

Beispiel 1 - Ticket-Terminal einer Reederei (55 Punkte)



Implementieren Sie in Java die im Klassendiagramm (s. Seite 9) rot markierten Klassen und Methoden in Entsprechung zu dem Klassen- und den Sequenzdiagrammen auf Seiten 3 u. 11 und den unten angeführten Anforderungen.

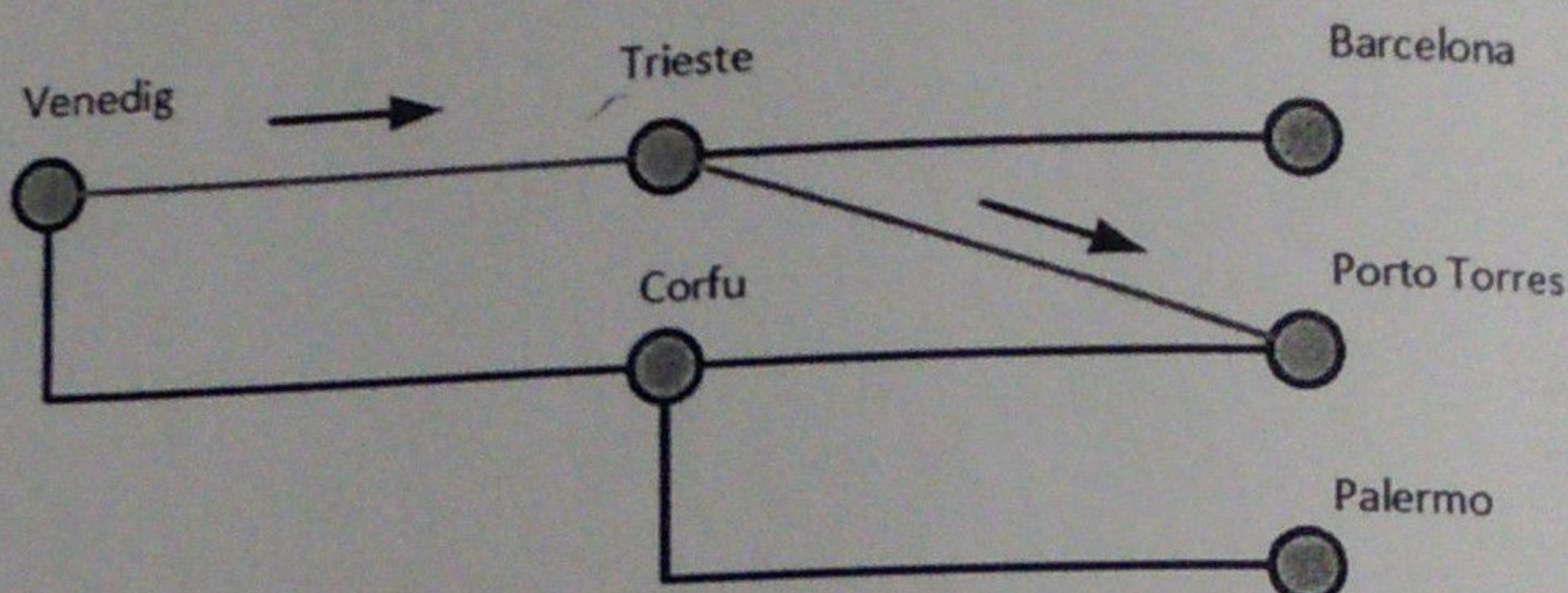
- Rot markierte Klassen erfordern die Klassendeklaration, Attribute und Assoziationen.
- Rot markierte Methoden müssen implementiert werden.

Das Sequenzdiagramm auf Seite 11 definiert den Vorgang der Buchung eines Tickets für eine Reiseroute durch einen Kunden über das User Interface eines Ticket-Terminals. Die Klasse UserInterface kapselt die Schnittstelle zum Anwender und ist hier *nicht* zu implementieren, sondern kann als gegeben angenommen werden.

