Hochschule
Zittau/Görlitz
UNIVERSITY OF APPLIED SCIENCES

Hochschule Zittau/Görlitz
Fakultät Elektrotechnik und Informatik

Softwaretests in dem Vier-Gewinnt-Projekt der IIb23

Beleg

Modul / Lehrveranstaltung: Grundlagen des Softwaretestens
Dozent: Prof. Dr. Matthias Längrich

Niklas Kaulfers

Matrikelnummer: 1064032

Abgabedatum: 25. Dezember 2025

Inhaltsverzeichnis

Kapitel 1

Einleitung

Fehler sind in der Softwareentwicklung nahezu nicht zu vermeiden, jedoch können Entwickler versuchen diesen so gut wie möglich vorzubeugen und somit nicht in entsprechende Produktionsumgebungen vordringen zu lassen. Dies tun sie mittels Softwaretests. In dieser Arbeit wird sich mit der Durchführung dieser Tests an einem Projekt auseinandergesetzt. Es wird tiefer auf spezielle Testfälle eingegangen und allgemeine Testvorgehensweisen werden angewandt. Um die Testfälle genauer zu verstehen wird zuerst eine Analyse des Testobjektes durchgeführt. Hieran wird die Komplexität des Projektes ermittelt und eventuelle bereits sichtbare Probleme diskutiert. Anschließend werden die entsprechenden Grundlagen des Softwaretestens, welche von Bedeutung in dieser Arbeit sind, vorgestellt und erläutert. Daraufhin wird eine Planung des Testablaufs durchgeführt und hierbei an einzelnen Fällen ein tieferer Einblick gegeben. Zur Durchführung wird zu jedem der vorgestellten Möglichkeiten des Testens eine an einem Beispiel des Projektes gezeigt. Diese sind im direkten Zusammenhang zu Softwareentwicklung innerhalb des Projektes, welche im Rahmen dieser Arbeit erfolgten. Schlussendlich werden Probleme des Projektes, welche Mithilfe der Tests aufgespürt wurden aufgezeigt und teils auch behoben. Inwiefern diese Arbeit erfolgreich war und mögliche Konsequenzen können aus dem Fazit entnommen werden.
Im Verlauf der Arbeit sind unterstrichene und *kursive* Textabschnitte zu finden. Unterstrichene sind Links und *kursive* Terminalanweisungen. Teile des Quellcodes, welcher in der Ausarbeitung dieser Arbeit entstand ist hingegen im Anhang C zu finden.

Kapitel 2

Testobjekt

2.1 Vorstellung des Projektes

Das Projekt, an welchem die Tests durchgeführt wurden ist ein Gemeinschaftsprojekt des Matrikels IIb23 aus dem Modul OOP (Objektorientierte Programmierung). Es handelt sich hierbei um eine Java-Implementation des Spiels Vier Gewinnt, einem bekannten und relativ leicht verständlichem Strategiespiel. Dieses Spiel zu zweit Spielen spielbar. Das Ziel ist es jeweils 4 der eigenen Steine in einer Reihe zu platzieren und mit eigenen Steinen verhindern, dass der Gegner dasselbe tut.
Die Studenten des Moduls wurden im Rahmen des Moduls in verschiedene Teams aufgeteilt um das Projekt gemeinsam anzufertigen. Die Teams waren Projektmanagement, Backend, Frontend und Testing (Ich war Teil des Projektmanagementteams). Zur Erstellung wurde über GitHub kollaboriert. Alle im Rahmen dieser Arbeit erbrachten Leistungen sind in dieser GitHub-Repository zu finden. Im Fall des Projektes gibt es Möglichkeiten, zum lokalen spielen gegen einen anderen Spieler oder gegen einen Computergegner. Das Spiel hat zudem ein GUI (Graphical User Interface), welches den Spielstand visualisiert und entsprechende Aktionen für den Spieler erlaubt[1].
Im Hintergrund besteht das Spiel aus 4 Komponenten; Api, Logik, Frontend und Tests. Diese sind voneinander logisch in der Ordnerstruktur abgetrennt[2]. Der api Ordern beinhaltet Interfaces, welche die Logik definieren. Hier gibt es auch ein Interface, welches ausschließlich zum Testen gemacht wurde. Die Logik implementiert diese Interfaces und versorgt sie mit Funktionalität. Für ein ordentliches grafisches Display sorgt das Frontend. Dieses ist im Vergleich zur Logik deutlich besser modularisiert und somit auch besser lesbar. Schlussendlich gibt es in dem Projekt bereits Tests, welche im Test-Ordner sind. Jedoch waren diese vor dieser Arbeit noch nicht ausgereift.

2.1.1 Psychologische Betrachtung

Tests werden häufig als unnötig und als Methode zum zeigen, dass es keine Errors gibt angesehen. Tatsächlich werden Tests jedoch verwendet um Errors zu finden und fehlverhalten der Software im Vorhinein zu verringern[3]. Auch hier im Projekt waren die bereits existieren Tests eher minimal und testen vor allem die Hauptszenarios. Vor allem explizite Logik ist noch ungetestet.
Implementation von Verbesserungen oder eventuelle Bugfixes sind zudem unerwartet, da es sich hierbei um ein abgeschlossenes Projekt handelt. Das Projekt hat keinerlei Bugreportmethoden oder ähnliches. Es gibt zwar Issues in der GitHub-Repository, diese werden jedoch nicht in Pullrequests (PR) thematisiert. Generell wurden die meisten Features nicht per PR in die Mainbranch gemerged. In diesem Projekt können Mitentwickler einfach merges in die Mainbranch durchführen, da diese nicht geschützt ist.

