



Technische Universität Dresden, 01062 Dresden




Klausur Softwaretechnologie WS 2018/19

Prof. Dr.rer.nat.habil.
Uwe Aßmann

Name:	
Vorname:	
Immatrikulationsnummer:	

Aufgabe	Maximale Punktzahl	Erreichte Punktzahl
1	24	
2	20	
3	46	
Gesamt	90	

Hinweise:

- In der Klausur ist als Hilfsmittel lediglich ein **A4-Blatt, beidseitig beschrieben**, zugelassen.
- Die Klammerung der Aufgabenblätter darf **nicht** entfernt werden.
- Tragen Sie bitte die Lösungen auf den Aufgabenblättern ein!
- Verwenden Sie keine roten, grünen Stifte oder Bleistifte!
- Es ist kein eigenes Papier zu verwenden! Bei Bedarf ist zusätzliches Papier bei der Aufsicht erhältlich. Bitte jedes zusätzliche Blatt mit Name, Vorname und Immatrikulationsnummer beschriften.
- Es sind alle Aufgabenblätter abzugeben!
- Ergänzen Sie das Deckblatt mit Name, Vorname und Immatrikulationsnummer!
- Halten Sie Ihren Studentenausweis und einen Lichtbildausweis zur Identitätsprüfung bereit.
- **Achtung!** Das Zeichen  heißt: **Hier ist Java-Text einzufügen!**

Aufgabe 1: Akademisches ID-Management (24 Punkte)

Die Reputation von Wissenschaftlern hängt u.a. von der Anzahl und der Wertigkeit ihrer Publikationen sowie davon ab, wie häufig ihre Publikationen von anderen Wissenschaftlern referenziert werden. Dazu ist in den letzten Jahren eine Reihe von Portalen im Web entstanden, die wissenschaftliche Arbeiten und ihre Autoren verwalten. Diese Anwendungsdomäne soll modelliert werden.

Ein Portal (*Portal*) ist durch seinen Namen (*Name*) und den Anbieter (*Provider*) gekennzeichnet. Jedes Portal verwaltet beliebig viele Autoren (*Author*). Oftmals bekommt jeder Autor vom Anbieter eine proprietäre Identifikationsnummer (*ID*). Zusätzlich haben sich verschiedene Anbieter entschieden, den Autor – sofern vorhanden – durch die sogenannte ORCID-ID (*ORCID*) zu kennzeichnen. Die ORCID-ID ist ein nicht-proprietärer, überwiegend numerischer Code zur weltweit eindeutigen Identifizierung eines wissenschaftlichen Autors. Jeder Autor wird des Weiteren durch seinen Namen (*Name*) und seine Institution (*Institution*) beschrieben. Jede Publikation (*Publication*) hat mindestens einen Autor und wird ihren Autoren zugeordnet (*myAuthors*). Umgekehrt gibt es zu jedem Autor, sofern vorhanden, die Liste seiner Publikationen (*myPublications*).

Jede Publikation wird durch ihren Titel (*Title*), das Erscheinungsjahr (*Year*) und sofern vorhanden eine Zusammenfassung (*Abstract*), den Digital Object Identifier im Web (*DOI*) sowie die Kennzeichnung als Open Access (*openAccess*) beschrieben. Zusätzlich wird für jede Publikation der Veröffentlichungstyp (*PublicationType*, *myPublicationType*) angegeben. Dieser ist entweder ein Artikel (*ARTICLE*), ein Buch (*BOOK*), ein Kapitel in einem Buch (*CHAPTER*), ein Konferenzartikel (*CONF PAPER*) oder ein Forschungsbericht (*RESEARCHREPORT*).

Außerdem gibt es für jede Publikation einen Herausgeber (*Publisher*, *myPublisher*) und möglicherweise eine beliebige Anzahl von Schlüsselwörtern (*Keyword*, *myKeywords*). Schlüsselwörter können in verschiedenen Publikationen auftauchen. Ein Schlüsselwort wird durch seinen Namen gekennzeichnet (*Name*).

Der Herausgeber hat einen Namen (*Name*) und ggfs. einen Score (*Score*), der seine Reputation in der Wissenschaftswelt beschreibt.

