Karlsruher Institut für Technologie

Programmieren WS 2012/2013

Forschungsgruppe: Verifikation trifft Algorithmik
http://verialg.iti.uka.de/prog_ws12.php
Dr. Carsten Sinz, Florian Merz, Markus Iser

Übungsblatt 2
(v1.0)

Ausgabe:	05.11.2012
Abgabe:	19.11.2012, 13Uhr
Besprechung:	26.11.2011 - 30.11.2011

Allgemeine Hinweise

Informieren Sie sich auf `http://www.oracle.com/technetwork/java/codeconv-138413.html` über Programmierrichtlinien in `Java`, und befolgen Sie diese Richtlinien auf diesem und allen weiteren Übungsblättern genau. Das Praktomat-System wird ab diesem Übungsblatt beim Hochladen der Lösung automatisch die Einhaltung einiger dieser Richtlinien kontrollieren. Zunächst ist das Ergebnis dieser Prüfungen nur informativ, aber auf zukünftigen Blättern können Lösungen, die diese Tests nicht bestehen, vom Praktomat zurückgewiesen werden. Üben Sie also jetzt schon die fehlerfreie Umsetzung der Programmierrichtlinien.

Der Praktomat benutzt `Checkstyle`[1] um die Einhaltung der Programmierrichtlinien zu kontrollieren. Checkstyle ist als Kommandozeilenprogramm erhältlich sowieso als Plugin für die meisten `Java`-Entwicklungsumgebungen. Nutzen Sie dies, um schon beim Programmieren zu erfahren, ob Sie die Richtlinien einhalten und nicht erst, wenn Sie die Lösung in den Praktomaten hochladen.

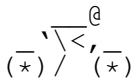
Auf `http://baldur.iti.uka.de/programmieren` können Sie die `Checkstyle`-Regeln herunterladen, die auch im Praktomaten Verwendung finden. Für dieses Übungsblatt wird nur der Whitespace-Regelsatz im Praktomat aktiviert sein.

In diesem Übungsblatt führen wir zum ersten Mal unbenotete Zusatzfragen ein. Die Beantwortung dieser Fragen ist optional und wird nicht bewertet. Die Lösungen der Zusatzfragen sollten Sie daher nicht im Praktomaten abgeben. Die Zusatzfragen werden nach Ablauf der Abgabefrist in den Tutorien diskutiert.

[1] `http://checkstyle.sourceforge.net/`

A Kontrollstrukturen (10 Punkte)

In Teil A des Übungsblatts sollen Sie lernen mit Kontrollstrukturen in `Java` umzugehen. Überlegen Sie sich gut, welche der Strukturen `if-then-else`, `switch-case`, `for`, `while` und `do-while` am geeignetsten für die Lösung der jeweiligen Aufgabe ist. Beachten Sie, dass für das Lösen dieser Teilaufgabe Arrays noch nicht benötigt werden und auch nicht benutzt werden sollten.



A.1 Bedingte Ausführung

Dieser Teil baut auf einer Aufgabe aus dem vorangegangenen Übungsblatt auf. Um die Korrektur zu vereinfachen, benutzen Sie bitte unseren Lösungsvorschlag[2] und *nicht* Ihre eigene Lösung als Grundlage der Lösung dieser Teilaufgabe.

A.1.1 Konstruktor von `Bike`

Modifizieren Sie den Konstruktor der Klasse `Bike`[2], indem Sie aus der Signatur das Argument, nach dem bisher der Preis gesetzt wird, entfernen. Setzen Sie stattdessen im Konstrutor den Preis des Fahrrads in Abhängigkeit vom Rahmenmaterial, und zwar so, dass Fahrräder mit Alu-Rahmen 200 €, Fahrräder mit Stahl-Rahmen 300 € und Fahrräder mit Titan-Rahmen 400 € kosten.

