UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

Software Engineering im Wintersemester 2021/2022
Prof. Dr. Martin Leucker, Malte Schmitz, Stefan Benox, Julian Schulz, Benedikt Stepanek, Friederike Weilbeer, Tom Wetterich

Übungszettel 1 (Lösungsvorschlag)

19.11.2021

Abgabe bis Donnerstag, 28. Oktober um 23:59 Uhr online im Moodle.

Aufgabe 1.1: Neue Herausforderungen der Softwareentwicklung

4 Punkte, leicht

In der Vorlesung wurden im historischen Abriss des Software Engineerung unter anderem folgende neue Herausforderungen der Softwareentwicklung benannt. Erklären Sie jeweils, worum es sich dabei handelt und wieso es sich bei dieser Entwicklung um eine neue Herausforderungen für die Softwareentwicklung handelt.

1. Nebenläufigkeit

▼ **Lösungsvorschlag**

Unter Nebenläufigkeit wird die Fähigkeit eines Systems verstanden, mehrere Befehle quasi gleichzeitig auszuführen. Während in der klassischen Softwarentwicklung immer nur genau eine Anweisung zur Zeit ausgeführt wird und es auch nur genau einen Ausführungspfad gibt, entstehen durch die Parallelisierung von Prozessen viele neue Herausforderungen, insbesondere in der Synchronisation von parallelen Prozessen. Unter anderem Deadlocks, Livelocks und Race Conditions sind neue Probleme, die in sequenziellen Programmabläufen nicht auftreten

können. Die Optimierung von Algorithmen auf Parallelrechensystemen ist ein neues Forschungsfeld, dessen Bedeutung immer weiter zunimmt.

2. Interaktive Systeme

▼ **Lösungsvorschlag**

Der klassische Batch-Prozess hat eine Eingabe, verarbeitet dieses und generiert eine Ausgabe. Dieses Grundprinzip der Datenverarbeitung ist auch als EVA-Prinzip bekannt. Interaktive Systeme nehmen während der Ausführung immer wieder Eingaben entgegen und behalten dabei trotzdem einen internen Zustand bei. Viele Angriffe und Probleme von Software-Systemen sind nur auf interaktiven Systemen und entsprechenden Protokollen möglich, sodass sich hier neue Herausforderungen und Forschungsfelder ergeben.

3. Eingebettete Systeme

▼ **Lösungsvorschlag**

Ein eingebettetes System ist ein Computer, der in einen technischen Kontext eingebettet ist. Entsprechende Systeme sind nur möglich durch die Miniaturisierung von Computersystemen. Der klassische Großrechner konnte nicht in technische Systeme eingebettet werden. Durch diese Entwicklung konnten Computer erstmal als reaktive Systeme und Regelschleifen genutzt werden. Dabei ergeben sich ganz neue Herausforderungen an die Echtzeitfähigkeit von Systemen aber auch an die Fehlerbehandlung in sicherheitskritischen Systemen.

4. Cyber-Physical-Systems

▼ **Lösungsvorschlag**

Als Cyber-Physical-Systems werden komplexe Verbände aus mechanischen und elektronischen Systemen, sowie Computern bezeichnet, die miteinander interagieren. Der Begriff wird aktuell sehr gerne aber dabei nicht immer trennscharf verwendet. Der deutsche Wikipedia-Artikel fasst es gut zusammen:

> Insbesondere im anglo-amerikanischen Raum wird eine Theoriebildung zum Begriff „cyber-physisches System“ vorangetrieben, hier steht eine klare Abgrenzung des Begriffes gegenüber anderen Trends und Entwicklungsrichtungen komplexer informations- und kommunikationstechnischer Systeme im Vordergrund. In weiteren akademischen Untersuchungen stehen die sich ergebenden Herausforderungen der Systemkonstruktion im Blickpunkt. Zu den Herausforderungen zählen:
>
> - Komplexitätsreduktion und Entwicklung von stabilisierenden Steuerungsarchitekturen für cyber-physische Systeme
> - Verteilte Sensornetzwerke
> - Erschließung von Wissen und Erkenntnissen aus dem System heraus
> - Behandlung der Problematik der Interaktionskomplexität
> - Verlässliche Integration von Standardkomponenten in cyber-physische Systeme
> - Handhabbarkeit des Zusammenflusses von Sensorik, Aktorik und Steuerung
> - Verifikation von cyber-physischen Systemen
> - Sicherheit

