Dr. D. Boles, Dr. C. Schönberg
Carl von Ossietzky Universität Oldenburg, Fakultät II, Department für Informatik

Wiederholungsklausur – Gruppe A (Lösungshinweise)

Objektorientierte Modellierung und Programmierung (inf031)

Sommersemester 2019

Aufgabe 1: *UML* **(17 Punkte)**

In dieser Aufgabe wird ein stark abgespecktes soziales Netzwerk namens Likebook modelliert.

Likebook besteht hierbei aus einer Menge an Nutzern (m/w/d) (user). Likebook kann seine aktuelle Datenbasis über zwei Methoden an einen nicht näher spezifizierten Ort in der Cloud sichern (save) und wieder herstellen (load). Diese Methoden sind *nicht* an eine konkrete Objektinstanz gebunden!

Jeder Nutzer hat einen Namen und eine Pinnwand (board), an der er Nachrichten (messages) hinterlassen kann. Jede Pinnwand kennt ihren Besitzer. Nur der Besitzer der Pinnwand kann dort Nachrichten schreiben, diesen Umstand müssen Sie allerdings nicht im Modell kenntlich machen.

Nutzer können Nachrichten (auch an fremden Pinnwänden) mit „Gefällt mir“ (like) markieren. Eine Nachricht besteht aus einem Text. Sie kennt die Pinnwand, an der sie geschrieben wurde. Für jede Gefällt-mir-Markierung ist bekannt, von welchem Nutzer sie stammt.

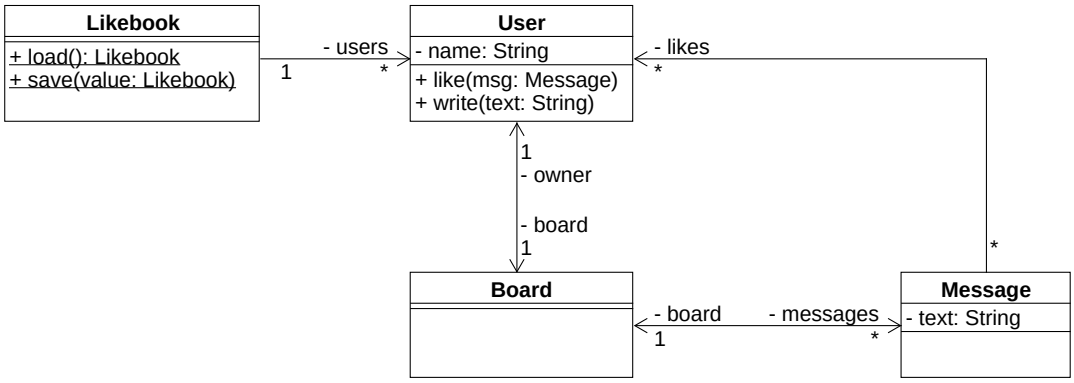
Nutzer können mit einer geeigneten Methode auf Gleichheit überprüft werden.

Aufgabe: Modellieren Sie Likebook als UML Klassendiagramm.

Hinweis 1: Achten Sie auf Vollständigkeit. Denken Sie insbesondere an Multiplizitäten, Bezeichner und Navigierbarkeiten.

Hinweis 2: Getter- und Setter-Methoden müssen Sie nicht angeben.

Lösungshinweise 1:



Aufgabe 2: *Lokale Suche* **(2 + 2 + 4 + 4 + 4 Punkte)**

Kreuzen Sie für jede Aussage entweder *Ja* oder *Nein* an. Korrekte Antworten werden positiv gewertet, falsche Antworten negativ, fehlende Antworten gehen nicht in die Bewertung ein. Sie können in jeder Teilaufgabe *keine* negativen Punkte erreichen, auch nicht bei überwiegend falschen Antworten. Falsche Antworten führen innerhalb einer Teilaufgabe zwar zu Punktabzug, haben aber keinen Einfluss auf andere Teilaufgaben und können auch in der gleichen Teilaufgabe nicht zu weniger als null Punkten führen.