A.1.2 Setter für `Gears`

Sie sollen nun in einer `set`-Methode sicherstellen, dass der Zustand der Klasse `Gears` einer Menge von Konsistenzbedingungen genügt. Fügen Sie der Klasse `Gears`[2] eine Methode mit folgender Signatur hinzu, die die Attribute für die Anzahl der Kettenräder vorne und hinten auf die Werte ihrer beiden Argumente `front` und `rear` setzt:

```
void setSprockets(int front, int rear) {
  // add your code here
}
```

Stellen Sie in dieser Methode sicher, dass am Ende die folgenden Konsistenzbedingungen gelten:

A.1 Die Anzahl der Kettenräder vorne ist positiv.

A.2 Die Anzahl der Kettenräder vorne ist echt kleiner als 4.

B.1 Die Anzahl der Kettenräder hinten ist positiv.

B.2 Die Anzahl der Kettenräder hinten ist echt kleiner als 10.

C.1 Die Anzahl der Kettenräder hinten ist mindestens so groß wie die Anzahl der Kettenräder vorne.

[2] Verfügbar unter `http://baldur.iti.kit.edu/programmieren/` ab Montag, den 05.11.2012, nachmittags.

C.2 Die Anzahl der Kettenräder hinten ist höchstens dreimal so groß wie die Anzahl der Kettenräder vorne.

Falls diese Konsistenzbedingungen eingehalten werden, weisen Sie den Attributen nur die Werte der Argumente zu. Sonst stellen Sie die Konsistenzbedingungen in folgender *Reihenfolge* wieder her:

- Falls Bedingung **A.1** verletzt ist, setzen Sie die Anzahl der Kettenräder vorne auf 1.
- Falls Bedingung **A.2** verletzt ist, setzen Sie die Anzahl der Kettenräder vorne auf 3.
- Falls Bedingung **B.1** verletzt ist, setzen Sie die Anzahl der Kettenräder hinten auf die Anzahl der Kettenräder vorne.
- Falls Bedingung **B.2** verletzt ist, setzen Sie die Anzahl der Kettenräder hinten auf die Anzahl der Kettenräder vorne multipliziert mit 3.
- Falls Bedingung **C.1** verletzt ist, setzen Sie die Anzahl der Kettenräder hinten auf die kleinstmögliche Zahl, sodass alle Bedingungen erfüllt sind.
- Falls Bedingung **C.2** verletzt ist, setzen Sie die Anzahl der Kettenräder hinten auf die größtmögliche Zahl, sodass alle Bedingungen erfüllt sind.

Schreiben Sie den Konstruktor so um, dass er die Methode `setSprockets` benutzt, um einen konsistenten Anfangszustand zu garantieren.

Nicht bewertete Zusatzfrage

Welche Bedingungen könnten Sie aus der Aufgabenstellung entfernen ohne die Menge der gültigen und ungültigen Kombinationen an Kettenrädern zu beeinflussen? Gibt es dann immer noch in jedem Fall die gleichen Ergebnisse beim Wiederherstellen der Konsistenzbedingungen?

A.2 Schleifen

A.2.1 Tribonacci-Folge

Hintergrundwissen

Die Fibonacci-Folge ist nach Leonardo Fibonacci — einem der bedeutendsten Mathematiker des Mittelalters — benannt. Er benutzte diese Zahlenfolge um das Wachstum einer Kaninchenpopulation mathematisch zu modellieren. Die Zahlenfolge war auch schon den Menschen in der Antike bekannt. Viele Phänomene aus der Natur lassen sich mit ihr beschreiben. Interessanterweise nähert sich der Quotient zweier aufeinanderfolgender Fibonacci-Zahlen dem goldenen Schnitt an. Was ist das für eine Zahlenfolge? Sie beginnt mit zwei Einsen, und jede weitere Zahl ergibt sich aus der Summe der beiden vorhergehenden Zahlen. Mathematisch würde man sie so definieren:

$$a_0 = 1$$
$$a_1 = 1$$
$$a_n = a_{n-1} + a_{n-2}, \text{ für } n \geq 2$$

Die Fibonacci-Folge beginnt also so: $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$