Übernommen aus https://de.wikipedia.org/w/index.php?title=Cyber-physisches_System&oldid=202204960

Aufgabe 1.2: Maßnahmen zur Qualitätssicherung

3 Punkte, leicht

In der Vorlesung haben Sie unter anderem folgende Maßnahmen zur Qualitätssicherung kennen gelernt. Erläutern Sie für drei der folgenden Maßnahmen jeweils, was mit diesem Stichpunkt gemeint ist und wie diese Maßnahme zur Qualitätssicherung beiträgt.

1. Konsequente Methodenanwendung im Entwicklungsprozess

▼ **Lösungsvorschlag**

Der beste Entwicklungsprozess ist nutzlos, wenn er nicht tatsächlich gelebt wird. Entsprechend ergeben sich zwei Richtungen der Qualitätssicherung. Zum einen muss darauf geachtet werden, dass der dokumentierte und kommunizierte Prozess auch tatsächlich eingesetzt wird, aber zum anderen müssen insbesondere auch gelebte Vorgehen und praktische Abweichungen vom dokumentierten Prozess in die Dokumentation und Formalisierung des Entwicklungsprozesses aufgenommen werden. Sonst ist es zum Beispiel für neue Mitarbeiter oder externe nur sehr schwer nachzuvollziehen, wie tatsächliche Hierarchien und Abläufe funktionieren, wenn diese nicht dokumentiert und kommuniziert werden.

2. Einsatz adäquater Entwicklungswerkzeuge (CASE)

▼ **Lösungsvorschlag**

Der Begriff CASE bezeichnet Computer Aided/Assisted Software Engineering und ist ein Sammelbegriff für Software-Unterstützung in allen Phasen des Entwicklungsprozesses.

Konkret kann dies zum Beispiel die Nachverfolgbarkeit von Anforderungen durch den Entwicklungsprozess bezeichnen: Bei einem Commit im Versionskontrollsystem wird angegeben zu welchem Ticket diese Änderungen gehören. In diesem Ticket wird Bezug auf eine Anforderung genommen. Fehler werden behoben, in dem zunächst ein Test angelegt wird, der diesen Fehler reproduziert. Dabei wird auf eine Anforderung bezug genommen, aus der Hervorgeht, dass das beobachtete Verhalten tatsächlich ein Fehler ist. Im entsprechenden Commit, der diesen Fehler behebt, kann nun auf den entsprechenden Testcase und das Ticket bezug genommen werden.

Unter CASE-Tools werden darüber hinaus aber auch Integrierte Entwicklungsumgebungen (IDEs) verstanden, die Code-Editoren, Build-Systeme, Versionskontrollsysteme, Debugger und idealerweise auch Ticket-Systeme in einer gemeinsamen Oberfläche integrieren.

3. Institutionalisierung der Qualitätssicherung

▼ **Lösungsvorschlag**

Der Begriff Institutionalisierung bezeichnet im Zusammenhang mit Qualitätssicherung die formale Einrichtung von Qualitätssicherungsmaßnahmen und insbesondere die Schaffung von offiziellen Zuständigkeiten. Es müssen Posten und Aufgaben formal benannt und auch entsprechend zeitlich und damit finanziell ausgestattet werden, damit Qualitätssicherungsmaßnahmen praktisch durchgeführt werden können. Maßnahmen einfach von allen Mitarbeitern nebenbei machen zu lassen, kann dazu führen, dass unangenehme und zeitintensive Maßnahmen aufgeschoben und nicht durchgeführt werden. Entsprechend müssen sie bereits in der Planung berücksichtigt und beziffert werden.