Mit dem Ticket-Terminal löst ein Kunde das Ticket für seine Reiseroute. Dazu meldet sich der Kunde am Ticket-Terminal an und wählt eine Bezahlweise. Anschließend legt er seine Reiseroute, die aus mehreren Fährverbindungen (im folgenden Verbindungen genannt) bestehen kann, fest. Ausgangspunkt dieser Reiseroute ist der Standort des Ticket-Terminals. Für diesen Standort werden dem Kunden alle ausgehenden Verbindungen vorgeschlagen und der Kunde wählt eine davon aus. Anschließend kann der Kunde weitere Verbindungen zu seiner Reiseroute hinzufügen oder die Buchung des Tickets durchführen.

Möchte der Kunde eine weitere Verbindung hinzufügen, werden ihm alle ausgehenden Verbindungen des zuletzt gewählten Zielorts vorgeschlagen. Dieser Ablauf wird so lange wiederholt, bis der Kunde keine weitere Fährverbindung hinzufügen möchte.

Beispiel: Die Reise von Venedig nach Porto Torres besteht aus den Verbindungen Venedig – Trieste und Trieste – Porto Torres (siehe Abbildung rechts).



Weitere Anforderungen:

- Alle Benutzerinteraktionen müssen über die Klasse *TicketTerminal* erfolgen!

Die Klasse *TicketTerminal*

- Beim Erzeugen eines Objekts dieser Klasse müssen auch alle zugehörigen Assoziationen im Klassendiagramm initialisieren werden.
- Im Zuge der Fehlerbehandlung müssen dem Benutzer über das User Interface eventuell beim Buchungsvorgang aufgetretene Fehlerfälle angezeigt werden.

Die Klasse *UserInterface*

- Alle Ausgaben und Fehlermeldungen müssen über die Methode *display(UIMessages message)* ausgegeben werden.

Die Klasse *EnterpriseServer*

- Die Methode *getConnections(String origin, String dest)* muss folgendes Verhalten implementieren:
 1. Ist der Parameter *Start* eine Referenz und der Parameter *Ziel* NULL, so werden alle Verbindungen zurückgegeben, bei denen der Startort gleich dem Parameter *Start* ist.
 2. Sind beide Parameter nicht NULL, so werden alle Verbindungen zurückgegeben, bei denen der Startort gleich dem Parameter *Start* und der Zielort gleich dem Parameter *Ziel* ist.
 3. Wurde keine Verbindung gefunden, muss eine *NoConnectionException* geworfen werden.

