Lösungsvorschläge zur Klausur zum Kurs 1618
Sommersemester 2007 am 28.7.2007

Aufgabe 1: Kurz gefasst

a) Am Ende der `main`-Methode ist folgendes Objektgeflecht entstanden:

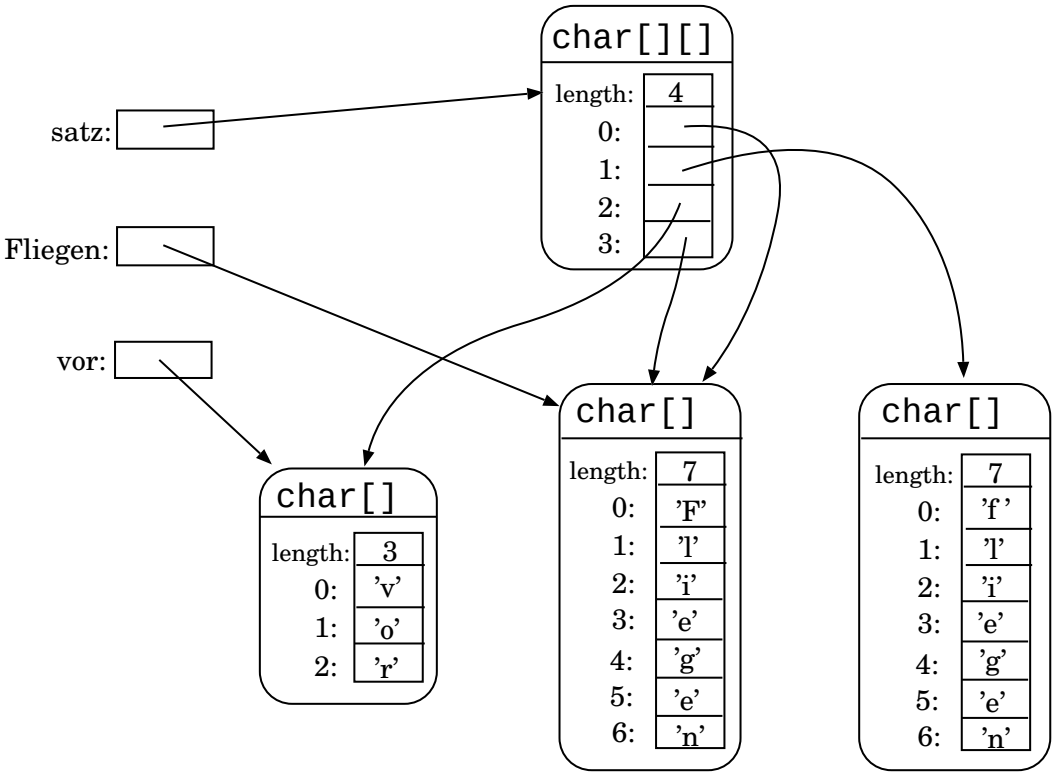

Abbildung 1: Das entstandene Objektgeflecht

b) Das folgende Beispiel demonstriert das Iterieren über Felder anhand der Variablen `satz` aus Aufgabenteil a) mit beiden Varianten der for-Anweisung.

```
public class ForSchleifen {
  public static void main(String[] args) {
    char[] vor;
    char[] Fliegen = {
        'F', 'l', 'i', 'e', 'g', 'e', 'n'};

    vor = new char[3];
    vor[0] = 'v';
    vor[1] = 'o';
    vor[2] = 'r';
```

```
    char[][] satz = {
        Fliegen, {
        'f', 'l', 'i', 'e', 'g', 'e', 'n'},
        vor,
        Fliegen};

    // Ausgabe des Feldinhalts mit Hilfe
    // der ersten Form der for-Anweisung
    System.out.println();
    System.out.println("** Ausgabe mit erster Form der for-Anweisung **");
    for (int i = 0; i < satz.length; i++) {
      System.out.print(satz[i]);
      System.out.print(" ");
    }
    System.out.println();

    // Ausgabe des Feldinhalts mit Hilfe
    // der zweiten Form der for-Anweisung (for-each)
    System.out.println();
    System.out.println("** Ausgabe mit zweiter Form der for-Anweisung **");
    for (char[] wort : satz) {
      System.out.print(wort);
      System.out.print(" ");
    }
    System.out.println();
  }
}
```

