Programmiermethodik
2. Klausur **Lösung**

2. 10. 2013

Name	
Matrikelnummer	

Aufgabe	**mögliche Punkte**	**erreichte Punkte**
1	20	
2	18	
3	42	
4	20	
5	20	
Gesamt	**120**	

Aufgabe 1) Objekt-Orientierung und Vererbung (20 Punkte)

1. Betrachten Sie das folgende Programm. Entscheiden Sie für die Variablen/Methodendeklarationen A1 bis A5, ob diese hier zulässig sind, und begründen Sie ihre Antwort. Entscheiden Sie anschließend für die Codezeilen B1 bis B4, ob diese korrekt sind (ohne Fehler compilieren), und sagen Sie die Ausgabe für korrekte Zeilen vorher. **12 Punkte**

Lösung A1: geht nicht, die Methode gibt es bereits und sie ist final 1 Pkt

A2: geht, andere Signatur, es wird nicht überschrieben 1 Pkt

A3: geht, draw ist zwar final, aber auch private! -> kein Konflikt 2 Pkt

A4: geht, ohne final, normales überschreiben 1 Pkt

A5: geht, final bei Attributen heißt, dass sie nur einmal initialisiert werden können, kein Konflikt 2 Pkt

B1: geht nicht (keine Methode win(String) in Game!) 1 Pkt

B2: geht nicht (keine Methode draw() in Game! -> private) 2 Pkt

B3: Ausgabe: “loser.” (dynamische Bindung) 1 Pkt

B4: Ausgabe: “$1,000,000” 1 Pkt

```
1  class Game {
2
3      public final void win() {
4          System.out.println("YAY!");
5      }
6
7      private final void draw() {
8          System.out.println("Next Time!");
9      }
10
11     public void lose() {
12         System.out.println("oooooh.");
13     }
14
15     public final String PRIZE = "$1,000,000";
16
17 }
18
19
20
21
22 public class AnotherGame extends Game {
23
24     // A1
25     public void win() {
26         System.out.println("WOOOOHOOOO!");
27     }
28
29     // A2
30     public void win(String text) {
31         System.out.println(text);
32     }
33
34     // A3
35     public void draw() {
36         System.out.println("Another Chance!");
37     }
38
39     // A4
40     public void lose() {
41         System.out.println("loser.");
42     }
43
44     // A5
45     public final String PRIZE = "$1";
46
47     public static void main(String[] args) {
48         Game game = new AnotherGame();
49         game.win("BINGO!"); // B1
50         game.draw(); // B2
51         game.lose(); // B3
52         System.out.println("You won: " + game.PRIZE); // B4
53     }
54 }
```

2. Woraus besteht die Signatur einer Methode? Ist der Rückgabetyp einer Methode Teil der Signatur? **3 Punkte**

   Lösung Signatur = Name + Typen (auch Reihenfolge!) der Argumentliste je 1 Punkt = 2 Pkt

   Die Signatur sollte prinzipiell eindeutig sein, um Methoden unterscheiden zu können. Der Rückgabewert ist daher NICHT Teil der Signatur. 1 Pkt

3. Erklären Sie den Begriff Autoboxing/Autounboxing? Wie würde folgender Ausschnitt aus einem Java Programm ohne Auto(un)boxing aussehen? **5 Punkte**

```
1 Integer i = 2;
2 int j = 2;
3 int k = i * j;
4 i = j - k;
```

   Lösung Primitive Datentypen werden automatisch in Objekte umgewandelt, falls nötig, und umgekehrt. 2 Pkt

```
Integer i = Integer.valueOf(2);
```
1 Pkt

```
int j = 2;
```
0 Pkt

```
int k = j * i.intValue();
```
1 Pkt

```
i = Integer.valueOf(j - k);
```
1 Pkt

Aufgabe 2) Schnittstellen und Abstrakte Klassen (18 Punkte)

1. Erklären Sie die Unterschiede zwischen einer abstrakten Klasse und einem Interface in Java. Skizzieren Sie jeweils einen Anwendungsfall, in dem es jeweils besser ist ein Interface bzw. eine abstrakte Klasse zu verwenden. **5 Punkte**
2. Wie unterscheidet sich eine abstrakte Klasse von einer herkömmlichen Klasse? **5 Punkte**
3. Beantworten Sie folgende Fragen kurz in Stichworten.
   (a) Kann eine abstrakte Klasse mehrere Interfaces implementieren? Wenn nicht, warum?
   (b) Können abstrakte Klassen von mehreren anderen abstrakten Klassen erben? Wenn nicht, warum?
   (c) Kann eine abstrakte Klasse von einer nicht abstrakten Klasse erben? Wenn nicht, warum?
   (d) Können in abstrakten Klassen statische Felder deklariert und initialisiert werden? Wenn nicht, warum?

