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: 20. Januar 2026

Inhaltsverzeichnis

Kapitel 1

Einleitung

Fehler sind in der Softwareentwicklung nahezu nicht zu vermeiden. Jedoch können Entwickler versuchen, diesen so gut wie möglich vorzubeugen, damit diese nicht in entsprechende Produktionsumgebungen vordringen können. Dies tun sie mittels Softwaretests. In dieser Arbeit wird sich mit der Durchführung dieser Tests an einem Projekt auseinandergesetzt, dabei wird die Frage beantwortet, wie diese innerhalb eines Projektes umgesetzt werden können. 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 werden 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, sind 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 ist zu zweit spielbar. Das Ziel ist es, jeweils vier der eigenen Steine in einer Reihe zu platzieren und mit eigenen Steinen zu verhindern, dass der Gegner dasselbe tut.
Die Studenten des Moduls wurden im Rahmen der Lehrveranstaltung in verschiedene Teams aufgeteilt, um das Projekt gemeinsam anzufertigen. Die Teams waren Projektmanagement, Backend, Frontend und Testing[1]. Zur Erstellung wurde über GitHub kollaboriert. Alle im Rahmen dieser Arbeit erbrachten Leistungen sind in dieser GitHub-Repository zu finden. Im Fall des Projekts 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[2].
Im Hintergrund besteht das Spiel aus 4 Komponenten; Api, Logik, Frontend und Tests. Diese sind voneinander logisch in der Ordnerstruktur abgetrennt[3]. Der api Order 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 zu Zeigen angesehen, dass es keine Errors gibt[4]. Tatsächlich werden Tests jedoch verwendet, um Errors zu finden und Fehlverhalten der Software im Vorhinein zu verringern[5]. Auch hier im Projekt waren die bereits existierenden 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] Ich war Teil des Projektmanagementteams
[2] A.1, Screenshot aus dem Spiel
[3] A.2, Layout des Projektes zu Beginn der Tests
[4] G. J. Myers u. a., *The art of software testing.* Wiley Online Library, 2004, Bd. 2, p.10.
[5] 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. Der 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 ist anzumerken, 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 Vorhinein 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[6].
Hier ist eine bessere Projektstruktur nötig, welche eine einfachere Ausführung und Verarbeitung ermöglicht. 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 ist hier, dass es ein eigenes UI gibt, an welchem 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 zum Testen jedoch irrelevant. Ein positiver langfristiger Einfluss kann hingegen für Studenten existieren, welche zu alten Projekten zurückkehren und ihre Kompetenzen testen wollen.

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

Kapitel 3

Grundlagen

3.1 Statisch oder dynamisch Tests

Statische und dynamische Tests sind in jeder modernen Software angewandt. Jedoch worin besteht 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. Die zwei bedeutendsten sind Whitebox- und Blackboxtesting. Blackboxtesting wird auch funktionales Testen genannt[4] in diese ´´schwarze (nicht durchsichtige) Box´´ 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. Whiteboxtesting wird 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 kennen 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, da diese als einzige in Isolation getestet werden können[13]. Da die Umsetzung von OOP in diesem Projekt mangelhaft 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 dem Code durchgeführt, welcher sich verändert hat[14].
Um die Funktionalität von mehreren Komponenten, welche zusammen agieren, zu testen, werden **Integrationstests** verwendet[15]. Es gibt Arbeiten, welche eine Art coupling basiertes Testing vorschlagen[16]. In diesem Beispiel wird genannt, dass Fehler in einem Modul immer dazu führen, dass die vom fehlerhaften Modul abhängigen Module davon beeinträchtigt werden[17]. Tests des gesamtem Systems sind **Systemtests**, bei diesen werden Tests durchgeführt, welche die vollständige Funktionsweise der Software testen. Zudem ist die Aufgabe dieser Tests, die Fehler beim Zusammenfügen der Module entstanden[18].

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[19]. 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[20] oder GitHub.
Eine weitere Form ist die **statische Analyse**. Diese bezieht sich auf das Betrachten einer Komponente in Bezug auf Form, Struktur, Inhalt und Dokumentation[21]. Es besteht in diesem Zusammenhang auch die Möglichkeit der Automatisierung. Mithilfe dieser Tools kann der Code untersucht werden und mögliche Anormalitäten aufgespürt werden[22].

