



UNIVERSITÄT STUTTGART

ADVANCED SOFTWARE ENGINEERING

Übungsblatt 4

Maximilian Peresunchak (st152466@stud.uni-stuttgart.de)

Nico Reng (st188620@stud.uni-stuttgart.de)

Viorel Tsigos (st188085@stud.uni-stuttgart.de)

Philip Reimann (st182312@stud.uni-stuttgart.de)

Christian Keller (st166512@stud.uni-stuttgart.de)

Johannes Heugel (st183360@stud.uni-stuttgart.de)

Benedikt Wachmer (st177118@stud.uni-stuttgart.de)

Miles Holl (st180549@stud.uni-stuttgart.de)

Wintersemester 2025

14. Dezember 2025

Integrationsteststrategien

a)

#	A	B	D	C	E
1	A	b		c	
2	A	B	d	c	e
3	A	B	D	c	e
4	A	B	D	C	e
5	A	B	D	C	E

Legende:

Integrierte Komponente, Platzhalter (Stubs)

Ausführliche Beschreibung:

Da es Top-Down ist, fangen wir oben bei Web-UI (A) an. Diese Komponente wird integriert. Dadurch müssen wir zwei Stubs einfügen (Platzhalter). Einmal für B und einmal für C. Dann gehen wir weiter zu B. Hier wird dann B integriert und es müssen zwei weitere Stubs für D und E eingefügt werden. Danach wird D integriert, welches keine weiteren Auswirkungen hat. Danach wird C integriert, hier würde man nochmal ein Stub für E einfügen, dieser hat aber schon einen Stub, also brauchen wir nicht erneut einen. Zum Schluss wird E integriert, welches das Ende ist und auch keine weiteren Auswirkungen hat. Einen Treiber braucht man hier nicht, da z.B. B hier D und E aufrufen möchte, aber diese noch gar nicht integriert sind, und deshalb durch einen Platzhalter ersetzt werden, die so tun als wären sie z.B. D oder E. Ein Treiber wäre nur nötig wenn man z.B. E testen wollen würde, aber noch niemand der E überhaupt ausführen könnte. Siehe Teilaufgabe b).

b)

#	A	B	C	D	E
1					E
2				D	E
3			C	D	E
4		B	C	D	E
5	A	B	D	C	E

Legende:

Integrierte Komponente

Ausführliche Beschreibung:

Hier fangen wir unten an, nämlich bei E. Wir integrieren E und brauchen für E einen Treiber, da B und C noch nicht existieren und wir aber E aufrufen wollen. Danach inte-

grieren wir D, hierfür brauchen wir auch einen Treiber, da B noch nicht existiert. Danach integrieren wir C, hierfür brauchen wir auch wieder einen Treiber, da A noch nicht existiert. Als nächstes integrieren wir B, hierfür brauchen wir auch einen Treiber, da A noch nicht existiert. Zum Schluss integrieren wir A, welches das Ende ist und keinen Treiber mehr braucht, da es ganz oben in der Hierarchie ist. Stubs werden hier nicht benötigt, da immer die abhängigen Komponenten schon integriert sind, wenn eine Komponente integriert wird.

- c) Wir würden eine Bottom-Up-Strategie (BUS) benutzen. D gilt als technisch anfällig, die BUS beginnt mit den untersten Modulen. Komponente D würde also sehr weit am Anfang getestet werden. Man könnte hier sogar schon in Schritt 1 anfangen D zu testen und danach erst E ausführen. Fehler werden also hier am frühesten gefunden. Bei der Top-Down-Strategie (TDS) wäre D erst sehr spät an der Reihe, was wiederum das Risiko erhöht. Wegen der Testumgebung, also A ist massiv im Verzug, ist es für die TDS die Komponente A zwingend als Startpunkt erforderlich, was schlecht ist, da man ohne A nicht mit der Integration überhaupt beginnen kann. BUS hingegen, braucht A erst sehr spät (nämlich im letzten Schritt). Die Entwicklung und Integration vom Backend kann also einfach weitergehen.

