# Prüfung 1 August 2009, Fragen und Antworten

Einführung in die Softwaretechnik (IN0006) (Technische Universität München)

Technische Universität München

Fakultät für Informatik
Lehrstuhl für Informatik 19

# Musterlösung: Klausur zur Veranstaltung „Einführung in die Softwaretechnik“

Lehrstuhl für Informatik 19 (sebis), 01. August 2009, Sommersemester 2009

| | |
|---|---|
| Nachname: | |
| Vorname: | |
| Matrikelnummer: | |
| Studiengang: | |

**Wichtige Hinweise**

- Füllen Sie auf diesem Blatt gut lesbar die obigen Felder (Nachname, Vorname etc.) aus.
- Notieren Sie auf jeder Seite des Arbeitspapiers gut lesbar Ihren Namen und die laufende Seitennummer!
- Es stehen 120 Minuten Bearbeitungszeit zur Verfügung. Maximal zu erringen sind 72 Punkte.
- Für korrekte Lösungsansätze werden auch dann Punkte vergeben, wenn das Endergebnis fehlerhaft ist oder fehlt. Erläutern Sie daher Ihre Lösungswege möglichst genau!
- Bei einem Täuschungsversuch wird die Klausur mit 5,0 bewertet.

**Erlaubte Hilfsmittel**

- Ein handbeschriebenes DIN A4 Blatt.

# Aufgaben

## Teil 1: Wissensfragen

1. **Erläutern Sie, was sich hinter dem Prinzip *DRY – Don´t Repeat Yourself* verbirgt und nennen sowie erläutern Sie zwei Arten von *Dopplung*. (4 Punkte)**

   DRY besagt, dass ein Sachverhalt in einer Software an genau einer Stelle steht, d.h. dasselbe Problem nicht redundant gelöst wird. Mögliche Arten von Dopplung sind:

   - Erzwungene Dopplung: Ein Sachverhalt muss mehrfach ausprogrammiert werden, da z.B. verschiedene Programmiersprachen für Client und Server zum Einsatz kommen.
   - Unabsichtliche Dopplung: Ein Sachverhalt wird mehrfach ausprogrammiert, da dem Entwickler nicht bewusst ist, dass er denselben Sachverhalt bereits implementiert hat.
   - Ungeduldige Dopplung: Ein Stück Code wird durch Kopieren „wiederverwendet"; aus Eile verzichtet der Entwickler auf eine strukturell saubere Lösung.
   - Dopplung durch mehrere Entwickler: Ein Sachverhalt wird durch verschiedene Entwickler mehrfach (unabhängig voneinander) ausprogrammiert, da sie von den Implementierungen des jeweils anderen nichts wissen.

| | Punkte |
|---|---|
| Durchschnitt | 2,9 |
| Maximal | 4 |
| Varianz | 1,8 |

2. **Erläutern Sie jeweils an einem konkreten Beispiel die Begriffe *Fehlerwirkung*, *Fehlerzustand* und *Fehlerhandlung*. (4,5 Punkte)**

   - Fehlerwirkung: beobachtbares, falsches Verhalten des Programms während seiner Ausführung, abweichend von der Spezifikation, z.B. Buchungsfunktion für Reisen druckt keine Buchungsbestätigung, obwohl dieser in der Spezifikation gefordert wird
   - Fehlerzustand: innerer Fehler eines Programms – Ursache für eine Fehlerwirkung, z.B. fehlerhafte Anweisung im Programm
   - Fehlerhandlung: Handlung, die zu einem Fehlerzustand führt, z.B. Fehlerhafte Implementierung durch einen Programmierer

| | Punkte |
|---|---|
| Durchschnitt | 0,9 |
| Maximal | 4,5 |
| Varianz | 1,5 |

**3. Erläutern Sie, was man im Requirements Engineering unter einem *Stakeholder* versteht. Nennen Sie drei Stakeholder für eine Auktionsplattform, wie z.B. eBay, und skizzieren Sie für jeden Stakeholder eines seiner Ziele. (5,5 Punkte)**

Stakeholder sind Personen, die funktionale oder nicht-funktionale Interessen an einem System oder dessen Entwicklung haben. Mögliche Stakeholder einer Auktionsplattform sind:

- Unternehmensleitung: Interessiert an geringen Kosten der Softwareentwicklung
- Käufer: Interessiert an der sicheren Abwicklung seiner Auktionen und Finanztransaktionen
- Verkäufer: Interessiert an der hohen Verfügbarkeit der Auktionssoftware

| | Punkte |
|---|---|
| Durchschnitt | 4,2 |
| Maximal | 5,5 |
| Varianz | 2,0 |

**4. Erläutern Sie die Idee von *CRC-Cards*, nennen Sie je eine *Stärke* und eine *Schwäche*, die mit dieser Methode einhergehen. (4 Punkte)**

Eine CRC-Card beschreibt eine Klasse im objektorientierten Entwurf sowie deren Aufgaben (*Responsibilities*) und Interaktionspartner (*Collaboration*) auf der stark beschränkten Fläche einer Karteikarte.

Mögliche Stärken von CRC-Cards sind:

- Blick auf die Verantwortlichkeiten nicht den Datengehalt von Klassen
- Benennung von Verantwortlichkeiten
- Nutzung als Diskussionsgrundlage im Entwicklerteam
- Reduktion auf die wichtigsten Verantwortlichkeiten

Mögliche Schwächen von CRC-Cards sind:

- Keine Unterstützung durch gegenwärtige (UML)-Standards
- Hoher Zeitaufwand
- Ergebnis liegt meist nicht in digitaler Form vor

| | Punkte |
|---|---|
| Durchschnitt | 3,1 |
| Maximal | 4 |
| Varianz | 1,6 |

**5. Nennen Sie drei konkrete *Software-Metriken* und diskutieren Sie die Aussagekraft einer dieser Metriken. (4,5 Punkte)**

- Depth-Of-Inheritance Tree: Hoher Wert bedeutet langen Vererbungsbaum zur Wurzelklasse. Ein Klasse mit großer DIT ist möglicherweise nur schwer zu warten. Viele Klassen mit einem DIT von 0 deuten auf geringe Wiederverwendung hin.
- Number Of Children: Hoher Wert bedeutet hohe Wichtigkeit der Klasse, da viele Klassen davon erben. Große NOC deutet auf schlecht gewählte Abstraktionshierarchie hin.
- Coupling Between Objects: Hohes CBO deutet auf schlechte Modularisierung und dadurch erhöhten Wartungsaufwand hin.
- Lack Of Cohesion: Großes LCOM deutet auf schlechte Kapselung hin, da funktional nicht verwandte Aufgaben innerhalb einer Klasse erledigt werden.
- Lines Of Code: Viele LOC deuten auf ein Projekt großen Umfangs hin. Die LOC können ex-post zur Abschätzung der Produktivität von Entwicklerteams verwendet werden.
- McCabes cyclomatic Complexity: Hoher Wert bedeutet hohe Anzahl verschiedener Pfade durch eine Methode. Eine derartige Methode kann kompliziert und schwer zu verstehen sein.
- Verhältnis statische Methoden zu Gesamtzahl der Methoden: Ein hohes Verhältnis deutet darauf hin, dass der Code wenig objektorientiert sondern hauptsächlich prozedural implementiert ist.

| | Punkte |
|---|---|
| Durchschnitt | 3,1 |
| Maximal | 4,5 |
| Varianz | 2,6 |

**6. Nennen und erläutern Sie drei wesentliche in der Vorlesung vorgestellte *Ziele* des *Architekturentwurfs*. (4,5 Punkte)**

- Unterstützung der effizienten Entwicklung durch Verteilung abgegrenzter Aufgaben auf Teams
- Minimierung von Projektrisiken durch Schaffung von Transparenz im Hinblick auf mögliche Risiken
- Schaffung eines gemeinsamen Grundverständnisses für das System im Entwicklerteam
- Dokumentation von Kernwissen der Problemdomäne in aggregierter Form
- Möglichmachen der Verteilung des Systems auf verschiedene Hardwarekomponenten
- Weiterentwicklung des Systems durch Anpassung oder Austausch einzelner Komponenten

| | Punkte |
|---|---|
| Durchschnitt | 2,3 |
| Maximal | 4,5 |
| Varianz | 2,4 |

**7. Erläutern Sie kurz die Begriffe *Refactoring* und *Regressionstest*, sowie die Bedeutung von Regressionstests für das Refactoring. (3 Punkte)**

Refactoring stellt eine Änderung an der internen Struktur einer Software dar, um diese leichter verständlich und besser änderbar zu machen. Dabei wird jedoch das beobachtbare Verhalten der Software nicht verändert.