Erinnerung: Es gibt u.a. die Booleschen Operatoren $\wedge$ (und), $\vee$ (oder) und $\neg$ (nicht).

Das Boolean Satisfiability Problem (kurz SAT) versucht (vereinfacht gesagt) für eine gegebene Boolesche Formel herauszufinden, ob es eine Variablenbelegung gibt, mit der die Formel erfüllt ist.

Beispielsweise ist die Formel $(a \wedge b) \vee (\neg a \wedge b)$ erfüllbar, weil sie z.B. für die Variablenbelegung $a = \text{true}, b = \text{true}$ gilt.

Die Formel $(a \vee b) \wedge \neg a \wedge \neg b$ ist nicht erfüllbar, weil es keine konsistente Variablenbelegung gibt, für die sie gilt.

Eine Lösung für SAT ist entweder eine Variablenbelegung, die zeigt, dass die gegebene Formel erfüllbar ist, oder die Aussage, dass die Formel nicht erfüllbar ist.

SAT ist NP-vollständig. Daher wird oft versucht, das Problem mittels Heuristiken zu lösen. Eine Heuristik für SAT kann eine definitive positive Antwort geben, indem sie eine gültige Variablenbelegung findet. Eine negative Antwort kann nur mit Sicherheit gegeben werden, nachdem alle (!) möglichen Belegungen durchprobiert wurden. Dies können Heuristiken oft nicht leisten.

a) Welche der folgenden Methoden werden für die *Lokale Suche* üblicherweise benötigt?

Ja	Nein	
[X]	[ ]	Gütefunktion: Bewertet die Qualität einer Lösung
[ ]	[X]	Verbesserungsfunktion: Leitet aus einer Lösung eine bessere Lösung her
[X]	[ ]	Startlösung: Erzeugt eine Lösung aus dem nichts
[ ]	[X]	Tabu-Funktion: Bestimmt, ob eine Lösung betrachtet werden darf

b) Welche der folgenden Aussagen treffen auf die *Lokale Suche* zu?

Ja	Nein	
[ ]	[X]	Sie baut schrittweise eine gültige Lösung auf
[X]	[ ]	Sie durchsucht den Lösungsraum
[X]	[ ]	Sie findet in der Regel eine korrekte Lösung
[ ]	[X]	Sie findet immer die optimale Lösung

c) Welche der folgenden Methoden erzeugen eine sinnvolle Nachbarschaft zu einer gegebenen Lösung für die *Lokale Suche bei SAT*?

Ja	Nein	
[ ]	[X]	Drehe alle Variablen um (true zu false und umgekehrt)
[X]	[ ]	Drehe eine zufällige Variable innerhalb einer unerfüllten Teilformel (z.B. $(b \vee \neg a)$) um
[X]	[ ]	Drehe zufällig gewählte 10 % aller Variablen um, mindestens aber eine
[ ]	[X]	Drehe die erste Hälfte aller Variablen um (z.B. a und b bei Variablen a, b, c, d)

d) Welche der folgenden Kriterien sind sinnvolle Abbruchkriterien für die *Lokale Suche bei SAT*?

Ja	Nein	
[X]	[ ]	Die aktuelle Lösung (Variablenbelegung) erfüllt die Formel
[ ]	[X]	Die aktuelle Lösung (Variablenbelegung) erfüllt die Formel *nicht*
[ ]	[X]	Die aktuelle Lösung (Variablenbelegung) enthält mehr true-Werte als die letzte Lösung
[ ]	[X]	Die Formel ist nicht erfüllbar

Weiter auf der nächsten Seite…

e) Für die Formel $a \vee (\neg a \wedge b \wedge c)$ soll eine *Lokale Suche für SAT* durchgeführt werden. Die Nachbarschaftsfunktion dreht eine zufällige Variable um. Die Suche läuft bereits, die aktuelle Variablenbelegung lautet $a = \text{false}, b = \text{false}, c = \text{false}$. Welche der folgenden Ereignisse können als nächstes eintreten?

