Prüfungsklausur zur Vorlesung

Programmierung von Systemen

im Sommersemester 2020

Institut für Softwaretechnik und Programmiersprachen
Fakultät für Ingenieurwissenschaften und Informatik
Universität Ulm

Prof. Dr. Matthias Tichy, Stefan Götz

M U S T E R L Ö S U N G

30. September 2020

(2. Prüfungszeitraum)

Aufgabe 1 - OOP (7 + 4 + 3 = 14 Punkte)

Gegeben sei das Klassendiagram, welchen Sie auf dem beiliegenden Extrablatt finden können. Das Interface `Openable` definiert die Möglichkeit einen Ort für Außenstehende zu öffnen (`open`) oder zu schließen (`close`).

a) Implementieren Sie die Methoden `move(Stable host)` der Klasse `Horse` und `checkAddable()` der Klasse Stable. Mittels der Methode `checkAddable()` in `Stable` kann geprüft werden ob der Stall für Außenstehende geöffnet ist und ob noch Platz darin frei ist. Die Methode `Move` soll ein Pferd in einen neuen Stall stellen insofern dieser offen ist und Platz hat. `Move` soll *true* bzw. *false* zurück geben abhängig davon ob das Pferd bewegt werden konnte oder nicht. *Hinweis: Beachten Sie, dass Sie auch alle Referenzen updaten müssen wenn ein Pferd bewegt wird.*

```
public abstract class Horse {
  private Stable pos;

  public boolean move(Stable host) {
    if (host.isHospitable()) {//1.5P
      pos.removeHorse(this);//0.5P
      pos = host;           //0.5P
      pos.addHorse(this);   //0.5P
      //+ 1P reihenfolge von removeHorse, pos change beruecksichtigt
      return true;          //0.25P
    } else
      return false;         //0.25P
  }
}

public class Stable {
    private Horse[] holds = new Horse[20];

    public boolean isHospitable() {
              //0.5P        //2P alternativ Loop mit null check
      return hospitable && Arrays.asList(holds).contains(null);
    }
}
```

b) Erläutern Sie was das Singelton-Pattern ist, wann es eingesetzt werden kann und Erläutern Sie (ohne Code) wie es in Java umgesetzt werden kann.

Pattern das beschreibt, dass es von einer Klasse nur eine Instanz geben darf (1P).
Wird eingesetzt wenn man einen einzigen Point-of-Reference für eine Klasse erzwingen möchte. Z.B Logging oder Driver/Factories (1P).
*Man verwendet z.B einen **privaten Konstruktor** und eine **getInstance()** Methode die einen **Null check** auf einem **statischen Attribut** das die Instanz entält macht und diese dann **entweder erzeugt oder zurückgibt** (0.5P pro Beschriebenem **fetten** bis max 2P).*

c) Welche der folgenden Codestücke sind korrekt und welche führen zu **Compile**- oder **Laufzeitfehlern**? Begründen Sie, warum ein Codestück nicht korrekt ist. Sie können davon ausgehen, dass die Aufrufe aus einer `main`-Methode in einem anderen *Package* gemacht werden welches die nötigen imports vorweist.

```
var stable = new Stable();
stable.addHorse(new Horse());
```

Falsch, weil `Horse` eine Abstrakte Klasse ist und somit nicht erzeugt werden kann.

0.5P falsch, 0.5P abstrakt

```
Horse h = new Pony();
Openable hos = new Stable();
h.move(hos);
```

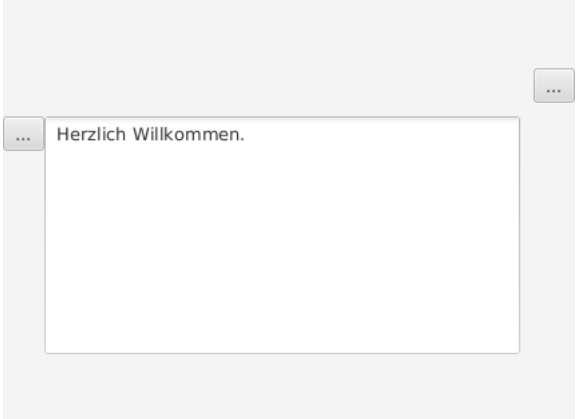
Falsch (0.5P), move braucht einen Stable (0.5P).

```
Stable stable = new Stable();
Openable h  = stable;
h.makeInhospitable();
```