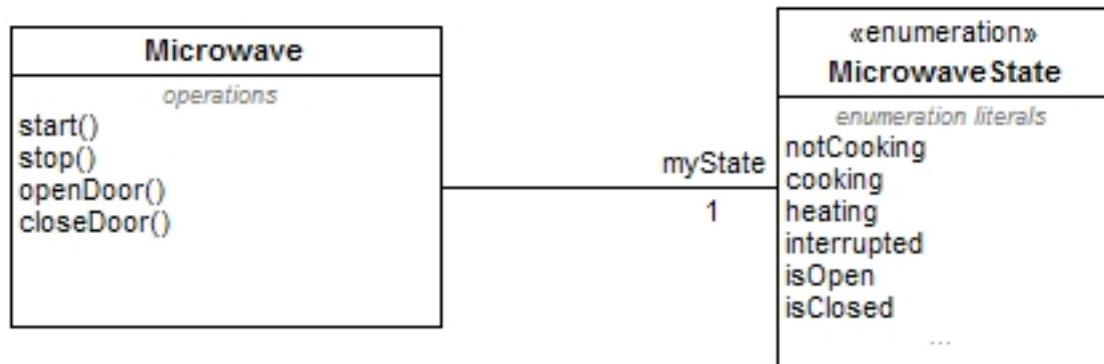
Wichtig ist für eine Publikation zusätzlich, welche anderen Publikationen eine Publikation referenziert (*references*) und von welchen Publikationen die Publikation ggfs. zitiert wird (*referencedBy*). Aus diesen Informationen und dem Score des Herausgebers einer Publikation wird der sogenannte h-Index (*hIndex*) eines Autors, d.h. eine Kennzahl für das weltweite Ansehen eines Wissenschaftlers in Fachkreisen, abgeleitet.

Erstellen Sie ein Domänenmodell (UML-Analyse-Klassendiagramm) für das akademische ID-Management! Berücksichtigen Sie dabei folgende Hinweise:

- **Nutzen Sie die Modellierungskonventionen, die in den Übungen verwendet wurden!**
- **Beschreiben Sie die Klassenbeziehungen möglichst genau mit Multiplizitäten und Rollennamen!**
- **Denken Sie auch daran, dass Attribute Multiplizitäten haben können und dass sie ggfs. abgeleitet sind.**
- **Alle domänenspezifischen Begriffe, die im obigen Text *kursiv* geschrieben sind, sollen dabei im Modell wiederzufinden sein! Es sind keine weiteren Namen notwendig!**

Aufgabe 2: Mikrowelle (20 Punkte)

Eine Mikrowelle (*Microwave*) wird aus Nutzersicht wie folgt (vereinfacht) modelliert:



Die Mikrowelle ist entweder im Zustand, dass kein Kochprozess gestartet wurde (*notCooking*) oder im Zustand des Kochens (*cooking*). Der Nutzer kann jederzeit den Button „Start“ (*start()*) und den Button „Stop“ (*stop()*) drücken und die Tür der Mikrowelle öffnen (*openDoor()*) oder schließen (*closeDoor()*). Was dann passiert, hängt davon ab, in welchem Zustand und Unterzustand sich die Mikrowelle befindet.

Im Zustand *notCooking* ist die Tür der Mikrowelle entweder geöffnet (*isOpen*) oder geschlossen (*isClosed*). Im geöffneten Zustand hat der Button *start()* keine Wirkung. Erst wenn die Tür der Mikrowelle geschlossen wurde (*closeDoor()*), beginnt durch das Drücken des Buttons *start()* die Mikrowelle zu arbeiten (Unterzustand *heating*) und die Mikrowelle kommt damit in den Zustand *cooking*.

