DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

Chess of Duty

Advanced Software-Engineering

im Rahmen der Prüfung zum

Bachelor of Science (B.Sc.)

des Studienganges Informatik

an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

Clemens Richter & Johannes Peters

Abgabedatum:	30. April 2023
Bearbeitungszeitraum:	04.10.2022 - 30.04.2023
Matrikelnummer, Kurs:	7661745 & 5802185, TINF20B1
Betreuer der Arbeit:	Herr Daniel Lindner

Inhaltsverzeichnis

1 Einleitung

Im Rahmen des Moduls „Advanced Software Engineering“ wurde ein Schachspiel als Projektgrundlage ausgewählt. Chess of Duty ist ein Offline-Multiplayer-Schachspiel für zwei Personen. Das Hauptziel des Programmentwurfs besteht darin, Schach gemäß den Standardregeln zu implementieren.

Der Nutzen des Schachspiels für unsere Kunden entspricht dem von anderen Videospielen. Die Anwendung dient ausschließlich der Unterhaltung der Nutzer. Zusätzlich können die Anwender ihre strategischen Fähigkeiten und logisches Denken trainieren.

Das Schachspiel wird objektorientiert konzipiert und in Processing programmiert. Processing ist eine Open-Source-Programmiersprache, die auf Java basiert und einen besonderen Schwerpunkt auf die einfache Erstellung von Grafiken und Animationen setzt. Dadurch eignet sich Processing besonders für die Gestaltung interaktiver Benutzeroberflächen.

Hinweis

Das GitHub-Repository ist mit folgendem Link erreichbar:
https://github.com/clemens1403/AdvSWE

1.1 Vorwort

Sehr geehrter Herr Lindner, anbei finden Sie unsere Ausarbeitung für den Programmentwurf aus dem fünften und sechsten Semester. Während der Projektarbeit sind verschiedene Herausforderungen aufgetreten, insbesondere bei der Projektwahl und der Auswahl des Technologiestacks. Für die Umsetzung einer Clean Architecture hätte sich im Nachhinein ein Verwaltungsprogramm als geeigneter erwiesen. Zudem gestaltete sich die Verwendung von Processing in einem Java-Projekt leider weniger intuitiv als von den Entwicklern angenommen. Insbesondere die Integration von Processing in IntelliJ bereitete längere Zeit Probleme.

Trotz dieser Schwierigkeiten ist das Ergebnis des Programmentwurfs ein Projekt, das die in der Vorlesung vermittelten Methoden und Prinzipien bestmöglich umsetzt. Sollte etwas nicht umgesetzt worden sein, wird darauf zumindest eingegangen.

Die Implementierung der Schachlogik war äußerst umfangreich, weshalb einige Funktionen nicht unterstützt werden, beispielsweise das Schlagen en passant. Die Durchführung eines einfachen Schachspiels ist jedoch problemlos möglich.

1.2 Inbetriebnahme

Für das Projekt, das auf GitHub einsehbar ist, wurde keine JAR-Datei erstellt. Das Projekt kann jedoch problemlos in IntelliJ oder Eclipse ausgeführt werden. Alle erforderlichen Abhängigkeiten sind in der pom-Datei des Maven-Projekts aufgeführt und im Repository verfügbar.

Es gibt eine wichtige Anmerkung, die beim Importieren der Abhängigkeiten und beim Ausführen des Projekts beachtet werden sollte. Abhängig von der verwendeten IDE und der Bildschirmgröße kann es vorkommen, dass Processing die Darstellung unterschiedlich skaliert, wodurch das Bild verzerrt erscheinen kann. Dieses Problem tritt hauptsächlich bei kleinen Laptop-Bildschirmen auf, während größere Desktop-Monitore davon nicht betroffen sind.

Um das Problem zu beheben, muss in der Run-Konfiguration der Klasse „Chess of Duty“ eine VM-Option angegeben werden. Diese Option kann ausgewählt werden, indem man auf „Modify Options“ im Reiter „Build and Run“ klickt und das entsprechende Feld auswählt.

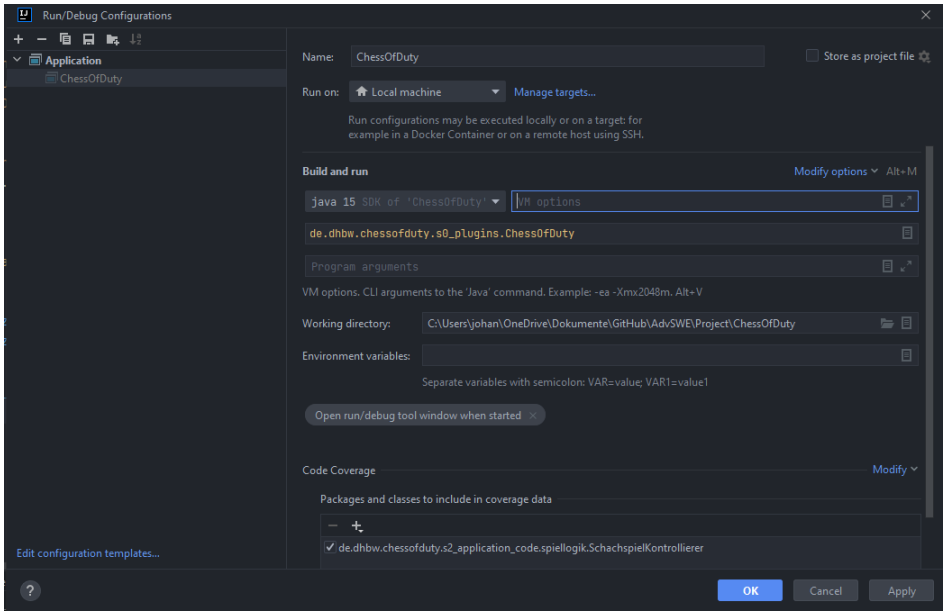


Abbildung 1.1: Eingabefeld für VM-Options

Sobald in das Feld Eingaben getätigt werden können, muss folgender Parameter eingetragen und gespeichert werden, bevor die Konfigurationsübersicht geschlossen und das Programm normal ausgeführt werden kann.

```
-Dsun.java2D.uiScale=1.0
```

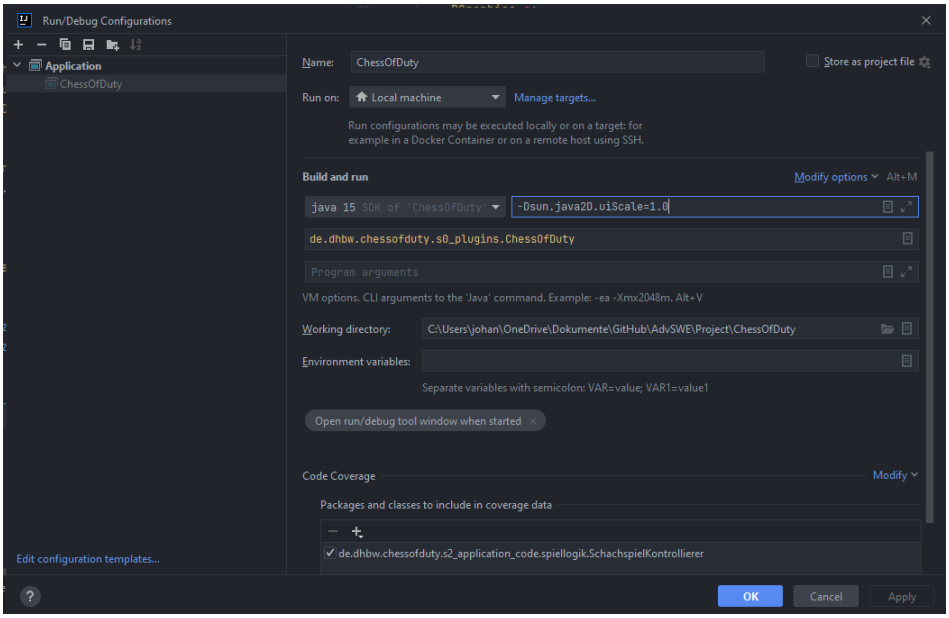


Abbildung 1.2: VM-Options-Parameter zur richtigen Skalierung

2 Domain Driven Design

2.1 Ubiquitous Language

Die Ubiquitous Language bezeichnet das Vokabular, welches zwischen Domänenexperten und Entwicklern, wodurch gewährleistet werden soll, dass die gleichen Begriffe in der Domäne und im Sourcecode verwendet werden und Missverständnisse minimiert werden.