Statische Code-Analyse und Datenflussanomalien

a)

Pfad \ Zeile	$n < 0 : n_1 n_5 n_8 n_{13}$
1	u
5	d
8	d
10	
13	r

Pfad \ Zeile	$n = 0 : n_1 n_8 n_{13}$
1	u
5	
8	d
10	
13	r

Pfad \ Zeile	$n = 1 : n_1 n_8 n_{10} n_{13}$
1	u
5	
8	d
10	r,d
13	r

Pfad \ Zeile	$n = 2 : n_1 n_8 n_{10} n_{10} n_{13}$
1	u
5	
8	d
10	r,d
10	r,d
13	r

Dadurch dass wir hier eine Schleife haben, haben wir uns alle Fälle angesehen und wie oben aufgelistet. Wenn wir jetzt unsere Ergebnisse zusammentragen, kommen wir auf folgende Tabelle, die uns die Analyse für die Variable *ergebnis* zeigt:

Zeile	Aktion auf <i>ergebnis</i>	Status nach Zeile	Anmerkung
START	-	u	Startzustand alle Variablen auf u
1	-	u	<i>ergebnis</i> nicht betroffen. Import von <i>n</i>
2	-	u	Deklaration von <i>ergebnis</i>
3	-	u	<i>ergebnis</i> nicht betroffen
4	-	u	<i>ergebnis</i> nicht betroffen
5	d	d	Erste Definition, <i>ergebnis</i> erhält Wert 0
6	-	d	<i>ergebnis</i> nicht betroffen
7	-	d	<i>ergebnis</i> nicht betroffen
8	d	d	Zweite Definition, <i>ergebnis</i> erhält Wert 1
9	-	d	<i>ergebnis</i> nicht betroffen
10	r, d	d	<i>ergebnis</i> wird gelesen und neu zugewiesen
11	-	d	<i>ergebnis</i> nicht betroffen
12	-	d	<i>ergebnis</i> nicht betroffen
13	r	d	Referenzierung, <i>ergebnis</i> Wert wird gelesen
14	-	d	<i>ergebnis</i> nicht betroffen
END	-	u	Out of scope deshalb <i>ergebnis</i> = u

Anomalie:

Im ersten Pfad also der Pfad für $n < 0$ entsteht durch die Zeilen 5 (erstes *d*) und 8 (zweites *d* ohne ein *r* dazwischen) eine *d* – *d* Anomalie. Beide Zeilen weisen der Variable einen Wert zu, ohne dass diese zwischen den Zeilen referenziert wurden.

- b) Die gefundene *d* – *d* Anomalie aus a) führt für die Berechnung der Fakultät zu einem Bug. In Zeile 5 wird *ergebnis* innerhalb des if-Blocks auf 0 gesetzt. Das ist wahrscheinlich von dem Entwickler so gewollt um negative Inputs so immer auf 0 zu setzen. Jedoch wird in Zeile 8 *ergebnis* bedingungslos auf 1 gesetzt. D.h. der Wert der vorher beabsichtigt vom Entwickler auf 0 gesetzt wurde, wird einfach immer überschrieben. Das führt dazu, dass die vom Entwickler eingebaute Sonderbehandlung der negativen Inputs einfach entfällt. In Zeile 9 ist es auch so dass die while Schleife in dem Case wenn *n* negativ ist, nie betreten wird, da *i* auf 0 gesetzt wird und *n* negativ ist.

Zusatz: Wir denken dass der Entwickler statt *while* in Zeile 9 eine *do while* ausführen wollte, bedeutet die Schleife wird auf jeden Fall einmal ausgeführt und dann in Zeile 10, *ergebnis* 0 gesetzt. Danach würde die *while*-Schleife false berechnen und es würde *ergebnis* = 0 returned werden.

