Übung zur Vorlesung Software Engineering

Klausurvorbereitung—Teil 2

TECHNISCHE UNIVERSITÄT DARMSTADT

Sammlung von Übungsaufgaben—Teil 2

Dieses Übungsblatt ist Teil 2 einer Sammlung von Übungsaufgaben zur zusätzlichen Vertiefung. Die Zusammenstellung erlaubt keinerlei Aussagen über die Klausur. Insbesondere werden nicht alle klausurrelevanten Themen abgedeckt.

Aufgabe 1: Factory Pattern

Beschreibung: Die Java-Anwendung *MultiDocEdit* erlaubt es, (Text-)Dokumente in unterschiedlichen Formaten zu erstellen. Zur Zeit werden als Format HTML und LaTeX unterstützt.

Um einheitlich mit den unterschiedlichen Formaten arbeiten zu können, ist die gemeinsame Struktur in allen Formaten mit Hilfe von Interfaces modelliert. Ein Dokument (`Document`) besteht aus mehreren Abschnitten (`Section`). Über die Methode `sections()` können alle verfügbaren Abschnitte abgefragt, und via `addSection(Section)` neue Abschnitte hinzugefügt werden. Ein Abschnitt selbst besitzt einen Titel, welcher über `getTitle()` ausgelesen und via `setTitle(String)` geändert werden kann.

Die Benutzeroberfläche zum Bearbeiten von Dokumenten ist in der Klasse `DocumentEditor` implementiert. Das Format des Dokuments wird über einen `String`-Bezeichner, welcher dem Konstruktor der Klasse übergeben wird, festgelegt. Nachdem der Benutzer den Titel eines neuen Abschnitts definiert hat, wird Methode `addSectionToDocument(Document , String)` mit dem zu erweiternden Dokument und dem Titel als Parameter aufgerufen. Sie erzeugt einen neuen Abschnitt, setzt den Titel und fügt ihn anschließend dem Dokument hinzu.

Aufgabe:

a) Instanzen der Interface-Implementierungen beider Formate sollen mit Hilfe des Entwurfsmusters Abstract Factory (Abstrakte Fabrik) erstellt werden. **Vervollständigen** Sie das in Abbildung 1 dargestellte **Klassendiagramm** um die benötigten Bestandteile des Entwurfsmusters, sodass Dokumente und Abschnitte in den Formaten HTML und LaTeX erzeugt werden können.
Fügen Sie zunächst die benötigten Klassen mit sinnvollen Bezeichnern hinzu. Geben Sie Methoden nur in der abstrakten Fabrik an. Modellieren Sie anschließend die Vererbungs-, Implementierungs- und Erzeugungsbeziehungen. Zuletzt soll die verwendete Fabrik im `DocumentEditor` über eine Assoziation mit dem Bezeichner **factory** zugreifbar sein. Geben Sie sowohl die Richtung als auch die Multiplizitäten auf beiden Seiten der Assoziation an.

b) Implementieren Sie die Methode zum Erzeugen von Abschnitten (`Section`) im **HTML**-Format in Java. Geben Sie die **vollständige Signatur** sowie den **Methodenkörper** an.

```
public Section createSection() {
  return new HTMLSection();
}
```

c) Implementieren Sie den **Konstruktor** der Klasse `DocumentEditor`, `DocumentEditor(String docType)`, in welchem eine Instanzvariable `factory` gemäß des übergebenen Bezeichners für den Dokumententyp initialisiert wird. Berücksichtigen Sie dabei Ihr Ergebnis aus Teilaufgabe a).

```
public DocumentEditor(String docType) {
  final Map<String, DocumentFactory> factories = new HashMap<>();
  factories.put("latex", LaTeXFactory.instance());
  factories.put("html", HTMLFactory.instance());
  if (!factories.containsKey(docType)) throw new RuntimeException("Unknown factory");
  this.factory = factories.get(docType);


}
```

d) Implementieren Sie die Methode `addSectionToDocument(Document, String)` unter Anwendung des Entwurfsmusters Abstract Factory (Abstrakte Fabrik) in Java. Sie erzeugt einen neuen Abschnitt, setzt den Titel und fügt ihn anschließend dem Dokument hinzu. Gehen Sie davon aus, dass die zu verwendende Fabrik als Instanzvariable **factory** zur Verfügung steht. Geben Sie den Methodenkörper an.

```
protected void addSectionToDocument(Document document, String title) {
  Section section = factory.createSection();
  section.setTitle(title);
  document.addSection(section);


}
```

e) Ordnen Sie die Elemente Ihrer Lösung aus Teil a) den Bestandteilen des Entwurfsmusters Abstract Factory zu.