[1] A.1, Screenshot aus dem Spiel
[2] A.2, Layout des Projektes zu Beginn der Tests
[3] G. J. Myers u. a., *The art of software testing*. Wiley Online Library, 2004, Bd. 2, p.10.

2.1.2 Ökonomische Betrachtung

Während im ökonomischen Sinn meist die Gesamtkosten betrachtet werden, ist in dieser Arbeit die Zeit von größerer Bedeutung. Rahmen des Projektes stellt das Wintersemester 2025/26.
Die Tests müssen deswegen in einem sehr geringem Zeitrahmen gefertigt werden.
Das Projekt war ein Teil eines Hochschulmoduls, somit waren die Ressourcen stark begrenzt. Auch hierzu anzumerken ist, dass es sich um ein Modul aus dem zweiten Semester handelt, weswegen auch die Erfahrungen der beteiligten noch nicht ausgereift waren.

2.2 Nötige Anpassungen

Das Projekt auszuführen stellte sich als unnötig komplex heraus, da das Mainfile, welches die Software startet, in einem von vielen Unterordnern aufzufinden war.
Im vor hinein war das Projekt sehr unübersichtlich und benötigte einige Verbesserungen, vor allem in Hinblick auf Namensgebung von Variablen. Diese waren Teils mit einzelnen Buchstaben benannt und somit schwer verständlich und lesbar[4].
Hier ist eine bessere Projektstruktur, welche eine einfachere Ausführung und Verarbeitung ermöglicht benötigt. Um dies zu verwirklichen wurde das Projekt in einer neuen Branch in gradle neu aufgesetzt. Nun kann das Projekt einfach über *gradle run* ausgeführt werden. Auch sämtliche Tests sind mit *gradle test* somit leichter zugänglich.
Alle diese Anpassungen geschahen vor einer Umsetzung der Tests um die Entwicklungsumgebung angenehmer zu gestalten.

2.3 Eignung zum Testen

Da das Projekt aus mehreren verschieden komplexen Codeteilen besteht ist es auch gut zum Testen geeignet. Auch von nutzen hier ist, dass es ein eigenes UI gibt, an welchem auch Tests durchgeführt werden können.
Problematisch hingegen ist, dass das Projekt nicht weiter in Bearbeitung ist und die Tests langfristig eher von geringem Nutzen sein werden. Da diese Arbeit als Teil eines Hochschulmoduls zum Lernen angefertigt wurde, ist das im Hinblick auf die Fähigkeit dieses Projekt zu Testen jedoch irrelevant. Einen positiven langfristigen Einfluss kann hingegen für Studenten existieren, welche zu alten Projekten zurückkehren und ihre Kompetenzen testen wollen.

[4] A.2, Layout des Projektes zu Beginn der Tests

Kapitel 3

Grundlagen

3.1 statisch oder dynamisch

Statische und dynamische Tests sind in jeder modernen Software angewandt. Jedoch was ist der Unterschied? Bei statischen oder manuellen[1] Tests wird die Struktur des Codes analysiert, aber der Code nicht ausgeführt[2]. Im Gegensatz dazu werden dynamische oder automatisierte Tests von einem Testplan abgeleitet, ausgeführt und als Resultat evaluiert[3]. Somit stellen statische Tests beispielsweise Code Reviews da, während dynamische Tests die Tests der Testsuit in der Codebasis sind.

3.2 Vorwissen und Einblick der Tester

Es gibt drei Methoden Tests durchzuführen. Diese sind Whitebox- und Blackboxtesting. Blackboxtesting wird auch funktionales Testen genannt[4] in diese ´´schwarzen Box´´(nicht durchsichtigen) kann der Tester nicht hineinsehen. Somit soll der Tester keinen Zugang zum Code haben und interessiert sich einzig und allein für den richtigen Ausgabewert[5] Im Gegensatz dazu hat Whiteboxtesting Zugang zum Code für die Tester, um auf die Analogie der ´´schwarze Box´´ zurückzukommen wird Whiteboxtesting auch als Struktur- oder Glasboxtests bezeichnet[6][7] Teils wird auch Greyboxtesting erwähnt[8]. Dies ist eine Methode, bei welcher die Tester etwas Vorwissen von der Codebasis haben[9] Sie versucht die Probleme von Blackbox- und Whiteboxtesting zu lösen. So kann beispielsweise Backendlogik über das Frontend getestet werden, die Tester kenne den Code des Backends hierbei, jedoch nicht den des Frontends. Diese Form des Testens eignet sich so sehr gut für Webanwendungen[10].

[1] A. A. Sawant u. a., „Software testing techniques and strategies," *International Journal of Engineering Research and Applications (IJERA)*, Jg. 2, Nr. 3, S. 980–986, 2012, S. 981.

[2] R. Fairley, „Tutorial: Static Analysis and Dynamic Testing of Computer Software," *Computer*, Jg. 11, Nr. 4, S. 14–23, 1978. DOI: 10.1109/C-M.1978.218132, S. 14.

[3] R. Fairley, „Tutorial: Static Analysis and Dynamic Testing of Computer Software," *Computer*, Jg. 11, Nr. 4, S. 14–23, 1978. DOI: 10.1109/C-M.1978.218132, S. 16.

[4] V. Basili und R. Selby, „Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, Jg. SE-13, Nr. 12, S. 1278–1296, 1987. DOI: 10.1109/TSE.1987.232881, S. 1278f.

[5] S. Nidhra und J. Dondeti, „Black box and white box testing techniques-a literature review," *International Journal of Embedded Systems and Applications (IJESA)*, Jg. 2, Nr. 2, S. 29–50, 2012, S. 29f.