- c) Der Input -1 führt mit der gefundenen *d* – *d* Anomalie aus a) zu einem grundlegenden Fehler. Die Fakultätsfunktion ist nämlich für negative Inputs nicht definiert. In Zeile 4 hat der Entwickler eindeutig eine solche Abfrage eingebaut und auch dann weiter behandelt

in Zeile 5 und diese auf 0 gesetzt. Wie vorher schon beschrieben wird dann in Zeile 8 der Wert für *ergebnis* immer auf 1 gesetzt, was mathematisch und systematisch nicht korrekt ist, da die Schleife für negative Inputs nicht betreten wird. Das Programm ignoriert also seine eigene Logik und Fehlerbehandlung für negative Inputs komplett und gibt immer 1 aus, obwohl die Code-Intention in Zeile 5 0 erwarten würde. Somit ist das Ergebnis für den Input -1 mit der Anomalie 1, was falsch ist. Korrekt wäre hier 0 oder eine Exception, damit man damit richtig weiterarbeiten kann. Das hat vor allem Auswirkungen auf die Zuverlässigkeit und Korrektheit des Programms und auch für die weiterführenden Programme die diese Funktion aufrufen und dann auf einer falschen Grundlage weiterarbeiten. Zudem ist hier das Prinzip des Design by Contract sehr wichtig, denn die volle Verantwortung für die Korrektheit liegt beim Aufrufer. Heißt also wenn wir hier eine Precondition hätten, die sagt dass keine negativen Werte initialisiert werden dürfen, liegt die Verantwortung beim Aufrufer. Wenn diese Bedingung eingehalten wird, wäre dann der Code den der Entwickler geschrieben hat für den Case $n < 0$ gar nicht erreichbar und toter Code.

Code Metriken in der Praxis

a)

```
1      /**
2       * An listener which listens for state changes in any of the
3       * incoming
4       * places. Used to signal when to fire this transition.
5       */
6      private final Observer incomingPlacesObserver = new
7          Observer() {
8
9          /**
10           * (non-Javadoc)
11           *
12           * @see java.util.Observer#update(java.util.Observable,
13           * java.lang.Object)
14           */
15          public void update(final Observable observable, final
16              Object arg) {
17              TokenCountChangedEvent event =
18                  (TokenCountChangedEvent) arg;
19              int newNoOfOccupiedPlaces;
20              if (event.placeBecameEmpty()) {
21                  newNoOfOccupiedPlaces =
22                      noOfOccupiedIncomingPlaces
23                      .decrementAndGet();
24              } else {
25                  if (event.placeBecameOccupied()) {
26                      newNoOfOccupiedPlaces =
27                          noOfOccupiedIncomingPlaces
28                          .incrementAndGet();
29                  } else {
```

```
24         newNoOfOccupiedPlaces =
25             noOfOccupiedIncomingPlaces.get();
26     }
27     unblockIfEnoughTokensAvailable(
28         newNoOfOccupiedPlaces);
29 }
30 };
```

Hier sieht man schnell dass es insgesamt 2 If-Blöcke gibt. Einmal in Zeile 16 und einmal in Zeile 20. Wir Starten mit dem Startwert 1, hier werden jetzt die 2 möglichen Pfade der If-Blöcke hinzugefügt mit jeweils ++. Somit entsteht folgende Berechnung für die zyklomatische Komplexität M mit Entscheidungswegen E :

$$M = E + 1$$

$$M = 2 + 1 = 3$$

```
1  /**
2   * Tries to fire this transition. It blocks until enough
3   * tokens are
4   * available and then tries to grab all locks of all
5   * incoming places. If
6   * successful, checks again whether it is still allowed to
7   * fire. If so, it
8   * fires.
9   * @throws InterruptedException thrown if an interrupt was
10  * requested
11  */
12  private void fire() throws InterruptedException {
13      blockUntilEnoughTokensAvailable();
14      if (lockAndCheckIncomingPlaces()) {
15          setChanged();
16          notifyObservers(new TransitionStartsToFireEvent());
17          for (Place incomingPlace : incomingPlaces) {
18              incomingPlace.decrementTokenCount();
19          }
20          for (Place outgoingPlace : outgoingPlaces) {
21              outgoingPlace.incrementTokenCount();
22          }
23          setChanged();
24          notifyObservers(new TransitionEndsFireEvent());
25          unlockLockedPlaces(incomingPlaces);
26      }
27  }
```