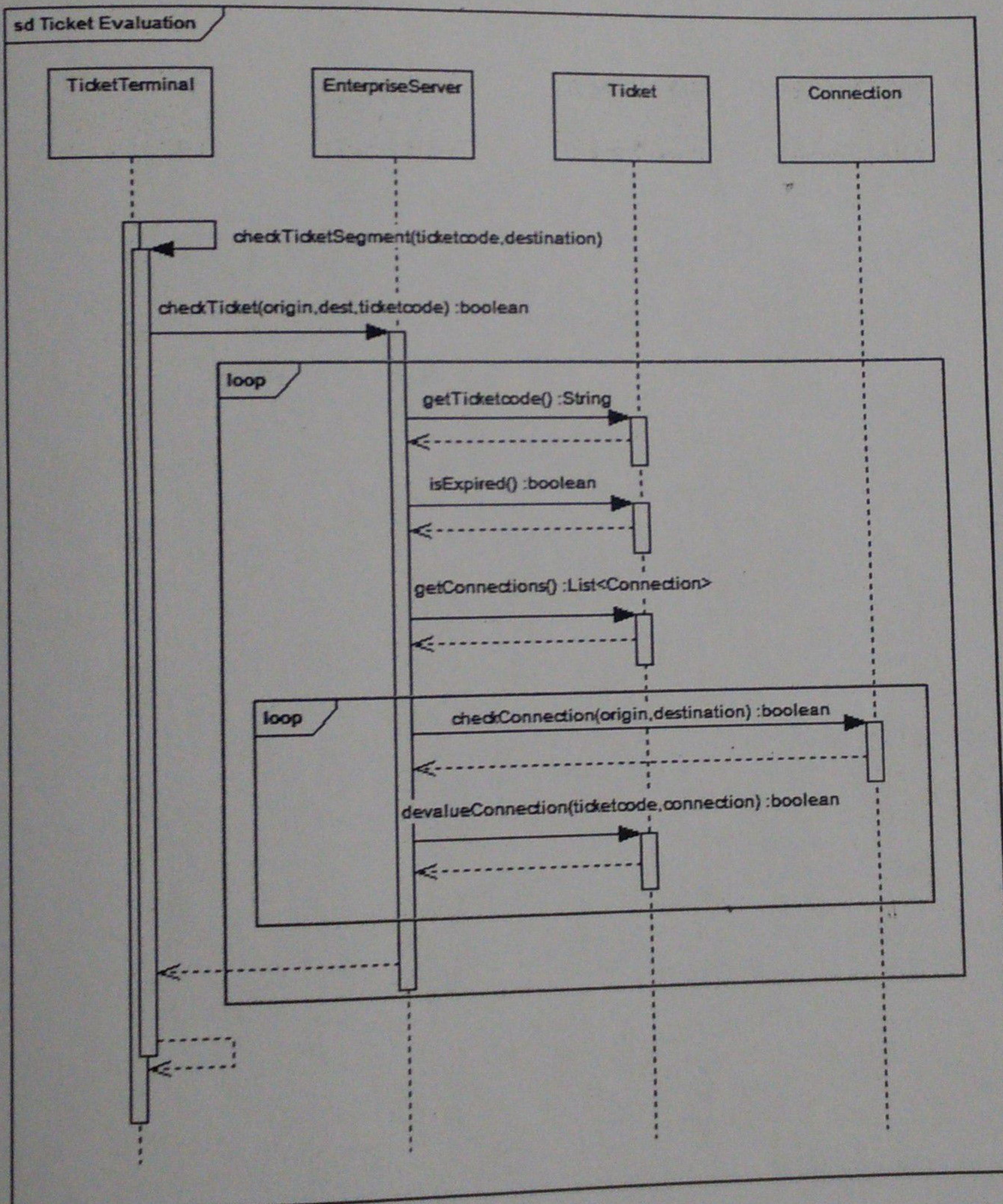
<< Enumeration >>	
UIMessages	
NO_CONNECTION	
CONFIRM_BOOKING	
GET_CONFIRMATION	
PAYMENT_FAILED	
NO_TICKET	
TICKET_DEVALUED	
TICKET_INVALID	

- Die Methode *createTicket(...)* muss eine *PaymentException* werfen, wenn die Prüfung der *PaymentCard* ergibt, dass diese ungültig ist. In diesem Fall kann die Buchung nicht durchgeführt werden und es darf auch kein Ticket gespeichert werden. Im Zuge der Fehlerbehandlung ist dem Benutzer die Fehlermeldung *PAYMENT_FAILED* anzuzeigen.
- Die Methode *checkTicket(...)* überprüft ein Ticket und entwertet es gegebenenfalls. Anhand des Ticketcodes wird das entsprechende Ticket ermittelt und überprüft, ob dieses (noch) gültig ist. Für ein gültiges

Ticket wird eine noch nicht entwertete Verbindung, die den Suchparametern entspricht, entwertet und *true* zurückgegeben. Am Ticket-Terminal wird dem Benutzer die Bestätigungsmeldung **TICKET_DEVALUED** angezeigt.

Folgende alternative Abläufe sind zu berücksichtigen:

1. Wird ein Ticket wie oben beschrieben gefunden, ist aber die den Suchparametern entsprechende Verbindung bereits entwertet, so wird *false* zurückgegeben. Am Ticket-Terminal wird dem Benutzer die Hinweismeldung **TICKET_INVALID** angezeigt.
2. Wird ein Ticket gefunden, ist dieses aber nicht (mehr) gültig, so wird *false* zurückgegeben. Am Ticket-Terminal wird dem Benutzer die Hinweismeldung **TICKET_INVALID** angezeigt.
3. Wird kein entsprechendes Ticket gefunden, muss eine *TicketException* geworfen werden. Im Zuge der Fehlerbehandlung ist dem Benutzer die Fehlermeldung **NO_TICKET** anzuzeigen.
4. Wird ein Ticket gefunden und ist dieses gültig, aber keine den Suchparametern entsprechende Verbindung gebucht, so muss eine *TicketException* geworfen werden. Im Zuge der Fehlerbehandlung ist dem Benutzer die Fehlermeldung **NO_TICKET** anzuzeigen.



