Programmiermethodik
1. Klausur **Lösung**

27. 6. 2013

Name	
Matrikelnummer	

Aufgabe	**mögliche Punkte**	**erreichte Punkte**
1	21	
2	20	
3	19	
4	19	
5	21	
6	20	
Gesamt	**120**	

Aufgabe 1) Objekt-Orientierung und Vererbung (21 Punkte)

1. Erklären Sie die drei Prinzipien der Objektorientierung: Datenkapselung, Vererbung und Polymorphismus. Geben Sie für jeden dieser Begriffe eine Definition (eventuell anhand eines Beispiels) und erläutern Sie, inwiefern dadurch die Codequalität erhöht wird. **10 Punkte**

2. Was versteht man unter dem Problem der instabilen Basisklasse? Nennen Sie eine Möglichkeit, mit der man dieses Problem umgehen kann, falls Änderungen in der Basisklassse erlaubt sein sollen. **4 Punkte**

3. Betrachten Sie das folgende Programm. Welche 6 Ausgaben entstehen? **3 Punkte**

```
1
2  public class Equals {
3
4      public static void main(String[] args) {
5          System.out.println(Integer.valueOf(42) == Integer.valueOf(42));
6          System.out.println(new Integer(42) == new Integer(42));
7          System.out.println(new Integer(42).equals(new Integer(42)));
8          System.out.println(Integer.valueOf(789) == Integer.valueOf(789));
9          System.out.println(new Integer(789) == new Integer(789));
10         System.out.println(new Integer(42).equals(42));
11     }
12
13 }
```

4. Versuchen Sie die Begriffe Überladen und Überschreiben zu definieren. Arbeiten Sie insbesondere den Unterschied der beiden Begriffe deutlich heraus – wann spricht man von Überladen, wann von Überschreiben? **4 Punkte**

Lösung

1. Datenkapselung: Zugriff auf Variablen nach außen hin nicht uneingeschränkt. Oftmals Getter und Setter um Zugriff zu Regeln. Vorteile: Schnittstelle per Methoden, schränkt Zugriff ein; Invarianten könnten einfach sichergestellt werden

   Vererbung: Variablen und Methoden können von Klassen übernommen, verändert und erweitert werden. Vorteile: Code ist übersichtlicher und weniger Redundant; Gemeinsamkeiten werden zusammengefasst; Prinzip der Ersetzbarkeit, ...

   Polymorphimus: Überladen und Überschreiben von Methoden; mehrere Objekte können verschiedene Varianten einer Methode anbiten, bzw. eine Klasse kann verschiedene Varianten einer Methode für verschiedene Argumente anbieten. Vorteile: gleiche Methodennamen für gleiche Funktionalität; weg von konkreter Methode und hin zu konkreter Funktionalität

2. Instabile Basisklasse: Veränderungen einer Klasse von der geerbt wird kann mitunter Probleme bereiten. Beispiel: Variable size bisher explizit gespeichert wird entfernt, da Methode size() aufgerufen werden kann. Verwendet jedoch eine Subklasse diese Variable muss diese Änderungen auch dort vollzogen werden. Lösungen: Keine Änderungen in der Basisklasse (trivial), Interfaces statt Vererbung, Datenkapselung in der Basisklasse

3. true (Referenzvergleich, aber gecachete Objekte), false (Referenzvergleich), true (equals prüft korrekt auf Gleichheit), false (Referenzvergleich), false (Referenzvergleich), true (equals und autoboxing)

4. Überladen: Selber Methodenname, aber andere Argumente in einer Klasse -> eine zusätzliche Option

   Überschreiben: Exakt selbe Signatur, auf Ursprüngliche Methode kann nicht mehr von außen zugegriffen werden

Aufgabe 2) Schnittstellen und Abstrakte Klassen (20 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. Ist folgendes Interface korrekt? Erklären Sie warum oder warum nicht.

```
public interface ReadOnly {
}
```

