Programmierkurs Java

Dr. Dietrich Boles

Aufgaben zu UE34-Interfaces
(Stand 28.09.2012)

Aufgabe 1:

Stellen Sie sich folgendes Szenario vor: Eine Person ist immer an wichtigen Ereignissen interessiert. Deshalb implementiert sie ein Interface `NachrichtenEmpfaenger`:

```
interface NachrichtenEmpfaenger {
     // Uebergabe einer neuen Nachricht
     public void empfangeNachricht(String nachricht);
}
```

Stellen Sie sich vor, dass ein Nachrichten-Empfänger eine empfangene Nachricht im einfachsten Fall auf den Bildschirm ausgibt.

Nachrichten können wiederum von verschiedenen Quellen erzeugt werden, z.B. Radio, TV, Zeitung. Die Fähigkeit, Nachrichten zu versenden, lässt sich somit auch in ein Interface `NachrichtenQuelle` abstrahieren:

```
interface NachrichtenQuelle {

     // Interessierte können sich bei der Quelle anmelden
     // (falls sie noch nicht angemeldet sind)
     public void anmelden(NachrichtenEmpfaenger empf);

     // Angemeldete können sich bei der Quelle wieder abmelden
     // (falls sie angemeldet sind)
     public void abmelden(NachrichtenEmpfaenger empf);

     // neue Nachricht wird an alle angemeldeten
     // Empfaenger uebergeben
     // (Aufruf deren Methode empfangeNachricht)
     public void sendeNachricht(String nachricht);
}
```

Etwas umständlich an dieser Struktur ist, dass sich ein Nachrichten-Empfänger bei jeder Nachrichten-Quelle anmelden muss, von der er neue Nachrichten zugeschickt bekommen möchte. Günstiger wäre ein Vermittler, der sich bei mehreren Nachrichten-Quellen anmeldet und alle Nachrichten, die er von den Quellen bekommt, direkt an Nachrichten-Empfänger weiterleitet, die sich bei ihm angemeldet haben. Ein Vermittler ist damit sowohl Nachrichten-Empfänger als auch Nachrichten-Quelle.

Aufgabe: Definieren und implementieren Sie eine solche Klasse `Vermittler`, die die beiden Interfaces `NachrichtenEmpfaenger` und `NachrichtenQuelle` entsprechend implementiert.

Hinweis: Zum Speichern von Nachrichten-Empfängern nutzen Sie bitte die Klasse `java.util.Vector`:

```
public class Vector {
      // kann beliebig viele Objekte speichern
      public Vector() {...}

      // abspeichern eines Objektes
      public void addElement(Object obj) {...}

      // entfernen eines Objektes
      public void removeElement(Object obj) {...}

      // ist Object enthalten?
      public boolean contains(Object obj) {...}

      // liefert die Anzahl an gespeicherten Objekten
      public int size() {...}

      // liefert an interner Position i gespeichertes Object;
      // die aktuell gespeicherten Objekte belegen dabei die
      // internen Positionen 0 bis size()-1
      public Object elementAt(int i) {...}
}
```

Aufgabe 2:

Schauen Sie sich das folgende Interface sowie die beiden folgenden Klasse an:

```
interface Vergleichbar {
       /*
        * vergleicht das aufgerufene Objekt mit dem als Parameter uebergebenen
        * Objekt; liefert: -1 falls das aufgerufene Objekt kleiner ist als das
        * Parameterobjekt 0 falls beide Objekte gleich gross sind 1 falls das
        * aufgerufene Objekt groesser ist als das Parameterobj.
        */
       public int vergleichenMit(Vergleichbar obj);
}

class NuetzlicheFunktionen {

       // liefert ein "kleinstes" (auf der Basis der
       // Vergleichbar-Implementierung!) Element des Parameter-Arrays
       // Achtung: Man kann davon ausgehen, dass das Parameter-Array
       // mindestens 1 Element enthaelt
       public static Vergleichbar kleinstesElement(Vergleichbar[] elemente) {
              // todo
       }
}

class Integer {
       protected int wert;
```

```
	public Integer(int w) {
		this.wert = w;
	}

	public int getWert() {
		return this.wert;
	}
}
```

Aufgabe:

1. Implementieren Sie die Methode kleinstesElement der Klasse NuetzlicheFunktionen

2. Leiten Sie von der Klasse Integer eine Klasse VInteger ab, die das Interface Vergleichbar implementiert

3. Schreiben Sie ein kleines Testprogramm, das zunächst ein Array mit VInteger-Objekten erzeugt und initialisiert, anschließend die Funktion kleinstesElement mit diesem Array aufruft und den Wert des ermittelten kleinsten Elements auf den Bildschirm ausgibt

Aufgabe 3:

In dieser Aufgabe geht es um die Verschlüsselung von Texten. Klartexte über einem Klartextalphabet werden durch die Anwendung von Verschlüsselungsalgorithmen in Geheimtexte über einem Geheimtextalphabet überführt. Verschlüsselungsverfahren sollen gewährleisten, dass nur Befugte bestimmte Botschaften lesen können. Schauen Sie sich dazu das folgende Interface an:

```
interface Chiffrierung {
	public char chiffrieren(char zeichen);
	public char dechiffrieren(char zeichen);
}
```

Das Chiffrieren ist ein Verschlüsselungsverfahren, bei dem jeder Buchstabe in einem Text durch einen anderen Buchstaben ersetzt wird. Die Methode `chiffrieren` des Interfaces soll eine entsprechende Umsetzung des übergebenen Zeichen vornehmen. Die Methode `dechiffrieren` bildet die reverse Funktion.

Ihre Aufgabe besteht nun darin, das Interface zweimal zu implementieren:

- Bei der ersten Implementierung sollen Sie die so genannte *Caesar-Verschiebung* implementieren. Die Caesar-Verschiebung beruht auf einem Geheimtextalphabet, das um eine bestimmte Stellenzahl *n* gegenüber dem Klartextalphabet verschoben ist. Beispiel für n = 3: a -> d, b -> e, c -> f, ..., w -> z, x -> a, y -> b, z -> c. Chiffriert werden sollen hier nur Kleinbuchstaben. Alle anderen Zeichen sollen unverändert zurückgegeben werden. Implementieren Sie neben den beiden Methoden des Interfaces einen Konstruktor, dem die Stellenzahl als Parameter übergeben wird
- Bei der zweiten Implementierung des Interface übergeben Sie im Konstruktor explizit das Geheimtextalphabet als 26elementiges char-Array *m*. Die

Chiffrierung erfolgt hierbei durch: a -> m[0], b -> m[1], ..., z -> m[25]. Chiffriert werden sollen auch hier nur Kleinbuchstaben. Alle anderen Zeichen sollen unverändert zurückgegeben werden (10 Punkte).

Implementieren Sie anschließend die folgende Klasse:

```
public class Verschluesselung {
  public static String verschluesseln(String klartext,
                                      Chiffrierung schluessel);
  public static String entschluesseln(String geheimtext,
                                      Chiffrierung schluessel);
}
```

zum Ver- bzw. Entschlüsseln von Botschaften gemäß eines zwischen Sender und Empfänger vereinbarten Chiffrierungsschlüssels (Hinweis: Die Klasse java.lang.String stellt eine Methode char charAt(int index) zur Verfügung, die den Charakter an der index-ten Stelle liefert)

Implementieren Sie weiterhin ein Testprogramm (= Aufruf aller Methoden) für die Klasse Verschluesselung, in dem beide Chiffrierungsverfahren eingesetzt werden

Aufgabe 4:

Sie haben in Aufgabe 3 mit der Caesarverschlüsselung ein Verschlüsselungsverfahren kennen gelernt, bei dem jeder Buchstabe der Klartextes im Geheimtext durch einen anderen Buchstaben ersetzt wird. Dieses Verschlüsselungsverfahren ist recht einfach zu knacken, und zwar mit Hilfe einer Häufigkeitsanalyse. Diese beruht darauf, dass in Texten einige Buchstaben häufiger vorkommen als andere. Man braucht also nur die Häufigkeit des Auftretens eines Buchstaben im Geheimtext zu ermitteln und kann dann daraus Schlussfolgerungen ziehen, welcher tatsächliche Buchstabe sich hinter dem Geheimtextbuchstaben verbirgt.

In dieser Aufgabe sollen Sie ein Java-Programm entwickeln, das eine Häufigkeitsanalyse durchführt. Im Detail soll das Programm folgenden Algorithmus implementieren:

- Zunächst wird ein Benutzer aufgefordert, einen String einzugeben.
- Anschließend erstellt das Programm eine Tabelle mit der prozentualen Häufigkeit des Auftretens einzelner Buchstaben (nur die Kleinbuchstaben!!!) im eingelesenen String.
- Im Anschluss daran kann ein Nutzer in einer Schleife die prozentuale Häufigkeit des Vorkommens einzelner Kleinbuchstaben im eingelesenen String abfragen.