Der Auftraggeber des Programmentwurfs ist ein deutscher Kunde. Obwohl in der Programmierung Englisch als die Standardsprache angenommen wird, wird aufgrund der Kundenlokalität die Projektsprache als „Deutsch“ festgelegt. Dadurch wird versucht, die meisten Ausdrücke aus der Domäne ins Deutsche zu übernehmen.

Von der deutschen Bezeichnung ausgeschlossen sind in diesem Projekt jedoch allgemein anerkannte Java-Konventionen wie „getter“ und „ setter“. IDEs wie Eclipse oder Intelij sind in der Lage diese speziellen Funktionsarten automatisch zu generieren und um diese Funktion zu nutzen, werden diese Funktionsbezeichnungen auch auf Englisch akzeptiert. Weiterhin von der deutschen Bezeichnung ausgeschlossen sind Funktionsnamen, Objekte und Variablen, die durch die Verwendung von Processing vorgegeben sind. Dazu gehören beispielsweise Funktionen wie „setup(), settings()“ oder „draw()“.

2.2 Domainenbausteine

Aus der Ubiquitous Language ergeben sich nun die folgenden taktischen Muster des Domain Driven Designs mit den entsprechenden Objekten. Dazu wird versucht die Quelldomaine, Schachspiel, so detailgetreu wie möglich in das Programm zu übertragen.

2.2.1 Value Objects

Ein Value Object repräsentiert ein bestimmtes unveränderliches Objekt, das einen bestimmten Wert besitzt und keine eigene Identität aufweist. Es definiert sich durch seine Eigenschaften und wird daher als austauschbar und vergleichbar angesehen.

Im Standardschach repräsentiert ein **Feld** eine Position, die durch eine Kombination aus *Spalte*, *Zeile* und *Farbe* definiert wird. Schachfelder haben keinen Lebenszyklus und werden aus diesem Grund als Value Objects betrachtet. Es ist von grundlegender Bedeutung für die Funktion eines Schachspiels, dass Schachfelder als unveränderliche Objekte behandelt werden, da ein Schachspiel auf einer „festen Unterlage“ gespielt wird.

Ein **Schachbrett** kann ebenfalls als Value Object betrachtet werden, da es einen unveränderlichen Zustand repräsentiert. In der domänenspezifischen Abbildung eines Schachspiels wird es als übergeordnete Verwaltungsstruktur der einzelnen Spielfelder betrachtet. Es besteht aus 64 Schachfeldern, die als zweidimensionale Spielebene dargestellt werden. Die Gleichheit von zwei Schachbrettern basiert auf der Gleichheit ihrer Felder, unabhängig von ihrer tatsächlichen Instanz. Da die Felder als Value Objects behandelt werden und ein Schachbrett zu Beginn eines Spiels initial erzeugt und danach unverändert bleibt, wird diese Klasse auch als Value Object geführt. Die Integrität des Objekts muss während des Spiels gewahrt bleiben, da es keine Verhaltensänderungen aufweist.

Ein **Schachzug** setzt sich aus einer ausgewählten *Figur*, der *Startposition*, der *Endposition* und textuellen Darstellung des Zugs zusammen. Bei der textuellen Darstellung werden Angaben darüber, ob eine andere Figur geschlagen wird, ob durch den Zug Schach geboten wird, ob eine Bauernumwandlung stattfindet oder ob eine Rochade durchgeführt wird, berücksichtigt. Die Implementierung eines Schachzugs als eigene Klasse dient der Protokollierung der gespielten Schachpartie. Ein Schachzug wird als Value Objekt klassifiziert, da er keinen erkennbaren Lebenszyklus vorweisen kann. Sobald ein Spieler in einem Spiel eine Figur von Position A zu Position B bewegt, wird eine Instanz der Klasse Schachzug erstellt und kann nicht nachträglich bearbeitet werden, da auch im Spiel ein Schachzug nicht zurückgenommen oder geändert werden kann.

2.2.2 Entities

Eine Entity ist ein Objekt, das innerhalb der Domäne eine einzigartige Identität besitzt und sich im Laufe der Zeit verändern kann. Entities werden durch ihre Identität und nicht durch ihre Attribute definiert und können einen Lebenszyklus vorweisen.

Im Domainencode existiert die Klasse **Figur**, die als abstrakte Oberklasse für alle Figuren in einem Schachspiel dient. Da die Figur als *abstract class* implementiert ist, kann sie nicht instanziiert werden. Aus diesem Grund wird nicht die Klasse Figur selbst, sondern alle Spielfiguren, die von Figur erben, als Entitäten zu betrachten. Zu diesen Spielfiguren gehören **König, Dame, Turm, Läufer, Springer** und **Bauer**. Jede Figur kann über das Attribut *position* eindeutig identifiziert werden. Der Schlüssel, den das genannte Attribut darstellt, ist kein natürlicher Schlüssel, da es kein wirklicher Identifikator aus der Domaine ist, sondern eine für den Betrachtungszeitpunkt eindeutige Eigenschaft der Figur. Da auf einem Feld im Schach immer nur eine Figur stehen kann, ist die Eindeutigkeit durch die Positionen gewährleistet. Wenn eine Figur auf ein Feld zieht, auf dem sich bereits eine andere Figur befindet, wird die Figur, welche ursprünglich auf dem Feld stand, geschlagen und aus dem Spiel entfernt. Somit steht letztendlich nur eine Figur auf dem Feld. Innerhalb der Domäne „Schach“ ist dieser Eigenschaftsschlüssel mit jedem Schachzug veränderlich, da sich die Schachfiguren auf dem Spielfeld bewegen können. Die Verwendung eines Surrogatschlüssels wäre grundsätzlich auch möglich, aber in diesem Fall wird sich so gut es geht an der Domaine orientiert. Im analogen Schach wird der Figurentyp und die Position zur eindeutigen Identifizierung verwendet. Die Farbe kann indirekt durch die Anordnung der Angaben bestimmt werden. Jede Schachfigur besitzt einen eigenen Lebenszyklus. Sobald ein Schachspiel startet, werden alle Figuren in ihrer Startposition instanziiert. Während des Spiels können die Figuren nahezu unbegrenzt bewegt werden. Wenn eine Figur geschlagen wird, wird sie wieder gelöscht.

Obwohl ein Schachzug ein Value Object ist, wird in diesem Projekt der **Spielzug** als Entity gewertet, obwohl nur zwei Schachzüge mit einer ID zusammengefasst werden. Der Spielzug lässt sich eindeutig durch eine *zugNummer* identifizieren, die mit jedem Spielzug um eins inkrementiert wird. Da diese Art der Zählung und Zugidentifizierung ebenfalls in der Domaine verwendet wird, ist die Einordnung als natürlicher Schlüssel passender als eine Bezeichnung als Surrogatschlüssel. Der Grund, warum der Spielzug als Entity und nicht als Value Object gehandelt wird, liegt am Lebenszyklus des Objekts.

Nach dem Erstellen einer Instanz des Spielzugs wird zuerst der Zug des Spielers „Weiß“ gesetzt. Im zweiten Schritt wird der schwarze Zug hinzugefügt, bevor der Schachzug zur Protokollierung der Schachpartie verarbeitet und final wieder verworfen wird.

Die Klasse **Schachspiel** sollte als Entity behandelt werden, da sie eine eindeutige Identität besitzt und sich im Verlauf der Schachpartie in verschiedensten Zuständen befindet. Sie bildet die übergeordnete logische Ebene, welche eine bestimmte Figurenkonstellation auf dem Schachbrett zu einem bestimmten Zeitpunkt zusammenführt. Im Schach sind pro Zug durchschnittlich 35 unterschiedliche Züge möglich, was die Anzahl der Stellungen in einem Schachspiel exponentiell steigen lässt, abhängig von der Anzahl bereits gespielter Züge, der Positionierung der Figuren und der Anzahl der geschlagenen Figuren. Obwohl die Ausgangslage der Figuren immer gleich ist, unterscheidet sich die Partie Schach individuell sehr stark in ihrem Verlauf. Um die Schachspiele eindeutig zu identifizieren, bietet sich die Verwendung eines Surrogatschlüssels an. In der verwendeten Programmiersprache Processing, einer Java-Erweiterung, eignet sich hierfür die Verwendung einer *UUID*.

2.2.3 Aggregate

Aggregate ermöglichen die Verwaltung von Entities und Value Objects als eine zusammengehörende Einheit. Ein Aggregat besteht dabei aus mindestens einer Entität und kann weitere Entitäten und Value Objects enthalten.