2 Punkte

3. Kompiliert dieser Code? Erklären Sie warum oder warum nicht.

```
1 interface Animal {
2     void eat();
3 }
4
5 public class Dog implements Animal {
6     void eat() {}
7 }
```

3 Punkte

4. Beantworten Sie folgende Fragen kurz in Stichworten
   (a) Können abstrakte Klassen einen Konstruktor haben? Wenn nicht, warum?
   (b) Kann man eine abstrakte Klasse instantiieren? Wenn nicht, warum?
   (c) Müssen abstrakte Klassen von Interfaces geerbte Methoden implementieren? Wenn nicht, warum?
   (d) Können abstrakte Klassen `final` sein? Wenn nicht, warum?
   (e) Müssen abstrakte Klassen abstrakte Methoden haben? Wenn nicht, warum?

5 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. Ja, Interface brauch keine Methoden haben (Marker Interface)
3. Nein, `eat()` müsste `public` sein.
4. Aufg4
   (a) Ja
   (b) Nein, da sie abstrakte Methoden beinhalten könnten, die nicht implementiert sind.
   (c) Nein, diese können weiterhin abstrakt bleiben.
   (d) Nein, sonst könnte man abstrakte Methoden nie implementieren.
   (e) Nein, ist nicht vorgeschrieben.

Aufgabe 3) Ausnahmen (19 Punkte) Betrachten Sie den folgenden Code.

```
1  package exam1;
2  import java.io.*;
3
4  class Pair<A, B> {
5      public A a;
6      public B b;
7  }
8
9  interface Map<K, V> {
10     public V getValue(K key);
11 }
12
13 class ArrayMap<K, V> implements Map<K, V> {
14
15     private Pair<K, V>[] map;
16     private int maxIndex;
17
18     // map entries are between 0 and maxIndex
19
20     // binary search, returns index or -1, if key is not present
21     private int find(K key, int left, int right) {
22         if (left > right) {
23             return -1;
24         }
25         int middle = (left + right) / 2;
26         int c = key.compareTo(map[middle].a);
27         if (c == 0) {
28             return middle;
29         }
30         if (c < 0) {
31             return find(key, left, middle - 1);
32         } else {
33             return find(key, middle + 1, right);
34         }
35     }
36
37     public V getValue(K key) {
38         int i = find(key, 0, maxIndex);
39         // TODO
40     }
41 }
42
43 class PhoneBook {
44
45     public Map<String, String> book; // name -> number
46     // book is initiliazed somewhere else
47
48     public void printNumbersFromFile(String fileName) {
49         File f = new File(fileName);
50         BufferedReader reader = new BufferedReader(new FileReader(f));
51         while (reader.ready()) {
52             String name = reader.readLine();
53             System.out.println(book.getValue(name));
54         }
55         reader.close();
56     }
57 }
```

1. Die Methode `find` in Zeilen 21–35 implementiert eine binäre Suche. Passen Sie die Klassendeklaration in Zeile 13 so an, dass `find` ohne Fehler kompilierbar ist. **2 Punkte**

2. Bilden Sie eine Klasse für eine Ausnahme, die der Anforderung eines Eintrags zu einem nicht-existierenden Schlüssel entspricht. Dies sollte keine Runtime-Exception sein! Implementieren Sie damit den fehlenden Code in Zeile 39 von `getValue`. Wie und wo muss die Methoden-Deklarationen von `getValue` angepasst werden? **6 Punkte**

3. Im Code von `printNumbersFromFile` in Zeilen 48–56 wurde die Ausnahme-Behandlung vergessen. Implementieren Sie die Methode erneut, wobei jetzt jedoch die Ausnahmen ordentlich behandelt werden sollen (hier: durch Ausgabe kurzer Fehlermeldungen auf `System.err`). Die Signatur der Methode darf dabei nicht verändert werden. **11 Punkte**