Beispiel für ein typisches Szenario (Benutzereingaben stehen in <>):

```
String eingeben: <Dies ist ja nur wirklich eine einfache Aufgabe !>
Prozentuale Häufigkeit von: <e>
Prozentuale Häufigkeit des Buchstabens e im String = 12.76 %
Prozentuale Häufigkeit von: <b>
Prozentuale Häufigkeit des Buchstabens b im String = 2.12 %
Prozentuale Häufigkeit von: <A>
A ist kein Kleinbuchstabe!
Prozentuale Häufigkeit von: <x>
Prozentuale Häufigkeit des Buchstabens x im String = 0.0 %
.....
```

Aufgabe 5:

In dieser Aufgabe geht es wieder um die Verschlüsselung von Texten. Klartexte über einem Klartextalphabet werden durch die Anwendung von Verschlüsselungsalgorithmen in Geheimtexte über einem Geheimtextalphabet überführt. Verschlüsselungsverfahren sollen gewährleisten, dass nur Befugte bestimmte Botschaften lesen können.

Nachdem Sie in Aufgabe 3 schon zwei Verschlüsselungsverfahren kennen gelernt haben, sollen Sie nun ein drittes implementieren, und zwar die Vignere-Verschlüsselung.

Die Vignere-Verschlüsselung basiert auf dem so genannten Vignere-Quadrat:

```
Klar   a b c d e f g h i j k l m n o p q r s t u v w x y z

0      a b c d e f g h i j k l m n o p q r s t u v w x y z
1      b c d e f g h i j k l m n o p q r s t u v w x y z a
2      c d e f g h i j k l m n o p q r s t u v w x y z a b
3      d e f g h i j k l m n o p q r s t u v w x y z a b c
…
11     l m n o p q r s t u v w x y z a b c d e f g h i j k
…
24     y z a b c d e f g h i j k l m n o p q r s t u v w x
25     z a b c d e f g h i j k l m n o p q r s t u v w x y
```

Zeile 1 des Quadrates enthält ein Geheimtextalphabet mit einer Caesar-Verschiebung von 1; Zeile 2 des Quadrates enthält ein Geheimtextalphabet mit einer Caesar-Verschiebung von 2; usw.

Vorteil der Vignere-Verschlüsselung ist, dass eine Häufigkeitsanalyse zum Knacken des Geheimcodes versagt (siehe auch Aufgabe 4). Bei der Vignere-Verschlüsselung geht man nämlich so vor, dass man jeden Buchstaben einer geheim zu haltenden Botschaft anhand einer anderen Zeile des Vignere-Quadrates verschlüsselt. Um die Botschaft wieder entschlüsseln zu können, muss der Empfänger allerdings wissen, welche Zeile des Vignere-Quadrates für den jeweiligen Buchstaben benutzt wurde. Deshalb tauschen Sender und Empfänger vorher ein Schlüsselwort aus.

Nehmen wir einmal an, die zu verschlüsselnde Botschaft sei „truppenabzugnachosten“ und das Schlüsselwort sei „licht“. Zunächst wird das Schlüsselwort über die Botschaft geschrieben und so lange wiederholt, bis jeder Buchstabe der Botschaft mit einem Buchstaben des Schlüsselwortes verknüpft ist.

```
Schlüsselwort     lichtlichtlichtlichtl
Klartext          truppenabzugnachosten
Geheimtext        ezwwipvcisfoehvswuaxy
```

Der Geheimtext wird dann folgendermaßen erzeugt: Um den ersten Buchstaben „t“ zu verschlüsseln, stellen wir zunächst fest, dass über ihm der Buchstabe „l“ steht, der wiederum auf eine bestimmte Zeile des Vignere-Quadrates verweist. Die mit „l“ beginnenden Reihe 11 enthält das Geheimtextalphabet, das wir benutzen, um den Stellvertreter des Klarbuchstabens „t“ zu finden. Also folgen wir der Spalte unter „t“ bis zum Schnittpunkt mit der Zeile „l“, und dort befindet sich der Buchstabe „e“ im Geheimtext. Genauso gehen wir mit allen weiteren Buchstaben der Botschaft vor.

Die Entschlüsselung eines Zeichen eines Geheimtextes erfolgt auf umgekehrten Weg: Anhand des aktuellen Schlüsselzeichens wird die aktuelle Zeile des Quadrates bestimmt. Aus der Spalte des Zeichens leitet sich dann das Klartextzeichen ab.

Konkrete Aufgabe: Implementieren Sie eine Klasse Vignere, die das folgende Interface Verschluesselung gemäß der Vignere-Verschlüsselung implementiert. Verschlüsselt werden sollen nur Kleinbuchstaben. Alle anderen Buchstaben sollen unverändert bleiben

```
interface Verschluesselung {
  public String verschluesseln(String klartext);
  public String entschluesseln(String geheimtext);
}
```

Das folgende Gerüst sei dabei vorgegeben:

```
public class Vignere implements Verschluesselung {

      // enthaelt nur Kleinbuchstaben!
```

```
	private String schluesselwort;

	// speichert das Vignere-Quadrat
	private char[][] quadrat = null;

	// schluessel darf nur Kleinbuchstaben enthalten!
	public Vignere(String schluessel) {
		this.schluesselwort = schluessel;
		// initialisieren des Vignere-Quadrates
		this.quadrat = new char[26][26];
		for (int i = 0; i < 26; i++) {
			for (int j = 0; j < 26; j++) {
				this.quadrat[i][j] = (char) ('a' + (i + j) %
26);
			}
		}
	}

	// liefert zu einem uebergebenen Klartext unter Nutzung des im
	// Attribut schluesselwort gespeicherten Schluesselwortes den
	// entsprechenden Geheimtext
	public String verschluesseln(String klartext) {

			// muss von Ihnen implementiert werden!

	}

	// liefert zu einem uebergebenen Geheimtext unter Nutzung des
	// im Attribut schluesselwort gespeicherten Schluesselwortes
   // den entsprechenden Klartext
	public String entschluesseln(String geheimtext) {

	 // muss von Ihnen implementiert werden!

	}
}
```

Aufgabe 6:

Implementieren Sie eine Methode *check*, die (auf eine von Ihnen zu konzipierende Art und Weise) als ersten Parameter eine beliebige (unter Umständen partielle) mathematische Funktion f mit dem Definitionsbereich int und dem Wertebereich int, und als zweiten und dritten Parameter die Unter- und Obergrenze eines geschlossenen int-Intervals übergeben bekommt. Die Methode *check* soll überprüfen (und einen entsprechenden booleschen Wert liefern), ob die übergebene Funktion f im übergebenen Interval linear ist (genauer: f(x) = n*x für alle x im Interval, und n ist ein beliebiger int-Wert zwischen 1 und 100)). Ist f für einen Wert im übergebenen Interval nicht definiert, soll eine Exception geworfen werden.

Implementieren Sie weiterhin ein Testprogramm, in dem die Methode *check* getestet wird!

Aufgabe 7:

In dieser Aufgabe geht es darum, die Größe der Oberflächen zweier volumenmäßig ungefähr gleich großer 3-dimensionaler Behälterobjekte miteinander zu vergleichen. Schauen Sie sich dazu die folgenden Interfaces/Klassen an:

```
interface Behaelter {
	public double getOberflaeche();

	public double getVolumen();

	public void veraendern(double inkrement);
}

class Wuerfel implements Behaelter {
	private double kantenLaenge;

	public Wuerfel(double kantenLaenge) {
		this.kantenLaenge = kantenLaenge;
	}

	public double getOberflaeche() {
		return 6.0 * this.kantenLaenge * this.kantenLaenge;
	}

	public double getVolumen() {
		return this.kantenLaenge * this.kantenLaenge *
this.kantenLaenge;
	}

	public void veraendern(double inkrement) {
		this.kantenLaenge += inkrement;
	}
}

class Kugel implements Behaelter {
	static final double PI = 3.1415;

	private double radius;

	public Kugel(double radius) {
		this.radius = radius;
	}

	public double getOberflaeche() {
		return 4.0 * PI * this.radius * this.radius;
	}

	public double getVolumen() {
		return 4.0 / 3.0 * PI * this.radius * this.radius *
this.radius;
	}

	public void veraendern(double inkrement) {
		this.radius += inkrement;
	}
}
```

Aufgabe: Schreiben Sie durch Benutzung dieser Klassen (sowie der bekannten Klasse IO) ein Java-Programm, das folgendes tut:

- Zunächst werden ein Wuerfel- und ein Kugel-Objekt mit jeweils einer Größe (Kantenlänge bzw. Radius) erzeugt, die zuvor vom Benutzer abgefragt wird. Achten Sie darauf, dass der Nutzer "ordentliche" Werte eingibt.

- Anschließend wird das Volumen der beiden Objekte auf den Bildschirm ausgegeben

- Danach wird ein Inkrementwert inkrement vom Typ double vom Benutzer abgefragt.

- Anschließend werden in einer Schleife die Volumen der beiden Objekte "angeglichen", und zwar auf folgende Art und Weise: Das anfangs volumenmäßig größere Objekt wird in jedem Schleifendurchlauf um den Inkrementwert verkleinert (Methode veraendern) und das anfangs volumenmäßig kleinere Objekt wird um den Inkrementwert vergrößert. Die Schleife wird solange durchlaufen, bis das anfangs volumenmäßig größere Objekt volumenmäßig kleiner ist als das anfangs volumenmäßig kleinere Objekt.