Richtig 1P

Aufgabe 2 - JavaFX (6 + 3 + 1 = 10 Punkte)

Gegeben Sei die folgende Grafik:



a) Ergänzen Sie die `start`- Methode so, dass die oben gezeigte Grafik daraus resultiert. Beachten Sie hierfür auch das **Extrablatt** am Ende der Klausur.

```
        @Override
        public void start(Stage stage) {
            Button buttonTop = new Button("...");
            Button buttonBottom = new Button("...");
            TextArea textArea = new TextArea("Herzlich␣Willkommen");
            GridPane grid = new GridPane();
            HBox box = new HBox();
            box.getChildren().add(buttonTop);
            box.getChildren().add(textArea);
            grid.add(buttonBottom, 1, 0);
            grid.add(box, 0, 1);
            Scene scene = new Scene(grid, 800, 500);
            stage.setScene(scene);
            stage.show();
    }
```

- 0.5P Buttons initializieren
- 0.5P Textarea initializieren
- 0.5P GridPane initializieren
- 0.5P HBox initializieren
- 1P pro Elemente hinzugefügt (0.5P wenn positionierung falsch)
- 1P scene mit grid
- 1P scene zu stage adden
- 1P show

b) Welchen Code müssen Sie wo einfügen, damit beim Drücken eines der zwei Buttons der gedrückte Button verschwindet?

```
button.setOnAction(new EventHandler<ActionEvent>() {

    @Override
    public void handle(ActionEvent event) {
        box.getChildren().remove(button);
    }
});
```

- 1P setOnAction
- 0.5P new ActionEvent
- 0.5P handle-Methode
- 1P box.getChildren().remove(button)

c) Geben Sie CSS Code an um die Hintergrundfarbe alle Buttons **rot** zu färben.

```
.button {
    -fx-background-color: red;
}
```

Aufgabe 3 - Java IO (7 + 4 = 11 Punkte)

a) Implementieren Sie eine Methode welche die ersten 20 Bytes von allen übergebenen Channels nacheinander in die Datei **out.data** schreibt. Verwenden Sie hierfür ausschließlich Methoden aus der Streams-API für Java Collections KEINE manuell implementierten Schleifen.

Achten Sie auf Behandlung der folgenden Exception mit einer passenden Meldung:

- `IOException`

Beachten Sie hierfür auch den JavaIO Teil des **CheatSheets** am Ende der Klausur.

```
public static void useBytes(FileChannel[] ics) {
    try {
        FileOutputStream fos = new FileOutputStream(new File("out.data"));

        var targetChannel = fos.getChannel();

        Stream<FileChannel> stream = Arrays.stream(ics);
        stream.forEach(fc -> fc.truncate(20));
        stream.forEach(fc -> fc.transferTo(0, fc.size(), targetChannel));
        stream.forEach(fc -> fc.close());

        // close target
        targetChannel.close();
        fos.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

- 1P FileOutputStream erzeugen
- 1P Channel aus outputStream generiern
- 1P stream aus ics erzeugen (auch ok wenn das jedes mal neu gemacht wird)
- 1P truncate(20) in forEach
- 1P transfer in forEach
- 0.5P fc close
- 0.5P targetChannel close
- 0.5P fos close;
- 0.5P für IOException tryCatch

b) Java bietet eine Vielzahl an Streams für die Verarbeitung von Daten an. Darunter auch gepufferte und nicht gepufferte Varianten. Erläutern Sie textuell anhand von `BufferedOutputStream` und `OutputStream` die Unterschiede zwischen gepufferten und nicht gepufferten Streams. Erläutern Sie zusätzlich mit Bezug auf `OutputStream` wie das Decorator-Pattern Anwendung findet. Beachten Sie hierfür auch den Dateiein- und -ausgabe Teil des **CheatSheets** am Ende der Klausur.

z.B BufferedOutputStream vs OutputStream Buffered kann bei großen Daten Mengen schneller abgearbeitet werden weil vorgepuffert werden kann. Rohdaten auf die an beliebigen Stellen zugegriffen werden soll ist man mit Output/InputStream besser bedient da man hier random access hat. alle outputStreams erben von OS und nehmen einen OS in einem der Konstruktoren. Somit kann man ganz leicht zusätzliche Funktionalität an einen Stream anbauen.

- 1P buff vs unbuf
- 1P decorator

Aufgabe 4 - XML (2 + 8 = 10 Punkte)

Betrachten Sie das folgende XML-Dokument:

```
1  <xs:schema attributeFormDefault="unqualified"
2   elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
3    <xs:element name="Horse">
4      <xs:complexType>
5        <xs:simpleContent>
6          <xs:extension base="xs:string">
7            <xs:attribute type="xs:byte" name="maxSpeed" use="optional"/>
8          </xs:extension>
9        </xs:simpleContent>
10     </xs:complexType>
11   </xs:element>
12   <xs:element name="Location" type="xs:string"/>
13   </xs:element name=Stable>
14     <xs:complexType>
15       <xs:sequence>
16         <xs:element ref="Horse" maxOccurs="unbounded" minOccurs="0"/>
17       </xs:Sequence>
18       <xs:attribute type="xs:byte" name=id/>
19     </xs:complexType>
20   </xs:element>
21   <xs:element name="Steadings">
22     <xs:complexType mixed="true">
23       <xs:sequence>
24         <xs:element ref="Location" minOccurs="1" maxOccurs="1">
25         <xs:element ref="Stable" minOccurs="0"/>
26       </xs:sequence>
27     </xs:complexType>
28   </xs:element>
29 </xs:schema>
```

a) Das XML Dokument ist nicht wohlgeformt. Geben Sie die Zeilennummern von **vier** der **fünf** vorkommenden Fehler an und erklären Sie wie diese korrigiert werden können. Konformität mit dem Schema für XSDs muss dabei nicht berücksichtigt werden.

- *jeweils 0.5P*
- *Z13 schließender Tag sollte öffnender sein*
- *Z13 keine Anführungszeichen*
- *Z17 Groß-/Kleinschreibung mit Z15*
- *Z18 fehlende Anführungszeichen*
- *Z24 fehlender schließender Tag*

b) Die obige XML-Datei beschreibt ein XML-Schema. Schreiben Sie eine DTD-Datei so, dass diese Dokumente von selber Struktur validiert. Markieren Sie die Stellen an denen dies nicht möglich ist.

```
<!ELEMENT Steading (Location,Stable*)>
<!ELEMENT Location (#PCDATA)>
<!ELEMENT Stable (Horse*)>
<!ELEMENT Horse>

