TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Software- und Multimediatechnik, Professur Softwaretechnologie

Technische Universität Dresden, 01062 Dresden

Prof. Dr.rer.nat.habil.
Uwe Aßmann

Klausur
Softwaretechnologie WS 2017/18

Name:	
Vorname:	
Immatrikulationsnummer:	

Aufgabe	Maximale Punktanzahl	Erreichte Punktanzahl
1	**32**	
2	**26**	
3	**4**	
4	**28**	
Gesamt		

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**!

<u>Aufgabe 1:</u> Scrum Process Management System (SPMS) (32 Punkte)

Für Projekte, denen das Scrum-Vorgehensmodell zugrunde liegt, soll ein entsprechendes Verwaltungssystem (SPMS) entwickelt werden. Für die Erstellung eines SPMS-Repository wurden zunächst alle Artefakte und deren Beziehungen textuell beschrieben.

Erstellen Sie für den nachfolgenden Text ein Domänenmodell (UML-Analyse-Klassen-diagramm)!
Berücksichtigen Sie dabei folgende Regeln:

- **Denken Sie an Klassen, Enumerationen, Klassenbeziehungen einschließlich Vererbung, Assoziationsklassen, Multiplizitäten und ggfs. Rollen- oder Assoziationsnamen!**
- **Es sollen KEINE Attribute und Operationen modelliert werden!**
- **Alle domänenspezifischen Begriffe, die im obigen Text *kursiv* geschrieben sind, müssen im Modell mit genau der angegebenen Bezeichnung wiederzufinden sein! Es sollen keine Klassen modelliert werden, die nicht als kursiv geschriebene Begriffe im Text auftreten.**

Das SPMS verwaltet mehrere Projekte (*Project*). Ein Projekt kann „große“ User Stories (*Epic*) und/oder nur „kleine“ User Stories (*User Story*) enthalten. Ein Epic besteht meistens aus mehreren User Stories. Für jede User Story werden Aufgaben (*Task*) definiert.
Zu jedem Project gibt es genau ein *Backlog*, welches die Anforderungen an das zu entwickelnde Produkt sowie die User Stories auflistet. Das Backlog kann nicht Teil eines anderen Projektes werden und wird gelöscht, sobald das Projekt entfernt wird.
Ein Project wird in beliebig viele Arbeitsschritte (Sprints) eingeteilt. Jeder Sprint wird durch ein *Sprint Backlog* repräsentiert. Eine User Story ist normalerweise genau einem Sprint zugeordnet. Allerdings kann es auch noch User Stories geben, für die noch nicht entschieden worden ist, in welchem Sprint diese realisiert werden sollen.
Wichtig für den Scrum-Prozess sind die verschiedenen Arten von Meetings (*Meeting*). Für die Erstellung des Backlogs wird mindestens ein *Planning Meeting* durchgeführt. Zu Beginn jedes Sprints findet als Kick-Off das *Sprint Planning Meeting* statt. Täglich findet ein *Daily Scrum Meeting* als Status-Meeting statt, welches sich auf den aktuellen Sprint bezieht. Am Ende eines jeden Sprints findet ein *Sprint Retrospective Meeting* statt, in welchem das Team rückblickend Erfahrungen aus dem soeben zu Ende gegangenen Sprint diskutiert.
Für jede User Story wird der aktuelle *Status* gespeichert. Der Status ist entweder *pending*, *todo*, *discussing*, *developing*, *confirming* oder *done*.
Epics, User Stories und Tasks bilden zusammen die vom SPMS verwaltenden Artefakte (*Artefact*).
Neben den Artefakten im Projekt werden alle Nutzer (*User*) und deren Rollen (*Role*) verwaltet. Jeder User kann im Projekt verschiedene Rollen haben. Zugriffsrechte (*Permission*) werden abhängig von einem Nutzer feingranular für die einzelnen Artefakte vergeben. Als Wert (*Right*) für eine Permission kommen *read*, *write*, *create* und *remove* infrage.
Der Aufwand, den ein User für eine User Story hat, wird in Punkten (*Points*) erfasst.