- Zum Schluss werden Volumen und Oberfläche der beiden Objekte auf den Bildschirm ausgegeben.

Im Folgenden wird ein Beispiel für eine mögliche Ausgabe des Programms gegeben (in <> die Eingaben des Benutzers):

```
Kantenlaenge: <10.0>
Radius: <9.0>
Wuerfel-Volumen (Anfang): 1000.0
Kugel-Volumen (Anfang): 3053.5
Inkrement: <0.001>
Wuerfel-Volumen: 1612.3
Wuerfel-Oberfläche: 824.9
Kugel-Volumen: 1612.1
Kugel-Oberflaeche: 664.8
```

Aufgabe 8:

Schreiben Sie ein Programm, das folgende GUI erzeugt:

Im linken Teil befindet sich ein Label (java.awt.Label) mit dem Text „Eingabe“, im rechten Teil ein TextField (java.awt.TextField), in dem der Nutzer einen Text eingeben kann.

Immer wenn ein Nutzer einen Text eingegeben und mit <RETURN> abgeschlossen hat, soll die entsprechende Eingabe auf der Console ausgegeben werden. Weiterhin soll im TextField der eingegebene Text gelöscht werden.

Die Behandlung von Nutzereingaben erfolgt analog zu dem im Unterricht behandelten Verfahren zur Registrierung von Button-Klicks.

Aufgabe 9:

Gegeben sei das folgende Java-Programm:

```
class Tierstimmen {
	public static void main(String[] args) {
		Tier[] tiere = new Tier[5];
		for (int i = 0; i < 5; i++) {
			tiere[i] = erzeugeTier();
		}
		for (int i = 0; i < 5; i++) {
			tiere[i].gibLaut();
		}
	}

	public static Tier erzeugeTier() {
		String eingabe = IO.readString("Tier (Hund/Katze): ");
		if (eingabe.equals("Hund")) {
			return new Hund();
		} else {
			return new Katze();
		}
	}
}
```

Implementieren Sie geeignete Interfaces bzw. Klassen `Hund` und `Katze`, damit sich das Programm compilieren lässt und für Katzen "Miau" und für Hunde "Wau wau!" auf den Bildschirm ausgibt.

Aufgabe 10:

Schauen Sie sich das folgende Java-Programm an:

```
import java.awt.BorderLayout;
import java.awt.Button;
import java.awt.Frame;
import java.awt.Label;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Vector;

/*
 aus dem AWT:

 public interface ActionListener {
     public void actionPerformed(ActionEvent e);
 }
 */
```

```
public class UE34Aufgabe10 {

	public static void main(String[] args) {
		MyFrame f = new MyFrame();
		f.setBounds(200, 200, 100, 80);
		f.setVisible(true);
		// ab hier ist der Event-Manager des AWT aktiv
	}
}

interface StateChangeListener {
	public void stateHasChanged();
}

final class MyFrame extends Frame implements ActionListener,
		StateChangeListener {

	private Button button;

	private Label label;

	private int numberOfClicks;

	private Vector<StateChangeListener> listeners;

	public MyFrame() {
		super("MyFrame");
		this.numberOfClicks = 0;
		this.listeners = new Vector<StateChangeListener>();
		this.button = new Button("Please click");
		this.label = new Label("" + this.numberOfClicks);

		// fuegt Button und Label in das Fenster ein
		this.add(this.button, BorderLayout.NORTH);
		this.add(this.label, BorderLayout.SOUTH);

		// registriert Listener
		this.addStateChangeListener(this);
		this.button.addActionListener(this);
	}

	// wird aufgerufen, wenn der Button geklickt wird
	public void actionPerformed(ActionEvent e) {
		this.numberOfClicks++;

		// alle Interessenten der Zustandsaenderung werden
	    // informiert
		for (StateChangeListener elem : this.listeners) {
			elem.stateHasChanged();
		}
	}

	// fuegt ein Objekt ein, das Interesse daran hat,
	// ueber Zustandsaenderung der Klickanzahl informiert zu werden
	public void addStateChangeListener(StateChangeListener listener) {
		this.listeners.add(listener);
	}

	// das Fenster selbst ist auch daran interessiert,
	// ueber Zustandsaenderungen der Klickanzahl informiert zu werden;
	// falls dies passiert, wird der Text des Labels auf die aktuelle
	// Anzahl der Button-Klicks gesetzt
```

```
        public void stateHasChanged() {
              this.label.setText("" + this.numberOfClicks);
        }

        // liefert die Anzahl an Klicks auf den Button
        public int getNumberOfClicks() {
              return this.numberOfClicks;
        }
}
```

Führt man dieses Programm aus, erscheint auf dem Bildschirm ein neues Fenster mit einem Button und einem Label. Jedes Mal, wenn der Benutzer auf den Button klickt, wird die Zahl im Label um 1 erhöht. In der folgenden Abbildung hat der Benutzer fünfmal auf den Button geklickt.

Aufgabe: Erweitern Sie das Programm derart, dass jedes Mal, wenn der Benutzer auf den Button klickt, zusätzlich die aktuelle Klickanzahl auch auf der Standard-Ausgabe (System.out) ausgegeben wird (siehe folgende Abbildung).

Aber Achtung: Sie dürfen die Klasse `MyFrame` nicht verändern! Implementieren Sie stattdessen einen neuen *StateChangeListener* und registrieren diesen beim Fenster.

Aufgabe 11:

Gegeben sei folgender Ausschnitt aus einem Java-Programm:

```
class Vogel {
      public void fliegen() {
      }
}

class Katze {
      public void laufen() {
      }
}

class Fisch {
      public void schwimmen() {
      }
}

class Tierwelt {

      public static void bewegen(Object object) {
            if (object.getClass().getName().equals("Vogel"))
                  ((Vogel) object).fliegen();
            else if (object.getClass().getName().equals("Katze"))
                  ((Katze) object).laufen();
```

```
            else if (object.getClass().getName().equals("Fisch"))
                  ((Fisch) object).schwimmen();
      }
}
```

Zum Hintergrund des Programms: Ein Programmierer A möchte anderen Programmierern in einer Klasse *Tierwelt* Funktionen zur Verfügung stellen, die bestimmte Methoden von Tier-Klassen aufrufen. Im konkreten Beispiel implementiert er eine Funktion *bewegen*, die die entsprechende "Bewegungsmethode" des übergebenen Objektes aufruft.

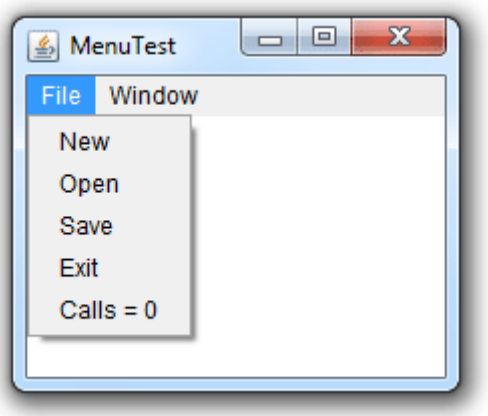
Es gibt aber ein Problem: Immer wenn ein Programmierer eine weitere Tier-Klasse hinzufügt, muss Programmierer A den Source-Code der Funktion *bewegen* ändern.

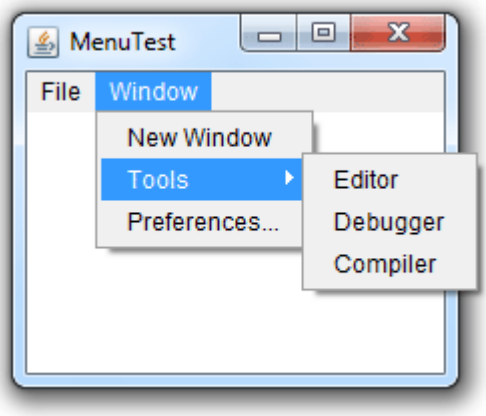
Aufgabe: Helfen Sie Programmierer A! Schreiben Sie die Funktion *bewegen* und evtl. auch andere Teile des obigen Programms so um, dass die Funktion *bewegen* immer die korrekte "Bewegungsmethode" einer Tier-Klasse aufruft, und zwar ohne dass ihr Source-Code geändert werden muss, wenn eine neue Tier-Klasse hinzukommt! Nutzen Sie dazu die Konzepte der Polymorphie und des dynamischen Bindens.

Achtung: Sie dürfen in Ihrem Programm keine if-, switch-, while-, for- und do-Anweisungen verwenden!