[6] S. Nidhra und J. Dondeti, „Black box and white box testing techniques-a literature review," *International Journal of Embedded Systems and Applications (IJESA)*, Jg. 2, Nr. 2, S. 29–50, 2012, S. 30.

[7] V. Basili und R. Selby, „Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, Jg. SE-13, Nr. 12, S. 1278–1296, 1987. DOI: 10.1109/TSE.1987.232881, S. 1279.

[8] M. A. Jamil u. a., „Software Testing Techniques: A Literature Review," in *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*, 2016, S. 177–182. DOI: 10.1109/ICT4M.2016.045, S. 2.

[9] I. Jovanović, „Software testing methods and techniques," *The IPSI BgD transactions on internet research*, Jg. 30, 2006, S. 2.

[10] S. Acharya und V. Pandya, „Bridge between black box and white box–gray box testing technique," *International Journal of Electronics and Computer Science Engineering*, Jg. 2, Nr. 1, S. 175–185, 2012, S. 179.

3.3 Spektrum an Tests

Um Software zu Testen gibt es eine Vielzahl an verschiedenen Tests, welche an der Software durchgeführt werden können. Diese Tests werden zu verschiedenen Zeitpunkten im Softwaredevelopmentlifecycle angewandt[11].

3.3.1 Arten der dynamischen Tests

Die wohl bekannteste Form der dynamischen Tests sind die **Unittests**. Bei diesen wird das kleinste, alleinstehende Modul der Software getestet[12]. Im Umfeld von Objektorientierter Programmierung (OOP) heißt dies vor allem Testen der einzelnen Klassen[13]. Da die Umsetzung von OOP etwas mangelhaft in diesem Projekt ist und keine tiefgreifende Modualisierung stattfindet, wird in dieser Arbeit Unittests auf die einzelnen Funktionen angewandt.
Um nach Codeveränderungen möglichen Regressionen entgegen zu wirken, werden **Regressionstests** ausgeführt. Diese Tests werden an Code durchgeführt, welcher sich verändert hat[14].

3.3.2 Arten der statischen Tests

Eine typische Form der statischen Tests ist das **Codereview**. Bei diesem wird der geschriebene Code von einem anderem Entwickler überprüft. Moderne Codereviews sind informell, toolbasiert, asynchron und fokussiert auf Codeänderungen[15]. Es gibt eine Menge an verschiedenen Arten, wie diese Codereviews durchgeführt werden können. Dazu zählen die Codeinspektion, asynchrone Reviews per Email, toolbasierte Reviews und pullbasierte Reviews. Diese pullbasierten Reviews beziehen sich auf Reviews von Pullrequests, beispielsweise über Git[16] oder genauer GitHub.
Eine weitere Form ist die **statische Analyse**, diese bezieht sich auf das Betrachten einer Komponente im Bezug auf Form, Struktur, Inhalt und Dokumentation[17]. Es besteht in diesem Zusammenhang auch die Möglichkeit der Automatisierung. Mithilfe dieser Tools kann der Code untersucht werden und mögliche Anormalitäten aufgespürt werden[18].

[11] S. Nidhra und J. Dondeti, „Black box and white box testing techniques-a literature review," *International Journal of Embedded Systems and Applications (IJESA)*, Jg. 2, Nr. 2, S. 29–50, 2012, S. 30.
[12] vgl. P. Runeson, „A survey of unit testing practices," *IEEE Software*, Jg. 23, Nr. 4, S. 22–29, 2006. DOI: 10.1109/MS.2006.91, S. 24.
[13] vgl. S. Wappler und J. Wegener, „Evolutionary unit testing of object-oriented software using strongly-typed genetic programming," in *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, 2006, S. 1925–1932, S. 1925f.
[14] vgl. O. Legunsen u. a., „An extensive study of static regression test selection in modern software evolution," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, S. 583–594, S. 583f.
[15] C. Sadowski u. a., „Modern code review: a case study at google," in *Proceedings of the 40th international conference on software engineering: Software engineering in practice*, 2018, S. 181–190, S. 181.
[16] vgl. C. Sadowski u. a., „Modern code review: a case study at google," in *Proceedings of the 40th international conference on software engineering: Software engineering in practice*, 2018, S. 181–190, S. 182.
[17] „IEEE Standard Glossary of Software Engineering Terminology," *IEEE Std 610.12-1990*, S. 1–84, 1990. DOI: 10.1109/IEEESTD.1990.101064, S. 70.
[18] vgl. J. Zheng u. a., „On the value of static analysis for fault detection in software," *IEEE transactions on software engineering*, Jg. 32, Nr. 4, S. 240–253, 2006, S. 240f.

Kapitel 4

Planung

4.1 Testziele

Um die Tests erfolgreich abzuschließen müssen zuerst die Ziele definiert werden. Hier werden diese vor allem von der Grundsicherung der Funktionalität beeinträchtigt. Das Ziel der Tests in diesem Produkt ist es eine hohe Testcoverage zu haben und auch jeweilige Edgecases abzudecken. Somit soll das Abstürzen des Spieles auf ein Minimum reduziert werden[1].
Um eine hohe Qualität der Tests sicherzustellen sollten gewisse Elemente und die damit verbundenen Anforderungen vor der tatsächlichen Umsetzung der Tests aufgelistet werden[2]. Eine solche Qualitätsplanung ist auch hier von Sinn[3]. Diese Planung ist nur für Board vorgenommen wurden um in einem sinngemäßen Rahmen zu bleiben.