Während die Mikrowelle arbeitet (Unterzustand *heating*), kann der Kochprozess durch Öffnen der Tür (*openDoor()*) unterbrochen werden (Unterzustand *interrupted*). In diesem Zustand kann die Tür wieder geschlossen (*closeDoor()*) und auch wieder geöffnet (*openDoor()*) werden, ohne dass die Mikrowelle den Unterzustand *interrupted* verlässt. Falls die Tür der Mikrowelle geschlossen ist (*isClosed*) und der Nutzer den Button *start()* drückt, geht der Kochprozess weiter (Unterzustand *heating*). Falls der Nutzer bei geöffneter Tür im unterbrochenen Unterzustand *interrupted* versucht, den Kochprozess fortzusetzen (*start()*), passiert nichts. Der Kochprozess wird beendet, wenn das Signal *timeOver()* ertönt. Die Mikrowelle geht dann in den Zustand *notCooking/isClosed* über.

Falls der Nutzer den Button *stop()* drückt, geht die Mikrowelle, egal in welchem Zustand sie sich befindet, in den Zustand *notCooking*. In Abhängigkeit davon, ob die Tür der Mikrowelle gerade geöffnet oder geschlossen ist, bleibt sie im Unterzustand *isOpen* oder *isClosed*.

Die Initialisierung der Mikrowelle erfolgt durch einen Übergang in den Zustand *notCooking/isOpen*.

Modellieren Sie die beschriebene Mikrowelle als eine Verhaltenszustandsmaschine!
Denken Sie dabei an die Modellierung von Oberzuständen und ggfs. eines History-Zustandes!
Verwenden Sie im Zustandsdiagramm ausschließlich die kursiv geschriebenen Bezeichnungen!

Aufgabe 3: Bankanwendung (46 Punkte)

Die folgende Abbildung zeigt das Klassendiagramm einer Bankanwendung. Die Anwendung verwaltet Banken (`Bank`), welche ihre Konten (`Account`) verwaltet und Transaktionen (`Transaction`) von eigenen Konten zu anderen Konten erlaubt.

Es gibt zwei Arten von Konten. Zum einen gibt es Girokonten (`CheckingAccount`), welche eine Beschränkung (`limit`) auf die Geldmenge festlegen, die mit einem Mal abgehoben werden kann.

Zum anderen gibt es Sparkonten (`SavingsAccount`), welche im Gegensatz zu Girokonten nicht überzogen werden dürfen, d.h. ihr Betrag darf nie negativ werden.

Unabhängig von der Art des Kontos, darf der Betrag, der abgehoben (`decrease()`) oder eingezahlt (`increase()`) wird, nie negativ sein. Für jedes Konto wird der aktuelle Kontostand (`balance`) gespeichert.

Eine Bank erlaubt es, neue Konten anzulegen (`addAccount()`) und ein existierendes Konto anhand seiner Kontonummer (`id`) zu finden (`getAccount()`). Hierzu muss eine Bank ihre Konten (`accounts`) anhand ihrer Kontonummern in einer Datenstruktur speichern, die den Direktzugriff über die Kontonummer erlaubt.

Zusätzlich erlaubt eine Bank, Transaktionen (`Transaction`) von genau einem eigenen Konto zu einem beliebigen anderen Konto anzulegen.

Dabei obliegt es den unterschiedlichen Banken, wann eine Transaktion als Überweisung (engl. `Transferal`) ausgeführt wird (`scheduleTransferal()`). Dies wird durch den Ausführungsplaner (`Scheduler`) einer Bank bestimmt. Hierfür gibt es zwei Implementierungen.

Die erste Variante (`InstantScheduler`) führt jede Transaktion sofort aus.

Die zweite Variante (`BulkScheduler`) sammelt erst fünf Transaktionen, bevor sie alle mit einem Mal, jedoch unter Einhaltung ihrer ursprünglichen Reihenfolge, ausgeführt werden.

Am Ende eines Quartals prüft jede Bank ihre Konten und belastet Konten mit negativem Betrag mit einem Überziehungszins von 10% (`bookOverdraftCharge()`), d.h. wenn ein Konto einen Betrag von -100.0 aufweist, werden ihm weitere 10.0 abgezogen.