Aufgabe 12:

In dieser Aufgabe geht es um die Erstellung einer GUI:





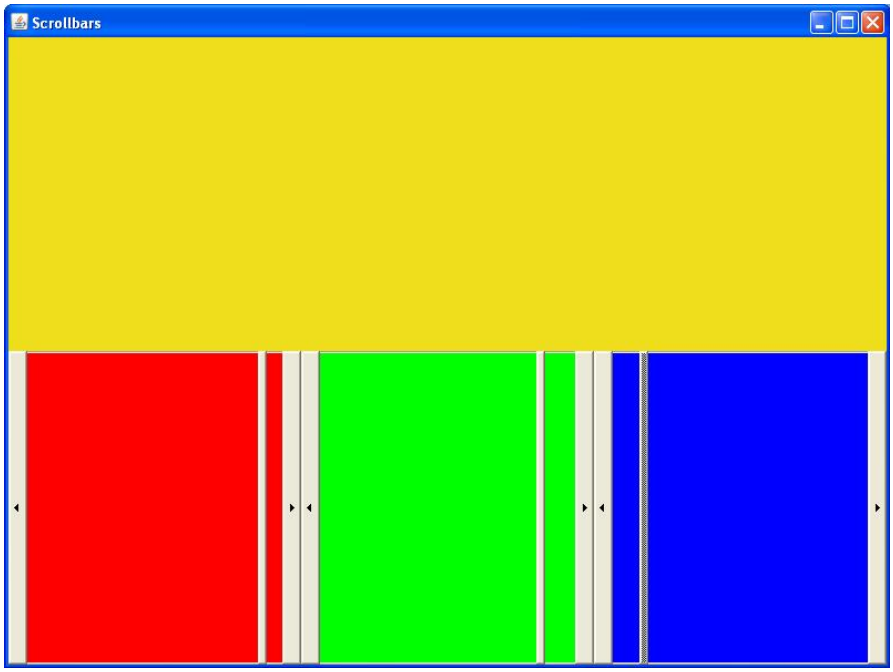
Aufgabe (a): Schreiben Sie ein kleines Programm, das ein Fenster mit den oben dargestellten Menüs erzeugt. Schauen Sie sich dazu die Klassen `java.awt.Frame`, `java.awt.MenuBar`, `java.awt.Menu` und `java.awt.MenuItem` an. Von besonderer Bedeutung sind die Methoden `setMenuBar` der Klasse `Frame` und die Methode `add` der Klasse `MenuBar` und der Klasse `Menu`.

Aufgabe (b): Realisieren Sie, dass beim Auswählen des letzten Menü-Items des File-Menüs der Label geändert wird. Und zwar soll dort immer die Anzahl der Aufrufe

dieses Items angezeigt werden („Calls = 1“, Calls = 2“, …). Hinweis: Das Vorgehen zum Zuordnen einer Aktion zu einem MenuItem ist identisch mit dem Zuordnen einer Aktion zu einem Button.

Aufgabe 13:

Entwickeln Sie ein Java-Programm mit einer interaktiven GUI, bei der ein Benutzer mit Hilfe dreier Scrollbars die Farbe eines Panels beeinflussen kann.



Im oberen Bereich des Fensters (Klasse `java.awt.Frame`) befindet sich ein Panel (Klasse `java.awt.Panel`), darunter befinden sich drei Scrollbars (Klasse `java.awt.Scrollbar`). Anfangs ist das Panel schwarz und die Regler der drei Scrollbars befinden sich jeweils ganz links. Sie sollen nun folgende Benutzereingriffe implementieren:
Die Farbe des Panels soll sich ergeben aus dem Stand der Regler der drei Scrollbars. Verschiebt der Benutzer den Regler des linken Scrollbars nach rechts, wird der Rot-Wert des Panels erhöht, verschiebt er den Regler nach links, wird der Rot-Wert erniedrigt. Entsprechend der RGB-Farben bewegt sich der Wert immer zwischen 0 (Regler links, kein Rot) und 255 (Regler rechts, volles Rot). Der mittlere Scrollbar regelt analog den Farbanteil von grün, der rechte den Farbanteil von blau.

Hinweise:

- Schauen Sie sich in der Java-API-Dokumentation die Klassen `java.awt.Panel` und `java.awt.Scrollbar` an.
- Farben lassen sich in Java durch die Klasse `java.awt.Color` repräsentieren.
- Die Farbe eines Panels sowie die Farbe von Scrollbars werden jeweils über die Methode `setBackground` festgelegt.
- Die Registrierung und Behandlung von Bewegungen eines Scrollbar-Reglers erfolgt durch so genannte `AdjustmentListener`. Diese müssen einem Scrollbar über die Methode `addAdjustmentListener` zugeordnet werden.

Das Prinzip ist analog zum Prinzip der `ActionListener`, die wir in den Vorlesungen für Buttons und MenuItems kennen gelernt haben. Schauen Sie sich in der Java-API-Dokumentation das Interface `java.awt.event.AdjustmentListener` und die Klasse `java.awt.event.AdjustmentEvent` an.

Aufgabe 14:

Definieren Sie eine ADT-Klasse `UKWRadio` zur Repräsentation eines UKW-Radios. Der Zustand des Radios ist gegeben durch eine Frequenz zwischen 87.5 und 108.0 MHz. Nach dem Erzeugen des Radio-Objektes beträgt die Frequenz 87.5 MHz. Die Frequenz kann in den angegebenen Frequenzen schrittweise um 0.5 MHz nach oben bzw. unten verändert werden. Weiterhin stehen N (> 0) Stationstasten zur Verfügung, auf denen man Frequenzen speichern bzw. gespeicherte Frequenzen wieder abrufen kann.

Insgesamt soll Ihre Klasse also die folgenden Methoden definieren:

- Einen Konstruktor, dem die Anzahl an Stationstasten übergeben wird
- Eine Methode, die die aktuelle Frequenz liefert
- Eine Methode zum Verringern der Frequenz um 0.5 MHZ
- Eine Methode zum Erhöhen der Frequenz um 0.5 MHZ
- Eine Methode, um die aktuelle Frequenz auf einer Stationstaste zu speichern
- Eine Methode, um die aktuelle Frequenz auf die Frequenz einer angegebenen Stationstaste einzustellen

Entwickeln Sie weiterhin ein Testprogramm für die Klasse `UKWRadio` mit einer graphisch-interaktiven Oberfläche analog zur folgenden Abbildung (2 Stationstasten).

Aufgabe 15:

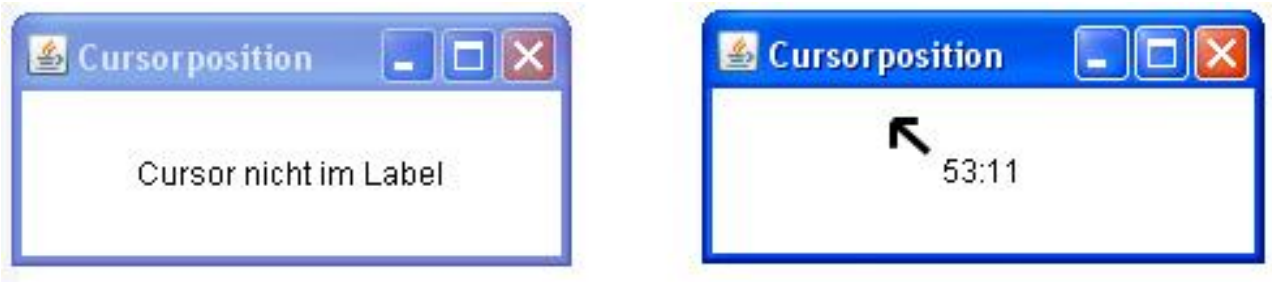
Schauen Sie sich die Klasse `java.util.Observable` und das Interface `java.util.Observer` an.

Leiten Sie zunächst von der Klasse `Observable` eine Klasse `Number` ab, in der eine Zahl (int) beobachtet wird. Immer wenn sich diese Zahl ändert, sollen Observer darüber informiert werden.

Schreiben Sie eine GUI mit einem Button und einem Label. Bei einem Button-Klick soll eine beobachtete Number erhöht werden. Die geänderte Zahl soll dann anschließend sowohl in dem Label als auch auf der Console ausgegeben werden. Realisieren Sie dies über das Interface `Observer`.

Aufgabe 16:

Entwickeln Sie ein Java-Programm, das ein Fenster auf dem Bildschirm öffnet. In dem Fenster ist als einzige Komponente ein Label (eine GUI-Komponente zur Darstellung von Strings) platziert, das anfangs den String „Cursor nicht im Label“ darstellt. Sobald der Benutzer den Mauscursor oberhalb des Labels bewegt, sollen jeweils die aktuellen Mauskoordinaten in der Form „x:y“ im Label erscheinen. Wird der Mauscursor aus dem Label entfernt, soll wieder der String „Cursor nicht im Label“ dargestellt werden.