c) Der folgende Java-Code zeigt die Implementierung eines Typs für Farben mit Hilfe eines Java-Interfaces und des, ab der Java-Version 5.0 verfügbaren, Programmierkonstrukts für Aufzählungstypen.

```
interface Farben {
  byte rot   = 0;
  byte gruen = 1;
  byte blau  = 2;
  byte gelb  = 3;
}

enum Farben {rot, gruen, blau, gelb};
```

Eine Realisierung mittels der neuen Aufzählungstypen bietet gegenüber der Schnittstellenvariante u.a. mehr Typsicherheit. Während im Fall der Realisierung von Aufzählungstypen mit Schnittstellen Methoden, die Werte von Aufzählungstypen als Parameter entgegennehmen, nur vom Typ der Konstanten (in unserem Beispiel `byte`) für ihre Parametertypen ausgehen können, können solche Methoden bei Verwendung von Aufzählungstypen diesen Aufzählungstyp als ihren Parametertyp wählen. So kann der Compiler prüfen, ob als aktuelle Parameter nur die zugelassenen Werte übergeben werden. Im Fall von Schnittstellen kann dies nicht mehr entschieden werden. In unserem Beispiel könnten einer Methode

`void farbVergleich (byte farbe1, byte farbe2);`,
die zwei übergebene Farben auf Gleichheit prüft, Werte außerhalb des gültigen Bereichs von 0 bis 3 als Parameter übergeben werden. Greift die Methode auf den Aufzählungstyp für ihre Parametertypen zurück wie in

`void farbVergleich (Farben farbe1, Farben farbe2);`,
können ihr als aktuelle Parameter nur die Werte `Farben.rot, Farben.gruen, Farben.blau` und `Farben.gelb` übergeben werden. Analoges gilt für Variablen, die Werte von Aufzählungstypen aufnehmen sollen.

d) i hat nach Auswertung von
`int i = 10 + (i = x) / 7 + 3 * i - 13;`
den Wert 78.
Der gleichwertige Ausdruck
`int i = 10 + ((i = x) / 7) + (3 * i) - 13;`
besteht aus den Summanden
`10, ((i = x) / 7), (3 * i) und -13`,
die von links nach rechts aufsummiert werden.
Bei der Auswertung von `(i = x) / 7`
wird zunächst der Variablen i der Wert von x, nämlich 26 zugewiesen.
Anschließend wird eine ganzzahlige Division durch 7 durchgeführt, deren Ergebnis 3 ist.
Da i bereits mit dem Wert 26 belegt wurde, wird
`(3 * i)`
zu 78 ausgewertet.
Als Summe ergibt sich daher `10 + 3 + 78 - 13 = 78`.
Dieser Wert wird anschließend der Variablen i zugewiesen.

e) Die Objekterzeugung kann aus den unten angeführten Gründen nicht erfolgreich durchgeführt werden.

Der Konstruktor von `B` wird bei der Erzeugung des B-Objektes `b` aufgerufen. Dies führt zum Aufruf des Konstruktors von `A` (`super(ia);`), wobei `b` als impliziter Parameter übergeben wird. Die Ausführung des Konstruktors von `A` initialisiert das Attribut `ia` von `b` und ruft dann auf diesem Objekt die Methode `numberOfElements` auf. Wegen dynamischer Bindung führt dieser Aufruf zur Ausführung der Methode `numberOfElements` aus der Klasse `B`. Da zu diesem Zeitpunkt das Attribut `ia2` noch nicht initialisiert ist, terminiert der Aufruf `ia2.length` abrupt mit einer NullPointer-Ausnahme. Zusammengefasst besteht das Problem also darin, dass ein Methodenaufruf in einem Superklassen-Konstruktor durch dynamische Bindung zur Ausführung einer Subklassen-Methode führt, bevor der Subklassen-Konstruktor die Initialisierung der Attribute abschließen konnte.