4.2 Qualitätssicherung

Um gute Qualität im Endprodukt zu erhalten muss eine Qualitätssicherung durchgeführt werden. Diese bezieht sich auf den Entwicklungs- als auch den Testprozess und liegt in der Verantwortung aller Projektbeteiligten[4]. Im Verlauf des Projektes wurde der Qualitätssicherungsprozess vollständig ignoriert. Daraus kann auch die inkonsistente Codequalität herausgeleitet werden. So sind Kommentare in sowohl deutsch als auch englisch und einige Codeabschnitte sind deutlich besser lesbar als andere. Somit ist das Projekt langfristig schwer aufrecht zu erhalten und zu erweitern. Da vor dieser Arbeit nur ein geringes Mass an Testen existierte sind auch viele Fehler unbehoben und in der Entwicklung ignoriert wurden.

[1] vgl. D. Sernow u. a., *CTFL4.0 Grundlagen des Testens*, Presentation, expleo/ISTQB, unpublished, WS25/26, 2025, K. 1 S. 12.
[2] vgl. L. Baresi und M. Pezze, „An introduction to software testing," *Electronic Notes in Theoretical Computer Science*, Jg. 148, Nr. 1, S. 89–111, 2006, S. 93.
[3] B.1, Qualitätsplanung für Board
[4] D. Sernow u. a., *CTFL4.0 Grundlagen des Testens*, Presentation, expleo/ISTQB, unpublished, WS25/26, 2025, K. 1 S. 23.

Kapitel 5

Durchführung

Zur Durchführung wird jeweils individuell ein Testfall detailliert erläutert.

5.1 Realisierung

Als Problem stellt sich direkt heraus, dass Testen nicht isoliert durchgeführt werden soll[1]. Dies ist hier nicht möglich, da keine tatsächliche Entwicklung mehr stattfindet.
Aufgrund der relativ niedrigen Komplexität ist das Einrichten der Entwicklungs-/Testumgebung verhältnismäßig einfach. Anforderungen des Projektes sind lediglich Java 21 oder höher. Empfohlen wird die IDE IntelliJ IDEA. Um in der Branch mit gradle zu arbeiten muss auch gradle cli installiert sein. Die Tests werden mit JUnit 5 durchgeführt und können mittels IntelliJ einfach im Code gestartet werden. In der Branch mit gradle ist es einfach, wie in 2.2, mit *gradle test* auszuführen.

5.2 dynamische Tests

Die dynamischen Softwaretests sind in drei Abschnitte zu unterteilen. Unit tests stellen die Funktionalität eines Moduls sicher und testen nur dieses. Mehrere Module werden im Laufe der Softwareentwicklung aneinandergereiht und verknüpft, um dieses Konstrukt zu Testen werden Integration Tests verwendet. Der finale Schritt, die System Tests, Testen die gesamte Software aus allen Perspektiven[2].

5.2.1 Unit Tests - Verbleibende Plätze in einer Spalte

Um zu schauen, ob weiteres platzieren von Steinen in einer Spalte erlaubt ist gibt es eine Funktion `int isTopOfColumn(int column)`. Diese überprüft, wie viele Plätze in einer Spalte noch frei sind.
Wenn man diese Funktion betrachtet fällt zuerst der Name auf. Mit `isTopOfColumn` ist ein boolean-Wert als Ausgabe zu erwarten. Jedoch ist der tatsächliche Rückgabewert vom Typen int. Somit wird die Anzahl der Plätze welche bis zur Maximalmenge zur Verfügung stehen gezählt und dementsprechend 0 zurückgegeben, wenn kein Platz frei ist.
Zur Implementierung der Tests müssen zuerst Grenzwerte gesetzt werden, in welchen sich die Funktion wie verhalten soll. Das Standartboard hat ein Layout von 7x7 somit kommt Abbildung 5.1 für die validen inputs der Funktion zustande. Daraus folgt, dass Tests für die Inputs -1, 0, 6, 7 durchgeführt werden müssen. Die Funktion ist nicht nur von der tatsächlichen Spalte abhängig, sondern auch von der Anzahl der platzierten Steine. Somit wird auch getestet, ob die richtigen Werte innerhalb des gültigen Bereichs zurückgegeben werden. Um hier das genaue Verhalten zu testen fehlt im Code eine Möglichkeit, um Steine über oder unter dem Spielfeld zu platzieren.
In der Durchführung der Tests stellt sich heraus, dass das Verhalten unterhalb der Untergrenze und überhalb

[1] D. Sernow u. a., *CTFL4.0 Grundlagen des Testens*, Presentation, expleo/ISTQB, unpublished, WS25/26, 2025, K. 1 S. 52.
[2] vgl. M. A. Jamil u. a., „Software Testing Techniques: A Literature Review," in *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*, 2016, S. 177–182. DOI: 10.1109/ICT4M.2016.045, S. 177.

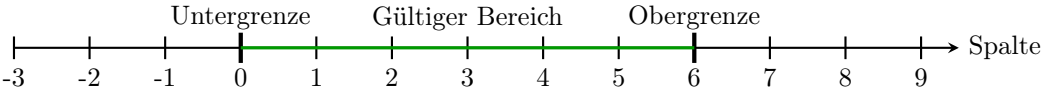

Abbildung 5.1: Grenzwerte der Inputs von `isTopOfColumn()` (im Fall des Boardlayouts 7x7)

der Obergrenze nicht definiert ist. Somit kommt es zu einer `ArrayIndexOutOfBoundsException` für beide Fälle. Da es keine Möglichkeit gibt, die Steine so zu platzieren, dass in einer Spalte mehr Steine liegen als vorgesehen, ist es auch nicht möglich das Verhalten in diesem Fall zu testen. Für alle anderen Fälle verhält sich die Funktion wie vorgesehen[3].