Wir definieren nun eine sogenannte Tribonacci-Folge. Diese soll anstatt mit zwei mit **drei** Einsen beginnen, d.h. die erste (a_0), zweite (a_1) und dritte (a_2) Zahl haben alle den Wert 1. Jede weitere Zahl soll sich dann aus der Summe der **drei** vorhergehenden Zahlen ergeben. Also, um es nochmal präzise auszudrücken, die Elemente a_i der Tribonacci-Folge sind wie folgt definiert:

$$
\begin{aligned}
a_0 &= 1 \\
a_1 &= 1 \\
a_2 &= 1 \\
a_n &= a_{n-1} + a_{n-2} + a_{n-3}, \text{ für } n \geq 3
\end{aligned}
$$

Schreiben Sie eine Klasse `Tribonacci` mit einer statischen Methode (z.B. `main`), welche die ersten siebenunddreißig Zahlen der Tribonacci-Folge mittels einer Schleife berechnet. Geben Sie **nur** die siebenunddreißigste Tribonacci-Zahl auf der Konsole aus.

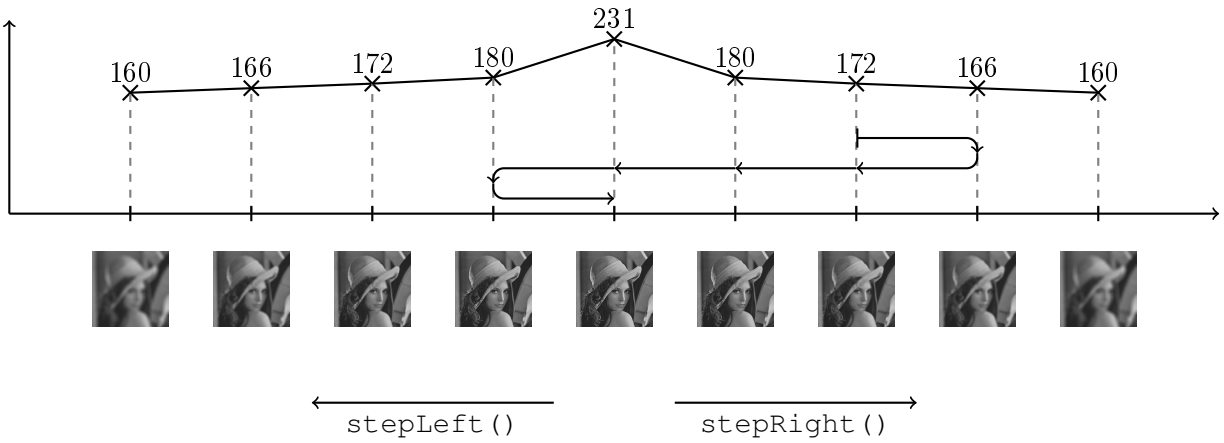
? Nicht bewertete Zusatzfrage

Was passiert, wenn Sie die achtunddreißigste Tribonacci-Zahl berechnen wollen? Wie können Sie das Problem beheben? Wann tritt es dann wieder auf?

A.2.2 Autofokus

Viele digitale Kameras fokussieren automatisch mithilfe des Kontrastvergleichsverfahrens. Ähnlich wie beim händischen Fokussieren wird dabei der Fokussierring so lange in die Richtung gedreht, in die der Kontrast zunimmt, bis dieser wieder abnimmt. Dann wird wieder zurückgedreht, um die optimale Schärfe einzustellen.

Abbildung 1: Kontrast-basierter Autofokus einer Kamera



Im Folgenden sollen Sie dieses automatische Fokussieren algorithmisch als `Java`-Programm beschreiben. Wir haben Ihnen dazu eine Klasse `Objective` zur Verfügung gestellt, über die Sie simulierten Zugriff auf Sensordaten und Schrittmotor des Objektivs einer Kamera erhalten. Die Methode `getContrast()` gibt die aktuelle Bildschärfe als `double` zurück (je größer der Wert, um so schärfer das Bild). Die beiden Methoden `stepLeft()` und `stepRight()` steuern den Schrittmotor, der den Fokus verschiebt. Sie können davon ausgehen, dass Sie immer ein eindeutiges Kontrast-Maximum finden, und dass die Funktion links und rechts davon streng monoton steigend bzw. streng monoton fallend ist. Wenn Sie am "Rand" anstoßen (d.h. der Fokussierring lässt sich nicht weiterdrehen), bleibt der Kontrast konstant (Benutzen Sie für Gleichheitstests in dieser Aufgabe ein Epsilon von $\epsilon = 10^{-6}$).