Aufgabe 2: Polymorphie

a) *Polymorphie* bedeutet Vielgestaltigkeit. Im Zusammenhang mit Programmiersprachen spricht man von Polymorphie, wenn Programmkonstrukte oder Programmteile für Objekte (bzw. Werte) mehrerer Typen einsetzbar sind.

b) Im Kurs werden die vier folgenden Arten von Polymorphie beschrieben:

1. *Subtyp-Polymorphie*: Programmkonstrukte bzw. Programmteile, die für Objekte eines Typs T formuliert sind, gelten unverändert auch für Objekte eines Typs S, wenn S ein Subtyp von T ist. D.h., Objekte vom Typ S können an allen Programmstellen verwendet werden, an denen Objekte vom Typ T zulässig sind. Insbesondere können mit Hilfe der Subtyp-Polymorphie inhomogene Behälter realisiert werden, d.h. Behälter, die Objekte verschiedener Typen enthalten können.
2. *Parametrische Polymorphie*: Dies ist eine Art der Polymorphie, bei der man Programmkonstrukte – in objektorientiertem Kontext sind das dann meist Klassen und Methoden – mit Typen parametrisiert. So kann man z.B. Behältertypen mit ihren Elementtypen parametrisieren. Der einmal für diesen Behälter geschriebene Programmcode kann durch einfaches Instanziieren des Typparameters für jeden beliebigen Elementtyp verwendet werden. Parametrische Polymorphie bietet weniger Flexibilität als Subtyp-Polymorphie. Beispielsweise lassen sich mit Sprachen, die **nur** parametrische Polymorphie unterstützen, keine inhomogenen Behälter realisieren. Andererseits gestattet parametrische Polymorphie bei vielen Anwendungen eine leistungsfähigere Typanalyse zur Übersetzungszeit.
3. *Beschränkt parametrische Polymorphie*: Beschränkt parametrische Polymorphie verbindet Subtyp-Polymorphie mit parametrischer Polymorphie. Sie ermöglicht es, Typparameter von parametrischen Typen durch Angabe von Supertypen zu beschränken. Für einen durch einen Typ T beschränkten Typparameter dürfen nur solche Typen S eingesetzt werden, die Subtypen von T sind. Die Verwendung solcher Typen bringt dem Anwender zunächst keinen Vorteil vor parametrischen Typen wie oben beschrieben. Vorteile bringt dieser Ansatz vor allem bei der Implementierung solcher beschränkt parametrischer Typen. In der Implementierung kann nämlich das Wissen ausgenutzt werden, dass Objekte eines Typs S, der für einen Typparameter ET eingesetzt wird, der selbst nach oben durch einen Supertyp T beschränkt ist, mindestens über alle Eigenschaften von T verfügen. Diese Eigenschaften können dann in der Implementierung ausgenutzt werden.
4. *Ad-hoc-Polymorphie*: Das Überladen von Operatoren bzw. Methodennamen bezeichnet man oft als *Ad-hoc-Polymorphie*. Überladung erlaubt es, Operationen, die man konzeptionell identifiziert, für ggf. verschiedene Typen unter dem gleichen Namen anzubieten – z.B. eine plus-Operation für komplexe oder reelle Zahlen. Im Gegensatz zu den anderen Formen von Polymorphie führt Überladung aber nur zu einer konzeptionellen Identifikation und nicht dazu, dass derselbe Programmcode für die Bearbeitung unterschiedlicher Typen herangezogen wird. Überladung wird zur Übersetzungszeit aufgelöst, also nicht dynamisch gebunden. Technisch gesehen bietet sie dementsprechend nur größere Flexibilität bei der Namensgebung. Deshalb sagt man auch, dass Überladung keine Form *echter* Polymorphie ist.

c) Im Folgenden wird je ein Beispiel für jede der vier in b) genannten Arten von Polymorphie angegeben.

