

2	3	5	7	11	13	17	19	23	29	31	37	41	43	47
53	59	61	67	71	73	79	83	89	97	101	103	107	109	113
127	131	137	139	149	151	157	163	167	173	179	181	191	193	197
199	211	223	227	229	233	239	241							

■ **Aufgabe 120.** (Klasse: PrimeThreadApp) Erweitern Sie Ihre Lösung für Aufgabe 119 derart, dass die Klasse *PrimeNumbers* Threads verwendet, um unterschiedliche Teil-Intervalle parallel auf Primzahlen zu untersuchen:

- ▶ Erstellen Sie eine Klasse *PrimeNumberThread* und überlagern Sie die *run()*-Methode derart, dass diese alle Primzahlen in einem Intervall $[n_0, n_1]$ bestimmt. Die ermittelten Primzahlen werden in einer Liste gespeichert und lassen sich auch nach Terminierung der parallelen Ausführung noch über einen entsprechenden Getter abrufen.
- ▶ Die Klasse *PrimeNumbers* besitzt einen Konstruktor, dem als zusätzliches Argument die Anzahl der zu verwendenden Threads übergeben wird. Dieser teilt das Intervall $[1, N]$ in entsprechend viele Subintervalle und initialisiert sowie startet für die Untersuchung jedes dieser Intervalle einen eigenen Thread. Anschließend wartet der Konstruktor bis alle gestarteten Threads terminiert sind und führt die Teilergebnisse in einer Liste zusammen.

- Das ausführbare Programm misst die zur Initialisierung der Objekte vom Typ *PrimeNumbers* benötigte Zeit und gibt diese mit der Präzision Millisekunde auf Konsole aus. Verwenden Sie zur Messung den Aufruf `System.nanoTime()`.

Beispieldaten:

```
Find prime numbers up to N = 250,000 using 1 thread(s).
Time taken: 4.454 s

Find prime numbers up to N = 250,000 using 4 thread(s).
Time taken: 1.998 s

Find prime numbers up to N = 250,000 using 8 thread(s).
Time taken: 1.207 s

Find prime numbers up to N = 250,000 using 16 thread(s).
Time taken: 0.807 s
```

13.2 Stoppuhr

■ **Aufgabe 121.** (Klasse: StopWatchApp) Erstellen Sie ein Programm, das eine Stoppuhr mit grafischer Benutzeroberfläche zur Verfügung stellt:

- Die Uhr kann gestartet und wieder angehalten werden.
- Es werden die seit dem Start vergangenen Minuten und Sekunden angezeigt.
- Nach Anhalten der Uhr kann die Zeit zurückgesetzt werden.
- Nach dem Start synchronisiert sich die Uhr auf ganze Sekunden. Die Wartezeit beträgt also nicht jeweils 1000 ms, sondern ist um die für den „Overhead“ benötigte Zeit (z. B. Prüfen der Abbruchbedingung einer Schleife) reduziert.

Abbildung 13.1 veranschaulicht einige der Anforderungen anhand einer Beispiel-Umsetzung. Ihre Lösung darf jedoch von der dargestellten grafischen Benutzeroberfläche abweichen.



Abbildung 13.1: Beispiel einer Stoppuhr nach Aufgabe 121

13.3 Fragen

Frage 88. Erläutern Sie die „Interface-basierte“ Methode zur Erstellung eines Threads.

Frage 89. Erläutern Sie die „Klassen-basierte“ Methode zur Erstellung eines Threads.

Frage 90. Was passiert, wenn Sie für ein Thread-Objekt die `run()`-Methode aufrufen?

Frage 91. Was ist die Aufgabe des Schedulers innerhalb der JVM?

Frage 92. Ein Thread führt für ein Objekt eine Methode aus, die den Modifier *synchronized* besitzt. Dürfen andere Threads zeitgleich Methoden desselben Objektes ausführen?

Praktikumsaufgaben (Casino, Aufgabenteil zum Multi-Threading)

Bearbeiten Sie nun Praktikum 3 auf Seite 97.

- ▶ Stellen Sie sicher, dass Sie das Praktikum lösen können, bevor Sie fortfahren.
- ▶ Es ist sinnvoll auch die weiteren Versuche zum Praktikum 3 zu bearbeiten.

Praktikumsaufgaben (Spiel des Lebens)

Bearbeiten Sie nun Praktikum 4 auf Seite 121.

- ▶ Stellen Sie sicher, dass Sie das Praktikum lösen können, bevor Sie fortfahren.
- ▶ Es ist sinnvoll auch die weiteren Versuche zum Praktikum 4 zu bearbeiten.

Teil II

Praktika: Casino

Kapitel 14

Praktikum: Einführung

Primäre Lernziele

- Einrichtung von IntelliJ IDEA und eines Projektes für die weiteren Praktika
- Mit der Entwicklungsumgebung und dem Debugger vertraut machen
- Gegebene JUnit-Tests nutzen

14.1 Aktivierung von IntelliJ IDEA (PC-Pool)

Sofern Sie diese Einführung im PC-Pool des Departments Informations- und Elektrotechnik an der HAW Hamburg bearbeiten, muss IntelliJ IDEA mindestens bei erstmaliger Verwendung aktiviert werden. Starten Sie hierzu das Programm und führen Sie die Schritte gemäß Abbildung 14.1 aus.

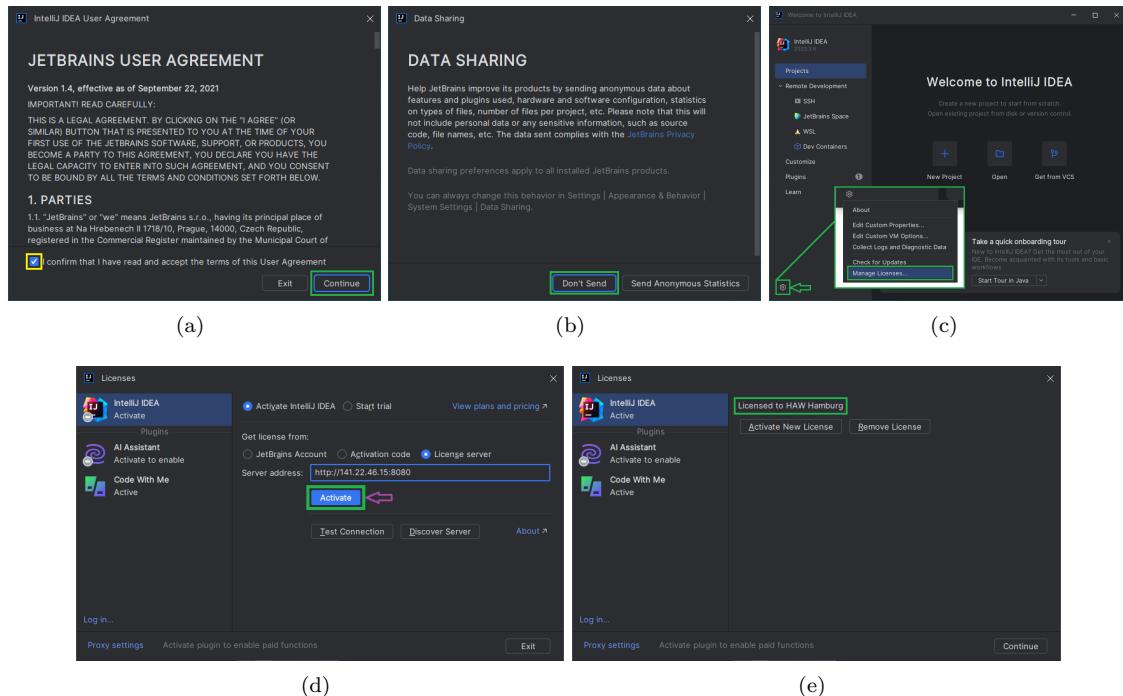


Abbildung 14.1: Aktivierung von IntelliJ IDEA im PC-Pool

14.2 Projektstruktur

Sie benötigen ein Projekt, in dem Sie Ihre Lösungen dieser Einführung sowie der nachfolgenden Praktika jeweils in einem eigenen Paket ablegen:

1. Erzeugen Sie ein neues IntelliJ IDEA-Projekt. Wahlweise dürfen Sie die Lösungen der Einführung und Labore auch in einem bestehenden anderen Projekt mit aufnehmen.
2. Erstellen Sie das Paket *introduction* für die in dieser Einführung erarbeiteten Klassen.

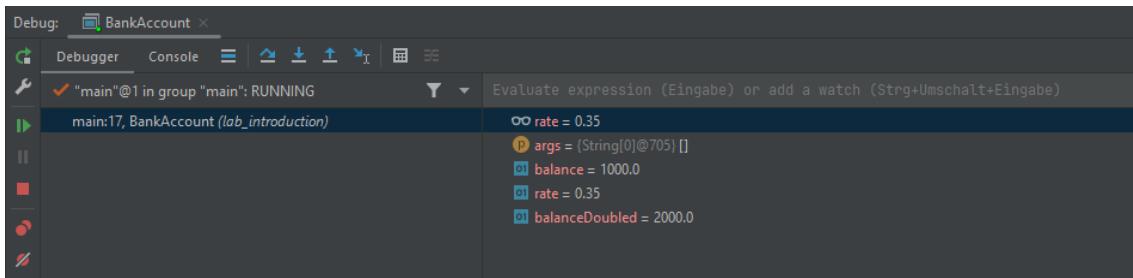


Abbildung 14.2: Haltepunkt im Debug-Modus

```

19     // Determine years until doubling balance
20     while (balance < balanceDoubled) { balanceDoubled: 2000.0
21         balance *= 1 + rate / 100.0;    balance: 1083.6765282886806   rate: 0.35
22         years++;    years: 22
23     }

```

Abbildung 14.3: Aktuelle Werte im Quelltext-Editor im Debug-Modus

14.3 Debugging

Als Erstes sollen Sie sich mit dem Erstellen eines ausführbaren Programms sowie der Arbeit mit dem Debugger vertraut machen. Gehen Sie hierfür wie folgt vor:

- Ausführbares Programm:* Lösen Sie Aufgabe 18 auf Seite 14 in einer Klasse *BankAccount*.
- Setzen von Breakpoints:* Setzen durch linken Mausklick neben die entsprechende Zeilennummer einen Haltepunkt („Breakpoint“) vor der in Ihrer Lösung implementierten Schleife.
- Ausführung im Debug-Modus:* Starten Sie das Programm mittels der in Tabelle A.1 auf Seite 147 angegebenen Tastenkombination im Debug-Modus. Das Programm sollte am gesetzten Haltepunkt stoppen und wie in Abbildung 14.2 dargestellt einen Debug-Bereich mit Kontrollelementen sowie den aktuellen Werten der am Haltepunkt deklarierten Variablen anzeigen.
- Auswerten von Ausdrücken:* Verwenden Sie das Eingabefeld *Evaluate expression*, um einen exemplarischen Ausdruck (z. B. Kontostand multipliziert mit Prozentsatz) auszuwerten. Stoppen Sie anschließend die Ausführung des Programms durch Auswahl des roten Quadrates im Kontroll-Bereich.
- Entfernen von Breakpoints:* Entfernen Sie den Haltepunkt durch erneuten Mausklick, setzen Sie einen neuen innerhalb der Schleife und starten Sie erneut im Debug-Modus.
- Schrittweise Ausführung:* Nutzen Sie das entsprechende Symbol (*Step over*) mit blauem Pfeil im Kontroll-Bereich, um das Programm ausgehend vom Haltepunkt Anweisung für Anweisung in Einzelschritten auszuführen. Beobachten Sie, wie die aktuellen Werte der Variablen als hilfreiche Zusatzinformation im Quelltext-Editor angezeigt werden (Abb. 14.3).
- Weitere Ausführung:* Nutzen Sie anschließend das Symbol (*Step out*), um aus dem aktuellen Schleifenkörper zu springen und das Programm bis zum erneuten Erreichen des Haltepunktes auszuführen.
- Ausführung bis Cursor:* Nutzen Sie das Symbol (*Run to cursor*), um das Programm bis zur aktuellen Cursor-Position auszuführen.

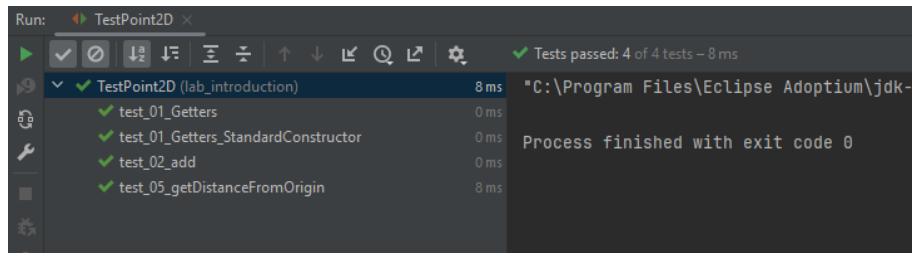


Abbildung 14.4: Ausgabe bei erfolgreicher Ausführung aller JUnit-Tests

14.4 JUnit-Tests

In diesem zweiten praktischen Teil sollen Sie die grundlegende Arbeit mit JUnit-Tests kennenlernen. Hierfür ist eine ganz kleine Mini-Klasse *Point2D* zu entwickeln, die Punkte $P(x, y) \in \mathbb{Z}^2$ in der Ebene durch ganzzahlige Koordinaten x und y vom Datentyp *int* repräsentiert. Gehen Sie wie folgt vor:

1. Konfigurieren Sie IntelliJ IDEA wie in Anhang A.3.1 beschrieben derart, dass Tests auch bei Kompilierfehlern ausgeführt werden.
2. Fügen Sie die gegebene Test-Datei *TestPoint2D.java* wie in Anhang A.3.2 beschrieben zum Paket *introduction* hinzu.
3. Binden Sie die JUnit-Bibliothek wie in Anhang A.3.2 beschrieben ein.
4. Erstellen Sie eine Klasse *Point2D* mit Attributen x und y vom Datentyp *int*.
5. Entnehmen Sie der Test-Datei, welche Methoden die Klasse *Point2D* besitzen soll. Implementieren Sie die Methoden derart, dass bei Ausführung der Klasse *TestPoint2D* alle Tests erfolgreich durchlaufen (Abb. 14.4).

Kapitel 15

Praktikum 1: Banditen

Primäre Lernziele

- ▶ Einfache Klassen mit Variablen und Methoden
- ▶ Ausführbare Klassen
- ▶ Statische Elemente
- ▶ Arrays

15.1 Übersicht

Lassen Sie uns spielen! In diesem und den nächsten beiden Kapiteln werden wir in aufeinander aufbauenden Aufgaben ein kleines Casino umsetzen und – traurig, aber wahr – zunächst etwas Geld verlieren. Glücksspiel lohnt sich halt nicht¹.

In Kapitel 16 schlagen wir jedoch zurück! Wie werden nicht nur eine weitere Art von Spiel hinzunehmen, sondern insbesondere einen Automaten einschleusen, bei dem wir langfristig gewinnen. Das Problem ist nur, dass wir nicht wissen, um *welchen* der vielen Automaten es sich dabei handelt. Daher müssen wir ganz im Geiste des Films *Ocean's Eleven* überlegen, wie wir dieses Problem knacken und das Casino um Bares erleichtern. Abschließend erzeugen wir in Kapitel 17 als Kommandozentrale eine Anwendung mit grafischer Benutzeroberfläche, um das Spiel als *Hensel's Eleven* mit verschiedenen Strategien zu steuern und zu visualisieren.

Ganz nebenbei handelt es sich bei unserer Gewinnstrategie um ein einführendes Beispiel aus dem Bereich des sogenannten *bestärkenden Lernens* bzw. *Reinforcement Learning*. Dieses spannende Teilgebiet der künstlichen Intelligenz befasst sich damit, wie Computer Verhaltensweisen bzw. Aktionen erlernen, und ist Grundlage diverser Anwendungen (z. B. autonomes Fahren).

15.2 Einarmige Banditen

In dieser ersten Teilaufgabe befassen wir uns mit *Slot Machines* – vermutlich *dem* Klassiker unter den Automaten in Spielhallen und Casinos. Der gebräuchliche Name *einarmiger Bandit* beschreibt das Spielprinzip sehr gut. Man wirft eine Münze ein, zieht an einem Hebel („Arm“) und hinterher ist das Geld weg. Natürlich gibt es eine geringe Chance, dass man gewinnt und der Bandit jede Menge Münzen ausspuckt². Auf lange Sicht verliert man jedoch.



15.2.1 Mathematische Beschreibung

Wir werden das Verhalten des Automaten wie folgt umsetzen:

1. Spieler werfen für jede Runde einen festgelegten Betrag (zum Beispiel 1,- €) ein.
2. Der Automat bestimmt für den als Gewinn auszuzahlenden Betrag zunächst eine Zufallszahl nach einer Gaußverteilung $p(x)$ mit Mittelwert μ und Standardabweichung σ :

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \cdot e^{-\frac{1}{2} \cdot \left(\frac{x-\mu}{\sigma}\right)^2} \quad (15.1)$$

¹Genau genommen lohnt es sich sehr wohl, allerdings nur für diejenigen, die die Spielautomaten betreiben.

²Das unterscheidet den Automaten dann doch von einem echten Banditen

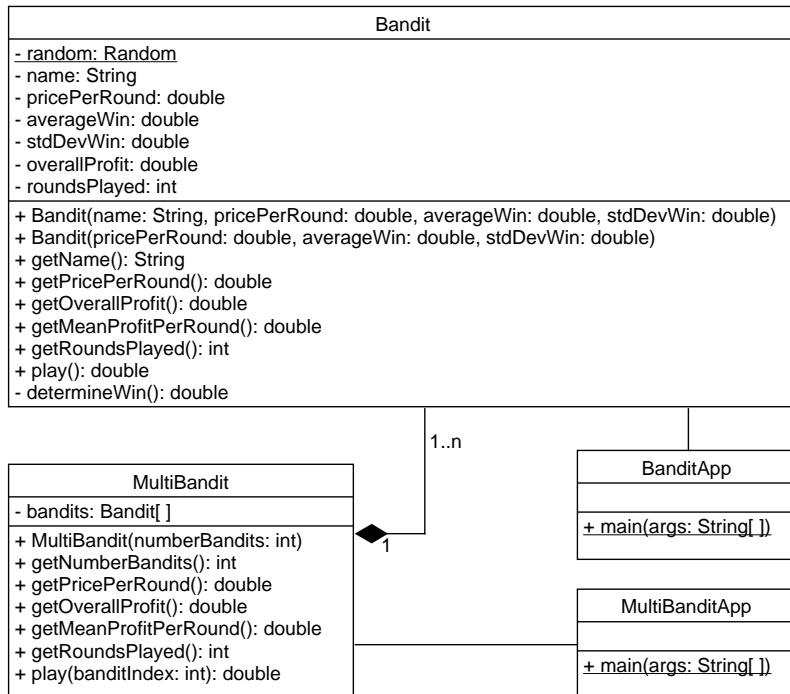


Abbildung 15.1: Klassendiagramm

- Der ermittelte Betrag wird vor Ausgabe auf Vielfache von 10 Cent gerundet. Zudem soll bei negativen Zufallszahlen kein weiteres Geld von Spielern eingefordert werden, sodass bei „negativen Gewinnen“ 0,- € ausgegeben werden.

Der Mittelwert μ steuert also den langfristigen Gewinn bzw. Verlust des Automaten. Sieht man von der Rundung und der Behandlung negativer Zahlen ab, würden Spieler bei der Wahl eines Mittelwertes kleiner dem Preis pro Runde langfristig weniger ausgezahlt bekommen als sie an Münzen in den Automaten einwerfen. Über die Standardabweichung σ lässt sich hingegen einstellen, wie stark der ausgezahlte Betrag um den Mittelwert schwankt.

15.2.2 Projektstruktur und JUnit-Tests

- Erzeugen Sie ein Paket `ai_bandit.lab1` und fügen Sie wie in Anhang A.1 beschrieben die gegebenen Dateien `TestBandit.java` und `TestMultiBandit.java` mit JUnit-Tests hinzu.
- Binden Sie wie in Anhang A.3 beschrieben eine JUnit-Bibliothek ein.

15.2.3 Klassendiagramm

Abbildung 15.1 enthält eine Übersicht aller in diesem Kapitel zu erstellenden Klassen mitsamt ihrer Attribute und Methoden:

- In den Kästen unterhalb der Klassennamen sind die Attribute mit jeweiligem Datentyp angegeben. Durch ein Minuszeichen wird gekennzeichnet, dass der Modifier `private` zu verwenden ist, während ein Pluszeichen `public` entspricht.
- Im unteren Abschnitt sind jeweils die Methoden mit Parameterlisten sowie gegebenenfalls nach dem Doppelpunkt dem Datentyp ihres Rückgabewertes aufgelistet.

- Unterstrichene Elemente sind statisch (d. h. Klassenvariablen bzw. Klassenmethoden).

In dieser ersten Teilaufgabe sind zunächst nur *Bandit* und *BanditApp* zu erstellen, wobei *Bandit* einen einarmigen Banditen repräsentiert und *BanditApp* diesen anhand einer interaktiven Anwendung demonstriert.

15.2.4 Klasse *Bandit*

Erstellen Sie die Klasse *Bandit* gemäß Abbildung 15.1 sowie den nachfolgenden Anforderungen. Führen Sie während der Entwicklung immer wieder die zur Verfügung gestellten JUnit-Tests aus, um Fehler zu identifizieren und beheben³.

Attribute und Konstruktoren

- R1 Die Klasse besitzt die im Klassendiagramm aufgeführten Variablen mit den jeweils angegebenen Bezeichnern und Datentypen. Hierbei entsprechen *averageWin* und *stdDevWin* den Parametern μ und σ der Gaußverteilung in Formel 15.1. Die Variable *overallProfit* enthält den gesamtes bisherigen Gewinn aus Sicht des Automaten.
- R2 Die Klassenvariable *random* wird mit einem Objekt der Klasse *Random* zur Erzeugung von Zufallszahlen initialisiert.
- R3 Es existieren die im Diagramm angegebenen Konstruktoren, welche die Parameterwerte den entsprechenden Attributen zuweisen.

Getter

- R4 Die Methoden *getName()*, *getPricePerRound()*, *getOverallProfit()* und *getRoundsPlayed()* geben die Werte der entsprechenden Attribute zurück.
- R5 Die Methode *getMeanProfitPerRound()* gibt den durchschnittlichen Gewinn pro Runde aus Sicht des Automaten zurück. Wurde noch keine Runde gespielt, wird der Wert 0 zurückgegeben.

Spiel

- R6 Die Methode *play()* spielt eine Runde und aktualisiert die entsprechenden Attribute der Spielstatistiken. Sie ermittelt den Gewinn durch Aufruf der nachfolgend beschriebenen Methode *determineWin()* und gibt diesen zurück.
- R7 Die Methode *determineWin()* ermittelt unter Verwendung des über *random* referenzierten Zufallsgenerators den Gewinn nach Abschnitt 15.2.1 und gibt diesen zurück.

Unit-Tests und Qualität

- R8 Stellen Sie sicher, dass alle gegebenen Unit-Tests fehlerfrei durchlaufen. Beachten Sie gegebenenfalls Anhang A.3 auf Seite 145.
- R9 Stellen Sie sicher, dass alle in der Checkliste zur Softwarequalität in Anhang C auf Seite 153 aufgeführten Qualitätskriterien erfüllt sind.

15.2.5 Klasse *BanditApp*

Bislang haben wir einen neuen Datentypen, jedoch noch kein Programm.

³Machen! Ganz ehrlich, das ist wichtig.

Gambling: One-armed bandit
Price : 1,00 EUR
How many rounds would you like to play? 10
Round Win [EUR] Net [EUR]
1 0,90 -0,10
2 1,30 0,20
3 0,10 -0,70
4 0,00 -1,70
5 0,10 -2,60
6 0,00 -3,60
7 0,40 -4,20
8 2,30 -2,90
9 0,00 -3,90
10 0,60 -4,30
One-armed bandit's statistics:
Rounds: 10
Profit: 4,30 (0,43 EUR/round)

Listing 15.1: Einarmiger Bandit (exemplarisches Spiel, Eingaben in **Fettdruck**)

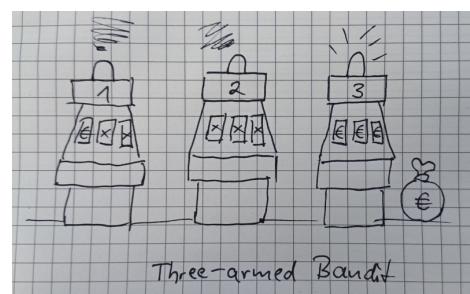
Funktionalität Erstellen Sie daher in einer Klasse *BanditApp* und unter Verwendung von *Bandit* eine Anwendung mit folgender Funktionalität:

1. Ausgabe des Automaten-Namens sowie des Preises pro Runde
2. Benutzereingabe der Anzahl zu spielenden Runden. Die Konsolen-Eingabe kann beispielsweise über die Methode *nextInt()* eines Objektes der Klasse *Scanner* eingelesen werden.
3. Tabellarische Ausgabe der gespielten Runden mit fortlaufender Nummer, vom Banditen zurückgegebenen Betrag und Entwicklung des Geldguthabens
4. Ausgabe der Anzahl gespielter Runden, des Gesamt-Profits und des Profits pro Runde aus Sicht des Automaten

Beispiel Betrachten Sie zur Verdeutlichung der Anforderungen die exemplarische Ausgabe in Listing 15.1. Aus Platzgründen wurden lediglich zehn Runden gespielt. Experimentieren Sie ruhig mit einer deutlich höheren Anzahl an Runden.

15.3 Mehrarmige Banditen

Im zweiten Aufgabenteil stehen wir nicht nur einem einarmigen Banditen, sondern gleich mehreren gegenüber. Wir haben daher in jeder Runde die Wahl, an welchem dieser Automaten wir spielen. Man spricht daher auch von *mehrarmigen Banditen*. Als Besonderheit werden wir die Banditen derart initialisieren, dass wir bei einem von ihnen langfristig gewinnen und bei allen anderen verlieren. Wir wissen jedoch nicht, welcher von den Automaten uns Gewinn statt Verlust einbringt.



```
Gambling: Multi-armed bandit (7 bandits)
Price   : 1,00 EUR
```

```
How many rounds would you like to play? 10
```

Round	Bandit	Win [EUR]	Net [EUR]
1	4	0,00	-1,00
2	2	0,00	-2,00
3	3	2,50	-0,50
4	6	0,00	-1,50
5	1	0,90	-1,60
6	3	1,80	-0,80
7	1	0,20	-1,60
8	7	0,00	-2,60
9	6	0,30	-3,30
10	4	0,60	-3,70

```
Multi-armed bandit's statistics:
Rounds: 10
Profit: 3,70 (0,37 EUR/round)
```

Listing 15.2: Mehrarmiger Bandit (exemplarisches Spiel, Eingaben in **Fettdruck**)

15.3.1 Klasse *MultiBandit*

Erstellen Sie die Klasse *MultiBandit* gemäß Abbildung 15.1 sowie den nachfolgenden Anforderungen. Führen Sie während der Entwicklung immer wieder die zur Verfügung gestellten JUnit-Tests aus, um Fehler zu identifizieren und beheben.

Attribute und Konstruktor

- R1 Die Klasse besitzt die im Klassendiagramm aufgeführte Variable *bandits* vom Typ *Bandit[]*.
- R2 Der Konstruktor erzeugt die als Parameter übergebene Anzahl einarmiger Banditen, wobei der Preis pro Runde einheitlich 1,- € beträgt. Die bei der Erzeugung der Banditen übergebene Standardabweichung beträgt in allen Fällen $\sigma = 1,0$.
- R3 Es wird zufällig bestimmt, bei welchem der Automaten Spieler langfristig gewinnen. Als Mittelwert μ wird eine Zufallszahl zwischen 1,1 und 1,3 gewählt. Bei allen anderen Automaten wird μ mit einer Zufallszahl zwischen 0,5 und 0,8 initialisiert.

