Übungsblatt 03

2 Aufgaben, 20 Punkte

Objektorientierte Modellierung und Programmierung (inf031) Sommersemester 2020
Carl von Ossietzky Universität Oldenburg, Fakultät II, Department für Informatik

Dr. C. Schönberg

Ausgabe: 2020-05-05 12:00 **Abgabe:** 2020-05-12 10:00

Aufgabe 1: *Bankkonten* **(3 + 2 + 5 Punkte)**

Gegeben sei folgendes Java-Programm:

```java
public class Bank {

  private String name;
  private Account[] accounts;

  public Bank(String name) { this.name = name; }

  // getter and setter methods

}
```

```java
public class Account {

  private Customer[] holders;
  private long balance;
  private String iban;

  public Account(String iban) { this.iban = iban; }

  // getter and setter methods

}
```

```java
public class Customer {

  private String name;
  private Account[] accounts;
  private Address homeAddress;
  private Address workAddress;

  // getter and setter methods

}
```

```java
public class Address {

  private String street;
```

```
4     private String postCode;
5     private String city;
6
7     // getter and setter methods
8
9   }
```

```
1   public class Banking {
2
3     public static void main(String[] args) {
4       Bank sbt = new Bank("Smaug Bank & Trust");
5       sbt.setAccounts(new Account[1]);
6       sbt.getAccounts()[0] = new Account("ER99123412341234123412");
7       sbt.getAccounts()[0].setBalance(54100000000L);
8       Customer thorin = new Customer();
9       thorin.setAccounts(new Account[1]);
10      thorin.getAccounts()[0] = sbt.getAccounts()[0];
11      thorin.setName("Thorin");
12      Address home = new Address();
13      home.setStreet("Kingsroad 1");
14      home.setPostCode("12345");
15      home.setCity("Dunland");
16      thorin.setHomeAddress(home);
17      Address work = new Address();
18      work.setStreet("Throneroom 1");
19      work.setPostCode("54321");
20      work.setCity("Erebor");
21      thorin.setWorkAddress(work);
22      sbt.getAccounts()[0].setHolders(new Customer[] { thorin });
23    }
24
25  }
```

a) Entwerfen Sie ein *UML-Klassendiagramm* für die Klassen `Bank`, `Account`, `Customer` und `Address`. Achten Sie auf die Verwendung von passenden Modellierungs- und Programmier-Konzepten. Die getter- und setter-Methoden dürfen Sie weglassen.

b) Entwerfen Sie außerdem ein passendes *UML-Objektdiagramm* für den Zustand des Systems nach Ausführung der `Banking.main()`-Methode.

c) Bei näherer Betrachtung fällt Ihnen auf, dass Sie der Arbeitsadresse `workAddress` (nicht aber der Privatadresse `homeAddress`) einen Firmennamen `companyName` hinzufügen wollen. Umgekehrt wollen Sie ausschließlich der Privatadresse ein optionales Postfach (bestehend aus Postleitzahl `poBoxCode` und Ort `poBoxCity`) hinzufügen.
Außerdem wollen Sie neben den Bankkunden auch die Bankberater `FinancialAdvisor` mit abbilden. Diese unterscheiden sich von den Kunden nur dadurch, dass sie keine Konten haben, sondern eine Reihe von Kunden betreuen.
Ergänzen Sie Ihr *UML-Klassendiagramm* und die *Java-Implementierung* um diese zusätzlichen Informationen. Achten Sie auf die Verwendung von passenden Modellierungs- und Programmier-Konzepten.

Aufgabe 2: *Listen* **(1 + 1 + 1 + 2 + 2 + 1 + 2 Punkte)**

Gegeben sei folgende Implementierung einer verketteten Liste (vgl. beiliegende zip-Datei):

```
1  public class LinkedStringList {
2
3    private LinkedStringListElement start;
4
5    public int size() {
6      // ...
7    }
8
9    public String get(int index) {
10     // ...
11   }
12
13   public void add(String value) {
14     // ...
15   }
16
17   public String remove(int index) {
18     // ...
19   }
20
21 }
22 class LinkedStringListElement {
23
24   private LinkedStringListElement next;
25   private String value;
26
27   // getter and setter methods
28
29 }
```

Schreiben Sie eine Spezialisierung `VersatileLinkedList` dieser Liste, die folgende zusätzliche Methoden zur Verfügung stellt:

a) Eine Methode `add` zum Einfügen von **`int`**-Werten in die Liste. Die **`int`**-Werte sollen dazu in `String`s umgewandelt werden (z.B. mit der Klassenmethode `Integer.toString(value: int): String`). Die Methode `add` wird im folgenden mehrfach überladen.

b) Eine Methode `add` zum Einfügen von **`boolean`**-Werten in die Liste. Die **`boolean`**-Werte sollen dazu jeweils in die `String`s "yes" und "no" umgewandelt werden.

c) Eine Methode `add` zum Einfügen *aller* Werte aus einer übergebenen `LinkedStringList` in die Liste. Es sollen die einzelnen Werte der übergebenen Liste eingefügt werden, nicht die übergebene Liste selbst.

d) Eine Methode `add` zum Einfügen *einiger* Werte aus einer `LinkedStringList` in die Liste. Die Methode bekommt neben der Liste noch zwei **`int`**-Parameter `start` und `end`. Es sollen die einzelnen Werte der übergebenen Liste mit einem Index zwischen `start` (enschließlich) und `end` (ausschließlich) eingefügt werden.
Bei einem Aufruf mit der Liste ["a", "b", "c", "d"], `start`=1 und `end`=3 sollen beispielsweise die Werte "b" und "c" eingefügt werden.

e) Eine Methode `reverse`, die eine `VersatileLinkedList` zurückgibt, welche alle Elemente der aktuellen Liste enthält, aber in genau umgekehrter Reihenfolge. Enthält die aktuelle Liste beispielsweise die Elemente ["a", "b", "c", "d"], so soll die zurückgegebene Liste die Elemente ["d", "c", "b", "a"] enthalten.

f) Eine Methode `equals`, die eine übergebene `VersatileLinkedList` mit den aktuellen Werten vergleicht. Stimmen alle Elemente in der richtigen Reihenfolge überein (`String.equals()`), so gibt die Methode **`true`** zurück, sonst **`false`**.

g) Eine statische `main`-Methode, die das Verhalten der von Ihnen geschriebenen `VersatileLinkedList`-Methoden testet.

Sie dürfen die Klassen `LinkedStringList` und `LinkedStringListElement` *nicht* verändern!