Abstrakte Produkte:

- Document,
- Section

Konkrete Implementierungen der abstrakten Fabrik:

- HTMLFactory,
- LaTeXFactory

Konkrete Implementierungen der Produkte:

- HTMLDocument,
- LaTeXDocument,
- HTMLSection,
- LaTeXSection

Methoden zum Erstellen von Produkten:

- createDocument(),
- createSection()

Abstrakte Fabrik:

- DocumentFactory

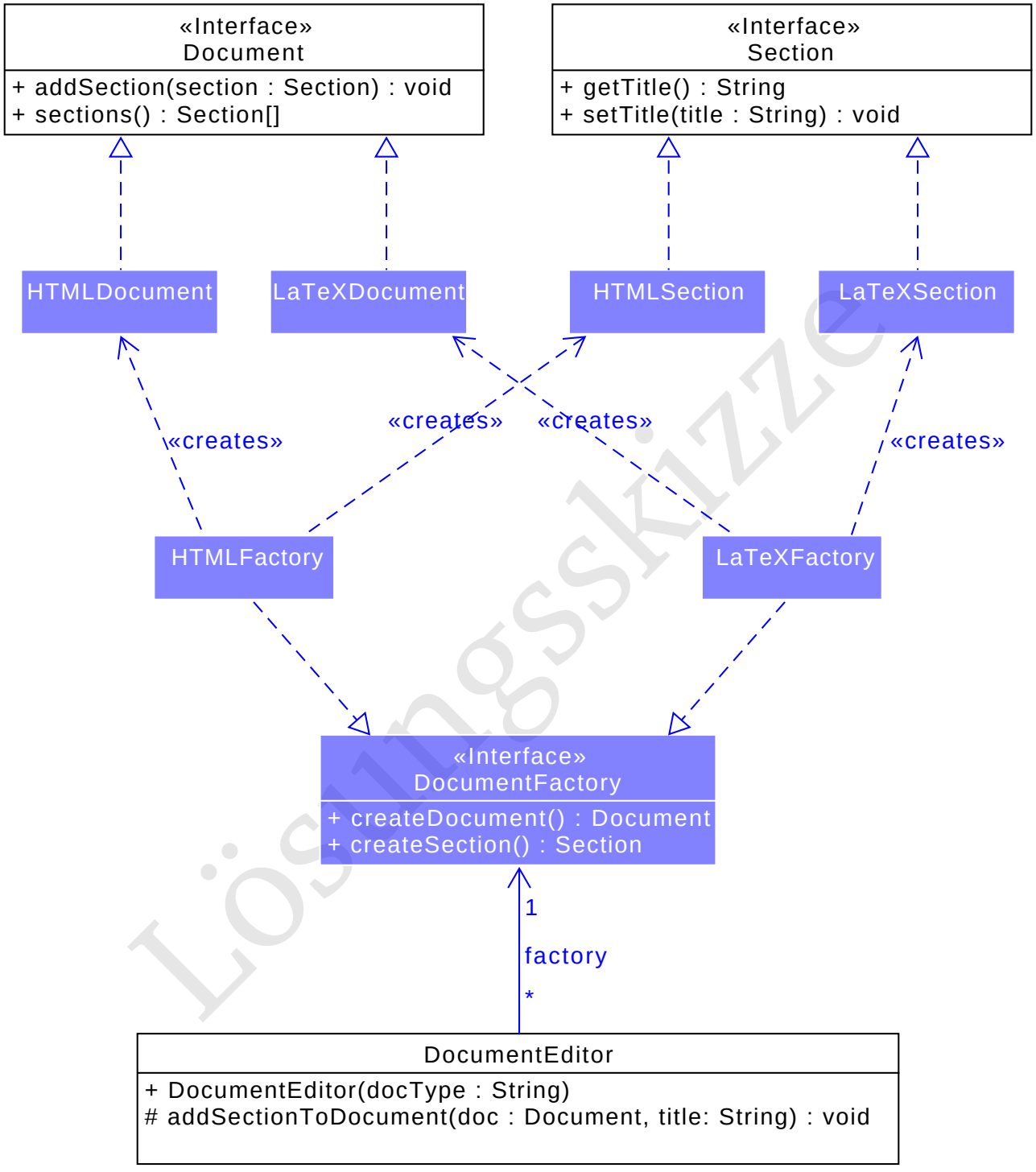


Abbildung 1: Klassendiagramm *MultiDocEdit*

Aufgabe 2: Systematisches Testen von Methoden

Aufgabe:

a) Erstellen Sie einen **minimalen** Testplan für 100 % **Statement Coverage (SC)** für die Java-Methode `magic` in Listing 1. Nutzen Sie die Grundstruktur eines Testplans wie sie in Tabelle 1 dargestellt wird und geben Sie Zeilennummern als Erklärung an, z.B. 3, 4 – 5, . . .

b) Erstellen Sie den Kontrollflussgraph für Methode `magic` in Listing 1 und nummerieren Sie die Kanten fortlaufend, z.B. E1, E2, . . . Erstellen Sie anschließend einen **minimalen** Testplan für 100 % **Branch Coverage (BC)**. Nutzen Sie die Grundstruktur eines Testplans wie sie in Tabelle 1 dargestellt wird und geben Sie die Namen der Kanten als Erklärung an, z.B. E1, E2, . . .

c) Erstellen Sie einen **minimalen** Testplan für **Path Coverage (PC)**, der alle technisch möglichen Pfade abdeckt mit Hilfe des Kontrollflussgraph aus Aufgabenteil b). Nutzen Sie die Grundstruktur eines Testplans wie sie in Tabelle 1 dargestellt wird und geben Sie die Namen der Kanten als Erklärung an, z.B. E1, E2, . . .

d) Erstellen Sie einen **minimalen** Testplan für 100 % **Condition Coverage (CC)** für Methode `magic` in Listing 1. Nutzen Sie die Grundstruktur eines Testplans wie sie in Tabelle 1 dargestellt wird und geben Sie Zeilennummern einer Condition mit dessen boolschen Wert als Erklärung an, z.B. 4 = true, 8 = true, . . .