<u>Aufgabe 2:</u> Alarmanlage (26 Punkte)

Gegeben ist der nachfolgende Java-Code für eine Alarmanlage. Da es dazu noch keine Dokumentation gibt, sollen im Nachhinein („revers“) Modelle erstellt werden.

```
public class Alarmanlage {
	private IZustandAlarmanlage aktiv = new AlarmanlageAktiv();
	private IZustandAlarmanlage inaktiv = new AlarmanlageInaktiv();
	private IZustandAlarmanlage zustand;

   public Alarmanlage() {
		zustand = inaktiv;
   }

   public void anschalten() {
		aendereZustand(aktiv);
   }

   public void ausschalten() {
		aendereZustand(inaktiv);
   }

   public void personErkannt() {
		zustand.personErkannt();
   }

   private void aendereZustand (IZustandAlarmanlage neuerZustand) {
		zustand = neuerZustand;
   }
}
```

```
public interface IZustandAlarmanlage {
	public void personErkannt();
}
```

```
public class AlarmanlageAktiv implements IZustandAlarmanlage {
	public void personErkannt() {
		System.out.println ("RING RING");
   }
}
```

```
public class AlarmanlageInaktiv implements IZustandAlarmanlage {
	public void personErkannt() {
		System.out.println ("Ruhig bleiben.");
   }
}
```

```
public class Client
{
   public static void main (String[] args)
   {
      Alarmanlage a = new Alarmanlage();

      a.anschalten();
      a.personErkannt();
      a.ausschalten();
      a.personErkannt();
   }
}
```

Erstellen Sie für diese Alarmanlage folgende drei Modelle:

I. Ein Entwurfsklassendiagramm (ohne die Klasse `Client`)

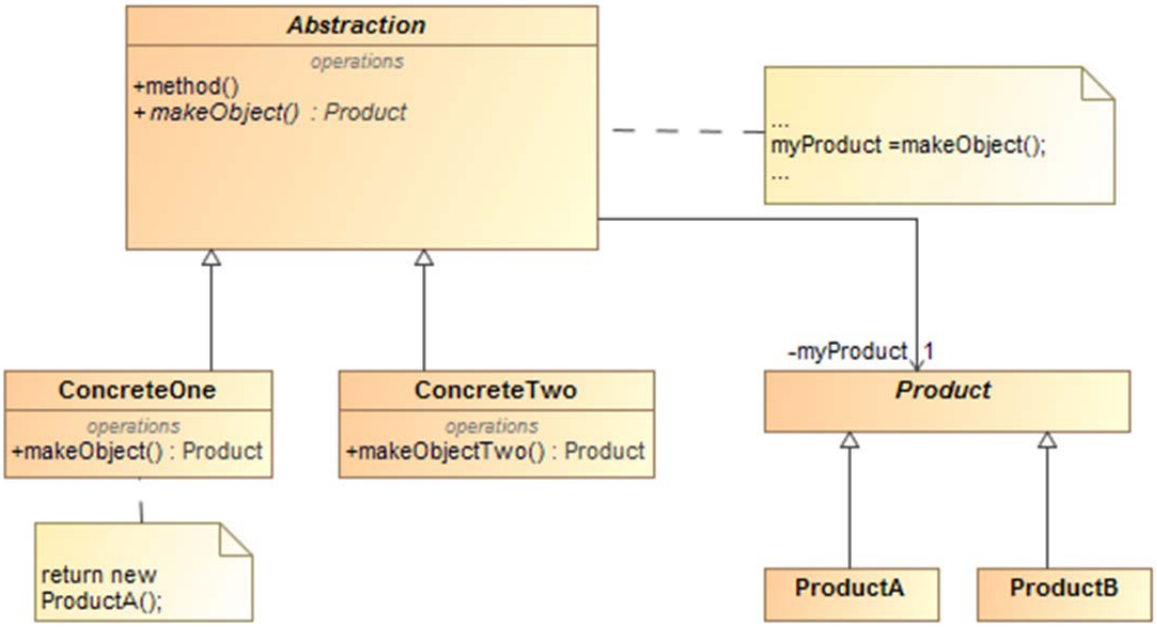
II. Ein Sequenzdiagramm für die main()-Methode der Client-Klasse. Stellen Sie die Interaktion zwischen allen beteiligten Objekten dar!