Lösung

```
interface Map<K, V> {
    public V getValue(K key) throws UnknownKeyException;
}

class UnknownKeyException extends Exception {
}

class ArrayMap<K extends Comparable<? super K>, V> implements Map<K, V> {
    public V getValue(K key) throws UnknownKeyException {
        int i = find(key, 0, maxIndex);
        if (i == -1) {
            throw new UnknownKeyException();
        } else {
            return map[i].b;
        }
    }
}

class PhoneBook {
    public void printNumbersFromFile(String fileName) {
        BufferedReader reader = null;
        File f = new File(fileName);
        try {
            reader = new BufferedReader(new FileReader(f));
            while (reader.ready()) {
                String name = reader.readLine();
                try {
                    System.out.println(book.getValue(name));
                } catch (UnknownKeyException e) {
                    System.err.println("no entry for " + name);
                }
            }
        } catch (IOException e) {
            System.err.println("some IO error occured");
        } finally {
            if (reader != null) {
                try {
                    reader.close();
                } catch (IOException e) {
                    System.err.println("error when closing file");
                }
            }
        }
```

```
    }
}
```

Aufgabe 4) Generische Programmierung (19 Punkte)

1. Erklären Sie den Unterschied zwischen den zwei nachfolgenden, generischen Klassen. Was könnte im Body von `foo(...)` (anstelle der ...) stehen, damit der Compiler die obere Version von Klasse `C` akzeptiert, nicht aber die untere?

```
1  class C<T> {
2    T x;
3
4    void foo(T y) {...}
5  }
```

```
6  class C<T> {
7    T x;
8
9    <T> void foo(T y) {...}
10 }
```

4 Punkte

2. Die Anweisungsfolge

```
1  String[] sarray = new String[10];
2  Object[] oarray = sarray;
```

wird vom Compiler akzeptiert, die Anweisungsfolge

```
3  List<String> slist = new LinkedList<String>();
4  List<Object> olist = slist;
```

jedoch nicht.

Erklären Sie kurz, warum dies so ist, warum also die Anweisungen mit den Arrays kein Problem für die Typsicherheit darstellen, die Anweisung in Zeile 4 bei den generischen Listen jedoch schon.
6 Punkte

3. Betrachten Sie das nachfolgende Interface `Addable`.

```
public interface Addable<T> {
  public T add(T x);
}
```

Implementieren Sie eine generische, statische Methode `sum`, die eine `java.util.ArrayList` mit `Addable` Elementen annimmt und die Summe der Elemente zurückgibt. Sie können davon ausgehen, dass die `ArrayList` mindestens ein Element enthält. **9 Punkte**

Lösung

1. `foo` in der unteren Klasse `C` ist eine generische Methode mit eigenem Typparameter `T`, welcher unabhängig vom Typparameter `T` der Klasse ist. Beispielsweise ist die Anweisung `x = y;` in der `foo`-Methode nur in der oberen Klasse möglich.

2. Die Typsicherheit bei den Arrays wird zur Laufzeit von der JVM sichergestellt. Zum Beispiel führt der Code `oarray[0] = new Object();` zur Laufzeit zu einer `ArrayStoreException`. Hingegen wird der Elementtyp bei der generischen Liste mittels Type-Erasure gelöscht. Demzufolge muss Zeile 4 zur Übersetzungszeit abgelehnt werden, damit etwa `olist.add(new Object()); String s = slist.get(0);` nicht zu Fehlern führt.

3.

```
public static <T extends Addable<T>> T sum(ArrayList<T> list) {
    Iterator<T> i = list.iterator();
    T current = i.next();
    while (i.hasNext()) {
        current.add(i.next());
    }
    return current;
}
```

Aufgabe 5) Multiple Choice (21 Punkte)

Markieren Sie die richtigen Antworten mit einem Haken (✓). Jede richtige Antwort gibt 3 Punkte. Jede falsche Antwort gibt 0 Punkte (für diese Antwort). Jede nicht beantwortete Frage ergibt 1 Punkt.