e) Erstellen Sie einen **minimalen** Testplan für 100 % **Decision Coverage (DC)** für Methode `magic` in Listing 1. Nutzen Sie die Grundstruktur eines Testplans wie sie in Tabelle 1 dargestellt wird und geben Sie Zeilennummern einer Decision mit dessen boolschen Wert als Erklärung an, z.B. 4 — 5 = false, 8 — 11 = true

f) Erstellen Sie je einen Testplan bestehend aus zwei Testfällen, um die Erfüllung der **Modified Condition Decision Coverage (MCDC)** für die Conditions in Zeile 4 und 5 sowie in Zeile 8, 9 und 10 in **Listing 2** zu zeigen. Nutzen Sie die Grundstruktur eines Testplans wie sie in Tabelle 1 dargestellt wird und geben Sie Zeilennummern einer Condition sowie der gesamten Decision mit zugehörigen boolschen Werten als Erklärung an, z.B. 4 = false, 4 — 5 = false, . . .

g) Erstellen Sie einen **minimalen** Testplan für 100 % **Multiple-Condition Coverage (MCC)** für Methode `magic` in Listing 1. Nutzen Sie die Grundstruktur eines Testplans wie sie in Tabelle 1 dargestellt wird. Eine Erklärung ist nicht notwendig.

h) Ist eine Methode, die erfolgreich zu 100 % mit allen Coverages getestet wurde, garantiert fehlerfrei? Begründen Sie Ihre Meinung mit einem kleinem Beispiel.

Hinweis: Ein minimaler Testplan enthält die kleinstmögliche Anzahl an Testfällen, mit denen die gewünschte Testabdeckung erreicht wird.

Test Case magic(a, b, c)	Erwartetes Ergebnis	Erklärung (z.B. mit Hilfe von Zeilennummern)

Tabelle 1: Grundstruktur eines Testplans

Listing 1: Magic (Aufgabenteile a bis e und g)

```
1  public class Magic {
2     public static int magic(boolean a, boolean b, String c) {
3        int result = 1;
4        if (a ||
5            b) {
6           result ++;
7        }
8        if (a &&
9            b ||
10           (c.isEmpty() &&
11            a)) {
12          return result;
13       }
14       else {
15          return result * -1;
16       }
17    }
18 }
```

Listing 2: Alternative Magic (Aufgabenteil f)

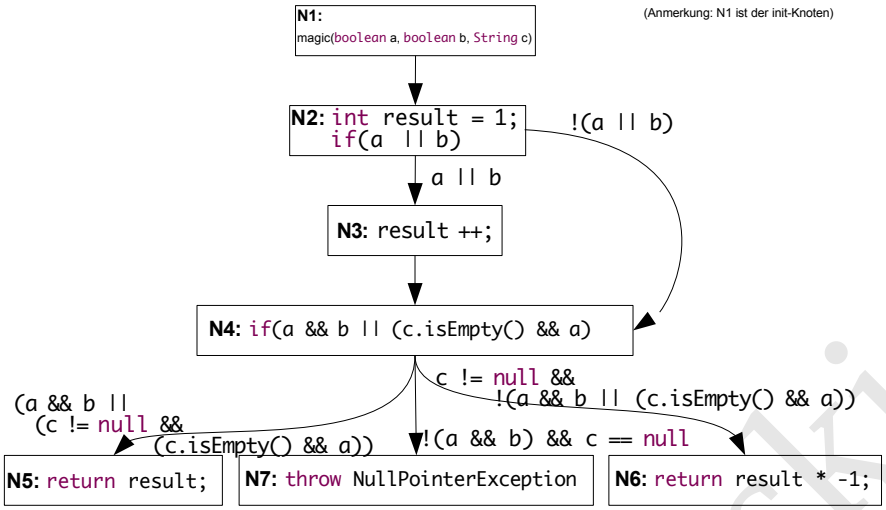
```
1  public class Magic {
2     public static int magic(boolean a, boolean b, String c) {
3        int result = 1;
4        if (a ||
5            b) {
6           result ++;
7        }
8        if ((b ||
9             c.isEmpty()) &&
10           a) {
11          return result;
12       }
13       else {
14          return result * -1;
15       }
16    }
17 }
```

Lösung:

a) Statement Coverage (SC)

Test Case magic(a, b, c)	Erwartetes Ergebnis	Erklärung
true, true, "C"	2	Zeilen: 3, 4 – 5, 6, 8 – 11, 12
false, false, "C"	-1	Zeilen: 3, 4 – 5, 8 – 11, 15

b) Branch Coverage (BC)



Test Case magic(a, b, c)	Erwartetes Ergebnis	Erklärung
true, true, "C"	2	Kanten: N1 – N2, N2 – N3, N3 – N4, N4 – N5
false, false, "C"	-1	Kanten: N1 – N2, N2 – N4, N4 – N6
false, false, null	NullPointerException	Kanten: N1 – N2, N2 – N4, N4 – N7

c) Path Coverage (PC)

Test Case magic(a, b, c)	Erwartetes Ergebnis	Erklärung
true, true, "C"	2	Kanten: N1 – N2, N2 – N3, N3 – N4, N4 – N5
true, false, "C"	-2	Kanten: N1 – N2, N2 – N3, N3 – N4, N4 – N6
true, false, null	NullPointerException	Kanten: N1 – N2, N2 – N3, N3 – N4, N4 – N7
false, false, "C"	-1	Kanten: N1 – N2, N2 – N4; N4 – N6
false, false, null	NullPointerException	Kanten: N1 – N2, N2 – N4; N4 – N7

