

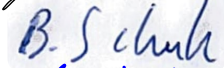
Eidesstattliche Erklärung der Studenten


Hiermit erklären wir an Eides statt, dass wir die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt haben. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder von uns noch von jemand anderem als Prüfungsleistung vorgelegt.

Datum: 07. November 2021

Timo Dohl: 

Maximilian Glöe: 

Benedikt Schuh: 

Mattes Witthöft: 

Inhaltsverzeichnis

1	Installationsanleitung	1
2	Nutzungsanleitung	1
2.1	Erstellen eines Schaltkreises	2
2.2	Einen Schaltkreis Analysieren	3
2.3	Auswertung	4
3	UML-Diagramm	4
4	Wart- und Erweiterbarkeit	5
5	Testszzenarien	6
5.1	Szenario 1:	6
5.2	Szenario 2:	7
6	Selbstreflexion	7
6.1	Timo Dohl	7
6.2	Maximilian Glöe	8
6.3	Benedikt Schuh	8
6.4	Mattes Witthöft	9
7	Kapitelautoren	9
8	Genutzte Hilfsmittel	9
A	Anhang	9
A.1	UML-Diagramm	10
A.2	Java Source Code	10

1 Installationsanleitung

Für die Installation des Glitch Analysierungs Programms wird die JRE (Java Runtime Environment), in mindestens der Version 8, benötigt. Aktuell ist die JRE 8 unter der Website <https://www.java.com/de/download/> zu finden und runterzuladen. Folgen Sie den Installationsanweisungen des Installers.

Um das Programm zu kompilieren und auszuführen wird Apache Maven und das JDK (Java Development Kit), in mindestens der Version 8, benötigt. Apache Maven ist aktuell auf der Webseite <https://maven.apache.org/download.cgi> zu finden und runterzuladen. Achten Sie dabei auf das korrekte Betriebssystem und die Bitarchitektur. Folgen Sie den Installationsanweisungen des Installers. Das JDK ist aktuell auf der Website <https://www.oracle.com/de/java/technologies/javase/javase8u211-later-archive-downloads.html> zu finden und runterzuladen. Achten Sie dabei auf das korrekte Betriebssystem und die Bitarchitektur. Folgen Sie den Installationsanweisungen des Installers. Das JDK ist aktuell auf der Website.

Für das Kompilieren und die Ausführung soll die IDE (Integrated Development Environment) IntelliJ IDEA von JetBrains verwendet werden. Um das Projekt in IntelliJ IDEA zu importieren, klonen Sie zunächst das Projekt aus dem Gitlab der Nordakademie (https://gitlab2.nordakademie.de/TimoDohl-A19/hausarbeit_i143_karow_c9). Wählen Sie dann den Punkt „*File -> New -> Project from Existing Sources*“. Wählen Sie jetzt den Ordner, in welchen Sie das Projekt geklont haben aus und klicken Sie „OK“.

Ggf. muss das Projekt die richtige JDK Version zugewiesen werden. Wählen Sie dafür den Punkt „*File -> Project Structure -> Project*“ aus. Wählen Sie nun unter „*Project SDK*“ die korrekte JDK Version (in diesem Fall 1.8) aus. Unter „*Project language Level*“ muss „*8 – Lambdas, type annotations etc.*“ ausgewählt werden.

2 Nutzungsanleitung

In den folgenden Abschnitten wird erläutert, wie ein Schaltkreis zu erstellen, zu analysieren und auszuwerten ist. Aufgrund der Lesbarkeit und Verständlichkeit wurde sich für eine schrittweise geschriebene Nutzungsanleitung entschieden, die die Schaltung 1 aus der Aufgabenstellung als praktisches Beispiel verwendet.

2.1 Erstellen eines Schaltkreises

1. Eine Java-Klasse erstellen z.B. ExampleCircuit1

```
public class ExampleCircuit1 implements ExampleCircuit{  
    public Circuit create(){  
    }  
}
```

2. Notwendige Imports hinzufügen Im Beispiel sind es: - Alle Komponenten aus dem Package „circuitcomponents“

```
import circuitcomponents.*;
```

3. Eine Create-Methode in der zuvor erstellten Klasse anlegen

```
public Circuit create(){}
```

4. Die Eingänge erstellen, indem für jeden Eingang ein neues Objekt der Klasse LogicalInput mit einem Channel (Integer) als Parameter erstellt wird. Eine Verzögerung lässt sich hier nicht hinzufügen, da ein Eingang logisch gesehen keine Schaltzeit hat. Es lassen sich auch keine Eingänge auf die Eingänge schalten.

```
LogicalInput inputX0 = new LogicalInput(0);  
LogicalInput inputX1 = new LogicalInput(1);  
LogicalInput inputX2 = new LogicalInput(2);
```

5. Jetzt werden alle Komponenten (Gatter) erstellt bzw. angelegt. Jedes Gatter kommt als eigenes Objekt daher wobei jedem Gatter eine Verzögerung für das Verzögerungsverhalten mitgegeben werden kann. Standardmäßig wird eine Verzögerung von eins gesetzt. Eine Verzögerung von null ist hier nicht erlaubt, ebenso eine negative Verzögerung.

```
LogicalAND a1 = new LogicalAND();  
LogicalAND a2 = new LogicalAND();  
LogicalNOT n1 = new LogicalNOT();  
LogicalOR o1 = new LogicalOR();
```

6. Nachdem alle Komponenten angelegt wurden, müssen nun die Entsprechenden Verbindungen zwischen den einzelnen Komponenten gesetzt werden. Also die Ausgänge mit den Eingängen verknüpfen. Dies wird mit dem Aufruf der Methode addInput() auf dem jeweiligen Gatter realisiert. Ein Ausgang als eigene Klasse gibt es nicht. Ein Output wird also durch eine Komponente (Gatter) oder auch als LogicalInput betrachtet. Diese Ausgänge werden den Eingängen der Gatter zugeordnet. Hierbei gilt ein LogicalNOT akzeptiert einen Eingang, ein LogicalAND und ein LogicalOR akzeptieren maximal vier Eingänge. Die Reihenfolge, wie die Inputs den Gattern übergeben werden spielen keine Rolle.

```
o1.addInput(a1, a2);  
a1.addInput(inputX0, inputX1);  
a2.addInput(n1, inputX2);  
n1.addInput(inputX1);
```

7. Als letzten Schritt, bevor das eigentliche Analysieren stattfinden kann muss ein Objekt der Klasse Circuit zurückgegeben. Diesem Objekt wird nun das letzte Gatter der Schaltung (welches den finalen Ausgang ausgibt) als Parameter mit übergeben.

```
return new Circuit(o1);
```

8. Vollständiges Beispiel:

```
import circuitcomponents.*;  
  
public class ExampleCircuit1 implements ExampleCircuit{  
    public Circuit create(){  
        LogicalInput inputX0 = new LogicalInput(0);  
        LogicalInput inputX1 = new LogicalInput(1);  
        LogicalInput inputX2 = new LogicalInput(2);  
  
        LogicalAND a1 = new LogicalAND();  
        LogicalAND a2 = new LogicalAND();  
        LogicalNOT n1 = new LogicalNOT();  
        LogicalOR o1 = new LogicalOR();  
  
        o1.addInput(a1, a2);  
        a1.addInput(inputX0, inputX1);  
        a2.addInput(n1, inputX2);  
        n1.addInput(inputX1);  
  
        return new Circuit(o1);  
    }  
}
```

2.2 Einen Schaltkreis Analysieren

Den zuvor erstellten Schaltkreis zu analysieren ist eine Sache, die Analyse muss natürlich auch entsprechende formatiert ausgegeben werden. In der Main Klasse Simulation ist der Analyzer und der Logger bereits eingerichtet.

1. Ein Objekt der zuvor erstellten Klasse anlegen auf dem die Methode create() aufgerufen wird.

```
Circuit e1 = new ExampleCircuit1().create();
```

2. Einen neuen Analyzer anlegen, mit der entsprechende Klasse GlitchAnalyzer. Diesem beim Erstellen den zuvor angelegt Circuit als Parameter übergeben.

```
GlitchAnalyzer analyzer = new GlitchAnalyzer(e1);
```

3. Einen GlitchLogger initialisieren ohne Parameter.

```
GlitchLogger logger = new ConsoleGlitchLogger();
```

4. Eine passende Ausgabe für die Schaltung, die analysiert werden soll hinzufügen.

```
System.out.println("Simulation_Schaltung_1");
```

5. Die Methode log() auf dem GlitchLogger aufrufen und als Parameter den GlitchAnalyzer mit dem Methodenaufruf analyzeForGlitches() übergeben.

```
logger.log(analyzer.analyzeForGlitches());
```

2.3 Auswertung

Die Ausgabe nach der Ausführung sieht wie folgt aus:

Simulation gestartet.

Glitch bei Starteinstellung 1100 und Änderung an x0

Glitch bei Starteinstellung 1110 und Änderung an x1

Simulation beendet.

Die erste und letzte Zeile geben an, dass Analyse gestartet wurde bzw. dass sie beendet wurde. Jede Zeile dazwischen gibt einen gefundenen Glitch an. Wenn keine Zeilen dazwischenstehen, wurde kein Glitch in der Schaltung gefunden.

Die Zahlenfolge, bestehend aus Einsen und Nullen, gibt an Wie die Eingänge der Schaltung belegt sind bevor der Glitch entsteht. In der Reihenfolge links anfangend mit X0 und nach rechts aufsteigend. Im Beispiel wäre der zweiten Zeile der Ausgabe wäre $x_0 = 1$, $x_1 = 1$, $x_2 = 0$ und $x_3 = 0$. Am Ender der Zeilen ist angegeben welcher Eingang negiert wird damit der Glitch entsteht.