[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] vgl. F. Häser u. a., „Software paradigms, assessment types and non-functional requirements in model-based integration testing: a systematic literature review," in *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, 2014, S. 1–10, S. 1.

[16] Z. Jin und A. Offutt, „Coupling-based criteria for integration testing," *Software Testing, Verification and Reliability*, Jg. 8, Nr. 3, S. 133–154, 1998.

[17] Z. Jin und A. Offutt, „Coupling-based criteria for integration testing," *Software Testing, Verification and Reliability*, Jg. 8, Nr. 3, S. 133–154, 1998, S. 134f.

[18] G. J. Myers u. a., *The art of software testing*. Wiley Online Library, 2004, Bd. 2, S. 128.

[19] 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.

[20] 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.

[21] „IEEE Standard Glossary of Software Engineering Terminology," *IEEE Std 610.12-1990*, S. 1–84, 1990. DOI: 10.1109/IEEESTD.1990.101064, S. 70.

[22] 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.

3.4 Einfluss von KI/LLMs

Large Language Models (LLM) sind in den letzten Jahren zu einem durchaus nützlichen Tool in der Softwareentwicklung geworden. Jedoch sind diese Tools vor allem im Bereich des Softwaretestens aktuell von hoher Bedeutung und von großem Nutzen. Im Jahr 2024 wurde ein Paper veröffentlicht, welches aktuelle LLMs als nicht ausreichend, sogar für Unittests, ansieht und dies mit Modellen wie GPT 3 vergleicht[23]. Während diese Arbeit von vielen Ungenauigkeiten der LLMs redet[24], sehen andere ein größeres Risiko im menschlichen Fehler und LLMs und KI im Generellem als möglichen Weg, diese zu reduzieren[25]. Wieder andere Artikel sind weniger optimistisch und sehen LLMs als sinnvolles Tool, welches jedoch Limits hat. Beispielsweise wird das Wort Bank im Zusammenhang mit Geldinstitut oder Sitzgelegenheit verwendet. Es ist fraglich inwiefern eine LLM hier in der Lage ist den richtigen Kontext zu ermitteln[26]. Generell können LLMs jedoch von großem Nutzen sein, vor allem wenn es um das Schreiben von Tests geht, welche sehr einfach zu verstehen sind und für den Menschen nur repetetiv sein können. Dazu zählen bekannte Algorithmen in Unittests, jedoch ist auch hier menschliches Überprüfen noch notwendig.
Zudem ist es sinnvoll anzumerken, dass KI/LLMs im Zusammenhang mit Softwareentwicklung und Softwaretesten eine relativ neue Technologie ist und selbst die Arbeiten von 2024 und 2025 nicht mehr zwingend aktuell sind.

[23] J. Wang u. a., „Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, Jg. 50, Nr. 4, S. 911–936, 2024, S. 917ff.
[24] J. Wang u. a., „Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, Jg. 50, Nr. 4, S. 911–936, 2024, S. 917.
[25] M. Baqar und R. Khanda, „The Future of Software Testing: AI–Powered Test Case Generation and Validation," in *Intelligent Computing-Proceedings of the Computing Conference*, Springer, 2025, S. 276–300, S. 5f.
[26] M. Alenezi und M. Akour, „Ai-driven innovations in software engineering: a review of current practices and future directions," *Applied Sciences*, Jg. 15, Nr. 3, S. 1344, 2025, S. 21f.

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 Elemente und die damit verbundenen Anforderungen vor der tatsächlichen Umsetzung der Tests aufgelistet werden[2]. Eine solche Qualitätsplanung ist sinnvoll[3]. Diese Planung ist nur für`Board.java` vorgenommen worden 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- und 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 hergeleitet werden. So sind Kommentare sowohl in deutsch als auch englisch vorhanden 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 Tests existierte sind, auch viele Fehler unbehoben und in der Entwicklung ignoriert worden.

[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. Alle der Testfälle wurden im Rahmen dieser Arbeit erstellt.

5.1 Realisierung

Als Problem stellt sich direkt heraus, dass Tests nicht isoliert durchgeführt werden sollen[1]. Dies ist nicht möglich, da keine tatsächliche Entwicklung mehr stattfindet.
Aufgrund der relativ niedrigen Komplexität ist das Einrichten der Entwicklungs-/Testumgebung verhältnismäßig einfach. Anforderungen des Projektes ist 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 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 auf eine bestimmte Art und Weise verhalten soll. Das Standardboard hat ein Layout von 6x7; 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 das genaue Verhalten zu testen, fehlt im Code eine Möglichkeit, um Steine über oder unter dem Spielfeld zu platzieren.