Der Pfad mit den Kanten N1 – N2, N2 – N4; N4 – N5 ist nicht möglich.

d) Condition Coverage (CC)

Test Case magic(a, b, c)	Erwartetes Ergebnis	Erklärung
true, false, ""	2	Zeilen: 4 = true, 8 = true, 9 = false, 10 = true, 11 = true
false, true, ""	-2	Zeilen: 4 = false, 5 = true, 8 = false, 10 = true, 11 = false
true, true, "C"	2	Zeilen: 4 = true, 8 = true, 9 = true
false, false, "C"	-1	Zeilen: 4 = false, 5 = false, 8 = false, 10 = false

e) Decision Coverage (DC)

Test Case magic(a, b, c)	Erwartetes Ergebnis	Erklärung
true, true, ""	2	Zeilen: 4 – 5 = true, 8 – 11 = true
false, false, "C"	-1	Zeilen: 4 – 5 = false, 8 – 11 = false

f) Modified Condition Decision Coverage (MCDC) für Condition in Zeile 4

Test Case magic(a, b, c)	Erwartetes Ergebnis	Erklärung
false, false, ""	-1	Zeilen: 4 = false, 5 = false, 4 – 5 = false
true, false, ""	2	Zeilen: 4 = true, 4 – 5 = true

f) Modified Condition Decision Coverage (MCDC) für Condition in Zeile 5

Test Case magic(a, b, c)	Erwartetes Ergebnis	Erklärung
false, false, "C"	-1	Zeilen: 4 = false, 5 = false, 4 – 5 = false
false, true, "C"	-2	Zeilen: 4 = false, 5 = true, 4 – 5 = true

f) Modified Condition Decision Coverage (MCDC) für Condition in Zeile 8

Test Case magic(a, b, c)	Erwartetes Ergebnis	Erklärung
true, false, "C"	-2	Zeilen: 8 = false, 9 = false, 8 – 10 = false
true, true, "C"	2	Zeilen: 8 = true, 10 = true, 8 – 10 = true

f) Modified Condition Decision Coverage (MCDC) für Condition in Zeile 9

Test Case magic(a, b, c)	Erwartetes Ergebnis	Erklärung
true, false, "C"	-2	Zeilen: 8 = false, 9 = false, 8 – 10 = false
true, false, ""	2	Zeilen: 8 = false, 9 = true, 10 = true, 8 – 10 = true

f) Modified Condition Decision Coverage (MCDC) für Condition in Zeile 10

Test Case magic(a, b, c)	Erwartetes Ergebnis	Erklärung
false, false, ""	-1	Zeilen: 8 = false, 9 = true, 10 = false, 8 – 10 = false
true, false, ""	2	Zeilen: 8 = false, 9 = true, 10 = true, 8 – 10 = true

g) Multiple-Condition Coverage (MCC)

Test Case magic(a, b, c)	Erwartetes Ergebnis	Erklärung
false, false, "C"	-1	
false, false, ""	-1	
false, true, "C"	-2	
false, true, ""	-2	
true, false, "C"	-2	
true, false, ""	2	
true, true, "C"	2	

100 % ist nicht möglich, da aufgrund der short-circuit Evaluation nicht alle MCC möglich sind (true, true, _) führt `c.isEmpty()` nicht aus.

h) Begründung zur Fehlerfreiheit

Nein ist sie nicht, denn obwohl die Methode aus Listing **??** mit dem folgenden Testplan getestet wurde und alle Coverages zu 100 % erfüllt sind, ist das Ergebnis bei gleichen Werten (z.B. 0, 0) fehlerhaft. Daher empfiehlt es sich weitere Verfahren, wie z.B. Äquivalenzklassenbildung und das Testen von Grenzwerten, bei der Testplanerstellung zu nutzen.

Test Case greaterBug(f, s)	Erwartetes Ergebnis	Erklärung
5, 1	true	Zeilen: 3, 4
1, 5	false	Zeilen: 3, 7

Listing 3: Numbers

```
public static boolean greaterBug(int f, int s) {
   if (f >= s) { // correct: f > s
      return true;
   }
   else {
      return false;
   }
}
```

Aufgabe 3: Black Box Testen

Die Java Methode `public static double median(int[] a)` berechnet den Median des beim Aufruf als Argument übergebenen Arrays. Im Falle eines leeren Arrays oder bei einer Nullreferenz wird eine `IllegalArgumentException` geworfen.

Geben Sie fünf nicht redundante Testfälle, die aus einer Blackbox Betrachtung (d.h. ohne Kenntnisse des Codes) der Methode heraus sinnvoll wären. Sie können sich hierfür auch an den in der Vorlesung vorgestellten Heuristiken zum Finden relevanter Testeingaben orientieren.

Tragen Sie die Testfälle in die unten gegebene Tabelle ein. Die Beschreibung jedes Testfalls muss möglichst prägnant sein. Geben Sie in der Wertebelegung den Inhalt des Arrays in eckigen Klammern, also zum Beispiel `[1,2,3]` für ein Array mit den drei Elementen 1, 2 und 3.