Das Fenster ist eine Instanz der Klasse `java.awt.Frame`, die Sie aus der Veranstaltung bereits kennen. Ansonsten benötigen Sie folgende Klassen und Methoden:

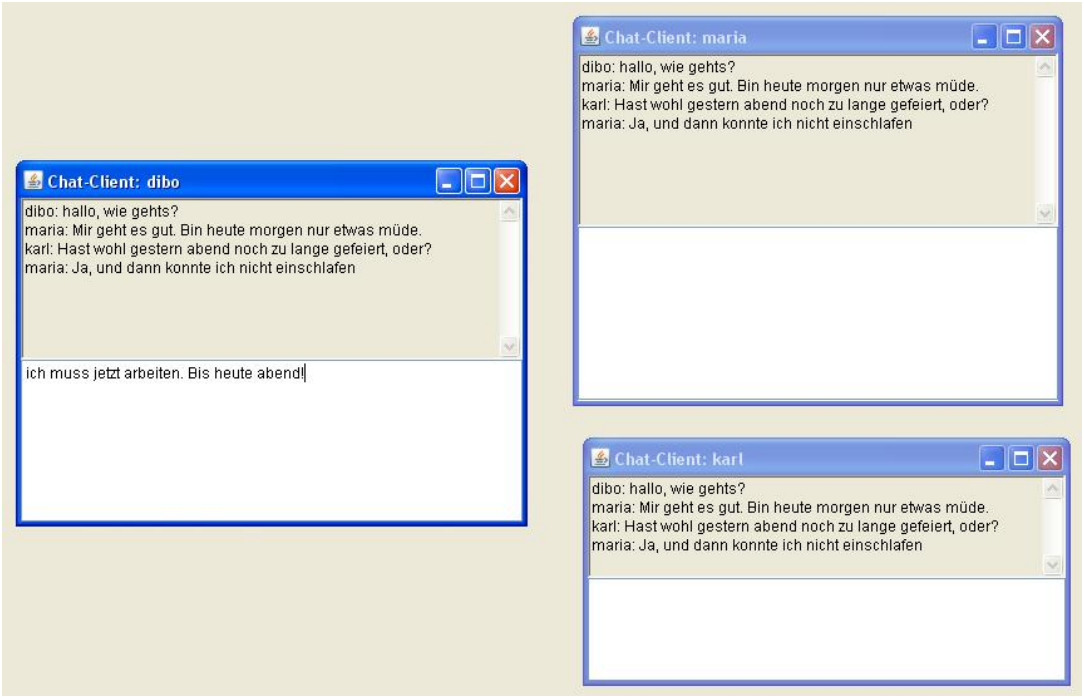
```
public class Label extends java.awt.Component {  // aus dem Paket java.awt

       // stellt den übergebenen String im Label dar
       public Label(String str)

       // ändert den im Label dargestellten String
       public void setText(String str)

       // registriert bei dem Label einen MouseListener
       public void addMouseListener(MouseListener l)

       // registriert bei dem Label einen MouseMotionListener
       public void addMouseMotionListener(MouseMotionListener l)
}


public class MouseEvent {  // aus dem Paket java.awt.event

       // liefert die aktuelle x-Koordinate des Mauscursors relativ zur betroffenen
       // Komponente
       public int getX()

       // liefert die aktuelle y-Koordinate des Mauscursors relativ zur betroffenen
       // Komponente
       public int getY()
}
```

```
public interface MouseListener {  // aus dem Paket java.awt.event

        // wird aufgerufen, wenn der Mauscorsor eine Komponente verlässt;
        // über das Event-Objekt werden weitergehende Informationen zum aktuellen
        // Status der Maus übergeben
        public void mouseExited(MouseEvent e);

        // wird aufgerufen, wenn der Mauscorsor eine Komponente betritt;
        // über das Event-Objekt werden weitergehende Informationen zum aktuellen
        // Status der Maus übergeben
        public void mouseEntered(MouseEvent e);

}

public interface MouseMotionListener { // aus dem Paket java.awt.event

        // wird aufgerufen, wenn die Maus oberhalb einer Komponente bewegt wird;
        // über das Event-Objekt werden weitergehende Informationen zum aktuellen
        // Status der Maus übergeben
        public void mouseMoved(MouseEvent e);
}
```

Der Mechanismus zum Agieren bei entsprechenden Mausoperationen erfolgt analog zum aus der Veranstaltung bekannten Reagieren auf Mausklicks auf Buttons.

Aufgabe 17:

Bei dieser Aufgabe sollen Sie ein einfaches Chat-System realisieren. Dieses besteht aus einem Chat-Server, bei dem sich prinzipiell beliebig viele Chat-Clients anmelden bzw. auch wieder abmelden können.



Hinweise:

- Implementieren Sie 3 Klassen: `ChatServer`, `ChatClient`, `ChatSystem`.

- Die Klasse `ChatServer` ist von der Klasse `java.util.Observable` abgeleitet. Ein Objekt der Klasse realisiert das Zusammenspiel der Clients, die sich als Observer bei ihm angemeldet haben.
- Die Klasse `ChatClient` ist ein Fenster (`java.awt.Frame`), das das Interface `java.util.Observer` implementiert. Es enthält im oberen Teil eine Anzeigekomponente (`java.awt.TextArea`) und im unteren Teil eine Eingabekomponente (`java.awt.TextField`). Gibt ein Benutzer in der Eingabekomponente einen Text ein und schließt ihn mit der <RETURN>-Taste ab, soll der Text inklusive des Benutzernamens als Präfix in den Anzeigekomponenten aller beim Chat-Server angemeldeter Chat-Clients erscheinen. Der Text der Eingabekomponente wird wieder gelöscht.
- Die Klasse `ChatSystem` ist das Hauptprogramm. Es wird mit `java ChatSystem <user1> <user2> … <userN>` aufgerufen. Es erzeugt einen Chat-Server und für alle als Parameter übergebenen Benutzernamen einen Chat-Client. Die Chat-Clients werden jeweils beim Chat-Server als Observer angemeldet.

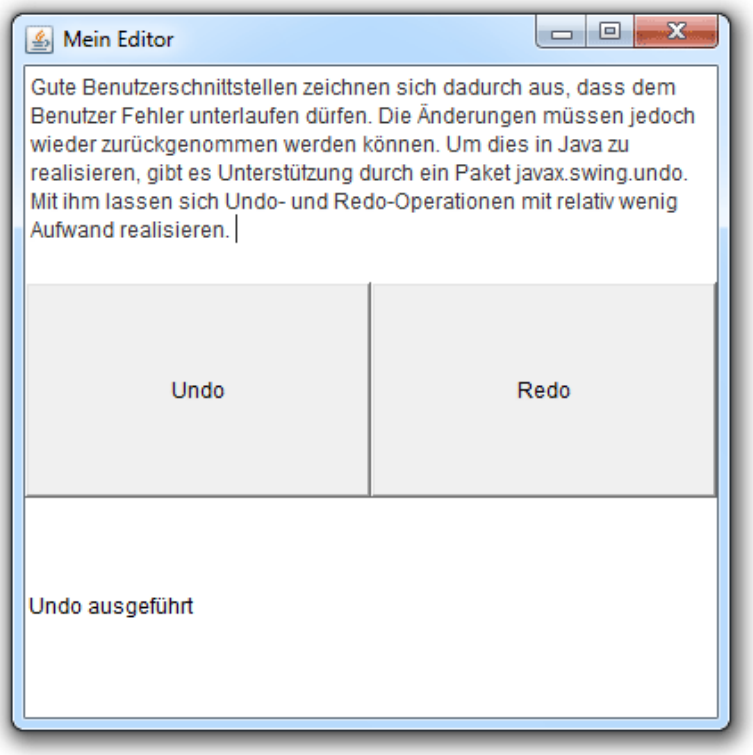
Aufgabe 18:

Bei dieser Aufgabe sollen Sie einen graphischen Editor mit Undo-Redo-Funktionalität implementieren. Konkret soll die zu erstellende GUI aus folgenden Komponenten bestehen:

- oben einer `javax.swing.JTextPane`-Komponente (das ist eine Java-Swing-Komponente mit vorgefertigter Editor-Funktionalität)
- darunter zwei `java.awt.Button`-Objekte, links für das Undo, rechts für das Redo.
- unten eine `java.awt.Label`-Komponente

In der JTextPane-Komponente kann ein Benutzer über die Tastatur Eingaben tätigen. Drückt er auf den Undo-Button soll, falls möglich, die letzte in der JTextPane-Komponente getätigte Aktion rückgängig gemacht werden und in jedem Fall in der Label-Komponente eine passende Ausgabe erscheinen. Drückt er auf den Redo-Button soll, falls möglich, die letzte Undo-Aktion rückgängig gemacht werden und in jedem Fall in der Label-Komponente eine passende Ausgabe erscheinen.

Hinweis: Im JDK steht bereits im Paket `javax.swing.undo` eine Klasse `UndoManager` zur Verfügung, die Methoden für ein Undo bzw. Redo sowie Methoden zum Überprüfen, ob die entsprechenden Aktionen überhaupt möglich sind, definiert. Schauen Sie sich diese Klasse genau an. Um ein UndoManager-Objekt mit Ihrer JTextPane-Komponente zu verknüpfen, nutzen Sie bitte die Methode `editor.getStyledDocument().addUndoableEditListener` (`editor` sei dabei eine Referenz auf Ihre JTextPane-Komponente).



Aufgabe 19:

In dieser Aufgabe geht es um die Verschlüsselung von Texten. Klartexte über einem Klartextalphabet werden durch die Anwendung von Verschlüsselungsalgorithmen in Geheimtexte über einem Geheimtextalphabet überführt. Verschlüsselungsverfahren sollen gewährleisten, dass nur Befugte bestimmte Botschaften lesen können.