   8 Punkte

Lösung

1. Abstrakte Klassen können Attribute haben und Methoden implementieren, während Interfaces nur Konstante und abstrakte Methoden definieren können. Interfaces werden dort benötigt, wo sonst Mehrfachvererbung gebraucht wird, während man abstrakte Klassen besser dort einsetzt, wo viele Subklassen die gleiche Funktionalität implementieren und somit Code Reuse sinnvoll ist.
2. Schlüsselwort `abstraft`, abstrakte Methoden können abstrakte Methoden beinhalten, von Interfaces geerbte Methoden müssen nicht implementiert werden, man kann nicht direkt von einer abstrakten Klasse eine Instanz erzeugen (nur von konkreten Unterklassen der abstrakten Klasse)
3. (a) Ja.
   (b) Nein, da Mehrfachvererbung in Java nicht möglich ist.
   (c) Ja.
   (d) Ja.

Aufgabe 3) Ausnahmen, Vererbung, Delegation (42 Punkte) Betrachten Sie den folgenden Code mit einem stark vereinfachten `Collection`-Interface, einer Array-basierten Implementierung dieses Interfaces und einem kleinen Test-Programm.

Bitte beachten Sie in der folgenden Aufgabenstellung, dass die die Aufgaben 1(a,b), 1(c), 2, 3(a) und 3(b) jeweils unabhängig voneinander gelöst werden können!

1. Programm-Vervollständigung und -Analyse

   (a) Ergänzen Sie den Code für den Konstruktor in Zeile 48. **3 Punkte**

   (b) Geben Sie an, welche Ausgaben beim Aufruf der `test`-Methode erfolgen. **3 Punkte**

   (c) Man könnte die Zeilen 33 – 35 durch die einzelne Anweisung `return elements[size];` ersetzen. Erläutern Sie kurz, welche Probleme diese Alternative mit sich bringt. **4 Punkte**

2. In der Klasse `BoundedArrayCollection` soll die Fehlerbehandlung nun über Ausnahmen realisiert werden.

   (a) Definieren Sie kurz eine eigene Ausnahme `CollectionFullException`. **2 Punkte**

   (b) Ändern Sie die Methoden `add` und `extract` so ab, dass die Ausnahmen `NoSuchElementException` und `CollectionFullException` verwendet werden. **3 Punkte**

   (c) Müssen Sie etwas an der Methode `test` ändern, um das Programm kompilieren zu können? Begründen Sie kurz. Was passiert nun beim Aufruf von `test`? **4 Punkte**

3. Entwerfen Sie zwei neue Klassen, die jeweils das `Collection`-**Interface** implementieren und einen **öffentlichen Konstruktor** implementieren. Beide Klassen sollten dabei die Elemente intern in Arrays speichern, sich aber – im Gegensatz zur `BoundedArrayCollection` – bei Bedarf automatisch vergrößern. Sie dürfen davon ausgehen, dass die Klasse `BoundedArrayCollection` importiert worden ist, diese darf aber nicht verändert werden.

   (a) Eine Klasse (`ArrayInheritanceCollection`) sollte dazu das Prinzip der Vererbung nutzen. Hierbei könnte die Methode `static <T> T[] copyOf(T[] original, int newLength)` in der Klasse `Arrays` von Nutzen sein: Zum Beispiel liefert ein Aufruf mit den Parametern `["foo", "bar"]` und `4` das neue Array `["foo", "bar", null, null]`. **10 Punkte**