[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.

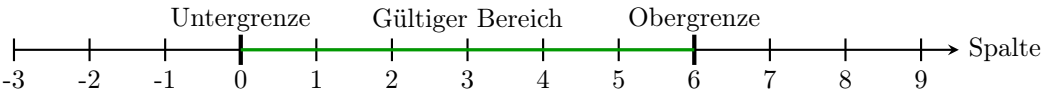

Abbildung 5.1: Grenzwerte der Inputs von `isTopOfColumn()` (im Fall des Boardlayouts 6x7)

In der Durchführung der Tests stellt sich heraus, dass das Verhalten unterhalb der Untergrenze und überhalb 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, dass kein Speicherstand existiert und versucht wird einen zu laden.
Um dem Ziel von Integration Tests zu folgen sind Zwischenwerte und interne Abläufe unwichtig. Von Relevanz ist lediglich das richtige Ergebnis, in diesem Fall das richtige Speichern und Auslesen.
Für den Testablauf muss nach jedem Durchlauf der Zustand der Dateien wieder zum Ausgangszustand zurückgebracht, da das Speichern des Spiels eine Datei erstellt, welche zuvor noch nicht da ist.
Das Diagramm 5.2 zeigt auf, wie der Speicher- & Ladeprozess im Spiel funktioniert. Dabei wird alles, was keinen direkten Einfluss auf den Speicher- & Ladeprozess, hat ignoriert.