1. *Subtyp-Polymorphie*:
   Der Programmcode der Klasse `Stack` ist für Objekte vom Typ `Object` formuliert, kann aber auch Objekte von Subtypen von `Object` verarbeiten wie z.B. Objekte vom Typ `Integer`, `String`, `Float` oder irgendeines selbst deklarierten Typs. Zur Demonstration dieser Art der Polymorphie kann die Klasse `Stack` unverändert bleiben. In der Klasse `TestStack` wird gezeigt, dass eine Instanz von `Stack` beliebige Objekte von Typen, die Subtypen von `Object` sind, enthalten und verarbeiten kann.

```
class TestStack {
  public static void main(String[] args) {
    Stack stack = new Stack();

    stack.push(new Integer(1));
    stack.push(new String("zweites Element"));
    stack.push(new Float(3.3));

    if (!stack.isEmpty())
      System.out.println(stack.top());
    else
      System.out.println("Stack ist leer");

    while (!stack.isEmpty())
      System.out.println(stack.pop());
  }
}
```

2. *Parametrische Polymorphie*:
   Die folgende Klasse implementiert einen generischen Stack, der Elemente eines beliebigen Typs `ET` verarbeiten kann. Ein `Integer`-Stack (`GenerischerStack<Integer>`) kann also nur `Integer`-Objekte verwalten, ein `String`-Stack nur `String`-Objekte etc. Zur Unterscheidung von der in der Aufgabenstellung angegebenen Klasse `Stack` nennen wir die modifizierte Klasse `GenerischerStack`. Der Einsatz dieses generischen Typs für verschiedene Elementtypen wird in der Klasse `TestGenerischerStack` demonstriert.

```
import java.util.*;

class GenerischerStack<ET> {
  ArrayList<ET> stack = new ArrayList<ET>();

  public boolean isEmpty() {
    return stack.isEmpty();
  }

  public void push (ET elem) {
    stack.add(elem);
  }
```

```
  public ET pop () {
    if (!stack.isEmpty())
      return stack.remove(stack.size()-1);
    else
      return null;
  }

  public ET top () {
    if (!stack.isEmpty())
      return stack.get(stack.size()-1);
    else
      return null;
  }
}

class TestGenerischerStack {
  public static void main(String[] args) {

    /*****************************************/
    /*       Stack für Integer-Objekte       */
    /*****************************************/
    GenerischerStack<Integer> gst_integer = new GenerischerStack<Integer>();

    for (int i = 1; i < 6; i++)
      gst_integer.push(i);

    System.out.println(gst_integer.top());

    while (!gst_integer.isEmpty())
      System.out.println(gst_integer.pop());

    System.out.println("Stack ist leer");

    /*****************************************/
    /*       Stack für String-Objekte        */
    /*****************************************/
    GenerischerStack<String> gst_string = new GenerischerStack<String>();
    gst_string.push("erstes Element");
    gst_string.push("zweites Element");
    /* ... */
  }
}
```

3. *Beschränkt parametrische Polymorphie*:

   Die folgende Klasse implementiert einen generischen Stack, dessen Elementtyp zusätzlich nach oben durch den Typ `Druckbar` beschränkt ist. D.h. alle Elemente, die im Stack gespeichert werden können, implementieren die Methode `drucken`. Diese Variante des Stacks enthält eine zusätzliche Methode `printAll`, die sich diese Eigenschaft zunutze macht und auf allen Elementen des Stacks die Methode `drucken` aufruft.

```
interface Druckbar {
  void drucken();
}

import java.util.*;
class BeschraenktGenerischerStack<ET extends Druckbar>  {
  ArrayList<ET> stack = new ArrayList<ET>();

  public boolean isEmpty() {
    return stack.isEmpty();
  }

  public void push (ET elem) {
    stack.add(elem);
  }

  public ET pop () {
    if (!stack.isEmpty())
      return stack.remove(stack.size()-1);
    else
      return null;
  }

  public ET top () {
    if (!stack.isEmpty())
      return stack.get(stack.size()-1);
    else
      return null;
  }

  public void printAll() {
    for (ET elem : stack) {
      elem.drucken();
    }
  }
}

// Klassen, die gültige Elementtypen darstellen
class Person implements Druckbar {
  private String name;

  public Person (String name) {
    this.name = name;
  }
  public void drucken() {
    System.out.println();
    System.out.println("Personendaten: Name: " + name);
    /* ... */
  }
}
```

```
class Trainer extends Person {
  private String verein;

  public Trainer (String name, String verein) {
    super(name);
    this.verein = verein;
  }

  public void drucken() {
    super.drucken();
    System.out.println("               Verein: " + verein);
  }
}

// Testklasse
class TestBeschraenktGenerischerStack {
  public static void main(String[] args) {

    /*****************************************/
    /*       Stack für Person-Objekte        */
    /*****************************************/
    BeschraenktGenerischerStack<Person> gst =
        new BeschraenktGenerischerStack<Person> ();

    gst.push(new Person("Otto Rehhagel"));
    gst.push(new Trainer("Ottmar Hitzfeld", "FC Bayern München"));

    while (!gst.isEmpty()) {
      gst.pop().drucken();
    }

    /*****************************************/
    /*       Stack für Trainer-Objekte       */
    /*****************************************/

    BeschraenktGenerischerStack<Trainer> gst_trainer =
        new BeschraenktGenerischerStack<Trainer> ();

    gst_trainer.push(new Trainer("Ottmar Hitzfeld", "FC Bayern München"));
    gst_trainer.push(new Trainer("Mirko Slomka", "FC Schalke 04"));
    gst_trainer.push(new Trainer("Thomas Schaaf", "Werder Bremen"));

    gst_trainer.printAll();

  }
}
```

Der Stack kann hier nur Objekte vom Typ `ET` und dessen Subtypen speichern, aber nicht gemischt beliebige Objekte vom Typ `Druckbar`.

4. *Ad-hoc-Polymorphie*:

   Man betrachte die folgende Variante der Klasse `Stack`, wobei der Methodenname `push` überladen ist. Diese Variante garantiert, dass nur `Integer, Float` und `Double`-Objekte auf dem Stack abgelegt werden können. Die drei `push`-Operationen sind konzeptionell identisch; sie legen jeweils ein Objekt auf dem Stack ab. Sie unterscheiden sich lediglich im Typ für ihren Parameter.

```
class StackMitAdHocPolymorphie {
  ArrayList stack = new ArrayList();

  public boolean isEmpty() {
    return stack.isEmpty();
  }

  public void push (Integer elem) {
    stack.add(elem);
  }

  public void push (Float elem) {
    stack.add(elem);
  }

  public void push (Double elem) {
    stack.add(elem);
  }

  public Object pop () {
    if (!stack.isEmpty())
      return stack.remove(stack.size()-1);
    else
      return null;
  }

  public Object top () {
    if (!stack.isEmpty())
      return stack.get(stack.size()-1);
    else
      return null;
  }
}
```

Aufgabe 3: Verhaltenskonformität

a) Die Grundregel für Subtyping besagt, dass ein Objekt von einem Subtyp an allen Programmstellen vorkommen kann, an denen Supertyp-Objekte zulässig sind. Dazu muss der Subtyp die Eigenschaften des Supertyps besitzen. Diese Eigenschaften betreffen sowohl syntaktische Bedingungen für z.B. die Deklaration von Attributen und Methodensignaturen als auch das *Verhalten* des Subtyps, das durch seine Methoden*implementierungen* bestimmt wird. Ein Subtyp verhält sich *konform* zu seinem Supertyp, wenn seine Methodenimplementierungen dasselbe Verhalten aufweisen, wie durch seinen Supertyp vorgegeben.