Ein Regressionstest wiederholt Testfälle, welche die Anwendung bereits erfolgreich absolviert hat, um Nebenwirkungen von Modifikationen in bereits getesteten Teilen der Software aufzuspüren.

Beim Refactoring sollten Regressionstests durchgeführt werden, um sicher zu stellen, dass durch die Verbesserung der Lesbarkeit am Verhalten des Programms kein Schaden entsteht.

| | Punkte |
|---|---|
| Durchschnitt | 1,6 |
| Maximal | 3,0 |
| Varianz | 1,0 |

## Teil 2: Verständnis- und Modellierungsfragen

### 1. Reverse-Engineering

Gegeben ist folgendes Code-Fragment (*Quelltext 1*):

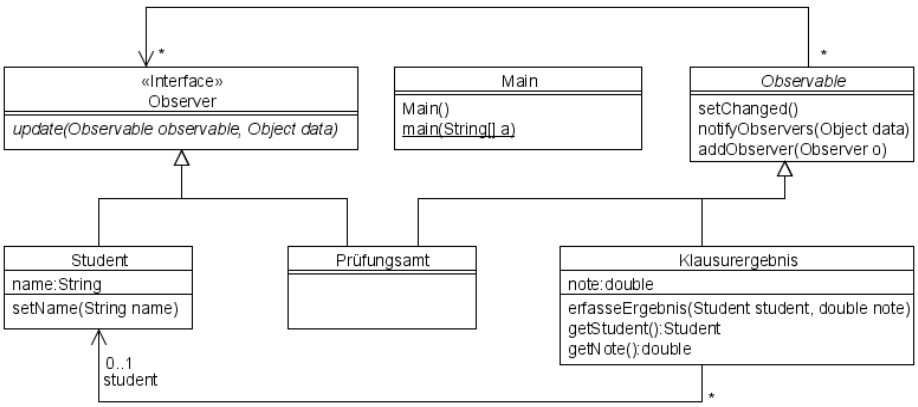
```
1  public class Main {
2          public static void main(String[] a) {
3                  new Main();
4          }
5          public Main() {
6                  Prüfungsamt prAmt = new Prüfungsamt();
7                  Student strebi = new Student();
8                  strebi.setName("Strebi");
9                  prAmt.addObserver(strebi);
10                 Klausurergebnis klausurStrebi = new Klausurergebnis();
11                 klausurStrebi.addObserver(prAmt);
12                 klausurStrebi.erfasseErgebnis(strebi, 1.0);
13         }
14 }
15 public class Klausurergebnis extends Observable {
16         private Student student;
17         private double note;
18         public void erfasseErgebnis(Student student, double note) {
19                 this.student = student;
20                 this.note = note;
21                 setChanged();
22                 notifyObservers(this);
23         }
24         public Student getStudent() {
25                 return student;
26         }
27         public double getNote() {
28                 return note;
29         }
30 }
31 public class Prüfungsamt extends Observable implements Observer {
32         public void update(Observable observable, Object data) {
33                 setChanged();
34                 notifyObservers(observable);
35         }
36 }
37
38 public class Student implements Observer {
39         private String name;
```

```
40          public setName(String name) {
41                  this.name = name;
42          }
43          public void update(Observable observable, Object data) {
44                  if (observable instanceof Prüfungsamt
45                                  && data instanceof Klausurergebnis) {
46                          Klausurergebnis ergebnis = (Klausurergebnis) data;
47                          if (ergebnis.getStudent().equals(this)) {
48                                  System.out.println("Klausurergebnisse
49                                          erhalten! Name: " + this.name +
50                                          " Note: " + ergebnis.getNote());
51                          } else {
52                                  System.out.println("Schade, leider nicht
53                                          mein Klausurergebnis");
54                          }
55                  }
56          }
57  }
```

**A) Erstellen Sie zu Quelltext 1 ein implementierungsnahes Klassendiagramm, das alle im Code enthaltenen Klassen, Attribute, Konstruktoren und Methoden sowie die Assoziationen mit entsprechenden Rollennamen beinhaltet. Beachten Sie dabei folgendes:**

- **`implements` und `extends`, sowie Methoden aus Superklassen und Interfaces, soweit im Quelltext verwendet oder implementiert, müssen modelliert werden.**
- **Sichtbarkeiten müssen nicht modelliert werden.**

**(8 Punkte)**




Häufige Fehler:

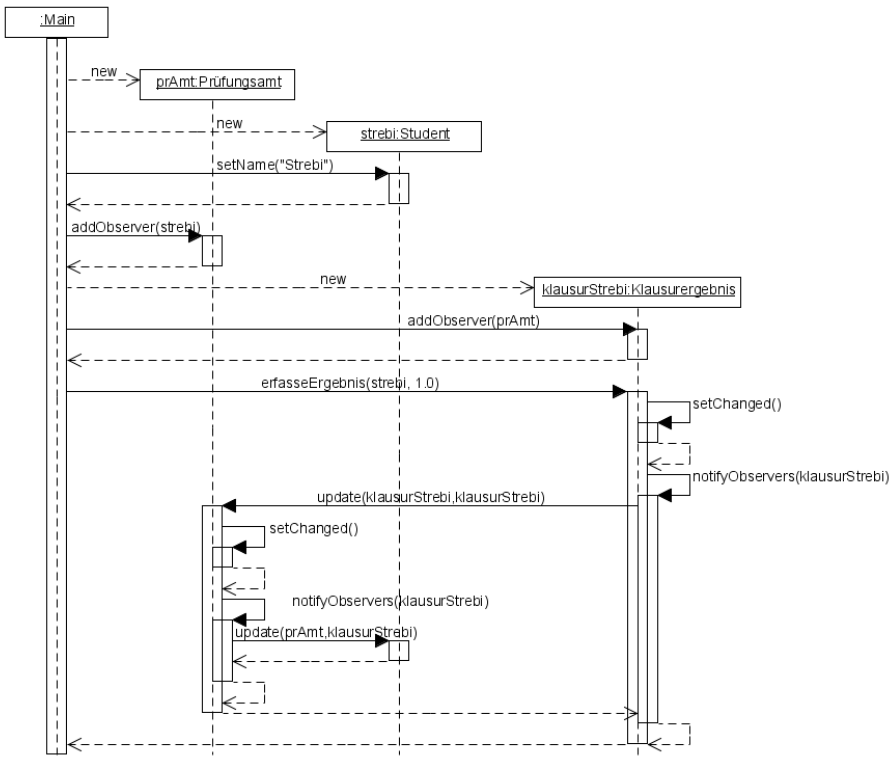
- Klassen *Main* und *Observer* sowie das Interface *Observable* wurden vergessen

- Falsche Notation für die Vererbungsbeziehungen zwischen Klassen, z.B. durch Aggregations- und Kompositionssymbole
- Fehlende Attribute
- Fehlende oder unvollständige Signatur der Methoden (fehlende Parameter oder fehlender Rückgabetyp)

**B) Erstellen Sie ein Sequenzdiagramm, das die Interaktionen der Objekte ausgehend vom Aufruf des Konstruktors `Main()` beschreibt. Beachten Sie dabei,**

- ❖ **dass Aufrufe von `get`- und `equals`-Methoden sowie Aufrufe zur Konsolenausgabe `System.out.println(…)` nicht berücksichtigt werden sollen,**
- ❖ **dass Aufrufe geerbter Methoden berücksichtigt werden müssen.**

**(14 Punkte)**




Häufige Fehler:

- Kein UML Sequenzdiagramm
- Fehlende Aufrufe von Methoden oder Aufruf falscher (nicht existierender) Methoden
- Fehlende Objekte und Lebenslinien oder nicht vorhandene Objekte
- Fehlende Signatur bei Methodenaufrufen
- Fehlende Aktivitätsbereiche nach einem Methodenaufruf

| | Punkte |
|---|---|
| Durchschnitt | 9,9 |
| Maximal | 21,5 |
| Varianz | 33,3 |