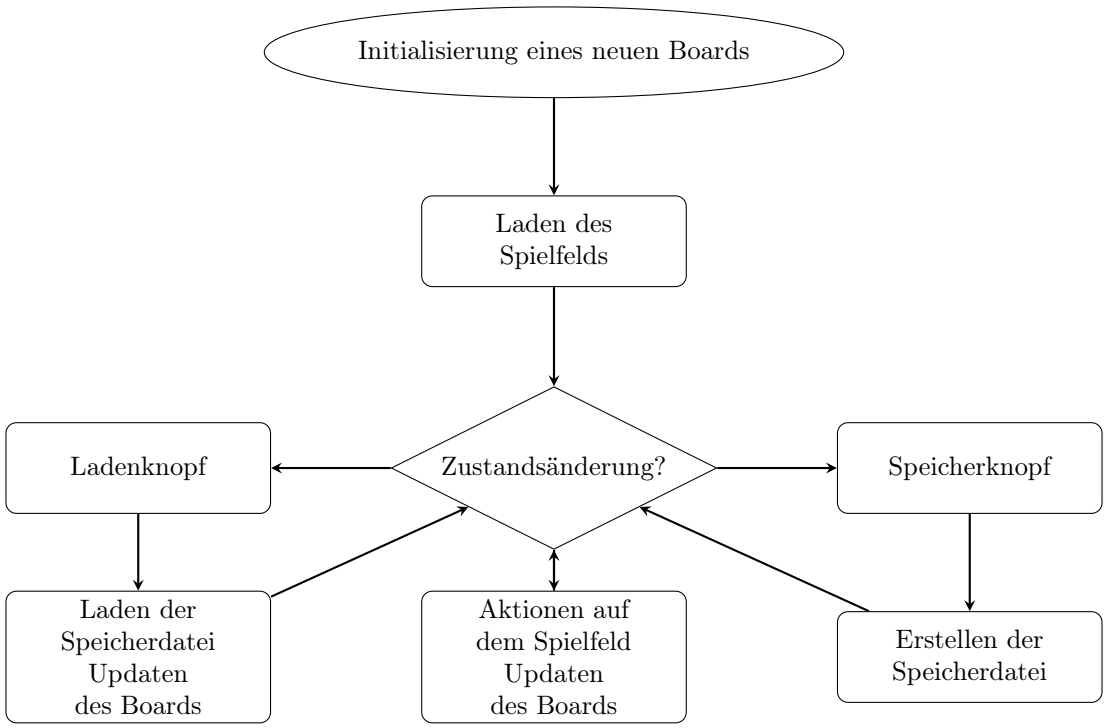

Abbildung 5.2: Flowchart für Speicher-/Ladeprozess

[3]siehe Code C.1

Aus der Figur 5.2 ist bereits ein Problem des Speicherkonzepts zu erkennen. Die Datei wird mit erstmaligem Speicher erstellt. Der Knopf zum Laden des Spiels hingegen ist bereits mit dem Öffnen des Spiels vorhanden und nutzbar. Dies kann zu Fehlern und undefiniertem 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 dies häufig zu komplex[4]. Dieses Projekt ist jedoch verhältnismäßig klein und somit auch in der Hinsicht leichter zu Testen.
Es besteht die Möglichkeit, entsprechende Tests mit Tools wie JUnit durchzuführen. Hierfür fehlen jedoch entsprechende Interfaces, um das Frontend ordentlich zu testen.

5.3 Statische Tests

Im Rahmen der statischen Tests wird eine Codeanalyse in Form eines Codereviews vorgenommen. Es wird der Commit `dec55bf` betrachtet, welcher die Funktion zum Speichern und Laden des Speicherstands hinzufügte. Zu betrachten ist der Code C.2 zum Dekodieren des Savefile-Inhalts.
Zuerst wird der Code nach seiner Lesbarkeit bewertet. Dabei kommt direkt ein Problem auf nämlich 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 bedeutsam, wenn nur ein Entwickler an dem Programm arbeitet, jedoch durchaus von größerer Bedeutung, wenn ein Team tätig ist, 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.

5.4 KI-basierte Tests

Der Einfluss von KI im Softwaretesten ist nicht zu bestreiten. Um die KI, zu testen wurden im Rahmen dieser Arbeit zwei Modelle zur Testgenerierung verwendet. Es wurde jeweils der Prompt D.1 verwendet. In diesem wird die KI aufgefordert, ausreichende Unittests für die Funktion `isTopOfColumn()` zu generieren. Als Kontext wurde stets die Datei `Board.java` verwendet.

5.4.1 GitHub Copilot mit GPT 5o-mini

GitHub Copilot ist ein in das Integrated Development Environment (IDE) integrierter KI Assistent, welcher auch in der Lage ist, mit dem Code und dem Terminal zu interagieren. So besteht die Möglichkeit, die KI den Code selber schreiben zu lassen und notfalls mit mehreren Iterationen diesen auch zu verbessern. Im Prompt D.1 ist erkennbar, wie eine Datei als Kontext verwendet werden kann. Dies geschieht mit dem Symbol #.
GPT 5o-mini war in der Lage, sowohl die Datei ordentlich zu lesen und zu verstehen, als auch entsprechende Ausführungen des Skripts durchzuführen. Hier versuchte es zuerst gradle, da das Projekt in der Ausführung jedoch kein gradle hat, hat die KI dann entsprechende Terminalanweisungen zur Ausführung dieser Tests mit javac und jar erstellt und diese selbstständig gestartet. Dies funktionierte auch und hat ähnlich wie die handgeschriebenen Unittests in Sektion 5.2.1 zwei Fehler gefunden. Anzumerken ist auch, dass Copilot Zugriff auch auf andere Dateien hat. So sind die Testnamen ähnlich der bereits bestehenden Tests. Möglicherweise ist dies auch ein Grund der Ähnlichkeit. Wie in Code D.3 ersichtlich wird, sind die Tests von einem Abdeckungspunkt deckungsgleich mit den bestehenden Tests. Somit war dies ein voller Erfolg.

5.4.2 Claude Sonnet 4.5 im Browser

Im Bereich der Softwareentwicklung hat Claude Sonnet den Ruf, sehr gut zum Schreiben von Code geeignet zu sein. Als Hypothese wurde formuliert, dass Claude Sonnet 4.5 auch gut für die Entwicklung von Tests sein. Auch hier wird der selbe Prompt verwendet. Der Unterschied besteht jedoch darin, dass anstatt von #Board.java der tatsächliche Inhalt von `Board.java` kopiert wird. Schnell entsteht hierbei ein Problem, so ist der Packagename nicht passend mit dem tatsächlichen Punkt, an dem die Datei eingesetzt werden soll. Dies ist schnell zu beheben. Ein weiteres Problem trat auf bei der Kompilierung der Tests, hier ist Claude Sonnet sich nicht von der Struktur des Projektes bewusst und erstellt seine Skripts gemäß eigener Erwartungen. Auch dafür musste zuerst der Prompt D.2 verwendet werden, da Claude das nicht selbstständig ausführt. Die entstandenen Resultate mussten dann stark angepasst werden, da sie nicht mit dem Projekt vereinbar waren. So wurde von einer älteren Version von JUnit ausgegangen, als in dem Projekt vorhanden ist.
Die Testfälle sind wieder sehr gut und stimmen mit den vorher kommenden Unittests überein. Hier hat Claude Sonnet ein ähnliches Resultat wie GPT 5o-mini in 5.4.1 und die eigenen Tests in 5.2.1. Der Erfolg ist beschränkter, zwar ist der Code richtig und befasst sich mit den korrekten Ansätzen, jedoch ist noch eine Menge eigenes Einwirken von Nöten.