3 UML-Diagramm

Die Klassen des UML-Diagramms (A.1) sind in drei Packages aufgeteilt „circuitcomponents“, „processors“ und „export“. Das Interface Component repräsentiert Komponenten in einer Schaltung wie z.B. ein AND-, XOR- oder OR-Gatter mit ihrer Schaltzeit. Die abstrakte Klasse LogicalComponent implementiert das Interface Component. Die Klassen LogicalNOT, LogicalOR, LogicalAND und LogicalInput erben von der abstrakten Klasse LogicalComponent und implementieren die Berechnung des Zustands. Jede Komponente kennt ihren Input. Die Klasse LogicalInput bietet eine Ausnahme, da diese im Gegensatz zu den anderen Implementationen keinen Input haben kann.

Die Klasse Circuit repräsentiert die Schaltung mit ihren Eingängen und ihrem Zustand. Eine Schaltung kann aus beliebig vielen Komponenten bestehen, hat jedoch nur einen Ausgang. Da jede Komponente ihre Eingänge kennt wird in Circuit nur die letzte Komponente der Schaltung benötigt. Der GlitchAnalyzer im Package „processors“ bekommt eine Schaltung (Circuit) übergeben und analysiert diese auf Glitches.

Das dabei entstandene Ergebnis wird von einem GlitchLogger verarbeitet und entsprechend geloggt. Die einzige Implementation des Interface GlitchLogger ist der ConsoleGlitchLogger. Der eine Ausgabe nach den definierten Anforderungen durchführt.

4 Wart- und Erweiterbarkeit

Um Wart- und Erweiterbarkeit umzusetzen, wurde das Projekt in Packages aufgeteilt. Diese grenzen die funktionalen Bausteine des Projektes ab und können nicht nur klassenweise, sondern komplett ausgetauscht werden.

Um die Erweiterbarkeit in den Packages zu gewährleisten wurden Interfaces und abstrakte Klassen eingeführt. Interfaces werden als Schnittstelle verwendet, um allen sie implementierenden Klassen eine feste Grundstruktur vorzugeben, es entsteht ein Vertrag. Ein Interface beinhaltet in unserem Fall nur Funktionssignaturen und Rückgabetypen. Die Sichtbarkeit ist stets public. In unserem Projekt sind zentrale Interfaces das „Component“ Interface und das „GlitchLogger“ Interface. Das „Component“ Interface gibt einem Component vor, was das Mindestverhalten eines technischen Bauteils sein muss.

In diesem Projekt wurde auf Modularität geachtet, denn alle Programmteile können wiederverwendet, ausgetauscht oder erweitert werden. Die Umsetzung erfolgt durch eine Bottom-up Strategie, da das Problem vom Kleinsten gelöst wird. Konkret werden Schaltungen modular durch ihre Komponenten erstellt, diese wiederum werden dann miteinander verbunden, sodass ein Schaltnetz entsteht. Wenn

eine Schaltung vollständig erzeugt wurde, kann eine Glitchanalyse beginnen. Diese Zergliederung des Problems in kleine Teilstücke erhöht zusätzlich auch die Wartbarkeit, dadurch können Fehler schneller identifiziert und behoben werden.

Wir haben bei der Umsetzung des Projektes auch sehr auf die Erweiterbarkeit geachtet, durch die Abstrakte „LogicalComponent“ Klasse muss der Nutzer für neue elektrische Bauteile nur die konkrete Logik des Berechnens implementieren. Dadurch wird Codeduplizierung vermieden und es macht die Erweiterbarkeit sehr leichtgängig. In der aktuellen Umsetzung wird angenommen, dass Komponenten nur einen Ausgang haben und bis zu 4 Eingänge. Damit auch Bauteile mit mehr Ausgängen realisiert werden können, kann die Implementierung noch um einen Ausgang erweitert werden oder ein komplexes Bauteil wird auf seine Einzelkomponenten reduziert. Als Beispiel kann ein Halbaddierer aus einem XOR und einem AND Baustein erstellt werden.

Auch die Ausgabeform ist sehr schnell erweiterbar. In der aktuellen Umsetzung erfolgt die Berechnung der Glitches und die Ausgabe separat. Dadurch kann die Konsolenausgabe auch durch eine Dateiausgabe oder eine GUI ersetzt werden.

Das Projekt ist auch nicht auf die reine Erkennung von Glitches beschränkt, denn das Schaltnetz existiert unabhängig und kann auf diverse weitere Arten verwendet werden. Zwei Beispiele wären die Analyse der maximalen Schaltzeit oder die Optimierungen innerhalb der Schaltung.

5 Testszenarien

5.1 Szenario 1:

In diesem Szenario wird die Schaltung 1 (Anhang 6.1) in der Software auf Glitches analysiert. Nach eigener Untersuchung der Schaltung mithilfe der Aufgabenbeschreibung unter Einbeziehung einer gleich langen Verzögerung an allen Gattern ist genau ein Glitch möglich. Die zu erwartende Ausgabe der Software lautet wie folgt:

Simulation gestartet

Glitch bei Starteinstellung 111 und Änderung an Input x1

Simulation beendet

5.2 Szenario 2:

In diesem Szenario wird die Schaltung 2 (Anhang 6.2) in der Software auf Glitches analysiert. Nach eigener Untersuchung wie in Szenario 1 sind in dieser Schaltung genau zwei Glitches möglich. Die zu erwartende Ausgabe der Software lautet wie folgt:

Simulation gestartet

Glitch bei Starteinstellung 1100 und Änderung an Input x0

Glitch bei Starteinstellung 1110 und Änderung an Input x1

Simulation beendet

6 Selbstreflexion

6.1 Timo Dohl

Das Projekt lief anfangs relativ schleppend. Es fiel uns deutlich schwerer „Test-First“ zu denken als einfach Implementationen dem Motto „Trial-and-Error“ auszutesten. Besonders mir, da ich zuvor in anderen Projekten kaum Tests geschrieben habe oder Test-First vorgegangen bin.

Nach anfänglichen Schwierigkeiten haben wir dann ein strukturiertes Klassendiagramm mit den nötigen Abhängigkeiten erarbeitet, welche unsere Logik abbilden soll. Dabei haben wir viel Wert auf Erweiterbarkeit gelegt. Anders als die ersten Überlegungen sind z.B. Gatter eigene Klassen geworden mit „doppelter“ Vererbung. Im Source-Code nur ein Interface mehr, bietet Entwicklern die Möglichkeit auch Gatter bzw. logische Komponenten mit völlig eigenen Logiken zu entwickeln, ohne dabei auf die abstrakte Klasse achten zu müssen.

Alles in allem ein gelungenes Projekt mit einem recht sauberen Ergebnis, welches das Problem der Aufgabenstellung löst. Rückblickend kann man jedoch sagen, dass etwas mehr „Verantwortung“ in die Circuit Klasse gekonnt hätte. Das würde das Problem nicht besser lösen, aber eine sehr große Erweiterungsmöglichkeit bieten – zu jeder Zeit und zu jedem Punkt in die Schaltung schauen zu können. Die Implementation der Logik erfordert aber ein höheres Maß an Komplexität bei der einmaligen Entwicklung. Aufgrund des begrenzten Zeitumfangs haben wir uns dagegen entschieden.

6.2 Maximilian Glöe

Wir haben uns bei der Umsetzung des Projektes am Anfang sehr auf die Lösungsfindung orientiert. Wie kann das Problem effizient gelöst werden und wie gehen wir an die einzelnen Herausforderungen heran. Die Implementierung nach dem „Test-Driven-Development“ hat uns anfänglich einige Probleme bereitet, da wir gerade mit dem Umgang mit Mock Objekten wenig Erfahrung hatten.

Nachdem wir dies aber besser verstanden hatten und einige Tests geschrieben hatten, wurde die Umsetzung immer einfacher. Durch die Verwendung von Interfaces und Abstraktion durch eine abstrakte Klasse, konnten wir die Verantwortung der einzelnen Klassen klar kapseln. Dieses strukturierte Vorgehen hat uns das Problem schnell lösen lassen und bietet ein gutes Maß an Erweiterbarkeit.

Im Großen und Ganzen bin ich mit dem Erreichten und auch der Teamleistung sehr zufrieden. Wir haben gut zusammengearbeitet und auch die Vorteile von GIT genutzt. Auch die „Code-with-me“ Funktion von IntelliJ hat Pair-programming noch einfacher gemacht. Dennoch hätte ich rückblickend die Erstellung von Schaltungen gerne über eine Konsoleneingabe oder das Einlesen von JSON Dateien realisiert. Die aktuelle Umsetzung erfordert von dem Nutzer ein hohes Maß an Codewissen und sollte über Dialoge vereinfacht werden. Leider konnten wir dies in der gegebenen Zeit nicht lösen und haben die Idee verworfen, da sie auch nicht Teil des Lösungsprozesses gewesen wäre.

6.3 Benedikt Schuh

Die Hausarbeit hat mir einige Vorteile in den Bereichen Test-First und Mob-Programming deutlich gemacht. Es sind mir jedoch auch manche Nachteile bewusst geworden. Der Anfang fiel mir vergleichsweise schwer, da die Anforderungen zwar gegeben waren, aber noch kein Konkretes Projekt bestand. Ich habe im Unternehmen bisher fast ausschließlich bestehende Projekte erweitert und angepasst. Test-First hat jedoch auch dabei geholfen, die Funktionen auf das nötige zu reduzieren und Fehler isoliert zu betrachten.