4. Statische Programmanalyse

▼ **Lösungsvorschlag**

Die statistische Programmanalyse bezeichnet automatische Analysen von Software basierend auf dem Quellcode oder abgleiteten Modellen. Diese Analysen führen dabei das Programm nicht aus. Während die konkrete Ausführung von Software immer nur exemplarisch ist und auch Tests immer nur einzelne Ausführungspfade der Software betrachten, können mit statischer Analyse Aussagen zur Korrektheit von Software getroffen werden.

Statische Programmanalyse kann dabei alles umfassen von einfachen Hinweisen im Code-Editor bis hin zu vollständigen Nachweisen zur Korrektheit von Software. Während einfache Code-Analysen in der Regel sehr leicht zu aktivieren sind und direkt die Softwarequalität im kleinen verbessern, sind komplexere statische Analysen oft schwer aufzusetzen und müssen umfangreich an die konkrete Aufgabe angepasst werden. Entsprechend lohnt sich der Aufwand nur bei größeren sicherheitskritischen Projekten.

Das Projekt SLAM von Microsoft ist ein bekanntes Beispiel für komplexe statische Programmanalyse:

> SLAM is a project for checking that software satisfies critical behavioral properties of the interfaces it uses and to aid software engineers in designing interfaces and software that ensure reliable and correct functioning. Static Driver Verifier is a tool in the Windows Driver Development Kit that uses the SLAM verification engine.

5. Dynamische Programmanalyse

▼ Lösungsvorschlag

Dynamische Programmanalyse bezeichnet alle automatischen Analysen, die die Software ausführen oder die Ausführung der Software simulieren. Dabei werden immer nur einzelne Ausführungspfade der Software exemplarisch betrachtet. Im Gegensatz zur statischen Analyse lassen sich entsprechende Tests und Simulationen oft sehr einfach in Projekte integrieren. Einfache Unit Tests sind ein Beispiel für dynamische Programmanalyse. Die Auswirkungen auf die Softwarequalität ergeben sich insbesondere, wenn durch regelmäßig ausgeführte Test Suites neu eingebrachte Fehler sofort automatisiert gefunden werden.

Aufgabe 1.3: Objektorientierte Programmierung in Java

5 Punkte, mittel

Erläutern Sie fünf der folgenden Konzepte der objektorientierten Programmierung in Java und geben Sie jeweils ein kurzes Java-Beispielprogramm an, welches das Konzept veranschaulicht.

1. Objektidentität und semantische Gleichheit

▼ Lösungsvorschlag

Zwei verschiedene Objekte sind nie identisch. Objektidentität beschreibt zwei Referenzen, die auf das gleiche Objekt zeigen. In Java wird die Identität von Objekten mit dem Operator `==` geprüft. Die Gleichheit von Objekten wird explizit durch den Programmierer definiert. In Java passiert das durch überschreiben der Methode `equals`, die auch explizit aufgerufen werden muss, um die Gleichheit von Objekten zu überprüfen.

Dieser folgende Beispielcode illustriert den Unterschied am Beispiel einer Datenklasse `Tuple`, die zwei Integer speichert:

```
public class Tuple {
    public final int a;
```

```
    public final int b;
    public Tuple(int a, int b) {
        this.a = a;
        this.b = b;
    }
    @Override
    public boolean equals(Object o) {
        if (o instanceof Tuple) {
            Tuple other = (Tuple) o;
            return this.a == other.a && this.b == other.b;
        }
        return false;
    }
}
```

Zwei Tupel sind genau dann gleich, wenn beide gespeicherten Werte `a` und `b` gleich sind:

```
Tuple a = new Tuple(1,2);
Tuple b = a;
Tuple c = new Tuple(1,2);
Tuple d = new Tuple(1,3);
System.out.println(a == b); // true
System.out.println(a.equals(b)); // true
System.out.println(b == c); // false
System.out.println(b.equals(c)); // true
System.out.println(b == c); // false
System.out.println(b.equals(d)); // false
```

2. Vererbung

▼ **Lösungsvorschlag**

Eine Klasse erbt Attribute und Methoden von ihrer Oberklasse.