Getter und Spiel

- R4 Die Methoden *getNumberBandits()*, *getPricePerRound()*, *getOverallProfit()*, *getMeanProfitPerRound()* und *getRoundsPlayed()* geben die von ihren Namen implizierten Werte zurück. Die Werte beziehen sich dabei auf die Gesamtheit aller Automaten, nicht eines bestimmten Banditen.
- R6 Die Methode *play()* spielt eine Runde mit dem als Index übergebenen Banditen und gibt den Gewinn zurück.

15.3.2 Klasse *MultiBanditApp*

Analog zu *BanditApp* ist eine ausführbare Klasse *MultiBanditApp* zu erstellen, welche die Funktionalität von *MultiBandit* demonstriert. Hierbei wird pro Runde per Zufall gewählt, an welchem Automaten gespielt wird. Erneut wollen wir uns das geforderte Programm an einer exemplarischen Ausgabe, dargestellt in Listing 15.2, veranschaulichen.

15.4 Ausblick

Na gut, wir haben nun einen mehrarmigen Banditen und wir wissen sogar, dass einer der Hebel für uns Gewinn abwirft, wenn wir nur oft genug spielen. Aber welcher? Bislang verlieren wir nur unser sauer verdientes Geld! Lassen Sie uns den Automaten daher im nächsten Kapitel knacken und fortan in Saus und Braus leben⁴.

⁴Man wird ja wohl noch träumen dürfen ...

Kapitel 16

Praktikum 2: Spielhalle und Gewinn-Strategie

Hinweise

- ▶ Primäre Lernziele: Vererbung, abstrakte Elemente
- ▶ Organisieren Sie die Quelltexte im Paket `ai_bandit.lab2`.
- ▶ Die JUnit-Tests müssen fehlerfrei durchlaufen.

16.1 Übersicht

In Kapitel 15 haben wir Klassen für ein- und mehrarmige Banditen erstellt und hilflos zugeschaut, wie diese uns das Geld aus der Tasche ziehen. Darauf aufbauend sollen Sie schrittweise die in Abbildung 16.1 dargestellten, farblich gekennzeichneten Erweiterungen umsetzen:

- ▶ *Spielhalle (grau)*: Neben Banditen sind diverse andere Glücksspiele denkbar, die ein Casino bzw. eine Spielhalle beinhalten kann. Da diese gemeinsame Eigenschaften wie den Preis pro Runde und Spielstatistiken aufweisen, werden wir die Gemeinsamkeiten in eine Basisklasse auslagern. Zudem werden wir als zweite Art Spiel exemplarisch Glücksräder umsetzen.
- ▶ *Hensel's Eleven (orange)*: Wie in Kapitel 15 versprochen, wollen wir nicht weiter unser sauer verdientes Geld einsetzen und verlieren. Lassen Sie uns daher das Wissen, dass einer der Automaten eines mehrarmigen Banditen langfristig Gewinn abwirft, nutzen.

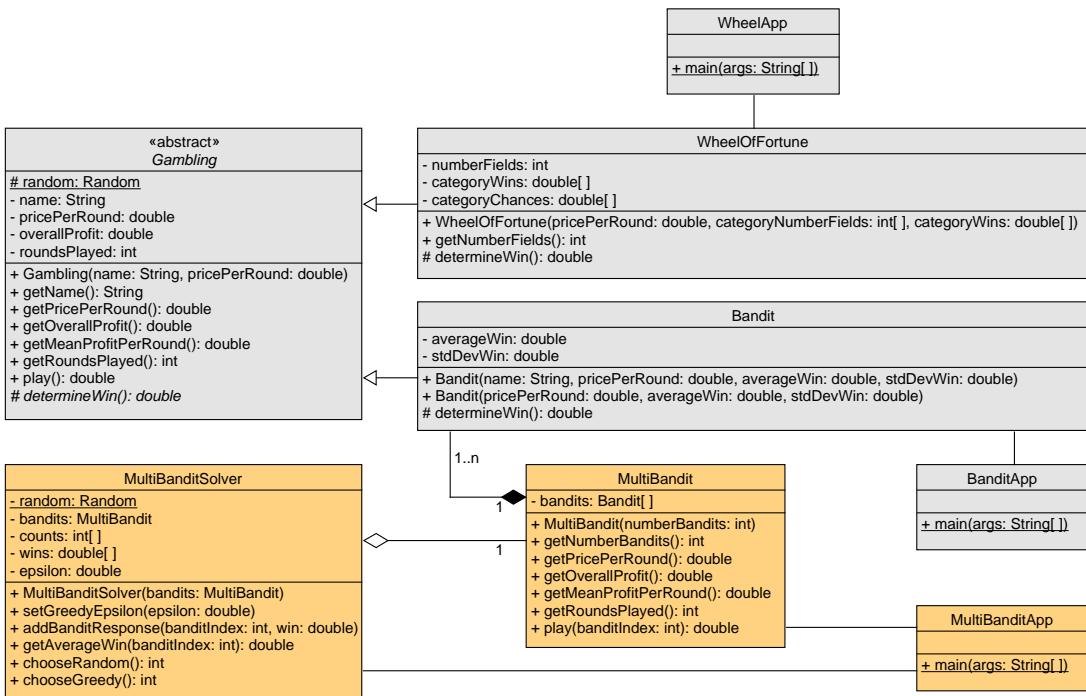


Abbildung 16.1: Klassendiagramm (vollständig)

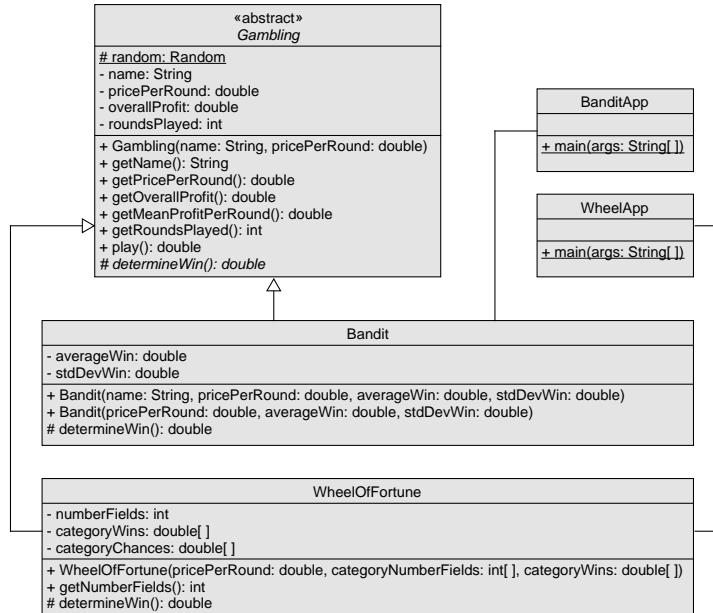
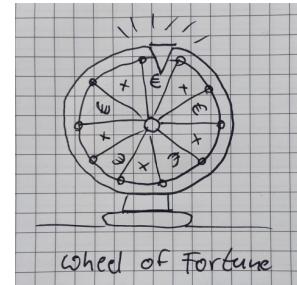


Abbildung 16.2: Klassendiagramm (Glücksspiele)

16.2 Spielhalle (Glücksräder)

Abbildung 16.2 beinhaltet die in dieser ersten Teilaufgabe umzusetzenden Klassen mit *Bandit* und *WheelOfFortune* als Repräsentationen einarmiger Banditen bzw. Glücksräder, einer gemeinsamen Basisklasse *Gambling* sowie den Anwendungen *BanditApp* und *WheelApp*. Die Klassen *Gambling* sowie *Bandit* ergeben sich im Wesentlichen durch Aufteilung der in Kapitel 15 erstellten Klasse *Bandit*. Die Repräsentation des Glücksrades kommt hingegen gänzlich neu hinzu. Die Anwendung *BanditApp* kann unverändert übernommen werden, während sich *WheelApp* mit nur geringen Anpassungen aus dieser ergibt.



16.2.1 Abstrakte Basisklasse

- R1 Die Klasse *Gambling* besitzt die im Diagramm dargestellten, ursprünglich in *Bandit* aus Kapitel 15 enthaltenen Attribute und Methoden.
- R2 Die Klassenvariable *random* ist *protected* und daher für abgeleitete Klassen sichtbar.
- R3 Die Methode *determineWin()* wird lediglich deklariert. Die Implementierung soll erst in abgeleiteten Klassen erfolgen.

16.2.2 Klassen *Bandit* und *BanditApp*

- R4 Die Klasse *Bandit* aus Kapitel 15 wird derart modifiziert, dass sie von *Gambling* erbt. Durch Vererbung enthaltene Attribute und Methoden werden nicht erneut deklariert bzw. definiert.
- R5 Die Klasse überschreibt die abstrakte Methode *determineWin()*.
- R6 Die Klasse *BanditApp* wird unverändert aus Kapitel 15 übernommen.

16.2.3 Klassen *WheelOfFortune* und *WheelApp*

R7 Die Klasse *WheelOfFortune* besitzt die im Klassendiagramm aufgeführten Variablen und Methoden mit den jeweils angegebenen Bezeichnern und Datentypen.

R8 Im Konstruktor gibt der Parameter *categoryNumberFields* an, wie viele Felder der jeweiligen Gewinnkategorie auf dem Glücksrad vorhanden sind. Der Parameter *categoryWins* gibt die zugehörige Beschriftung bzw. den Gewinn an.

Beispiel: Bei *categoryNumberFields* = `new int[]{ 20, 5}` und *categoryWins* = `new double[]{ 0.5, 2.0}` besitzt das Glücksrad 20 Felder mit einem Gewinn von 0,50€ sowie 5 Felder, auf denen man 2,-€ gewinnt.

Der Konstruktor initialisiert die Variable *numberFields* mit der gesamte Anzahl Felder auf dem Glücksrad sowie *categoryWins* mit dem übergebenen Parameter. Die Variable *categoryChances* wird hingegen mit der Wahrscheinlichkeit zwischen 0.0 und 1.0, dass ein Feld der entsprechenden Kategorie gedreht wird, initialisiert.

Beispiel: Bei *categoryNumberFields* = `new int[]{ 20, 5}` existieren 25 Felder. Die Wahrscheinlichkeiten bzw. Anteile der Felder betragen daher *categoryChances* = `new double[]{ 20/25.0, 5/25.0}`.

R9 Die Methode *getNumberFields()* gibt die Anzahl der Felder auf dem Glücksrad zurück.

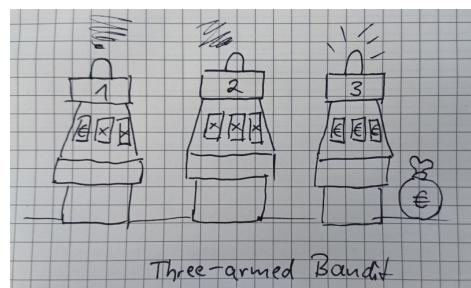
R10 Die Methode *determineWin()* gibt den zufälligen Gewinn beim Drehen am Glücksrad zurück.

R11 Die Anwendung *WheelApp* setzt die zu *BanditApp* äquivalente Funktionalität für das Spiel mit einem Glücksrad um.

Die exemplarischen Ausgabe in Listing 16.1 veranschaulicht die Anwendung *WheelApp* bei einem Glücksrad mit insgesamt 30 Feldern, die sich in 15, 10, 4 und 1 Felder der Gewinnkategorien 0,-€, 1,-€, 2,-€ bzw. 5,-€ unterteilen. Wie in Kapitel 15 wurden aus Platzgründen nur wenige Runden gespielt und angegeben. Experimentieren Sie mit anders aufgebauten Glücksrädern und vielen gespielten Runden.

16.3 Spiel mit mehrarmigen Banditen

Abbildung 16.3 beinhaltet die für diese Teilaufgabe relevanten Klassen. Die Klassen *Gambling* und *Bandit* wurden bereits in Abschnitt 16.2 erzeugt. *MultiBandit* unterscheidet sich nicht von der in Kapitel 15 implementierten Klasse, sollte aber dennoch kopiert werden, um *Bandit* aus dem aktuellen Paket *ai_bandit.lab2* zu verwenden. Die eigentliche Aufgabe besteht also in der Umsetzung der Klasse *MultiBanditSolver*, über welche der jeweils nächste zu spielende Bandit nach unterschiedlichen Strategien ausgewählt werden kann.



16.3.1 Strategie

Die bislang in *MultiBanditApp* umgesetzte Strategie besteht darin, bei jeder Runde einen zufälligen Banditen auszuwählen. Bedenkt man, dass nur ein einziger Automat langfristig Gewinn abwirft, während wir bei allen anderen verlieren, ist dies nicht sonderlich geschickt. Lassen Sie uns stattdessen beim Spielen beobachten, welche Gewinne bzw. Verluste wir mit den jeweiligen Banditen erzielen, um herausfinden, mit welchem der Banditen wir gewinnen.

```
Gambling: Wheel of fortune
Price   : 1,00 EUR
Fields  : 30
```

```
How many rounds would you like to play? 10
```

Round	Win [EUR]	Net [EUR]
1	1,00	0,00
2	2,00	1,00
3	0,00	0,00
4	0,00	-1,00
5	1,00	-1,00
6	0,00	-2,00
7	0,00	-3,00
8	0,00	-4,00
9	1,00	-4,00
10	1,00	-4,00

```
Wheel of fortune's statistics:
Rounds: 10
Profit: 4,00 (0,40 EUR/round)
```

Listing 16.1: Glücksrad (exemplarisches Spiel, Eingaben in **Fettdruck**)

Durchschnittlicher Gewinn Als Qualitätsmaß eignet sich der zu einem gegebenen Zeitpunkt erzielte durchschnittliche Gewinn, also der bislang insgesamt mit einem Automaten erzielte Gewinn geteilt durch die Anzahl Spiele mit diesem Automaten. Mathematisch ausgedrückt ist dies

$$\bar{r}_k = \frac{1}{N_k} \cdot \sum_{n=1}^{N_k} r_k(n) , \quad (16.1)$$

wobei $k \in \{0, 1, \dots, K-1\}$ den konkreten Banditen, N_k die bisherige Anzahl Spiele mit Bandit k und $r_k(n)$ den beim n -ten Spiel erzielten Gewinn (bzw. *Reward*) bezeichnet.

Exploitation Ein naheliegender Ansatz wäre, das bisherige Wissen zu nutzen, indem man bei jedem Spiel denjenigen Automaten k^* mit dem bislang höchsten durchschnittlichen Gewinn wählt:

$$k^* = \operatorname{argmax}_k \{\bar{r}_k\} \quad (16.2)$$

Dieser auch als Ausbeute bzw. *Exploitation* bezeichnete Ansatz funktioniert jedoch nicht sonderlich gut. Zwar nutzen wir das Wissen, welcher Bandit bislang die beste Performanz gezeigt hat. Allerdings ist die Gefahr groß, dass wir uns auf den falschen Automaten einstellen und daher nicht oft genug oder erst sehr spät mit dem Banditen spielen, der tatsächlich langfristig Gewinn abwirft.

Epsilon-Greedy-Verfahren Aus dem zuvor genannten Grund ist es sinnvoll, zumindest ab und zu mit zufälligen Automaten zu spielen, um diese zu „erkunden“ (*Exploration*). Konkret legen wir zunächst eine Wahrscheinlichkeit $\epsilon \in [0, 1]$ fest (z. B. $\epsilon = 15\%$). In jeder Runde wählen wir mit dieser Wahrscheinlichkeit ϵ einen zufälligen Banditen und ansonsten, also mit einer Wahrscheinlichkeit $1 - \epsilon$, denjenigen mit dem bislang höchsten durchschnittlichen Gewinn \bar{r}_k .

Bezeichnen $p \in [0, 1]$ eine pro Runde erzeugte Zufallszahl und $\operatorname{rand}(K)$ die zufällige Wahl eines Automaten k , so gilt also:

$$k^* = \begin{cases} \operatorname{rand}(K) & : p \leq \epsilon \\ \operatorname{argmax}_k \{\bar{r}_k\} & : p > \epsilon \end{cases} \quad (16.3)$$

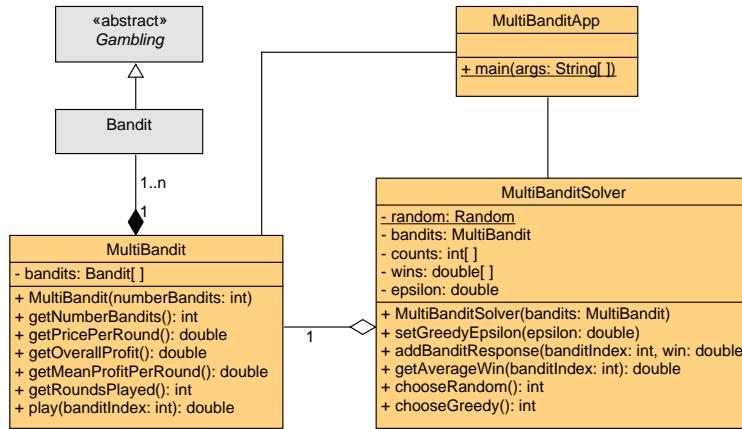


Abbildung 16.3: Klassendiagramm (Lösen von mehrarmigen Banditen)

16.3.2 Klasse `MultiBanditSolver`

Lassen Sie uns nun eine Klasse `MultiBanditSolver` erstellen, welche sowohl die zufällige Wahl des zu spielenden Banditen als auch die Wahl nach dem Epsilon-Greedy-Verfahren gemäß Formel 16.3 umsetzt. Hierfür muss die Klasse zum Ermitteln der durchschnittlichen Gewinne Statistik über die Anzahl der Spiele (Attribut `counts`) sowie die insgesamt erzielten Gewinne (Attribut `wins`) der Automaten führen.

- R1 Die Klasse `MultiBanditSolver` besitzt die im Klassendiagramm (Abb. 16.3) aufgeführten Variablen mit den jeweils angegebenen Bezeichnern und Datentypen.
- R2 Der Konstruktor weist `bandits` eine Referenz zu den zu spielenden K einarmigen Banditen mit Indizes $0, 1, \dots, K-1$ zu und initialisiert die Arrays. Das Array `counts` der Größe K zählt, wie viele Runden auf den einzelnen Banditen gespielt wurden. Das Array `wins` summiert die an den Banditen erzielten Umsätze aller bisherigen Spiele (d. h. zurückgegebener Betrag minus Kosten pro Runde).
- R3 Über die Methode `setGreedyEpsilon()` setzt den im Epsilon-Greedy-Verfahren verwendeten Parameter ϵ .
- R4 Über die Methode `addBanditResponse()` wird eine neue Runde zu der internen Statistik hinzugefügt. Hierfür wird übergeben, mit welchem Automaten gespielt wurde und welches Ergebnis (d. h. zurückgegebener Betrag minus Kosten der Runde) erzielt wurde.
- R5 Die Methode `getAverageWin()` gibt den bisherigen durchschnittlichen Gewinn (d. h. zurückgegebener Betrag minus Kosten der Runde) eines konkreten Banditen zurück.
- R6 Die Methoden `chooseRandom()` bzw. `chooseGreedy()` wählen den nächsten zu spielenden Automaten zufällig oder mittels des Epsilon-Greedy-Verfahrens und geben den entsprechenden Index zurück.

16.3.3 Klasse `MultiBanditApp`

Modifizieren Sie die Anwendung `MultiBanditApp` aus Kapitel 15 derart, dass Anwender eingeben können, ob die Auswahl der Banditen zufällig oder per Epsilon-Greedy-Verfahren mit einem speziellen Wert für den Parameter ϵ ausgewählt werden.

Die exemplarische Ausgabe in Listing 16.2 veranschaulicht die Anwendung zunächst für ein Spiel mit zufälliger Wahl. Bei Wahl der Epsilon-Greedy-Strategie mit $\epsilon = 0,15$ ist exemplarisch

```
Gambling: Multi-armed bandit (7 bandits)
Price   : 1,00 EUR

How many rounds would you like to play? 10
Enter epsilon in [0,100] percent (typical value: 15) or any other number for random
strategy: -1

Round | Bandit | Win [EUR] | Net [EUR]
-----|-----|-----|-----
  1  |    4  |  0,60  | -0,40
  2  |    3  |  0,60  | -0,80
  3  |    5  |  0,40  | -1,40
  4  |    1  |  0,00  | -2,40
  5  |    7  |  0,00  | -3,40
  6  |    3  |  0,60  | -3,80
  7  |    4  |  1,70  | -3,10
  8  |    2  |  0,50  | -3,60
  9  |    4  |  1,40  | -3,20
 10 |    7  |  0,40  | -3,80

Applied strategy: random

Multi-armed bandit's statistics:
Rounds: 10
Profit: 3,80 (0,38 EUR/round)
```

Listing 16.2: Mehrarmiger Bandit (zufällige Wahl, Eingaben in **Fettdruck**)

der Programmablauf in Listing 16.3, in dem in nur zehn gespielten Runden der 7. Bandit als gewinnbringend identifiziert und insgesamt ein Gewinn erzielt wurde.

```
Gambling: Multi-armed bandit (7 bandits)
Price   : 1,00 EUR

How many rounds would you like to play? 10
Enter epsilon in [0,100] percent (typical value: 15) or any other number for random
strategy: 15

Round | Bandit | Win [EUR] | Net [EUR]
-----|-----|-----|-----
  1  |    1  |  2,50  |  1,50
  2  |    1  |  0,30  |  0,80
  3  |    3  |  0,40  |  0,20
  4  |    1  |  0,70  | -0,10
  5  |    7  |  2,20  |  1,10
  6  |    7  |  1,20  |  1,30
  7  |    7  |  2,60  |  2,90
  8  |    7  |  0,80  |  2,70
  9  |    7  |  0,10  |  1,80
 10 |    7  |  0,70  |  1,50

Applied strategy: epsilon-greedy (epsilon = 0,15)

Multi-armed bandit's statistics:
Rounds: 10
Profit: -1,50 (-0,15 EUR/round)
```

Listing 16.3: Mehrarmiger Bandit ($\epsilon = 0,15$, Eingaben in **Fettdruck**)

Kapitel 17

Praktikum 3: GUI für mehrarmige Banditen

Hinweise

- ▶ Primäre Lernziele: Grafische Benutzeroberflächen, Parallelverarbeitung
- ▶ Organisieren Sie die Quelltexte im Paket `ai_bandit.lab3`.

17.1 Übersicht

In Kapitel 15 haben wir ein- und mehrarmige Banditen erstellt und es in Kapitel 16 geschafft, den „guten Arm“ zu identifizieren und mit mehr Geld in den Taschen aus dem Casino zu kommen als wir beim Hereingehen hatten. Das soll uns erstmal jemand nachmachen! Nun ist es an der Zeit, die Spielstrategien mit mehrarmigen Banditen in einer Anwendung mit ansprechender grafischen Oberfläche zu visualisieren.

17.1.1 Exemplarische Oberfläche

Abbildung 17.1 zeigt exemplarisch, wie die Benutzeroberfläche beim Start des Programms aussehen könnte. Beachten Sie in diesem Zusammenhang, dass in der Regel lediglich Funktionalitäten gefordert sind. Wie Sie diese konkret umsetzen bzw. grafisch gestalten und wie Sie die Software in Klassen strukturieren bleibt hingegen Ihnen überlassen.