Eine der beiden grundlegenden Verschlüsselungsklassen ist die *Transposition*. Dabei werden die Zeichen einer Botschaft (des Klartextes) umsortiert. Jedes Zeichen bleibt zwar unverändert erhalten, jedoch wird die Stelle, an der es steht, geändert. Als sehr einfaches und anschauliches Beispiel einer geregelten Transposition soll hier die *Gartenzaun-Transposition* dienen: Die Buchstaben des Klartextes werden abwechselnd auf zwei Zeilen geschrieben, so dass der erste auf der oberen, der zweite auf der unteren, der dritte Buchstabe wieder auf der oberen Zeile steht und so weiter. Der Geheimtext entsteht, indem abschließend die Zeichenkette der unteren Zeile an die der oberen Zeile angefügt wird. (aus Wikipedia)

Beispiel:

Klartext: `Gartenzaun`
Verfahren:

```
G r e z u
 a t n a n
```

Geheimtext: `Grezuatnan`

Aufgabe:

Definieren und implementieren Sie eine Klasse `GartenzaunTransposition`, die das folgende Interface `Verschluesselung` mit dem Verfahren der Gartenzaum-Transposition implementiert.

```
interface Verschluesselung {

     // liefert den verschluesselten Text
     String verschluesseln(String klartext);

     // liefert den entschluesselten Text
     String entschluesseln(String geheimtext);
}
```

Aufgabe 19:

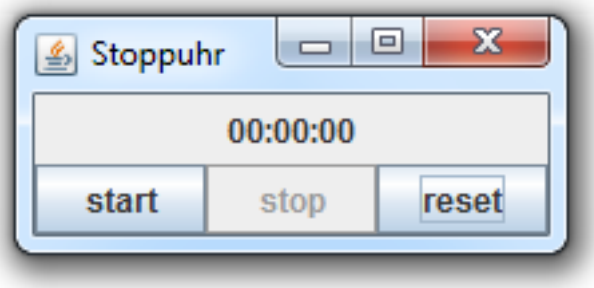
Teilaufgabe (a):

Implementieren Sie eine ADT-Klasse `Clock`, die Uhrzeiten von Stoppuhren (siehe auch Aufgabe 3) repräsentiert. Intern sollen dabei Stunden (0-23), Minuten (0 – 59) und Sekunden (0 – 59) verwaltet werden. Beachten Sie dabei die bekannten Umrechnungsregeln für Stunden, Minuten und Sekunden (zum Beispiel: 61 Minuten = 1 Stunde + 1 Minute). Die Klasse soll folgende Methoden zur Verfügung stellen:

- Einen Default-Konstruktor, der ein Clock-Objekt mit 00:00:00 initialisiert.
- Einen Konstruktor, der ein Clock-Objekt mit Werten für die Stunden, Minuten und Sekunden initialisiert, die ihm als Parameter übergeben werden.
- Einen Copy-Konstruktor.
- Eine Methode `clone` zum Klonieren eines Clock-Objektes (Überschreiben der von der Klasse `Object` geerbten Methode `clone`).
- Eine Methode `equals` zum Überprüfen der Gleichheit zweier Clock-Objekte (Überschreiben der von der Klasse `Object` geerbten Methode `equals`).
- Eine Methode `toString`, die eine String-Repräsentation des aktuellen Clock-Objektes liefert (Überschreiben der von der Klasse `Object` geerbten Methode `toString`). Die String-Repräsentation soll dabei folgendermaßen aufgebaut sein: `hh:mm:ss` (bspw. 23:08:34 für 23 Stunden, 8 Minuten und 34 Sekunden).
- Eine Methode `addSeconds`, die eine als Parameter übergeben Anzahl an Sekunden zur aktuell repräsentierten Uhrzeit addiert.
- Eine Klassenmethode `add`, die zwei Clock-Objekte als Parameter übergeben bekommt und ein Clock-Objekt erzeugt und liefert, das die Summe der Uhrzeiten der beiden Parameter-Objekte repräsentiert.
- Eine Methode `reset`, die die interne Uhrzeit auf 00:00:00 zurücksetzt.

Teilaufgabe (b):

Entwickeln Sie eine Stoppuhr als GUI-Applikation, die die in Aufgabe 2 implementierte ADT-Klasse `Clock` nutzt.



Die GUI besteht dabei aus einem Fenster, in dem (entsprechend obiger Abbildung) im oberen Teil die von einem Clock-Objekt repräsentierte Uhrzeit (anfangs 00:00:00) angezeigt wird. Im unteren Teil gibt es drei Buttons (start, stop, reset) zum Bedienen der Stoppuhr.

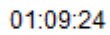
- Klickt der Benutzer den start-Button, wird die Stoppuhr gestartet, d.h. nach Verstreichen jeder realen Sekunde wird die Uhr bzw. die Uhranzeige automatisch um eine Sekunde erhöht.
- Klickt der Benutzer den stop-Button, wird die Stoppuhr gestoppt.
- Klickt der Benutzer den reset-Button, wird die Ausgangssituation wieder hergestellt.

Beachten Sie: Wenn die Stoppuhr läuft, soll es nicht möglich sein, den start-Button zu klicken. Wenn die Stoppuhr nicht läuft, soll es nicht möglich sein, den stop-Button zu drücken (Methode `setEnabled`).

Hinweis: Nutzen Sie zum automatischen Erhöhen der Uhrzeit die Klassen `java.util.Timer` und `java.util.TimerTask`. Schauen Sie sich in der JDK-Dokumentation an, was diese Klassen leisten und wie sie zu nutzen sind.

Aufgabe 20:

Laden Sie sich zunächst die Datei `clock.zip` herunter und entpacken Sie sie. Nehmen Sie dann die Datei `clock.jar` in den BuildPath Ihres Eclipse-Klausurprojektes auf. Schauen Sie sich weiterhin die API der Klassen und Interfaces der jar-Datei an, indem Sie die Datei `doc/index.html` in Ihrem Web-Browser öffnen. Die jar-Datei enthält u.a. eine Klasse `ClockLabel`, die in einem Java-AWT-Label die jeweils aktuelle Uhrzeit (Stunden, Minuten, Sekunden) anzeigt.



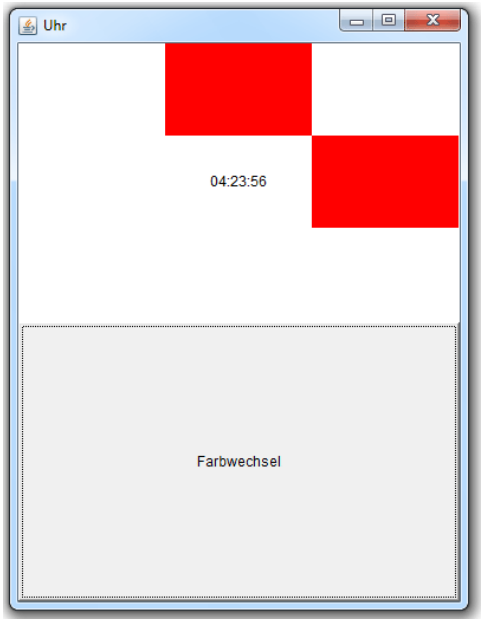
Teilaufgabe (a): Implementieren Sie eine neue GUI-Komponentenklasse namens `DigitalClock`, die von `java.awt.Panel` abgeleitet sein soll. Eine DigitalClock besteht dabei aus 3 * 3 Teilbereichen. Diese Teilbereiche enthalten folgende Inhalte:

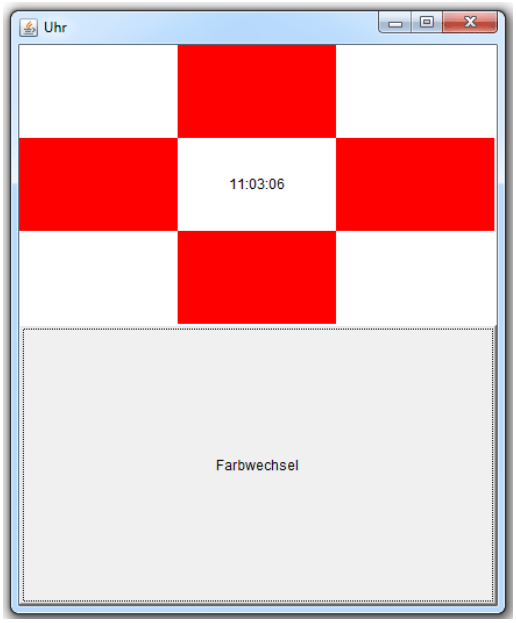
- In der Mitte befindet sich ein ClockLabel.
- Die Teilbereiche links oben, links unten, rechts oben und rechts unten haben keine Bedeutung (leere weiße Panel)
- Teilbereich oberhalb des ClockLabel: ein Panel mit farbigem (Standardfarbe ist rot) Hintergrund zwischen 1 Uhr (einschließlich) und 0 Uhr (ausschließlich), ansonsten weiß.
- Teilbereich rechts vom ClockLabel: ein Panel mit farbigem (Standardfarbe ist rot) Hintergrund zwischen 4 Uhr (einschließlich) und 0 Uhr (ausschließlich), ansonsten weiß.
- Teilbereich unterhalb des ClockLabel: ein Panel mit farbigem (Standardfarbe ist rot) Hintergrund zwischen 7 Uhr (einschließlich) und 0 Uhr (ausschließlich), ansonsten weiß.
- Teilbereich links vom ClockLabel: ein Panel mit farbigem (Standardfarbe ist rot) Hintergrund zwischen 10 Uhr (einschließlich) und 0 Uhr (ausschließlich), ansonsten weiß.