Das Mob-Programming hat insofern geholfen, dass man sich gut gegenseitig mit Ideen anregen kann. Jedoch hat das teilweise dazu geführt, dass wir uns zusammen in einer Lösung festgefahren haben, die nicht sinnvoll war. Das hätte wahrscheinlich dadurch unterbunden werden können, dass jemand nicht dabei gewesen wäre und ihm das vorgestellt worden wäre.

Insgesamt bin ich sehr zufrieden mit dem Ergebnis. Während der Entwicklung kamen häufiger Gedanken auf, wie die Software außerhalb der gegebenen Anforderungen sinnvoll erweitert werden könnte. Diese Ideen wurden jedoch nicht umgesetzt, da dies die Komplexität stark erhöht hätte und in dem gegebenen Zeitraum nicht angemessen umsetzbar war.

6.4 Mattes Witthöft

Zu Beginn des Projektes hat es recht viel Zeit gekostet die Struktur der Klassen und somit einen möglichen Lösungsansatz zu finden. Es hat wesentlich mehr Zeit in Anspruch genommen als ich gedacht hätte. Nachdem wir besprochen hatten und das Klassendiagramm erstellt war, ging es an die Umsetzung.

In der Umsetzung fiel es mir zu Anfang nicht leicht sowohl mit Java als auch mit dem OO-Prinzip zu arbeiten, da ich dahingehend leider nicht viel Praxiserfahrung sammeln konnte. Während der Entwicklung habe ich immer mal wieder in die Lehrunterlagen aus den vorherigen Semestern geblickt und meine Kommilitonen gefragt. Durch dieses Vorgehen konnte ich meine Kenntnisse sehr effektiv erweitern bzw. auffrischen.

In der Zusammenarbeit innerhalb der Gruppe, kann ich keine Kritik äußern. Hier hat das Zusammenspiel der unterschiedlichen Erfahrungen und die Aufgabenteilung sehr gut zusammengepasst. Darüber hinaus haben wir oft zu zweit oder dritt gleichzeitig an dem Projekt gearbeitet. Die Stärken des Pair-Programming konnten wir gut für uns nutzen.

Schlussendlich lässt sich meinerseits sagen, dass ich mit dem Ergebnis und dem Lerneffekt dieses Projektes sehr zufrieden bin.

7 Kapitelautoren

Installationsanleitung: Timo Dohl

Nutzungsanleitung: Mattes Witthöft

UML-Diagramm: Benedikt Schuh

Wart- und Erweiterbarkeit: Maximilian Glöe

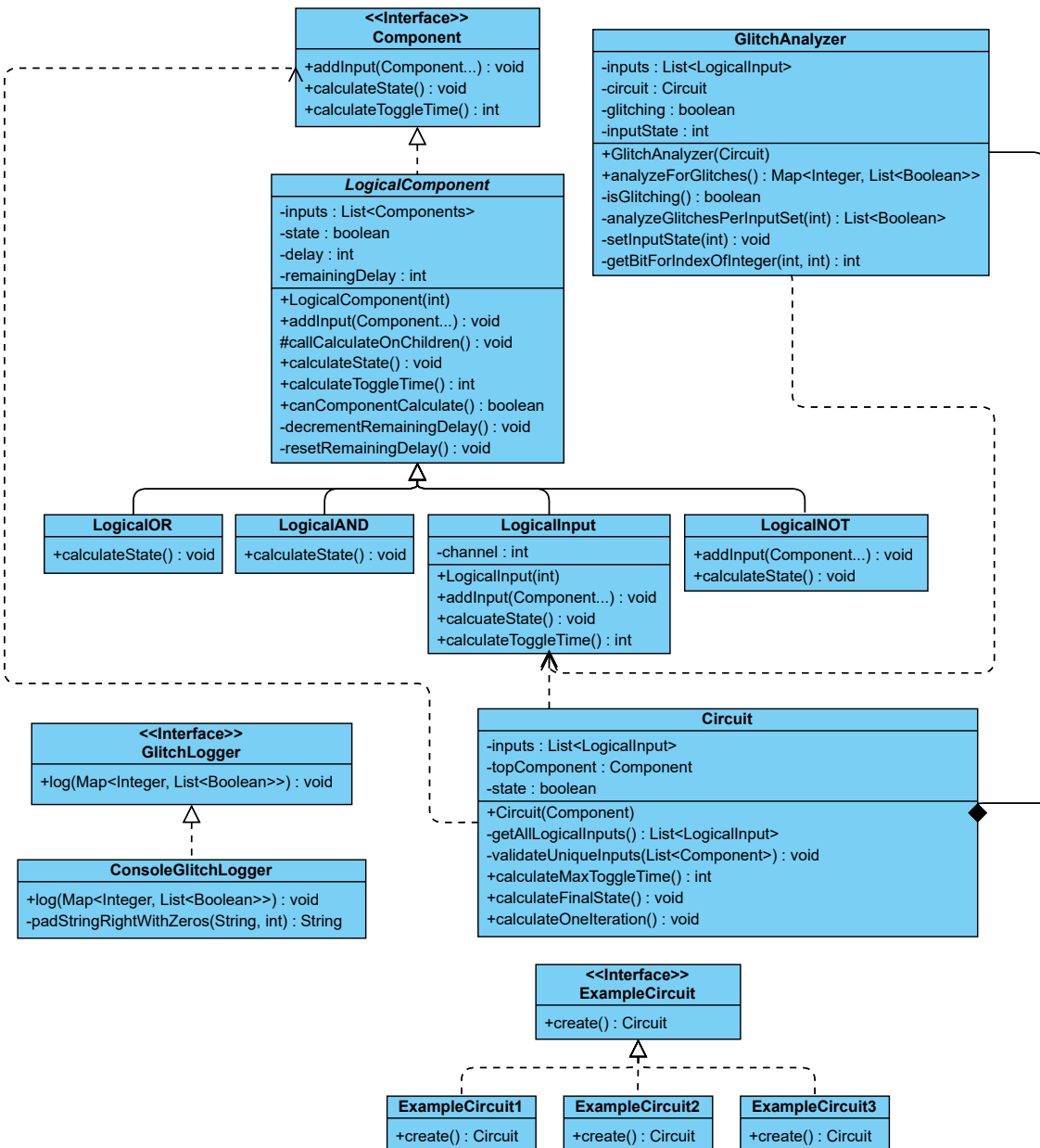
Testzenarien: Timo Dohl

8 Genutzte Hilfsmittel

Vorlesungsinhalte (Folien aus den Vorlesungen und Beispielprojekte)

A Anhang

A.1 UML-Diagramm



A.2 Java Source Code

```
1 import circuitcomponents.Circuit;
2 import examples.ExampleCircuit1;
3 import examples.ExampleCircuit2;
4 import examples.ExampleCircuit3;
5 import export.ConsoleGlitchLogger;
6 import export.GlitchLogger;
7 import processors.GlitchAnalyzer;
8
9 /**
10  * This is the main class to run a simulation.
11  */
12 public class Simulation {
13     public static void main(String[] args) {
14         Circuit e1 = new ExampleCircuit1().create();
15         Circuit e2 = new ExampleCircuit2().create();
16         Circuit e3 = new ExampleCircuit3().create();
17         GlitchAnalyzer analyzer = new GlitchAnalyzer(e1);
18         GlitchLogger logger = new ConsoleGlitchLogger();
19         System.out.println("Simulation Schaltung 1");
20         logger.log(analyzer.analyzeForGlitches());
21         analyzer = new GlitchAnalyzer(e2);
22         System.out.println("Simulation Schaltung 2");
23         logger.log(analyzer.analyzeForGlitches());
24         analyzer = new GlitchAnalyzer(e3);
25         System.out.println("Simulation Schaltung 3");
26         logger.log(analyzer.analyzeForGlitches());
27     }
28 }
```

```
1 package export;
2
3 import java.util.List;
4 import java.util.Map;
5
6 public interface GlitchLogger {
7     void log(Map<Integer, List<Boolean>> glitches);
8 }
9
10
```

```

1 package export;
2
3 import java.util.List;
4 import java.util.Map;
5
6 /**
7  * The ConsoleGlitchLogger is an output implementation of the @GlitchLogger
8  * interface.
9  * It is given a Map filled with all possible glitches when changing the specific
10 * input-set in a circuit.
11 * The ConsoleGlitchLogger does not calculate these glitches, it is only for user
12 * feedback.
13 */
14 public class ConsoleGlitchLogger implements GlitchLogger {
15     public void log(Map<Integer, List<Boolean>> glitches) {
16         boolean glitchFound = false;
17         System.out.println("Simulation gestartet.");
18         for (Map.Entry<Integer, List<Boolean>> pair : glitches.entrySet()) {
19             List<Boolean> glitchesForInputSet = pair.getValue();
20             int numberOfInputs = glitchesForInputSet.size();
21
22             for (int i = 0; i < numberOfInputs; i++) {
23                 if (glitchesForInputSet.get(i)) {
24                     glitchFound = true;
25                     String state = padStringRightWithZeros(Integer.toBinaryString
26 (pair.getKey()), numberOfInputs);
27                     System.out.printf("Glitch bei Starteinstellung %s und
28 Änderung an x%d %n", state, i);
29                 }
30             }
31         }
32         if (!glitchFound){
33             System.out.println("Es wurden keine Glitches in dieser Schaltung
34 gefunden.");
35         }
36         System.out.println("Simulation beendet.");
37     }
38
39     private String padStringRightWithZeros(String s, int n) {
40         return String.format("%-" + n + "s", s).replace(" ", "0");
41     }
42 }