   Beispielsweise könnte man die Methode `equals`, die zu jeder Klasse gehört (sie wird von `Object` geerbt) und die zwei Objekte auf Gleichheit prüft, in beliebigen Subtypen von `Object` z.B. so implementieren, dass sie die syntaktischen Bedingungen erfüllt, aber immer `true` liefert. Damit würde sich diese Methode und insgesamt der Typ, zu dem sie gehört, nicht erwartungsgemäß verhalten. Für ein weiteres, nicht konformes Verhalten siehe b).

b) Im Folgenden setzen wir die Implementierung eines Stacks mit dem üblichen Verhalten voraus wie im Beispiel zu Aufgabe 2 c) 2. (`GenerischerStack`) gezeigt: seine `push`-Methode legt das als Parameter übergebene Objekt oben auf dem Stack ab, seine `pop`-Methode nimmt das oberste Element vom Stack und gibt es zurück. D.h. das Element, das zuletzt auf dem Stack abgelegt wurden, wird mittels `pop` als erstes wieder vom Stack herunter genommen. Die `top`-Methode der Klasse Stack liefert das oberste Stackelement zurück, lässt es aber, im Gegensatz zu `pop`, auf dem Stack.

   Bilden wir jetzt einen Subtyp von `GenerischerStack`, der die Implementierung der Methode `pop` so abändert, dass sie dasjenige Element zurückgibt und aus dem Stack löscht, das ganz unten auf dem Stack liegt, realisiert er eine FIFO-Queue. Er verhält er sich also *nicht konform* zu seinem Supertyp `GenerischerStack`.

```
import java.util.*;

class GenerischerStackSub<ET> extends GenerischerStack<ET> {

  public ET pop () {
    if (!stack.isEmpty())
      return stack.remove(0);
    else
      return null;
  }
}
```

Aufgabe 4: AWT und Ereignissteuerung

a)
1. Die Ereignissorte *MouseEvent* gruppiert Ereignisse, die durch die Mausbewegungen und Maustasten ausgelöst werden wie z.B. mousePressed, mouseReleased, mouseClicked, mouseEntered, mouseExited, mouseDragged, mouseMoved.
2. Die Ereignissorte *WindowEvent* gruppiert Ereignisse, die an Fenstern auftreten wie z.B. wenn das Fenster durch Klicken auf ein entsprechendes Symbol geschlossen, ikonifiziert oder wieder vergrößert werden soll, wenn das Fenster durch Klicken in die Titelzeile zum aktiven Fenster wird etc. Zu den WindowEvents gehören die Folgenden: windowOpened, windowClosing, windowClosed, windowIconified, windowDeiconified, windowActivated, windowDeactivated, windowGainedFocus, WindowLostFocus, WindowStateChanged.
3. Die Ereignissorte *ActionEvent* enthält ein einziges Ereignis, nämlich actionPerformed. Im Gegensatz zu den beiden zuvor genannten Ereignissorten, die elementare Ereignisse gruppieren, handelt es sich hierbei um eine Ereignissorte mit einem semantische Ereignis. Tritt an einer Schaltfläche oder an einer Komponenten vom Typ `List`, `MenuItem` bzw. `TextField` ein mousePressed-Ereignis für die linke Maustaste gefolgt von einem entsprechenden mouseReleased-Ereignis auf, ohne dass die Mausposition zwischenzeitlich die Komponente verlassen hat, wird zusätzlich an dieser Komponente das daraus resultierende semantische Ereignis actionPerformed erzeugt.

b) Eine Komponente muss bekannt geben, Ereignisse welcher Ereignissorten an ihr auftreten können. Dies tut sie durch Bereitstellen von Registriermethoden, die einem bestimmten Muster folgen. Für eine Ereignissorte *ESEvent* muss die Registriermethode wie folgt aussehen: `public void add`*ES*`Listener (`*ES*`Listener l)`.