III. Eine Verhaltenszustandsmaschine (einschließlich der Modellierung von Aktionen)

<u>Aufgabe 3:</u> Entwurfsmuster (4 Punkte)

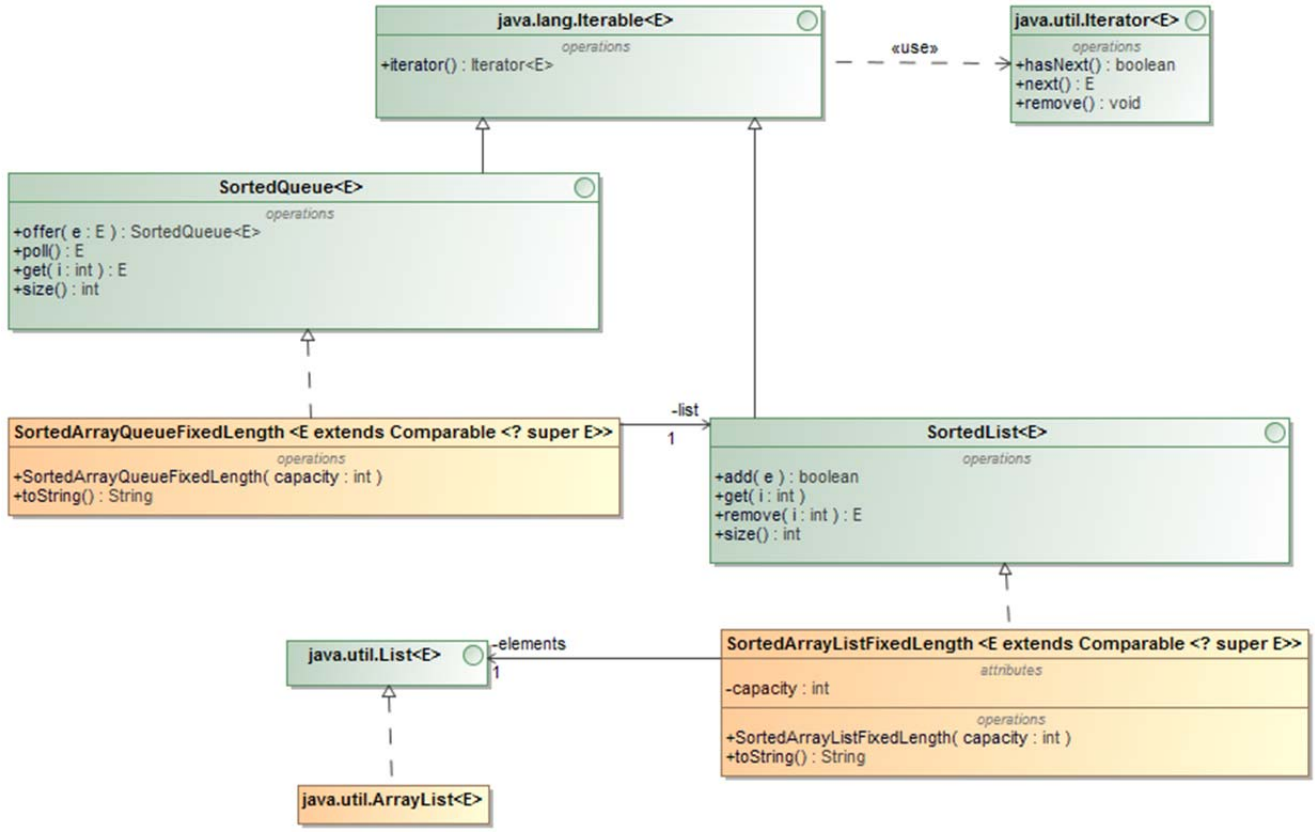
Es wird eine Schnittstelle für die Erzeugung von Objekten definiert. Die Entscheidung, welche konkrete Klasse zu instanziieren, zu konfigurieren und schließlich zurückzugeben ist, wird konkreten (Unter-) Klassen überlassen, die diese Schnittstelle implementieren.