5.2.2 Integration Tests - Laden und speichern des Spielstands

Das Spiel hat eine Speicherlogik. Für diese wird ein String erstellt, welcher den aktuellen Stand des Spieles aufzeichnet. Dieser funktioniert wie folgt:`{playerTurn}a{rows}a{columns}a{isFull}aB{boardLayout}`. Dieser String wird dann in einem txt-File gespeichert und kann beim neustarten des Spiels aufgerufen werden. Für den Test ist es wichtig, dass sich das txt-File ordentlich öffnet und ausliest. Zudem muss getestet werden, ob der Speichercode den Spielstand ordnungsgemäß darstellt. Auch ist das Verhalten interessant, für den Fall das kein Speicherstand existiert und versucht wird einen zu laden.
Um dem Ziel von Integration Tests zu folgen sind zwischenwerte und interne Abläufe hier unwichtig. Von Relevanz is lediglich das richtige Ergebnis, in diesem Fall das richtige Speichern und Auslesen.
Für den Testablauf muss hier nach jedem Durchlauf der Zustand der Dateien wieder zum ausgangszustand zurückgebracht werden. Da das Speichern des Spiels eine Datei erstellt, welche zuvor noch nicht da ist.
Das Diagram 5.2 zeigt auf, wie der Speicher- & Ladeprozess im Spiel funktioniert. Dabei wird alles, was keinen direkten Einfluss auf den Speicher- & Ladeprozess hat ignoriert.

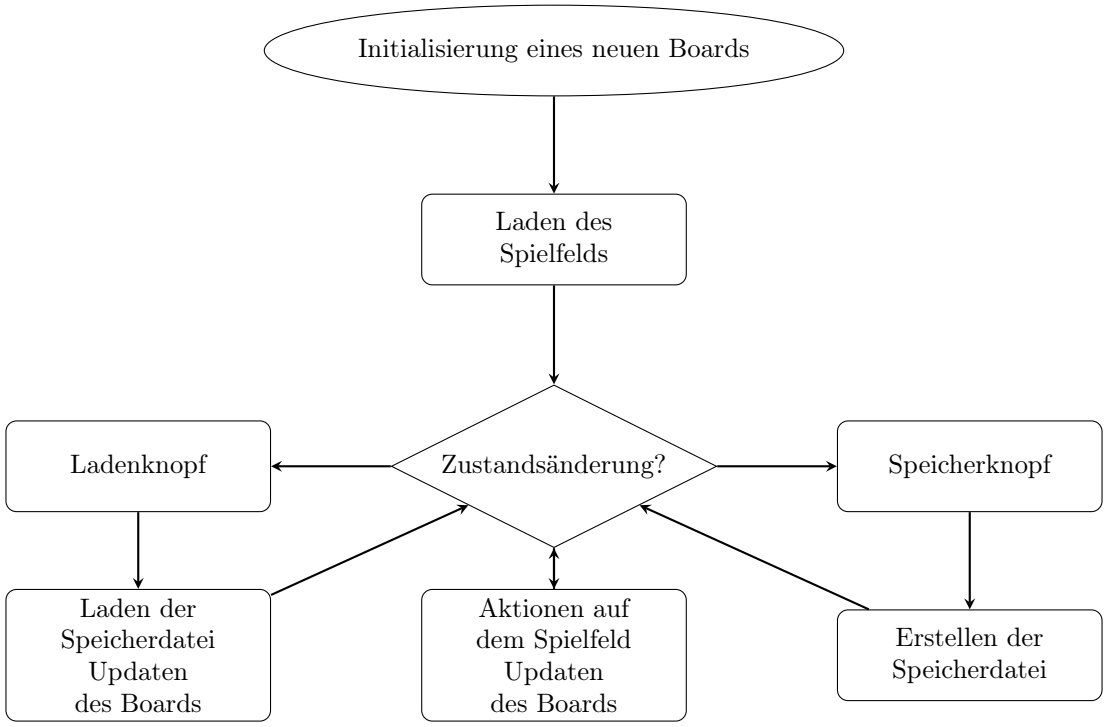

Abbildung 5.2: Flowchart für Speicher-/Ladeprozess

Aus der Figur 5.2 ist bereits ein Problem des Speicherkonzepts zu erkennen, die Datei wird erst mit erst-

[3] siehe Code C.1

maligem Speicher erstellt. Der Knopf zum Laden des Spiels hingegen ist bereits mit dem Öffnen des Spiels möglich. Dies kann zu Fehlern und undefinierten Verhalten führen.
In der tatsächlichen Durchführung der Tests bewahrheitet sich dies. Der `FileReader` findet keine Datei zum auslesen und wirft ein Error, wenn man den Ladeprozess direkt nach dem Start des Spiels aufruft, ohne zuvor gespeichert zu haben.

5.2.3 System Testing

Während effektive Unit- und Integration Tests verhältnismäßig effektiv umzusetzen sind. Sind System Tests deutlich komplexer und schwieriger zu implementieren.
Vor allem in sehr großen Projekten ist diese häufig zu komplex[4]. Dieses Projekt ist jedoch verhältnismäßig klein und somit auch in der Hinsicht leichter zu Testen.