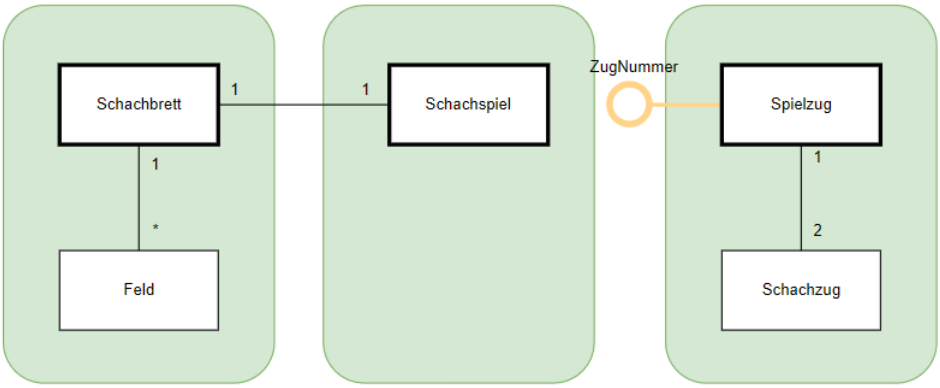


Abbildung 2.1: Ausschnitt aus dem Domainenmodell eines Schachspiels

In der Domaine „Schach“ können die Klassen **Schachbrett** und **Feld** als Aggregat zusammengefasst werden, da sie logisch miteinander verknüpft sind. Für ein Schachbrett werden 64 Instanzen der Klasse Feld in einer zweidimensionalen Datenstruktur

gespeichert. Da über das Schachbrett in der Domaine die gesamte Handlung des Spiels stattfindet (Figuren werden über das Schachbrett bewegt), hat die Klasse Schachbrett eine zentrale Rolle. Um die feste Anordnung der Felder im zweidimensionalen Rahmen des Spielfeldes durch Zeilen und Spaten positionsgetreu abzufragen, kann auf jedes Feld auf dem Schachbrett nur über das übergeordnete Schachbrett zugegriffen werden. Durch diese Art des Zugriffs fungiert die Klasse Schachbrett automatisch als *Root-Entity* für die Klasse Feld. Zwischen der Root-Entität Schachbrett und der Entität Schachspiel kann die Assoziation nicht durch eine Entity-ID gelöst werden, da es sich bei dem Schachbrett um ein Value Object handelt und einem Schachbrett daher keine ID zugewiesen wird.

Die Klasse **Spielzug** dient ebenfalls als *Root-Entity* für das Aggregat, das aus den Klassen Spielzug und **Schachzug** besteht. In der Domaine kann jeder Spieler, wenn er an der Reihe ist, einen Schachzug ausführen, indem er eine Figur von Position A nach Position B bewegt. Ein Spielzug fasst jeweils zwei aufeinanderfolgende Schachzüge von Spieler „Weiß“ und Spieler „Schwarz“ zusammen. Die schriftliche Protokollierung einer Partie Schach erfolgt immer über die Spielzüge und gibt jeweils die einzelnen Schachzüge der unterschiedlichen Spieler an. Auf diese Weise kann man auf die einzelnen Schachzüge nur über die Spielzüge zugreifen, was für eine Betrachtung als Aggregat spricht.

2.2.4 Repositories

Im Domain Driven Design werden Repositories genutzt, um Daten in einem Projekt zu persistieren und zu laden.

Im Falle des programmierten Schachspiels soll nach jedem abgeschlossenen Spiel ein lesbares Protokoll in eine eigene Datei exportiert werden. Obwohl die Daten nie gelesen werden, ist es erforderlich, dass für die Klasse Spielzug ein Repository verwendet wird, um die Spielzüge nacheinander in der Protokolldatei zu persistieren.

In den Vorlesungsfolien wird beschrieben, dass meist zu jedem Aggregat auch ein entsprechendes Repository existiert. Für das Aggregat Schachbrett ist dies nicht der Fall. Das Schachbrett muss in keinem Fall persistiert werden und darum wird kein Repository benötigt.

2.2.5 Domain Service

Ein Domain Service ist ein Konzept, das genutzt wird, um Logik abzubilden, die nicht direkt mit einer Entität oder einem Aggregat verbunden ist.

Im Schachspiel sind alle Figuren als Entitäten zu betrachten, wobei jede Figur eigene Regeln hat, nach denen sie sich auf dem Schachbrett bewegen darf. Die Funktionalität, die die möglichen Bewegungen der verschiedenen Figurtypen ausgehend von ihrer aktuellen Position wiedergibt, befindet sich in den Klassen **Bewegungsmatrizen** und **Bewegungsrichtung**. Beide Klassen können daher als Services in der Domaine betrachtet werden.

3 Clean Architecture

Das Projekt implementiert Clean Architecture durch die Verwendung von Modulen für jede Schicht. Dadurch wird sichergestellt, dass nur die äußeren Schichten auf die inneren Schichten zugreifen können und nicht umgekehrt. Die Verwendung von Schichten ermöglicht eine klare Trennung zwischen der Anwendung und der Infrastruktur, was die Wiederverwendbarkeit, Testbarkeit und Weiterentwicklung der einzelnen Komponenten erleichtert. Zudem ist es einfacher, einzelne Infrastrukturkomponenten auszutauschen, insbesondere je weiter sie von der Kernfunktionalität entfernt sind.

Für die Implementierung eines sauberen Schichtenmodells in einem Java-Projekt werden in der Regel separate Projekte für jede Schicht erstellt und dann in einer Entwicklungsumgebung wie IntelliJ mithilfe von Maven zusammengeführt. Maven kann auch verwendet werden, um Processing als festgelegten Technologiebestandteil des Projektes in IntelliJ zu integrieren. Allerdings stellte sich heraus, dass der Import der Processing-Dependency unzuverlässig und fehleranfällig ist. Um die Umsetzung der Schichtenarchitektur nicht durch ständige Maven-Installationsfehler zu behindern, wurde entschieden, die einzelnen Schichten als Packages innerhalb eines Projekts einzubinden.

Ursprünglich war das Projekt nicht in Schichten strukturiert, sondern der Code befand sich in einem einzigen Ordner, entwickelt nach dem Prinzip „quick and dirty“. Die Überarbeitung des unstrukturierten Codes und die Umstellung auf die Schichtenarchitektur erwiesen sich als komplex und zeitaufwändig.

3.1 Schicht 4 - Abstraction Code

Diese Schicht bildet den Kern der Applikation und wird in der Regel durch die verwendete Programmiersprache repräsentiert. Zum Beispiel werden in Java Klassen wie *String*, *Boolean* oder *Integer* verwendet, aber auch übergeordnete abstrakte Konzepte wie Algorithmen oder eigene Datentypen. Im vorliegenden Projekt konnte kein abstrakter Code identifiziert werden. Alle mathematischen Konzepte, einschließlich der Darstellung von Bewegungsmustern einzelner Figuren, wurden als Domain Service in der dritten Schicht umgesetzt.

3.2 Schicht 3 - Domain Code

Der Domain-Code implementiert alle Bausteine der Domain, die die Quelldomäne beschreiben oder damit direkt verbunden sind und bereits im Kapitel 2 ausgearbeitet wurden. Diese Schicht enthält Value Objects wie *Feld*, *Schachbrett* und *Schachzug*, die Entities *Bauer*, *Läufer*, *Springer*, *Turm*, *Dame*, *König* und *Schachspiel*, sowie das eine Repository *SpielzugRepository* und die beschriebenen Domain-Services *Bewegungsmatrizen* und *Bewegungsrichtung*.

3.3 Schicht 2 - Application Code

Der Application Code implementiert die gesamte Anwendungslogik des Schachspiels, die mithilfe der Domain-Bausteine die Spielfunktionen umsetzt. Die Anwendungsfälle des Projekts sind über Services realisiert, wobei für jede Entity und jedes Value Object ein eigener Service vorhanden ist, der die Arbeit mit diesen Elementen ermöglicht. Aufgrund der Verwendung der Übiquitous Languageïn deutscher Sprache werden die Services in diesem Projekt entsprechend als "Dienst"bezeichnet.

Die in dieser Schicht implementierten Anwendungsfälle umfassen:

- **Neues Schachspiel beginnen:** Um ein Spiel zu starten, wird der vorherige Spielstand, falls vorhanden, gelöscht und ein neues Spiel mit der Grundkonfiguration der Figuren initialisiert.