```

```
1 package examples;
2
3 import circuitcomponents.Circuit;
4
5 public interface ExampleCircuit {
6     Circuit create();
7 }
8
```



```
1 package examples;
2
3 import circuitcomponents.*;
4
5 /**
6  * The example circuit 1 from the task description
7  */
8 public class ExampleCircuit1 implements ExampleCircuit{
9     public Circuit create(){
10         LogicalInput inputX0 = new LogicalInput(0);
11         LogicalInput inputX1 = new LogicalInput(1);
12         LogicalInput inputX2 = new LogicalInput(2);
13
14         LogicalAND a1 = new LogicalAND();
15         LogicalAND a2 = new LogicalAND();
16         LogicalNOT n1 = new LogicalNOT();
17         LogicalOR o1 = new LogicalOR();
18
19         o1.addInput(a1, a2);
20         a1.addInput(inputX0, inputX1);
21         a2.addInput(n1, inputX2);
22         n1.addInput(inputX1);
23
24         return new Circuit(o1);
25     }
26 }
27
```

```
1 package examples;
2
3 import circuitcomponents.*;
4
5 /**
6  * The example circuit 2 from the task description
7  */
8 public class ExampleCircuit2 implements ExampleCircuit{
9
10     public Circuit create(){
11         LogicalInput inputX0 = new LogicalInput(0);
12         LogicalInput inputX1 = new LogicalInput(1);
13         LogicalInput inputX2 = new LogicalInput(2);
14         LogicalInput inputX3 = new LogicalInput(3);
15
16         LogicalAND a1 = new LogicalAND();
17         LogicalAND a2 = new LogicalAND();
18         LogicalAND a3 = new LogicalAND();
19         LogicalNOT n0 = new LogicalNOT();
20         LogicalNOT n1 = new LogicalNOT();
21         LogicalNOT n2 = new LogicalNOT();
22         LogicalNOT n3 = new LogicalNOT();
23         LogicalOR o1 = new LogicalOR();
24
25         o1.addInput(a1, a2, a3);
26         a1.addInput(inputX0, n1, inputX2);
27         a2.addInput(inputX0, inputX1, n3);
28         a3.addInput(n0, inputX1, n2);
29         n0.addInput(inputX0);
30         n1.addInput(inputX1);
31         n2.addInput(inputX2);
32         n3.addInput(inputX3);
33
34         return new Circuit(o1);
35     }
36 }
37
```

```
1 package examples;
2
3 import circuitcomponents.Circuit;
4 import circuitcomponents.LogicalAND;
5 import circuitcomponents.LogicalInput;
6 import circuitcomponents.LogicalOR;
7
8 /**
9  * Based on the example circuit 1 from the task description but without the NOT
10  * Gate
11  */
12 public class ExampleCircuit3 implements ExampleCircuit{
13     public Circuit create(){
14         LogicalInput inputX0 = new LogicalInput(0);
15         LogicalInput inputX1 = new LogicalInput(1);
16         LogicalInput inputX2 = new LogicalInput(2);
17
18         LogicalAND a1 = new LogicalAND();
19         LogicalAND a2 = new LogicalAND();
20         LogicalOR o1 = new LogicalOR();
21
22         o1.addInput(a1, a2);
23         a1.addInput(inputX0, inputX1);
24         a2.addInput(inputX1, inputX2);
25
26         return new Circuit(o1);
27     }
28 }
29
```

```
1 package examples;
2
3 import circuitcomponents.Circuit;
4 import circuitcomponents.LogicalInput;
5 import circuitcomponents.LogicalNOT;
6
7 /**
8  * A simple circuit with a topComponent without inputs
9  */
10 public class ExampleCircuit4 implements ExampleCircuit{
11     @Override
12     public Circuit create() {
13         LogicalInput inputX0 = new LogicalInput(0);
14         LogicalInput inputX1 = new LogicalInput(1);
15         LogicalInput inputX2 = new LogicalInput(2);
16         LogicalNOT n1 = new LogicalNOT();
17         return new Circuit(n1);
18     }
19 }
20
```

```
1 package exceptions;
2
3 public class NoZeroDelayException extends RuntimeException{
4     private static final long serialVersionUID = 3300192216411898779L;
5 }
6
```

```
1 package exceptions;
2
3 public class MissingInputException extends RuntimeException{
4     private static final long serialVersionUID = 6397217998707865963L;
5
6     public MissingInputException(String message) {
7         super(message);
8     }
9
10    public MissingInputException() {
11
12    }
13 }
14
```

```
1 package exceptions;
2
3 public class TooManyInputsException extends RuntimeException {
4     private static final long serialVersionUID = 61637881287801682L;
5
6     public TooManyInputsException(String message) {
7         super(message);
8     }
9 }
10
```

```
1 package exceptions;
2
3 public class MissingCircuitException extends RuntimeException{
4     private static final long serialVersionUID = -5831970283099106013L;
5
6     public MissingCircuitException(String message) {
7         super(message);
8     }
9
10 }
11
```



```
1 package exceptions;
2
3 public class NoInputsAllowedForInputException extends RuntimeException {
4     private static final long serialVersionUID = -477562218360795765L;
5 }
6
```

```
1 package exceptions;
2
3 public class DuplicateInputChannelDetectedException extends RuntimeException {
4     private static final long serialVersionUID = -7632207422147304556L;
5
6     public DuplicateInputChannelDetectedException() {
7     }
8
9     public DuplicateInputChannelDetectedException(String message) {
10         super(message);
11     }
12 }
13
```

```

1 package processors;
2
3 import circuitcomponents.Circuit;
4 import circuitcomponents.LogicalInput;
5 import exceptions.MissingCircuitException;
6
7 import java.util.ArrayList;
8 import java.util.HashMap;
9 import java.util.List;
10 import java.util.Map;
11
12 /**
13  * The GlitchAnalyzer (GA) takes a circuit and if it is a valid circuit it
14  * analyzes for glitches.
15  * It leads to a glitch if the output state of a circuit changes more than 1 time
16  * when only one input is flipped once.
17  * To analyze the complete circuit for glitches it takes exactly (2^(number of
18  * inputs)) base calculation steps.
19  * Within every step we need to flip every input once.
20  * The maxcalculationtime is the time it takes max to iterate once through the
21  * circuit.
22  * To simulate a clock for every inputstate you need to iterate the number of
23  * maxcalculationtime and check if the state is changed more than once.
24  *
25  * The interesting result is within the return of the analyzeForGlitches() Method
26  * .
27  * It contains a representation of Truth Table for input xn, ..., x1, x0 but
28  * filled with true if glitching in this constellation or false if not.
29  */
30 public class GlitchAnalyzer {
31     private Circuit circuit;
32     private List<LogicalInput> inputs;
33
34     public GlitchAnalyzer(Circuit circuit) {
35         if (circuit == null) throw new MissingCircuitException("No circuit set
36 for GlitchAnalyzing");
37         this.circuit = circuit;
38         this.inputs = circuit.getInputs();
39     }
40
41     public Map<Integer, List<Boolean>> analyzeForGlitches() {
42         Map<Integer, List<Boolean>> glitches = new HashMap<>();
43         for (int i = 0; i < Math.pow(2, inputs.size()); i++) {
44             glitches.put(i, analyzeGlitchesPerInputSet(i));
45         }
46         return glitches;
47     }
48
49     private List<Boolean> analyzeGlitchesPerInputSet(int i) {
50         List<Boolean> glitches = new ArrayList<>();
51         for (LogicalInput input : inputs) {
52             setInputState(i);
53             circuit.calculateFinalState();
54             input.setState(!input.getState());
55         }
56     }
57
58     private void setInputState(int i) {
59         // Implementation of setInputState
60     }
61 }

```

```
47         glitches.add(isGlitching());
48     }
49     return glitches;
50 }
51
52 private boolean isGlitching() {
53     int stateChanges = 0;
54     boolean oldState;
55
56     for (int i = 0; i < circuit.calculateMaxToggleTime(); i++) {
57         if (stateChanges >= 2) break;
58         oldState = circuit.getState();
59         circuit.calculateOneIteration();
60         if (oldState != circuit.getState()) stateChanges++;
61     }
62     return stateChanges >= 2;
63 }
64
65 private void setInputState(int i) {
66     for (LogicalInput input : inputs) {
67         input.setState(getBitForIndexOfInteger(i, inputs.indexOf(input)) !=
68 0);
69     }
70
71 private int getBitForIndexOfInteger(int n, int index) {
72     return (n >> index) & 1;
73 }
74 }
75
```

```

1 package circuitcomponents;
2
3 import exceptions.DuplicateInputChannelDetectedException;
4 import exceptions.MissingInputException;
5
6 import java.util.*;
7 import java.util.stream.Collectors;
8
9 /**
10  * Circuit represents a physical logic circuit.
11  * It has a topNode which is the last logicalNode of the circuit and the logical
12  * Inputs.
13  * It is able to calculate it's state based on the input states either
14  * in multiple steps (for analyzing glitches) or in one step.
15  */
16 public class Circuit {
17     private List<LogicalInput> inputs;
18     private Component topComponent;
19     private boolean state;
20
21     public Circuit(Component topComponent) {
22         this.topComponent = topComponent;
23         if (this.topComponent.getInputs().size() <= 0) {
24             throw new MissingInputException("No inputs set for topComponent");
25         }
26         validateUniqueInputs(topComponent.getInputs());
27         this.inputs = getAllLogicalInputs();
28     }
29
30     private List<LogicalInput> getAllLogicalInputs() {
31         List<Component> components = this.topComponent.getInputs();
32         List<Component> newFoundComponents = new ArrayList<>();
33         List<Component> toRemoveComponents = new ArrayList<>();
34         List<LogicalInput> usedInputs = new ArrayList<>();
35         boolean allFound = false;
36         while (!allFound) {
37             for (Component component : components) {
38                 if (component instanceof LogicalInput) {
39                     usedInputs.add((LogicalInput) component);
40                 } else {
41                     newFoundComponents.addAll(component.getInputs());
42                 }
43                 toRemoveComponents.add(component);
44             }
45             newFoundComponents.removeAll(toRemoveComponents);
46             components = new ArrayList<>(newFoundComponents);
47             if (components.size() == 0) {
48                 allFound = true;
49             }
50         }
51         return usedInputs.stream()
52             .distinct()
53             .sorted(Comparator.comparing(LogicalInput::getChannel))
54             .collect(Collectors.toList());