Beispiel:

An Objekten der Klasse `Button` können ActionEvents auftreten. Das gibt die Klasse `Button` bekannt, indem sie die Registriermethode

```
public void addActionListener (ActionListener l)
```

implementiert.

c) Sollen Instanzen einer Klasse `B` als Beobachter von Ereignissen der Sorte *ESEvent* fungieren, muss `B` den Schnittstellentyp mit Namen *ES*`Listener` implementieren. In der Schnittstelle *ES*`Listener` gibt es zu jedem Ereignis e der Sorte *ESEvent* genau eine Methode mit Namen e. Die Implementierung dieser Methode legt fest, was passieren soll, wenn der Beobachter von dem Ereignis benachrichtigt wird. Typischerweise stößt die Methode e eine entsprechende Operation der Anwendung an.

Beispiel:

Objekte, die ActionEvents beobachten sollen, müssen die Schnittstelle `ActionListener` implementieren. Dabei besteht diese Schnittstelle nur aus der Methode

```
void actionPerformed (ActionEvent e);
```

über die das einzige zur Sorte ActionEvent gehörende Ereignis kommuniziert wird. Die Beobachterobjekte in unserem Beispiel sollen bei Auftreten des Ereignisses actionPerformed auf der Systemausgabe lediglich den Text “Schaltflaeche betaetigt” ausgeben.

Beobachter-Objekte für unser Beispiel können dann durch folgende Klasse beschrieben werden:

```
class B implements ActionListener {
  public  void actionPerformed( ActionEvent e ) {
    System.out.println("Schaltflaeche betaetigt");
  }
}
```

d) Tritt an einer Komponente k ein Ereignis e der Sorte $ESEvent$ auf, werden die Beobachter-Objekte, die bei k für Ereignisse der Sorte $ESEvent$ registriert sind, automatisch durch Aufruf der Methode mit Namen e benachrichtigt. Dabei wird der Methode als Parameter ein Objekt übergeben, das Informationen zu dem aufgetretenen Ereignis enthält.

Beispiel:

Beim Auftreten des Ereignisses `actionPerformed` an einer Button-Komponente *sf* werden alle an dieser Komponente für ActionEvents registrierten Beobachter *b* durch Aufruf von `b.actionPerformed (e)` benachrichtigt. Dabei ist e ein ActionEvent-Objekt, das zusätzliche Informationen zu dem aufgetretenen Ereignis enthält. Für ActionEvents, die an Button-Objekten aufgetreten sind, enthält e z.B. Informationen über die Beschriftung des Buttons, einen Zeitstempel, der angibt, wann das Ereignis aufgetreten ist etc.

e) Die folgende Klasse `CountFrame` enthält alle geforderten Erweiterungen.

```
import java.awt.*;
import java.awt.event.*;

class CountFrame extends Frame {
  private int counter = 0;
  private Panel p = new Panel(new FlowLayout());
  private TextField tCounter = new TextField(5);
  private Button bInc = new Button(">");
  private Button bDec = new Button("<");

  public CountFrame() {
    tCounter.setText("" + counter);

    // Komponenten in Fenster einfügen
    p.add(bDec);
    p.add(tCounter);
    p.add(bInc);
    this.add(p);

    // Beobachter registrieren
    bDec.addActionListener (new ActionListener () {
      public void actionPerformed (ActionEvent e) {
        counter--;
        tCounter.setText("" + counter);
      }
```

```
    });

    bInc.addActionListener (new ActionListener () {
      public void actionPerformed (ActionEvent e) {
        counter++;
        tCounter.setText("" + counter);
      }
    });

    addWindowListener (new WindowAdapter() {
      public void windowClosing (WindowEvent e) {
        System.exit(0);
      }
    });

    this.setSize(100, 100);
    this.setLocation(100, 100);
    this.setVisible(true);
  }
}
```