Implementieren Sie Ihren Autokfokus-Algorithmus in folgender Methode:

```
class Camera {
  Objective objective;

  public Camera(Objective objective) {
    this.objective = objective;
  }

  void autofocus() {
    // add your code here
  }
}
```

Dabei sollen die Methoden des `Objective`-Objekts benutzt werden, um den aktuellen Kontrast auszulesen und um den Schrittmotor in eine Richtung zu drehen. Der Algorithmus soll folgendes bewerkstelligen:

- Feststellen, in welcher Richtung der Kontrast zunimmt.
- Solange schrittweise in diese Richtung drehen, bis der Kontrast wieder abnimmt.
- Einen Schritt zurückdrehen, um den optimalen Kontrast wiederherzustellen.

Benutzen Sie die Klasse `Objective` in Ihrer Lösung, geben Sie diese Klasse aber *nicht* mit Ihrer Lösung ab.

B Arrays (10 Punkte)

In dieser Teilaufgabe sollen Sie lernen mit Arrays in `Java` umzugehen. Sie werden lernen über 1-dimensionale und 2-dimensionale Arrays zu iterieren, Werte auszulesen und Werte zu setzen. In der Teilaufgabe B.1 werden Sie zunächst den Umgang mit Arrays von elementaren Typen üben, in Teilaufgabe B.2 dann den Umgang mit Arrays von Objekten.

B.1 Kontrastberechnung

In dieser Teilaufgabe sollen Sie eine Methode schreiben, die für ein Graustufenbild ein einfaches Kontrastmaß berechnet. Der Kontrast eines Bildes gibt den Helligkeitsunterschied zwischen den hellsten und den dunkelsten Bereichen eines Bildes an. Es gibt verschiedene Möglichkeiten den Kontrast eines Bildes zu bemessen. In dieser Aufgabe benutzen wir die einfachste Möglichkeit: Die Differenz der Helligkeiten des hellsten und des dunkelsten Bildpunktes.

Da Sie noch nicht gelernt haben echte Bilddateien zu öffnen, werden wir diese durch zwei-dimensionale Arrays von `int`-Werten simulieren. Jede Zahl in diesem Array stellt dabei einen Bildpunkt dar, wobei der Wert der Zahl die Helligkeit des Bildes an dieser Position repräsentiert. Eine Wert von 0 steht dabei für einen komplett schwarzen Bildpunkt, ein Wert von 255 für einen komplett weißen Bildpunkt. Werte kleiner 0 oder größer 255 kommen in der Eingabe nicht vor und müssen daher nicht behandelt werden.