   (b) Die andere Klasse (`ArrayDelegateCollection`) soll das Interface erfüllen und dabei das Prinzip der Delegation nutzen. **13 Punkte**

```
1  public interface Collection<T> {
2      /** adds an element into the collection */
3      void add(T elem);
4      /** delivers and removes an arbitrary element from the collection */
5      T extract();
6      /** checks whether the collection is empty */
7      boolean isEmpty();
8  }
9
10 public class BoundedArrayCollection<T> implements Collection<T> {
11     protected T[] elements; // the elements are stored
12     protected int size;     // in positions [0,...,size-1]
13
14     public boolean isEmpty() {
15         return size == 0;
16     }
17
18     public boolean isFull() {
19         return size == elements.length;
20     }
21
22     public void add(T elem) {
23         if (! isFull())  {
24             elements[size] = elem;
25             size++;
26         } else
27             System.err.println("collection full");
28     }
29
30     public T extract() {
31         if (! isEmpty()) {
32             size--;
33             T elem = elements[size];
34             elements[size] = null;
35             return elem;
36         } else {
37             System.err.println("no such element");
38             return null;
39         }
40     }
41
42     public int arraySize() {
43         return elements.length;
44     }
45
46     /** creates an empty collection, which can store at most n elements */
47     public BoundedArrayCollection(int n) {
48         // TODO
49     }
50 }
51
52 public class Test {
53     public static void test() {
54         Collection<Integer> coll = new BoundedArrayCollection<>(1);
55         coll.add(3);
56         coll.add(5);
57         System.out.println(coll.extract());
58         System.out.println(coll.extract());
59     }
60 }
```

Lösung

1. (a)

```
    public BoundedArrayCollection(int n) {
        size = 0;
        elements = (T []) (new Object[n]);
    }
```

   (b) `collection full`, `3`, `no such element`, `null`

   (c) Wenn man die Anweisung `elements[size] = null;` nicht durchführt, ist der dort gespeicherte Eintrag logisch nicht mehr in der Sammlung, bleibt aber trotzdem referenzierbar. Dementsprechend kann der Garbage Collector den Speicher für den Eintrag nicht freigeben.

2. (a)

```
public class CollectionFullException extends RuntimeException {}
```

   (b)

```
    public void add(T elem) {
        if (! isFull()) {
            elements[size] = elem;
            size++;
        } else
            throw new CollectionFullException();
    }

    public T extract() {
        if (! isEmpty()) {
            size--;
            T elem = elements[size];
            elements[size] = null;
            return elem;
        } else
            throw new NoSuchElementException();
    }
```

   (c) Die Methode `test` kann unverändert bleiben, da die verwendeten Ausnahmen `RuntimeException`s sind. Der Aufruf von `test` führt nun direkt mittels einer `CollectionFullException` zu einem Programmabbruch in Zeile 56.

3. (a)

```
public class ArrayInheritanceCollection<T>
  extends BoundedArrayCollection<T> {  // 2 points

    public ArrayInheritanceCollection() { // 3 points
        super(10);
    }

    public void add(T elem) {
        if (super.isFull()) { // 3 points
            elements = Arrays.copyOf(elements, 2 * elements.length);
        }
        super.add(elem);
    }
```

```
    public boolean isFull() { // 2 points
        return false;
    }
}
```

(b)

```
public class ArrayDelegateCollection<T>
  implements Collection<T> {  // 2 points

    private BoundedArrayCollection<T> collection = new
        BoundedArrayCollection<>(10); // 3 points

    public void add(T elem) {  // 6 points
        if (collection.isFull()) {
            BoundedArrayCollection<T> newCollection =
                new BoundedArrayCollection<>(collection.arraySize() * 2);
            while (!collection.isEmpty()) {
                newCollection.add(collection.extract());
            }
            collection = newCollection;
        }
        collection.add(elem);
    }

    public T extract() {  // 1 point
        return collection.extract();
    }

    public boolean isEmpty() { // 1 point
        return collection.isEmpty();
    }
}
```

Aufgabe 4) Generische Programmierung (20 Punkte)

1. Erklären Sie den Hauptzweck von Generics. Des weiteren geben Sie mindestens 4 wichtige Syntaxelemente (im Zusammenhang mit Generis) und deren Bedeutung an. **7 Punkte**

2. Erklären Sie die Schritte, welche notwendig sind, um ein generisches Java Programm in eines ohne Generics umzuschreiben. Geben Sie dazu ein anschauliches Codefragment an (jeweils mit und ohne Generics). **7 Punkte**

3. Schreiben Sie eine generische Swap–Methode, welche es erlaubt, die Werte zweier Variablen (bzw. Speicherplätze) zu tauschen, unabhängig von deren Typ. Begründen Sie Ihre Lösung! **6 Punkte**

Lösung

1. Generics werden verwendet, um parametrischen Polymorphismus zu realisieren. D.h., Methoden und Klassen können für einen Typ-Parameter definiert werden, der dann später beliebig instantiiert werden kann. Dabei bleibt die Typsicherheit erhalten.