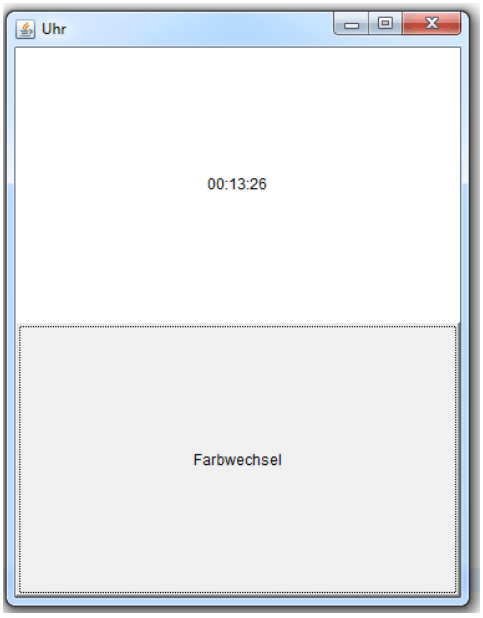
Nutzen Sie zur Realisierung der zeitbedingten Farbumstellungen innerhalb der Klasse `DigitalClock` die Möglichkeit, ClockListener bei dem ClockLabel zu registrieren (siehe API-Dokumentation).

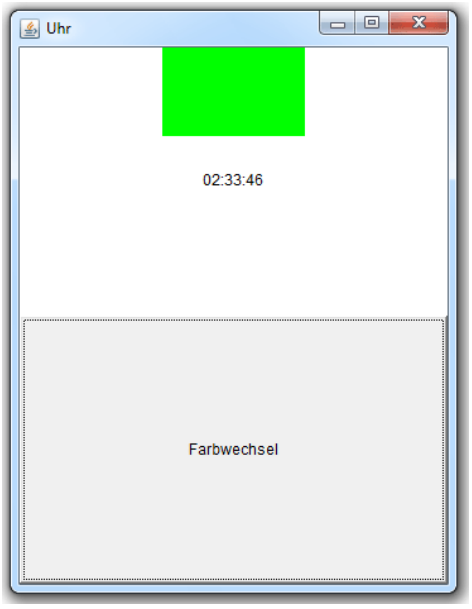
Teilaufgabe (b): Implementieren Sie dann eine Java-GUI-Anwendung mit einem Fenster, das, wie in den folgenden Abbildungen skizziert, eine DigitalClock und einen Button mit der Beschriftung „Farbwechsel“ enthalten soll. Bei Klick auf den Farbwechsel-Button soll die Farbe der farbig dargestellten Panel der DigitalClock wechselweise zwischen rot und grün umgeschaltet werden können.

Beispiele:



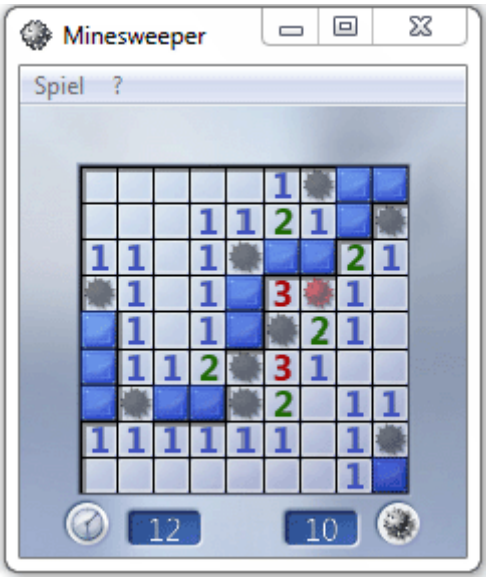






Aufgabe 21:

Sie kennen sicher das Spiel *Minesweeper*. Es ist ein Computerspiel, bei dem ein Spieler durch logisches Denken herausfinden muss, hinter welchen Feldern einer N * M-Matrix Mienen versteckt sind.

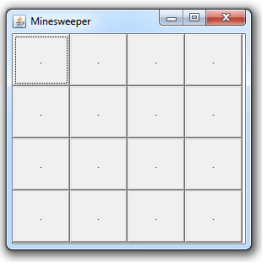


Während des Spiels wird beim Anklicken eines Feldes angezeigt, wie viele Mienen auf Nachbarfeldern platziert sind. Jedes Feld hat dabei 8 Nachbarfelder, Felder an den Rändern entsprechend weniger. Ziel des Spiels ist es, alle sicheren, also nicht mit Bomben belegten Felder zu finden. Sobald man auf ein Mienenfeld klickt, hat man verloren.

Implementieren Sie das Minesweeper-Spiel als Java-GUI-Anwendung. Ausgangspunkt ist eine im Quellcode vorgegebene (aber prinzipiell beliebige) M*N-Minesweeper-Lösungmatrix mit `int`-Werten, wobei eine Bombe durch den Wert `-1` repräsentiert wird, zum Beispiel:

```
final int[][] feld = { { -1, 1, 0, 0 },
                       { 2, 2, 1, 0 },
                       { 1, -1, 1, 0 },
                       { 1, 1, 1, 0 } };
```

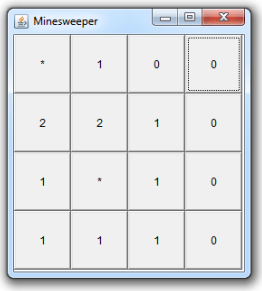
Die einzelnen Felder des Minesweeper-Spiels werden in der GUI durch Buttons repräsentiert. Anfangs enthalten alle Buttons das Label „`.`“.



Klickt der Benutzer auf ein Bombenfeld, ist das Spiel beendet und der Benutzer hat verloren. In diesem Fall soll das Label des Buttons durch das Label „`*`“ ersetzt werden. Außerdem sollen alle Buttons für Benutzerinteraktionen gesperrt werden (Button-Methode `setEnabled`).

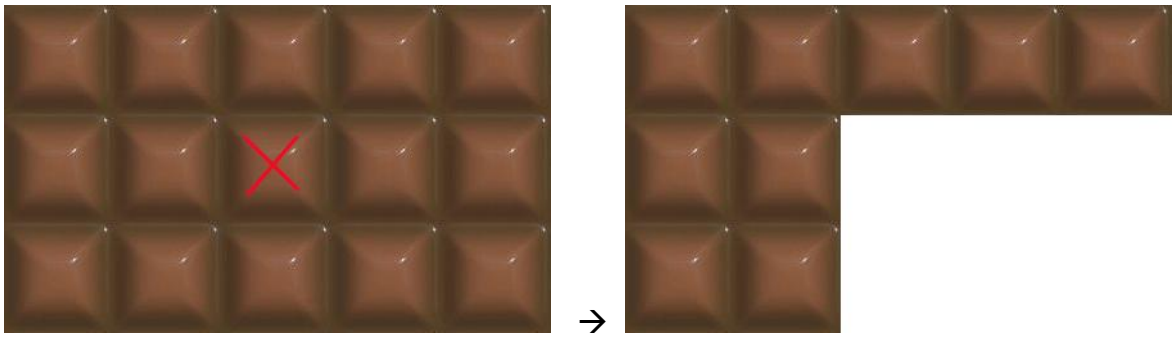
Klickt der Benutzer auf ein Nicht-Bombenfeld, soll das Label des Buttons durch den entsprechenden Wert des zugehörigen Matrix-Elementes ersetzt werden (also die Anzahl an Bomben auf Nachbarfeldern).

Beim Anklicken des letzten Nicht-Bombenfeldes, ist das Spiel beendet und der Benutzer hat gewonnen. In diesem Fall sollen unmittelbar alle Bombenfelder offengelegt werden, d.h. die Label der entsprechenden Buttons sollen durch das Label „`*`“ ersetzt werden.



Aufgabe 22:

Chomp ist ein Spiel für 2 Spieler. Gegeben eine Tafel Schokolade in Form eines rechteckiges Gebietes der Größe N * M, das aus einzelnen gleich großen Stückchen besteht. Die beiden Spieler essen abwechselnd ein (noch existierendes) Stückchen und damit gleichzeitig auch automatisch alle Stücken, die sich in Reihen darunter und Spalten rechts davon befinden. Wer das letzte Stückchen isst, bekommt Bauchschmerzen und verliert. Wer sich nicht an die Regeln hält, verliert ebenfalls unmittelbar.



Aufgabe:
Implementieren Sie das Chomp-Spiel, so dass es Menschen und Computer als Consolen-Spiel (ohne GUI) gegeneinander spielen können. Erweitern Sie dabei die folgende vorgegebene Klasse `Chomp`; genauer: Implementieren Sie die fehlende Methode `spielen` der Klasse `Chomp`.
Implementieren Sie eine `Spieler`-Klasse, die einen menschlichen Spieler repräsentiert. Eingaben von Spielzügen sollen über die Console erfolgen.
Implementieren Sie weiterhin eine `Spieler`-Klasse, die einen Computer-Spieler repräsentiert. Sie brauchen dabei keine Gewinnstrategie implementieren. Liefern Sie einfach zufällig ermittelte (aber korrekte!) Spielzüge.
Passen Sie die Zeilen 3 und 4 der `main`-Funktion der Klasse `Chomp` so an, dass ein Mensch gegen einen Computer-Spieler spielt.