5.4.3 Vergleich

Die grundlegenden Tests sind bei beiden Fällen nahezu gleich und decken die selben Äquivalenzklassen wie in der Grafik 5.1 ab. Auch werden in beiden Fällen Steine platziert, um das erwartete Verhalten bei steigender Steinezahl pro Spalte zu testen. Der Zugriff auf die gesamte Codebasis, welchen GitHub Copilot hat, ist zudem sehr stark in den Resultaten abzulesen. Im Prompt D.1 wird der entsprechenden LLM Zugang zu `Board.java` gewährt. GitHub Copilot hat jedoch mehr benutzt als ursprünglich erlaubt. So war Copilot in der Lage, die Packages richtig zu benennen, Testfälle mit Namensgebung ähnlich der existierenden Testfälle zu erstellen und kontextbasierte Anweisungen zu geben, welche die Testsuit selbstständig ausgeführt haben. Im Gegensatz dazu hat Claude Sonnet im Browser sehr schwach den Kontext verwendet und viel Architektur des Projektes angenommen, ohne das diese tatsächlich wahrheitsgemäß war.
Somit ist GitHub Copilot hier wesentlich besser geeignet. Fragwürdig ist nun, wo die Limits von Copilot und einem leistungsstarken, integrierten KI-Modell sind. Die angegebenen Testfälle sind verhältnismäßig sehr einfach zu lösen.

Kapitel 6

Probleme des Projektes

6.1 Nicht behobene Probleme

Viele Probleme des Projektes konnten im Verlauf der Arbeit nicht behoben werden, da sie zu umfangreich sind. So besteht weiterhin eine schlechte Grundstruktur mit wenigen und zu großen Dateien, welche zudem zu viele Tätigkeiten gleichzeitig ausführen. Auch wurden nicht alle Errors behoben. So besteht der Error, bei welchem der Nutzer einen Speicherstand laden kann, es jedoch keinen Speicherstand gibt und das Programm somit abstürzt. Zudem sind grundsätzliche Logikfehler im Code nicht behoben, da dies nicht im Rahmen der Arbeit war und das Projekt keine weitere Entwicklung erfährt.

6.2 Behobene Probleme

Obwohl noch viele Probleme existieren, konnten trotzdem einige behoben werden. Dies gelang vor allem im Hinblick auf die Testdeckung. Die bereits stehenden Tests welche vor dem Beginn dieser Arbeit existierten waren unzureichend. Das Problem bestand vor allem darin, dass es keine spezifischen Tests der einzelnen Bestandteile gab, sondern nur generelle Tests, in welchen mehrere Features auf einmal getestet wurden. Die neuen Tests sind wahrhafte Unittests und Integrationstests und fokussieren sich auf ihren jeweiligen Bereich. Dies sollte langfristig positive Folgen haben, da diese Tests auch in Zukunft verwendet werden können, um die Funktionalität zu überprüfen.

Kapitel 7

Fazit

Diese Arbeit zeigt, wie wichtig Softwaretests für Produkte sind, da diese sonst keinen Erfolg haben können. Auch wird ersichtlich, wie bedeutsam Rückblicke auf eigene Projekte sind, um seinen eigenen Fortschritt besser zu verstehen und aus alten Fehlern zu lernen. Eine Durchführung eine solchen Projekts sollte zukünftig auf eine andere Art erfolgen, da es nahezu nicht wartbar und nicht ausreichend modular ist. Für nächste Projekte sollte ein Plan, welcher auch genaue Testabläufe definiert, früh erstellt werden. Auch statische Reviews in Form von Codereviews sollten eingeführt werden, um eine hohe Qualität zu garantieren. Generell sind im Verlauf der Arbeit häufig Fehler aufgetreten, welche vor allem der Unerfahrenheit der Entwickler zuzurechnen waren.
Auch zeigt sich, dass vor allem für einfache Tests KI gewaltige Zeit- und Ressourceneinsparungen erzeugt. Solche Tests, wie in Abschnitt 5.2.1, werden in Zukunft wahrscheinlich nicht mehr menschlich geschrieben sein.