5.3 statische Tests

Im Rahmen der statischen Tests wird eine Codeanalyse in Form eines Codereviews vorgenommen. Hier wird der Commit `dec55bf` betrachtet, welcher die Funktion zum Speichern und Laden der Speichern. Zu betrachten ist der Code C.2 zum Decoden des Savefile-Inhalts.
Zuerst wird der Code nach seiner Lesbarkeit bewertet. Hier kommt direkt ein Problem auf: Die Benennung der Variablen. Wie in 5.3 zu erkennen ist sind die Namen nicht leicht zu entschlüsseln, vor allem da sie häufig in Kombination miteinander auftreten und so schwer zu verstehen ist, welche Variable wofür zuständig ist. Dieses Problem ist wenig problematisch, wenn nur ein Entwickler an dem Programm arbeitet, jedoch durchaus von größerer Bedeutung, wenn ein Team daran arbeitet. Da gute Namensgebung von fundamentaler Bedeutung für die Lesbarkeit des Codes ist[5] Es stellt sich so das Problem heraus, wie der Code in Zukunft zu warten ist. Die Entwicklung kann so verlangsamt werden.

Abbildung 5.3: Variablen der Funktion

```
Tile[][][] spFeld;
boolean player1Turn = false;
boolean full = false;
int row = -1;
int column = -1;

char[] scCh = savecode.toCharArray();
int countA = 0;
String v = "";
boolean checkB = false;
int r=0;
int c=0;
```

Zudem hat der Code leichte Formatierungsfehler, so sind Teils Leerzeilen ohne Bedeutung platziert. Prinzipiell hätte die gesamte Speicherlogik eine eigene Klasse sein sollen, um den Code besser zu modularisieren und langfristig sicher zu stellen, dass weitere Features hinzugefügt werden können.

[4] vgl. J. Hayes und A. Offutt, „Increased software reliability through input validation analysis and testing," in *Proceedings 10th International Symposium on Software Reliability Engineering (Cat. No.PR00443)*, 1999, S. 199–209. DOI: 10.1109/ISSRE.1999.809325, S. 199.

[5] vgl. F. Deissenboeck und M. Pizka, „Concise and consistent naming," *Software Quality Journal*, Jg. 14, Nr. 3, S. 261–282, 2006, S. 281.