   - <*T*> ... zur Deklaration eines generischen Typs
   - <T *extends* V> ... zur Deklaration einer upper bound
   - <T *super* V> ... zur Deklaration einer lower bound
   - <*?*> ... zur Deklaration einer Wildcard
   - <*String*> ... Instantiierung eines Typ-Arguments
   - <T extends V *&* W> ... zur Forderung zusätzlicher Interfacerealisierungen

2. (a) Typvariablen entfernen
   (b) Typparameter entfernen
   (c) Typecasts einführen

```
//with generics

class Pair<S, T> {
    S x;
    T y;

    S getX() {
        return x;
    }
}

...
Pair<String,Integer> siPair;
...
String s = siPair.getX();
...


//legacy code

class Pair {
    Object x;
    Object y;
```

```
    Object getX() {
        return x;
    }
}

...
Pair siPair;
...
String s = (String) siPair.getX();
...
```

3. Java verwendet Referenzen, daher kann es zwei Variablen nur dann vertauschen, wenn sie Teil derselben Klasse sind.

```
class Pair<T> {
    T first;
    T second;
}

public static <T> void swap(Pair<T> p) {
   T temp = p.first;
   p.first = p.second;
   p.second = temp;
}
```

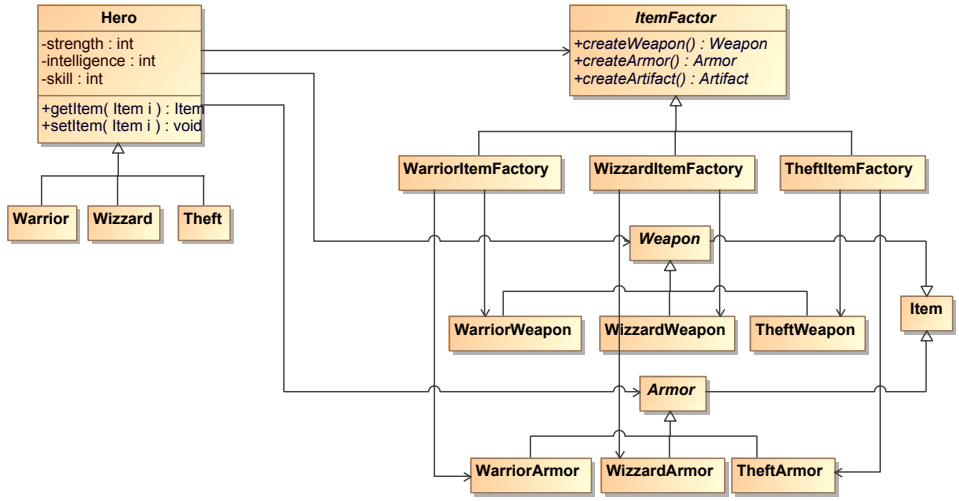

Abbildung 1: Lösung

Aufgabe 5) Modellierung (20 Punkte)

1. Es soll ein Fantasy Rollenspiel implementiert werden. Es gibt drei verschiedene Sorten von Helden: Krieger, Magier und Dieb. Ein Charakter definiert sich durch die Eigenschaften Stärke, Intelligenz und Geschicklichkeit. Charaktere sollen Items aufnehmen und Ablegen können. Für jeden Charakter gibt es folgende Items: Je eine Waffe, ein Kleidungsstück und einen Spezialgegenstand. Diese sehen je nach Charakter unterschiedlich aus. Beschreiben Sie in UML ein Designpattern, mit dem Sie die unterschiedlichen Aussehen der Charaktere erzeugen können. Es soll ein Onlinespiel ermöglicht werden, bei dem die Daten des Charakters auf Anforderung übertragen werden müssen. Vom Server soll nur eine Instanz zu erzeugen sein. **15 Punkte**
2. Erläutern Sie, welche Designpattern Sie verwendet haben, und wie Sie diese angewendet haben. **5 Punkte**

Lösung

1. Siehe Abbildung 1
2. Identifizierte Pattern:
   (a) Abstract Factory Pattern: Das unterschiedliche Aussehen der Items
   (b) Proxy Pattern: Übertragen nur auf Anforderung
   (c) Singleton Pattern: Server