Ja	Nein	
[X]	[ ]	Abbruch der Suche, weil zu viele Iterationen durchlaufen wurden
[X]	[ ]	Neue Variablenbelegung über die Nachbarschaftsfunktion ist $a = \text{false}, b = \text{false}, c = \text{true}$. Die Suche läuft weiter.
[ ]	[X]	Neue Variablenbelegung über die Nachbarschaftsfunktion ist $a = \text{false}, b = \text{true}, c = \text{false}$. Die Suche bricht erfolgreich ab: Die Belegung ist ausreichend gut.
[X]	[ ]	Neue Variablenbelegung über die Nachbarschaftsfunktion ist $a = \text{true}, b = \text{false}, c = \text{false}$. Die Suche bricht erfolgreich ab: Die Formel ist erfüllbar.

Aufgabe 3: *Prolog* **(1+1+1+1+1+1+1+1+2+2+2 Punkte)**

Gegeben sei folgende Prolog-Wissensbasis (die Sie bitte *nicht* als diagnostisches Werkzeug in die Realität übernehmen):

```
halsschmerzen(dr_house).
husten(dr_house).
kopfschmerzen(dr_wilson).
fieber(dr_wilson).
verstopfte_nase(dr_chase).
erkaeltung(dr_cuddy).

erkaeltung(X) :- halsschmerzen(X), husten(X).
erkaeltung(X) :- halsschmerzen(X), verstopfte_nase(X).
erkaeltung(X) :- husten(X), verstopfte_nase(X).

grippe(X) :- kopfschmerzen(X), fieber(X).

kollegen(dr_house, dr_cuddy).
kollegen(dr_house, dr_wilson).
kollegen(dr_house, dr_chase).

team(X, Y) :- kollegen(X, Y) ; kollegen(Y, X).
team(X, Z) :- team(X, Y), kollegen(Y, Z).

push([], X, [X]).
push([F|R], X, [F|RX]) :- push(R, X, RX).
```

Geben Sie die Ausgabe (z.B. `true`, `false` oder `dr_house`) des Prolog-Systems bei folgenden Eingaben an:

a) `?- erkaeltung(dr_house).`

b) `?- grippe(Y).`

c) `?- halsschmerzen(dr_cuddy).`

d) `?- halsschmerzen(dr_cuddy) ; husten(dr_cuddy) ; verstopfte_nase(dr_cuddy).`

e) `?- erkaeltung(dr_chase).`

f) `?- kollegen(dr_house, dr_cuddy).`

g) `?- kollegen(dr_chase, dr_house).`

h) `?- team(dr_chase, dr_wilson).`

i) `?- push([dr_house, dr_wilson, dr_cuddy], dr_chase, S).`

j) `?- push(S, dr_chase, [dr_house, dr_chase]).`

k) `?- push(S, dr_wilson, [dr_cuddy]).`

Mögliche Lösung 3:

a) `?- erkaeltung(dr_house). true .`

b) `?- grippe(Y). Y = dr_wilson.`

c) `?- halsschmerzen(dr_cuddy). false.`

d) `?- halsschmerzen(dr_cuddy) ; husten(dr_cuddy) ; verstopfte_nase(dr_cuddy). false.`

e) `?- erkaeltung(dr_chase). false.`

f) `?- kollegen(dr_house, dr_cuddy). true.`

g) `?- kollegen(dr_chase, dr_house). false.`

h) `?- team(dr_chase, dr_wilson). true .`

i) `?- push([dr_house, dr_wilson, dr_cuddy], dr_chase, S). S = [dr_house, dr_wilson, dr_cuddy, dr_chase].`