- **Schachfigur bewegen:** Jede Figur kann sich basierend auf ihrer Art (Bauer, Dame usw.) auf dem Schachbrett bewegen. Abhängig von ihrer aktuellen Position werden alle möglichen Felder ermittelt, die die Figur erreichen kann. Dabei werden auch die Positionen der anderen Figuren auf dem Brett berücksichtigt.
- **Schachfigur schlagen:** Beim Schlagen einer anderen Figur wird eine Figur bewegt, wobei dieser Anwendungsfall auf dem vorherigen aufbaut. Da das Schlagen einer Figur Auswirkungen auf die im Spiel befindlichen Figuren hat, wird dies als eigener Fall betrachtet.
- **Schach/Schachmatt geben:** Wenn eine Figur eines Spielers ein Feld abdeckt, auf dem sich der König des anderen Spielers befindet, wird Schach geboten. Ein Spieler kann den Gegenspieler durch einfaches Bewegen oder Schlagen einer anderen Figur in Schach setzen. Es ist die Pflicht des Spielers, mit seinem nächsten Zug dem Schach zu entkommen. Kann er dies nicht, liegt ein SSchachmatt"vor, und der Spieler, der Schach bietet, gewinnt. Da das Vorhandensein von Schach oder Schachmatt den Spielverlauf entscheidend beeinflusst, handelt es sich hier um einen eigenen Anwendungsfall.
- **Schachzug anzeigen:** Um einen Überblick über den Spielverlauf zu erhalten, werden alle während eines Spiels ausgeführten Züge dokumentiert und übersichtlich angezeigt.
- **Schachspiel exportieren:** Um die Historie eines bestimmten Schachspiels zu sichern, bietet die Anwendung die Möglichkeit, den Spielverlauf in eine Datei zu exportieren. Dabei wird auf die angezeigten Züge zurückgegriffen.

Neben den Services, die die Logik der einzelnen Domain-Bausteine umsetzen, gibt es im Application Code noch die besondere Klasse *Schachspielkontrollierer*. Diese Klasse enthält die gesamte Spiellogik, die den Ablauf des Schachspiels steuert.

3.4 Schicht 1 - Adapters

In einem Schichtenaufbau, der Clean Architecture berücksichtigt, dienen die Adapter dazu, die Informationen und Daten aus der Domänen- und Anwendungsschicht in ein Format umzuwandeln, mit dem die visuelle Darstellung der Anwendung arbeiten kann. Dies beinhaltet zum Beispiel die Konvertierung verschiedener Formate. Da die Entities und Value Objects in der Domänenschicht bereits alle *toString*-Methoden implementieren, aus denen die erforderlichen Informationen für die visuelle Darstellung abgeleitet werden können, ist in diesem Projekt keine separate Umsetzung der Adapter-Schicht erforderlich. Gemäß Clean Architecture müssten diese Datenbereitstellungen für die Nutzerinteraktion in die Adapter-Schicht ausgelagert werden. In der Praxis wären die entsprechenden Methoden jedoch in der Domänenschicht und den Adaptern sehr ähnlich, sodass ein Mapping für dieses Projekt nur zusätzliche Redundanz erzeugen würde.

3.5 Schicht 0 - Plugins

Die äußerste Schicht der Clean Architecture enthält alle extern eingebundenen Klassen, wie Frameworks und Programmschnittstellen nach außen, beispielsweise zur Persistierung von Programmdaten. Die Persistenz der Schachspiele ist nur in eine Richtung ausgelegt. In der Klasse *SpielzugRepositoryBruecke* wurde die Funktion implementiert, den Verlauf eines Schachspiels in eine einfache Textdatei zu schreiben. Die *RepositoryBruecke* ist eine Implementierung des *SpielzugRepository*-Interfaces, das in der Domänenschicht liegt.

Zusätzlich zur Datenpersistenz stellt die Plugin-Schicht auch die grafische Benutzeroberfläche bereit. Offiziell ist Processing eine eigenständige Programmiersprache, die in Java eingebettet werden kann. Da Processing jedoch mit Maven in das Projekt geladen werden muss, kann man es auch als „Bibliothek zum Zeichnen grafischer Komponenten“ bezeichnen. Die Anwendung von Processing wird ebenfalls in die Plugin-Schicht ausgelagert.

Die Hauptklasse der Benutzeroberfläche ist die Klasse *Benutzeroberflaeche*. Diese Klasse zeigt entweder einen Startbildschirm an oder lädt die Spielansicht. In der Spielansicht werden die verschiedenen Domänenbausteine mithilfe verschiedener Zeichner-Klassen in

der GUI dargestellt, die von der *SchachspielZeichner-Klasse* koordiniert werden. Hierzu gehören die Klassen *FeldZeichner*, *SchachbrettZeichner* und *FigurZeichner*.

3.6 Dependency Inversion

Hinweis

Commit:
https://github.com/clemens1403/AdvSWE/commit/ff87fffb24abc4050ca30019bf6800d0b60b7229

Vor der Einführung der Schichtenarchitektur bestand das Programm aus einer einzelnen Ordnerebene. Die Funktionen der verschiedenen Domainbausteine waren nicht in separate Schichten (Bausteine, Applikationslogik, Darstellung) aufgeteilt, sondern wurden in einer einzigen Klasse implementiert.

Processing bietet eingebaute Funktionen, die die Erstellung von Grafiken sehr einfach machen. Zum Beispiel gibt es die globalen Variablen *mouseX* und *mouseY*, die von Processing automatisch bereitgestellt werden. Dadurch kann die Mausposition auf der aktuellen Zeichenfläche jederzeit abgerufen werden. Diese Positionserfassung ist erforderlich, um Klicks auf bestimmten Elementen in der GUI zu erkennen. Durch die Einbindung von Processing in Java mit IntelliJ und Maven war es jedoch so, dass die von der Klasse *Benutzeroberflaeche* erzeugte Zeichenfläche nur im Kontext dieser Klasse bearbeitet werden konnte. Daher war es nicht möglich, aus anderen Klassen auf die Mausparameter zuzugreifen. Um dennoch Mausklicks, z.B. bei der Auswahl eines Feldes, auswerten zu können, mussten die Mauswerte als Funktionsparameter von der *Benutzeroberflaeche* an die Klassen der Domainbausteine übergeben werden.

Nachdem die Programmstruktur in Schichten aufgeteilt wurde, wurden die Repräsentation der Domainbausteine, die zugehörige Logik und die Darstellung mittels Processing voneinander entkoppelt. Durch die Auslagerung der Ermittlung der Zeichenparameter in die Plugin-Schicht wurde sichergestellt, dass die Prinzipien der Clean Architecture eingehalten wurden. Funktionen auf der Anwendungsebene, die ebenfalls die Mausparameter zur Positionsbestimmung benötigen, halten die Dependency Rule ein, indem Aufrufe nur aus der Plugin-Schicht und nicht aus der Domain-Schicht erfolgen.

Eine fachlich richtige Anwendung einer Dependency Inversion, wie sie in der Vorlesung besprochen wurde, ist die Eingrenzung von Processing auf die Zeichenebene nicht wirklich. In diesem Abschnitt wurde eher die Einhaltung der Dependency Rule beschrieben, doch erschien bei der Implementierung keine geeigneter Vorwand, um eine Dependency Inversion durchzuführen.

3.7 main-Methode

Die *main*-Methode einer Anwendung ist der Startpunkt für alle Aktionen, die ein Programm ausführen kann. Gemäß der Schichtenarchitektur befindet sich die *main*-Methode *ChessOfDuty* in der äußersten Schicht.

4 Entwurfsmuster

Für „Chess of Duty“ wurde ein Observer-Pattern eingebaut.

Hinweis
Commit:
https://github.com/clemens1403/AdvSWE/commit/c04b243f87959c6f80f7d75ccf31cba30af50bd5

Ein Observer, auch als Beobachter bezeichnet, ist ein Entwurfsmuster in der Softwareentwicklung, das eine lose Kopplung zwischen Objekten ermöglicht. Mithilfe von Beobachtern können 1:N-Beziehungen hergestellt werden, wobei Änderungen an einem Objekt automatisch an alle abhängigen Objekte weitergegeben werden. Das Muster basiert auf zwei Komponenten:

- Subject: Dieses Element im Code dient als Grundlage für die Beobachtung. Das Subjekt enthält eine Liste aller registrierten Beobachter und stellt Methoden für Registrierung, Entfernung und Benachrichtigung bereit. Sobald das Subjekt Änderungen erfährt, werden diese automatisch an alle registrierten Beobachter weitergegeben.
- Observer: Ein Beobachter implementiert die vom Subjekt definierte Schnittstelle, um Benachrichtigungen zu erhalten und auf Änderungen des Subjekts reagieren zu können.