Die folgende Klasse `A` definiert die Integer-Attribute `a` und `b`.

```
class A {
    int a, b;
}
```

Die folgende Klasse `B` erbt von der Klasse `A` die Attribute `a` und `b` und ergänzt eine eigene Methode `go`:

```
class B extends A {
    int go() {
        return a * b + 10 + b;
```

```
    }
}
```

Der folgende Code-Schnipsel zeigt eine Verwendung dieser Klasse `B`:

```
B b = new B();
b.a = 15;
b.b = 2;
System.out.println(b.go()); // 42
```

3. Schnittstelle

▼ **Lösungsvorschlag**

Eine Schnittstelle definiert eine Menge von Methoden, die von allen Klassen implementiert werden müssen, die diese Schnittstelle realisieren. Auf diese Weise können konkrete Implementierungen zur Laufzeit ausgetauscht werden, solange sie die gleiche Schnittstelle implementieren. Die folgende Schnittstelle schreibt eine Method `f` vor, die zwei Integer verrechnet:

```
interface Operation {
    int f(int a, int b);
}
```

Diese Schnittstelle kann nun von einer Klasse `Numbers` verwendet werden, um eine Methode bereitzustellen, die alle ihre Zahlen miteinander verrechnet:

```
public class Numbers {
    final int[] numbers;
    public Numbers(int[] numbers) {
        this.numbers = numbers;
    }
    int go(Operation operation) {
        int result = numbers[0];
        for (int i = 1; i < numbers.length; i++) {
            result = operation.f(result, numbers[i]);
        }
        return result;
    }
}
```

Die Schleife zum Verarbeiten aller Zahlen ist dabei unabhängig von der konkreten Rechenoperation. Mögliche Berechnungen könnten zum Beispiel die Summe:

```
class Add implements Operation {
    @Override
    public int f(int a, int b) {
        return a + b;
    }
}
```

oder das Produkt aller Zahlen sein:

```
class Mul implements Operation {
    @Override
    public int f(int a, int b) {
        return a * b;
    }
}
```

Folgender Code verwendet diese Klassen um die Summe und das Produkt der Zahlen von 1 bis 5 zu berechnen:

```
Numbers numbers = new Numbers(new int[] {1,2,3,4,5});
System.out.println(numbers.go(new Add())); // 15
System.out.println(numbers.go(new Mul())); // 120
```

4. Überladung von Methoden

▼ Lösungsvorschlag

Als Überladen von Funktionen oder Methoden bezeichnet man die Deklaration mehrere Funktionen oder Methoden im gleichen Scope, die sich nicht im Namen, sondern nur ihrer Signatur unterscheiden. Der Compiler wählt die richtige Methode dann anhand der Typen aus. In Java können Methoden nach Parametertypen überladen werden. Die folgende Klasse enthält drei verschiedene Methoden `add`:

```
public class Adder {
    int add(int a, int b) {
        return a + b;
    }
    int add(String a, int b) {
        return Integer.parseInt(a) + b;
    }
    int add(int a, String b) {
        return a + Integer.parseInt(b);
    }
    int go() {
        return add(add("9", 10), add(11, "12"));
```

```
    }
}
```

Wird die Methode `go` aufgerufen, so liefert diese den Wert `42`, da jeweils die richtige Methode `add` aufgerufen wird, sodass die Strings zunächst in Integer konvertiert werden:

```
System.out.println(new Adder().go()); // 42
```

5. Überschreiben von Methoden und Polymorphie

▼ **Lösungsvorschlag**

Als polymorphisch oder virtuell bezeichnet man Methoden, bei denen während der Übersetzung noch nicht klar ist, welchen Code sie enthalten. In Java sind alle Methoden einer Klasse virtuell, da sie in erbenden Klassen überschrieben werden können.