j) `?- push(S, dr_chase, [dr_house, dr_chase]). S = [dr_house] .`

k) `?- push(S, dr_wilson, [dr_cuddy]). false.`

Aufgabe 4: *Interfaces* **(18 Punkte)**

n und m seinen ganzzahlige Konstanten größer als 2. In einem Kreis stehen n Kinder (durchnummeriert von 0 bis $n-1$). Mit Hilfe eines m-silbigen Abzählreims wird das jeweils m-te unter den noch im Kreis befindlichen Kindern ausgeschieden, bis kein Kind mehr im Kreis steht.

Schreiben Sie ein Java-Programm, das nach Vorgabe von n und m die Nummern der Kinder in der Reihenfolge ihres Ausscheidens auf die Konsole ausgibt.

Beispiel 1: Für $n = 6$ und $m = 5$ ergibt sich die Folge $4, 3, 5, 1, 2, 0$
Beispiel 2: Für $n = 7$ und $m = 9$ ergibt sich die Folge $1, 4, 2, 3, 0, 5, 6$

Definieren Sie eine Klasse `CountingOutRhymeList`, die das Interface `java.lang.Iterable<T>` implementiert:

```
1  /** Implementing this interface allows an object to be the
2   * target of the ”for-each loop” statement */
3  public interface Iterable<T> {
4    // Returns an iterator over elements of type T.
5    Iterator<T> iterator();
6  }
```

mit

```
1  public interface Iterator<E> {
2    /** Returns true if the iteration has more elements */
3    boolean hasNext();
4
5    /** Returns the next element in the iteration */
6    E next();
7  }
```

Die Iterationsreihenfolge soll dabei die Reihenfolge sein, in der die Kinder ausscheiden. Sehen Sie in der Klasse `CountingOutRhymeList` einen Konstruktor vor, dem die Werte von n und m übergeben werden. Mit einer solchen Klasse kann das Abzählreim-Problem dann wie folgt umgesetzt werden:

```
1  final int n = 6; // n children
2  final int m = 5; // each m-th child is dropped out
3  CountingOutRhymeList children = new CountingOutRhymeList(n, m);
4  for (int child : children) {
5      System.out.println(child);
6  }
```

Hinweis 1: Überlegen Sie sich zunächst, mit welchem Typ das Interface `Iterable` parametrisiert werden muss.

Hinweis 2: Sie können eine Hilfsklasse `RhymeIterator` anlegen, um den von `CountingOutRhymeList` benötigten `Iterator` umzusetzen.

Hinweis 3: Welchen Lösungsalgorithmus Sie wählen, ist prinzipiell Ihnen überlassen. Der sicher einfachste Algorithmus, den Sie ja auch in PDA kennen gelernt haben, ist die Benutzung einer Queue (die Klasse können Sie als gegeben ansehen), in die anfangs alle n Kinder in der entsprechenden Reihenfolge eingefügt werden (enqueue). Beim Durchzählen wird das aktuelle Kind jeweils aus der Queue entfernt (dequeue) und direkt wieder eingefügt (enqueue). Das jeweils m-te Kind wird allerdings nur entfernt und nicht wieder eingefügt. Der Algorithmus endet, wenn die Queue leer ist.