Um das Schachspiel funktionsfähig zu machen, müssen alle grafischen Elemente regelmäßig gezeichnet werden. Dabei muss immer der aktuellste Zustand des Spiels dargestellt werden, der die Informationen aus der Anwendungsschicht entnimmt und über die Plugin-Schicht gezeichnet wird. Die Zeichner-Klassen müssen über Änderungen im aktuellen Spielstand informiert werden, sobald diese auftreten.

Im Folgenden werden der vorherige Stand und der nachherige Stand genauer beschrieben. Dabei werden UML-Diagramme für die visuelle Darstellung verwendet. Da die betrachteten Klassen im Kontext des gesamten Projekts ziemlich umfangreich sind und eine hohe Anzahl von Attributen und Funktionen aufweisen, wurde die Darstellung in den Diagrammen auf die wichtigsten Klassenbestandteile für die Implementierung des Observer-Musters reduziert.

4.1 Vor dem Entwurfsmuster

Auch ohne die Implementierung des beschriebenen Beobachter-Musters war es erforderlich, den aktuellen Spielzustand aus der Spiellogikklasse an die Zeichnerklassen zu übergeben, um die grafische Oberfläche korrekt darzustellen. In Processing gibt es eine draw()-Methode, die zur Zeichnung von Grafiken verwendet werden kann. In dieser Methode werden die verschiedenen Zeichnerklassen koordiniert, um sicherzustellen, dass alle grafischen Elemente mehrmals pro Sekunde gezeichnet werden.

Abbildung 4.1: Programmausschnitt ohne Observer-Pattern

Ursprünglich wurde bei jedem Aufruf der draw()-Methode in der Zeichenklasse die Funktion *setSchachspielKontrollierer* aufgerufen. Diese Funktion nimmt eine Instanz der Klasse *SchachspielKontrollierer* entgegen und setzt sie als globale Variable in der Zeichnerklasse. Da jede übergebene Kontrollierer-Instanz die Variable *Schachspiel* enthält, kann auch diese Variable lokal in der Zeichnerklasse gesetzt werden, um die Grafiken basierend darauf zu erstellen.

4.2 Mit dem Entwurfsmuster

Um das Beobachter-Muster einzubauen, werden zwei zusätzliche Klassen benötigt: ein Interface für den Beobachter und ein Interface für das Subjekt. Im Rahmen des Programmentwurfs wurde ein Beobachter-Muster für das Attribut „Schachspiel“ implementiert, um alle Änderungen zu registrieren.

Die Interfaceklasse für den Beobachter beschreibt die Methode „aktualisiere()“. Das Interface für das Subjekt enthält Methoden zum Registrieren, Entfernen und Benachrichtigen von Beobachtern. Diese Methoden müssen in den verschiedenen Klassen, in denen die Interfaces verwendet werden, überschrieben werden, um ihre Funktionalität genauer zu definieren.

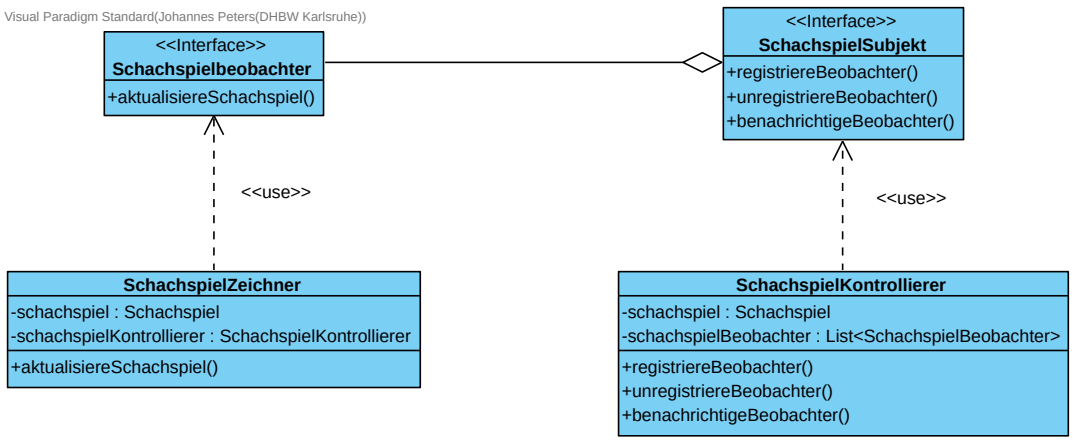


Abbildung 4.2: Programmausschnitt mit Observer-Pattern

In der eigenen Implementierung des Schachprogramms verwendet die Zeichnerklasse *SchachspielZeichner* das Beobachter-Interface. Die Klasse *SchachspielKontrollierer*, die die Spiellogik steuert, implementiert das Subjekt-Interface. Da beide Klassen jeweils ein Interface verwenden, müssen die in den Interfaces beschriebenen Methoden überschrieben und implementiert werden.

Im Kontrollierer wird eine Liste von Beobachtern erstellt. Die Methoden *registriereBeobachter()* und *unregistriereBeobachter()*, die bereits im Interface definiert sind, verwalten die Beobachter-Instanzen, indem sie neue Beobachter hinzufügen oder bestehende Be-

obachter entfernen. Die Methode *benachrichtigeBeobachter()* iteriert über die Liste der Beobachter und ruft für jeden Beobachter die Funktion *aktualisiereSchachspiel()* auf.

In der Zeichnerklasse wird das Beobachter-Interface verwendet, und die Methode *aktualisiereSchachspiel()* muss überschrieben werden. Diese neue Methode übernimmt die ursprüngliche Funktion der Methode *setSchachspielKontrollierer*. Im Detail bedeutet dies, dass die neu geschriebene Methode durch den Aufruf des Beobachters einen neuen Kontrollierer erhält, aus dem die neueste Version von *schachspiel* entnommen werden kann, sobald eine Änderung an diesem Attribut auftritt.

Wenn man die vorherigen und nachherigen Zustände betrachtet, könnte man denken, dass kein echter Mehrwert geschaffen wurde, sondern nur eine erhöhte Komplexität im Code entstanden ist. Doch dem ist nicht so. Anfangs wurde bei jedem Aufruf der draw()-Methode von Processing das Attribut *schachspiel* an die Zeichnerklasse übergeben. Das Beobachter-Muster verhindert nun, dass das Attribut unnötigerweise mehrmals pro Sekunde neu gesetzt wird. Mit dem eingebauten Beobachter für das Schachspiel-Attribut wird *schachspiel* nur dann neu gesetzt, wenn es explizit vom Subjekt übergeben wird. Eine solche Übergabe wird nur ausgelöst, wenn sich das Schachspiel-Attribut tatsächlich ändert.

5 Unit Tests

Dieses Kapitel befasst sich mit den Unit-Tests des Programmes. Dazu werden zuerst die ATRIP-Regeln erläutert und deren Anwendung beschrieben. Im Anschluss wird die Code Coverage erläutert. Zum Schluss wird noch auf die Verwendung von Mock-Objekten in den Unit-Tests eingegangen.

5.1 ATRIP-Regeln

Die ATRIP-Regeln sind Regeln, welche gute Unit-Tests einhalten sollten. Die Regeln setzen sich wiefolgt zusammen:

- Automatic - Eigenständig
- Thorough - Gründlich (genug)
- Repeatable - Wiederholbar
- Independent - Unabhängig voneinander
- Professional - Mit Sorgfalt hergestellt

5.1.1 Automatic

Die Regel „Automatic“ besagt, dass die Tests ohne manuelle Eingriffe selbstständig ablaufen müssen. Weiterhin müssen die Tests auch ihre Ergebnisse selbst überprüfen. Als Ergebnisse sind dabei nur „bestanden“ oder „nicht bestanden“ zulässig. Durch die Anwendung dieser Regel ist es möglich Unit Tests zu automatisieren.

```
1  @Before
2  public void setup() {
3      MockitoAnnotations.initMocks(this);
4      bauerDienst = new BauerDienst();
5      schachbrett = new Schachbrett();
6  }
7
8  @Test
9  public void testErzeugeBauer() {
10     int farbe = 1;
11     Feld startPosition = new Feld(4,1);
12     Bauer ergebnis = bauerDienst.erzeugeBauer(farbe, startPosition);
13
14     Assertions.assertNotNull(ergebnis);
15     Assertions.assertEquals(ergebnis.getFarbe(), farbe);
16     Assertions.assertEquals(ergebnis.getPosition(), startPosition);
17 }
```

Listing 5.1: Automatic Unit Test

In Quellcode 5.1 ist ein Unit Test aus der Klasse BauerDienstTest zu sehen. In den Zeilen 14-16 überprüft der Test mithilfe von Assertions selbst sein Ergebnis. Das eigenstädnige ablaufen des Tests wird über Maven gewährleistet. Somit ist der Test Automatic.