Abbildung 2: Kontrastarme Bilddaten, sowie deren Histogramm und Kontrastwert

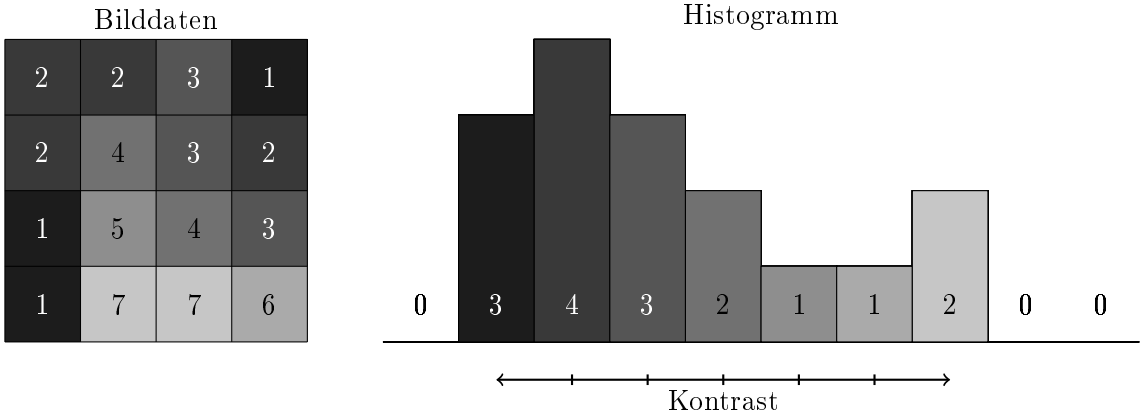
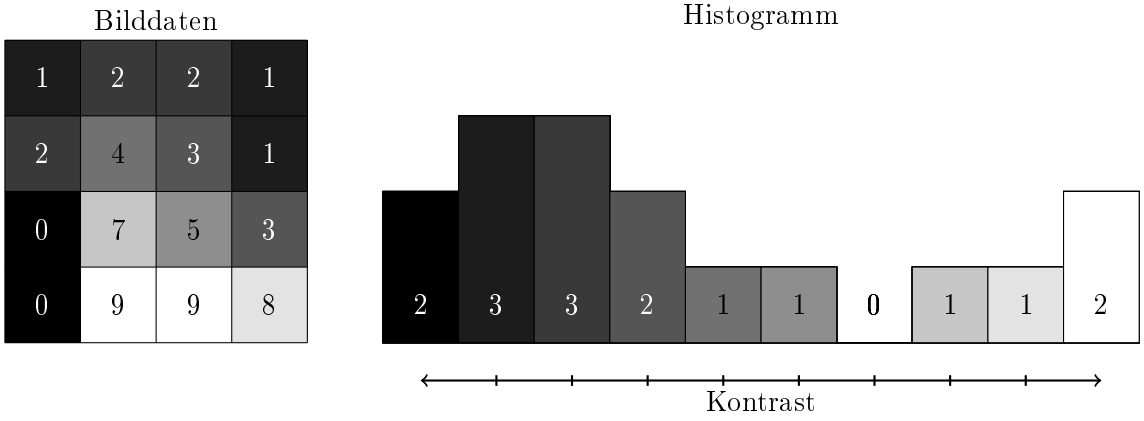


Abbildung 3: Kontrastreiche Bilddaten, sowie deren Histogramm und Kontrastwert



Um nun den Kontrast eines solchen Bildes zu bestimmen, sollen Sie zunächst ein *Histogramm* des Bildes erstellen. In einem Histogramm wird die Häufigkeit des Vorkommens der unterschiedlichen Helligkeitswerte im Bild gezählt. Aus diesem Histogramm sollen Sie dann jeweils den hellsten und den dunkelsten Helligkeitswert bestimmen. Die Differenz dieser beiden Werte ergibt dann den Kontrast des Bildes.

Implementieren Sie Ihren Algorithmus zur Kontrastberechnung in folgender Methode:

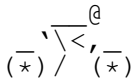
```
class Contrast {
    static int calculateContrast(int[][] image) {
        // add your code here
    }
}
```

Halten Sie sich dabei an folgende Vorgehensweise:

- Erstellen Sie für das Histogramm ein `int`-Array der Größe 256, welches an allen Positionen mit dem Wert 0 initialisiert ist. Der i-te Wert in diesem Array soll dabei angeben, wie häufig ein Bildpunkt mit der Helligkeit i im Bild vorkommt.
- Iterieren Sie in Ihrer Methode über alle Pixel des Eingabe-Arrays. Für jeden Bildpunkt den Sie besuchen inkrementieren (erhöhen) Sie den Wert des Histogramm-Arrays an der Position der Helligkeit dieses Bildpunktes um eins.
- Nun sollten Sie aus dem Histogramm-Array für jede Helligkeitsstufe die Häufigkeit auslesen können, mit der diese Helligkeit im Originalbild vorkam. Testen Sie dies.
- Iterieren Sie nun vorwärts (der Index wird also bei jeder Iteration größer) über das Histogramm, bis Sie im Array auf einen Wert ungleich 0 treffen. Halten Sie den aktuellen Index in einer Variablen fest.
- Wiederholen Sie dies, diesmal jedoch rückwärts, also mit kleiner werdendem Index, bis sie hier ebenfalls einen Wert ungleich 0 finden. Halten Sie diesen Index ebenfalls fest.
- Der Kontrast ist nun die Differenz der beiden ermittelten Werte. Berechnen Sie den Kontrast und geben Sie ihn als Rückgabewert der Funktion zurück. Beachten Sie, dass der Kontrast immer positiv ist.