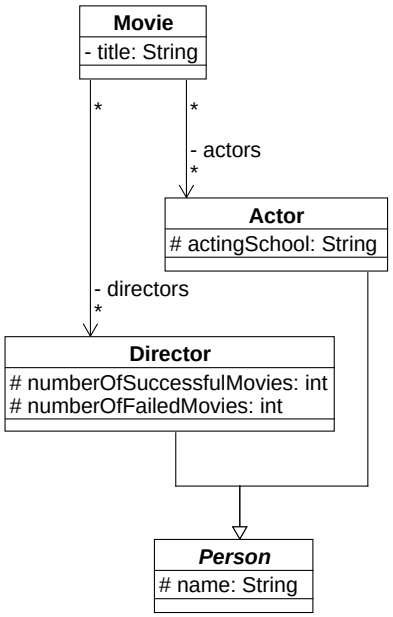
Lösungshinweise 4:

```
1  public class CountingOutRhymeList implements Iterable<Integer> {
2    private int number;
3    private int dropNumber;
4
5    public CountingOutRhymeList(int n, int m) {
6      number = n;
7      dropNumber = m;
8    }
9
10   @Override
11   public Iterator<Integer> iterator() {
12     return new RhymeIterator(number, dropNumber);
13   }
14
15   public static void main(String[] argv) {
16     final int n = 6; // n children
17     final int m = 5; // each m child is dropped out
18     CountingOutRhymeList children = new CountingOutRhymeList(n, m);
19     for (int child : children) {
20       System.out.println(child);
21     }
22   }
23 }
24
25 class RhymeIterator implements Iterator<Integer> {
26
27   private LinkedList<Integer> children;
28   private int number;
29   private int dropNumber;
30
31   public RhymeIterator(int n, int m) {
32     number = n;
33     dropNumber = m;
34     children = new LinkedList<>();
35     for (int i = 0; i < number; i++) {
36       children.add(i);
37     }
38   }
39
40   @Override
41   public boolean hasNext() {
42     return !children.isEmpty();
43   }
44
45   @Override
46   public Integer next() {
47     for (int i = 1; i < dropNumber; i++) {
48       children.add(children.remove());
49     }
50     return children.remove();
51   }
52
53 }
```

Aufgabe 5: *Collections, Streams* **(6 + 6 + 7 Punkte)**

Um eine Datenbank von Filmen abzubilden, stehen folgende Klassen zur Verfügung (Getter- und Setter-Methoden wurden der Übersichtlichkeit halber weggelassen, stehen aber natürlich zur Verfügung):

```
1  public class Movie {
2
3    // Filmtitel
4    private String title = "";
5
6    // Schauspieler
7    private List<Actor> actors = new ArrayList<>();
8
9    // Regisseure
10   private List<Director> directors = new ArrayList<>();
11
12 }
13 abstract class Person {
14
15   // Name
16   protected String name = "";
17
18 }
19 class Actor extends Person {
20
21   // Schauspielschule
22   protected String actingSchool = "";
23
24 }
25 class Director extends Person {
26
27   // Anzahl von erfolgreichen Filmen
28   protected int numberOfSuccessfulMovies = 0;
29
30   // Anzahl von gefloppten Filmen
31   protected int numberOfFailedMovies = 0;
32
33 }
```



Sie dürfen alle bekannten Klassen und Methoden verwenden, insbesondere die Klassen der Collections-API und der Stream-API. Hier sind zur Erinnerung einige Methoden (teils in vereinfachter Form) aufgeführt:

```
1  public Interface Collection<T> {
2    boolean add(T value);
3    boolean contains(Object value);
4    boolean remove(T value);
5    int size();
6    Stream<T> stream();
7  }
8  public Interface List<T> extends Collection<T
       > {
9    T get(int index);
10   int indexOf(Object value);
11   T remove(int index);
12 }
13 public Interface Set<T> extends Collection<T>
       { }
14 public Interface Map<K,V> {
15   V get(K key);
16   Set<K> keySet();
17   V put(K key, V value);
18   V remove(Object key);
19   int size();
20 }
```

```
1  public class ArrayList<T> implements List<T> { }
2  public class HashSet<T> implements Set<T> { }
3  public class HashMap<K,V> implements Map<K,V> { }
4  public Interface Stream<T> {
5    boolean allMatch(Predicate<T> p);
6    boolean anyMatch(Predicate<T> p);
7    long count();
8    Stream<T> distinct();
9    Stream<T> filter(Predicate<T> p);
10   Optional<T> findAny();
11   Stream<R> flatMap(
12       Function<T, Stream<R>> f);
13   void forEach(Consumer<T> c);
14   Stream<R> map(Function<T, R> f);
15   Optional<T> max(Comparator<T> c);
16   Optional<T> min(Comparator<T> c);
17   boolean noneMatch(Predicate<T> p);
18   Optional<T> reduce(
19       BinaryOperator<T> o);
20   Stream<T> sorted(Comparator<T> c);
21 }
```