5.1.2 Thorough

Die Regel „Thorough“ besagt, dass alles Notwendige getestet werden muss. Die Definition von notwendig hängt dabei immer mit den Rahmenbedingungen zusammen. Mindestens muss aber jede missionskritische Funktionalität getestet werden und für jeden aufgetretenen Fehler muss ein Testfall existieren, welcher das erneute Auftreten des Fehlers verhindert. Durch diese Regel entstehen zusätzliche Tests im Umfeld von einem Fehler, um weitere Fehler zu verhindern.

Für den konkreten Fall des Schachspiels ist es notwendig jegliche Figuren-Dienste sowie den Kontrollierer für das Schachspiel zu testen. Daher sind für alle diese Klassen Unit Tests erstellt worden. Da es sich bei dem Programm um ein Spiel handelt, ist nahezu jede Funktion die Klassen missionskritisch. Da der Aufwand für jede Methode einen Unit Test zu programmieren zu hoch für dieses Projekt wäre, wurden zu jeder Klasse ein bis zwei Unit Tests implementiert.

5.1.3 Repeatable

Die Regel „Repeatable“ besagt, dass jeder Test automatisch durchführbar sein sollte und dabei stets das gleiche Ergebnis liefern sollte. Dazu muss der Test unabhängig von der Umgebung sein. Dabei sind vor allem der Umgang mit einem Datum oder mit Zufallszahlen problematisch. Auch der Zugriff auf das Dateisystem stellt eine Abhängigkeit von der Umgebung dar.

Bei den implementierten Unit Tests gibt es keine Abhängigkeiten von der Umgebung. Die standardmäßigen Problemquellen sind für das Testen der Anwendung irrelevant, da weder Daten oder Zufallszahlen für das Testen des Schachspiels relevant sind noch ein Zugriff auf das Dateisystem in den Tests erfolgt. Das Spiel ist in sich selbst abgeschlossen, weshalb die Umgebung automatisch immer gleich ist.

5.1.4 Independent

Die Regel „Independent“ besagt, dass Tests unabhängig von einander funktionieren müssen. Reihenfolge und Zusammenstellung müssen für das ausführen der Tests irrelevant sein. Im Idealfall testet jeder Test genau einen Aspekt der zu testenden Komponente.

Da jeder Test alle benötigten Abhängigkeiten für sich selbst erzeugt erzeugt (mit zum Beispiel Mocks) gibt es keine Abhängigkeiten zu anderen Tests.

5.1.5 Professional

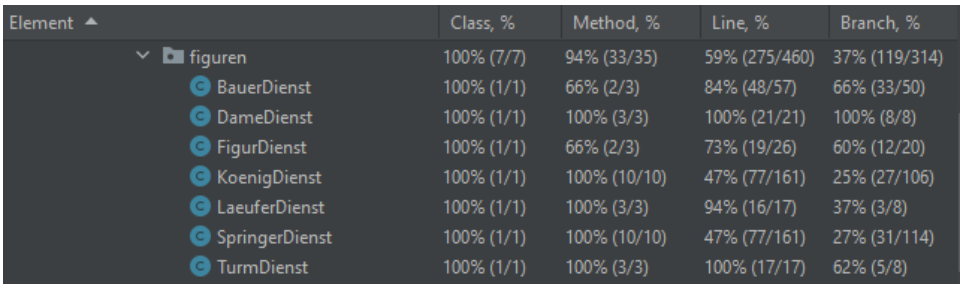
Die Regel „Professional“ besagt, dass Testcode zum relevanten Produktionscode gehört und so leicht verständlich wie mögich sein sollte.

Um dies umzusetzen wurden zum einen sinnvolle Namen für die Testmethoden vergeben. So sind die Namen der Tests stets so aufgebaut, dass sie aus Methodenname + „Test“ bestehen. Weiterhin sind die Tests an sich alle stets gleich aufgebaut. Zuerst werden die benötigten Abhängigkeiten erzeugt. Im Anschluss wird die zu testende Funktion ausgeführt und zum Schluss wird das Ergebnis überprüft. Durch diese Einheitliche Struktur wird für die Lesbarkeit der Tests gesorgt. Weiterhin sorgt auch eine simple Benennung von Variablen für gute Lesbarkeit.

5.2 Code Coverage

Die Code Coverage gibt an wie viel Quellcode mit Tests abgedeckt ist. Um die Code Coverage zu ermitteln kann eine Code Coverage Analyse in einer IDE ausgeführt werden. Bei Code Coverage unterschiedet man zwischen Line Coverage und Branch Coverage. Bei Line Coverage wird die Anzahl der getesten Zeilen des Quellcodes ins Verhältnis zur Gesamtzahl der Zeilen des Quellcodes gestellt. Branch Coverage hingegen beschreibt die Abdeckung von Verzweigungen im Code (if-Statements). Wenn ein Test nur einen von zwei Pfaden abdeckt so liegt die Branch Coverage bei nur 50 Prozent. Bei der Code Coverage ist es wichtig dass stets angegeben wird, ob Line Coverage oder Branch Coverage verwendet wurde, da sich das Ergebnis dieser beiden Verfahren stark unterscheiden kann.

In Abbildung 5.1 sind die Ergebnisse einer Code Coverage Analyse für die Figuren-Dienste zu sehen. Dabei werden alle Klassen abgedeckt. Auch die Abdeckung der Methoden ist mit 94 Prozent sehr gut. Bei der Line Coverage kommen die Tests auf 59 Prozent und bei Branch Coverage nur auf 37 Prozent.



Element	Class, %	Method, %	Line, %	Branch, %
figuren	100% (7/7)	94% (33/35)	59% (275/460)	37% (119/314)
BauerDienst	100% (1/1)	66% (2/3)	84% (48/57)	66% (33/50)
DameDienst	100% (1/1)	100% (3/3)	100% (21/21)	100% (8/8)
FigurDienst	100% (1/1)	66% (2/3)	73% (19/26)	60% (12/20)
KoenigDienst	100% (1/1)	100% (10/10)	47% (77/161)	25% (27/106)
LaeuferDienst	100% (1/1)	100% (3/3)	94% (16/17)	37% (3/8)
SpringerDienst	100% (1/1)	100% (10/10)	47% (77/161)	27% (31/114)
TurmDienst	100% (1/1)	100% (3/3)	100% (17/17)	62% (5/8)

Abbildung 5.1: Code Coverage-Analyse in IntelliJ für die Figuren-Dienste (nicht aktuell)

5.3 Mocks

Als Mock-Objekte werden Stellvertreter für echte Objekte bezeichnet. Mithilfe dieser Stellvertreter können Abhängigkeiten bei der Durchführung von Tests ersetzt werden. Durch das ersetzen der Abhängigkeiten einer Klasse mit Mocks wird das isolierte Testen dieser Klasse möglich. Da es sehr aufwendig ist Mocks selber zu programmieren werden häufig Mock-Tools verwendet. Für den Einsatz eines Mocks muss das Mock-Objekt zuvor

trainiert werden. Insgesamt durchläuft ein Mock-Objekt somit drei Phasen: Training-Phase, Einsatz-Phase und Verifikation-Phase.

Für das Schachspiel wurde Mockito als Mock-Tool verwendet. Ein Beispiel für die Verwendung von Mocks ist in der Testklasse KoenigDienstTest zu sehen (siehe Quellcode 5.2 oder Github).

```
1  @BeforeEach
2  public void setUp() {
3      koenigDienst = new KoenigDienst();
4      schachbrett = new Schachbrett();
5      koenigMock = mock(Koenig.class);
6      when(koenigMock.getPosition()).thenReturn(new Feld(1, 1)); // Beispielposition
7      when(koenigMock.getFarbe()).thenReturn(1);
8  }
9
10 @Test
11 public void getMoeglicheZuegeTest() {
12     Dame figur1 = mock(Dame.class);
13     Dame figur2 = mock(Dame.class);
14     when(figur1.getPosition()).thenReturn(new Feld(2, 2)); // Beispielposition
15     when(figur1.getFarbe()).thenReturn(1);
16     when(figur2.getPosition()).thenReturn(new Feld(1, 2)); // Beispielposition
17     when(figur2.getFarbe()).thenReturn(0);
18     ArrayList<Figur> figuren = new ArrayList<>(Arrays.asList(figur1, figur2));
19
20     ArrayList<Feld> moeglicheZuege = new ArrayList<>(Arrays.asList(new Feld(1, 2), new Feld(2, 1)));
21
22     assertEquals(moeglicheZuege, koenigDienst.getMoeglicheZuege(figuren, schachbrett, koenigMock));
23
24     verify(koenigMock, atLeast(1)).getPosition();
25     verify(koenigMock, atLeast(1)).getFarbe();
26
27     verify(figur1, atLeast(1)).getPosition();
28     verify(figur1, atLeast(1)).getFarbe();
29
30     verify(figur2, atLeast(1)).getPosition();
31     verify(figur2, atLeast(1)).getFarbe();
32 }
```