Nr.	Beschreibung	Wertebelegung von `a`	Erwartetes Ergebnis / Exception

Nr.	Beschreibung	Wertebelegung von `a`	Erwartetes Ergebnis
1	Übergabe einer Nullreferenz	`null`	IllegalArgumentException
2	Übergabe eines leeren Arrays	[]	IllegalArgumentException
3	Übergabe eines mit gerader Anzahl an Elementen	[1,2]	1.5
4	Übergabe eines mit ungerader Anzahl an Elementen	[1,2,3]	2
5	Übergabe eines mit gerader Anzahl an Elementen, bei denen die Summe der beiden benachbarten Zahlen größer als `Integer.MAX_VALUE` ist	[`Integer.MAX_VALUE`,`Integer.MAX_VALUE`]	`Integer.MAX_VALUE`
6	Übergabe eines mit gerader Anzahl an Elementen, bei denen die Summe der beiden benachbarten Zahlen kleiner als `Integer.MIN_VALUE` ist	[`Integer.MIN_VALUE`,`Integer.MIN_VALUE`]	`Integer.MIN_VALUE`
7	Array der Länge `Integer.MAX_VALUE`	[1,1,1,1,....]	1
8	...	...	...

Aufgabe 4: Planen einer Iteration

Beschreibung:

Ihr Unternehmen hat bei einer Firma die Entwicklung einer Software beauftragt. Als Vorgehensmodell wurde vertraglich Extreme Programming (XP) mit **10-tägigen Iterationen** vereinbart. Als Fachexperte sind Sie mit der Auswahl der zu implementierenden User Stories beauftragt. Tabelle 2 zeigt alle verfügbaren User Stories sowie die tatsächlich benötigte Zeit bei der Implementierung aus der vorherigen (dritten) Iteration. Kundenprioritäten für die vierte Iteration sind ebenfalls angegeben (**1 ist die höchste Priorität**). Alle User Stories sind unabhängig voneinander und sollten in der Reihenfolge gemäß der Kunden-Priorität realisiert werden.

User Story	Story Points	Geplant für Iteration	Tatsächliche Zeit (Tage)	Kundenpriorität
US 2	5	3	2 Tage	
US 5	2	3	2 Tage	
US 11	13	2	Nicht abgeschlossen	3
US 12	3	3	1 Tage	
US 13	3	3	Nicht abgeschlossen	4
US 14	2	3	1 Tage	
US 15	8	3	3 Tage	
US 16	8			7
US 17	5			1
US 18	13			5
US 19	5			8
US 20	8			2
US 21	3			6
US 22	2			9

Tabelle 2: User Stories

Aufgabe:

a) Bestimmen Sie die Velocity für Iteration 3. **Geben Sie den vollständigen Rechenweg an!**

b) Legen Sie die User Stories für Iteration 4 unter Berücksichtigung der Prioritäten des Kunden und der maximal realisierbaren Story Points fest. **Begründen Sie die Auswahl kurz in ein bis zwei Sätzen.**

Lösung a:

$$\begin{aligned}\text{Velocity} =& \textit{Summe der Story Points der abgeschlossenen User Stories}\\ =& \frac{5\ \textit{Story Points} + 2\ \textit{Story Points} + 3\ \textit{Story Points} + 2\ \textit{Story Point} + 8\ \textit{Story Point}}{Iteration}\\ =& 20 \frac{\textit{Story Points}}{Iteration}\end{aligned}$$

Lösung c:

Liste der User Stories für Iteration 4: US 17, US 20, US 13, US 21. ($\Rightarrow$ 19 SP)

US 17 und 20 werden am dringendsten vom Kunden benötigt.
US 11 hat die nächsthöhere Kundenpriorität, überschreitet jedoch die maximale Anzahl an Story Points.
US 13 ist die nächstdringliche User Story, welche die maximale Anzahl an Story Points nicht überschreitet.
US 21 ist die nächstdringliche User Story, welche die maximale Anzahl an Story Points nicht überschreitet.

Aufgabe 5: Feature Diagramm Constraints

Abbildung 2 zeigt das Feature Diagramm für diese Aufgabe.

1. Geben Sie die Boolean Integer Formel für das Diagramm aus Abbildung 2 an.
2. Begründen Sie kurz, ob die folgenden Produkte möglich sind (die Produkte sind jeweils als Liste der gewünschten (aktivierten) Features gegeben):
   a) A,B,C,E
   b) A,B,C,D,E
3. Geben Sie ein äquivalentes, aber vereinfachtes Feature Diagramm an.

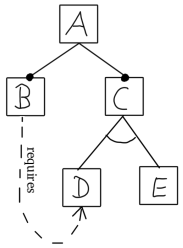


Abbildung 2: Feature Diagramm für Aufgabe 5

Lösung:

1. $A \wedge (A \leftrightarrow B) \wedge (A \leftrightarrow C) \wedge (D \leftrightarrow (C \wedge \neg E)) \wedge (E \leftrightarrow (C \wedge \neg D)) \wedge (B \rightarrow D)$
2. a) Kein gültiges Produkt: Wenn B vorhanden ist, muss auch D vorhanden sein.
   b) Kein gültiges Produkt: Es dürfen entweder nur D oder nur E vorhanden sein.
3. Ein vereinfachtes Diagramm kann durch das Entfernen von Feature E erreicht werden, da B zwingend (mandatory) vorhanden sein muss und damit auch D, E kann somit nie als Feature aktiviert werden (requires lässt sich auch entfernen, falls D als mandatory markiert wird). Achtung, dass so erstellte Diagramm ändert die Featuremenge, da E als Feature entfernt wird.