Für einen Stream `stream` vom Typ `Stream<T>` kann mit `stream.collect(Collectors.toList())` eine `List<T>` erzeugt werden, die die Werte des Streams enthält.

Für ein Optional `opt` vom Typ `Optional<T>` kann mit `opt.orElse(null)` der enthaltene Wert vom Typ T zurückgegeben werden, oder **`null`**, falls der Optional keinen Wert enthält.

Weiter auf der nächsten Seite…

Aufgabe: Implementieren Sie innerhalb der Klasse `Movie` folgende statische Methoden:

```
1  public class Movie {
2    // ...
3
4    /** Gibt den Regisseur mit den meisten erfolgreichen Filmen zurueck, oder null, falls kein solcher
         Regisseur existiert. Wenn mehrere Regisseure qualifiziert sind, wird ein beliebiger
         zurueckgegeben. */
5    public static Director getBestDirector(Collection<Movie> movies) {
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25   }
26
27   /** Gibt alle Schauspieler zurueck, die in mindestens einem Film sowohl als Schauspieler als auch
         als Regisseur aufgetreten sind (Namensgleichheit). */
28   public static Collection<Actor> getActorsAsDirectors(Collection<Movie> movies) {
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48   }
49
50   /** Gibt alle Filme zurueck, in denen jeweils alle Schauspieler an der gleichen Schauspielschule
         gelernt haben (Namensgleichheit). */
51   public static Collection<Movie> getActorCliques(Collection<Movie> movies) {
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71   }
72 }
```

Lösungshinweise 5:

```
1  public static Director getBestDirector(Collection<Movie> movies) {
2    return movies.stream()
3    .flatMap((m) -> m.directors.stream())
4    .distinct()
5    .max((d1, d2) -> Integer.compare
6    (d1.getNumberOfSuccessfulMovies(), d2.getNumberOfSuccessfulMovies()))
7    .orElse(null);
8  }
9
10 public static Collection<Actor> getActorsAsDirectors(Collection<Movie> movies)
       {
11   return movies.stream()
12   .flatMap((m) -> m.actors.stream()
13     .filter((a) -> m.directors.stream()
14       .anyMatch((d) -> d.getName().equals(a.getName()))))
15   .distinct().collect(Collectors.toList());
16 }
17
18 public static Collection<Movie> getActorCliques(Collection<Movie> movies) {
19   return movies.stream()
20   .filter((m) ->
21     !m.actors.isEmpty()
22     && m.actors.stream().allMatch
23   ((a) -> a.getActingSchool().equals(
24     m.actors.get(0).getActingSchool())))
25   .collect(Collectors.toList());
26 }
```

Lösungshinweise 5:

```
1  public static Director getBestDirector(Collection<Movie> movies) {
2    Director result = new Director("", Gender.OTHER, 0, 0);
3    for (Movie movie : movies) {
4      for (Director director : movie.getDirectors()) {
5        if (director.getNumberOfSuccessfulMovies() >
6              result.getNumberOfSuccessfulMovies()) {
7          result = director;
8        }
9      }
10   }
11   return result;
12 }
13
14 public static Collection<Actor> getActorsAsDirectors(Collection<Movie> movies)
       {
15   List<Actor> result = new ArrayList<>();
16   for (Movie movie : movies) {
17     for (Actor actor : movie.getActors()) {
18       for (Director director : movie.getDirectors()) {
19         if (actor.getName().equals(director.getName())) {
20           result.add(actor);
21         }
22       }
23     }
24   }
25   return result;
26 }
27
28 public static Collection<Movie> getActorCliques(Collection<Movie> movies) {
29   List<Movie> result = new ArrayList<>();
30   for (Movie movie : movies) {
31     if (!movie.getActors().isEmpty()) {
32       String school = null;
33       boolean same = true;
34       for (Actor actor : movie.getActors()) {
35         if (school == null) {
36           school = actor.getActingSchool();
37         } else if (!school.equals(actor.getActingSchool())) {
38           same = false;
39           break;
40         }
41       }
42       if (same) {
43         result.add(movie);
44       }
45     }
46   }
47   return result;
48 }
```