Appendices

Anhang A

Grafiken

Abbildung A.1: Bild des Spiels

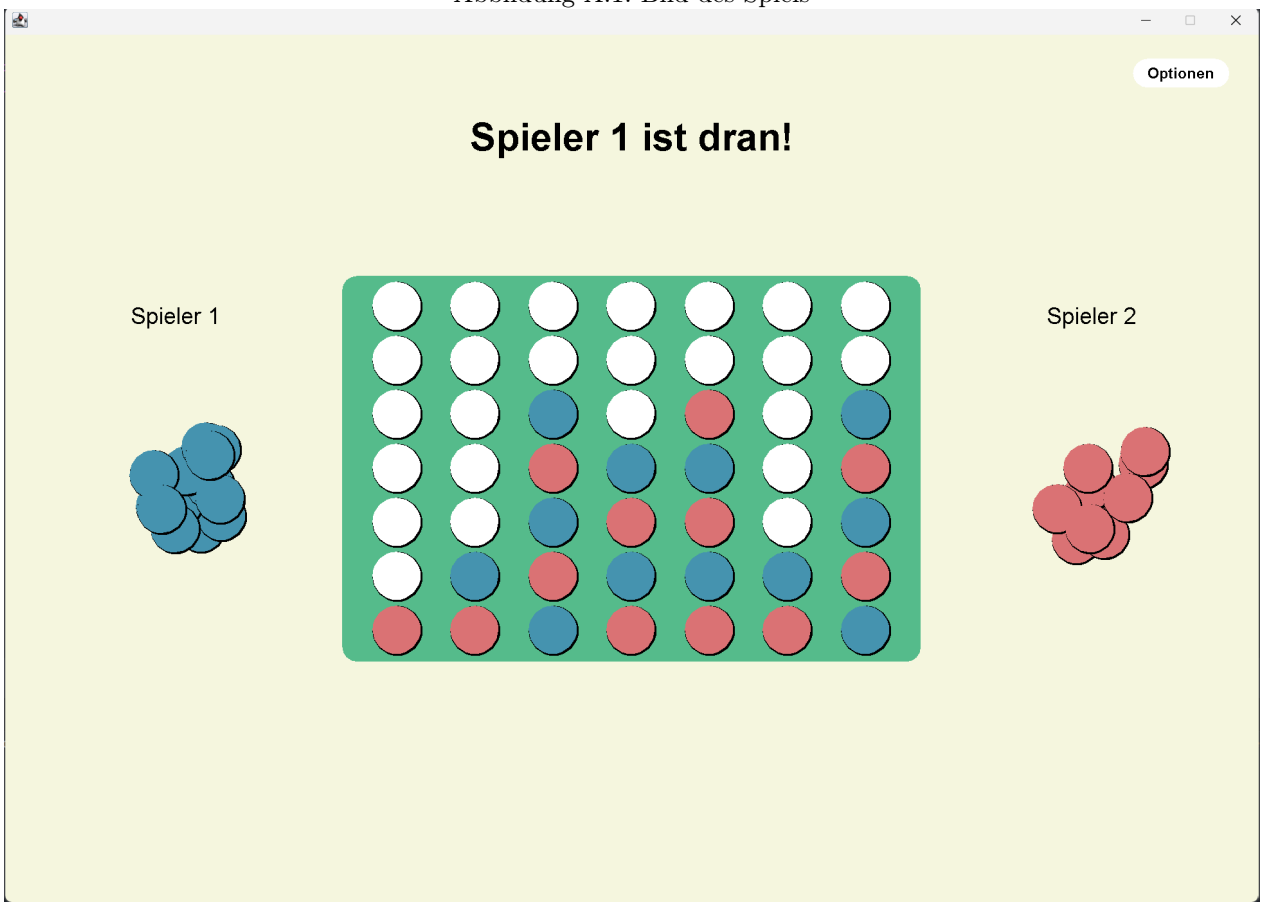

Abbildung A.2: Projektstruktur vor Verbesserungen (Einstiegspunkt in rot)

```
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);
    }
```

Anhang D

KI-Nutzung

Anmerkung: Die Nutzung von KI erfolgte ausschließlich an Punkten, welche sich mit KI auseinandersetzten und KI Nutzung benötigen. Alle verwendete Prompts sind hier zu sichten.