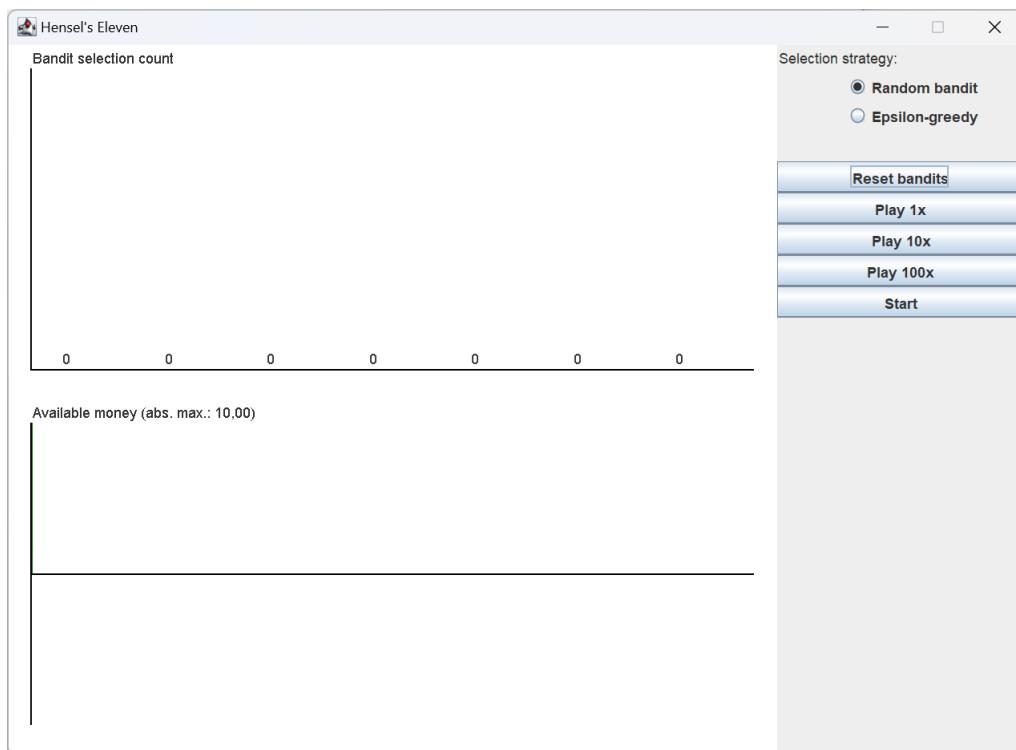


Abbildung 17.1: Exemplarische grafische Benutzeroberfläche

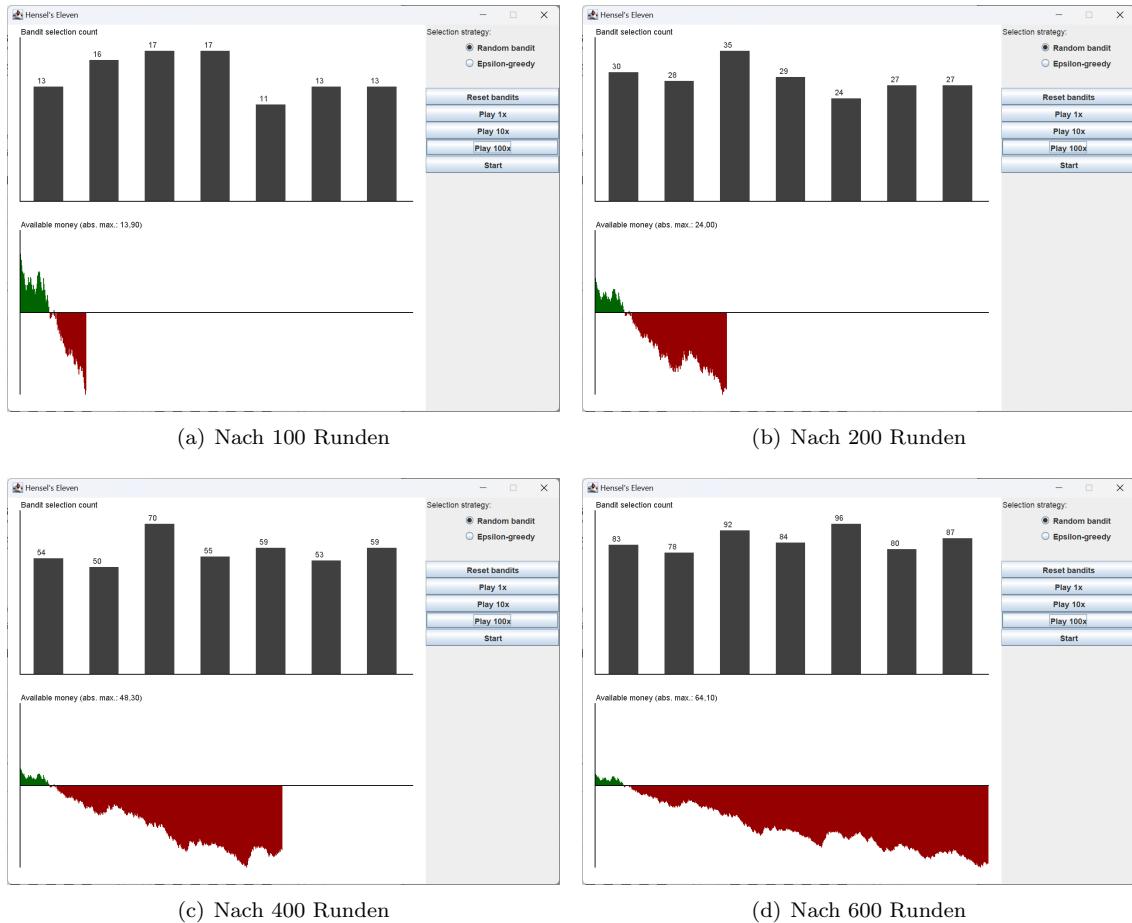


Abbildung 17.2: Exemplarisches Spiel mit zufälliger Wahl

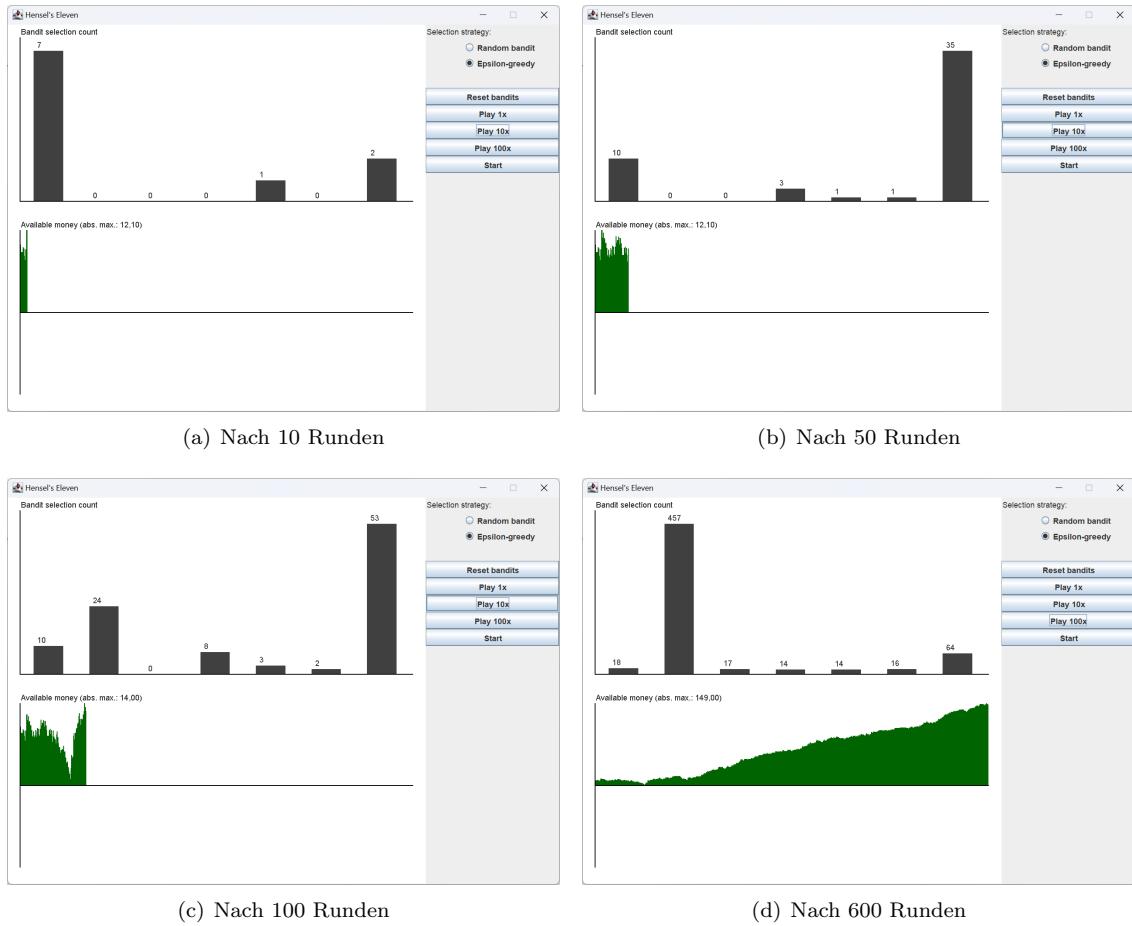
17.1.2 Funktionalität

Im Wesentlichen muss Ihre Anwendung Folgendes beinhalten:

- Spielen einer oder mehrerer Runden (in Abb. 17.1 eine, zehn und 100)
- Auswahl der Spielstrategie (zufällige Wahl oder Epsilon-Greedy-Verfahren)
- Zurücksetzen des Guthabens und der gespielten Runden
- Grafische Darstellung, wie oft mit den einzelnen Banditen gespielt wurde
- Grafische Darstellung des Kontostandes im Verlauf der gespielten Runden
- Automatisches Spiel über die Zeit

Abbildung 17.2 veranschaulicht das Verhalten des Programms exemplarischen, wobei der Zustand nach 100, 200, 400 und 600 gespielten Runden dargestellt ist. Als Spielstrategie wurde die zufällige Wahl der Banditen eingestellt. Erwartungsgemäß werden die Automaten in etwa gleich häufig gewählt (siehe Histogramme im oberen Bereich der Anwendung) und das zur Verfügung stehende Geld sinkt zunächst auf null und geht dann immer weiter in die Schulden (siehe Kurverlauf im unteren Bereich).

In Abbildung 17.3 ist ein ungewöhnlicher Spielverlauf bei Anwendung des Epsilon-Greedy-Verfahrens dargestellt. Nach 10 Runden scheint der erste Bandit als „gut“ angenommen zu werden.

Abbildung 17.3: Exemplarisches Spiel mit Epsilon-Greedy-Strategie ($\epsilon = 0, 15$)

Nach 50 Runden überwiegt hingegen eindeutig der letzte Bandit. Zu diesem Zeitpunkt wurde noch nicht mit dem zweiten und dritten Automaten gespielt. Aus dem weiteren Verlauf ist jedoch ersichtlich, dass der zweite Bandit langfristig Gewinn abwirft und das Guthaben nach anfänglichen Verlusten im Mittel kontinuierlich ansteigt.

17.2 Anforderungen

Die Anforderungen sollten nach dem im vorherigen Abschnitt gegebenen Überblick keine Überraschungen beinhalten. Entwickeln Sie ein Programm mit grafischer Benutzeroberfläche, welche die nachfolgenden Anforderungen erfüllt. Die konkrete Umsetzung muss hierbei nicht mit der in den Abbildungen dargestellten exemplarischen Anwendung übereinstimmen.

- R1 Die Anwendung visualisiert das Spiel mit einem 7-armigen Banditen der Klasse *MultiBandit* aus Kapitel 15.
- R2 Das initiale Guthaben beträgt 10,- €.
- R3 Es lässt sich einstellen, ob die zu spielenden Banditen zufällig oder mittels Epsilon-Greedy-Verfahren aus Kapitel 16 gewählt werden.
- R4 Nutzer können ausgehend vom jeweiligen Zustand des mehrarmigen Banditen eine einzelne Runde, 10 Runden oder 100 Runden spielen.

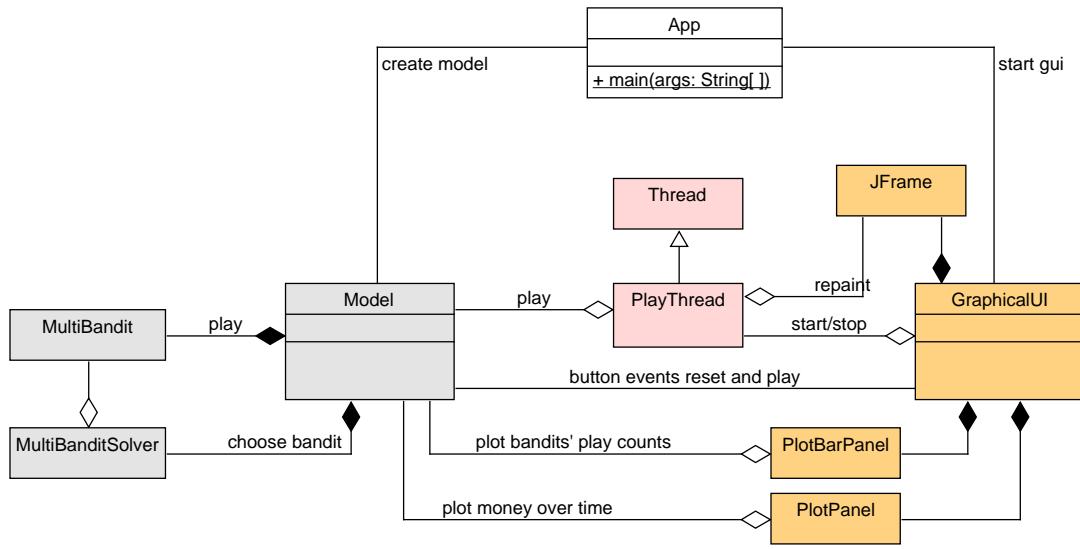


Abbildung 17.4: Vereinfachtes exemplarisches Klassendiagramm (Grau: Daten. Orange: Grafische Oberfläche. Rot: Multi-Threading.)

- R5 Das Spiel kann zurückgesetzt werden, wobei das Guthaben auf das initiale Guthaben und die Anzahl gespielter Runden auf null gesetzt wird.
 - R6 Es existiert eine grafische Balken-Darstellung, wie oft die einzelnen Banditen seit Programmstart bzw. dem letzten Zurücksetzen gespielt wurden.
 - R7 Es existiert eine grafische Darstellung des Guthabens als Funktion der gespielten Runden seit Programmstart bzw. dem letzten Zurücksetzen.
 - R8 Es lässt sich ein „automatisches Spiel“ starten und stoppen, bei dem fortlaufend (z. B. alle $\Delta t = 100$ ms) die jeweils nächste Runde gespielt wird.
- Die Anforderung R8 muss nur erfüllt werden, sofern wir vor dem Labortermin Parallelverarbeitung mittels Threads in der Vorlesung behandelt haben.*

17.3 Lösungsstrategie

- Entwerfen und implementieren Sie zunächst das Aussehen der grafischen Oberfläche (d. h. die Art und Anordnung der Elemente).
- Implementieren Sie erst im zweiten Schritt die geforderte Funktionalität.
- Verzetteln Sie sich nicht in Details bezüglich des Aussehens der grafischen Oberfläche. (Für besonders kreative und ästhetische Lösungen gibt es Bewunderung und Begeisterung, aber keinen Schönheitspreis.)
- Abbildung 17.4 enthält exemplarisch das Klassendiagramm meiner Musterlösung. Ihre Lösung muss sich nicht daran orientieren. Das Diagramm ist lediglich als Anregung zu verstehen.

Teil III

Praktika:

Geografische Koordinaten

Kapitel 18

Praktikum 1: Geografische Koordinaten

Primäre Lernziele

- ▶ Einfache Klassen mit Variablen und Methoden
- ▶ Ausführbare Klassen
- ▶ Statische Elemente



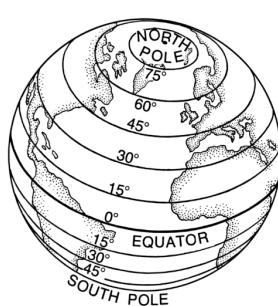
18.1 Theorie: Geografische Koordinaten und Entfernung

18.1.1 Breiten- und Längengrade

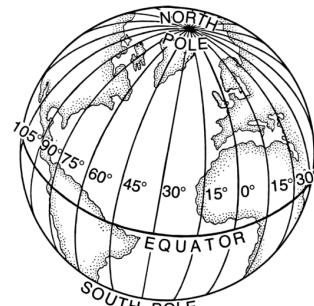
Geografische Koordinaten $g = (lat, lon)$ werden durch die *Breite* (*latitude*) und *Länge* (*longitude*) in Grad dargestellt.

- ▶ Breitengrade verlaufen parallel zum Äquator (Abb. 18.1a¹). Der Wertebereich umfasst -90° („südliche Breite“) am Südpol bis 90° („nördliche Breite“) am Nordpol.
- ▶ Längengrade laufen durch den Nordpol und den Südpol (Abb. 18.1b²). Der Wertebereich umfasst -180° („westliche Länge“) bis 180° („östliche Länge“). Der sogenannte Nullmeridian (Längengrad 0°) verläuft durch den Ort Greenwich in England.

Was	Bezeichner	Verlauf	Entspricht	Wertebereich
Breite	Latitude <i>lat</i>	Parallel zum Äquator	<i>y</i> -Koordinate	$[-90^\circ, 90^\circ]$
Länge	Longitude <i>lon</i>	Durch Nord- und Südpol	<i>x</i> -Koordinate	$[-180^\circ, 180^\circ]$



(a) Breitengrade



(b) Längengrade

Abbildung 18.1: Linien gleicher Längen- und Breitengrade

Hinweise:

- ▶ Beachten Sie, dass hierbei im Vergleich zu herkömmlichen Koordinaten die *x*- und *y*-Komponenten vertauscht sind.

¹[https://commons.wikimedia.org/wiki/File:Latitude_\(PSF\).png](https://commons.wikimedia.org/wiki/File:Latitude_(PSF).png) (Public Domain, besucht am 18.08.2019)

²[https://commons.wikimedia.org/wiki/File:Longitude_\(PSF\).png](https://commons.wikimedia.org/wiki/File:Longitude_(PSF).png) (Public Domain, besucht am 18.08.2019)

- In *Google Maps* erhalten Sie die geografischen Koordinaten, indem Sie auf einen Ort klicken.

18.1.2 Lokale Entfernungs berechnung

Wir wollen im Folgenden die Entfernung zweier Orte auf der Erdoberfläche bestimmen. Liegen zwei Orte relativ dicht beieinander, kann man näherungsweise die Erdkrümmung vernachlässigen:

1. Bestimme die Abstände in Richtung der Breiten- und Längengrade in km.
2. Berechne hieraus den direkten Abstand durch den Satz des Pythagoras

Teilt man den Erdumfang in 360° so entspricht je 1° in etwa 111,3 km. Da der Abstand benachbarter Breitengrade überall auf der Erde gleich groß ist (Abb. 1), entspricht ein Unterschied von 1° Breite jeweils 111,3 km. Der Abstand zweier Breitengrade lat_1 und lat_2 in km beträgt daher:

$$\Delta y = 111,3 \cdot |lat_1 - lat_2| \quad (18.1)$$

Der Abstand benachbarter Längengrade hängt vom Breitengrad der Orte ab. Am Äquator (0° Breite) entspricht der Abstand 111,3 km. Zu den Polen hin laufen die Längengrade aufeinander zu und der Abstand wird kleiner (Abb. 2). Am Nord- und Südpol schneiden sich die Längengrade, sodass dort der Abstand 0 km entspricht. Dies wird in folgender Formel durch den Kosinus ausgedrückt, der am Äquator (0° Breite) 1 und an den Polen ($\pm 90^\circ$ Breite) 0 ergibt. Als Argument wird der Mittelwert der Breitengrade beider Orte verwendet:

$$\Delta x = 111,3 \cdot \cos\left(\frac{lat_1 + lat_2}{2}\right) \cdot |lon_1 - lon_2| \quad (18.2)$$

Insgesamt ergibt sich für den Abstand d in km zweier geografischer Orte g_1 und g_2 (Abb. 18.2):

$$d = \sqrt{\Delta x^2 + \Delta y^2} \quad (18.3)$$

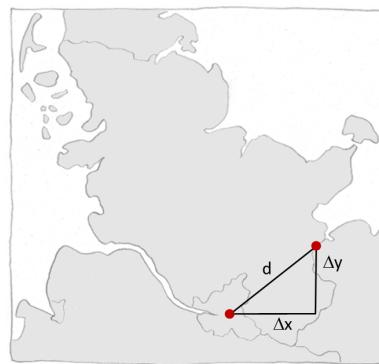


Abbildung 18.2: Lokale Entfernung zwischen Hamburg und Lübeck

18.1.3 Entfernungs berechnung auf der Erdkugel

Für eine genauere Berechnung der Entfernung zweier Orte auf der Erdoberfläche sei ohne Herleitung nachfolgende Formel gegeben³. Hierbei entspricht der Faktor 6378,388 km dem Erdradius.

$$d = 6378,388 \cdot \arccos(\sin(lat_1) \cdot \sin(lat_2) + \cos(lat_1) \cdot \cos(lat_2) \cdot \cos(lon_2 - lon_1)) \quad (18.4)$$

18.2 Klasse *GeoPosition*

³<https://www.kompf.de/gps/distcalc.html> (Besucht am 17.03.2020)

Es ist eine Klasse *GeoPosition* zur Repräsentation geografischer Koordinaten zu erstellen. Die Klasse wird grundlegend durch das nebenstehende UML-Symbol beschrieben. In dem Kasten unterhalb des Klassennamens sind die Attribute mit jeweiligem Datentyp angegeben. Durch ein Minuszeichen wird gekennzeichnet, dass der Modifier *private* zu verwenden ist.

Im unteren Abschnitt sind die Methoden mit Parameterlisten sowie nach dem Doppelpunkt dem Datentyp ihres Rückgabewertes aufgelistet. Unterstrichene Elemente sind Klassenmethoden.

GeoPosition
-latitude : double
-longitude : double
+GeoPosition(latitude : double, longitude : double)
+getLatitude() : double
+getLongitude() : double
+isNorthernHemisphere() : boolean
+isSouthernHemisphere() : boolean
+distanceInKm(other : GeoPosition) : double
+distanceInKm(a : GeoPosition, b : GeoPosition) : double
+localDistanceInKm(a : GeoPosition, b : GeoPosition) : double
+toString() : String

18.2.1 Eclipse und Projektstruktur

- Erstellen Sie in Eclipse ein Java-Projekt namens *Praktikum*, wobei Sie unter „*Project layout*“ die Option „*Use project folder as root for sources and class files*“ wählen. Erzeugen Sie im Projekt das Paket *lab1.geoPosition*.
- Fügen Sie dem Projekt die aktuellste *JUnit*-Bibliothek hinzu.
- Fügen Sie zum Paket die gegebene Test-Klasse *TestGeoPosition* hinzu. Sie können hierfür die Datei *TestGeoPosition.java* aus dem Windows-Explorer per „*Drag&Drop*“ in das Paket im Eclipse *Package Explorer* ziehen.

18.2.2 Attribute, Konstruktoren und Getter

- R1 Die Klasse besitzt private Variablen namens *latitude* und *longitude* zur Speicherung des Breitengrades bzw. Längengrades. Beide Variablen sind vom Typ *double*.
- R2 Es gibt einen Konstruktor mit zwei *double*-Parametern, der den Attributen die übergebenen Parameterwerte zuweist. (Tipp: Nutzen Sie in Eclipse den Menüpunkt „*Source / Generate ...*“, um die Methode automatisch erzeugen zu lassen.)
- R3 Es gibt Getter-Methoden, die den Wert von *latitude* bzw. *longitude* zurückgeben. (Lassen Sie auch diese Methoden automatisch in Eclipse erzeugen.)

Deklarationen:

```
public GeoPosition(double latitude, double longitude)
public double getLatitude()
public double getLongitude()
```

18.2.3 Abfrage der Hemisphäre

- R4 Es gibt Methoden zur Abfrage, ob sich eine geografische Position auf der nördlichen bzw. auf der südlichen Erdhalbkugel befindet.

Deklarationen:

```
public boolean isNorthernHemisphere()
public boolean isSouthernHemisphere()
```

18.2.4 Abstände zwischen zwei geografischen Orten

R5 Es gibt zwei Klassenmethoden zur Berechnung des Abstandes („Luftlinie“) zwischen zwei als Parameter übergebenen Orten. Der Abstand wird jeweils in Kilometern zurückgegeben. Die Methode *localDistanceInKm()* verwendet zur Berechnung das lokale Verfahren nach Abschnitt 18.1.2. Die Methode *distanceInKm()* verwendet die genauere Berechnung nach Abschnitt 18.1.3.

R6 Es gibt eine nicht-statische Methode *distanceInKm()*, die den Abstand zu einem als Parameter übergebenen Ort berechnet. Die Berechnung erfolgt nach Abschnitt 18.1.3.

Hinweise:

- ▶ Nutzen Sie für mathematische Funktionen die Methoden der Klasse *Math*.
- ▶ Beachten Sie, dass die trigonometrischen Funktionen Winkel in Radian (nicht Grad) erwarten.

Deklarationen:

```
public static double localDistanceInKm(GeoPosition a, GeoPosition b)
public static double distanceInKm(GeoPosition a, GeoPosition b)
public double distanceInKm(GeoPosition other)
```

18.2.5 Methode *toString()*

R7 Die Klasse besitzt eine Methode *toString()*, die eine wie folgt formatierte Zeichenkette zurückgibt: (<latitude>, <longitude>). (Tipp: Lassen Sie das Grundgerüst dieser Methode automatisch in Eclipse erzeugen.)

Deklaration:

```
public String toString()
```

18.2.6 Unit-Tests

R8 Stellen Sie sicher, dass alle gegebenen Unit-Tests fehlerfrei durchlaufen. Beachten Sie gegebenenfalls Anhang B.1 auf Seite 149.

18.3 Abstände

Erzeugen Sie eine ausführbare Klasse *GeoApp* (d. h. ein ausführbares Programm), das den Abstand der HAW Hamburg und den in Tabelle 18.1 aufgelisteten Orten auf Konsole ausgibt. Verwenden Sie zur Abstandsbestimmung jeweils das lokale sowie das genaue Verfahren und übertragen Sie die Ergebnisse in die Tabelle. Tragen Sie zudem ein, um wie viel Prozent das lokale Verfahren vom genauen Abstand abweicht.

Bestimmen Sie auf gleiche Weise die Entfernung der HAW zu den Polen und zum Äquator und tragen Sie diese in Tabelle 18.2 ein.

18.4 Einhaltung der Programmierrichtlinien (Qualität)

Stellen Sie sicher, dass alle in der Checkliste zur Softwarequalität in Anhang C auf Seite 153 aufgeführten Qualitätskriterien erfüllt sind.

Tabelle 18.1: Entfernung ausgewählter Orte zur HAW Hamburg

Ort	Breitengrad	Längengrad	Entfernung (18.4) zur HAW [km]	Lokale Entfernung (18.3) zur HAW [km]
HAW Hamburg	53,557078	10,023109	0,0	0,0
Eiffelturm	48,858363	2,294481	750,3	750,9
Palma de Mallorca	39,562553	2,661947		
Las Vegas	36,156214	-115,148736		
Copacabana	-22,971177	-43,182543		
Waikiki Beach	21,281004	-157,837456		
Surfer's Paradise	-28,002695	153,431781		

Tabelle 18.2: Entfernung des Äquators und der Pole zur HAW Hamburg

Ort	Breitengrad	Längengrad	Entfernung (18.4) zur HAW [km]	Lokale Entfernung (18.3) zur HAW [km]
Nordpol				
Äquator				
Südpol				

18.5 Theoretische Fragen

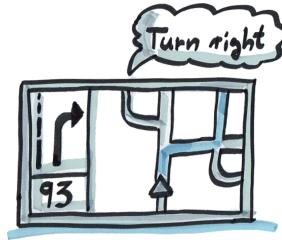
1. Wie unterscheiden sich *primitive Variablen* und *Referenzvariablen*?
2. Wie unterscheiden sich *Instanzvariablen*, *Objektvariablen*, *Attribute* und *lokale Variablen*?
3. Was wäre die Folge, wenn Sie die Variable *latitude* mit dem Schlüsselwort *static* versehen?
4. Es gibt zwei Methoden mit Namen *distanceInKm()*. Erläutern Sie den Unterschied.

Kapitel 19

Praktikum 2: Wegstrecken (Routen)

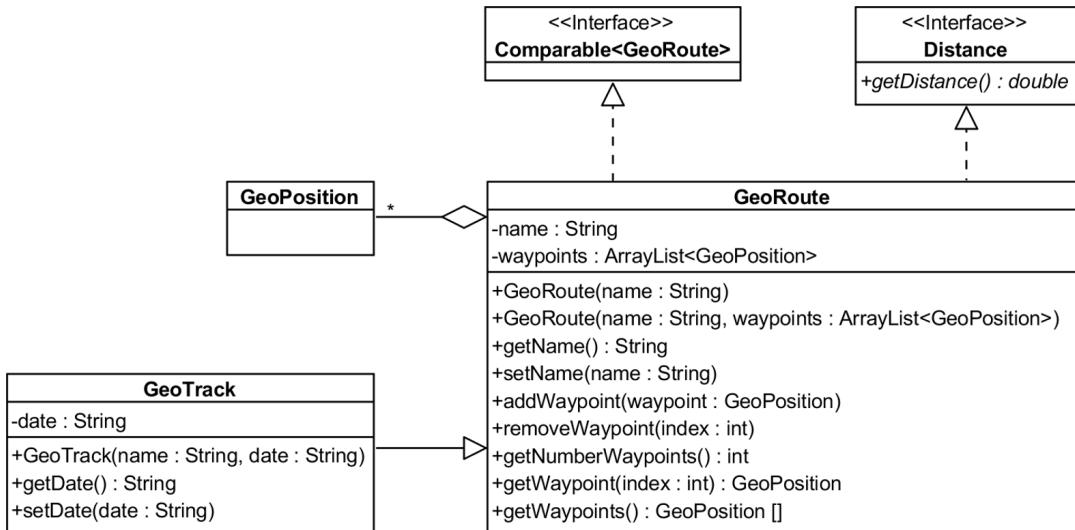
Hinweise

- ▶ Primäre Lernziele: Listen, Interfaces, Vererbung
- ▶ Organisieren Sie die Quelltexte im Paket `lab2.geoPosition`.
- ▶ Alle Instanzvariablen besitzen den Modifier `private`.
- ▶ Die JUnit-Tests müssen fehlerfrei durchlaufen.
- ▶ Erzeugen Sie Konstruktoren, Getter, Setter und `toString()` durch die Menüpunkte „*Source/Generate ...*“ und passen Sie diese gegebenenfalls den Anforderungen an.



19.1 Geografische Wegstrecken

Man kann nicht nur studieren, denken Sie auch mal an sich! Wie wäre es mit einer Runde um die Alster oder einer Reise? Lassen Sie uns hierzu aus geografischen Koordinaten Wegstrecken bzw. *Routen* erzeugen. Folgendes UML-Diagramm gibt einen Überblick der zu erstellenden Klassen:



- ▶ Ein durchgezogener Pfeil zeigt zur Basisklasse, während ein gestrichelter Pfeil mit ausgefüllter Pfeilspitze zu einem implementierten Interface zeigt.
- ▶ Die leere Raute bedeutet, dass Objekte der Klasse `GeoRoute` Objekte der Klasse `GeoPosition` enthalten können. Die Anzahl der enthaltenen Objekte ist durch die Angabe * beliebig.