Listing 5.2: Verwendung von Mocks beim Testen

In Quellcode 5.2 sind zwei Methoden aus der Klasse KoenigDienstTest zu sehen. Die Methode setUp (Zeile 1-8) wird vor jeder Testfunktion ausgeführt und initialisiert alle für die Tests benötigten Objekte. Unter diesen Objekten befindet sich neben einem KoenigDienst-Objekt und und einem Schachbrett-Objekt auch ein Mock-Objekt der Klasse Koenig. Das Mock-Objekt wird in Zeile 5 von Quellcode 5.2 erzeugt und in den Zeilen 6 und 7 eingelernt. In der Test-Methode getMoeglicheZuegeTest (Zeile 10-25 in Quellcode 5.2) werden zunächst Testspezifische Mock-Objekte erzeugt. Da mit dieser Methode Interaktionen mit anderen Schachfiguren überprüft werden sollen, wird jeweils eine weiße und eine schwarze Dame als Mock-Objekt erzeugt (Zeile 12-13). In den Zeilen 14 bis 17 werden diese beiden Mock-Objekte eingelernt. Abgespielt werden die Mocks mit dem Funktionsaufruf *koenigDienst.getMoeglicheZuege(figuren, schachbrett, koenigMock)*

in Zeile 22. Im Anschluss wird überprüft, ob die angelernten Aufrufe mindestens ein mal genutzt wurden (Zeile 24 bis 31).

6 Refactoring

Dieses Kapitel befasst sich mit dem Refactoring der Anwendung. Dazu werden zuerst Code Smells identifiziert und erläutert warum diese so schlecht sind. Danach wird darauf eingegangen wie diese Code Smells durch Refactoring behoben werden können und wie die Probleme des Code Smells dadurch aufgehoben werden.

6.1 Code Smells

Bei Code Smells handelt es sich um eine verbesserungswürdige Quellcodestelle. Um festzustellen, ob eine Quellcodestelle verbesserungswürdig ist, gibt es keine festen Messwerte. Stattdessen werden solche Stellen hauptsächlich durch Erfahrung, aber auch teilweise durch Algorithmen oder Heuristiken lokalisiert. Code Smells können in verschiedene Kategorien gegliedert werden: Duplicated Code, Long Method, Large Class, Shotgun Surgery, Switch Statement und Code Comments. In den folgende Abschnitten werden Duplicated Code, Long Method und Switch Statement genauer an einem Beispiel aus dem Quellcode der ChessOfDuty-Anwendung erläutert.

6.1.1 Code Duplication

Eine Code Duplication beschreibt eine Stelle im Quellcode, an welcher doppelter Code vorhanden ist. Dies ist sehr problematisch, da im Falle einer Änderung alle Stellen des duplizierten Codes geändert werden müssen. Somit entsteht ein vielfacher Pflegeaufwand. Ein extremer Fall von Code Duplication ist in Quellcode 6.1 zu sehen.

```
1  public ArrayList<Feld> getMoeglicheZuege(ArrayList<Figur> figuren, Schachbrett schachbrett, Springer springer){
2
3      ArrayList<Feld> moeglicheZuege = new ArrayList<>();
4
5      int spalte = springer.getPosition().getSpalte();
6      int zeile = springer.getPosition().getZeile();
7
8      moeglicheZuege.addAll(zweiNachVorneEinsNachLinks(figuren, schachbrett, spalte, zeile, springer));
9      moeglicheZuege.addAll(zweiNachVorneEinsNachRechts(figuren, schachbrett, spalte, zeile, springer));
10     moeglicheZuege.addAll(zweiNachRechtsEinesNachOben(figuren, schachbrett, spalte, zeile, springer));
11     moeglicheZuege.addAll(zweiNachRechtsEinesNachUnten(figuren, schachbrett, spalte, zeile, springer));
12     moeglicheZuege.addAll(zweiNachUntenEinesNachRechts(figuren, schachbrett, spalte, zeile, springer));
```

```
13      moeglicheZuege.addAll(zweiNachUntenEinesNachLinks(figuren,schachbrett, spalte, zeile, springer));
14      moeglicheZuege.addAll(zweiNachLinksEinesNachUnten(figuren, schachbrett, spalte, zeile, springer));
15      moeglicheZuege.addAll(zweiNachLinksEinensNachOben(figuren,schachbrett, spalte, zeile, springer));
16
17      for (Feld eintrag : moeglicheZuege) {
18          System.out.println("(" + eintrag.getSpalte() + ", " + eintrag.getZeile() + ")");
19      }
20
21      return moeglicheZuege;
22  }
23
24  private ArrayList<Feld> zweiNachVorneEinsNachLinks(ArrayList<Figur> figuren, Schachbrett schachbrett, int spalte, int ↙
        ↳ zeile, Springer springer){
25
26      ArrayList<Feld> moeglicheZuege = new ArrayList<>();
27      boolean kollisionGefunden = false;
28      Figur kollidierteFigur = null;
29
30      if(zeile >= 7 || spalte == 1) return moeglicheZuege;
31
32      for(Figur f : figuren){
33          if(f.getPosition().getZeile() == (zeile+2) && f.getPosition().getSpalte() == spalte-1){
34              kollisionGefunden = true;
35              kollidierteFigur = f;
36              break;
37          }
38      }
39
40      if(kollisionGefunden){
41          System.out.println("Auf diesem Feld wurde eine Kollision festgestellt");
42
43          if(springer.getFarbe() == kollidierteFigur.getFarbe()){
44              System.out.println("Die erkannte Kollision ist mit einer Figur der gleichen Farbe");
45          } else{
46              System.out.println("Die erkannte Kollision ist mit einer Figur der anderen Farbe");
47              moeglicheZuege.add(schachbrett.getFeld(spalte-1, zeile+2));
48          }
49      } else {
50          moeglicheZuege.add(schachbrett.getFeld(spalte-1, zeile+2));
51      }
52
53      return moeglicheZuege;
54  }
55
56  private ArrayList<Feld> zweiNachVorneEinsNachRechts(ArrayList<Figur> figuren, Schachbrett schachbrett, int spalte, int ↙
        ↳ zeile, Springer springer){
57
58      ArrayList<Feld> moeglicheZuege = new ArrayList<>();
59      boolean kollisionGefunden = false;
60      Figur kollidierteFigur = null;
61
62      if(zeile >= 7 || spalte == 8) return moeglicheZuege;
63
64      for(Figur f : figuren){
65          if(f.getPosition().getZeile() == (zeile+2) && f.getPosition().getSpalte() == spalte+1){
66              kollisionGefunden = true;
67              kollidierteFigur = f;
68              break;
69          }
70      }
71
72      if(kollisionGefunden){
73          System.out.println("Auf diesem Feld wurde eine Kollision festgestellt");
74
75          if(springer.getFarbe() == kollidierteFigur.getFarbe()){
76              System.out.println("Die erkannte Kollision ist mit einer Figur der gleichen Farbe");
77          } else{
78              System.out.println("Die erkannte Kollision ist mit einer Figur der anderen Farbe");
79              moeglicheZuege.add(schachbrett.getFeld(spalte+1, zeile+2));
80          }
```

```
81      } else {
82          moeglicheZuege.add(schachbrett.getFeld(spalte+1, zeile+2));
83      }
84
85      return moeglicheZuege;
86  }
87  ...
```

Listing 6.1: Beispiel für Code Duplication

Der Quellcode 6.1 entstammt aus der SpringerDients-Klasse. Der Quellcode zeigt die Methode getMoeglicheZuege, welche die Züge berechnet, welche der entsprechende Springer machen kann. Bei der initialen implementierung wurde für jede grundsätzliche Schrittmöglichkeit eine eigene Methode implementiert, welche ausgibt bestimmt ob dieser Schritt möglich ist. Diese Methoden sind nahezu identisch. Zwei dieser Methoden sind mit zweiNachVorneEinsNachLinks und zweiNachVorneEinsNachRechts ebenfalls in Quellcode 6.1 zu sehen. Wenn man zum Beispiel das Bewegungsmuster des Springers ändern wöllte müsste man in jeder dieser 8 Methoden etwas ändern.