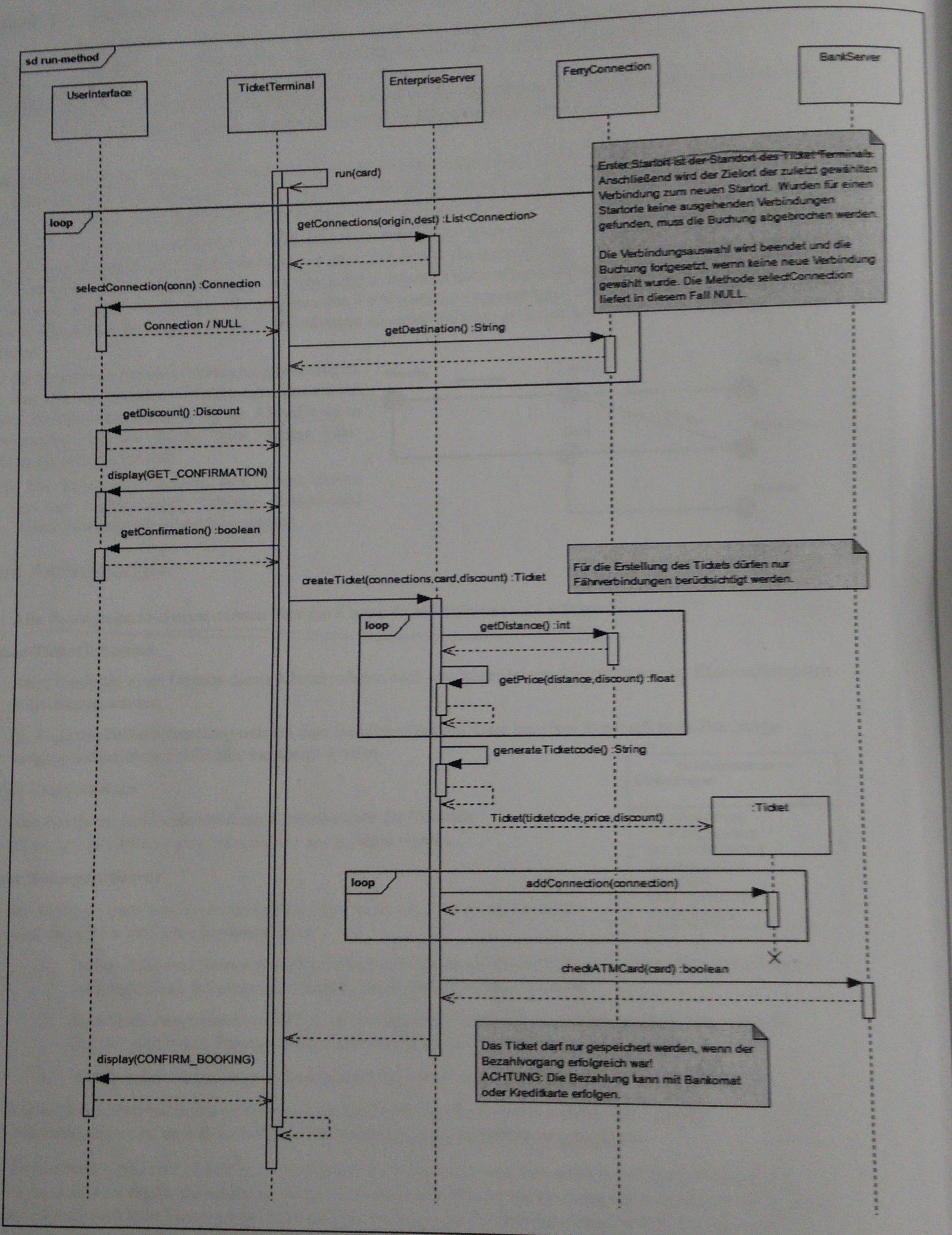
Die Klasse FerryConnection

- Die Methode `checkConnection(String origin, String destination)` muss *true* zurückgeben, wenn der Parameter *Start* mit dem Startort der Verbindung und der Parameter *Ziel* mit dem Zielort übereinstimmen. In allen anderen Fällen muss *false* zurückgegeben werden.
- Die Methode `compareOrigin(String origin)` muss *true* zurückgeben, wenn der Startort der Verbindung mit dem Parameter *Start* übereinstimmt. In allen anderen Fällen muss *false* zurückgegeben werden.