Kapitel 6

Probleme des Projektes

6.1 Behobene Probleme

6.2 Nicht behobene Probleme

Kapitel 7

Fazit

Appendices

Anhang A

Grafiken

Abbildung A.1: Bild des Spiels

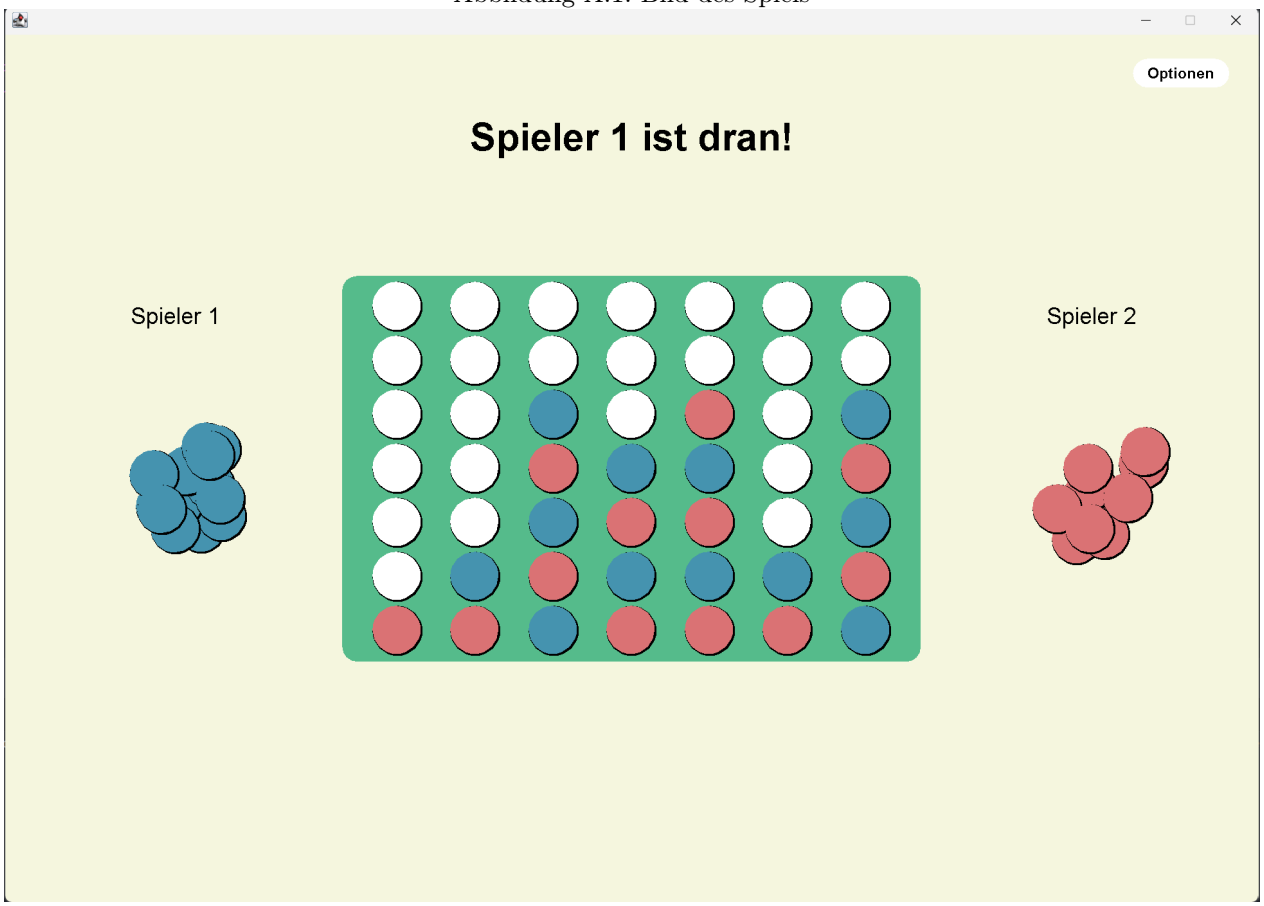

Abbildung A.2: Projektstruktur vor Verbesserungen (Einstiegspunkt in rot)

```
VierGewinnt/
├── lib/
│   └── junit-platform-console-standalone.jar
├── res/
│   ├── SpielerVsComputer.png
│   └── SpielerVsSpieler.png
├── src/
│   ├── api/
│   │   ├── BoardInterface.java
│   │   ├── BoardTestInterface.java
│   │   └── TileInterface.java
│   ├── gui/
│   │   ├── entity/
│   │   │   ├── BordersForCircle.java
│   │   │   └── Circle.java
│   │   ├── frames/
│   │   │   ├── EndFrame.java
│   │   │   ├── MainPanel.java
│   │   │   ├── OptionsFrame.java
│   │   │   └── StartFrame.java
│   │   ├── handler/
│   │   │   └── MouseHandler.java
│   │   └── main/
│   │       └── App.java
│   ├── logic/
│   │   ├── Board.java
│   │   └── Tile.java
│   └── test/
│       └── GameTest.java
├── .gitignore
├── README.md
├── TEST.md
├── VierGewinnt.jar
└── win4.jar
```

Anhang B

Tabellen

Abbildung B.1: Planung der Tests für Board.java (nach L. Baresi und M. Pezze, „An introduction to software testing," *Electronic Notes in Theoretical Computer Science*, Jg. 148, Nr. 1, S. 89–111, 2006, S. 93f.)

Test Items	Board.java
Features to be tested	placeStone, computer algorithm, save/load, winning
Features not to be tested	Setters/Getters
Approach	Unit Tests mit Grenzwerten oder Äquivalenzklassen für zu umfangreichen Code Integration Tests für einen Spielablauf mit Use Case KI-basiertes Testen für höhere Effizienz bei redundanten Fällen
Pass/Fail criteria	Alle Funktionen verhalten sich wie erwartet Fail bei unter 100% Success
Suspension and resumption criteria	Codequalität zu niedrig bzw unlesbar: Codeverbesserungen sind umgesetzt Kritischer Error/Crash: Fix des kritischen Errors
Risks and contingencies	Zeitliche Begrenzungen
Deliverables	Bugs und generelle QA
Tasks & Schedule	Zeit bis Februar 2025
Staff & responsibilities	Niklas Kaulfers (Umsetzung), IIb23 (Bereitstellung des Projekts und Softwareentwicklung)
Environment needs	JUnit, Java correto 23, IntelliJ IDEA

Anhang C

Code

Listing C.1: IsTopOfColumnTest.java

```
package test.logic.board;

import api.BoardTestInterface;
import logic.Board;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

public class IsTopOfColumnTest {

    BoardTestInterface board;

    @BeforeEach
    void setUp() {
        board = new Board();
    }

    @Test
    @DisplayName(
        "WHEN column 1 has 3 stones placed THEN return 3"
    )
    void halfwayFullColumnTest() {
        board.placeStone(1);
        board.placeStone(1);
        board.placeStone(1);

        Assertions.assertEquals(
            3,
            board.isTopOfColumn(1)
        );
    }

    @Test
    @DisplayName(
        "WHEN column 1 has 6 stones THEN return 0"
    )
    void fullColumnTest() {
```

```
        for (int i = 0; i < 6; i++) {
            board.placeStone(1);
        }

        Assertions.assertEquals(
            0,
            board.isTopOfColumn(1)
        );
    }

    @Test
    @DisplayName(
        "WHEN column 1 has 0 stones THEN return 6"
    )
    void emptyColumnTest() {
        Assertions.assertEquals(
            6,
            board.isTopOfColumn(1)
        );
    }

    @Test
    @DisplayName(
        "GIVEN a 10x10 board WHEN column 10 has 0 "
      + "stones THEN return 10"
    )
    void emptyColumn10Test() {
        board = new Board(10, 10);

        Assertions.assertEquals(
            10,
            board.isTopOfColumn(9)
        );
    }

    @Test
    @DisplayName(
        "GIVEN a 10x10 board WHEN column 1 has 10 "
      + "stones THEN return 0"
    )
    void fullColumn10Test() {
        board = new Board(10, 10);

        for (int i = 0; i < 10; i++) {
            board.placeStone(1);
        }

        Assertions.assertEquals(
            0,
            board.isTopOfColumn(1)
        );
    }

    @Test
```

```
    @DisplayName(
        "WHEN column -1 is called THEN do nothing"
    )
    void outOfBoundsTooLowTest() {
        Assertions.assertEquals(
            0,
            board.isTopOfColumn(-1)
        );
    }

    @Test
    @DisplayName(
        "WHEN column 7 is called THEN do nothing"
    )
    void outOfBoundsTooHighTest() {
        Assertions.assertEquals(
            0,
            board.isTopOfColumn(7)
        );
    }

    @Test
    @DisplayName(
        "WHEN column 0 is called and no stones are "
      + "placed THEN return 6"
    )
    void emptyColumn0Test() {
        Assertions.assertEquals(
            6,
            board.isTopOfColumn(0)
        );
    }

    @Test
    @DisplayName(
        "WHEN column 6 is called and no stones are "
      + "placed THEN return 6"
    )
    void emptyColumn6Test() {
        Assertions.assertEquals(
            6,
            board.isTopOfColumn(6)
        );
    }
}
```