Aufgabe 6: Feature Diagramme

Aufgabe: Die Firma *CYoD* (*C*onfigure *Y*our *o*wn *D*evice) stellt eine Produktfamilie von Smartphones her, die sich in Ihrer Ausstattung unterscheiden und selbst konfiguriert werden können.

Ihre Aufgabe ist es die untenstehende Beschreibung in ein Feature Diagramm mit dem Wurzelfeature (*CYoD*-Phone) zu überführen, welches die nachstehende Beschreibung möglichst genau abbildet. Achten Sie insbesondere auf eine sinnvolle Gruppierung der Features.

Beschreibung: Das *CYoD-Phone* ist im Hinblick auf Batteriekapazität, Prozessor, Bildschirmqualität, drahtlose Kommunikationstechnik, Kameraausstattung sowie biometrische Authentifikation konfigurierbar.

Bei den Batterien stehen Batterien mit niedriger Kapazität (frei wählbar zw. 1500 und 2500 mAh) und hoher Kapazität (frei wählbar zw. 3000 und 6000mAh) zur Auswahl.

Es kann zwischen einem langsameren, aber energiesparsamen Prozessor (STP, steady turtle processor) oder einem schnellen, aber energiehungrigen Prozessor (ERP, exhausted rabbit processor) gewählt werden. Smartphones mit einem ERP benötigen zwingend Batterien hoher Kapazität.

Die Bildschirmqualität muss genau einer der Varianten Standard-Definition (SD), High-Definition (HD) oder Ultra-High-Definition (UHD) entsprechen.

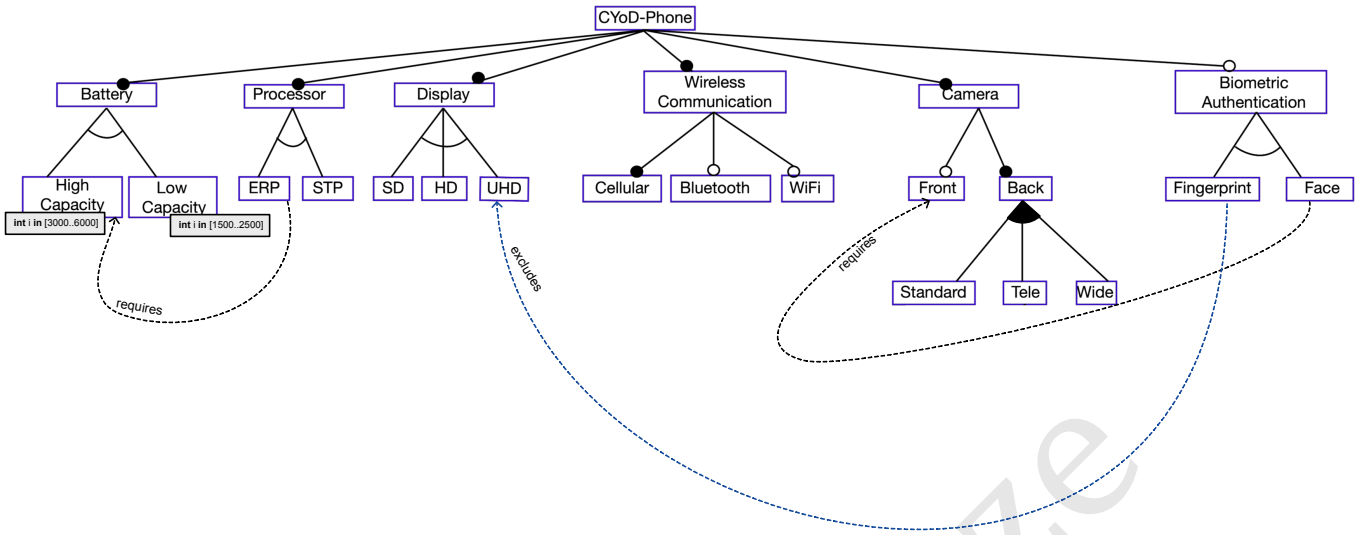
Jedes *CYoD-Phone* muss mindestens eine der folgenden drahtlosen Kommunikationstechniken unterstützen: Cellular, Bluetooth und WiFi. Die Technik *Cellular* muss auf jeden Fall unterstützt werden.

Bei den Kameras wird zwischen Frontkameras (auf der Vorderseite) und Backkameras (auf der Rückseite) unterschieden. Jedes *CYoD-Phone* hat höchstens eine Frontkamera, aber mindestens eine Kamera auf der Rückseite. Bei den Backkameras können bis zu drei Kameras ausgewählt werden, aber höchstens je eine aus den Arten Standard, Tele und Weitwinkel.

Wenn biometrischen Authentifizierung unterstützt werden soll, muss entweder Fingerabdruck- oder Gesichtserkennung als Authentifizierungstechnik gewählt werden. Im Falle der Gesichtserkennung muß eine Frontkamera vorhanden sein. Soll die biometrische Authentifizierung über den Fingerabdruck geschehen, so schließt dies einen UHD-Bildschirm aus, da der Fingerabdruckssensor im Bildschirm integriert ist und die Integration bei den verwendeten UHD Bildschirmen noch nicht möglich ist.

Bei der Ausstattung ist neben der Einhaltung der oben beschriebenen Details, insbesondere auch zu berücksichtigen, dass jedes *CYoD-Phone* genau einen Prozessor, einen Bildschirm und eine Batterie besitzt. Die Unterstützung einer biometrischen Authentifizierungstechnik ist optional. Einschränkungen an andere Ausstattungselemente (sofern es bei diesen Einschränkungen gibt) sind im oberen Text genannt.