```

```
54     }
55
56     private void validateUniqueInputs(List<Component> inputs) {
57         long numberOfUniqueInputs = inputs.stream().distinct().count();
58         if (numberOfUniqueInputs != inputs.size()) {
59             throw new DuplicateInputChannelDetectedException("InputChannels are
unique, please check your used channels");
60         }
61     }
62
63     public int calculateMaxToggleTime() {
64         if (Objects.isNull(topComponent)) return 0;
65         return topComponent.calculateToggleTime();
66     }
67
68     public List<LogicalInput> getInputs() {
69         return this.inputs;
70     }
71
72     public boolean getState() {
73         return this.state;
74     }
75
76     public void calculateFinalState() {
77         for (int i = 0; i < calculateMaxToggleTime(); i++) {
78             calculateOneIteration();
79         }
80     }
81
82     public void calculateOneIteration() {
83         if (!Objects.isNull(topComponent)) {
84             topComponent.calculateState();
85             this.state = topComponent.getState();
86         }
87     }
88 }
89
```

```
1 package circuitcomponents;
2
3 import exceptions.MissingInputException;
4
5 import java.util.List;
6
7 public interface Component {
8
9     List<Component> getInputs();
10
11     void addInput(Component... component);
12
13     void calculateState() throws MissingInputException;
14
15     boolean canComponentCalculate();
16
17     int calculateToggleTime();
18
19     boolean getState();
20
21     void setState(boolean state);
22
23 }
24
```

```
1 package circuitcomponents;
2
3 import exceptions.MissingInputException;
4
5 import java.util.List;
6
7 /**
8  * LogicalOR represents the physical logic gate OR.
9  * It takes up to 4 inputs and only outputs true when at least one is true.
10 */
11 public class LogicalOR extends LogicalComponent {
12
13     public LogicalOR() {
14         super();
15     }
16
17     public LogicalOR(int delay) {
18         super(delay);
19     }
20
21     @Override
22     public void calculateState() throws MissingInputException {
23         List<Component> inputs = getInputs();
24         if (inputs.isEmpty()) throw new MissingInputException();
25         if (canComponentCalculate()) {
26             boolean result = false;
27             for (Component input : inputs) {
28                 result = result || input.getState();
29             }
30             setState(result);
31         }
32         callCalculateOnChildren();
33     }
34 }
35
```



```
1 package circuitcomponents;
2
3 import exceptions.MissingInputException;
4
5 import java.util.List;
6
7 /**
8  * LogicalAND represents the logic gate AND.
9  * It takes any number of inputs and only outputs true when all of them are true.
10 */
11 public class LogicalAND extends LogicalComponent {
12
13
14     public LogicalAND() {
15         super();
16     }
17
18     public LogicalAND(int delay) {
19         super(delay);
20     }
21
22     @Override
23     public void calculateState() throws MissingInputException {
24         List<Component> inputs = getInputs();
25         if (inputs.isEmpty()) throw new MissingInputException();
26         if (canComponentCalculate()) {
27             boolean result = true;
28             for (Component input : inputs) {
29                 result = result && input.getState();
30             }
31             setState(result);
32         }
33         callCalculateOnChildren();
34     }
35 }
36
```

```
1 package circuitcomponents;
2
3 import exceptions.MissingInputException;
4 import exceptions.TooManyInputsException;
5
6 import java.util.Collections;
7 import java.util.List;
8
9 /**
10  * LogicalNOT represents the physical logic gate NOT.
11  * It receives one input and outputs the opposite boolean state of it.
12  * If attached to multiple inputs an exception will be thrown.
13  * It can only have one input
14  */
15 public class LogicalNOT extends LogicalComponent {
16
17     public LogicalNOT() {
18         super();
19     }
20
21     public LogicalNOT(int delay) {
22         super(delay);
23     }
24
25     @Override
26     public void calculateState() {
27         List<Component> inputs = getInputs();
28         if (inputs.isEmpty()) throw new MissingInputException();
29         if (canComponentCalculate()) {
30             Component input = inputs.get(0);
31             boolean result = !input.getState();
32             setState(result);
33         }
34         callCalculateOnChildren();
35     }
36
37     @Override
38     public void addInput(Component... component) {
39         List<Component> inputs = getInputs();
40         if (component.length <= 0)
41             throw new MissingInputException("No Input to add");
42
43         if (inputs.size() == 1 || component.length > 1)
44             throw new TooManyInputsException("A LogicalNOT can only handle 1
45 input");
46         Collections.addAll(getInputs(), component);
47     }
48 }
49
```

```
1 package circuitcomponents;
2
3 import exceptions.NoInputsAllowedForInputException;
4
5 /**
6  * LogicalInput represents a physical Input to a Circuit.
7  * It does not implement any logic or has any behaviour.
8  * Not tested, because of missing testable logic.
9  * It cannot have any inputs
10 * The channel is representing the integer of an LogicalInput e.g. the input x2
    has channel 2 and so on
11 */
12 public class LogicalInput extends LogicalComponent {
13     private int channel;
14
15     public LogicalInput(int channel) {
16         this.channel = channel;
17     }
18
19     public int getChannel() {
20         return channel;
21     }
22
23     @Override
24     public void addInput(Component... component) {
25         throw new NoInputsAllowedForInputException();
26     }
27
28     @Override
29     public void calculateState() {
30     }
31
32     @Override
33     public int calculateToggleTime() {
34         return 0;
35     }
36 }
37
```

```

1 package circuitcomponents;
2
3 import exceptions.MissingInputException;
4 import exceptions.NoZeroDelayException;
5 import exceptions.TooManyInputsException;
6
7 import java.util.ArrayList;
8 import java.util.Collections;
9 import java.util.List;
10
11 /**
12  * LogicalComponent is a abstract class which represents any logical component
13  * in a circuit.
14  * It has a state, inputs and a delay (physical delay)
15  * It can calculate the toggleTime: toggleTime of inputs + delay
16  * It can have up to 4 inputs
17  */
18 public abstract class LogicalComponent implements Component {
19     private List<Component> inputs = new ArrayList<>();
20     private boolean state = false;
21     private int delay;
22     private int remainingDelay;
23
24     public LogicalComponent() {
25         this.delay = 1;
26         resetRemainingDelay();
27     }
28
29     public LogicalComponent(int delay) {
30         if (delay <= 0) throw new NoZeroDelayException();
31         this.delay = delay;
32         resetRemainingDelay();
33     }
34
35     private void resetRemainingDelay(){
36         this.remainingDelay = this.delay;
37     }
38
39     private void decrementRemainingDelay(){
40         this.remainingDelay--;
41     }
42
43     public boolean canComponentCalculate(){
44         decrementRemainingDelay();
45         if (this.remainingDelay != 0) return false;
46         resetRemainingDelay();
47         return true;
48     }
49
50     public void addInput(Component... component) {
51         if (component.length <= 0)
52             throw new MissingInputException("No Input to add");
53         if (this.inputs.size() + component.length > 4)
54             throw new TooManyInputsException("LogicalComponents only allow up to

```

```
54 4 inputs, cascade the component if you need more.");
55     Collections.addAll(this.inputs, component);
56 }
57
58 public abstract void calculateState() throws MissingInputException;
59
60
61 public int calculateToggleTime() {
62     List<Component> inputs = getInputs();
63     int maxTreeSize = 0;
64     for (Component input : inputs) {
65         maxTreeSize = Math.max(maxTreeSize, input.calculateToggleTime());
66     }
67
68     return maxTreeSize + delay;
69 }
70
71 void callCalculateOnChildren() {
72     for (Component input : inputs) {
73         input.calculateState();
74     }
75 }
76
77 public List<Component> getInputs() {
78     return inputs;
79 }
80
81 public boolean getState() {
82     return state;
83 }
84
85 public void setState(boolean state) {
86     this.state = state;
87 }
88
89 }
```

```

1  import circuitcomponents.Circuit;
2  import examples.ExampleCircuit1;
3  import examples.ExampleCircuit2;
4  import examples.ExampleCircuit3;
5  import examples.ExampleCircuit4;
6  import exceptions.MissingCircuitException;
7  import exceptions.MissingInputException;
8  import export.ConsoleGlitchLogger;
9  import export.GlitchLogger;
10 import org.junit.After;
11 import org.junit.Before;
12 import org.junit.Test;
13 import processors.GlitchAnalyzer;
14
15 import java.io.ByteArrayOutputStream;
16 import java.io.PrintStream;
17
18 import static org.junit.Assert.assertThrows;
19 import static org.junit.Assert.assertTrue;
20
21 /**
22  * Testing the real circuits. In this part we dont need any mocked data.
23  * If the circuit creation is changed the output should never differ.
24  */
25 public class IntegrationTest {
26     private final ByteArrayOutputStream outContent = new ByteArrayOutputStream();
27     private final ByteArrayOutputStream errContent = new ByteArrayOutputStream();
28     private final PrintStream originalOut = System.out;
29     private final PrintStream originalErr = System.err;
30
31     @Before
32     public void setUpStreams() {
33         System.setOut(new PrintStream(outContent));
34         System.setErr(new PrintStream(errContent));
35     }
36
37     @After
38     public void restoreStreams() {
39         System.setOut(originalOut);
40         System.setErr(originalErr);
41     }
42
43     @Test
44     public void testExampleCircuit1() {
45         Circuit circuit = new ExampleCircuit1().create();
46         GlitchAnalyzer analyzer = new GlitchAnalyzer(circuit);
47         GlitchLogger logger = new ConsoleGlitchLogger();
48         logger.log(analyzer.analyzeForGlitches());
49         assertTrue(outContent.toString().contains("Glitch bei Starteinstellung
111 und Änderung an x1"));
50     }
51
52     @Test
53     public void testExampleCircuit2(){