<!ATTLIST Stable id CDATA #REQUIRED> <-- typ kann nicht spezifiziert werden
<!ATTLIST Horse maxSpeed CDATA #IMPLIED> <-- ditto
```

- 0.5P Steading mit Contents
- 1P korrekter Inhalte von Steading
- 1P Location
- 1P Stable mit korrektem Inhalt
- 1P Horse
- 0.5P Stable id Attribute
- 0.5P Stable id REQUIRED
- 0.5P Horse speed Attribute
- 0.5P Horse speed IMPLIED
- 1.5P erkannt, dass typen nicht gleich spezifiziert werden können

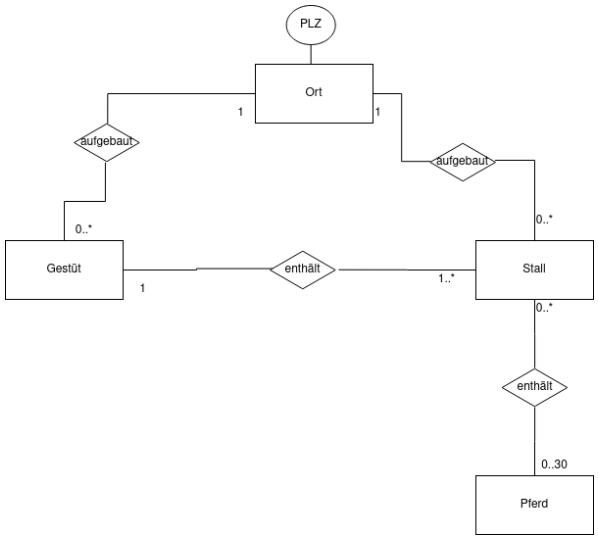
Aufgabe 5 - ER-Modellierung (1.5 + 6 + 3.5 = 11 Punkte)

Ein Gestüt besteht aus einer Menge an Pferdeställen aber mindestens einem. Sowohl Pferdeställe als auch Gestüte befinden sich an einem bestimmten Ort, welcher eine PLZ hat. Pferde können nur genau einem Gestüt aber mehreren Ställen auf diesem Gestüt zugehören. Ein Stall kann maximal 30 Pferde halten.

a) Gibt es eine Anforderung aus dem obigen Sachverhalt die nicht im **E-R-Diagramm** oder dem **relationalen Datenbankschema** modeliert werden kann. Wenn ja, wie könnte diese statdessen modeliert werden?

Nein

b) Modellieren Sie den beschriebenen Sachverhalt als **E-R-Diagramm**.



- 0.5P pro Entity (mit Attributen)
- 1P pro Beziehung (0.5P falls kardinalitäten falsch)
- WICHTIG Notation der Kardinalitäten ist egal SOLANGE sie einheitlich ist!

c) Erstellen Sie zu dem E-R-Diagramm das dazugehörige **relationale Datenbankschema**. Achten Sie hierbei auf sinnvoll gewählte **Primärschlüssel** und stellen Sie sicher, dass das Schema mindestens in 3. Normalform ist. Sie können bei Bedarf auch IDs einführen, um die Eindeutigkeit eines Primärschlüssels zu gewährleisten.

Gestüt(GID, *PLZ*)
Stall(SID, *PLZ*)
SP(*SID*,*PID*)
Pferd(PID)
Ort(PLZ)

- 0.5P pro korrektes Schema Gestüt, Stall, Pferd, Ort
- 0.5P für FK Beziehung von Gestüt und Stall zu Ort
- 1P korrektes Schema SP (0.5P falls PK/FK Beziehung zu C/CP falsch)

Aufgabe 6 - SQL (1.5 + 3.5 + 4 + 4 = 13 Punkte)

Gegeben seien die folgenden Relationenschemata (auch zu finden auf dem beiliegenden Extrablatt):

Schiffe		
SID	Name	**MID**

Rennen			
RID	RennName	Startzeit	Preisgeld

TeilnehmendeSchiffe	
SID	**RID**

Matrosen	
MID	MName

Besatzung	
MID	**SID**

Primärschlüssel sind unterstrichen, Fremdschlüssel sind **fett** dargestellt. Formulieren Sie, insofern nicht anders spezifiziert, die **SQL-Statements** zur Lösung folgender Teilaufgaben:

a) Fügen Sie den **neuen** Matrosen 'Matrose Stefan' mit der ID '42' in die Matrosentabelle ein.

```
INSERT INTO Matrosen (MID, MName)
VALUES ('42','Matrose Stefan')
```

- 0.5P INSERT
- 1P VALUES

b) Geben Sie an, wie viele Matrosen existieren, deren Namen an 2. Stelle ein **e** enthält.

```
SELECT COUNT(*)
FROM Matrosen
WHERE MName LIKE '_e%'
```

- 1P COUNT
- 0.5P FROM
- 2P WHERE (0.5P davon ist die %, 0.5P davon ist der _ und weitere 0.5P das LIKE statt =)

c) Geben Sie die Namen aller Matrosen aus, die auf mehr als 3 Schiffen arbeiten. Besatzungen werden mittels der Tabelle Besatzungen die zwischen Schiffe und Matrosen Tabelle gestellt ist definiert.

```
SELECT MName
FROM Matrosen NATURAL JOIN Besatzung
GROUP BY MName
HAVING COUNT(*) > 3
```

- 0.5P SELECT
- 1P FROM
- 1P GROUP BY
- 1.5P HAVING (0.5P falls Bedingung falsch)

d) Geben Sie die Namen aller der Schiffe aus, deren Besitzer Teil der Besatzung der 'Titanic III' ist. *Hiweis: Besitzer sind als MID in den Schiffen vermerkt!*