Folgenden Zweck haben die Klassen und Interfaces:

- ▶ `GeoPosition` repräsentiert einen Ort. Verwenden Sie die in Praktikum 1 erstellte Klasse.
- ▶ `GeoRoute` repräsentiert eine Wegstrecke bzw. Route aus geografischen Orten.
- ▶ `GeoTrack` repräsentiert eine (z. B. beim Joggen) aufgezeichnete Wegstrecke.

- *RouteData* erzeugt einige Wegstrecken, um Ihnen Tipparbeit zu ersparen.
- Das Interface *Distance* erlaubt die Abfrage einer Länge.
- Das Interface *Comparable* ist aus der Vorlesung bekannt und durch Java vorgegeben.

19.2 Funktionalität

19.2.1 Interface *Distance*

R1 Die Schnittstelle deklariert eine abstrakte Methode *getDistance()* zur Rückgabe einer Entfernung.

Deklaration:

```
double getDistance()
```

19.2.2 Klasse *GeoRoute*

R2 Eine Route hat einen Namen sowie eine geordnete Liste von Wegpunkten.

R3 Die Klasse besitzt je einen Konstruktor, dem der Name übergeben wird, sowie einen Konstruktor, dem sowohl der Name als auch eine Liste von Wegpunkten übergeben wird.

R4 Der Name kann über einen Getter erfragt sowie über einen Setter zugewiesen werden.

R5 Die Methode *addWaypoint()* fügt einen Wegpunkt am Ende der Liste hinzu.

R6 Die Methode *removeWaypoint()* entfernt den Wegpunkt mit Index *i* aus der Liste. Beachten Sie, dass das erste Element der Liste den Index 0 hat.

R7 Die Methode *getNumberWaypoints()* gibt die Anzahl der Wegpunkte in der Liste zurück.

R8 Die Methode *getWaypoint()* gibt eine Referenz auf den Wegpunkt mit Index *i* zurück.

R9 Die Methode *getWaypoints()* gibt ein Array aller in der Liste enthaltener Wegpunkte zurück. (Hinweis: Verwenden Sie eine der *toArray()*-Methoden der Klasse *ArrayList*. Wie unterscheiden sich die Methoden? Welche sollten Sie am besten verwenden?)

R10 Die Klasse implementiert das Interface *Distance*. Die Methode *getDistance()* gibt die Gesamtstrecke der Route in Kilometern zurück.

R11 Die Klasse implementiert das Interface *Comparable<GeoRoute>*. Das Kriterium für den Vergleich ist die Gesamtstrecke.

R12 Die Klasse überschreibt die Methode *toString()*. Die Rückgabe hat folgende Formatierung, wobei Ausdrücke in spitzen Klammern durch die entsprechenden Werte zu ersetzen sind: <Name> (<Strecke> km). Die Strecke ist auf eine Stelle hinter dem Komma auszugeben.

Deklarationen:

```
public GeoRoute(String name)
public GeoRoute(String name, ArrayList<GeoPosition> waypoints)
public String getName()
public void setName(String name)
public void addWaypoint(GeoPosition waypoint)
public void removeWaypoint(int index)
public int getNumberWaypoints()
public GeoPosition getWaypoint(int index)
public GeoPosition[] getWaypoints()
public double getDistance()
public int compareTo(GeoRoute other)
public String toString()
```

19.2.3 Klasse *GeoTrack*

- R13 Die Klasse erweitert die Klasse *GeoRoute* um das Datum, an dem die repräsentierte Strecke zurückgelegt wurde. Das Datum wird als Zeichenkette im Format *yyyy-mm-dd* mit Jahr *yyyy*, Monat *mm* und Tag *dd* gespeichert.
- R14 Es existiert genau ein Konstruktor. Diesem werden ein Name der Strecke sowie das Datum übergeben.
- R15 Das Datum kann über einen Getter erfragt sowie über einen Setter zugewiesen werden.

Deklarationen:

```
GeoTrack(String name, String date)
public String getDate()
public void setDate(String date)
```

19.2.4 Unit-Tests

- R16 Stellen Sie sicher, dass alle gegebenen Unit-Tests fehlerfrei durchlaufen. Beachten Sie gegebenenfalls Anhang B.1 auf Seite 149.

19.3 HAW-Lauftreff

Sie wollen einen HAW-Lauftreff ins Leben rufen. Zur Auswahl stehen folgende drei Laufstrecken: von der HAW um die Binnenalster oder die Außenalster und zurück oder aber um den Stadtpark.

Erstellen Sie eine ausführbare Klasse, um die Längen dieser Strecken abzuschätzen. Die Klasse *RouteData* enthält hierfür Methoden, die Strecken um die Binnen- bzw. Außenalster erzeugen. Die Route um den Stadtpark müssen Sie hingegen noch erstellen. Koordinaten der Wegpunkte erhalten Sie beispielsweise, indem Sie in *Google Maps* auf die entsprechenden Positionen klicken. Wählen Sie als Ausgangsort die Verbindung der Ohlsdorfer Straße zur Jahnkampfbahn und fügen Sie einige Koordinaten hinzu, die Sie in einer möglichst großen Runde durch den Stadtpark zurück zum Ausgangspunkt führen. Übertragen Sie die Länge der Routen in die nachfolgende Tabelle.

Route	Länge [km]
Binnenalster	
Außenalster	
Stadtpark	

19.4 Flugrouten

Die Methode *createFlightRoutes()* der Klasse *RouteData* erzeugt eine Liste mit unterschiedlichen Flugrouten. Erstellen Sie eine ausführbare Klasse, die die Liste nach aufsteigender Strecke der Flugroute sortiert und anschließend die Flugrouten mit Namen und Strecke ausgibt. (Hinweis: Eine Liste vom Typ *ArrayList* lässt sich über die Klassenmethode *Collections.sort()* sortieren.)

19.5 Einhaltung der Programmierrichtlinien (Qualität)

Stellen Sie sicher, dass alle in der Checkliste zur Softwarequalität in Anhang C auf Seite 153 aufgeführten Qualitätskriterien erfüllt sind.

19.6 Theoretische Fragen

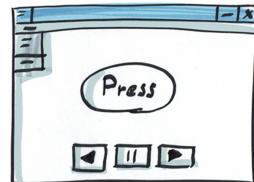
1. Erläutern Sie, warum die Klasse *GeoRoute* zum Referenzieren der Wegpunkte kein Feld, sondern eine Liste verwendet.
2. Könnte *GeoRoute* die Methode *getDistance()* implementieren, ohne das Interface *Distance* zu implementieren?
3. Könnte *GeoRoute* das Interface *Distance* implementieren, ohne die Methode *getDistance()* zu implementieren?
4. Welche Methoden besitzt die Klasse *GeoTrack*?
5. Wie kann ein Objekt der Klasse *GeoTrack* auf die gespeicherten Wegpunkte zugreifen?

Kapitel 20

Praktikum 3: GUI für Wegstrecken

Hinweise

- ▶ Primäre Lernziele: Grafische Benutzeroberflächen (GUI)
- ▶ Organisieren Sie die Quelltexte im Paket *lab3.geoPosition*.



20.1 Überblick

In diesem Praktikum sollen Sie eine grafische Benutzeroberfläche (*Graphical User Interface, GUI*) erstellen. Es gibt keinerlei Vorgaben bezüglich der zu implementierenden Klassen und/oder des genauen Aussehens der Oberfläche. Stattdessen ist ausschließlich von Bedeutung, dass Ihr Programm die in Abschnitt 20.2 geforderte Funktionalität aufweist.

Die nachfolgende Abbildung zeigt als Beispiel eine mögliche Lösung der zu erstellenden Anwendung. Die grafische Oberfläche beinhaltet eine Landkarte, in der Nutzer eine aus mehreren Abschnitten bestehende Wegstrecke festlegen können (hier rot dargestellt von Hamburg nach Kiel). Eine gewählte Route lässt sich wieder löschen. Es steht Ihnen frei, diese Funktionalität z. B. über einen Button und/oder einen Menueintrag zu realisieren. Die Beispilllösung enthält zudem Anzeigen der Gesamtlänge der gewählten Route sowie der aktuellen Mausposition in geografischen Koordinaten.

20.2 Aufgabe

Erstellen Sie eine grafische Benutzeroberfläche mit nachfolgenden Eigenschaften. Beachten Sie, dass nur die Anforderungen aus Abschnitt 20.2.1 umgesetzt werden müssen. Beachten Sie zudem die Hinweise und Tipps in Abschnitt 20.2.3 sowie die Hinweise zur Lösungsstrategie in Abschnitt 20.3.

20.2.1 Pflicht

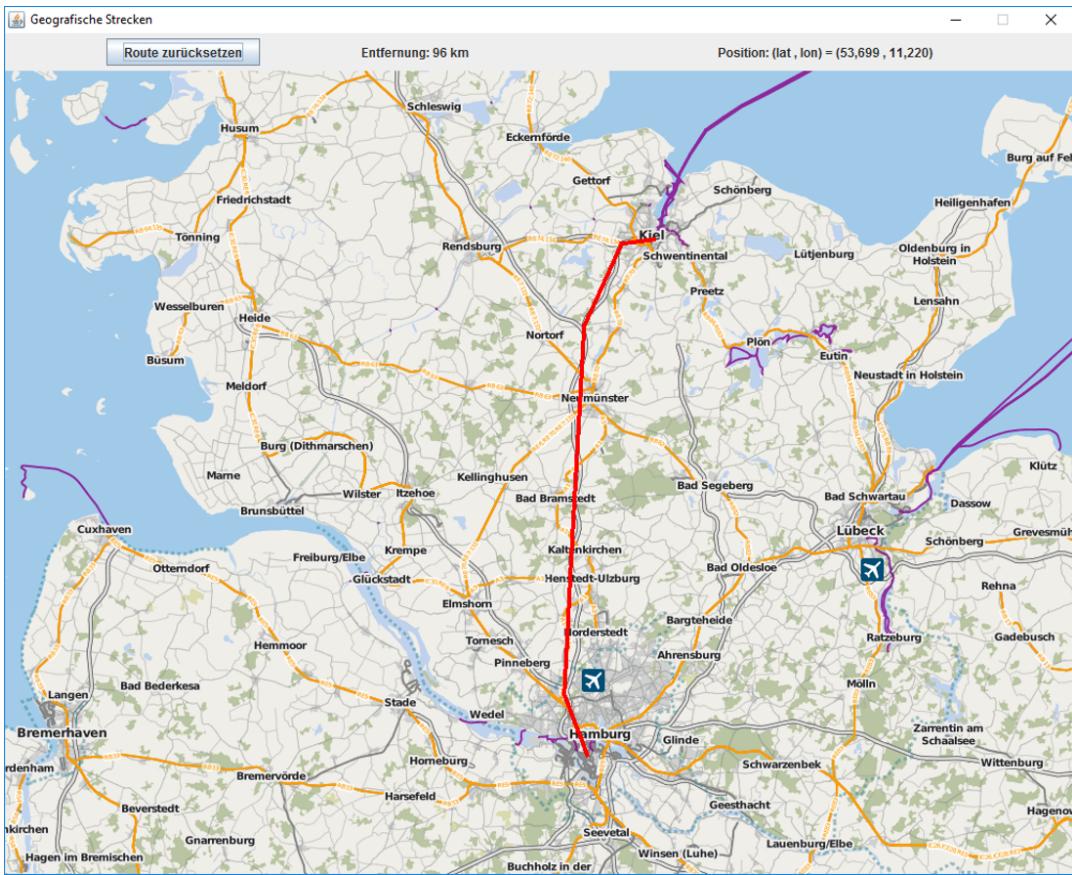
- ▶ Die Anwendung beinhaltet die in der Datei *OSM_Map.png* enthaltene Landkarte¹.
- ▶ Anwender können Wegpunkte setzen, indem sie mit der linken Maustaste auf eine Position innerhalb der Landkarte klicken.
- ▶ Wegpunkte bilden eine Route, wobei neue Punkte an das Streckenende angehängt werden.
- ▶ Die aktuelle Route wird an korrekter Stelle auf der Landkarte dargestellt.
- ▶ Anwender können die aktuelle Route löschen.

20.2.2 Optional

Die nachfolgenden Anforderungen müssen zum Bestehen des Praktikums nicht umgesetzt werden. Sie sind aber eine gute Übung, falls Sie noch etwas Zeit haben und eine kleine Herausforderung suchen.

- ▶ Die Anwendung zeigt die aktuelle Position (x, y) der Maus über der Landkarte an.

¹Freie OSM-Karte (<http://www.openstreetmap.de/>), erzeugt mit MOBAC (<http://mobac.sourceforge.net/>)



- Die Anwendung zeigt die aktuelle Maus-Position in geografische Koordinaten (*lat, lon*) an.
- Die Anwendung zeigt die Länge der Route in Kilometern an.

20.2.3 Hinweise und Tipps

- Landkarte:
 - Lesen Sie die Datei mittels *ImageIO.read()* in ein Objekt des Typs *BufferedImage* und zeichnen Sie dies über *drawImage()*.
 - Eclipse verwendet zum Lesen und Schreiben das Verzeichnis des Java-Projektes. Fügen Sie die Bilddatei *OSM_Map.png* daher beispielsweise per Drag&Drop aus dem Explorer zum Paket *lab3.geoPosition* hinzu und verwenden Sie zum Lesen den relativen Dateipfad *lab3/geoPosition/OSM_Map.png*.
- Mausereignisse: Implementieren Sie einen *MouseListener* und fügen Sie diesen Ihrem Panel über *addMouseListener()* hinzu. Verwenden Sie *MouseMotionListener* und *addMouseMotionListener()*, um auf Mausbewegungen zu reagieren.
- Umrechnung in geografische Koordinaten: Die Datei *OSM_Map.txt* beinhaltet die Breiten- und Längengrade der linken oberen sowie der rechten unteren Ecke der Landkarte.
- Ändern der Linienstärke: Casten Sie die *Graphics*-Referenz auf eine Referenz *Graphics2D*. Ändern Sie die Linienstärke über *setStroke(new BasicStroke(x))*, wobei x der neuen Stärke entspricht.

20.3 Lösungsstrategie

- ▶ Entwerfen und implementieren Sie zunächst das Aussehen der grafischen Oberfläche (d. h. die Art und Anordnung der Elemente).
- ▶ Implementieren Sie erst im zweiten Schritt die geforderte Funktionalität.
- ▶ Verzetteln Sie sich nicht in Details bezüglich des Aussehens der grafischen Oberfläche. (Für besonders kreative und ästhetische Lösungen gibt es Bewunderung und Begeisterung, aber keinen Schönheitspreis.)

20.4 Einhaltung der Programmierrichtlinien (Qualität)

Sie kennen das schon: Stellen Sie sicher, dass alle in der Checkliste zur Softwarequalität in Anhang C auf Seite 153 aufgeführten Qualitätskriterien erfüllt sind.

Bei diesem Versuch bietet es sich zudem an, Ihre Quelltexte von einer anderen Gruppe überprüfen zu lassen und im Gegenzug deren Quelltexte zu lesen. Diskutieren Sie Ihre Eindrücke mit der jeweils anderen Gruppe, insbesondere:

- ▶ War es einfach, die Quelltexte der anderen Gruppe zu verstehen?
- ▶ Was war ausschlaggebend, dass Quelltext verständlich oder nicht gut verständlich ist?
- ▶ Wodurch könnte die Verständlichkeit erhöht werden?

Kapitel 21

Praktikum 4: Wegstrecken (Tracking)

Hinweise

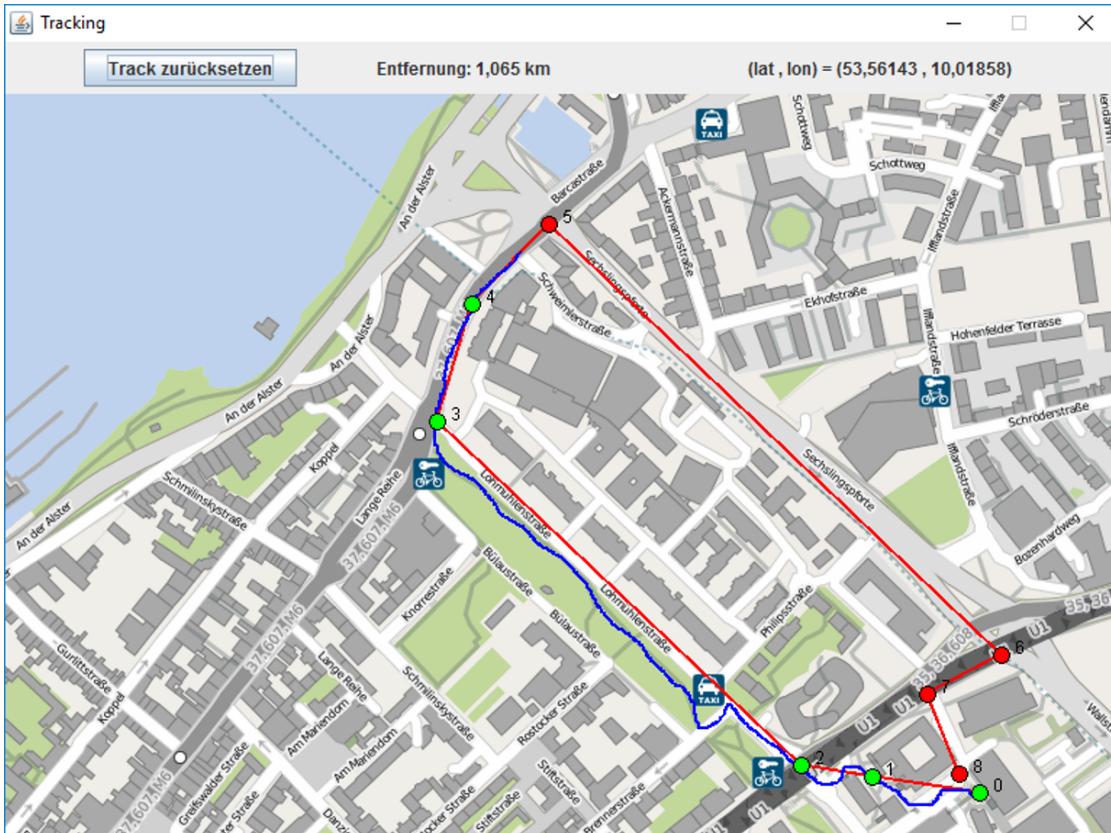
- Primäre Lernziele: Lesen von Textdateien, grafische Benutzeroberflächen (GUI)
- Organisieren Sie die Quelltexte im Paket `lab4.geoPosition`.



21.1 Überblick

Gehen Sie an die frische Luft! – Nun gut, gehen Sie zunächst rein „virtuell“ an die frische Luft. Ihr Weg führt Sie von der HAW entlang des Lohmühlenparks um die Klinik St. Georg.

In diesem Praktikum sollen Sie (beispielsweise beim Joggen oder Radfahren aufgezeichnete) Trackingdaten simulieren, indem Sie die Maus entlang einer Route auf einer Landkarte bewegen. Die Route muss hierzu aus einer Textdatei eingelesen werden. Die nachfolgende Abbildung zeigt als Beispiel eine mögliche Lösung der zu erstellenden Anwendung. In dieser ist die Route mit ihren Wegpunkten rot eingezeichnet, während die virtuell zurückgelegte Strecke blau dargestellt ist. Bereits passierte Wegpunkte sind grün ausgefüllt.



21.2 Aufgabe

Erstellen Sie ein Programm mit nachfolgenden Eigenschaften.

21.2.1 Pflicht

- ▶ Die Anwendung beinhaltet die in der Datei *OSM_BerlinerTor.png* enthaltene Landkarte¹.
- ▶ Die Anwendung liest die in der Datei *Route.txt* gespeicherten Koordinatenpaare ein und stellt die aus diesen Koordinaten bestehende Route auf der Landkarte dar.
- ▶ Anwender können eine Strecke aus Wegpunkten zeichnen, indem sie die Maus mit gedrückter linker Taste über die Karte bewegen.
- ▶ Anwender können die aktuelle Strecke löschen.

21.2.2 Optional

- ▶ Verwenden Sie geografische Koordinaten (*lat, lon*) anstelle der Bildkoordinaten (*x, y*). Die entsprechenden Koordinatenpaare der Route sind in *RouteGeo.txt* gespeichert.
- ▶ Die Anwendung zeigt die Länge der zurückgelegten Wegstrecke in Kilometern an.
- ▶ Die Anwendung erkennt, an welchen Wegpunkten der Route Sie bereits vorbeigekommen sind und stellt diese wie in der Abbildung gezeigt andersfarbig dar.

21.2.3 Hinweise und Tipps

- ▶ Landkarte: Beachten Sie die Hinweise in Praktikumsaufgabe 3 (Darstellung, Dateipfade).
- ▶ Textdatei: Verwenden Sie zum Einlesen der Textzeilen einen *BufferedReader* und trennen Sie die Werte in den Textzeilen über den Methodenaufruf `split(",")`.
- ▶ Bei Mausbewegung mit gedrückter Taste wird die Methode *mouseDragged()* eines verbundenen *MouseMotionListener* aufgerufen.
- ▶ Umrechnung in geografische Koordinaten: Die Datei *OSM_BerlinerTor.txt* beinhaltet die Breiten- und Längengrade der linken oberen sowie der rechten unteren Ecke der Landkarte.
- ▶ Wegpunkte der Route erkennen: Ergänzen Sie z. B. die Klasse *GeoTrack* aus Praktikum 2 um eine Methode, die eine Route übergeben bekommt und die Anzahl der passierten Wegpunkte zurückgibt. Durchlaufen Sie hierzu die Strecke des *GeoTrack*-Objektes und vergleichen Sie zunächst jeweils die Entfernung zum ersten Wegpunkt der Route. Sobald diese „klein genug“ ist (z. B. 25 m), vergleichen Sie die weiteren Streckenpunkte mit dem zweiten Wegpunkt der Route bis entweder die gesamte Route oder Trackingstrecke durchlaufen wurde.
- ▶ Beachten Sie auch die Hinweise und Tipps für Praktikum 3.

¹Freie OSM-Karte (<http://www.openstreetmap.de/>), erzeugt mit MOBAC (<http://mobac.sourceforge.net/>)

Teil IV

Praktika: Spiel des Lebens (Game of Life)

Kapitel 22

Praktikum 3: GUI (Teil 1)

Hinweise

- ▶ Primäre Lernziele: Grafische Benutzeroberflächen (GUI)
- ▶ Organisieren Sie die Quelltexte im Paket `lab3_4.gameOfLife`.



22.1 Überblick

Konsolenausgaben sind langweilig und das Leben ist bunt. Also lassen Sie uns ein Programm mit einer grafischen Benutzeroberfläche (*Graphical User Interface, GUI*) erstellen. Zudem wollen wir eine etwas komplexere Software-Struktur als in den vorherigen Praktika umsetzen. Da dies den Rahmen eines einzigen Terms sprengen würde, legen wir in diesem Praktikum den Grundstein und werden das Programm in Praktikum 4 vervollständigen.

22.1.1 Spiel des Lebens

Als Anwendung ist Conways *Spiel des Lebens*¹ umzusetzen. Hierbei besteht das Spielfeld aus einer Matrix von Zellen, die entweder den Zustand *lebendig* oder *besitzen. Ausgehend von einer Anfangspopulation von lebendigen Zellen wird in jedem Schritt die jeweils nächste Generation von lebendigen Zellen bestimmt. Ob eine Zelle der nächsten Generation lebendig oder tot ist, hängt davon ab, wie viele lebendige Zellen angrenzen. Da die Umsetzung der Spiellogik in Praktikum 4 erfolgt, werden die genauen Regeln erst in der zugehörigen Aufgabenstellung beschrieben.*

22.1.2 Vorgaben und Beispieldlösung

Es gibt keinerlei Vorgaben bezüglich der zu implementierenden Klassen und/oder des genauen Aussehens der Oberfläche. Stattdessen ist ausschließlich von Bedeutung, dass Ihr Programm die in Abschnitt 22.2 geforderte Funktionalität aufweist.

Abbildung 22.1 zeigt als Beispiel eine mögliche Lösung der zu erstellenden Anwendung. Beachten Sie, dass diese auch Elemente beinhaltet, der erst im Praktikum 4 umzusetzen sind.

- ▶ Die grafische Oberfläche beinhaltet im oberen Bereich eine Darstellung der aktuellen Population, wobei lebendige Zellen durch schwarze Quadrate repräsentiert werden.
- ▶ Es lässt sich über einen Button eine neue Ausgangspopulation erzeugen. Weitere Buttons dienen der Generierung der folgenden Generationen.
- ▶ Es wird über ein Textfeld ausgegeben, um die wievielte Generation es sich handelt. Zudem wird die Anzahl lebender Zellen in dieser Generation angezeigt.
- ▶ Im unteren Bereich wird der Verlauf der Anzahl lebender Zellen von Generation zu Generation grafisch veranschaulicht.

¹https://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens (Abgerufen am 22.03.2020)

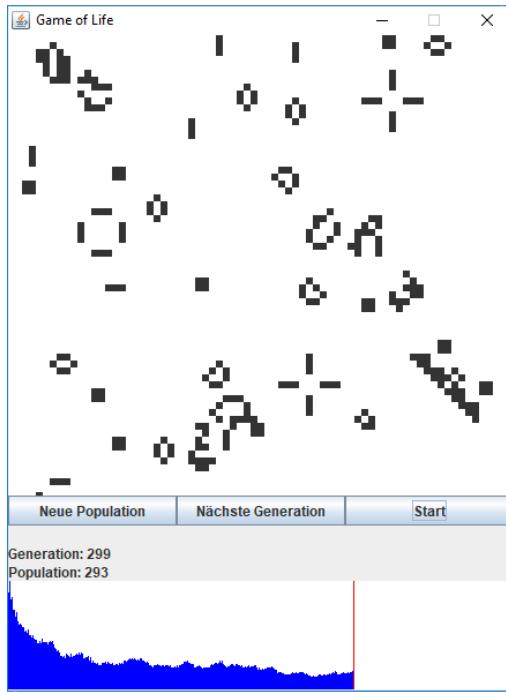


Abbildung 22.1: Beispielumsetzung (inklusive Praktikum 4)

22.2 Aufgabe

Erstellen Sie eine grafische Benutzeroberfläche mit nachfolgenden Eigenschaften. Beachten Sie, dass nur die Anforderungen aus Abschnitt 22.2.1 umgesetzt werden müssen. Beachten Sie zudem die Hinweise zur Lösungsstrategie in Abschnitt 22.3.

22.2.1 Pflicht

- ▶ Die Anwendung stellt eine Matrix von mindestens 50×50 Zellen des Spiels des Lebens dar. Lebendige Zellen entsprechen hierbei einem ausgefüllten Quadrat von mindestens 6×6 Pixeln.
- ▶ Anwender können eine neue zufällige Population erzeugen. Die Wahrscheinlichkeit, mit der eine Zelle hierbei den Zustand *lebendig* erhält, ist in der Anwendung festgelegt (z. B. 20%).
- ▶ Die absolute Anzahl lebendiger Zellen der Population ist in der grafischen Oberfläche ersichtlich.

22.2.2 Optional

Die nachfolgenden Anforderungen müssen nicht umgesetzt werden. Sie sind aber eine gute Übung, falls Sie noch etwas Zeit haben und eine kleine Herausforderung suchen.