Die Klasse Ticket

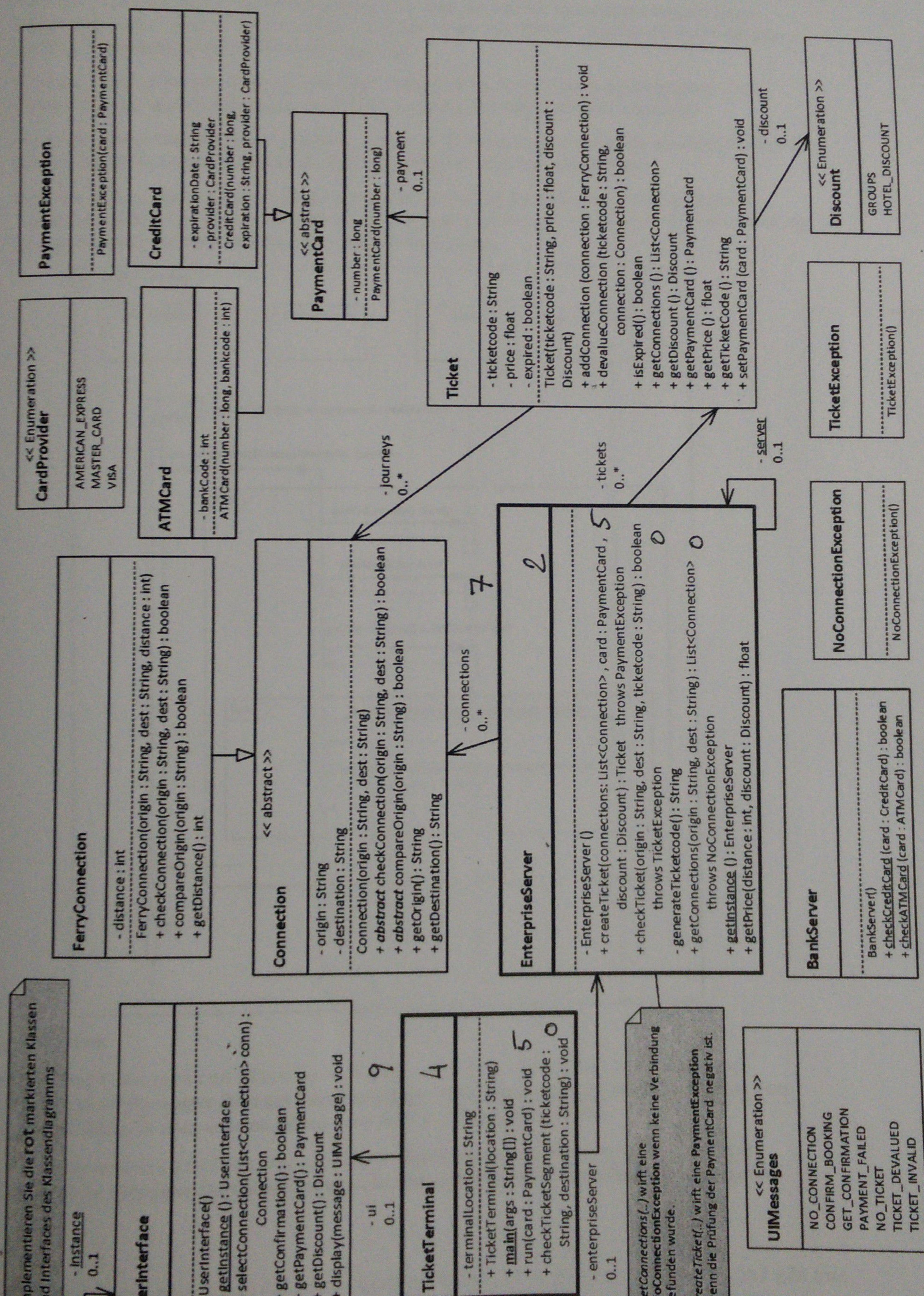
- Die Methode `devalueConnection(String ticketcode, Connection connection)` gibt *true* zurück, wenn die Verbindung erfolgreich entwertet werden konnte. In allen anderen Fällen wird *false* zurückgegeben.