```

```
54     Circuit circuit = new ExampleCircuit2().create();
55     GlitchAnalyzer analyzer = new GlitchAnalyzer(circuit);
56     GlitchLogger logger = new ConsoleGlitchLogger();
57     logger.log(analyzer.analyzeForGlitches());
58     assertTrue(outContent.toString().contains("Glitch bei Starteinstellung
1100 und Änderung an x0"));
59     assertTrue(outContent.toString().contains("Glitch bei Starteinstellung
1110 und Änderung an x1"));
60 }
61
62 @Test
63 public void testExampleCircuit3(){
64     Circuit circuit = new ExampleCircuit3().create();
65     GlitchAnalyzer analyzer = new GlitchAnalyzer(circuit);
66     GlitchLogger logger = new ConsoleGlitchLogger();
67     logger.log(analyzer.analyzeForGlitches());
68     assertTrue(outContent.toString().contains("Es wurden keine Glitches in
dieser Schaltung gefunden."));
69 }
70
71 @Test(expected = MissingCircuitException.class)
72 public void testExampleCircuitIsNull(){
73     GlitchAnalyzer analyzer = new GlitchAnalyzer(null);
74     GlitchLogger logger = new ConsoleGlitchLogger();
75     assertThrows(MissingCircuitException.class, () -> logger.log(analyzer.
analyzeForGlitches()));
76 }
77
78 @Test(expected = MissingInputException.class)
79 public void testExampleCircuit4(){
80     Circuit circuit = new ExampleCircuit4().create();
81     GlitchAnalyzer analyzer = new GlitchAnalyzer(circuit);
82     GlitchLogger logger = new ConsoleGlitchLogger();
83     assertThrows(MissingInputException.class, () -> logger.log(analyzer.
analyzeForGlitches()));
84 }
85 }
86
```

```

1 package processors;
2
3 import circuitcomponents.Circuit;
4 import circuitcomponents.LogicalInput;
5 import exceptions.MissingCircuitException;
6 import org.junit.Before;
7 import org.junit.Test;
8
9 import java.util.*;
10
11 import static org.junit.Assert.*;
12 import static org.mockito.Mockito.mock;
13 import static org.mockito.Mockito.when;
14
15 public class GlitchAnalyzerTest {
16
17     private GlitchAnalyzer glitchAnalyzer;
18     private LogicalInput inputX0;
19     private LogicalInput inputX1;
20     private LogicalInput inputX2;
21     private Circuit circuit;
22
23     @Before
24     public void setUp() {
25         circuit = mock(Circuit.class);
26         inputX0 = mock(LogicalInput.class);
27         inputX1 = mock(LogicalInput.class);
28         inputX2 = mock(LogicalInput.class);
29     }
30
31     @Test
32     public void analyzeForGlitchesWithOneGlitch() {
33         List<LogicalInput> inputs = new ArrayList<>();
34         inputs.add(inputX0);
35         inputs.add(inputX1);
36         inputs.add(inputX2);
37
38         when(circuit.getInputs()).thenReturn(inputs);
39         when(circuit.getState()).thenReturn(false, false, false, false, false,
40 false, false, false,
41 false, false, false, false, false, false, false, false, true,
42 true, true, false,
43 false, false, false, false, false, false, false, false, true,
44 true, true, false,
45 false, false, true, true, true, false, false, false, true, true,
46 true, false,
47 false, false, false, false, false, false, false, false, false,
48 false, false, true,
49 true, true, false, false, false, true, true, true, false, false,
50 false, true,
51 true, true, true, true, true, false, true, true, true, true, true
52 , true,
53 true, true, true, false, false, false, true, true, false, false,
54 false, true,

```



```

47         true, true, true, true, true, true, true, true, true, true, true
    , false,
48         true, true, false, false, false, true, false, false, true, true
    , true, false,
49         false, false, false, true, true, true, false, false, false,
    false, false, true,
50         true, true, false, false, false, true, true, true, false, true,
    true, true, true, true, true, true);
51     when(circuit.calculateMaxToggleTime()).thenReturn(3);
52
53     glitchAnalyzer = new GlitchAnalyzer(circuit);
54
55     Map<Integer, List<Boolean>> expectedGlitches = new HashMap<>();
56     List<Boolean> lFalseFalseFalse = Arrays.asList(false, false, false);
57     List<Boolean> lFalseTrueFalse = Arrays.asList(false, false, false);
58
59     expectedGlitches.put(0, lFalseFalseFalse);
60     expectedGlitches.put(1, lFalseFalseFalse);
61     expectedGlitches.put(2, lFalseFalseFalse);
62     expectedGlitches.put(3, lFalseFalseFalse);
63     expectedGlitches.put(4, lFalseFalseFalse);
64     expectedGlitches.put(5, lFalseFalseFalse);
65     expectedGlitches.put(6, lFalseFalseFalse);
66     expectedGlitches.put(7, lFalseTrueFalse);
67
68     assertEquals(expectedGlitches.size(), glitchAnalyzer.analyzeForGlitches
    ().size());
69     assertTrue(glitchAnalyzer.analyzeForGlitches().entrySet().stream().
    allMatch(e -> e.getValue().equals(expectedGlitches.get(e.getKey()))));
70 }
71
72 @Test
73 public void analyzeForGlitchesWithZeroGlitch() {
74     List<LogicalInput> inputs = new ArrayList<>();
75     inputs.add(inputX0);
76     inputs.add(inputX1);
77     inputs.add(inputX2);
78
79     when(circuit.getInputs()).thenReturn(inputs);
80     when(circuit.getState()).thenReturn(false, false, false, false, false,
    false, false, false,
81         false, false, false, false, false, false, false, false,
82         false, false, false, true, true, false, false, false,
83         false, false, false, true, true, false, false, false,
84         false, false, false, true, true, true, true, false,
85         false, true, true, false, false, true, true, true,
86         true, false, false, false, false, false, false, true,
87         true, false, false, false, false, false, false, false,
88         false, false, false, true, true, false, false, false,
89         false, true, true, true, true, true, true, false,
90         false, true, true, false, false, true, true, true,
91         true, true, true, false, false, true, true, true);
92     when(circuit.calculateMaxToggleTime()).thenReturn(2);
93

```

```
94     glitchAnalyzer = new GlitchAnalyzer(circuit);
95
96     Map<Integer, List<Boolean>> expectedGlitches = new HashMap<>();
97     List<Boolean> lFalseFalseFalse = Arrays.asList(false, false, false);
98
99     expectedGlitches.put(0, lFalseFalseFalse);
100    expectedGlitches.put(1, lFalseFalseFalse);
101    expectedGlitches.put(2, lFalseFalseFalse);
102    expectedGlitches.put(3, lFalseFalseFalse);
103    expectedGlitches.put(4, lFalseFalseFalse);
104    expectedGlitches.put(5, lFalseFalseFalse);
105    expectedGlitches.put(6, lFalseFalseFalse);
106    expectedGlitches.put(7, lFalseFalseFalse);
107
108    assertEquals(expectedGlitches.size(), glitchAnalyzer.analyzeForGlitches
109    ().size());
110    assertTrue(glitchAnalyzer.analyzeForGlitches().entrySet().stream().
111    allMatch(e -> e.getValue().equals(expectedGlitches.get(e.getKey()))));
112    }
113
114    @Test(expected = MissingCircuitException.class)
115    public void testCircuitIsNull(){
116        glitchAnalyzer = new GlitchAnalyzer(null);
117        assertThrows(MissingCircuitException.class, () -> glitchAnalyzer.
118        analyzeForGlitches());
119    }
120 }
```

```

1 package circuitcomponents;
2
3 import exceptions.DuplicateInputChannelDetectedException;
4 import org.junit.Before;
5 import org.junit.Test;
6
7 import java.util.Arrays;
8
9 import static org.junit.Assert.*;
10 import static org.mockito.Mockito.mock;
11 import static org.mockito.Mockito.when;
12
13 public class CircuitTest {
14     private Circuit circuit;
15     private LogicalInput inputX1;
16     private LogicalInput inputX2;
17     private LogicalInput inputX3;
18     private Component logicalOR;
19     private Component logicalAND;
20     private LogicalNOT logicalNOT;
21
22     @Before
23     public void setUp() {
24         logicalOR = mock(LogicalOR.class);
25         logicalAND = mock(LogicalAND.class);
26         inputX1 = mock(LogicalInput.class);
27         inputX2 = mock(LogicalInput.class);
28         inputX3 = mock(LogicalInput.class);
29         logicalNOT = mock(LogicalNOT.class);
30         when(logicalNOT.calculateToggleTime()).thenReturn(1);
31         when(inputX1.calculateToggleTime()).thenReturn(0);
32         when(inputX2.calculateToggleTime()).thenReturn(0);
33     }
34
35     @Test
36     public void calculateMaxToggleTimeWithOneLogicalComponent() {
37         when(inputX1.getChannel()).thenReturn(1);
38         when(inputX2.getChannel()).thenReturn(2);
39
40         logicalOR.addInput(inputX1, inputX2);
41
42         when(logicalOR.getInputs()).thenReturn(Arrays.asList(new Component[]{
inputX1, inputX2}));
43
44         circuit = new Circuit(logicalOR);
45
46         when(logicalOR.calculateToggleTime()).thenAnswer(I -> Math.max(1 +
inputX1.calculateToggleTime(), 1 + inputX2.calculateToggleTime()));
47         assertEquals(1, circuit.calculateMaxToggleTime());
48     }
49
50     @Test
51     public void getInputs() {
52         when(inputX1.getChannel()).thenReturn(1);