Teilaufgabe 3.1

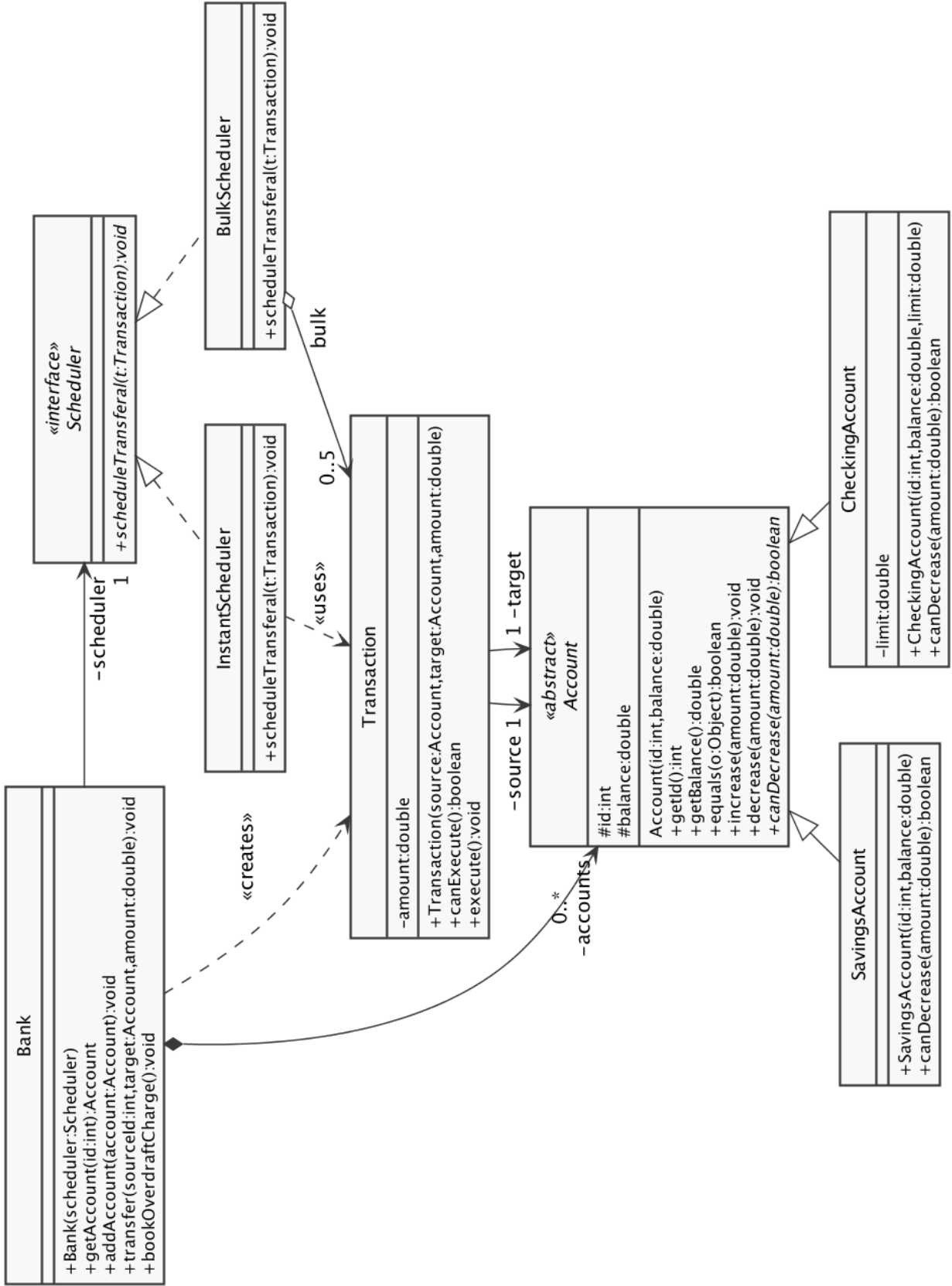
Finden Sie zwei Entwurfsmuster in der Bankanwendung und tragen Sie diese als UML-Kollaboration mit den beteiligten Rollen in das Klassendiagramm (S. 7) ein!

Hinweis für alle Implementierungsaufgaben:

Prüfen Sie, wo notwendig, auf `null`-Werte!