- ▶ Anwender können die Wahrscheinlichkeit, mit der jede Zelle bei der Erzeugung einer Zufallspopulation den Zustand *lebendig* erhält, auswählen bzw. angeben.
- ▶ Anwender können beispielsweise über einen Button die jeweils nächste Generation erzeugen. Die Regeln zur Erzeugung der nächsten Generation können Sie hierbei selbst vorgeben (z. B. Matrix invertieren oder jeweils eine lebendige Zelle entfernen).

- Anwender können den Zustand einer Zelle durch Anklicken mit der Maus ändern. (Hinweis: Implementieren Sie hierfür einen *MouseListener* und fügen Sie diesen Ihrem Panel über *addMouseListener()* hinzu.)

22.3 Lösungsstrategie

- Entwerfen und implementieren Sie zunächst das Aussehen der grafischen Oberfläche (d. h. die Art und Anordnung der Elemente).
- Implementieren Sie erst im zweiten Schritt die geforderte Funktionalität.
- Verzetteln Sie sich nicht in Details bezüglich des Aussehens der grafischen Oberfläche. (Für besonders kreative und ästhetische Lösungen gibt es Bewunderung und Begeisterung, aber keinen Schönheitspreis.)

22.4 Einhaltung der Programmierrichtlinien (Qualität)

Sie kennen das schon: Stellen Sie sicher, dass alle in der Checkliste zur Softwarequalität in Anhang C auf Seite 153 aufgeführten Qualitätskriterien erfüllt sind.

Bei diesem Versuch bietet es sich zudem an, Ihre Quelltexte von einer anderen Gruppe überprüfen zu lassen und im Gegenzug deren Quelltexte zu lesen. Diskutieren Sie Ihre Eindrücke mit der jeweils anderen Gruppe, insbesondere:

- War es einfach, die Quelltexte der anderen Gruppe zu verstehen?
- Was war ausschlaggebend, dass Quelltext verständlich oder nicht gut verständlich ist?
- Wodurch könnte die Verständlichkeit erhöht werden?

Kapitel 23

Praktikum 4: Logik & Threads (Teil 2)

Hinweise

- ▶ Primäre Lernziele: Grafische Benutzeroberflächen (GUI), Threads
- ▶ Dieses Praktikum ist eine Fortführung des Praktikums 3 zum *Spiel des Lebens*.



23.1 Einleitung

Im zugehörigen Praktikum 3 haben Sie grundlegende Elemente der grafischen Benutzeroberfläche (*Graphical User Interface, GUI*) erstellt. In diesem Praktikum sollen Sie die Spiellogik und weitere grafische Elemente hinzufügen. Zudem erhalten Sie Hinweise bezüglich einer sinnvollen Klassenstruktur bzw. Architektur.

23.1.1 Spielregeln

Die grundlegende Idee des Spiels des Lebens wurde bereits im letzten Praktikum erläutert:

1. Erzeugung einer Ausgangspopulation *lebender* und *toter* Zellen
2. Iterative Bestimmung der jeweils nächsten Generation anhand von vorgegebenen Regeln

Was bislang noch fehlt, sind die Regeln bei der Erzeugung der nächsten Generation. Hierfür werden für jede Zelle die acht direkt angrenzenden Nachbarn betrachtet¹:

- ▶ Eine lebende Zelle überlebt, wenn sie zwei oder drei lebenden Nachbarn besitzt. Ansonsten stirbt sie an Einsamkeit bzw. Überbevölkerung.
- ▶ Aus einer toten Zelle mit genau drei lebenden Nachbarn entsteht eine lebende Zelle.

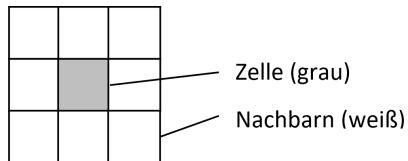


Abbildung 23.1: Zellen mit benachbarten Zellen

23.1.2 Vorgaben und Beispiellösung

Es gibt erneut keinerlei Vorgaben bezüglich der zu implementierenden Klassen und/oder des genauen Aussehens der Oberfläche. Stattdessen ist ausschließlich von Bedeutung, dass Ihr Programm die in Abschnitt 23.2 geforderte Funktionalität aufweist. Die in der Aufgabenbeschreibung zu Praktikum 3 dargestellte und erläuterte Beispiellösung behält ihre Gültigkeit.

¹https://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens#Die_Spielregeln (Abgerufen am 22.03.2020)

23.2 Aufgabe

Erweitern Sie das im Praktikum 3 erstellte Programm um nachfolgende Eigenschaften. Beachten Sie, dass nur die Anforderungen aus Abschnitt 23.2.1 umgesetzt werden müssen. Beachten Sie zudem die Hinweise in Abschnitt 23.3.

23.2.1 Pflicht

- ▶ Anwender können die jeweils nächste Generation gemäß den aufgestellten Spielregeln erzeugen.
- ▶ Anwender können eine automatische fortlaufende Erzeugung der jeweils nächsten Generation starten und wieder beenden. Während dieser wird jeweils nach einem bestimmten Zeitintervall (z. B. 100 ms) die nächste Generation erzeugt. Die Evolution der Generation lässt sich also quasi als „Video“ betrachten.

23.2.2 Optional

Die nachfolgenden Anforderungen müssen nicht umgesetzt werden. Sie sind aber eine gute Übung, falls Sie noch etwas Zeit haben und eine kleine Herausforderung suchen.

- ▶ Zum Starten und Beenden der fortlaufenden automatischen Erzeugung von Generationen wird derselbe Button verwendet. Er ist je nach Zustand beschriftet (z. B. „Start“ bzw. „Stopp“).
- ▶ Während der automatischen Erzeugung von Generationen kann nur der Button zum Beenden angewählt werden. Alle anderen Steuerelemente (z. B. Button zur Erzeugung einer neuen Zufallspopulation) sind ausgegraut.
- ▶ Anwender können das Zeitintervall für die automatische Erzeugung von Generationen auswählen bzw. angeben.
- ▶ Die Anzahl der lebenden Zellen im Verlauf der Generationen wird grafisch angezeigt (siehe Abbildung in Beschreibung Praktikum 3).

23.3 Hinweise

23.3.1 Pflicht

Implementierung der Regeln:

- ▶ Obwohl die Zellen Boolesche Zustände besitzen („ist lebend“ ist wahr oder falsch), sollten Sie diese durch Ganzzahlen darstellen, wobei eine lebende Zelle den Wert 1 und eine tote den Wert 0 erhält. Die Anzahl lebender Nachbarn entspricht der Summe der angrenzenden Zellen.
- ▶ Erstellen Sie bei der Erzeugung der nächsten Generation zunächst eine Kopie der aktuellen Population. Bestimmen Sie in einer der Matrizen für jedes Element die Summe der lebenden Nachbarn und schreiben Sie das Resultat (*lebendig* = 1 bzw. *tot* = 0) in die zweite Matrix.
- ▶ Implementieren Sie die Bestimmung der Summe lebender Nachbarn in einer eigenen Methode.

Fortlaufende Erzeugung von Generationen:

- ▶ Erstellen Sie einen Thread, der in einer Schleife die nächste Generation erzeugt und dann über *sleep()* das vorgegebene Intervall wartet.

- ▶ Fügen Sie eine Variable hinzu, deren Wert darüber bestimmt, ob die Schleife erneut durchlaufen wird. Beim Drücken des „Stopp“-Buttons sollte der Wert dieser Variable geändert werden, wodurch kein neuer Schleifendurchlauf stattfindet und die `run()`-Methode beendet wird.

23.3.2 Architektur (optional)

Grundlegende Struktur

Im Kapitel über grafische Oberflächen wurde *Model-View-Control (MVC)* eingeführt. Hieran angelehnt sollten Sie folgende drei Klassen erstellen (Abb. 23.2):

- ▶ *GameModel* beinhaltet die Daten des Spiels, insbesondere das 2D-Feld der Zellen sowie ein Zähler, um die wievielte Generation es sich handelt. Hinzu kommen Methoden wie z. B. die Initialisierung mit einer Zufallspopulation und die Erzeugung der jeweils nächsten Generation.
- ▶ *GameView* beinhaltet die Darstellung die einer grafischen Oberfläche. Für die Zeichenflächen muss zudem jeweils die Klasse *JPanel* abgeleitet und die Methode `paintComponent()` entsprechend überlagert werden.
- ▶ *GameOfLife (Control)* beinhaltet die `main()`-Methode und steuert den grundlegenden Programmfluss. Hier werden Objekte *GameModel* und *GameView* erzeugt und der *ActionListener* für gedrückte Buttons implementiert. Dies beinhaltet unter anderem auch die Erzeugung der nächsten Generation über das *GameModel*-Objekt sowie die Erzeugung und Starten bzw. Stoppen des Threads.

Aktualisierung der grafischen Oberfläche (*Observer Pattern*)

Es stellt sich die Frage, wie erreicht werden kann, dass die GUI stets die aktuellen Daten darstellt. Eine elegante Lösung ist die Folgende:

- ▶ Das *GameModel*-Objekt enthält eine Instanzvariable mit der Referenz auf das *GameView*-Objekt. Umgekehrt enthält auch *GameView* eine Instanzvariable mit der Referenz auf *GameModel*.
- ▶ Wenn sich die Daten in *GameModel* ändern (z. B. nächste Generation erzeugt), benachrichtigt es *GameView* durch den Aufruf einer speziellen Methode (z. B. `update()`). In dieser Methode holt sich *GameView* die aktuellen Daten von *GameModel* und stellt diese in der GUI dar.
- ▶ Architektonisch sauber würde man hierfür wie im Klassendiagramm dargestellt eine Klasse *Subject* sowie ein Interface *Observer* verwenden. Dies ist aber nicht zwingend erforderlich.

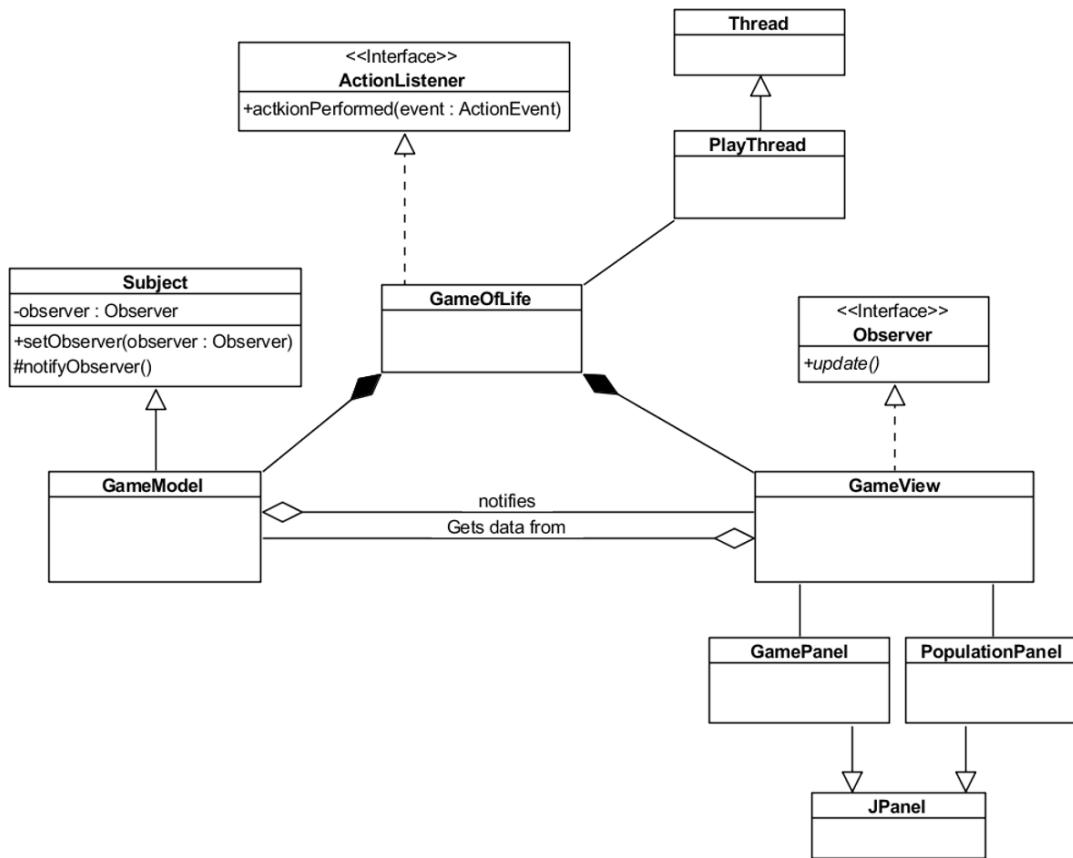


Abbildung 23.2: Übersicht der Klassen und Interfaces

Teil V

Praktika: Sonstige

Praktikum 1: Komplexe Zahlen und Schwingkreis

Primäre Lernziele

- ▶ Einfache Klassen mit Variablen und Methoden
- ▶ Ausführbare Klassen
- ▶ Statische Elemente

24.1 Klasse *Complex*

Es ist eine Klasse *Complex* zur Repräsentation von komplexen Zahlen $z \in \mathbb{C}$ zu erstellen. Eine derartige Klasse könnte unter anderem dazu verwendet werden, um elektrische Kapazitäten und Impedanzen zu beschreiben und Netzwerke aus diesen Bauelementen automatisch zu berechnen. Die Klasse wird grundlegend durch das nebenstehende UML-Symbol beschrieben. In dem Kasten unterhalb des Klassennamens sind die Attribute mit jeweiligem Datentyp angegeben. Durch ein Minuszeichen wird gekennzeichnet, dass der Modifier *private* zu verwenden ist.

Im unteren Abschnitt sind die Methoden mit Parameterlisten sowie nach dem Doppelpunkt dem Datentyp ihres Rückgabewertes aufgelistet.

Complex
- real: double
- imag: double
+ Complex()
+ Complex(real: double, imag: double)
+ getReal(): double
+ getImag(): double
+ add(z: Complex): Complex
+ subtract(z: Complex): Complex
+ multiply(z: Complex): Complex
+ divide(z: Complex): Complex
+ getAbsolute(): double
+ getPhase(): double
+ add(z1: Complex, z2: Complex): Complex
+ subtract(z1: Complex, z2: Complex): Complex
+ multiply(z1: Complex, z2: Complex): Complex
+ divide(z1: Complex, z2: Complex): Complex

24.1.1 Benötigte Formeln

Für komplexe Zahlen $z = x + jy \in \mathbb{C}$ gilt mit $j^2 = -1$ und dem konjugiert Komplexen $\bar{z} = x - jy$:

- ▶ Addition und Subtraktion:

$$z_1 \pm z_2 = (x_1 \pm x_2) + j \cdot (y_1 \pm y_2)$$

- ▶ Multiplikation:

$$z_1 \cdot z_2 = (x_1 x_2 - y_1 y_2) + j \cdot (x_1 y_2 + x_2 y_1)$$

- ▶ Division:

$$\frac{z_1}{z_2} = \frac{z_1}{z_2} \cdot \frac{\bar{z}_2}{\bar{z}_2} = \frac{z_1 \bar{z}_2}{|z_2|^2} = \frac{x_1 x_2 + y_1 y_2}{x_2^2 + y_2^2} + j \cdot \frac{x_2 y_1 - x_1 y_2}{x_2^2 + y_2^2}$$

- ▶ Betrag:

$$|z| = \sqrt{x^2 + y^2}$$

24.1.2 Eclipse und Projektstruktur

- a) Erstellen Sie in Eclipse ein Java-Projekt namens *Praktikum*, wobei Sie unter „*Project layout*“ die Option „*Use project folder as root for sources and class files*“ wählen. Erzeugen Sie im Projekt das Paket *lab1.complex*.
- b) Fügen Sie dem Projekt die aktuellste *JUnit*-Bibliothek hinzu.

- c) Fügen Sie zum Paket die gegebene Test-Klasse *TestComplex* hinzu. Sie können hierfür die Datei *TestComplex.java* aus dem Windows-Explorer per „Drag&Drop“ in das Paket im Eclipse *Package Explorer* ziehen.

24.1.3 Attribute und Konstruktoren

- R1 Die Klasse besitzt private Variablen namens *real* und *imag* zur Speicherung des Real- und Imaginärteils. Beide Variablen sind vom Typ *double*.
- R2 Es gibt einen Konstruktor ohne Parameter (*Standardkonstruktor*), der die beiden Attribute mit dem Wert 0.0 vorbelegt.
- R3 Es gibt einen Konstruktor mit zwei *double*-Parametern, der den Attributen die übergebenen Parameterwerte zuweist. (Tipp: Nutzen Sie in Eclipse den Menüpunkt „*Source / Generate ...*“, um die Methode automatisch erzeugen zu lassen.)

Deklarationen:

```
public Complex()
public Complex(double real, double imag)
```

24.1.4 Getter und arithmetische Operatoren

- R4 Es gibt Getter-Methoden, die den Wert von *real* bzw. *imag* zurückgeben. (Lassen Sie auch diese Methoden automatisch in Eclipse erzeugen.)
- R5 Es gibt Methoden zur Addition, Subtraktion, Multiplikation und Division zweier komplexer Zahlen. Diesen wird jeweils ein Parameter *z* vom Typ *Complex* übergeben. Der erste Operand der jeweiligen Operation ist das Objekt, für das die Methode aufgerufen wird. Der zweite Operand ist die beim Aufruf übergebene Zahl *z*. Das Ergebnis wird im Objekt, für das die Methode aufgerufen wird, gespeichert. Der Rückgabewert ist eine Referenz auf dieses Objekt.

Deklarationen:

```
public double getReal()
public double getImag()
public Complex add(Complex z)
public Complex subtract(Complex z)
public Complex multiply(Complex z)
public Complex divide(Complex z)
```

24.1.5 Betrag und Phase

- R6 Es gibt eine Methode zur Berechnung des Betrages der komplexen Zahl. (Hinweis: Verwenden Sie zur Berechnung der Quadratwurzel die Klasse *Math*.)
- R7 Es gibt eine Methode zur Berechnung der Phase (*Winkel*) der komplexen Zahl. Die Phase wird im Bereich $[0, 2\pi]$ zurückzugeben. Für $z = 0 + 0j$ wird die Phase als $\angle z = 0$ definiert.

Hinweise zur Phase:

- ▶ Mathematisch berechnet sich die Phase über $z = \text{atan}(\frac{y}{x})$, wobei je nach Quadrant gegebenenfalls π oder 2π addiert werden muss.
- ▶ Betrachten Sie die Methoden *atan()* und *atan2()* der Klasse *Math*. Worin unterscheiden sich die beiden Methoden? Welche ist für die Berechnung der Phase geeigneter?

Deklarationen:

```
public double getAbsolute()
public double getPhase()
```

24.1.6 Klassenmethoden für arithmetische Operationen

R8 Es gibt Klassenmethoden zur Addition, Subtraktion, Multiplikation und Division zweier komplexer Zahlen. Diesen werden beide Operanden als Parameter vom Typ *Complex* übergeben. Das Ergebnis der Operation wird in einer neu erzeugten komplexen Zahl gespeichert. Der Rückgabewert ist eine Referenz auf das neue Objekt.

Deklaration:

```
public static Complex add(Complex z1, Complex z2)
public static Complex subtract(Complex z1, Complex z2)
public static Complex multiply(Complex z1, Complex z2)
public static Complex divide(Complex z1, Complex z2)
```

24.1.7 Methode *toString()*

R9 Die Klasse besitzt eine Methode *toString()*, die eine wie folgt formatierte Zeichenkette zurückgibt: (*<real>* + *<imag>**j) bzw. (*<real>* - *<imag>**j)

Deklaration:

```
public String toString()
```

24.1.8 Ausführbare Klasse und Unit-Tests

R10 Erstellen Sie eine ausführbare Klasse *ExampleComplex*. Probieren Sie in *main()* die Methoden der Klasse *Complex* an eigenen Zahlenbeispielen aus und geben Sie die Ergebnisse auf Konsole aus.

R11 Stellen Sie sicher, dass alle gegebenen Unit-Tests fehlerfrei durchlaufen. Beachten Sie gegebenenfalls Anhang B.1 auf Seite 149.

24.2 Schwingkreis

Erstellen Sie unter Verwendung der Klasse *Complex* ein Programm, das für den Reihenschwingkreis in Abbildung 24.1 gemäß nachfolgenden Anforderungen die Spannung U_R am Widerstand R berechnet und in einem Frequenzbereich ausgibt. Formeln:

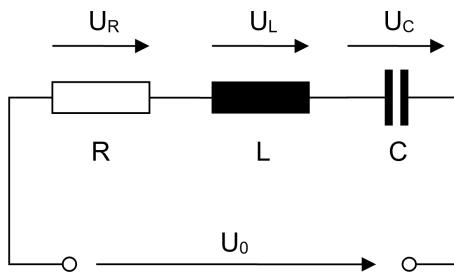


Abbildung 24.1: Reihenschwingkreis

► Impedanzen:

$$Z_R = R, \quad Z_L = j\omega L, \quad Z_C = \frac{1}{j\omega C}$$

► Gesamtimpedanz:

$$Z = Z_R + Z_L + Z_C = R + j\omega L + \frac{1}{j\omega C}$$

- Spannung am Widerstand R :

$$U_R = R \cdot \frac{U_0}{Z}$$

R12 Erstellen Sie eine ausführbare Klasse *ResonantCircuit*. Die Klasse wird durch die nachfolgenden Anforderungen weiter spezifiziert.

R13 Die Zahlenwerte für den Widerstand $R = 10 \Omega$, die Induktivität $L = 10 \text{ mH}$, die Kapazität $C = 10 \text{ nF}$ sowie die Eingangsspannung $U_0 = 1 \text{ V}$ werden als konstante Klassenvariablen gespeichert.

R14 Die statische Methode *voltageOverResistor()* gibt für eine bestimmte Kreisfrequenz ω die komplexe Spannung U_R über dem Widerstand R zurück. Legen Sie hierfür in der Methode zunächst für die Impedanzen Z_R , Z_L und Z_C sowie die Eingangsspannung U_0 Variablen des Typs *Complex* an. Berechnen Sie die Gesamtimpedanz Z sowie anschließend U_R durch Verwendung der arithmetischen Methoden (z. B. *add()*) der Klasse *Complex*.

R15 Die *main()*-Methode gibt die Spannung U_R im Bereich von 10 kHz bis 100 kHz aus. Die Schrittweite beträgt hierbei 10 kHz. Erstellen Sie hierzu zunächst ein Array, das alle Kreisfrequenzen ω enthält. Berechnen Sie anschließend die zugehörigen Spannungen U_R und geben Sie die Beträge $|U_R|$ auf Konsole aus.

24.3 Einhaltung der Programmierrichtlinien (Qualität)

Stellen Sie sicher, dass alle in der Checkliste zur Softwarequalität in Anhang C auf Seite 153 aufgeführten Qualitätskriterien erfüllt sind.

Kapitel 25

Praktikum 1: 2D-Vektoren und Flugplan

Primäre Lernziele

- ▶ Einfache Klassen mit Variablen und Methoden
- ▶ Ausführbare Klassen
- ▶ Statische Elemente

25.1 Flugplan

Sie möchten mit einem Flugzeug bei einem Rundflug die in Abbildung 25.1 dargestellte Route fliegen. Die Tankfüllung reicht für eine Flugstrecke von 200 km. Schreiben Sie ein Programm zur Beantwortung der Frage, ob Sie ausreichend Kerosin haben, um vom Punkt B wieder sicher zum Flugplatz zurückzukehren. Verwenden Sie mathematische Vektoren, um die einzelnen Abschnitte Ihres geplanten Rundfluges darzustellen. Hoppla ... eine Vektor-Klasse mit den zugehörigen Vektor-Operationen kennen wir ja noch gar nicht. Also lassen Sie uns eine schreiben!

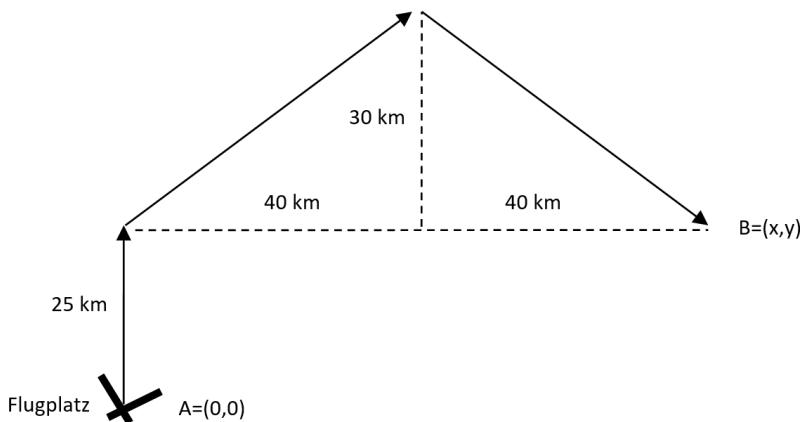


Abbildung 25.1: Geplante Flugroute

25.2 Klasse *Vector2D*

Es ist eine Klasse *Vector2D* zur Repräsentation eines zweidimensionalen Vektors $\mathbf{a} = (x, y)^T$ zu erstellen. Die Klasse wird grundlegend durch das nebenstehende UML-Symbol beschrieben. In dem Kasten unterhalb des Klassennamens sind die Attribute mit jeweiligem Datentyp angegeben. Durch ein Minuszeichen wird gekennzeichnet, dass der Modifier *private* zu verwenden ist.

Im unteren Abschnitt sind die Methoden mit Parameterlisten sowie nach dem Doppelpunkt dem Datentyp ihres Rückgabewertes aufgelistet. Unterstrichene Elemente sind Klassenmethoden.

Vector2D
-x: double
-y: double
+Vector2D(double, double)
+getX(): double
+getY(): double
+add(Vector2D): Vector2D
+subtract(Vector2D): Vector2D
+multiply(double): Vector2D
+getAbsolute(): double
+multiply(Vector2D, Vector2D): double
+toString(): String
+equals(Object): boolean

25.2.1 Benötigte Formeln

Für Vektoren $\mathbf{a} = (x, y)^T \in \mathbb{R}^2$ und Skalare $c \in \mathbb{R}$ gilt:

- Addition und Subtraktion:

$$\mathbf{a}_1 \pm \mathbf{a}_2 = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \pm \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_1 \pm x_2 \\ y_1 \pm y_2 \end{pmatrix} \in \mathbb{R}^2$$

- Skalare Multiplikation:

$$c \cdot \mathbf{a} = c \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} cx \\ cy \end{pmatrix} \in \mathbb{R}^2$$

- Skalarprodukt:

$$\mathbf{a}_1 \cdot \mathbf{a}_2 = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \cdot \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = x_1 x_2 + y_1 y_2 \in \mathbb{R}$$

- Betrag:

$$|\mathbf{a}| = \left| \begin{pmatrix} x \\ y \end{pmatrix} \right| = \sqrt{x^2 + y^2} \in \mathbb{R}$$

25.2.2 Eclipse und Projektstruktur

- a) Erstellen Sie in Eclipse ein Java-Projekt namens *Praktikum*, wobei Sie unter „*Project layout*“ die Option „*Use project folder as root for sources and class files*“ wählen. Erzeugen Sie im Projekt das Paket *lab1.vector*.
- b) Fügen Sie dem Projekt die aktuellste *JUnit*-Bibliothek hinzu.
- c) Fügen Sie zum Paket die gegebene Test-Klasse *TestVector2D* hinzu. Sie können hierfür die Datei *TestVector2D.java* aus dem Windows-Explorer per „*Drag&Drop*“ in das Paket im Eclipse *Package Explorer* ziehen.