B.2 Fahrradladen

Erstellen Sie eine Klasse `BikeShop`, die einen Fahrradladen modelliert. Modellieren Sie das Fahrradlager dieses Ladens, indem Sie der Klasse ein Array von Elementen des Typs `Bike` als Attribut hinzufügen. Jedes Feld in diesem Array stellt dabei einen Platz im Lager dar. Fügen Sie der Klasse ein weiteres Attribut hinzu, das die Anzahl der Fahrräder im Lager kodiert.



B.2.1 Konstruktor

Schreiben Sie zunächst einen Konstrutor für den Fahrradladen. Schaffen Sie in Ihrem Fahrradlager Platz für zehn Fahrräder, indem Sie das Array-Attribut mit einem neuen Array der Größe 10 initialisieren. Zunächst ist das Lager noch leer, setzen Sie also die Anzahl der Fahrräder auf 0.

B.2.2 Fahrrad ankaufen

Schreiben Sie eine Methode, die den Zukauf eines Fahrrads modelliert. Diese Methode hat als Argument eine Variable vom Typ `Bike` und Rückgabetyp `void`. Die Methode soll folgendes bewerkstelligen:

- Die übergebene `Bike`-Referenz hinter dem letzen Fahrrad im Fahrradlager verstauen, falls im Lager noch Platz für mindestens ein weiteres Fahrrad ist.
- Falls nicht mehr ausreichend Platz im Lager ist, sollen Sie vor dem Hinzufügen des Fahrrads die Größe des Fahrradlagers zunächst um 5 Stellplätze vergrößern.

  Sie können ein Array vergrößern, indem Sie es durch ein neues, größeres Array ersetzen. Vergessen Sie dabei nicht die Fahrräder aus dem alten Array in das neue Array zu kopieren.

Sorgen Sie immer dafür, dass die tatsächliche Anzahl der Fahrräder im Fahrradlager mit dem zugehörigen Attribut der Fahrradladen-Klasse konsistent ist.

B.2.3 Fahrrad verkaufen

Schreiben Sie eine Methode, die den Verkauf eines Fahrrads modelliert. Diese Methode hat als Argument eine Zahl, die den Index des verkauften Fahrrads angibt und gibt das verkaufte Fahrrad selbst zurück. Falls am übergebenen Platz kein Fahrrad lagert, geben Sie die `null`-Referenz zurück. Implementieren Sie die Methode so, dass am Ende das verkaufte Fahrrad nicht mehr im Array auftaucht. Falls durch den Verkauf eines Fahrrads eine Lücke im Array entstehen, arrangieren Sie die restlichen Fahrräder so um, dass diese Lücke geschlossen wird. Die Reihenfolge der Fahrräder muss dabei nicht erhalten bleiben.

Verkleinern Sie das Array immer, wenn mehr als fünf Felder frei sind. Verfahren Sie hierbei analog zur Vergrößerung des Lagers.

Hinweise zur Abgabe

Geben Sie für die Teilaufgabe A.1 die Dateien `Bike.java` und `Gears.java` ab. Für Teilaufgabe A.2.1 geben sie bitte eine Datei `Tribonacci.java` ab. Geben Sie für die Teilaufgabe A.2.2 nur die Datei `Camera.java` ab, nicht jedoch die Datei `Objective.java`, und für Teilaufgabe B.1 die Datei `Contrast.java`. Für Teilaufgabe B.2 geben Sie bitte die Datei `BikeShop.java` ab. Achten Sie darauf, dass alle von Ihnen abgegebenen Dateien mit `javac`[3] fehlerfrei kompilieren und die Überprüfungen aus dem Whitespace-Regelsatz bestehen.

[3] Version 1.7