Sequenzdiagramm:



Klassendiagramm

Bitte beachten Sie, dass Methoden, Referenzen und Attribute mit unterstrichenen Bezeichnungen statisch (static) sind. Variablen und Methoden sind entweder private, gekennzeichnet durch ein „-“ im Diagramm, oder public, gekennzeichnet durch ein „+“.



BAAHnVDE

Implementieren Sie hier die geforderten Klassen und Methoden:

en

```
t java.util.Collection;
```

```
t java.util.Vector;
```

```
c class TicketTerminal {
```

```
public static void main(String[] args) {
```

```
    TicketTerminal t = new TicketTerminal("Venedig");
```

```
    t.run(new ATMCARD(11,1));
```

```
}
```

```
// IMPLEMENTIEREN SIE AB HIER IN ENTSPRECHUNG ZU DEM KLASSEN- UND  
// DEM SEQUENZDIAGRAMM und den zusätzlich definierten Anforderungen!  
//  
// Viel Erfolg!
```

Beispiel 2

A) (5 Punkte) 1/5

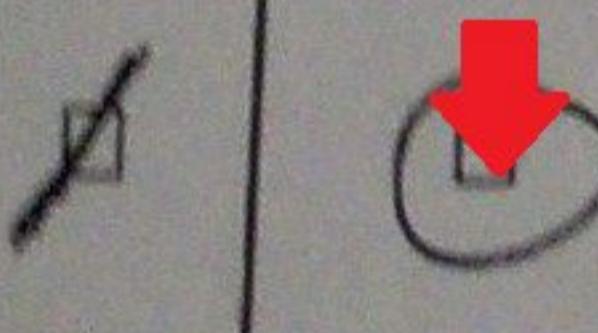
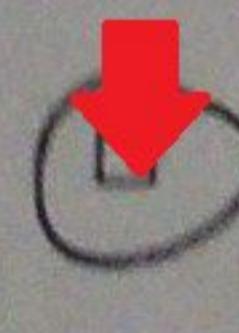
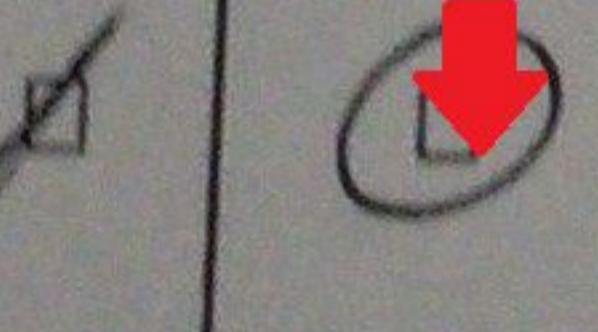
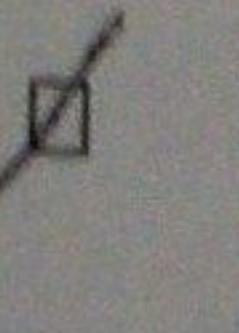
Gegeben sind die vier Java-Klassen **Person**, **Student**, **Teacher** und **Professor** sowie das Interface **Instructor**.

```
public abstract class Person {  
    protected String degree = "None";  
  
    public void changeDegree( Integer degree){  
        this.degree = degree.toString();  
    }  
  
}  
  
public interface Instructor {}  
  
public class Student extends Person {}  
  
public class Professor extends Person  
    implements Instructor {  
    public void changeDegree( String degree){  
        this.degree = degree;  
    }  
  
}  
  
public class Teacher  
    implements Instructor {}
```

```
public class Testklasse {  
    public static void main( String[] args ) {  
  
        Person them = new Teacher(); // C1  
  
        Student eg = new Student(); // C2  
        eg.changeDegree("Bakk"); // C2  
  
        Professor hk = // C3  
            (Instructor) new Professor(); // C3  
  
        Instructor he = new Professor(); // C4  
  
        Person aj = new Professor(); // C5  
        ((Professor)aj).changeDegree("Bakk"); // C5  
    }  
}
```

Sind die Aktionen/Operationen C1 bis C5 in der Klasse **Testklasse** erlaubt?