Listing D.1: Prompt zum erstellen der Tests

```
For the function isTopOfColumn in #file:Board.java
create unit tests with JUnit in this file.
```

Listing D.2: Claude Sonnet 4.5 Prompt um die Ausführung zu gewähren

```
How do i start those tests,
 in native java with JUNIT 5 jar
```

Listing D.3: Code von GPT 5o-Mini

```
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 Gpt5oMiniTests {

    BoardTestInterface board;

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

    @Test
    @DisplayName("WHEN column 0 is empty THEN return number of rows (6)")
    void emptyColumnZero() {
        Assertions.assertEquals(6, board.isTopOfColumn(0));
    }

    @Test
    @DisplayName("WHEN column 2 has 3 stones placed THEN return 3")
    void partlyFilledColumn() {
        board.placeStone(2);
        board.placeStone(2);
        board.placeStone(2);

        Assertions.assertEquals(3, board.isTopOfColumn(2));
    }

    @Test
    @DisplayName("WHEN column 3 is filled completely THEN return 0")
    void fullColumn() {
        for (int i = 0; i < board.getrows(); i++) {
            board.placeStone(3);
        }
        Assertions.assertEquals(0, board.isTopOfColumn(3));
    }

    @Test
    @DisplayName("WHEN negative column index is given
    THEN return 0 (out of bounds)")
    void negativeColumnIndex() {
        Assertions.assertEquals(0, board.isTopOfColumn(-1));
    }

    @Test
    @DisplayName("WHEN column index is greater than last column
    THEN return 0 (out of bounds)")
```

```
    void tooLargeColumnIndex() {
        // default board has 7 columns, last index is
        // 6 -> use 7 as out of bounds
        Assertions.assertEquals(0, board.isTopOfColumn(7));
    }
}
```