Aufgabe 5: Kooperierende Threads

Der folgende Programmcode zeigt eine korrekte Implementierung des Verpackungsprozesses:

```
public class Verpackungsprozess {
  public static void main(String[] args) {
    Signale signale = new Signale();
    new Kartonzufuhrprozess(signale).start();
    new Produktablageprozess(signale).start();
    new Kartonabtransportprozess(signale).start();
  }
}

class Signale {
  private boolean ablageposFrei = true;
  private boolean produktablageErlaubt = false;
  private boolean produktAbgelegt = false;

  // Darauf warten, dass Signal auf 'true' gesetzt wird
  public synchronized void WarteAblageposFrei() throws InterruptedException {
    while (!ablageposFrei) wait();
  }
  public synchronized void WarteProduktablageErlaubt() throws InterruptedException {
    while (!produktablageErlaubt) wait();
  }
  public synchronized void WarteProduktAbgelegt() throws InterruptedException {
    while (!produktAbgelegt) wait();
  }

  // Signalzustand setzen
  public synchronized void setAblageposFrei(boolean value) {
    ablageposFrei = value;
    System.out.println("ablageposFrei: " + ablageposFrei);
    if (ablageposFrei) notifyAll();
  }
  public synchronized void setProduktablageErlaubt(boolean value) {
    produktablageErlaubt = value;
    System.out.println("produktablageErlaubt: " + produktablageErlaubt);
    if (produktablageErlaubt) notifyAll();
  }
  public synchronized void setProduktAbgelegt(boolean value) {
    produktAbgelegt = value;
    System.out.println("produktAbgelegt: " + produktAbgelegt);
    if (produktAbgelegt) notifyAll();
  }
}
```

```
class Kartonzufuhrprozess extends Thread {
  private Signale signale;

  public Kartonzufuhrprozess(Signale signale) {
    this.signale = signale;
  }

  private void fahreKartonVor() throws InterruptedException {
    sleep(1000); // Verfahrzeit bis zur Ablageposition
  }

  private void fahreKartonZurAblageposition() throws InterruptedException {

    signale.WarteAblageposFrei();
    System.out.println("-1- Karton wird vorgefahren");
    fahreKartonVor();
    signale.setAblageposFrei(false);
    signale.setProduktablageErlaubt(true);
  }

  public void run() {
    while (true) {
      try {
        fahreKartonZurAblageposition();
      }
      catch (InterruptedException e) {
        System.out.println("Unzulaessige Unterbrechung");
        System.exit(0);
      }
    }
  }
}

class Produktablageprozess extends Thread {
  private Signale signale;

  public Produktablageprozess(Signale signale) {
    this.signale = signale;
  }

  private void legeProduktAb() throws InterruptedException {
    sleep(500); // Zeit, um das Produkt in den Karton zu legen
  }

  private void legeProduktAbFallsErlaubt() throws InterruptedException {

    signale.WarteProduktablageErlaubt();
    System.out.println("-2- Produkt wird abgelegt");
    legeProduktAb();
    signale.setProduktablageErlaubt(false);
    signale.setProduktAbgelegt(true);
  }
```

```
  public void run() {
    while (true) {
      try {
        legeProduktAbFallsErlaubt();
      }
      catch (InterruptedException e) {
        System.out.println("Unzulaessige Unterbrechung");
        System.exit(0);
      }
    }
  }
}

class Kartonabtransportprozess extends Thread {
  private Signale signale;

  public Kartonabtransportprozess(Signale signale) {
    this.signale = signale;
  }

  private void fahreKartonWeg() throws InterruptedException {
    sleep(250); // Zeit, in der der Karton von der
                // Ablageposition weggefahren wird.
  }

  private void fahreKartonVonAblageposWeg() throws InterruptedException {

    signale.WarteProduktAbgelegt();
    System.out.println("-3- Karton wird weggefahren");
    fahreKartonWeg();
    signale.setProduktAbgelegt(false);
    signale.setAblageposFrei(true);
  }

  public void run() {
    while (true) {
      try {
        fahreKartonVonAblageposWeg();
      }
      catch (InterruptedException e) {
        System.out.println("Unzulaessige Unterbrechung");
        System.exit(0);
      }
    }
  }
}
```