25.2.3 Attribute, Konstruktoren und Getter

- R1 Die Klasse besitzt private Variablen namens *x* und *y* zur Speicherung der Zahlenwerte der Komponenten. Beide Variablen sind vom Typ *double*.
- R2 Es gibt einen Konstruktor mit zwei *double*-Parametern, der den Attributen die übergebenen Parameterwerte zuweist. (Tipp: Nutzen Sie in Eclipse den Menüpunkt „*Source / Generate ...*“, um die Methode automatisch erzeugen zu lassen.)
- R3 Es gibt Getter-Methoden, die den Wert von *x* bzw. *y* zurückgeben. (Lassen Sie auch diese Methoden automatisch in Eclipse erzeugen.)

Deklarationen:

```
public Vector2D(double x, double y)
public double getX()
public double getY()
```

25.2.4 Arithmetische Operatoren und Betrag

- R4 Es gibt Methoden zur Addition und Subtraktion zweier Vektoren sowie zur Multiplikation mit einem skalaren Zahlenwert. Das Ergebnis wird in dem Objekt, für das die Methode aufgerufen wird, gespeichert. Der Rückgabewert ist eine Referenz auf dieses Objekt.
- R5 Es gibt eine Methode zur Berechnung des Betrages des Vektors. (Hinweis: Verwenden Sie zur Berechnung der Quadratwurzel die Klasse *Math*.)

Deklarationen:

```
public Vector2D add(Vector2D a)
public Vector2D subtract(Vector2D a)
public Vector2D multiply(double c)
public double getAbsolute()
```

25.2.5 Klassenmethode

R6 Es gibt eine Klassenmethode zur Berechnung des Skalarprodukts zweier Vektoren.

Deklaration:

```
public static double multiply(Vector2D a1, Vector2D a2)
```

25.2.6 Methode *toString()*

R7 Die Klasse besitzt eine Methode *toString()*, die eine wie folgt formatierte Zeichenkette zurückgibt: (<x>, <y>)^T. (Tipp: Lassen Sie das Grundgerüst dieser Methode automatisch in Eclipse erzeugen.)

Deklaration:

```
public String toString()
```

25.2.7 Ausführbare Klasse und Unit-Tests

R8 Erstellen Sie eine ausführbare Klasse *ExampleVector2D*. Probieren Sie in *main()* die Methoden der Klasse *Vector2D* an eigenen Zahlenbeispielen aus und geben Sie die Ergebnisse auf Konsole aus.

R9 Stellen Sie sicher, dass alle gegebenen Unit-Tests fehlerfrei durchlaufen. Beachten Sie gegebenenfalls Anhang B.1 auf Seite 149.

25.3 Flugplan „Reloaded“

Lassen Sie uns nun zum Flugplan aus Abschnitt 25.1 zurückkehren. Erstellen Sie eine ausführbare Klasse *FlightPlan*. Die Klasse wird durch die nachfolgenden Anforderungen weiter spezifiziert:

R10 Die Klasse stellt in der *main()*-Methode die einzelnen Etappen des Fluges als Vektoren der Klasse *Vector2D* dar.

R11 Die Klasse berechnet in *main()* durch Operationen der Klasse *Vector2D* folgende Daten, die zudem auf der Konsole ausgegeben werden:

1. Koordinaten des Punktes *B*
2. Beim Flug vom Flugplatz *A* nach *B* insgesamt zurückgelegte Strecke
3. Entfernung des Punktes *B* vom Flugplatz *A*
4. Beim Rundflug insgesamt zurückzulegende Strecke

Beantworten Sie die in Abschnitt 25.1 gestellte Frage, ob die Tankfüllung ausreicht, um bei diesem Rundflug wieder von *B* zum Flughafen *A* zurückzukehren. Anders ausgedrückt: Würden Sie in das Flugzeug einsteigen und mitfliegen?

25.4 Einhaltung der Programmierrichtlinien (Qualität)

Stellen Sie sicher, dass alle in der Checkliste zur Softwarequalität in Anhang C auf Seite 153 aufgeführten Qualitätskriterien erfüllt sind.

25.5 Theoretische Fragen

1. Welche Auswirkungen hat das Schlüsselwort *static* auf Instanzvariablen?
2. Welche Auswirkungen hat das Schlüsselwort *static* auf lokale Variablen?
3. Welche Auswirkungen hat das Schlüsselwort *static* auf Methoden?

Kapitel 26

Praktikum 2: Kalender

Primäre Lernziele

- ▶ Primäre Lernziele: Listen, Interfaces, Vererbung
- ▶ Organisieren Sie die Quelltexte im Paket `lab2.calendar`.
- ▶ Alle Instanzvariablen besitzen den Modifier `private`.
- ▶ Die JUnit-Tests müssen fehlerfrei durchlaufen.
- ▶ Erzeugen Sie Konstruktoren, Getter, Setter und `toString()` durch die Menüpunkte „*Source/Generate ...*“ und passen Sie diese gegebenenfalls den Anforderungen an.

26.1 Überblick

In diesem Praktikum sollen Sie ein sehr einfaches Kalendersystem mit mehreren Nutzern erstellen. Lassen Sie uns hierzu zunächst anhand des UML-Diagramms in Abbildung 26.1 veranschaulichen, aus welchen Klassen und Interfaces das System besteht:

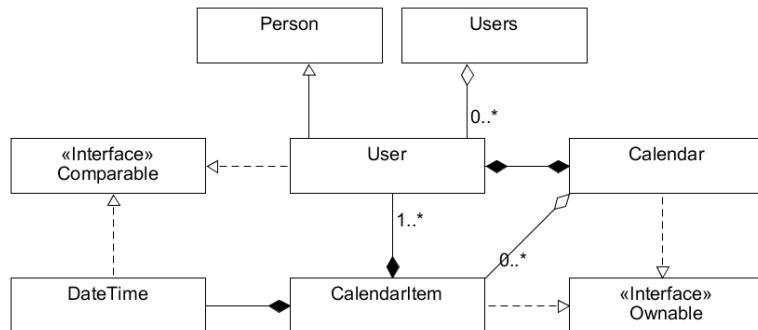


Abbildung 26.1: Klassendiagramm

- ▶ Ein durchgezogener Pfeil zeigt hierbei zur Basisklasse, während ein gestrichelter Pfeil zu einem implementierten Interface zeigt.
- ▶ Eine ausgefüllte Raute bedeutet, dass ein Objekt der Klasse, an der die Raute gezeichnet ist, aus einem oder mehreren Objekten der anderen Klasse zusammengesetzt ist (*Komposition*). So hat z. B. jeder Kalendereintrag (*CalendarItem*) einen Zeitpunkt vom Typ *DateTime*.
- ▶ Eine leere Raute bedeutet, dass ein Objekt der anderen Klasse enthalten sein kann. So kann ein Kalender (*Calendar*) Termine (*CalendarItem*) enthalten – muss aber nicht.

Lassen Sie uns kurz betrachten, welchen Zweck die Klassen und Interfaces haben:

- ▶ *Person* repräsentiert eine Person mit Vornamen und Nachnamen.
- ▶ *User* erweitert *Person* um einen persönlichen Kalender vom Typ *Calendar*.
- ▶ *Users* verwaltet alle im System angelegten Benutzer vom Typ *User*.

- ▶ *Calendar* repräsentiert einen Kalender, in dem Termine angelegt werden können.
- ▶ *CalendarItem* repräsentiert einen Termin.
- ▶ *DateTime* repräsentiert einen Zeitpunkt durch Datum und Uhrzeit. Diese Klasse ist vorgegeben und muss nicht von Ihnen erstellt werden.
- ▶ Das Interface *Ownable* erlaubt die Abfrage eines „Besitzers“ über die Methode *getOwner()*.
- ▶ Das Interface *Comparable* ist aus der Vorlesung bekannt und durch Java vorgegeben.

26.2 Funktionalität

26.2.1 Klasse *Person*

R1 Eine Person hat einen Vornamen und einen Nachnamen. (Hinweis: Überlegen Sie, welche Bezeichner die Variablen haben sollten, sodass die Getter *getFirstName()* und *getSurname()* in Eclipse automatisch erzeugt werden können.)

R2 Die Klasse besitzt genau einen Konstruktor, dem Vor- und Nachname übergeben werden.

R3 Vor- und Nachname können über Getter erfragt werden.

R4 Die Klasse überschreibt die Methode *toString()*. Der Rückgabewert hat die Formatierung <Vorname> <Nachname>.

Deklarationen:

```
public Person(String firstName, String surname)
public String getFirstName()
public String getSurname()
public String toString()
```

26.2.2 Klasse *User*

R5 Die Klasse erweitert die Klasse *Person* um einen Kalender vom Typ *Calendar* sowie eine eindeutige Benutzernummer (*User ID*). Der erste erzeugte User erhält als ID den Zahlenwert 1, der zweite den Wert 2, der dritte den Wert 3 und so weiter. (Hinweis: Verwenden Sie für die Vergabe der ID eine Klassenvariable, die stets den Wert der nächsten zu vergebenen ID enthält.)

R6 Die Klasse besitzt genau einen Konstruktor, dem Vor- und Nachname übergeben werden.

R7 Der Kalender und die ID können über Getter erfragt werden.

R8 Der Wert der nächsten zu vergebenen Benutzernummer, kann über die Klassenmethode *setNextUserID()* gesetzt werden.

R9 Die Klasse implementiert das Interface *Comparable<User>*. Das Kriterium für den Vergleich ist der Nachname, gefolgt vom Vornamen.

Deklarationen:

```
public User(String firstName, String surname)
public Calendar getCalendar()
public int getUserId()
public static void setNextUserID(int nextID)
public int compareTo(User other)
```

26.2.3 Klasse *Users*

- R10 Die Klasse besitzt eine Liste vom Typ *ArrayList<User>* zur Verwaltung der Nutzer.
- R11 Die Klasse besitzt einen Methode *createNewUser()* zum Anlegen eines neuen Benutzers. Die Nutzer werden hierbei in der Liste alphabetisch nach Nachname und Vorname sortiert.
- R12 Die Klasse besitzt eine Methode *getUsers()*, die ein Array aller Nutzer zurückgibt. (Hinweis: Verwenden Sie eine der *toArray()*-Methoden der Klasse *ArrayList*. Wie unterscheiden sich die Methoden? Welche sollten Sie am besten verwenden?)
- R13 Die Klasse überschreibt die Methode *toString()*. Implementieren Sie die Methode derart, dass der mitgelieferte JUnit-Test fehlerfrei durchläuft.

Deklaration:

```
public User createNewUser(String firstName, String surname)
public User[] getUsers()
public String toString()
```

26.2.4 Interface *Ownable*

- R14 Die Schnittstelle deklariert eine abstrakte Methode zur Rückgabe eines Besitzers (*Owner*).

Deklaration:

```
public Person getOwner()
```

26.2.5 Klasse *CalendarItem*

- R15 Die Klasse besitzt Instanzvariablen zur Speicherung des Betreffs (Typ *String*), der zu dem Termin einladenden Person (*User*), der eingeladenen Personen (*ArrayList<User>*), der Uhrzeit (*DateTime*) sowie der Dauer in Minuten (*int*) des Termins.
- R16 Die Klasse besitzt einen Konstruktor, in dem die Werte aller Instanzvariablen, außer den eingeladenen Personen, übergeben und gesetzt werden.
- R17 Die Klasse besitzt die in den nachfolgenden Deklarationen aufgeführten Getter. Beachten Sie bei der Implementierung der Methode *getParticipants()* den Hinweis zu Requirement 12.
- R18 Die Klasse implementiert die Schnittstelle *Ownable*.
- R19 Die Klasse besitzt eine Methode *addParticipant()*, die einen Teilnehmer zur Liste der eingeladenen Personen hinzufügt. Ein Teilnehmer wird nur dann hinzugefügt, falls er noch nicht in der Liste vorhanden ist (d. h. keine doppelten Einträge).
- R20 Die Klasse besitzt eine Methode *removeParticipant()*, die einen Teilnehmer aus der Liste der eingeladenen Personen entfernt.
- R21 Die Klasse überschreibt die Methode *toString()*. Implementieren Sie die Methode derart, dass der mitgelieferte JUnit-Test fehlerfrei durchläuft.

Deklaration:

```
public CalendarItem(String subject, User owner, DateTime dateTIme, int
durationInMinutes)
public String getSubject()
public Person getOwner()
public User[] getParticipants()
public DateTime getDateTIme()
public int getDurationInMinutes()
public void addParticipant(User participant)
public void removeParticipant(User participant)
public String toString()
```

26.2.6 Klasse *Calendar*

- R22 Die Klasse hat einen Besitzer (Typ *User*) sowie eine Liste der im Kalender enthaltenen Termine (*ArrayList<CalendarItem>*).
- R23 Die Klasse besitzt einen Konstruktor, in dem der als Parameter übergebene Besitzer gesetzt wird.
- R24 Die Klasse implementiert die Schnittstelle *Ownable*.
- R25 Die Klasse besitzt eine Methode *getItems()*, die ein Array aller im Kalender enthaltenen Termine zurückgibt.
- R26 Die Klasse besitzt eine Methode *createItem()*, die einen neuen Termin erzeugt und zur Liste der Termine hinzufügt. Zudem wird der Termin den Kalendern der eingeladenen Teilnehmer hinzugefügt. Nutzen Sie hierzu die nachfolgende Methode *addItem()* der Kalender der Teilnehmer.
- R27 Die Klasse besitzt eine Methode *addItem()*, die ein existierendes Terminobjekt zur Liste der Termine hinzufügt.
- R28 Die Klasse besitzt eine Methode *removeItem()*, die einen Termin aus dem Kalender entfernt. Hat der Kalenderbesitzer zu diesem Termin eingeladen (d. h. ist er auch Besitzer des Termins vom Typ *CalendarItem*), so wird der Termin abgesagt und daher auch in den Kalendern aller eingeladenen Teilnehmer entfernt.
- R29 Die Klasse überschreibt die Methode *toString()*. Implementieren Sie die Methode derart, dass der mitgelieferte JUnit-Test fehlerfrei durchläuft.

Deklaration:

```
public Calendar(User owner)
public Person getOwner()
public CalendarItem[] getItems()
public CalendarItem createItem(String subject, DateTime date, int
durationInMinutes, User[] participants)
private void addItem(CalendarItem item)
public void removeItem(CalendarItem item)
public String toString()
```

26.3 Anwendung

Erstellen Sie eine Anwendung (d. h. ausführbare Klasse), in der Sie die implementierten Funktionalitäten ausprobieren – beispielsweise:

- ▶ Erzeugen mehrerer Benutzer
- ▶ Erzeugen von Terminen mit mehreren Teilnehmern
- ▶ Absage eines Termins durch einen Teilnehmer
- ▶ Absage eines Termins durch den Benutzer, der eingeladen hat

Überprüfen Sie die Funktionalität anhand von Konsolenausgaben (*toString()*-Methoden).

Praktikum 3: Alarmanlage

Hinweise

- ▶ Organisieren Sie die Quelltexte im Paket `lab3.alarmSystem`.
- ▶ Alle Instanzvariablen besitzen den Modifier `private`.
- ▶ Die mitgelieferten JUnit-Tests müssen fehlerfrei durchlaufen.

27.1 Überblick

In diesem Praktikum sollen Sie eine einfache Alarmanlage mit Sensoren (z. B. für Türen oder Fenster) erstellen. Das UML-Diagramm in Abbildung 27.1 zeigt eine Übersicht der zu erstellenden Klassen und Schnittstellen, die nachfolgend erläutert werden:

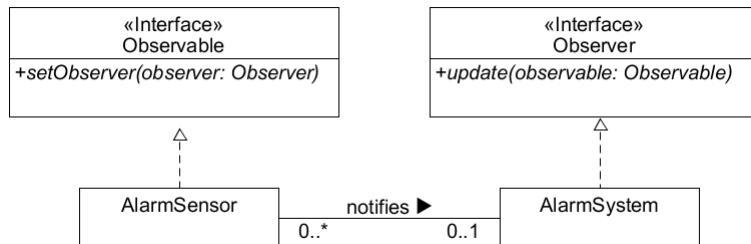


Abbildung 27.1: Klassendiagramm

Struktur:

- ▶ Die Klasse `AlarmSystem` repräsentiert eine Alarmanlage.
- ▶ Die Klasse `AlarmSensor` repräsentiert einen Sensor (z. B. an einer Tür oder einem Fenster).
- ▶ Ein Sensor kann mit einer Alarmanlage verbunden werden.
- ▶ Eine Alarmanlage kann mit beliebig vielen Sensoren verbunden sein.
- ▶ Wenn ein Sensor ausgelöst wird (d. h. Tür oder Fenster wird geöffnet), benachrichtigt er die Alarmanlage. Hierfür werden die Interfaces verwendet.

Interfaces:

- ▶ Die Alarmanlage „beobachtet“ wie folgt die Sensoren: Falls sich der Status eines Sensors ändert, so benachrichtigt der Sensor die Alarmanlage. Hierzu implementieren die Alarmanlage das Interface `Observer` und die Sensoren das Interface `Observable`.
- ▶ Die Alarmanlage ruft die Methode `setObserver()` eines Sensors aus, um sich mit ihm zu verbinden. Der Sensor speichert beim Verbinden eine Referenz auf die Alarmanlage. Die Alarmanlage speichert Referenzen auf alle verbundenen Sensoren.
- ▶ Wird ein Sensor ausgelöst, so benachrichtigt er die Alarmanlage, indem er ihre Methode `update()` aufruft. Hierbei übergibt er eine Referenz auf sich selber, damit die Alarmanlage „weiß“, welcher Sensor ausgelöst wurde.

27.2 Aufgabe

- ▶ Erzeugen Sie die Interfaces *Observer* und *Observable*.
- ▶ Implementieren Sie die Klassen *AlarmSystem* und *AlarmSensor*.
- ▶ Die zur Verfügung gestellten JUnit-Tests müssen fehlerfrei durchlaufen.
- ▶ Die ausführbare Klasse *AlarmMain* muss folgende Konsolen-Ausgabe erzeugen:

```
Alarmanlage einschalten:
Anlage nicht scharfgeschaltet
Anlage scharfgeschaltet

Sensoren auslösen:
<!> Sensor ausgelöst: Haustuer
<!> Sensor ausgelöst: Bad

Alarmanlage ausschalten:
Anlage scharfgeschaltet
Anlage nicht scharfgeschaltet
```

27.3 Lösungsstrategie

Analysieren Sie die ausführbare Klasse *AlarmMain* sowie die Klasse *UnitTests*:

- ▶ Welche Methoden werden für Objekte der Klassen *AlarmSystem* und *AlarmSensor* aufgerufen? Diese Methoden müssen Sie implementieren.
- ▶ Welchen Datentyp und welche Bedeutung haben die an die Methoden übergebenen Argumente?
- ▶ Werden gegebenenfalls Rückgabewerte verwendet?

Sie werden feststellen, dass in diesem Versuch nur sehr wenig zu implementieren ist.

Kapitel 28

Praktikum 4: Alarmanlagen-GUI

28.1 Überblick

In diesem Praktikum sollen Sie eine grafische Benutzeroberfläche (*Graphical User Interface, GUI*) erstellen. Es gibt *keinerlei* Vorgaben bezüglich der zu erstellenden Klassen und/oder des genauen Aussehens der Oberfläche. Stattdessen ist ausschließlich von Bedeutung, dass Ihr Programm die in Abschnitt 28.2 geforderte Funktionalität aufweist.

Die GUI soll in sehr vereinfachter Form die Bedienelemente sowie den Zustand einer Alarmanlage enthalten bzw. darstellen. Die Anlage kann scharf geschaltet und über einen Ziffernkode wieder deaktiviert werden. In Abbildung 28.1 werden hierfür entsprechende Buttons verwendet. Es steht Ihnen frei, die Steuerung über andere Elemente (z. B. Menüeinträge oder Textfelder) zu realisieren.

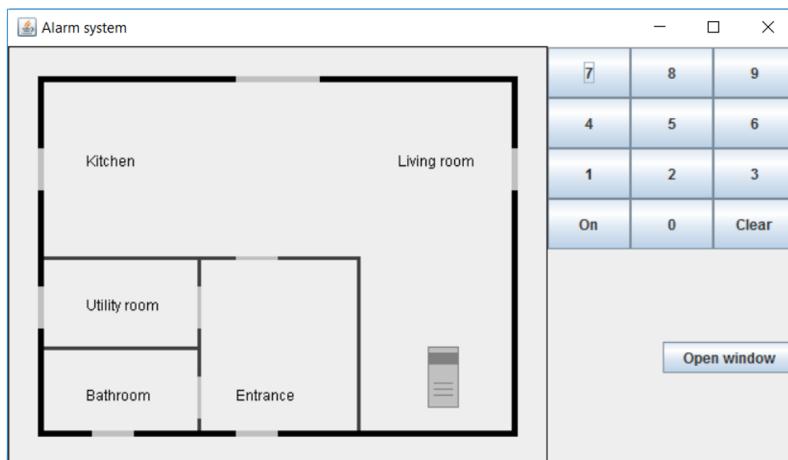


Abbildung 28.1: Beispiel einer grafischen Benutzeroberfläche

Des Weiteren soll die Möglichkeit bestehen, (rein virtuell) ein Fenster zu öffnen. Geschieht dies während die Anlage scharf geschaltet ist, so wird ein Alarm ausgelöst. Der Zustand der Anlage muss aus der GUI ersichtlich werden. In der abgebildeten Beispiel-Oberfläche in den Abbildungen 28.1 und 28.2 geschieht dies durch ein kleines Symbol mit den Farben grau (*deaktiviert*), grün (*scharf geschaltet*) und rot (*Alarm ausgelöst*). Sie dürfen den Zustand jedoch auch auf andere Art und Weise visualisieren.

28.2 Aufgabe

Erstellen Sie eine grafische Benutzeroberfläche mit nachfolgenden Eigenschaften.

Aktivierung:

- ▶ Benutzer können eine Alarmanlage scharf schalten.

Deaktivierung:

- ▶ Benutzer können die Alarmanlage durch Eingabe des korrekten, aus vier Ziffern bestehenden Zahlencodes wieder deaktivieren. Der korrekte Code ist fest hinterlegt und lautet 2016.



Abbildung 28.2: Zustände der eingeschalteten Alarmanlage

- ▶ Bei Eingabe eines fehlerhaften Codes erhalten Benutzer eine entsprechende Rückmeldung. Verwenden Sie hierfür z. B. die Methode `showMessageDialog()` der Klasse `JOptionPane`.

Alarmzustand:

- ▶ Der Benutzer kann ein Fenster des von der Alarmanlage überwachten Gebäudes öffnen.
- ▶ Befindet sich die Alarmanlage beim Öffnen eines Fensters im scharf geschalteten Zustand, so wird ein Alarm ausgelöst.
- ▶ Der Alarmzustand wird durch Deaktivierung der Alarmanlage beendet.

Visualisierung des Zustandes:

- ▶ Die grafische Oberfläche visualisiert den Zustand der Alarmanlage, wobei die Zustände deaktiviert, scharf geschaltet sowie Alarm ausgelöst zu unterscheiden sind.

28.3 Lösungsstrategie

- ▶ Entwerfen und implementieren Sie zunächst das Aussehen der grafischen Oberfläche (d. h. die Art und Anordnung der Elemente).
- ▶ Implementieren Sie erst im zweiten Schritt die geforderte Funktionalität.
- ▶ Verzetteln Sie sich nicht in Details bezüglich des Aussehens der grafischen Oberfläche. (Für besonders kreative und ästhetische Lösungen gibt es Bewunderung und Begeisterung, aber keinen Schönheitspreis.)

Teil VI

Appendix

Anhang A

IntelliJ IDEA

Nachdem die zugehörige Lehrveranstaltung viele Jahre die Entwicklungsumgebung *Eclipse* (siehe Anhang B) verwendet hat, wurde diese inzwischen durch *IntelliJ IDEA* ersetzt. Die nachfolgenden Abschnitte sollen als Sammelsurium für hilfreiche Tipps und Tricks mit IntelliJ IDEA dienen und dementsprechend im Laufe der Zeit erweitert werden.

A.1 Tipps & Tricks

Dateien zu einem Projekt hinzufügen Prinzipiell lassen sich Dateien (z. B. Unit-Tests) zwar per Drag & Drop zu einem Projekt hinzufügen. Allerdings werden keine Kopien erzeugt, sondern die Dateien verschoben. Dies führt teils zu einem Fehler, da der Windows-Explorer das Löschen und damit das Verschieben verhindert. Kopieren Sie Dateien daher zunächst im Windows-Explorer in das Ziel-Verzeichnis (z. B. das Verzeichnis eines Paketes) und ziehen sie diese anschließend per Drag & Drop in das entsprechende Paket innerhalb der IntelliJ IDEA-Oberfläche.

Inlay Hints ein-/ausschalten IntelliJ IDEA zeigt standardmäßig eine Reihe Informationen wie beispielsweise die Verwendungen von Variablen und Methoden dar. Während dies mitunter durchaus hilfreich ist, kann jedoch insbesondere die zusätzliche Angabe über jedem Attribut die Übersichtlichkeit verringern – und einfach stören. Bei Bedarf lassen sich die zusätzlichen Angaben wie in Abbildung A.1 dargestellt über *File / Settings... / Inlay Hints* ein- bzw. ausschalten.

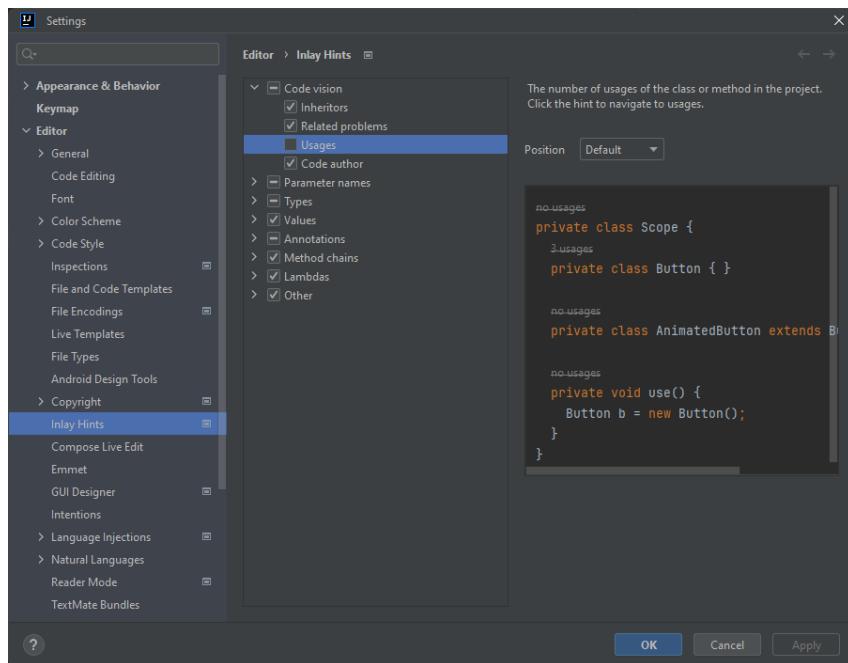


Abbildung A.1: Aktivieren und deaktivieren zusätzlicher Angaben (*Inlay Hints*)

A.2 Probleme und Lösungen

Java Development Kit (JDK) ändern IntelliJ IDEA-Projekte beinhalten Informationen über das verwendete Java Development Kit (JDK). Daher können die von mir zur Verfügung gestellten

Projekte ein anderes JDK erwarten als auf Ihrem Rechner installiert ist. In diesen Fällen erscheint oberhalb des Quelltext-Editor eine Fehlermeldung. Wählen Sie dort die Option *Configure* aus, lässt sich ein auf dem konkreten Rechner installiertes JDK auswählen.