```

```

53         when(inputX2.getChannel()).thenReturn(2);
54         when(inputX3.getChannel()).thenReturn(3);
55         when(logicalOR.getInputs()).thenReturn(Arrays.asList(inputX1, inputX2));
56         when(logicalAND.getInputs()).thenReturn(Arrays.asList(logicalOR, inputX3
57     ));
58     circuit = new Circuit(logicalAND);
59
60     assertEquals(circuit.getInputs(), Arrays.asList(inputX1, inputX2,
61     inputX3));
62
63     @Test
64     public void calculateMaxToggleTimeWithTwoLogicalComponentsInARow() {
65         when(inputX1.getChannel()).thenReturn(1);
66         when(inputX2.getChannel()).thenReturn(2);
67
68         logicalNOT.addInput(inputX1);
69         logicalOR.addInput(inputX2, logicalNOT);
70
71         when(logicalNOT.getInputs()).thenReturn(Arrays.asList(new Component[]{
72     inputX1}));
73         when(logicalOR.getInputs()).thenReturn(Arrays.asList(new Component[]{
74     inputX2, logicalNOT}));
75
76         circuit = new Circuit(logicalOR);
77
78         when(logicalOR.calculateToggleTime()).thenAnswer(I -> Math.max(1 +
79     inputX2.calculateToggleTime(), 1 + logicalNOT.calculateToggleTime()));
80         assertEquals(2, circuit.calculateMaxToggleTime());
81     }
82
83     @Test(expected = DuplicateInputChannelDetectedException.class)
84     public void addNotUniqueInputsToCircuit() {
85         when(logicalOR.getInputs())
86             .thenReturn(Arrays.asList(new Component[]{inputX1, inputX1}));
87
88         new Circuit(logicalOR);
89     }
90
91     @Test
92     public void calculateFinalState() {
93         when(inputX1.getChannel()).thenReturn(1);
94         when(inputX2.getChannel()).thenReturn(2);
95         when(inputX1.getState()).thenReturn(false);
96         when(inputX2.getState()).thenReturn(true);
97         when(logicalNOT.getState()).thenReturn(true);
98
99         logicalNOT.addInput(inputX1);
100
101         when(logicalNOT.getInputs()).thenReturn(Arrays.asList(new Component[]{
102     inputX1}));
103         when(logicalOR.getState()).thenReturn(true);