Aufgabe 6: *Threads, Interfaces, Vererbung* **(6 + 10 Punkte)**

Gegeben seien folgende Interfaces und Klassen:

```
1  interface Callable<V> {
2    public V call();
3  }
4
5  class TaskPerformer<V> {
6
7    protected Callable<V> callable;
8
9    public TaskPerformer(Callable<V> callable) {
10     this.callable = callable;
11   }
12
13   public V get() {
14     return callable.call();
15   }
16 }
```

a) Implementieren Sie eine Klasse `Sum`, die das Interface `Callable` implementiert:

- Im Konstruktor der Klasse wird ein **`int`**-Wert `to` übergeben.
- Die Methode `call` berechnet und liefert die Summe der Zahlen von 1 bis `to`.

b) Das unten angegebene Programm nutzt die Klassen aus Teilaufgabe a).
Zunächst wird hier das *Produkt* der Zahlen von 1 bis 10 berechnet und anschließend in der Methode `get` die *Summe* der Zahlen von 1 bis 20. Das könnte eigentlich parallel geschehen und darum geht es in dieser Aufgabe.
Leiten Sie von der Klasse `TaskPerformer<V>` eine Klasse `BackgroundTaskPerformer<V>` ab. Dem Konstruktor wird ein `Callable`-Objekt übergeben. Im Konstruktor soll dann ein Thread erzeugt und gestartet werden, der das `Callable`-Objekt ausführt (Aufruf der Methode `call`), das gelieferte Ergebnis speichert und anschließend endet.
Überschreiben Sie weiterhin die Methode `get`. Von dieser soll das vom Thread berechnete und gespeicherte Ergebnis geliefert werden. U.U. muss jedoch noch gewartet werden, bis der Thread auch wirklich beendet ist.
Hinweis: Sie können Teilaufgabe b) unabhängig von Teilaufgabe a) bearbeiten.

```
1  public class BackgroundTasks {
2
3    public static void main(String[] args) throws Exception {
4      TaskPerformer<Integer> sum20 = new TaskPerformer<>(new Sum(20));
5      int prod = 1;
6      for (int i = 1; i <= 10; i++) {
7        prod *= i;
8      }
9      System.out.println(prod + sum20.get());
10   }
11
12 }
```

Lösungshinweise 6:

```
1  class Sum implements Callable<Integer> {
2
3    private int to;
4
5    public Sum(int to) {
6      this.to = to;
7    }
8
9    @Override
10   public Integer call() {
11     int result = 0;
12     for (int i = 1; i <= to; i++) {
13       result += i;
14     }
15     return result;
16   }
17
18 }
19 class BackgroundTaskPerformer<V> extends TaskPerformer<V> {
20
21   private V result;
22   private Thread backgroundThread;
23
24   public BackgroundTaskPerformer(Callable<V> callable) {
25     super(callable);
26     result = null;
27     backgroundThread = new Thread() {
28       @Override
29       public void run() {
30         result = callable.call();
31       }
32     };
33     backgroundThread.start();
34   }
35
36   public V get() {
37     try {
38       backgroundThread.join();
39     } catch (InterruptedException e) {
40     }
41     return result;
42   }
43 }
```