	Wahr	Falsch
Kontravarianz des Rückgabewert beim Überschreiben von Methoden bedeutet, dass man eine Methode `Object foo()` mit einer Methode `String foo()` überschreiben kann.		✓
Generische Ausnahmetypen in Java sind deshalb nicht erlaubt, weil die Ausnahmebehandlung zur Laufzeit geschieht, und die Laufzeitumgebung generische Typen nicht kennt.	✓	
Black-box Testen erstellt Tests anhand des zu testenden Programms, bei dem versucht wird, alle Verzweigungen im Programm abzudecken.		✓
Unter Kohäsion versteht man, wie stark Klassen miteinander in Verbindung stehen. Ziel einer guten Modellierung ist eine möglichst geringe Kohäsion, damit Klassen leicht ausgetauscht werden können.		✓

Weil Java einen objekt-basierten Zugriffsschutz benutzt, ist der Zugriff auf `x` in der Methode `foo()` nicht gestattet.
`class A { private int x; }`
`class B { static int foo(A a){return a.x;}}`
		✓

Schwache Typisierung bedeutet, dass es erlaubt ist, ein Objekt einer Variable zuzuweisen, ohne dass das Objekt die Spezifikation des Typs der Variable erfüllt.	✓	
Die Verwendung des "Erbauer"-Musters erleichtert den Austausch von Produkt-Familien.		✓

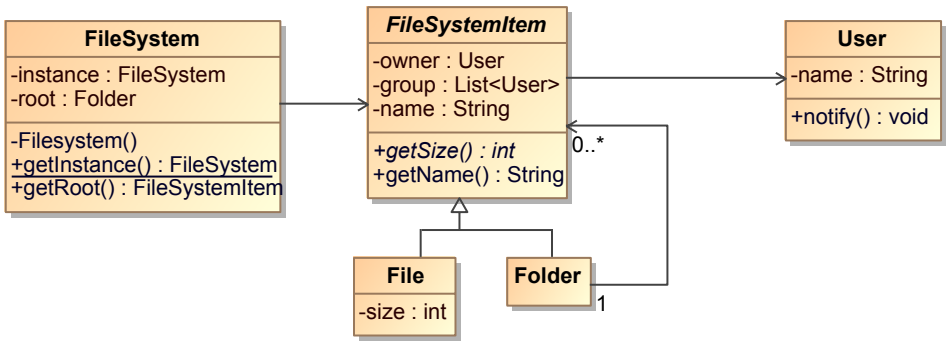

Abbildung 1: Lösung

Aufgabe 6) Modellierung (20 Punkte)

1. Modellieren Sie ein Dateisystem in UML. In dem Dateisystem soll es Ordner und Dateien geben, die jeweils über einen Namen verfügen. In einem Ordner können beliebig viele Unterordner und Dateien liegen. Dateien haben eine Größe. Es soll möglich sein die Größe der Dateien in allen Unterordnern eines bestimmten Ordners zu berechnen. Dateien und Ordner gehören immer einem bestimmten User. User verfügen über einen Namen. User können andere User zu einer Liste hinzufügen, die besagt, dass der andere User eine bestimmte Datei bearbeiten darf. Wenn ein User die Datei eines anderen verändert, bekommt dieser eine Benachrichtigung. Es soll eine Klasse im Dateisystem geben, von der es genau eine Instanz gibt, über das die User auf den Ordner, der die Wurzel des Dateisystems darstellt, zugreifen können. Im nächsten Schritt soll das Dateisystem fit für eine Serveranwendung gemacht werden. **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) Composite Pattern: File System aus, Abstrakter Klasse und 2 erbenden Klassen, bei der eine Erbende die Abstrakte beinhaltet.
   (b) Observer Pattern: User hat notify. Push.
   (c) Singleton Pattern: Klasse mit Konstruktor private und static Methode.