Um welches Muster handelt es sich? Zeichnen Sie das Entwurfsmuster mit den beteiligten Rollen als UML-Kollaboration in das Diagramm ein!



Aufgabe 4: Datenstrukturen (SortedQueue) (28 Punkte)

Es soll eine sortierte Warteschlange mit einer festen Länge (`SortedArrayQueueFixedLength`) implementiert werden, die das Interface `SortedQueue` realisiert. Die Implementierung dieses Interfaces verwendet die Klasse `SortedArrayListFixedLength`. Diese Klasse wiederum realisiert das Interface `SortedList`. Das zugehörige Entwurfsklassendiagramm ist das folgende.



Implementieren Sie die Klassen `SortedArrayQueueFixedLength` und `SortedArrayListFixedLength` entsprechend dem UML-Modell! Ergänzen Sie dazu den folgenden Java-Code! Verzichten Sie auf den Test auf null-Objekte!

Hinweis zur Veranschaulichung der Datenstrukturen:
Die auf Seite 9 stehende `main()`-Methode gibt auf der Konsole aus:

```
list overflow
3
[a, c, d]
a
2
[c, d]
index is out of range
```

```
public static void main(String[] args) {
        SortedQueue <String> queue = new SortedArrayQueueFixedLength<>(3);
        queue.offer("d");
        queue.offer("c");
        queue.offer("a");
        queue.offer("e");
        System.out.println(queue.size());
        System.out.println(queue);
        System.out.println(queue.poll());
        System.out.println(queue.size());
        System.out.println(queue);
        queue.get(6);
    }
```

```
public interface SortedList <E> extends Iterable<E> {

    // Fügt ein Element hinzu.
    boolean add(E e);

    // Gibt die Referenz auf das i.-kleinste Element der Liste zurück.
    E get(int i);

    // Entfernt das i.-kleinste Element aus der Liste und gibt es zurück
    E remove(int i);

    // Gibt die Anzahl der in der Liste gespeicherten Elemente zurück.
    int size();
}
```

```
public interface SortedQueue<E> extends Iterable<E> {
    /**
     * Fügt das Element e in die Warteschlange ein.
     * Gibt die vergrößerte Warteschlange zurück.
     */
    SortedQueue<E> offer(E e);

    // Entfernt das kleinste Element aus der Warteschlange und gibt es zurück.
    E poll();

    // Gibt das i.-kleinste Element der Warteschlange zurück.
    E get(int i);

    // Gibt die aktuelle Länge der Warteschlange zurück.
    int size();
}
```

```
import java.util.*;

public class SortedArrayListFixedLength <E extends Comparable <? super E>>
implements SortedList<E> {
    ✍

    // Erzeugt eine sortierte Liste mit fester Maximalgröße (capacity).
    public SortedArrayListFixedLength (int capacity) {
    ✍

    }

    public int size() {
    ✍

    }

    // Gibt true zurück, wenn das Element eingefügt werden konnte, ansonsten false
    public boolean add (E e) {
    ✍

     }

     public String toString() {
         return elements.toString();
     }
```

```
    /**
     * Implementieren Sie die Methoden get() und remove() mit Ausnahmebehandlung!
     * Falls eine IndexOutOfBoundsException()-Exception geworfen wird, soll
     * auf der Konsole "index is out of range" ausgegeben werden und
     * es wird das null-Objekt zurückgegeben.
     */

    public E get(int i) {
    ✍

    }

    public E remove(int i) {
    ✍

    }

    public Iterator<E> iterator() {
    ✍

    }
}
```

```
import java.util.*;

public class SortedArrayQueueFixedLength <E extends Comparable <? super E>>
implements SortedQueue<E>{

   ✍


    // Erzeugt eine sortierte Warteschlange mit fester Maximalgröße (capacity).
    public SortedArrayQueueFixedLength(int capacity) {
    ✍



    }

    public E poll() {
    ✍


    }

    public int size() {
    ✍


    }

    public SortedQueue<E> offer(E e) {
    ✍



    }

    public E get(int i) {
    ✍


    }

    public Iterator<E> iterator() {
    ✍


    }

    public String toString() {
       return list.toString();
    }
}
```