Listing D.4: Test mit Claude Sonnet 4.5

```
package logic;

import api.TileInterface;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.DisplayName;
import static org.junit.jupiter.api.Assertions.*;

class BoardIsTopOfColumnTest {

    private Board board;

    @BeforeEach
    void setUp() {
        // Create a standard 6x7 board before each test
        board = new Board();
    }

    @Test
    @DisplayName("Empty column should return the bottom row + 1")
    void testEmptyColumn() {
        // All columns in a new board are empty
        // For a 6-row board, bottom row is index 5, so should return 6
        int result = board.isTopOfColumn(0);
        assertEquals(6, result,
        "Empty column should return board.length (6)");
    }

    @Test
    @DisplayName("Column with one stone should return correct position")
    void testColumnWithOneStone() {
        // Place one stone in column 3
        board.placeStone(3);

        // After placing one stone at the bottom,
        // top should be at row 5 (index 4 + 1)
        int result = board.isTopOfColumn(3);
        assertEquals(5, result,
        "Column with one stone should return 5");
    }

    @Test
    @DisplayName("Column with multiple stones should return correct position")
    void testColumnWithMultipleStones() {
        // Place three stones in column 2
        board.placeStone(2);
```

```
        board.placeStone(2);
        board.placeStone(2);

        // After placing three stones, top should be at row 3 (index 2 + 1)
        int result = board.isTopOfColumn(2);
        assertEquals(3, result,
        "Column with three stones should return 3");
    }

    @Test
    @DisplayName("Almost full column should return 1")
    void testAlmostFullColumn() {
        // Fill column 4 with 5 stones (one space left at top)
        for (int i = 0; i < 5; i++) {
            board.placeStone(4);
        }

        int result = board.isTopOfColumn(4);
        assertEquals(1, result,
        "Almost full column should return 1");
    }

    @Test
    @DisplayName("Completely full column should return 0")
    void testFullColumn() {
        // Fill column 1 completely (6 stones)
        for (int i = 0; i < 6; i++) {
            board.placeStone(1);
        }

        int result = board.isTopOfColumn(1);
        assertEquals(0, result,
        "Full column should return 0");
    }

    @Test
    @DisplayName("First column (index 0) should work correctly")
    void testFirstColumn() {
        int result = board.isTopOfColumn(0);
        assertEquals(6, result,
        "First column when empty should return 6");

        board.placeStone(0);
        result = board.isTopOfColumn(0);
        assertEquals(5, result,
        "First column with one stone should return 5");
    }

    @Test
    @DisplayName("Last column (index 6) should work correctly")
    void testLastColumn() {
        int result = board.isTopOfColumn(6);
        assertEquals(6, result,
        "Last column when empty should return 6");
```

```
        board.placeStone(6);
        result = board.isTopOfColumn(6);
        assertEquals(5, result,
        "Last column with one stone should return 5");
    }

    @Test
    @DisplayName("Different columns should not affect each other")
    void testColumnsAreIndependent() {
        // Place stones in column 2
        board.placeStone(2);
        board.placeStone(2);

        // Column 3 should still be empty
        int resultColumn3 = board.isTopOfColumn(3);
        assertEquals(6, resultColumn3,
        "Adjacent empty column should still return 6");

        // Column 2 should show 2 stones
        int resultColumn2 = board.isTopOfColumn(2);
        assertEquals(4, resultColumn2,
        "Column with stones should return correct value");
    }

    @Test
    @DisplayName("Custom board size should work correctly")
    void testCustomBoardSize() {
        // Create a custom 4x5 board (4 rows, 5 columns)
        Board customBoard = new Board(5, 4);

        // Empty column should return 4 (number of rows)
        int result = customBoard.isTopOfColumn(0);
        assertEquals(4, result,
        "Empty column in 4-row board should return 4");

        // Fill the column completely
        for (int i = 0; i < 4; i++) {
            customBoard.placeStone(0);
        }

        result = customBoard.isTopOfColumn(0);
        assertEquals(0, result,
        "Full column in custom board should return 0");
    }

    @Test
    @DisplayName(
        "isTopOfColumn should work correctly after mixed placements"
        )
    void testMixedPlacements() {
        // Create a pattern:
        // column 0: 2 stones,
        // column 1: 4 stones,
```

```
        // column 2: 1 stone
        board.placeStone(0);
        board.placeStone(1);
        board.placeStone(0);
        board.placeStone(1);
        board.placeStone(2);
        board.placeStone(1);
        board.placeStone(1);

        assertEquals(4, board.isTopOfColumn(0),
        "Column 0 should have 2 stones (return 4)");
        assertEquals(2, board.isTopOfColumn(1),
        "Column 1 should have 4 stones (return 2)");
        assertEquals(5, board.isTopOfColumn(2),
        "Column 2 should have 1 stone (return 5)");
        assertEquals(6, board.isTopOfColumn(3),
        "Column 3 should be empty (return 6)");
    }

    @Test
    @DisplayName(
        "Manually setting board state should work with isTopOfColumn"
    )
    void testManuallySetBoard() {
        TileInterface[][] customBoard = new Tile[6][7];

        // Initialize all tiles as empty
        for (int i = 0; i < 6; i++) {
            for (int j = 0; j < 7; j++) {
                customBoard[i][j] = new Tile();
            }
        }

        // Manually set bottom 3 positions of column 2 to be occupied
        customBoard[5][2].setStatus(1);
        customBoard[4][2].setStatus(2);
        customBoard[3][2].setStatus(1);

        board.setBoard(customBoard);

        int result = board.isTopOfColumn(2);
        assertEquals(3, result,
        "Column with manually set 3 stones should return 3");
    }
}
```

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] F. Häser, M. Felderer und R. Breu, „Software paradigms, assessment types and non-functional requirements in model-based integration testing: a systematic literature review," in *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, 2014, S. 1–10.

[13] Z. Jin und A. Offutt, „Coupling-based criteria for integration testing," *Software Testing, Verification and Reliability*, Jg. 8, Nr. 3, S. 133–154, 1998.

[14] 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.

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

[16] 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.

[17] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang und Q. Wang, „Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, Jg. 50, Nr. 4, S. 911–936, 2024.

[18] M. Baqar und R. Khanda, „The Future of Software Testing: AI–Powered Test Case Generation and Validation," in *Intelligent Computing-Proceedings of the Computing Conference*, Springer, 2025, S. 276–300.

[19] M. Alenezi und M. Akour, „Ai-driven innovations in software engineering: a review of current practices and future directions," *Applied Sciences*, Jg. 15, Nr. 3, S. 1344, 2025.

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

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

[22] 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.

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