Auch hier sieht man schnell dass es ein If-Block gibt in Zeile 10. In diesem werden dann nochmals zwei for-Schleifen (Zeile 13 und 16) eingebaut was die Komplexität erhöht. Diese muss man beide mit einrechnen also auch jeweils ++. Somit entsteht folgende Berechnung

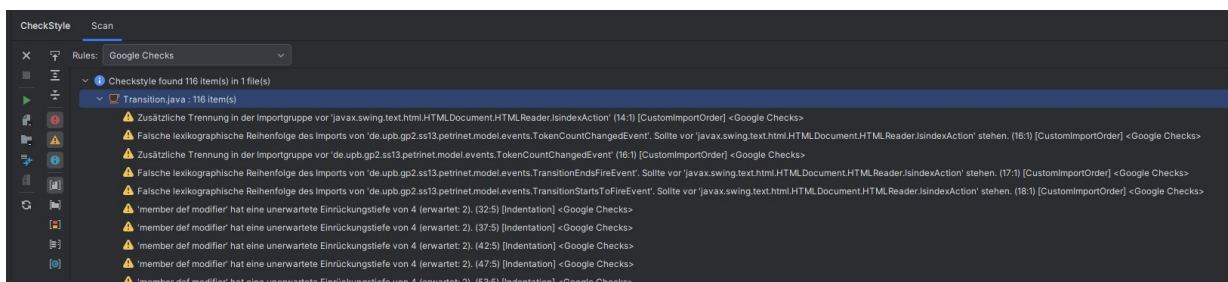
für die zyklomatische Komplexität M mit Entscheidungswegen E :