Wichtig:
Sie dürfen keine Änderungen (sondern ausschließlich Erweiterungen) an den vorgegebenen Klassen vornehmen; mit Ausnahme der Implementierung der Methode `spielen` und der beiden Anweisungen in den Zeilen 3 und 4 der `main`-Funktion.

```
/**
 * Repraesentation von Spielzuegen
 */
class Spielzug {
	private int reihe; // Index der Reihe (ab 0)
	private int spalte; // Index der Spalte (ab 0)

	public Spielzug(int reihe, int spalte) {
		this.reihe = reihe;
		this.spalte = spalte;
	}

	public int getReihe() {
		return this.reihe;
	}

	public int getSpalte() {
		return this.spalte;
	}

	@Override
	public String toString() {
		return "Zug: r=" + this.reihe + "; s=" + this.spalte;
	}
}

/**
```

```java
 * Schnittstelle fuer Chomp-Spieler; auf der Basis dieses Interfaces sollen
 * Sie einen menschlichen Spieler und einen Computer-Spieler implementieren
 */
interface Spieler {
	/**
	 * liefert den naechsten Spielzug des Spielers
	 *
	 * @param schokolade
	 *            die aktuelle Schokolade (darf nicht veraendert werden!)
	 * @return der naechste Spielzug des Spielers
	 */
	public Spielzug naechsterSpielzug(boolean[][] schokolade);
}

/**
 * Chomp-Spiel
 */
public class Chomp {

	// die beiden Spieler; Index 0 = Spieler 1; Index 1 = Spieler 2
	Spieler[] spieler;

	// die Schokolade; true bedeutet: Stueckchen existiert noch
	boolean[][] schokolade; // die Schokolade

	// initialisiert ein Chomp-Spiel
	public Chomp(int anzahlReihen, int anzahlSpalten, Spieler spieler1,
			Spieler spieler2) {
		this.spieler = new Spieler[2];
		this.spieler[0] = spieler1;
		this.spieler[1] = spieler2;
		this.schokolade = new boolean[anzahlReihen][anzahlSpalten];
		for (int r = 0; r < anzahlReihen; r++) {
			for (int s = 0; s < anzahlSpalten; s++) {
				this.schokolade[r][s] = true;
			}
		}
	}

	// diese Methode muessen Sie implementieren
	public void spielen() {
		// - es beginnt Spieler 1
		// - Spielfeld ausgeben
		// - Endlosschleife:
		//   - Ausgabe des aktuellen Spielers
		//   - Spielzug beim aktuellen Spieler erfragen
		//       (ueber das interface Spieler!)
		//   - Spielzug ausgeben
		//   - Spielzug ueberpruefen (ggfl. Verlierer bekanntgeben
		//       und Ende)
		//   - Spielzug ausfuehren
		//   - Spielende ueberpruefen (ggfl. Verlierer bekanntgeben
		//       und Ende)
		//   - Spielfeld ausgeben
		//   - Spielerwechsel
	}

	public static void main(String[] args) {
		final int anzahlReihen = 3;
		final int anzahlSpalten = 5;
		/* 3 */ Spieler spieler1 = // Mensch oder Computer
		/* 4 */ Spieler spieler2 = // Mensch oder Computer
		Chomp chomp = new Chomp(anzahlReihen, anzahlSpalten,
```

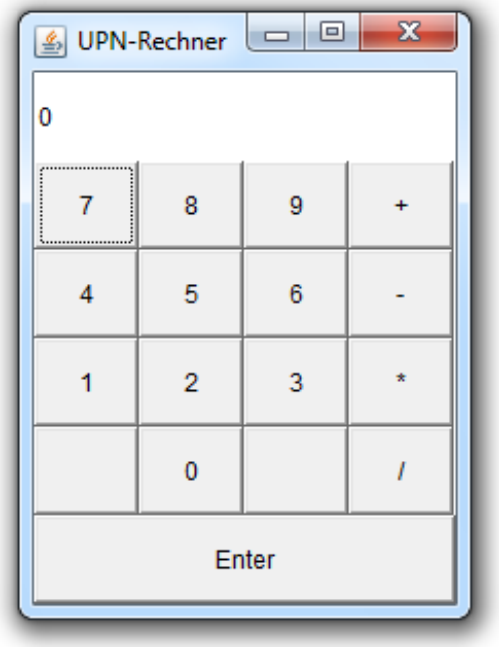
```
                                        spieler1, spieler2);
                chomp.spielen();
        }
}
```

Aufgabe 23:

In dieser Aufgabe sollen Sie einen UPN-Rechner mit interaktiver GUI implementieren. Bei der UPN (Umgekehrte Polnische Notation) werden eingelesene Zahlen jeweils oben auf den Stapel gelegt. Wird ein Operator eingegeben, werden die beiden oberen Elemente vom Stapel entfernt, darauf die Operation angewendet, das Ergebnis ausgegeben und das Ergebnis auf den Stapel gelegt.

Teilaufgabe A:

Bilden Sie mit den Ihnen bekannten Java-AWT-Klassen exakt folgende GUI nach:



In der obersten Zeile wird die aktuelle Zahl angezeigt. Darunter befindet sich eine Menge an Buttons, die die einzelnen Ziffern und Operationen repräsentieren.

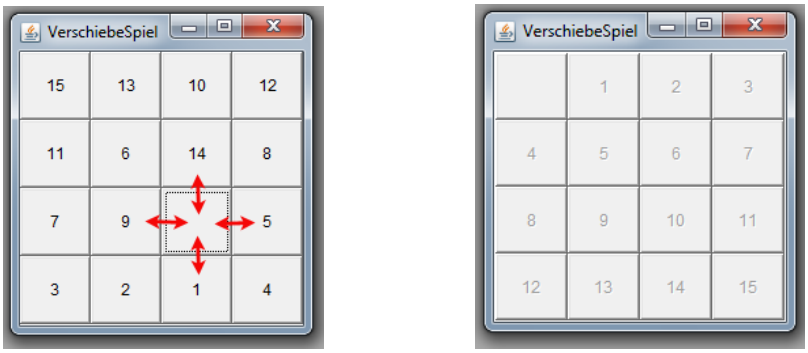
Teilaufgabe B:

Implementieren Sie das „Verhalten“ des UPN-Rechners:

- Eingabe einer Ziffer → Beginn einer neuen oder Fortsetzung einer existierenden Zahl und Anzeige der aktuellen Zahl im oberen Label.
- Eingabe von „ENTER“ → die aktuelle Zahl wird auf den Stack gelegt.
- Eingabe eines Operators → Ausführung der Operation mit den beiden obersten Stack-Einträgen und Anzeige der berechneten Zahl im oberen Label.

Aufgabe 24:

Sie alle kennen aus Ihrer Kindheit das so genannte Slider- oder Verschiebe-Spiel. Es ist ein Spiel für eine Person. Das Spielgerät ist eine Tafel mit N * M Feldern. In dieser Tafel sind auf N * M - 1 Feldern Plättchen mit den Ziffern 1 bis N * M - 1 platziert. Ein Feld ist leer. Die Plättchen sind verschiebbar. Gegeben eine bestimmte Ausgangsstellung der Plättchen ist es das Ziel, die Plättchen in der Reihenfolge 1 bis 15 anzuordnen (das Feld oben links soll leer sein).



Ihre Aufgabe ist es, eine interaktive GUI-Anwendung zum Spielen des Slider-Spiels zu implementieren.

Hinweise und Teilaufgaben:

Die Ausgangsstellung soll über eine int-Matrix im Code festgelegt werden. In der Matrix stehen die Zahlen von 0 bis N*M-1. Die 0 repräsentiert das fehlende Plättchen. Für die obige Abbildung links entspricht das:

```
int[][] matrix = { { 15, 13, 10, 12 },
                   { 11, 6, 14, 8 },
                   { 7, 9, 0, 5 },
                   { 3, 2, 1, 4 } };
```

Sie dürfen nicht davon ausgehen, dass die Matrix immer aus 16 Elementen besteht. Sie kann prinzipiell aus N Reihen und M Spalten bestehen, mit N > 1 und M > 1.

Im Fenster werden die N*M Felder durch N*M Buttons repräsentiert. Die Buttons werden mit den entsprechenden Zahlen der Matrix beschriftet. Der Button, der das fehlende Plättchen repräsentiert, bleibt leer.

Beim Klick auf einen dem leeren Button angrenzenden Nachbar-Button (in der obigen Abbildung die Buttons 9, 14, 5 und 1) sollen der angeklickte und der leere Button „getauscht“ werden. Realisieren Sie einen Tausch durch einen Wechsel der Button-Beschriftung (Methoden `getLabel` und `setLabel`). Klicks auf andere Buttons bleiben ohne Wirkung.

Ist der Endzustand erreicht (siehe Abbildung rechts) sollen alle Buttons disabled werden, d.h. nicht mehr anklickbar sein (Methode `setEnabled`).