Lösung:



Aufgabe 7: Zusammenhänge zwischen internen und externen Faktoren

Aufgabe: Ordnen sie zehnmal einen internen Faktor einen externen Faktor zu, sofern dieser Einfluss auf ihn hat. Begründen Sie jede Zuordnung aussagekräftig in einem Satz.

Software Qualität: Interne Faktoren

- modular
- comprehensible (dt. verständlich)
- cohesion: clear responsibilities
- concision: (almost) no code duplication

Software Qualität: Externe Faktoren

- Robustness
- Extendibility
- Reusability
- Compatibility
- Portability
- Efficiency
- Ease of use
- Functionality (meets user expectations)

Lösung:

Software Qualität: Interne Faktoren

- modular
  - Extendibility: Modularer Code erlaubt es Module auszutauschen oder hinzuzufügen.
  - Reusability: Module können in anderen Anwendungen wiederverwendet werden.
  - Portability: Plattformspezifische Funktionalität kann durch austauschbare Module realisiert werden.
  - Efficiency: Erlaubt es einfachere Algorithmen später gegen komplexere Auszutauschen.
- comprehensible (dt. verständlich)
  - Extendibility: Nur verständliche Funktionen können erweitert werden.
  - Reusability: Nur verständliche Komponenten können wiederverwendet werden.
  - Efficiency: Nur wer den Algorithmus versteht, kann ihn optimieren.
- cohesion: clear responsibilities
  - Extendibility: Klare Trennung der Verantwortlichkeiten erlaubt es Funktionen auszutauschen oder hinzuzufügen.
  - Reusability: Klare Verantwortlichkeiten führen zu modularen Komponenten die wiederverwendbar sind.
  - Portability: Trennung erlaubt den Austausch von Klassen, die plattformspezifisch sind.
  - Efficiency: Trennung erlaubt es, nicht performante Funktionen zu erkennen.
- concision: (almost) no code duplication
  - Robustness: Wird kein Code dupliziert, werden auch keine Fehler dupliziert.
  - Extendibility: Wird kein Code dupliziert, können Erweiterungen zentral an einer Stelle vorgenommen werden.
  - Reusability: Das Vermeiden von Code-Duplikationen, z.B. durch Auslagerung von Teilfunktionalität in Methoden, ermöglicht eine Wiederverwendung.
  - Efficiency: Wird kein Code dupliziert, wird auch kein ineffizienter Code dupliziert.

Aufgabe 8: Systematisches Testen von Methoden

Aufgabe:

a) Erstellen Sie einen **minimalen** Testplan für 100 % **Statement Coverage (SC)** für die Java-Methode `magic` in Listing 3. Nutzen Sie die Grundstruktur eines Testplans wie sie in Tabelle 3 dargestellt wird und geben Sie Zeilennummern als Erklärung an, z.B. 3, 4 – 6, ...

b) Erstellen Sie einen **minimalen** Testplan für **Branch Coverage (BC)**, der alle erreichbaren Kanten des Kontrollflussgraphen in Abbildung 3 abdeckt. Nutzen Sie die Grundstruktur eines Testplans wie sie in Tabelle 3 dargestellt wird und geben Sie die Kanten im Kontrollflussgraphen als Erklärung an, z.B. E1, E4, ...

c) Erstellen Sie einen **minimalen** Testplan für 100 % **Condition Coverage (CC)** für Methode `magic` in Listing 3. Nutzen Sie die Grundstruktur eines Testplans wie sie in Tabelle 3 dargestellt wird und geben Sie Zeilennummern einer Condition mit dessen booleschen Wert als Erklärung an, z.B. 4 = true, 5 = true, ...

d) Erstellen Sie je einen Testplan bestehend aus zwei Testfällen, um die Erfüllung der **Modified Condition Decision Coverage (MCDC)** für die Conditions in Zeile 4, 5 und 6 in Listing 3 zu zeigen. Nutzen Sie die Grundstruktur eines Testplans wie sie in Tabelle 3 dargestellt wird und geben Sie Zeilennummern einer Condition sowie der gesamten Decision mit zugehörigen booleschen Werten als Erklärung an, z.B. 4 = false, 4 — 6 = true, ...

Hinweis: Ein minimaler Testplan enthält die kleinstmögliche Anzahl an Testfällen, mit denen die gewünschte Testabdeckung erreicht wird. Arrays können Sie als Listen darstellen, bspw. $(1, 2, 3)$.

Test Case magic(array, i, b)	Erwartetes Ergebnis/Exception	Erklärung (z.B. mit Hilfe von Zeilennummern)

Tabelle 3: Grundstruktur eines Testplans

Listing 4: Magic

```
1  public class Magic {
2     public static int magic(int[] array, int i, boolean b) {
3        int result = 0;
4        if ((array.length >= 2 ||
5             i >= 4) &&
6            b) {
7           int index = 0;
8           while (index < i) {
9              result += array[index];
10             index++;
11          }
12       }
13       return result;
14    }
15 }
```

Lösung:

a) Statement Coverage (SC)

Test Case magic(array, i, b)	Erwartetes Ergebnis/Exception	Erklärung (z.B. mit Hilfe von Zeilennummern)
(0, 0), 2, true	0	Zeilen: 3, 4 – 6, 7, 8, 9, 10, 13

b) Branch Coverage (BC)