A.3 Unit-Tests

A.3.1 Compiler-Einstellungen und Konfiguration

Typischer Weise wird die Funktionalität einer bestimmten Software nicht mit einem einzigen Test sondern einer ganzen Sammlung sehr vieler Test, einer sogenannten *Test Suite*, überprüft. Führt man diese aus, so kann es passieren, dass bestimmte Tests aufgrund von Kompilierfehlern (z. B. Aufruf einer noch nicht implementierten Methode) nicht gebaut und ausgeführt werden können. In derartigen Fällen sollten trotz auftretender Fehler alle Tests, die gebaut und ausgeführt werden können, laufen. Andernfalls könnte beispielsweise ein einziger „fehlerhafter“ Test die Ausführung einer sehr umfangreichen Test Suite mit mehrstündiger Laufzeit vollkommen blockieren – nicht schön.

Leider blockieren Kompilierfehler in den Standard-Einstellungen von IntelliJ IDEA die Ausführung von JUnit-Tests. Dieses in meinen Augen unerwünschte Verhalten lässt sich jedoch wie nachfolgend beschrieben durch eine Änderung der Einstellungen sowie eine entsprechende *Run*-Konfiguration für JUnit-Tests abstellen.

Einstellungen Zunächst muss in den allgemeinen Einstellungen ein Compiler ausgewählt werden, der auch bei auftretenden Fehlern mit dem Übersetzen von Quelltexten fortfährt. Wählen Sie hierfür wie in Abbildung A.2 dargestellt unter *File / Settings...* auf Karte *Build / Java Compiler* den Compiler *Eclipse*. Stellen Sie zudem sicher, dass der Haken bei *Proceed on Errors* gesetzt ist.

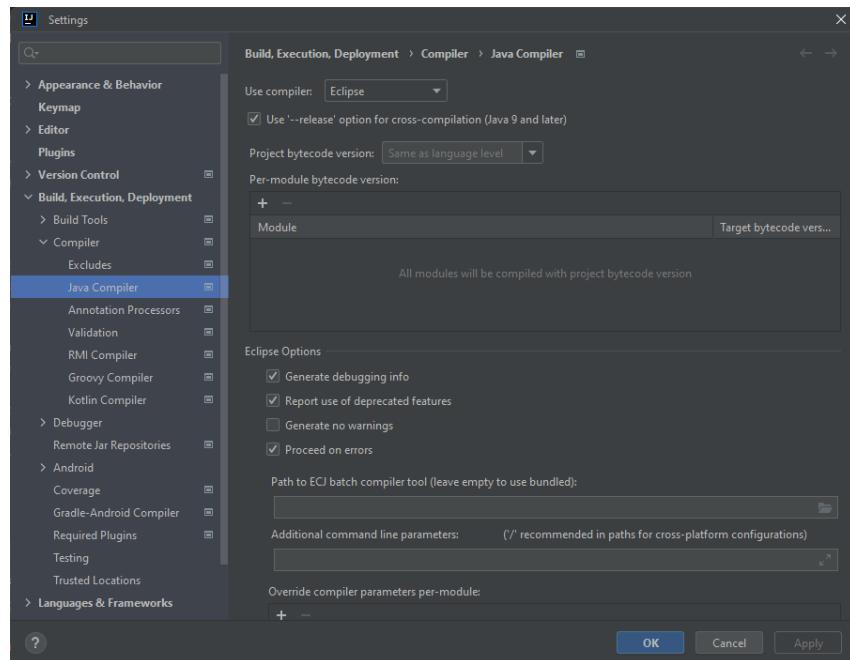


Abbildung A.2: Auswahl eines Compilers, der bei auftretenden Fehlern fortfährt

Konfiguration Als nächstes muss eine geeignete Konfiguration zur Ausführung von JUnit-Tests erstellt werden. Wählen Sie hierfür *Run / Edit Configurations...* und erstellen Sie über das Plus-Symbol eine neue *JUnit*-Konfiguration. Wie in Abbildung A.3 dargestellt, muss vor den Tests

Build, no error check ausgeführt werden. Dieser Punkt kann über *Modify options* (Abb. A.3a) und Setzen des Hakens *Before launch / Add before launch task* (Abb. A.3b) hinzugefügt werden. Im selben Dialog ist zudem der Haken *Java / Do not build before run* zu setzen.

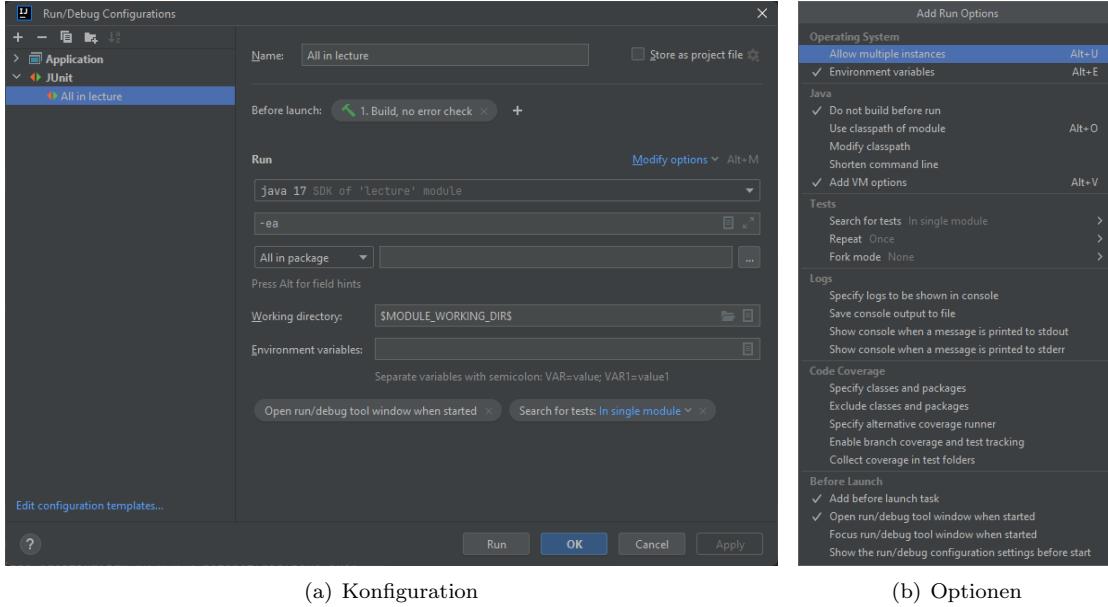


Abbildung A.3: Konfiguration von JUnit-Tests

Abbildung A.4 zeigt eine exemplarische Ausgabe bei der Ausführung von Test. Trotz diverser Kompilierfehler wurden alle Tests unabhängig voneinander ausgeführt.

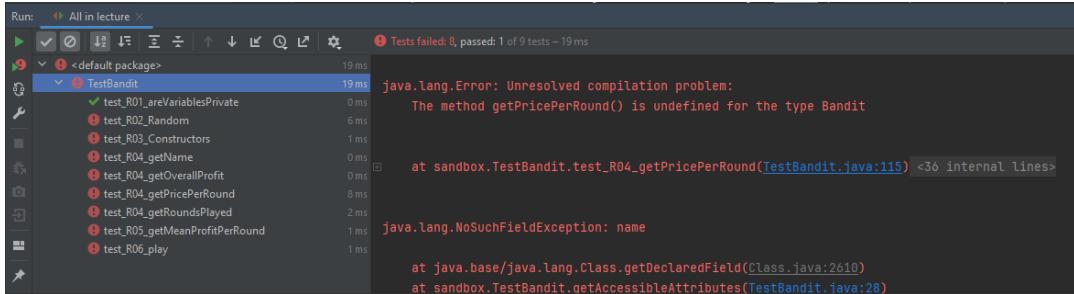


Abbildung A.4: Ausführung von JUnit-Tests mit Kompilierfehlern

A.3.2 Tests einbinden

Dateien mit Unit-Tests hinzufügen Das Hinzufügen von Dateien zu einem Projekt ist in Anhang A.1 beschrieben.

JUnit-Bibliothek einbinden Sie werden JUnit typischer Weise einbinden müssen, um gegebene Tests auszuführen. Am einfachsten fügen Sie zunächst die Datei mit JUnit-Tests Ihrem Paket hinzu und öffnen Sie. Machen Sie gegebenenfalls die Imports durch Auswahl des Plus-Zeichens neben *import ...* sichtbar. Halten Sie anschließend wie in Abbildung A.5 dargestellt den Mauszeiger über den roten Text *junit* und wählen Sie *Add 'JUnit4' to class path* oder ähnlich.

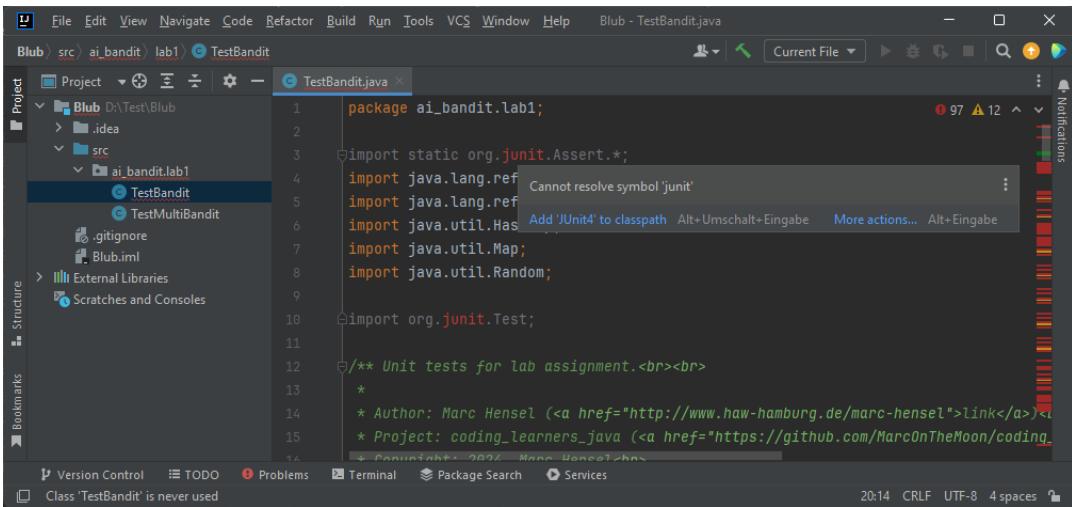


Abbildung A.5: JUnit-Bibliothek einbinden

A.3.3 Ausführung

Deutung der Test-Ergebnisse Die Deutung der Ergebnisse unterscheidet sich naturgemäß nicht von den Beschreibungen in Abschnitt B.1.

Vergleich von Zeichenketten Erwähnenswert ist hingegen, dass sich auch in IntelliJ IDEA bei einem fehlgeschlagenen Vergleich von Zeichenketten die erwartete und die tatsächlich Vorliegende Zeichenketten sehr hilfreich nebeneinander darstellen lassen. Hierzu ist wie in Abbildung A.6 ersichtlich innerhalb der Fehlerbeschreibung *Click to see differences* auszuwählen, woraufhin die Zeichenketten darüber im Bereich des Quelltext-Editor dargestellt werden.

A.4 Debugging

Abschnitt folgt

A.5 Wichtige Tastenkombinationen

Tabelle A.1 gibt eine Übersicht ausgewählter Tastenkombinationen.

Tabelle A.1: Ausgewählte Tastenkombinationen

Kombination	Effekt
Alt+Shift+F10	Im Editor geöffnete Quelldatei ausführen
Alt+Shift+F9	Im Editor geöffnete Quelldatei ausführen (Debug-Modus)
Alt+F6	Umbenennen (Refactoring)
Alt+Einfg	Methode an Cursor-Position generieren (z. B. Getter oder Konstruktor)
Ctrl+{/}	Zeilen kommentieren ein/aus (Nur mit / des Ziffernblocks!)
Ctrl+Shift+{/}	Blockkommentar ein/aus (Nur mit / des Ziffernblocks!)
Ctrl+A	Gesamten Quelltext auswählen
Ctrl+Alt+I	Linkseinzüge im ausgewählten Bereich korrigieren
psvm Tab	Erzeuge main()-Methode („public static void main“)
sout Tab	Erzeuge System.out.println () („string out“)
Ctrl+J	Alle Kürzel zur Code-Erzeugung anzeigen

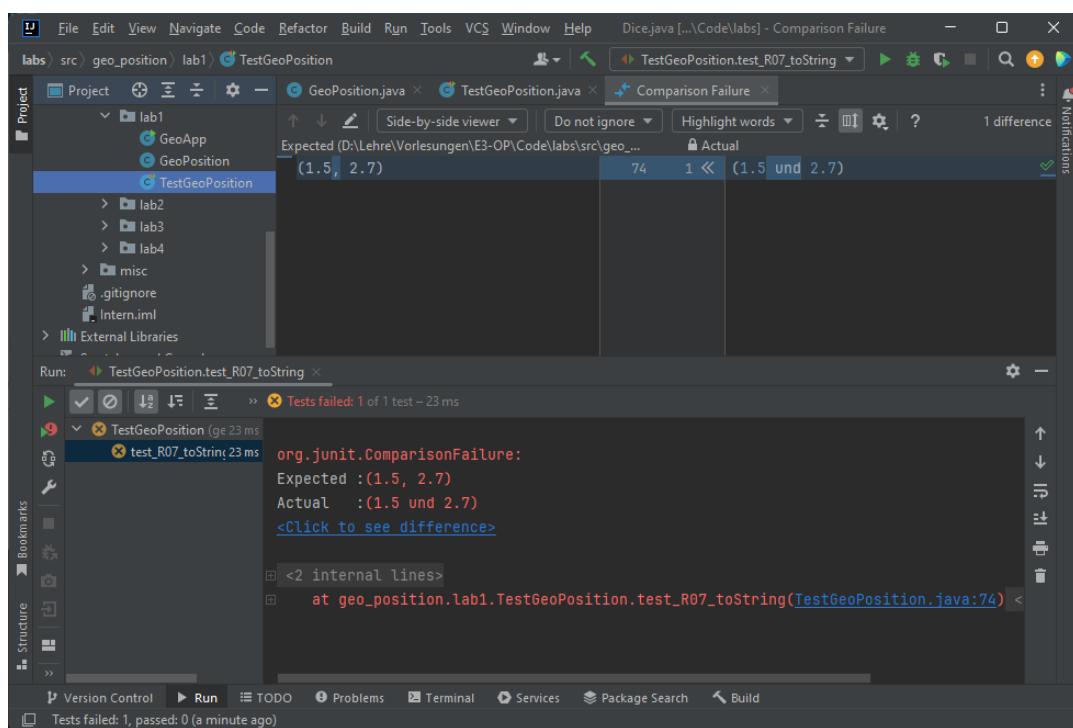


Abbildung A.6: Gegenüberstellung der Zeichenketten bei fehlgeschlagenem Test

Anhang B

Eclipse

Für die zugehörige Lehrveranstaltung wurde viele Jahre die Entwicklungsumgebung *Eclipse* genutzt, welche inzwischen durch *IntelliJ IDEA* ersetzt wurde. Dennoch habe ich mich entschlossen, die nachfolgenden Abschnitte nicht zu entfernen bzw. sie Ihnen nicht vorzuenthalten. Tipps und Tricks zu IntelliJ IDEA werden fortan in Anhang A gesammelt.

B.1 Unit-Tests

Es gibt unterschiedliche Ansätze, Systeme und deren Bestandteile auf korrekte Funktionalität zu testen. Beispielsweise betrachtet man bei *Black Box*-Tests das zu prüfende Objekt als „schwarzen“ Kasten, in den man nicht hineinschauen kann. Daher überprüft man, ob auf bestimmte Eingaben die erwarteten Ausgaben bzw. Reaktionen erfolgen. Bei *White Box*-Tests ist hingegen das Innere des Prüfobjektes bekannt und es können daher interne Werte überprüft werden.

In der Software-Entwicklung werden typischer Weise *Unit-Tests* geschrieben, die Quelltexte automatisch testen. Hierbei wird anhand von sogenannten *Asserts*¹ überprüft, ob z. B. Methoden die erwarteten Werte zurückgeben. Ist dies nicht der Fall, so schlägt der entsprechende Test fehlt und erzeugt eine Ausgabe mit Informationen zum Fehler. Werden Unit-Tests automatisch als Programm ausgeführt, kann bei Änderungen der Quelltexte ohne großen Aufwand überprüft werden, ob die Software noch wie erwartet funktioniert.

Für Java steht mit *JUnit* ein Test-Framework zur Verfügung, das sich sehr gut in Eclipse integriert. Für einige Praktikumsaufgaben und gegebenenfalls auch in der Laborprüfung erhalten Sie *JUnit-Tests*. Diese sollen Ihnen helfen, syntaktische und logische Fehler zu entdecken, und Sie dabei unterstützen, die Fehlerquelle zu identifizieren, um diese zu beheben.

B.1.1 Tests einbinden

In einem Eclipse-Projekt lassen sich die Tests nur dann ausführen, wenn die *JUnit*-Bibliothek hinzugefügt wurde. Gehen Sie hierfür wie folgt vor:

1. Klicken Sie mit der rechten Maustaste auf das Java-Projekt.
2. Wählen Sie im Kontextmenü *Build Path / Add Libraries....*
3. Wählen Sie *JUnit* und beenden Sie den Dialog über die Buttons *Next* und *Finish*.

Nach dem Schließen des Dialogfensters erscheint im Package-Explorer neben der Runtime Environment (JRE) auch die hinzugefügte JUnit-Bibliothek. Anschließend fügen Sie die Datei mit den Tests (z. B. *TestComplex.java*) zum Eclipse-Projekt hinzu. Sie können die Datei aus dem Windows-Explorer per Drag&Drop in das entsprechende Paket des Java-Projekts ziehen. Gegebenenfalls muss in der ersten Codezeile der Test-Datei der Paketname angepasst werden.

B.1.2 Tests ausführen und deuten

Sofern die *JUnit*-Bibliothek erfolgreich eingebunden wurde, lässt sich eine Test-Datei wie jede ausführbare Klasse in Eclipse über *Strg-F11* ausführen. Die ausgeführten Tests werden im Anschluss aufgelistet und das jeweilige Ergebnis farblich markiert:

- *Grün* (Abb. B.1): Der Test ist erfolgreich durchgelaufen.

¹to assert (engl.): bestätigen, versichern, durchsetzen

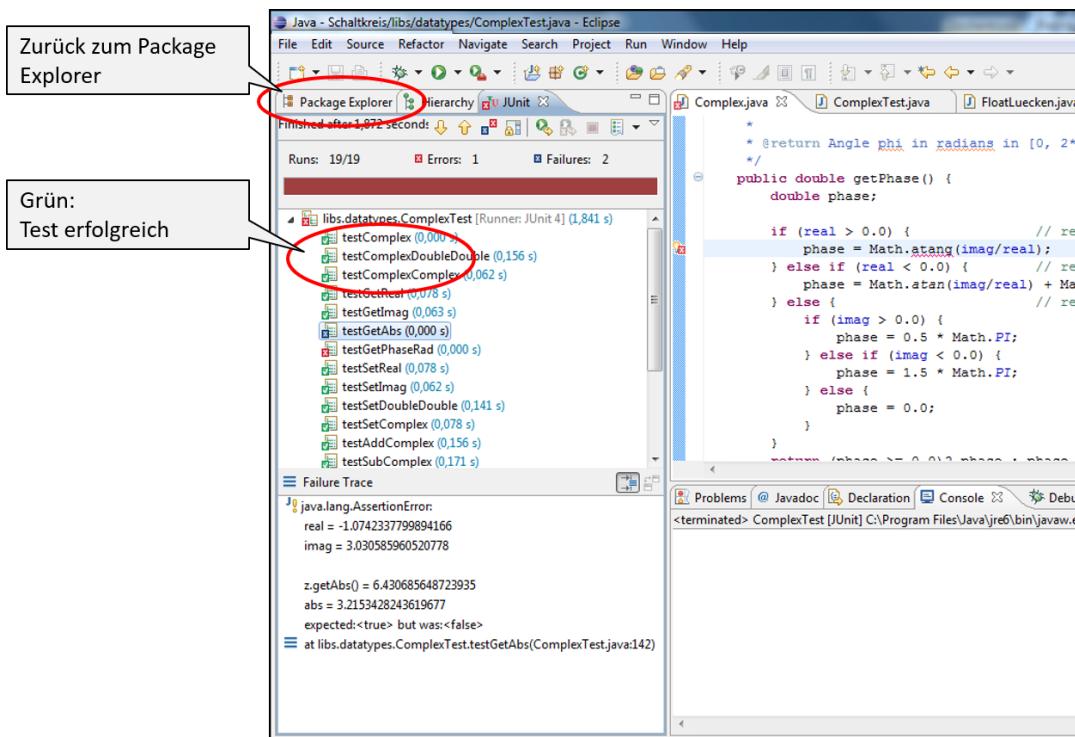


Abbildung B.1: Beispielausgabe erfolgreicher Unit-Tests

- *Blau* (Abb. B.2): Das Testprogramm konnte ausgeführt werden, aber ein *Assert* ist fehlgeschlagen. In der Regel ist dies der Fall, falls ein tatsächlicher Wert wie z. B. der Rückgabewert einer Methode nicht mit dem vom Unit-Test erwarteten Wert übereinstimmt.
- *Rot* (Abb. B.3): Es ist ein Fehler aufgetreten. Dies ist beispielsweise der Fall, wenn das zu testende Programm nicht erfolgreich kompiliert oder eine unzulässige Operation wie die Division durch Null aufgetreten ist.

Falls ein Test nicht erfolgreich durchgelaufen ist, lässt sich die Ursache recht bequem untersuchen. Durch einen Doppelklick auf den Test springt der Editor zu der entsprechenden Codezeile. Zudem erscheinen im Bereich links unten in Eclipse weiterführende Informationen und Hilfen. Neben der Art des Fehlers werden bei einem fehlgeschlagenen Vergleich der erwartete sowie der tatsächliche Wert angegeben (Abb. B.2).

Schlägt in einem Unit-Test der Vergleich zweier Zeichenketten fehl, so sind die Unterschiede zwischen einer erwarteten und der tatsächlichen Zeichenkette mitunter nur sehr schwierig zu sehen. Durch Doppelklick auf diese oberste Zeile im Informationsbereich links unten öffnet sich jedoch ein hilfreicher Analyse-Dialog. In diesem werden beide Zeichenketten gegenüber gestellt und Unterschiede rot markiert. Zudem werden in den Zeichenketten enthaltene Zeilenumbrüche dargestellt.

B.2 Debugging

Die Möglichkeit, ein Programm zum Identifizieren von Fehlern schrittweise zu durchlaufen und dabei den jeweils aktuellen Wert der Variablen zu betrachten, sollten Sie bereits aus den Modulen *Programmieren 1* und *Programmieren 2* kennen. Selbstverständlich lässt sich auch Quelltexte in Eclipse entsprechend *debuggen*.

Eine einfache Vorgehensweise ist, zunächst im Editor einen Haltepunkt durch Doppelklick auf die gewünschte Zeilennummer links des Quelltextes zu setzen. Haltepunkte werden durch einen

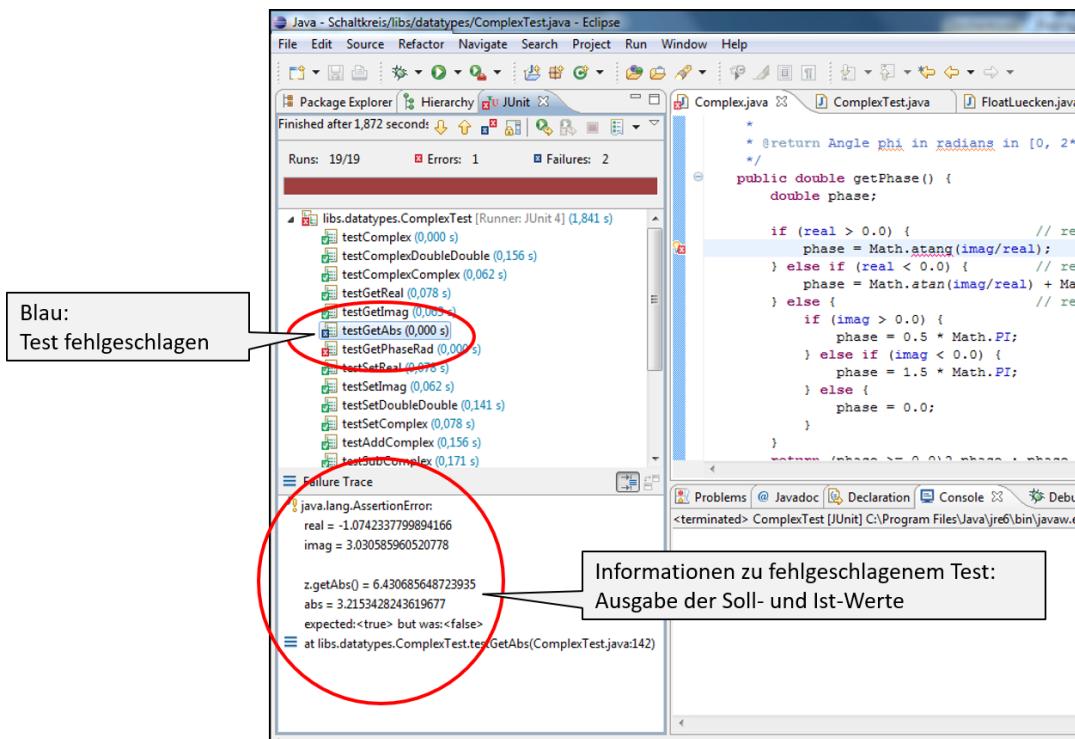


Abbildung B.2: Beispieldausgabe eines fehlgeschlagenen Unit-Tests

blauen Kreis symbolisiert (Abb. B.4) und lassen sich durch erneuten Doppelklick wieder entfernen.