Folgende Klasse hat eine Methode `calc`, die die übergebenen Integer aufaddiert. Sie tut dies unter Verwendung der Methode `op`, die zwei Integer addiert, und der Methode `c`, die das neutrale Element `0` liefert:

```
public class Sum {
    int c() { return 0; }
    int op(int a, int b) { return a + b; }
    int calc(int... args) {
        int result = c();
        for (int i:args)
            result = op(result, i);
        return result;
    }
}
```

Die Klasse `Product` erbt von `Sum` und überschreibt die Methoden `op` und `c`. Sie erbt dadurch die Methode `calc`, die nun aber das Produkt und nicht mehr die Summe berechnet:

```
public class Product extends Sum {
    int c() { return 1; }
    int op(int a, int b) { return a * b; }
}
```

Bei der Verwendung ruft man auf beiden Klassen die Methode `calc` auf, die sich jedoch unterschiedlich verhält, je nach dem, wie Methoden `c` und `op` implementiert wurden. Das besondere an Polymorphie ist dabei, dass während der Übersetzung der Methode `calc` aus der Klasse `sum` noch nicht klar ist, welcher Code beim Aufruf der Methoden `c` und `op` innerhalb der Methode `sum` ausgeführt wird:

```
System.out.println((new Sum()).calc((new Product()).calc(2, 4, 4), 5, 2, 3)); // 42
```

6. Klassenmethode und Klassenattribut

▼ **Lösungsvorschlag**

Als Klassenmethoden und -attribute bezeichnet man in Java Methoden und Attribute, die über alle Instanzen der Klasse hinweg gleich sind und entsprechend auch ohne Instanziierung direkt auf der Klasse aufgerufen werden können. Solche Attribute und Methoden werden daher auch als statisch bezeichnet, da nicht dynamisch anhand der Objektinstanz entschieden wird, wohin die Referenzen zeigen. In Java werden sie daher durch das Keyword `static` gekennzeichnet.

Die folgende Klasse `Counter` nutzt ein statisches Attribut um zu zählen, wie oft sie instanziiert wurde:

```
public class Counter {
    static private int c = 0;
    static int getCount() {
        return c;
    }
    Counter() {
        c += 1;
    }
}
```

Über den statischen Getter `getCount` kann der aktuelle Stand abgefragt werden:

```
new Counter();
new Counter();
System.out.println(Counter.getCount()); // 2
```

Statische Attribute und Methoden sollten in Objektorientiertem Code nur sehr selten eingesetzt werden. Durch sie können sehr schnell die Effekte von globalen veränderlichen Variablen herbeigeführt werden, die man durch Kapselung vermeiden möchte.

7. Generische Klassen und Methoden

▼ **Lösungsvorschlag**

Generics helfen dabei, von konkreten Typen zu abstrahieren, ohne vollständig auf eine Typüberprüfung zur Compile-Zeit zu verzichten.

Angenommen wir wollen die Klasse `Tuple` aus dem ersten Beispiel so erweitern, dass diese nicht nur Integer, sondern beliebige Wertpaare speichern kann. Wir können dies leicht erreichen, in dem wir `int` durch `Object` ersetzen, da in Java alles ein Objekt sein kann. (Dank Autoboxing auch ein Integer, der eigentlich ein primitiver Datentyp und kein Objekt ist.)

```
public class ObjectTuple {
    public final Object a;
    public final Object b;
    public ObjectTuple(Object a, Object b) {
        this.a = a;
        this.b = b;
    }
}
```