Listing C.2: CodeSnippet: Funktion setValuesFromSavecode

```
private void setValuesFromSavecode(String savecode){
    Tile[][][] spFeld;
    boolean player1Turn = false;
    boolean full = false;
    int row = -1;
    int column = -1;

    char[] scCh = savecode.toCharArray();//string zu array von chars
    int countA = 0;
    String v = "";
    boolean checkB = false;
    int r=0;
    int c=0;

    //iteration ueber jeden einzelnen char von String savecode
    for (char ch : scCh){
        //wenn a
        if (ch == 'a'){

            switch(countA){
            case 0:
                if (v == "1"){
                    player1Turn = true;
                }
                else if (v == "2"){
                    player1Turn = false;
                }
                break;
            case 1:
                row = Integer.parseInt(v);
                break;
            case 2:
                column = Integer.parseInt(v);
                break;
            case 3:
                if (v == "1"){
                    full = true;
                }
                else if (v == "0"){
                    full = false;
                }
                break;
            default:
                System.out.println(
                    "countA in werteVonSavecode ist groesser als 3: "
                    + countA
                );
                break;
            }

            countA += 1;
            v = "";              //zuruecksetzen von v
        }
```

```
            //wenn B
            else if (ch == 'B'){
                break;
            }
            else{
                v += Character.toString(ch);
            }
        }
    //fuellen des spielfeldVektors
        spFeld = new Tile[row][column];
        for (char ch : scCh){

            if (checkB == false){
                if (ch == 'B'){
                    checkB = true;

                }
            }
            else{

                spFeld[r][c] = new Tile(Character.getNumericValue(ch));

                if (c < column-1){
                    c += 1;
                }
                else{
                    c = 0;
                    r +=1;
                }

            }

        }

        this.setBoard(spFeld);
        this.setIsFull(full);
        this.setTurn(player1Turn);
    }
```

Literatur

[1] G. J. Myers, T. Badgett, T. M. Thomas und C. Sandler, *The art of software testing*. Wiley Online Library, 2004, Bd. 2.

[2] A. A. Sawant, P. H. Bari und P. Chawan, „Software testing techniques and strategies," *International Journal of Engineering Research and Applications (IJERA)*, Jg. 2, Nr. 3, S. 980–986, 2012.

[3] R. Fairley, „Tutorial: Static Analysis and Dynamic Testing of Computer Software," *Computer*, Jg. 11, Nr. 4, S. 14–23, 1978. DOI: 10.1109/C-M.1978.218132.

[4] V. Basili und R. Selby, „Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, Jg. SE-13, Nr. 12, S. 1278–1296, 1987. DOI: 10.1109/TSE.1987.232881.

[5] S. Nidhra und J. Dondeti, „Black box and white box testing techniques-a literature review," *International Journal of Embedded Systems and Applications (IJESA)*, Jg. 2, Nr. 2, S. 29–50, 2012.

[6] M. A. Jamil, M. Arif, N. S. A. Abubakar und A. Ahmad, „Software Testing Techniques: A Literature Review," in *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*, 2016, S. 177–182. DOI: 10.1109/ICT4M.2016.045.

[7] I. Jovanović, „Software testing methods and techniques," *The IPSI BgD transactions on internet research*, Jg. 30, 2006.

[8] S. Acharya und V. Pandya, „Bridge between black box and white box–gray box testing technique," *International Journal of Electronics and Computer Science Engineering*, Jg. 2, Nr. 1, S. 175–185, 2012.

[9] P. Runeson, „A survey of unit testing practices," *IEEE Software*, Jg. 23, Nr. 4, S. 22–29, 2006. DOI: 10.1109/MS.2006.91.

[10] S. Wappler und J. Wegener, „Evolutionary unit testing of object-oriented software using strongly-typed genetic programming," in *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, 2006, S. 1925–1932.

[11] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang und D. Marinov, „An extensive study of static regression test selection in modern software evolution," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, S. 583–594.

[12] C. Sadowski, E. Söderberg, L. Church, M. Sipko und A. Bacchelli, „Modern code review: a case study at google," in *Proceedings of the 40th international conference on software engineering: Software engineering in practice*, 2018, S. 181–190.

[13] „IEEE Standard Glossary of Software Engineering Terminology," *IEEE Std 610.12-1990*, S. 1–84, 1990. DOI: 10.1109/IEEESTD.1990.101064.

[14] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl und M. A. Vouk, „On the value of static analysis for fault detection in software," *IEEE transactions on software engineering*, Jg. 32, Nr. 4, S. 240–253, 2006.

[15] D. Sernow, expleo group, ISTQB u. a., *CTFL4.0 Grundlagen des Testens*, Presentation, expleo/ISTQB, unpublished, WS25/26, 2025.

[16] L. Baresi und M. Pezze, „An introduction to software testing," *Electronic Notes in Theoretical Computer Science*, Jg. 148, Nr. 1, S. 89–111, 2006.

[17] J. Hayes und A. Offutt, „Increased software reliability through input validation analysis and testing,“ in *Proceedings 10th International Symposium on Software Reliability Engineering (Cat. No.PR00443)*, 1999, S. 199–209. DOI: 10.1109/ISSRE.1999.809325.

[18] F. Deissenboeck und M. Pizka, „Concise and consistent naming,“ *Software Quality Journal*, Jg. 14, Nr. 3, S. 261–282, 2006.