$$M = E + 1$$

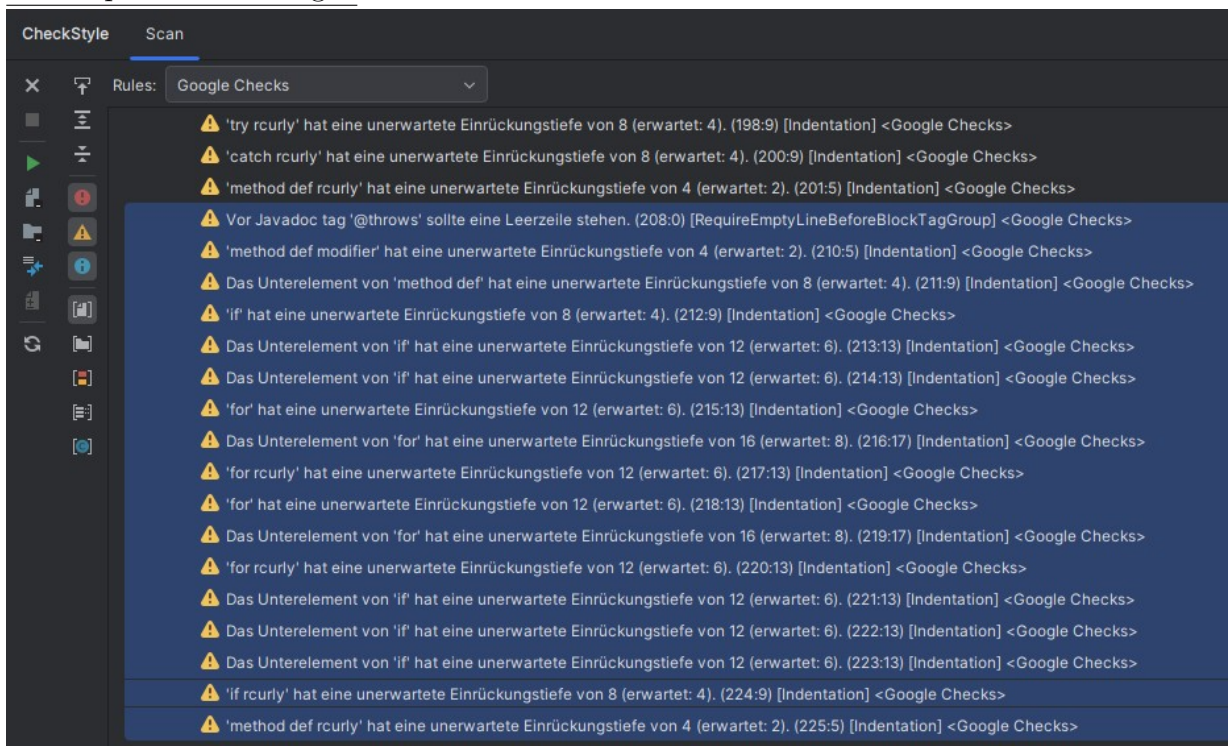
$$M = 3 + 1 = 4$$

- b) Wir haben uns hier für diese Teilaufgabe für das Plugin Checkstyle in IntelliJ entschieden. Zudem haben wir für die Configuration Files Google Checks verwendet wie man im nachfolgenden Screenshot entnehmen kann. Nachdem wir das Checkstyle Plugin laufen lassen haben, sind insgesamt 116 Warnungen aufgetreten. Man kann hier leider nicht nach Metriken und Hotspots sortieren oder filtern, deshalb haben wir uns alle Meldungen angeschaut und die relevantesten für diese Teilaufgabe herausgesucht und zusammengefasst.

Man kann schon mal direkt sagen dass die Code-Qualität nicht im akzeptablen Bereich liegt. Es ist sehr offensichtlich dass so viele Meldungen nicht akzeptabel sind. Es gibt massive Verletzungen der Coding-Standards von Google Checks welche wir verwendet haben. Falsche Einrückung, fehlende Dokumentation und Import-Fehler.



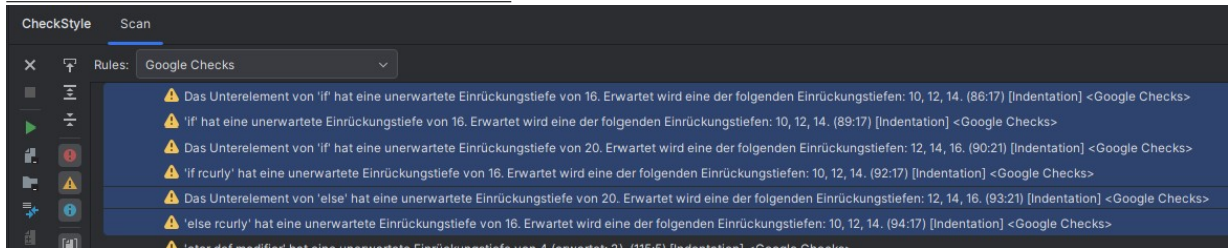
1. Hotspot: Einrückungen



Die Liste zeigt sehr hohe Anzahl an Einrückungstiefen, z.B. in Zeile 212, 214, 215 usw..

Das korreliert natürlich mit der Verschachtelungstiefe und das weist auch auf eine hohe Komplexität hin (Das wir nochmal in Teilaufgabe c) besprochen.) In der Methode `fire()` wird in den zuvor aufgeführten Zeilen massive Indentation Warnungen ausgegeben. Checkstyle erwartet hier eine Einrückung von 4, 6 oder 8 aber findet 8, 12 oder 16 vor. Das ist genau Nesting Depth welche zu hoch ist.