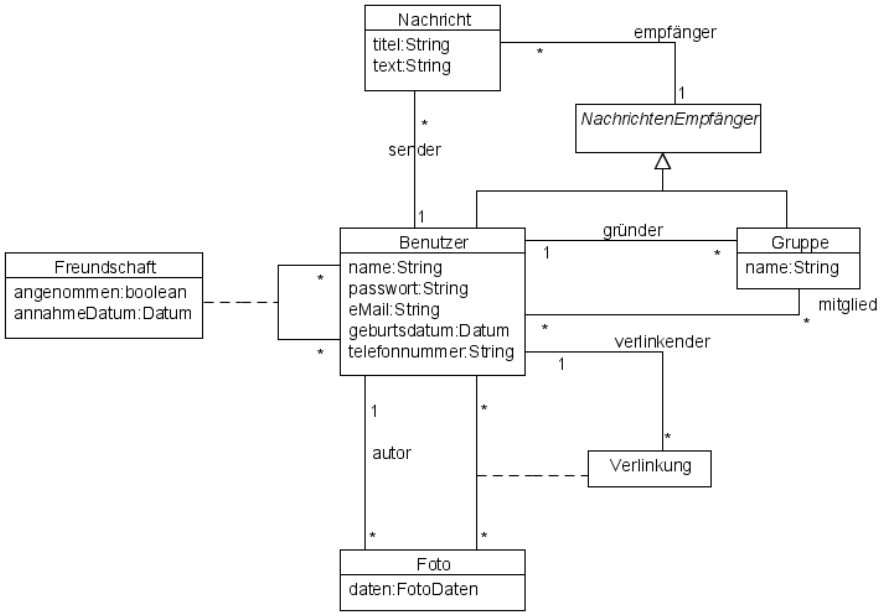
## 2. Konzeptuelle Modellierung (20 Punkte)

**Erstellen Sie ein konzeptuelles Klassendiagramm für die Social-Network Plattform *sebisFriends*. Modellieren Sie dabei die Konzepte aus folgender Beschreibung:**

Auf der Plattform *sebisFriends* können sich Benutzer registrieren sowie unter Angabe ihres Benutzernamens und ihres Passworts anmelden. Im Benutzerprofil werden als weitere Informationen die eMail-Adresse des Benutzers, sein Geburtsdatum und seine Telefonnummer hinterlegt. Benutzer können Gruppen gründen und bei der Gründung mit einem Namen versehen sowie bestehenden Gruppen beitreten; die Informationen über die Gruppenzugehörigkeit sowie den Gründer einer Gruppe werden von der Plattform gespeichert. Dasselbe gilt für Nachrichten; ein Benutzer kann Nachrichten bestehend aus Titel und Nachrichtentext an einen anderen Benutzer oder eine Nutzergruppe versenden.

Ein Benutzer kann einem anderen Benutzer von *sebisFriends* ein Freundschaftsangebot unterbreiten. Durch die Annahme dieses Freundschaftsangebots entsteht eine Freundschaft zwischen den Benutzern. Die Plattform speichert dabei auch das Datum, an dem die Freundschaft geschlossen wurde.

Ein Benutzer kann mehrere Fotos in der Plattform *sebisFriends* hochladen. Die Plattform speichert dabei die Binärdaten des Fotos sowie den hochladenden Benutzer als den Autor des Bildes. Es steht allen Benutzern von *sebisFriends* frei sich selbst oder andere Benutzer auf den Fotos zu verlinken. Dabei kann ein Benutzer höchstens einmal auf einem Bild verlinkt, d.h. auf diesem Foto „entdeckt“ werden. Die Plattform speichert ab, welcher Benutzer auf einem Foto verlinkt wurde und darüber hinaus, wer (welcher Benutzer) diese Verlinkung angelegt hat.




Häufige Fehler:

- Doppelung der Assoziationen durch Attribute oder Umsetzung einer Assoziation durch ein Attribut, z.B. Gründer einer Gruppe

- Fehlende Attribute, fehlende Datentypen bei den Attributen oder falsche Datentypen, speziell bei der *Telefonnummer*; diese ist eine Zeichenkette und keine Zahl, da führende Nullen vorkommen können
- Fehlende oder falsche Multiplizitäten bei den Assoziationen, speziell bei der Assoziation *mitglied* zwischen Gruppe und Benutzer
- Modellierung einer Klasse *sebisFriends* oder *Plattform*; diese ist nicht notwendig, da nur eine Freundschaftsplattform modelliert werden soll (keine multimandantenfähige Lösung)
- Fehlende Klasse oder Assoziationsklasse um eine *Verlinkung* darzustellen. Diese dreiwertige Beziehung á la *ein-Benutzer-verlinkt-einen-anderen-Benutzer-auf-einem-Foto* ist anders nicht darstellbar.
- Navigierbarkeiten von Assoziationen sind im konzeptuellen Klassendiagramm nicht notwendig, sondern sogar falsch.
- Vielfältige andere Diagrammtypen, z.B. Objektdiagramme, oder Notationen, welche in der UML nicht angeboten werden

| | Punkte |
|---|---|
| Durchschnitt | 11,3 |
| Maximal | 20,0 |
| Varianz | 18,8 |