Durch Drücken der Taste *F11* bzw. durch Auswahl des Menüpunktes *Run / Debug* wird das

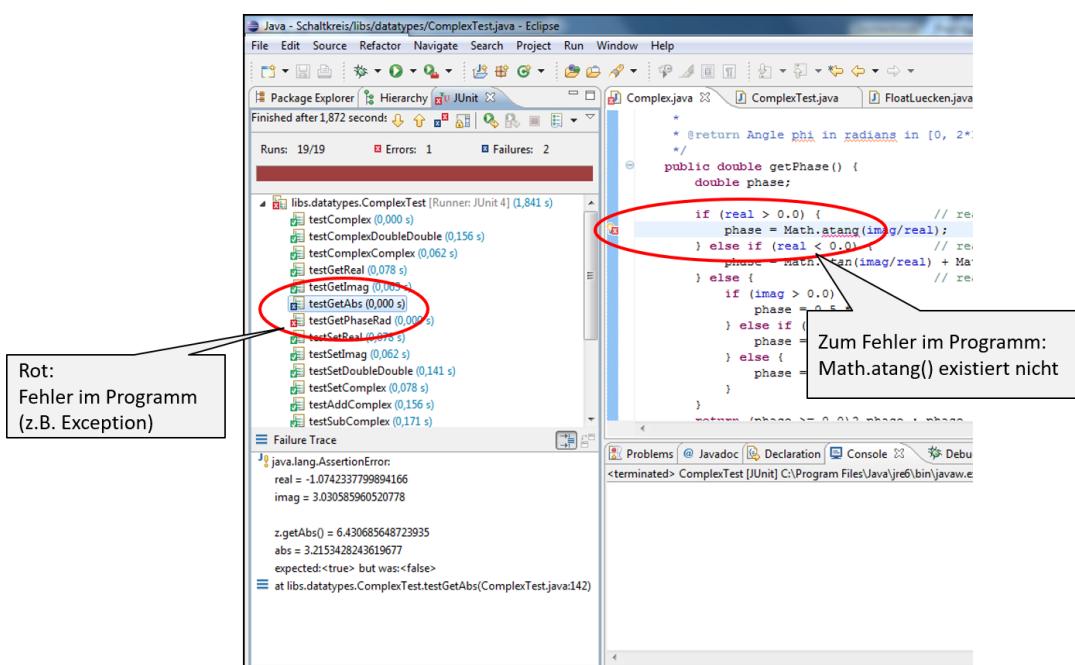


Abbildung B.3: Beispieldausgabe eines Fehlers bei der Testausführung

```

1 package folien05.geometrie;
2
3 import java.util.ArrayList;
4
5 public class AreaDemo {
6     public static void main(String[] args) {
7         ArrayList<Shape> shapes = new ArrayList<Shape>();
8         shapes.add(new Circle(2.0, 3.0, 1.0));
9         shapes.add(new Rectangle(-1.0, 0.0, 3.5, 4.0));
10        shapes.add(new Square(0.0, 0.0, 2.5));
11
12        double sumArea = 0.0;
13        for (Shape shape : shapes) {
14            sumArea += shape.getArea();
15        }
16
17        System.out.println("Overall area of shapes = " + sumArea);
18    }
19 }

```

Abbildung B.4: Gesetzter Haltepunkt

Programm gestartet und bis zum nächsten Haltepunkt ausgeführt. Anschließend stehen insbesondere die in Abbildung B.5 dargestellten Ansichten und Ausführungsoptionen zur Verfügung.

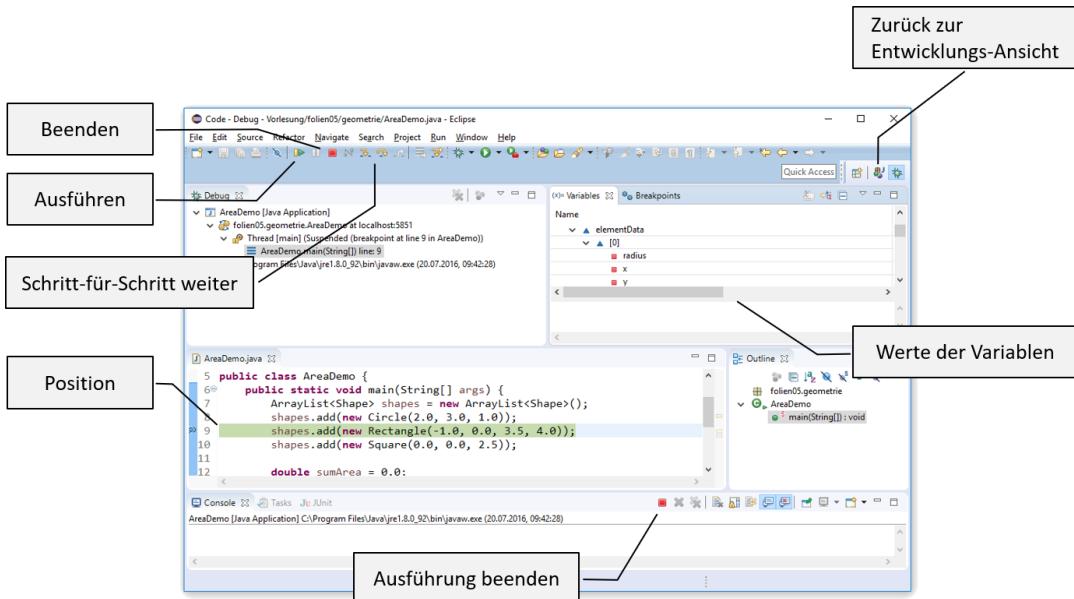


Abbildung B.5: Ansicht im Debug-Modus

Anhang C

Checkliste Softwarequalität

Kriterium	Ok
Quelltexte kompilieren <i>ohne</i> Warnungen und Fehler.	<input type="checkbox"/>
Bezeichner sind englischsprachig.	<input type="checkbox"/>
Bezeichner sind aussagekräftig.	<input type="checkbox"/>
Bezeichner für Variablen beginnen mit kleinem Buchstaben.	<input type="checkbox"/>
Mehrere Wörter sind als <i>CamelCase</i> zusammengesetzt (nicht mit Unterstrichen).	<input type="checkbox"/>
Bezeichner für Konstanten bestehen nur aus Großbuchstaben. Mehrere Wörter sind durch Unterstrich zusammengesetzt.	<input type="checkbox"/>
Öffnende geschweifte Klammern stehen nicht in einer neuen Zeile. Nach öffnender geschweifter Klammer erfolgt ein Zeilenumbruch.	<input type="checkbox"/>
Nach schließender geschweifter Klammer erfolgt ein Zeilenumbruch. Ausnahme: Das Schlüsselwort <i>else</i> steht in gleicher Zeile.	<input type="checkbox"/>
Vor logischen Abschnitten innerhalb einer Methode steht ein Kommentar als Überschrift.	<input type="checkbox"/>
Jede Blockebene wird um eine Tabulatorstufe horizontal eingerückt.	<input type="checkbox"/>
Zwischen Methoden steht eine Leerzeile.	<input type="checkbox"/>
Zwischen Klassen steht eine Leerzeile.	<input type="checkbox"/>
Zwischen logischen Blöcken steht eine Leerzeile.	<input type="checkbox"/>
Klassen und Interfaces sind durch eine Leerzeile von <i>import</i> - und <i>package</i> -Anweisungen getrennt.	<input type="checkbox"/>
Zwischen Operanden und Operatoren steht ein Leerzeichen.	<input type="checkbox"/>
Nicht mehr als <i>eine</i> Leerzeile hintereinander. Nicht mehr als <i>ein</i> Leerzeichen hintereinander.	<input type="checkbox"/>
Reihenfolge innerhalb einer Klasse oder eines Interfaces: <ol style="list-style-type: none">1. Attribute (Variablen und Konstanten)2. Konstruktoren3. Getter und Setter für Attribute4. Sonstige Methoden	<input type="checkbox"/>

Index

- π , *siehe* Kreiszahl
- A-Moll, *siehe* Musik
- abstrakte Klasse, 48, 53
- abstrakte Methode, 53
- Abtastfrequenz, 62
- Abtastintervall, 62
- Abtastung, 62
- Äquator, 102
- Akkord, *siehe* Musik
- Alarmanlage, 138, 140
- Amplitude, 62
- Android-App, 13
- Anzeige
- 7-Segment, 29
- Architektur, 123
- Arduino
- Nano R3, 12
- Array, *siehe* Feld
- Audio
- Dauer, 44
 - Audio-Signal
 - Frequenzbereich, 63
 - Zeitbereich, 62
 - Audiodatei, *siehe* Dateien
 - Audioverarbeitung, 61
- Ausgabe
- Dezimalkomma, 14
 - Dezimalpunkt, 14
 - formatiert, 12, 14
- Ausnahme, 36, 68
- eigene Klasse, 68
 - fangen, 68
 - Stack-Trace, 69
- Auswahlkasten, *siehe* Check box
- Bandit
- einarmig, 85, 91
 - mehrarmig, 88, 91, 93, 97
- Bandreject filter, *siehe* Bandsperrfilter
- Bandsperrfilter, *siehe* Filter
- Bankkonto, 14
- Barnsleys Farn, 58
- Basisklasse, 47
- Benachrichtigung, 50
- Benutzereingabe, 88
- Beobachter, *siehe* Observer
- Beobachter-Muster, *siehe* Observer pattern
- Bestärkendes Lernen, *siehe* Reinforcement Learning
- Bild, 35
- Graustufenbild, 35
 - Kante, 40
- Kodierung, 37
- Negativ, 35, 39
- Wertebereich, 40
- Bilddatei
- Format, 35
 - lesen, 35
 - Projekt hinzufügen, 37
 - schreiben, 35
- Bildpunkt, *siehe* Pixel
- Bildverarbeitung, 35
- Gradienten-basierte Kantendetektion, 40, 42
 - Invertierten, 39
 - Kantendetektion, 35, 40
 - Programm, 35
 - Punktoperationen, 39
 - Sobel-Operator, 40, 42
- Binärzahl, 11, 14
- Bisektionsverfahren, 16, 21, 33, 49
- Black Box-Test, *siehe* Tests
- Bluetooth
- Modul HC-06, 12
- Bogenmaß, 31
- Breitengrad, *siehe* Koordinaten
- Buch, 18
- Bücherstapel, 32
- Button, *siehe* Grafische Benutzeroberfläche
- Byte-Strom, *siehe* Dateien
- Bytecode, 10
- C
- Programmiersprache, 12
- C-Dur, *siehe* Tonleiter
- Casino, 85
- Check box, *siehe* Grafische Benutzeroberfläche
- Coding style, *siehe* Software-Qualität
- Compiler, 10
- Container, 49
- Dateien
- Audiodatei erzeugen, 73
 - Byte-Strom, 73
 - Textdatei lesen, 71, 115
 - Textdatei schreiben, 71
 - WAVE-Format, 73
- Datentyp, 11
- primitiver, 17, 24
- Dezimalkomma, *siehe* Ausgabe
- Dezimalpunkt, *siehe* Ausgabe
- DFT, *siehe* Fourier-Transformation
- Diagramm
- Balken, 100
 - Funktion, 100
- Diode, 11

- Diskrete Fourier-Transformation, *siehe* Fourier-Transformation
- Division mit Rest, 65
- Dunkel war's, der Mond schien helle, 71
- Eclipse, 149
 - Debugging, 150
 - Tests ausführen, 149
 - Tests deuten, 149
 - Tests einbinden, 149
 - Unit-Tests, 149
- Einarmiger Bandit, *siehe* Bandit
- Eingabe
 - Tastatur, 26, 71
- Empfänger, 50
- Entfernung
 - geografische Orte, 103
- Epsilon-Greedy-Verfahren, 94
- Erdradius, 103
- Ereignis
 - Maus, 112, 115
- Euler-Zahl, 15
- Event, *siehe* Ereignis
- Exception, *siehe* Ausnahme
- Exploitation, 94
- Exploration, 94
- Fakultät, 22
- Fast Fourier Transformation, *siehe* Fourier-Transformation
- Feld, 27
 - mehrdimensional, 29
 - sortieren, 28
- FFT, *siehe* Fourier-Transformation
- Filter
 - Bandsperrfilter, 67
 - Frequenzbereich, 66
- Fläche
 - Kreis, 14
- Fortschrittsbalken, *siehe* Check box
- Fourier-Transformation
 - DFT, 61, 63
 - FFT, 65
 - IDFT, 67
 - Logarithmus, 64
 - Rechenzeit, 65
 - Spektrum, 61, 63
 - Verschiebung des Spektrums, 64
- Frequenz, *siehe* Ton
- Frequenzanteile, 61
- Frequenzbereich, 61, 63, 128
 - Filter, 66
- Funktion
 - differenzierbar, 48
- erste Ableitung, 21
- Graph, 57
- Kosinus, 65
- Logarithmus, 48
- Polynom, 20, 21, 48
- Sinus, 61, 65
- Game of Life, *siehe* Spiel des Lebens
- Gaußverteilung, 85
- GIMP, 35
- Glücksrad, 92
- Gradient, 40
- Grafische Benutzeroberfläche, 55, 97
 - Button, 55
 - Check box, 56
 - Dialog-Fenster, 68
 - Layout-Manager, 59
 - Menü, 55
 - Progress bar, 59
 - Radio button, 57
 - Slider, 59
- Graphical user interface, *siehe* Grafische Benutzeroberfläche
- Graustufenbild, *siehe* Bild
- Grundton, *siehe* Ton
- GUI, *siehe* Grafische Benutzeroberfläche
- Häufigkeit
 - relative, 29
- Haltepunkt, 150
- Hemisphäre, 104
- Icon, 68
- IDFT, *siehe* Fourier-Transformation
- Impedanz, 126
- Induktivität, 129
- Instanz, 24
- Instanzvariable, *siehe* Variable
- IntelliJ IDEA, 10, 144
 - Aktivierung, 82
 - Dateien hinzufügen, 144, 146
 - Debugging, 83, 147
 - Inlay Hints, 144
 - JDK ändern, 144
 - JUnit, 84
 - JUnit einbinden, 146
 - PC-Pool, 82
 - Projekt, 10
 - Tastenkombinationen, 147
 - Unit-Tests, 145
- Interface, 48, 53
- Inverse Fourier Transformation, *siehe* Fourier-Transformation
- Iterieren, 49

- Java Virtual Machine, 10, 78
 JDK, *siehe* Java Development Kit
 JUnit, *siehe* Tests
 JVM, *siehe* Java Virtual Machine
- Kalender, 134
 Kanal, 73
 Kantendetektion, *siehe* Bildverarbeitung
 Kapazität
 elektrisch, 126, 129
 Kartenleser, 13
 Klasse
 ActionEvent, 56
 ActionListener, 123
 AlarmMain, 139
 AlarmSensor, 138, 139
 AlarmSystem, 138, 139
 ArrayList, 108, 109, 136
 ArrayList<CalendarItem>, 137
 ArrayList<Container>, 50
 ArrayList<User>, 136
 Audio, 44–47
 AudioData, 75
 AudioDataException, 75
 AudioFileWAV, 75
 AudioGUI, 63, 65
 AudioModel, 62
 Bandit, 87, 88, 92, 93
 BanditApp, 87–89, 92, 93
 BankAccount, 83
 Book, 18, 32
 BookStack, 32
 BufferedImage, 36–38, 112
 BufferedReader, 115
 Calendar, 134, 135, 137
 CalendarItem, 134–137
 Collections, 49
 Comparable, 49, 108, 135
 Comparable<Container>, 49
 Comparable<GeoRoute>, 108
 Comparable<User>, 135
 CompareArrays, 28
 Complex, 64, 126–129
 Container, 49, 50
 Container[], 50
 DateTime, 134–136
 Dice, 19, 20
 Distance, 108, 110
 Duration, 44, 45
 ExampleComplex, 128
 ExampleVector2D, 132
 File, 36, 71
 FileOutputStream, 73
 FileReader, 71
- FileStream, 72
 FileWriter, 71
 FirstApp, 10
 FlightPlan, 132
 Fourier, 64, 66
 Function, 48, 49
 Gambling, 92, 93
 GameModel, 123
 GameOfLife, 123
 GameView, 123
 GeoApp, 105
 GeoPosition, 103, 104, 107
 GeoRoute, 107–110
 GeoTrack, 107, 109, 110, 115
 Graphics, 60, 112
 Graphics2D, 112
 ImageIO, 36
 ImageSize, 44
 Imaging, 37–43
 InputStream, 71
 IOException, 36
 Iterator, 50
 JCheckBox, 56
 JOptionPane, 26, 30, 68, 141
 JPanel, 60, 63, 123
 JTextField, 56
 LinearAlgebraException, 68
 Logarithm, 48
 Lottery, 28, 33
 Math, 13, 31, 105, 127, 131
 MathFunctions, 22
 Matrix, 31, 68
 Media, 46, 47
 MediaApp1, 44
 MediaLib, 44, 46, 47
 Member, 22, 23
 MouseListener, 112, 120
 MouseMotionListener, 112, 115
 MultiBandit, 89, 93, 99
 MultiBanditApp, 89, 93, 95
 MultiBanditSolver, 93, 95
 MusicNote, 75
 MusicNotes, 23
 News, 50
 NewsApp, 52
 NewsChannel, 51, 52
 NewsProvider, 50–52
 NewsReceiver, 52
 Object, 47
 Observable, 138, 139
 Observer, 123, 138, 139
 Ownable, 135–137
 Parabola, 20, 21, 48, 57
 Person, 18, 19, 22, 26, 32, 134, 135

- PersonQueue, 32
- PlotPanel, 63, 65, 66
- Point2D, 84
- Point[], 33
- PrimeNumbers, 77, 78
- PrimeNumberThread, 77
- PrintLines, 10
- Random, 87
- RandomArray, 27, 28
- Reader, 71
- ResonantCircuit, 129
- RouteData, 108, 109
- Scanner, 88
- SegmentDisplay, 29
- Signal, 62, 64, 66, 74, 75
- Song, 18, 19, 26
- String, 19, 49, 136
- StringBuilder, 27
- Subject, 123
- TestComplex, 127
- TestGeoPosition, 104
- TestPoint2D, 84
- TestVector2D, 131
- TimePeriod, 19, 20
- UnitTests, 139
- User, 134–137
- Users, 134, 136
- Vector2D, 130, 132
- Video, 44–47
- Warehouse, 49, 50
- WheelApp, 92, 93
- WheelOffFortune, 92, 93
- ZeroCrossing, 21, 22, 33, 49, 58
- Klassenmethoden, *siehe* Methoden
- Klassenvariable, *siehe* Variable
- Komplexe Zahlen, 126
 - Betrag, 127
 - Imaginärteil, 127
 - Phase, 127
 - Realteil, 127
- Konsolen-Eingabe, 88
- Konstruktor, 18
 - Standard, 24
- Koordinaten
 - Breitengrad, 102
 - geografische, 102, 107, 111, 114
 - Längengrad, 102
- Koordinatensystem, 57
- Kosinus, *siehe* Funktion
- Kreis, 14
- Kreisfrequenz, 129
- Kreiszahl, 12, 14, 15
- Künstliche Intelligenz, 85
- Lager
 - Container, 50
- Layout-Manager, *siehe* Grafische Benutzeroberfläche
- Leibniz
 - Gottfried Wilhelm, 12
- Leibniz-Reihe, *siehe* Reihenentwicklung
- Leistung
 - elektrische, 11
- Listen, 32
- Logarithmus, *siehe* Funktion, 64
- Logik-Pegel, 12
- logischer Wert, 17
- Lookup table, 66
- Lotto, 28, 33
- Luftdruck, 62
- LUT, *siehe* Lookup table
- Längengrad, *siehe* Koordinaten
- Matrix, 31
 - Matrix-Vektor-Multiplikation, 30
 - Multiplikation, 31, 68
 - Rotation, 30
- Medien-Bibliothek, 44
- Mehrarmiger Bandit, *siehe* Bandit
- Mehrwertsteuer, 56, 68
- Menü, *siehe* Grafische Benutzeroberfläche
- Methode
 - add(), 44, 75, 129
 - addBanditResponse(), 95
 - addContainer(), 50
 - addItem(), 137
 - addMouseListener(), 112, 120
 - addMouseMotionListener(), 112
 - addParticipant(), 136
 - additem(), 137
 - addWaypoint(), 108
 - atan(), 127
 - atan2(), 127
 - bisection(), 21
 - chooseGreedy(), 95
 - chooseRandom(), 95
 - Collections.sort(), 109
 - compareTo(), 49, 53
 - contains(), 28
 - createFlightRoutes(), 109
 - createItem(), 137
 - createNews(), 51
 - createNewUser(), 136
 - deregisterReceiver(), 51, 52
 - determineWin(), 87, 92, 93
 - dft(), 64
 - digitToSignals(), 29
 - displayNext(), 52

distanceInKm(), 105, 106
 Double.parseDouble(), 68
 drawImage(), 112
 drawNumbers(), 28
 enter(), 32
 equalContents(), 27
 equals(), 47
 f(), 20, 48
 f1(), 20, 48
 factorial(), 22
 getAverageWin(), 95
 getCount(), 20
 getDistance(), 108, 110
 getDurationString(), 46
 getFirstName(), 135
 getFullName(), 19
 getItems(), 137
 getLastValue(), 20
 getMaxPixelValue(), 40
 getMeanProfitPerRound(), 87, 89
 getMinPixelValue(), 40
 getName(), 87
 getNews(), 51
 getNumberBandits(), 89
 getNumberFields(), 93
 getNumberWaypoints(), 108
 getOverallProfit(), 87, 89
 getOwner(), 135
 getParticipants(), 136
 getPricePerRound(), 87, 89
 getPrimes(), 77
 getRoundsPlayed(), 87, 89
 getSpectrumLog2(), 64
 getSubstring(), 27
 getSurname(), 135
 getTitle(), 19
 getUsers(), 136
 getWaypoint(), 108
 getWaypoints(), 108
 getX0(), 21
 gradient(), 42
 gradientX(), 41
 gradientY(), 41
 hasNext(), 50, 52
 ImageIO.read(), 112
 invert(), 40
 isPrime(), 77
 leave(), 32
 localDistanceInKm(), 105
 main(), 10, 38, 68, 123, 128, 129, 132, 147
 mouseDragged(), 115
 multiply(), 31, 68
 newton(), 21, 22, 33
 next(), 50
 nextInt(), 88
 onNews(), 51
 paintComponent(), 60, 123
 play(), 87, 89
 pop(), 32
 print(), 10, 21, 29, 31, 44
 printArchive(), 51
 printf(), 12
 println(), 10
 printRotatedVector(), 31
 push(), 32
 random(), 13
 read(), 36
 registerReceiver(), 51, 52
 removeContainer(), 50
 removeItem(), 137
 removeParticipant(), 136
 removeWaypoint(), 108
 reset(), 50
 resetCounters(), 20
 run(), 77, 78, 123
 setFunction(), 21
 setGreedyEpsilon(), 95
 setNextUserID(), 135
 setObserver(), 138
 setPixels(), 38
 setSamplesSine(), 62
 setStroke(new BasicStroke(x)), 112
 showInputDialog(), 26, 30
 showMessageDialog(), 68, 141
 sleep(), 122
 sobel(), 43
 sort(), 28
 sum(), 31
 System.out.printf(), 12, 14
 this(), 24
 throwDice(), 20
 toArray(), 108, 136
 tone2FrequencyHz(), 23, 75
 toString(), 24, 26–28, 32, 44–47, 49, 50, 105,
 107, 108, 128, 132, 134–137
 update(), 123, 138
 updateFrequencyDomain(), 64
 voltageOverResistor(), 129
 write(), 36
 writeFilePNG(), 38
 writeUInt16(), 74
 writeWavFile(), 75
Methoden
 automatisch erzeugen, 104
 Klassenmethoden, 22
 Objektmethoden, 18
Mittelwert, 85
Model-View-Control, 123

- Multithreading, *siehe* Threads
 Musik
 A-Moll, 75
 Akkord, 75
 Musikinstrument, 23
 Musiktitel, 18, 44
 MVC, *siehe* Model-View-Control
 Nachricht, 50
 Negativ, *siehe* Bild
 Newtonverfahren, 22, 33, 49, 58
 Nordpol, 102
 Normalverteilung, *siehe* Gaußverteilung
 Nullmeridian, 102
 Nullstelle, 16, 21, 22, 33, 48, 58
 Observer, 50
 Observer Pattern, 60, 123, 138
 Oktavraum, 23
 Operator, 11
 Optionsfeld, *siehe* Radio button
 Overloading, *siehe* Überladen
 Overriding, *siehe* Überlagern
 Paket, 10
 Parallelverarbeitung, *siehe* Threads
 PCM, *siehe* Pulse Code Modulation
 Person, 18
 Phasenverschiebung, 62
 Pixel, 35, 44
 Polynom, *siehe* Funktion
 Primzahl, 77
 Programmfluss, 14
 Programmierrichtlinien, *siehe* Software-Qualität
 Progress bar, *siehe* Grafische Benutzeroberfläche
 Pulse Code Modulation, 74
 Punktmenge, 59
 Punktoperation, *siehe* Bildverarbeitung
 Queue, 32
 Radio button, *siehe* Grafische Benutzeroberfläche
 Radius, 14
 Rauschen
 additiv, 66
 Rauschreduktion, 61
 Rechenzeit
 Reduktion, 65
 Referenzvariable, *siehe* Variable
 Reihenentwicklung
 Euler-Zahl, 15
 Leibniz, 12, 15
 mathematische, 15
 Reinforcement Learning, 85
 Rotationsmatrix, *siehe* Matrix
 Route, *siehe* Wegstrecke
 Satz des Pythagoras, 103
 Schaltjahr, 14
 Scheduler, *siehe* Threads
 Schieberegler, *siehe* Slider
 Schlüsselwort
 abstract, 53
 abstract final, 53
 break, 17
 catch, 69
 continue, 17
 do/while, 14, 17
 else, 153
 extends, 47
 false, 21, 50, 77
 final, 47
 finally, 69
 for, 14, 17
 foreach, 46
 if, 14
 import, 153
 null, 49
 package, 153
 private, 44, 46, 47, 53, 55, 86, 104, 107, 126,
 130, 134, 138
 protected, 47, 55, 92
 public, 47, 86
 static, 53, 106, 133
 switch, 17
 synchronized, 78
 this, 24
 throws, 68, 75
 true, 21, 27, 28, 77
 try/catch/finally, 69
 while, 14, 15, 17
 Schwingkreis
 elektrisch, 128
 Sensor, 138
 Signal
 Sinus, 61
 Signalverarbeitung, 61
 Sinus, *siehe* Funktion
 Slider, *siehe* Grafische Benutzeroberfläche
 Slot Machine, *siehe* Bandit
 Sobel-Operator, *siehe* Bildverarbeitung
 Software-Qualität, 153
 Sortieren, 49
 alphabetisch, 49
 Spannung, 11, 13, 128
 Spannungspiegel, 12
 Spannungsteiler, 12

- Spektrum, 61
- Spiel des Lebens, 118, 121
- Spielstandsanzeige, 13, 29
- Stack, 32
- Standardabweichung, 85
- Standardkonstruktor, *siehe* Konstruktor
- Steuerleitung, 29
- Störsignal, 61, 66
- Stoppuhr, 78
- Stream, *siehe* Strom
- Strom
 - Eingabe, 71
- Stromfluss, 11, 12
- Subklasse, 47
- Sudoku, 71
- Südpol, 102
- Synthesizer, 23
- Tastatur, *siehe* Eingabe
- Tavata, 13
- Tests
 - Black Box, 149
 - Deutung der Ergebnisse, 147
 - JUnit, 149
 - Unit-Tests, 145, 149
 - Vergleich von Zeichenketten, 147
 - White Box, 149
- Textdatei, *siehe* Dateien
- Threads, 77, 97
 - Interface-basiert, 78
 - Klassen-basiert, 78
 - Scheduler, 78
- Ton
 - Frequenz, 23
 - Grundton, 23
 - Höhe, 23
 - Kammerton, 75
 - Musik, 23, 75
 - Tonika, 23
- Tonika, *siehe* Ton
- Tonleiter
 - C-Dur, 23
- Überladen, 47
- Überlagern, 47
- Umfang
 - Kreis, 14
- Unit-Test, *siehe* Tests
- Variable
 - Instanzvariable, 18
 - Klassenvariable, 22, 47
 - Referenzvariable, 24
- Vektor, 30
- Betrag, 131
- Drehung, 30
- mathematisch, 130
- skalare Multiplikation, 131
- Skalarprodukt, 132
- Vererbung, 46
- Verzeichnisse, 71
- Video, 35, 44
 - Bildgröße, 44
 - Dauer, 44
- Warteschlange, 32
- WAVE-Datei, *siehe* Dateien
- Wegstrecke, 107, 111
- Wertebereich
 - short*, 73
 - byte*, 15
- White Box-Test, *siehe* Tests
- Widerstand, 11, 13, 128
- Wrapper-Klasse, 26
- Würfel, 13, 14, 19, 29, 55
- Zeichenbereich
 - Hintergrundfarbe, 57
- Zeichenkette, 26
- Zeitbereich, 61–63
- Zeitdauer, 19
- Zinsen, 14
- Zufallszahl, 13, 27, 29, 85