Teilaufgabe 3.2

Die Implementierung der `Bank`-Klasse wurde von Ihrem Chef bereits begonnen und muss von Ihnen vervollständigt werden (S. 8).



```

import java.util.*;

public class Bank {

    private Scheduler scheduler = null;


    public Bank(Scheduler scheduler) {
        this.scheduler = scheduler;
    }

    // Erzeugt eine neue Transaktion von einem eigenen Konto zu einem
    // beliebigen Konto und plant dessen Ausführung ein.
    public void transfer(int sourceId, Account target, double amount) {
        Account source = getAccount(sourceId);
        if (source != null && (!source.equals(target))) {
            Transaction trans = new Transaction(source, target, amount);
            if (trans.canExecute()) {
                scheduler.scheduleTransferal(trans);
            }
        }
    }

    // Sucht in der Bank nach einem Konto mit der gegebenen Kontonummer und
    // gibt dieses zurück. Sonst wird null zurückgegeben.
    public Account getAccount(int id) {



    }

    // Fügt das gegebene Konto der Bank hinzu, falls es noch kein Konto mit
    // derselben Kontonummer gibt.
    public void addAccount(Account account) {



    }

    // Belastet alle Konten mit negativem Betrag mit einem konstanten
    // Überziehungszins von 10%.
    public void bookOverdraftCharge() {



    }
}

```


Teilaufgabe 3.3

Ergänzen Sie für die beiden Klassen Girokonto (**CheckingAccount**) und Sparkonto (**SavingsAccount**) jeweils die unten angegebenen Testfall-Tabellen, welche die Methode **decrease ()** testen.

Testen Sie ausschließlich Grenzfälle der Methodenaufrufe!

Testfalltabelle für decrease() von CheckingAccount

Nummer	Zustand (balance, limit)	Parameter (amount)	Resultat (balance)	Erwarteter Status (ok oder fail)
1				
2				
3				
4				
5				

Testfalltabelle für decrease () von SavingsAccount

Nummer	Zustand (balance)	Parameter (amount)	Resultat (balance)	Erwarteter Status (ok oder fail)
1				
2				
3				
4				

Teilaufgabe 3.4

Implementieren Sie die Klassen Girokonto (CheckingAccount) und Sparkonto (SavingsAccount) auf Basis der folgenden abstrakten Klasse Account.

```

public abstract class Account {
    protected int id;
    protected double balance;

    public Account(int id, double balance) {
        this.id = id;
        this.balance = balance;
    }

    public double getBalance() {
        return balance;
    }

    public int getId() {
        return id;
    }

    public boolean equals(Object a) {
        if (a instanceof Account) {
            return id == ((Account) a).id;
        } else {
            return false;
        }
    }

    // Schreibt den angegebenen Betrag dem Konto gut.
    public void increase(double amount) {
        if (amount >= 0) {
            balance = balance + amount;
        }
    }

    // Hebt den angegebenen Betrag vom Konto ab.
    public void decrease(double amount) {
        if (canDecrease(amount)) {
            balance = balance - amount;
        }
    }

    // Prüft, ob der gegebene Betrag vom konkreten Konto
    // abgehoben werden darf.
    public abstract boolean canDecrease(double amount);
}

```

```
// Implementierung von CheckingAccount und SavingsAccount
```

```
✎
```

Teilaufgabe 3.5

Ihr Chef hat bereits die Transaktion (`Transaction`) wie folgt implementiert. Ihre nächste Aufgabe ist es, den Rest der Bankanwendung zu implementieren, indem Sie die Schnittstelle `Scheduler`, sowie die Klassen `InstantScheduler` und `BulkScheduler` implementieren.

```
public class Transaction {  
    private Account source;  
    private Account target;  
    private double amount;  
  
    public Transaction(Account source, Account target, double amount) {  
        this.source = source;  
        this.target = target;  
        this.amount = amount;  
    }  
}
```

```
public boolean canExecute() {
    return (!source.equals(target)) && amount >= 0.0
        && source.canDecrease(amount);
}

public void execute() {
    if (canExecute()) {
        source.decrease(amount);
        target.increase(amount);
    }
}
}

// Implementierung von Scheduler
ⓧ

// Implementierung von InstantScheduler
ⓧ

// Implementierung von BulkScheduler
ⓧ
```