```

```
101         logicalOR.addInput(inputX2, logicalNOT);
102
103         when(logicalOR.getInputs()).thenReturn(Arrays.asList(new Component[]{
inputX2, logicalNOT}));
104
105         circuit = new Circuit(logicalOR);
106
107         when(logicalOR.calculateToggleTime()).thenAnswer(invocationOnMock ->
Math.max(1 + inputX2.calculateToggleTime(), 1 + logicalNOT.calculateToggleTime
()));
108         circuit.calculateFinalState();
109         assertTrue(circuit.getState());
110     }
111
112     @Test
113     public void calculateOneIteration() {
114         when(inputX1.getChannel()).thenReturn(1);
115         when(inputX2.getChannel()).thenReturn(2);
116         when(inputX1.getState()).thenReturn(false);
117         when(inputX2.getState()).thenReturn(true);
118         when(logicalNOT.getState()).thenReturn(true);
119
120         logicalNOT.addInput(inputX1);
121
122         when(logicalNOT.getInputs()).thenReturn(Arrays.asList(new Component[]{
inputX1}));
123         when(logicalOR.getState()).thenReturn(true);
124
125         logicalOR.addInput(inputX2, logicalNOT);
126
127         when(logicalOR.getInputs()).thenReturn(Arrays.asList(new Component[]{
inputX2, logicalNOT}));
128
129         circuit = new Circuit(logicalOR);
130
131         when(logicalOR.calculateToggleTime()).thenAnswer(invocationOnMock ->
Math.max(1 + inputX2.calculateToggleTime(), 1 + logicalNOT.calculateToggleTime
()));
132         circuit.calculateOneIteration();
133         assertTrue(circuit.getState());
134     }
135 }
136
```

```
1 package circuitcomponents;
2
3 import exceptions.MissingInputException;
4 import org.junit.Before;
5 import org.junit.Test;
6
7 import static org.junit.Assert.*;
8 import static org.mockito.Mockito.mock;
9 import static org.mockito.Mockito.when;
10
11 public class LogicalORTest {
12     private LogicalOR logicalOR;
13     private Component logicalInput1;
14     private Component logicalInput2;
15
16     @Before
17     public void setUp() {
18         logicalOR = new LogicalOR();
19         logicalInput1 = mock(LogicalInput.class);
20         logicalInput2 = mock(LogicalInput.class);
21     }
22
23     @Test
24     public void calculateStateWithOneInputF() {
25         when(logicalInput1.getState()).thenReturn(false);
26         logicalOR.addInput(logicalInput1);
27
28         logicalOR.calculateState();
29
30         assertFalse(logicalOR.getState());
31     }
32
33     @Test
34     public void calculateStateWithOneInputT() {
35         when(logicalInput1.getState()).thenReturn(true);
36         logicalOR.addInput(logicalInput1);
37
38         logicalOR.calculateState();
39
40         assertTrue(logicalOR.getState());
41     }
42
43     @Test
44     public void calculateStateWithTwoInputsFF() {
45         when(logicalInput1.getState()).thenReturn(false);
46         when(logicalInput2.getState()).thenReturn(false);
47         logicalOR.addInput(logicalInput1);
48         logicalOR.addInput(logicalInput2);
49
50         logicalOR.calculateState();
51
52         assertFalse(logicalOR.getState());
53     }
54 }
```

```
55     @Test
56     public void calculateStateWithTwoInputsFT() {
57         when(logicalInput1.getState()).thenReturn(false);
58         when(logicalInput2.getState()).thenReturn(true);
59         logicalOR.addInput(logicalInput1);
60         logicalOR.addInput(logicalInput2);
61
62         logicalOR.calculateState();
63
64         assertTrue(logicalOR.getState());
65     }
66
67     @Test
68     public void calculateStateWithTwoInputsTT() {
69         when(logicalInput1.getState()).thenReturn(true);
70         when(logicalInput2.getState()).thenReturn(true);
71         logicalOR.addInput(logicalInput1);
72         logicalOR.addInput(logicalInput2);
73
74         logicalOR.calculateState();
75
76         assertTrue(logicalOR.getState());
77     }
78
79     @Test(expected = MissingInputException.class)
80     public void calculateStateWithNoInput() {
81         logicalOR.calculateState();
82
83         assertThrows(MissingInputException.class, () -> logicalOR.calculateState
84             ());
85     }
```

```
1 package circuitcomponents;
2
3 import exceptions.MissingInputException;
4 import org.junit.Before;
5 import org.junit.Test;
6
7 import static org.junit.Assert.*;
8 import static org.mockito.Mockito.mock;
9 import static org.mockito.Mockito.when;
10
11 public class LogicalANDTest {
12     private LogicalAND logicalAND;
13     private Component logicalInput1;
14     private Component logicalInput2;
15
16     @Before
17     public void setUp() {
18         logicalAND = new LogicalAND();
19         logicalInput1 = mock(LogicalInput.class);
20         logicalInput2 = mock(LogicalInput.class);
21     }
22
23     @Test(expected = MissingInputException.class)
24     public void calculateStateWithNoInput() {
25         logicalAND.calculateState();
26
27         assertThrows(MissingInputException.class, () -> logicalAND.calculateState
28     );
29
30     @Test
31     public void calculateStateWithOneInputF() {
32         when(logicalInput1.getState()).thenReturn(false);
33         logicalAND.addInput(logicalInput1);
34
35         logicalAND.calculateState();
36
37         assertFalse(logicalAND.getState());
38     }
39
40     @Test
41     public void calculateStateWithOneInputT() {
42         when(logicalInput1.getState()).thenReturn(true);
43         logicalAND.addInput(logicalInput1);
44
45         logicalAND.calculateState();
46
47         assertTrue(logicalAND.getState());
48     }
49
50     @Test
51     public void calculateStateWithTwoInputsFF() {
52         when(logicalInput1.getState()).thenReturn(false);
53         when(logicalInput2.getState()).thenReturn(false);
```



```
54         logicalAND.addInput(logicalInput1);
55         logicalAND.addInput(logicalInput2);
56
57         logicalAND.calculateState();
58
59         assertFalse(logicalAND.getState());
60     }
61
62     @Test
63     public void calculateStateWithTwoInputsFT() {
64         when(logicalInput1.getState()).thenReturn(false);
65         when(logicalInput2.getState()).thenReturn(true);
66         logicalAND.addInput(logicalInput1);
67         logicalAND.addInput(logicalInput2);
68
69         logicalAND.calculateState();
70
71         assertFalse(logicalAND.getState());
72     }
73
74     @Test
75     public void calculateStateWithTwoInputsTT() {
76         when(logicalInput1.getState()).thenReturn(true);
77         when(logicalInput2.getState()).thenReturn(true);
78         logicalAND.addInput(logicalInput1);
79         logicalAND.addInput(logicalInput2);
80
81         logicalAND.calculateState();
82
83         assertTrue(logicalAND.getState());
84     }
85 }
```

```
1 package circuitcomponents;
2
3 import exceptions.MissingInputException;
4 import exceptions.TooManyInputsException;
5 import org.junit.Before;
6 import org.junit.Test;
7
8 import static org.junit.Assert.*;
9 import static org.mockito.Mockito.mock;
10 import static org.mockito.Mockito.when;
11
12 public class LogicalNOTTest {
13
14     private LogicalNOT logicalNOT;
15     private Component logicalInput1;
16     private Component logicalInput2;
17
18     @Before
19     public void setUp(){
20         logicalNOT = new LogicalNOT();
21         logicalInput1 = mock(LogicalInput.class);
22         logicalInput2 = mock(LogicalInput.class);
23     }
24
25     @Test(expected = MissingInputException.class)
26     public void calculateStateWithNoInput() {
27         logicalNOT.calculateState();
28
29         assertThrows(MissingInputException.class, () -> logicalNOT.calculateState
30             ());
31     }
32
33     @Test
34     public void calculateStateWithOneInputF() {
35         when(logicalInput1.getState()).thenReturn(false);
36         logicalNOT.addInput(logicalInput1);
37
38         logicalNOT.calculateState();
39
40         assertTrue(logicalNOT.getState());
41     }
42
43     @Test
44     public void calculateStateWithOneInputT() {
45         when(logicalInput1.getState()).thenReturn(true);
46         logicalNOT.addInput(logicalInput1);
47
48         logicalNOT.calculateState();
49
50         assertFalse(logicalNOT.getState());
51     }
52
53     @Test(expected = MissingInputException.class)
54     public void addZeroInputs() {
```

```
54     logicalNOT.addInput();
55
56     assertThrows(MissingInputException.class, () -> logicalNOT.addInput());
57 }
58
59 @Test
60 public void addOneInputs() {
61     logicalNOT.addInput(logicalInput1);
62
63     assertEquals(1, logicalNOT.getInputs().size());
64     assertTrue(logicalNOT.getInputs().get(0) instanceof LogicalInput);
65 }
66
67 @Test(expected = TooManyInputsException.class)
68 public void addTwoInputs() {
69     logicalNOT.addInput(logicalInput1, logicalInput2);
70
71     assertThrows(TooManyInputsException.class, () -> logicalNOT.addInput(
logicalInput1, logicalInput2));
72 }
73
74 @Test(expected = TooManyInputsException.class)
75 public void addTwoTimesInputs() {
76     logicalNOT.addInput(logicalInput1);
77     logicalNOT.addInput(logicalInput2);
78
79     assertThrows(TooManyInputsException.class, () -> logicalNOT.addInput(
logicalInput2));
80 }
81 }
82
```

```
1 package circuitcomponents;
2
3 import exceptions.NoInputsAllowedForInputException;
4 import org.junit.Before;
5 import org.junit.Test;
6
7 import static org.junit.Assert.assertThrows;
8 import static org.mockito.Mockito.mock;
9
10 public class LogicalInputTest {
11
12     private LogicalInput logicalInput1;
13     private Component logicalInput2;
14
15     @Before
16     public void setUp() {
17         logicalInput1 = new LogicalInput(1);
18         logicalInput2 = mock(LogicalInput.class);
19     }
20
21     @Test(expected = NoInputsAllowedForInputException.class)
22     public void addInput() {
23         logicalInput1.addInput(logicalInput2);
24
25         assertThrows(NoInputsAllowedForInputException.class, () -> logicalInput1.
addInput(logicalInput2));
26     }
27 }
```

```
1 package circuitcomponents;
2
3 import exceptions.MissingInputException;
4 import exceptions.NoZeroDelayException;
5 import exceptions.TooManyInputsException;
6 import org.junit.Before;
7 import org.junit.Test;
8
9 import static org.junit.Assert.*;
10 import static org.mockito.Mockito.mock;
11 import static org.mockito.Mockito.verify;
12
13
14 public class LogicalComponentTest {
15
16     private LogicalComponent logicalComponentDefault1;
17     private LogicalComponent logicalComponentDefault2;
18     private LogicalComponent logicalComponentDelayZero;
19     private Component logicalInput1;
20     private Component logicalInput2;
21     private Component logicalInput3;
22     private Component logicalInput4;
23     private Component logicalInput5;
24
25     @Before
26     public void setUp() {
27         logicalComponentDefault1 = new ImplLogicalComponent();
28         logicalComponentDefault2 = new ImplLogicalComponent();
29
30         logicalInput1 = mock(LogicalInput.class);
31         logicalInput2 = mock(LogicalInput.class);
32         logicalInput3 = mock(LogicalInput.class);
33         logicalInput4 = mock(LogicalInput.class);
34         logicalInput5 = mock(LogicalInput.class);
35     }
36
37
38     @Test(expected = MissingInputException.class)
39     public void addInputWithoutArgument() {
40         logicalComponentDefault1.addInput();
41
42         assertThrows(MissingInputException.class, () -> logicalComponentDefault1.
addInput());
43     }
44
45     @Test(expected = TooManyInputsException.class)
46     public void addMoreThanFourInputs(){
47         logicalComponentDefault1.addInput(logicalInput1, logicalInput2,
logicalInput3, logicalInput4, logicalInput5);
48
49         assertThrows(MissingInputException.class, () -> logicalComponentDefault1.
addInput());
50     }
51
```

```

52     @Test
53     public void addWithOneInput() {
54         logicalComponentDefault1.addInput(logicalInput1);
55
56         assertEquals(1, logicalComponentDefault1.getInputs().size());
57         logicalComponentDefault1.getInputs().forEach(component -> assertTrue(
component instanceof LogicalInput));
58     }
59
60     @Test
61     public void addWithTwoInputs() {
62         logicalComponentDefault1.addInput(logicalInput1);
63         logicalComponentDefault1.addInput(logicalInput2);
64
65         assertEquals(2, logicalComponentDefault1.getInputs().size());
66         logicalComponentDefault1.getInputs().forEach(component -> assertTrue(
component instanceof LogicalInput));
67     }
68
69     @Test(expected = NoZeroDelayException.class)
70     public void calculateToggleTimeZero() {
71         logicalComponentDelayZero = new ImplLogicalComponent(0);
72
73         assertThrows(NoZeroDelayException.class, () -> logicalComponentDelayZero
= new ImplLogicalComponent(0));
74     }
75
76     @Test
77     public void calculateDefaultToggleTime() {
78         assertEquals(logicalComponentDefault1.calculateToggleTime(), 1);
79     }
80
81     @Test
82     public void calculateToggleTimeWithInputsInDifferentPathLength() {
83         logicalComponentDefault2.addInput(logicalInput1, logicalInput2);
84         logicalComponentDefault1.addInput(logicalInput2,
logicalComponentDefault2);
85
86         assertEquals(2, logicalComponentDefault1.calculateToggleTime());
87     }
88
89     @Test
90     public void callCalculateOnChildren() {
91         logicalComponentDefault1.addInput(logicalInput3);
92
93         logicalComponentDefault1.callCalculateOnChildren();
94
95         verify(logicalInput3).calculateState();
96     }
97
98     @Test
99     public void testCanComponentCalculateTrueDefaultDelay() {
100         assertTrue(logicalComponentDefault1.canComponentCalculate());
101     }

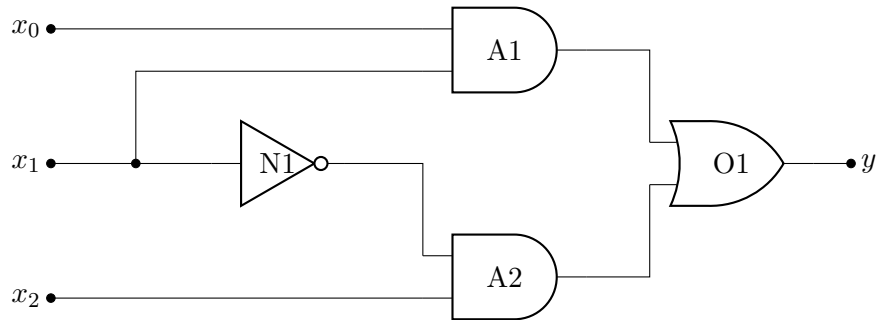
```

```
102
103     @Test
104     public void testCanComponentCalculateFalseDelayTwo() {
105         logicalComponentDefault1 = new ImplLogicalComponent(2);
106         assertFalse(logicalComponentDefault1.canComponentCalculate());
107     }
108
109     @Test
110     public void testCanComponentCalculateTrueDelayTwo() {
111         logicalComponentDefault1 = new ImplLogicalComponent(2);
112         assertFalse(logicalComponentDefault1.canComponentCalculate());
113         assertTrue(logicalComponentDefault1.canComponentCalculate());
114     }
115
116     class ImplLogicalComponent extends LogicalComponent {
117         ImplLogicalComponent() {
118             super();
119         }
120
121         ImplLogicalComponent(int delay) {
122             super(delay);
123         }
124
125         @Override
126         public void calculateState() throws MissingInputException {
127             // do nothing
128         }
129     }
130
131 }
```

6 Anhang - Startkonfiguration

6.1 Schaltung 1:

Funktion: $x_0x_1 + \bar{x}_1x_2$.



6.2 Schaltung 2:

Funktion: $x_0\bar{x}_1x_2 + x_0x_1\bar{x}_3 + \bar{x}_0x_1\bar{x}_2$.