2. Hotspot: Fehlende Dokumentation



Auch hier werden zahlreiche Warnungen bezüglich der Einrückungstiefe ausgegeben. Erwartet 10, 12, 14 und findet 16 oder 20 vor. Auch hier ist die Verschachtelungstiefe zu hoch. Zudem ist eine Tiefe von 20 für eine Methode sehr hoch was genau auf die Verschachtelung von if-else-Blöcken hinweist.

3. Weitere Warnungen:

1. Javadoc-Zusammenfassung fehlt → zeigt dass hier nicht gut dokumentiert wurde wiederum die Wartbarkeit und Lesbarkeit verschlechtert. (Zeile 148 ff.)
2. Vor Javadoc tag '@throws' sollte eine Leerzeile stehen. (Zeile 208)
3. Falsche lexikographische Reihenfolge des Imports (Zeile 17)
4. Zusätzliche Trennung der Importgruppe (Zeile 14)

c) **Maßnahme 1:** Extract Method

Hier sollte eigentlich der Schaltvorgang des Bewegen der Token in einer eigenen privaten Methode ausgelagert werden. Also z.B. `moveTokens()`. `fire()` sollte auch dem Single-Responsibility Principle folgen, sie sollte also den Ablauf steuern also locking → Action → Unlocking machen, aber nicht die Details der Token-Manipulation auch noch selbst implementieren. Das erhöht z.B. die Lesbarkeit und Wartbarkeit des Codes.

Die Methode `fire()` prüft nämlich Bedingungen, benachrichtigt den Observer, iteriert über eingehende Stellen (also Token entfernen), iteriert über ausgehende Stellen (Token hinzufügen) und entsperrt wieder. Also vermischt die Methode `fire()` aktuell die Steuerung des Ablaufs mit der eigentlichen Logik (Token Bewegung) Man müsste also den Code-Block innerhalb der if-Abfrage in Zeile 10 vor allem die beiden for-Schleifen in eine eigene Methode auslagern (z.B `moveTokens()`).

Auswirkungen:

Zyklomatische Komplexität würde hierbei sinken. Beispielsweise von `fire()`, beide for-Schleifen würden hier in eine eigene Methode ausgelagert werden. Wir würden hierbei von unseren ursprünglichen zyklomatischen Wert von 4 auf 2 heruntersetzen. Da hier nur noch neben dem Startwert die if-Bedingung übrig bleibt. Zudem würden die Lines of Code reduziert werden, dadurch würde sich natürlich auch die Lesbarkeit und Wartbar-

keit verbessern.

Maßnahme 2: Logik vereinfachen - Verschachtelung auflösen:

`update()` enthält viele unnötige Verschachtelungen (nested ifs). Würden wir hier "else if" verwenden, könnten wir die Verschachtelung reduzieren. Dies würde die Lesbarkeit und Wartbarkeit verbessern. Verschachtelte ifs sind schlecht zum lesen und schwer zu warten, vor allem wenn es immer mehr werden. Flache Strukturen hingegen sind übersichtlicher und leichter zu verstehen. Was noch besser wäre, wäre dass man die Berechnung von "newNoOfOccupiedPlaces" auch ausgelagert.

Auswirkungen:

Nesting depth: Die maximale Verschachtelungstiefe würde sich reduzieren, wenn wir auslagern. Die Zyklomatische Komplexität würde sich hier auch ebenfalls reduzieren, da weniger Pfade durch die Methode führen. Die Zyklomatische Komplexität von `update()` würde sich von 3 auf 1 reduzieren, da wir nur noch den Startwert hätten, da wir die if und fors ausgelagert hätten.

Was uns auch noch aufgefallen ist, ist dass "incomming" in der Methode `fire()` in Zeile 13 und 14 falsch geschrieben ist. Das kann später Auswirkungen auf die Wartbarkeit haben, da man gegebenenfalls nicht mehr über "ctrl + F" die Variable finden kann. Eine konsistente Benennung ist essenziell für die Wartbarkeit, das hätte zwar aktuell keine Auswirkungen auf die Metriken, aber könnte wie gesagt zu Problemen führen.