Kreuzen Sie an und begründen Sie Ihre Antworten (Antworten ohne Begründung werden nicht gewertet!).

Codezeile:	Erlaubt?		Begründung:
	Ja	Nein	
C1:	<input type="checkbox"/>	<input checked="" type="checkbox"/> 	
C2:	<input checked="" type="checkbox"/> 	<input type="checkbox"/> 	
C3:	<input checked="" type="checkbox"/> 	<input type="checkbox"/> 	
C4:	<input checked="" type="checkbox"/> 	<input type="checkbox"/> 	
C5:	<input checked="" type="checkbox"/> 	<input type="checkbox"/> 	

B) (10 Punkte)

2/10

Mat.Nr.: _____

Analysieren Sie die folgenden Java-Klassen **Bike** und **MountainBike**. Die **main**-Methode der Klasse **Test** erzeugt Objekte und sendet diesen Objekten Nachrichten. Geben Sie die jeweiligen Ausgaben zu den gefragten Zeitpunkten (T1 bis T8, siehe Kommentare in **main**) an.

```
public class Bike {
    protected int max_gear = 10;
    protected int curr_gear = 5;

    public int gearUp ( int num ) {
        int curr_gear = this.curr_gear; // 5
        this.curr_gear += num; // 10
        return curr_gear; // 10
    }

    public int gearUp () {
        return curr_gear + 1;
    }

    public int gearDown ( int num ) {
        return curr_gear;
    }

    public int getGear( int offset ) {
        return this.curr_gear;
    }

    public int getGear () {
        if(curr_gear > max_gear) {
            return max_gear;
        }
        return curr_gear;
    }
}
```

```
public class MountainBike extends Bike {
    protected int chainrings = 2;
    protected int curr_chainring = 1;

    public int gearUp ( int num ) {
        this.curr_gear = super.curr_gear;
        this.curr_gear += num; // 5+5=10
        return super.curr_gear;
    }

    public int gearDown ( int num ) {
        this.curr_gear = this.curr_gear - num;
        return super.curr_gear;
    }

    public int getGear ( int offset ) {
        return this.curr_gear - super.curr_gear + offset;
    }

    public int getGear () {
        return super.curr_gear +
            (super.max_gear * (curr_chainring - 1));
    }
}
```

```
public class Test {
    public static void main( String[] args ) {
        Bike b = new Bike();
        MountainBike mB = new MountainBike();

        out.println( mB.gearUp(5) );
    }
}
```

// Notizen sind hilfreich!

→
// T1

out.println(b.gearUp(5));

→
// T2

out.println(((Bike)mB).gearDown(2));

→
// T3

out.println(((Bike)mB).getGear(3));

→
// T4

out.println(b.getGear());

→
// T5

b = mB;
out.println(b.getGear());

→
// T6

out.println(((MountainBike)b).gearUp(((Bike)mB).gearDown(2))); // T7

out.println(((Bike)mB).gearUp(((MountainBike)b).gearDown(2))); // T8

Ausgaben:

T1: 10 ✓

T2: 5 ✓

T3:

T4:

T5:

T6:

T7:

T8:

Beispiel 4 (20 Punkte)

13

Beantworten Sie die vier nachfolgenden Fragen (je max. 5 Punkte) kurz aber dennoch prägnant, d.h. vollständig:

215

- A) Was ist Polymorphismus? Demonstrieren Sie dieses Konzept anhand von eigenen Java Code Beispielen.

4/5

- B) Nennen Sie fünf Arten des Testens, außer Black-Box und White-Box-Testen. Erklären und beschreiben Sie die von Ihnen genannten Test-Arten.

Beispiel 4

4/5

- C) Was versteht man unter „Patterns“? Erklären Sie zwei Patterns genauer.

3/5

- D) Erklären Sie das Konzept der Datenkapselung (Data encapsulation) allgemein und anhand eines Java Code Beispiels. Wie könnten Sie das Konzept der Datenkapselung verletzen? Wie steht dieses Konzept mit Klassenhierarchie und Vererbung in Zusammenhang?