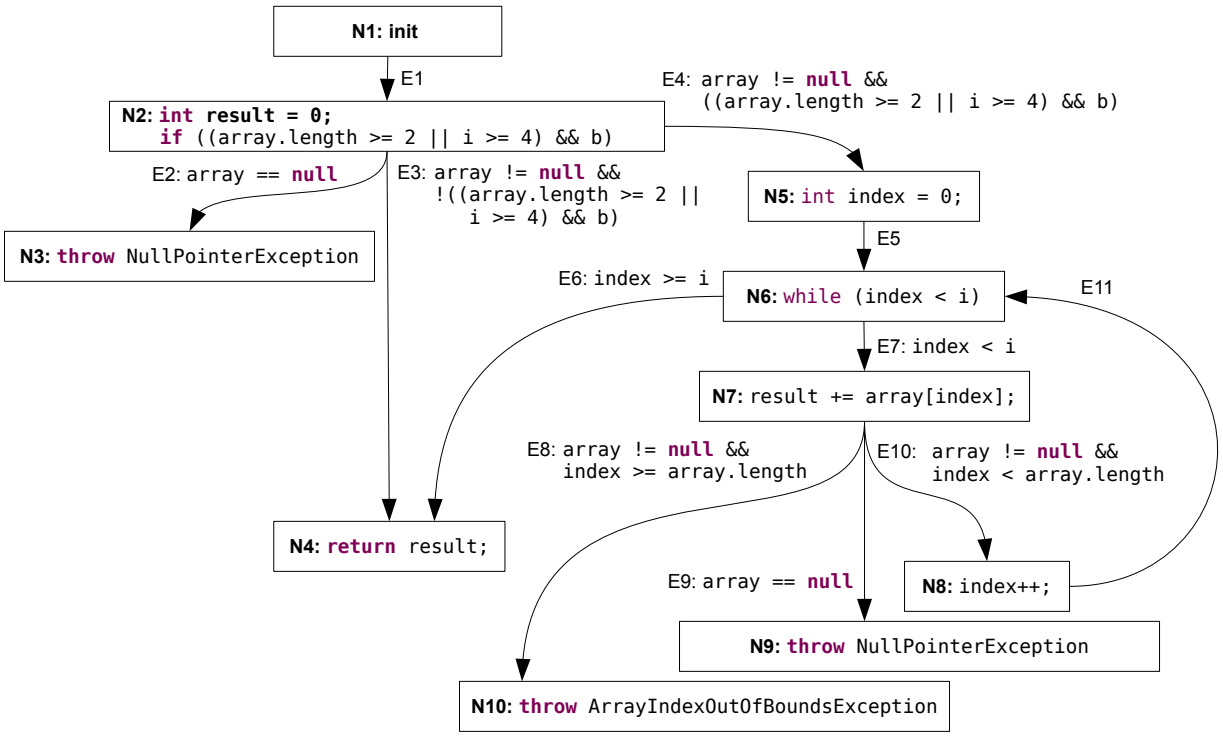


Abbildung 3: Kontrollflussgraph zu Listing 3

Test Case magic(array, i, b)	Erwartetes Ergebnis/Exception	Erklärung (z.B. mit Hilfe von Zeilennummern)
null, 2, true	NullPointerException	Kanten: E1, E2
(), 0, false	0	Kanten: E1, E3
(0, 0), 2, true	0	Kanten: E1, E4, E5, E7, E10, E11, E7, E10, E11, E6
(), 4, true	ArrayIndexOutOfBoundsException	Kanten: E1, E4, E5, E7, E8

Die Kante E9 ist nicht möglich, da im Falle, dass `array == null` gilt, die Exception bereits bei Kante E2 geworfen wird.

c) Condition Coverage (CC)

Test Case magic(array, i, b)	Erwartetes Ergebnis/Exception	Erklärung (z.B. mit Hilfe von Zeilennummern)
(0, 0, 0), 2, true	0	Zeilen: 4 = true, 6 = true, 8 = true/false
(0), 2, true	0	Zeilen: 4 = false, 5 = false
(0), 5, false	0	Zeilen: 4 = false, 5 = true, 6 = false

d) Modified Condition Decision Coverage (MCDC) für Condition in Zeile 4

Test Case magic(array, i, b)	Erwartetes Ergebnis/Exception	Erklärung (z.B. mit Hilfe von Zeilennummern)
(0, 0, 0), 2, true	0	Zeilen: 4 = true, 6 = true, 4 – 6 = true
(0), 2, true	0	Zeilen: 4 = false, 5 = false, 4 – 6 = false

d) Modified Condition Decision Coverage (MCDC) für Condition in Zeile 5

Test Case magic(array, i, b)	Erwartetes Ergebnis/Exception	Erklärung (z.B. mit Hilfe von Zeilennummern)
(0), 2, true	0	Zeilen: 4 = false, 5 = false, 4 – 6 = false
(0), 5, true	ArrayIndexOutOfBoundsException	Zeilen: 4 = false, 5 = true, 6 = true, 4 – 6 = true

d) Modified Condition Decision Coverage (MCDC) für Condition in Zeile 6

Test Case magic(array, i, b)	Erwartetes Ergebnis/Exception	Erklärung (z.B. mit Hilfe von Zeilennummern)
(0, 0, 0), 2, true	0	Zeilen: 4 = true, 6 = true, 4 – 6 = true
(0, 0, 0), 2, false	0	Zeilen: 4 = true, 6 = false, 4 – 6 = false