Der Code funktioniert, allerdings kann der Compiler nun nicht mehr überprüfen, ob die Werte aus dem Tupel korrekt verwendet werden. Wir müssen diese manuell in den richtigen Typ casten:

```
ObjectTuple t = new ObjectTuple(1, "Hallo");
System.out.println((int) t.a + 1); // 2
System.out.println((int) t.a + (int) t.b); // Laufzeitfehler: class java.lang.String cann
```

Mit Generics können wir die Klasse so umbauen, dass diese zwar immer noch für beliebige Typen funktioniert, aber der Compiler sich diesen Typ merkt, da die Klasse bei der Übersetzung für verschiedene Typen instanziiert wird:

```
public class GenericTuple<A,B> {
    public final A a;
```

```
    public final B b;
    public GenericTuple(A a, B b) {
        this.a = a;
        this.b = b;
    }
}
```

Jetzt funktioniert obiges Beispiel ohne manuelle Typkonvertierungen und der Fehler wird bereits bei der Übersetzung gefunden:

```
GenericTuple<Integer, String> t2 = new GenericTuple<>(1, "Hallo");
System.out.println(t2.a * 2); // 2
System.out.println(t2.a * t2.b); // Compilefehler: bad operand types for binary operator
```

8. Nullreferenzen und Optionals

▼ Lösungsvorschlag

Ein Nullreferenz bezeichnet eine Referenz, die nicht auf ein Objekt zeigt. In Java wird eine solche Referenz durch das Schlüsselwort `null` angegeben. Angenommen, wir haben eine Funktion, die das erste Element eines Arrays zurückgibt, so kann diese zum Beispiel eine Nullreferenz zurückgeben, wenn das Array leer ist, also kein erstes Element existiert:

```
public class UsingNull {
    static <A> A first(A[] a) {
        if (a.length > 0) {
            return a[0];
        }
        return null;
    }
}
```

In der Verwendung müsste der Rückgabewert dieser Methode nun immer auf `null` überprüft werden, bevor mit dem Wert gearbeitet werden kann:

```
Integer[] a = {1, 2, 3};

Integer i = UsingNull.first(a);
if (i != null) {
    System.out.println(i2);
}
```

Eine modernere Variante, die von funktionalen Programmiersprachen inspiriert ist, ist die Verwendung optionaler Typen. Dabei wird immer eine Instanz vom Typ `Optional<Integer>` statt `Integer` zurückgegeben:

```
public class UsingOptional {
    static <A> Optional<A> first(A[] a) {
        if (a.length > 0) {
            return Optional.of(a[0]);
        }
        return Optional.empty();
    }
}
```

Mit dieser Instanz kann nun weiter gearbeitet werden:

```
Optional<Integer> i = UsingOptional.first(a);
i.ifPresent(System.out::println);
```

Neben einer manchmal kompakteren Notation ist der zentrale Vorteil dieser Lösung, dass das Typsystem bereits zur Compile-Zeit darauf achtet, dass ein entsprechender Test durchgeführt wird. Eine Variable vom Typ `Optional<Integer>` kann nicht ohne weitere Überprüfung an Stellen verwendet werden, wo ein `Integer` erwartet wird.

9. Java Stream API

▼ **Lösungsvorschlag**

Die Stream API stellt eine Möglichkeit dar, die aus funktionalen Sprachen bekannten Konzepte `map`, `reduce` und `flatMap` in Java zu verwenden. Die Idee ist dabei, Operationen auf Mengen von Werten anzugeben, ohne explizit den Zugriff auf die einzelnen Werte in der Datenstruktur zu beschreiben. So werden in folgendem Beispiel die Elemente einer Liste gefiltert, verändert und sortiert ohne explizit eine Schleife anzugeben, die über die einzelnen Elemente der Liste iteriert:

```
List<String> myList =
Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList
    .stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
```

```
        .sorted()
        .forEach(System.out::println);
```

Neben `filter` und `map` sind `reduce`, `collect` und `flatMap` weitere zentrale Konzepte der Stream API.