```
SELECT Name
FROM Schiffe AS S
WHERE S.MID NOT IN (
  SELECT MID
  FROM Schiffe AS SS JOIN Besatzung AS B ON SS.SID = B.SID
  WHERE SS.Name = 'Titanic III'
)
```

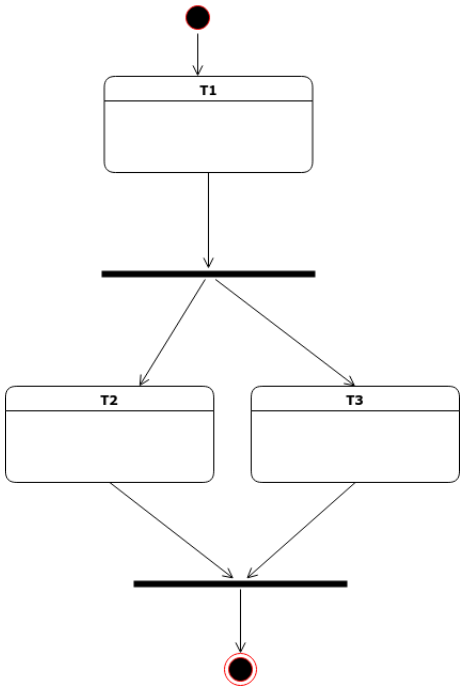
- 0.5P SELECT
- 0.5P FROM
- 2P Subquery (0.5P SELECT; 1P FROM!KEIN NJOIN MÖGLICH; 0.5P WHERE)
- 1P WHERE NOT IN

Aufgabe 7 - Threads (2 + 6 + 3 = 11 Punkte)

a) Betrachten Sie den folgenden Code:

```
1   public class StateManager {
2       public static void main(String... args) throws Exception{
3           Thread t1  = new S1();
4           Thread t2  = new S2();
5           Thread t3  = new S3();
6           t1.start();
7           t1.join();
8           t2.start();
9           t3.start();
10          t3.join();
11          t2.join();
12          System.out.println("end");
13      }
14  }
```

Ergänzen Sie den untenstehenden UML-Zustandsautomat mit sinnvollen Zustandsübergängen, Start- und Endmarkierungen so wie gegebenenfalls forks und joins um den im oben abgebildeten Code dargestellten Sachverhalt abzubilden.



- 0.5 fork nach T1
- 0.5 join von T2 und T3
- 0.5P start
- 0.5P end

b) Erläutern Sie anhand eines selbst gewählten Code-Beispiels wie **Deadlocks** entstehen. Gehen Sie in Ihrer Erklärung auch darauf ein warum die Entstehung eines Deadlocks nicht zwingend deterministisch ist.

Bsp.:

```
1   public class DeadLock extends Thread {
2
3       public static void main(String[] args) throws Exception {
4           Object l1 = new Object();
5           Object l2 = new Object();
6           // l1 und l2 werden jeweils von einem Thread geblockt
7           // und nicht wieder freigegeben da dann jeweils auf das andere lock
8           // gewartet wird
9           Thread t1 = new DeadLock(l1, l2);
10          Thread t2 = new DeadLock(l2, l1);
11          t1.start();
12          t2.start();
13      }
14
15      private Object lock1;
16      private Object lock2;
17
18      public DeadLock(Object lock1, Object lock2) {
19          this.lock1 = lock1;
20          this.lock2 = lock2;
21      }
22
23      public void run() {
24          synchronized(lock1) {
25              System.out.println("First lock acquired!");
26              synchronized(lock2) {
27                  System.out.println("Second lock acquired!");
28              }
29          }
30          System.out.println("Done! All locks released!");
31      }
32  }
```

3P Bsp. das Deadlock erzeugt. 2 Threads benötigen die selben Ressourcen aber akquirieren diese in verschiedener Reihenfolge . Hierbei kann es dann sein dass jeder eine der Ressourcen bekommt und sich die Threads dann gegenseitig die jeweils 2. Ressource blockieren (1.5P). Da dies von der Geschwindigkeit in welcher die beiden Threads die Ressourcen akquirieren abhängt ist der Deadlock nicht deterministisch (1.5P)

c) Erläutern Sie wie mittels einer `BlockingQueue` mehrere Threads gemeinsam an einer Aufgabe arbeiten können. Erklären Sie zusätzlich wie dies mittels `Futures` ebenfalls erreicht werden kann. Gehen Sie in ihren Erklärungen darauf ein wie `BlockingQueue` und `Futures` grundsätzlich funktionieren und wie sie sich unterscheiden.

BQ: Producer kann solange Elemente einstellen bis upper limit erreicht ist. Consumer kann Threadsafe Elemente auslesen und weiterverarbeiten (1P). Futures sind Objekte die ein Ergebnis halten, welches irgendwann berechnet wird und solange blocken. Diese können anderen Threads gegeben werden die mit ihren 'Berechnungen' warten bis im Future wirklich ein wert steht (1P). Unterschiede: BQ kann immer neue Werte enthalten die gelesen werden können während Futures einmal ein Ergebnis liefern. Somit eignen die zwei Konzepte sich für verschiedene Asynchrone Aufgaben (1P).