6.1.2 Long Method

Eine Long Method ist eine lange Methode. Lange Methoden sind schlecht, weil sie unübersichtlich sind und somit die Verständlichkeit des Quellcodes sinkt. Dadurch wird die Wartbarkeit des Codes schlechter und die Entwicklungsgeschwindigkeit sinkt. Ein Beispiel für eine Long Method ist in der Klasse BauerDienst mit der Methode getMoeglicheZuege zu finden (siehe Quellcode 6.2). Die Methode erstreckt sich über unglaubliche 87 Zeilen, wodurch es sehr schwierig ist nachzuvollziehen, was genau in der Methode passiert.

```
1   public ArrayList<Feld> getMoeglicheZuege(ArrayList<Figur> figuren, Schachbrett schachbrett, Bauer bauer){
2
3       ArrayList<Feld> moeglicheZuege = new ArrayList<>();
4
5       Feld aktuellePosition = bauer.getPosition();
6       System.out.println("Aktuelles Feld: " + aktuellePosition);
7       if (aktuellePosition != bauer.getStartposition()) {
8           bauer.setDoppelschrittMoeglich(false);
9       }
10      int aktuelleZeile = aktuellePosition.getZeile();
11      int aktuelleSpalte = aktuellePosition.getSpalte();
12
13
14      boolean einzelschritt = true;
15      boolean doppelschritt = bauer.getDoppelschrittMoeglich();
16
17      if (bauer.getFarbe() == 1) {
18          for (Figur f : figuren) {
19              Feld positionFigur = f.getPosition();
20              int zeileFigur = positionFigur.getZeile();
21              int spalteFigur = positionFigur.getSpalte();
```

```
22              if (aktuelleSpalte == spalteFigur) {
23                  if (aktuelleZeile == zeileFigur - 1) {
24                      einzelschritt = false;
25                      doppelschritt = false;
26                  } else if (doppelschritt) {
27                      if (aktuelleZeile == zeileFigur - 2) {
28                          doppelschritt = false;
29                      }
30                  }
31              }
32
33              if (aktuelleSpalte - 1 == spalteFigur || aktuelleSpalte + 1 == spalteFigur) {
34                  if (aktuelleZeile == zeileFigur - 1) {
35                      if (bauer.getFarbe() != f.getFarbe()) {
36                          moeglicheZuege.add(positionFigur);
37                      }
38                  }
39              }
40          }
41
42          if (einzelschritt) {
43              moeglicheZuege.add(schachbrett.getFeld(aktuelleSpalte, aktuelleZeile + 1));
44          }
45          if (doppelschritt) {
46              moeglicheZuege.add(schachbrett.getFeld(aktuelleSpalte, aktuelleZeile + 2));
47          }
48
49      } else {
50          for (Figur f : figuren) {
51              Feld positionFigur = f.getPosition();
52              int zeileFigur = positionFigur.getZeile();
53              int spalteFigur = positionFigur.getSpalte();
54              if (aktuelleSpalte == spalteFigur) {
55                  if (aktuelleZeile == zeileFigur + 1) {
56                      einzelschritt = false;
57                      doppelschritt = false;
58                  } else if (doppelschritt) {
59                      if (aktuelleZeile == zeileFigur + 2) {
60                          doppelschritt = false;
61                      }
62                  }
63              }
64
65              if (aktuelleSpalte - 1 == spalteFigur || aktuelleSpalte + 1 == spalteFigur) {
66                  if (aktuelleZeile == zeileFigur + 1) {
67                      if (bauer.getFarbe() != f.getFarbe()) {
68                          moeglicheZuege.add(positionFigur);
69                      }
70                  }
71              }
72          }
73
74          if (einzelschritt) {
75              moeglicheZuege.add(schachbrett.getFeld(aktuelleSpalte, aktuelleZeile - 1));
76          }
77          if (doppelschritt) {
78              moeglicheZuege.add(schachbrett.getFeld(aktuelleSpalte, aktuelleZeile - 2));
79          }
80      }
81
82      for (Feld eintrag : moeglicheZuege) {
83          System.out.println("(" + eintrag.getSpalte() + ", " + eintrag.getZeile() + ")");
84      }
85
86      return moeglicheZuege;
87  }
```

Listing 6.2: Beispiel für Long Method

6.1.3 Switch-Statement

Switch-Statements haben verschiedene Probleme. Das erste Problem ist, dass meist gleiche Switch-Statements an mehreren Stellen verwendet werden. Weiterhin können Switch-Statements nicht aufgeteilt werden, sondern nur wachsen, wodurch eine hohe Komplexität entsteht. Zuletzt ist auch die Syntax sehr Fehleranfällig durch die Verwendung von Breaks oder Fall-Throughs. Ein Beispiel für die hohe Komplexität von Switch-Statements ist in Quellcode 6.3 aus der Klasse ChessOfDuty zu sehen. Dargestellt ist die Methode mousePressed. In dieser Methode wird zuerst mit einem Switch-Statement überprüft in welcher Ansicht man sich befindet, bevor über alle Buttons der Ansicht iteriert wird und im Falle eines geklickten Buttons mit einem weiteren Switch-Statement eine auszuführende Funktion zugeordnet wird.

```
1  public void mousePressed() {
2      switch (benutzeroberflaeche.getStatus()) {
3          case "Start":
4              ArrayList<Knopf> knoepfeStart = benutzeroberflaeche.getStartKnoepfe();
5              for (Knopf k : knoepfeStart) {
6                  if (k.pruefeMausPosition()) {
7                      switch (k.getId()) {
8                          case "Spielen":
9                              this.schachspiel = schachspielDienst.erstelleSchachspiel();
10                             schachspielKontrollierer.erstelleNeuesSpiel(schachspiel);
11                             benutzeroberflaeche.setStatus("Spiel");
12                             break;
13                         default:
14                             break;
15                     }
16                 }
17             }
18             break;
19         case "Spiel":
20             schachspielKontrollierer.klickAuswerten(mouseX, mouseY);
21             ArrayList<Knopf> knoepfeSpiel = benutzeroberflaeche.getSpielKnoepfe();
22             for (Knopf k : knoepfeSpiel) {
23                 if (k.pruefeMausPosition()) {
24                     switch (k.getId()) {
25                         case "neustart":
26                             this.schachspiel = schachspielDienst.erstelleSchachspiel();
27                             schachspielKontrollierer.erstelleNeuesSpiel(schachspiel);
28                             break;
29                         case "Start":
30                             benutzeroberflaeche.setStatus("Start");
31                             break;
32                         case "export":
33                             spielzugRepositoryBruecke.dokumentiere(schachspiel);
34                             break;
35                         default:
36                             break;
37                     }
38                 }
39             }
40             break;
41         default:
42             break;
43     }
44 }
```

Listing 6.3: Beispiel für Switch-Statement

6.2 Refactoring der Code Smells

In den folgenden Abschnitten wird erläutert, wie die Code Smells aus dem vorherigen Abschnitt behoben werden können. Dabei wird zum einen auf die Code Duplication aus Quellcode 6.1 und zum anderen auf die Long Method aus Quellcode 6.2 eingegangen.

6.2.1 Code Duplication

Um eine Code Duplication zu beheben kann die Extract Method-Technik angewendet werden. Dabei werden zusammenhängende Quellcodefragmente in eigene Methoden ausgelagert Dadurch wird der Code feingranularer und es bilden sich Abstraktionsstufen aus. Bei extrahierten Methoden sollte nach Möglichkeit auf Eingabe- und Ausgabe-Parameter verzichtet werden.

Im Quellcode 6.1 ist es jedoch nicht sinnvoll eine Methode zu extrahieren, da die Methoden, in welchen die Code Duplication vorliegt, nahezu identisch sind. Daher werden die Methoden zu einer Methode, die alle 8 Methoden ersetzt, zusammengefasst. Dazu wird die Bewegung als zusätzliches Parameter übergeben und anstatt der fest definierten Integerwerte verwendet (siehe Quellcode 6.4).

```
1  public ArrayList<Feld> getMoeglicheZuege(ArrayList<Figur> figuren, Schachbrett schachbrett, Springer springer){
2
3      ArrayList<Feld> moeglicheZuege = new ArrayList<>();
4
5      int spalte = springer.getPosition().getSpalte();
6      int zeile = springer.getPosition().getZeile();
7
8      moeglicheZuege.addAll(checkZugMoeglich(2,-1, figuren, schachbrett, spalte, zeile, springer));
9      moeglicheZuege.addAll(checkZugMoeglich(2,1,figuren, schachbrett, spalte, zeile, springer));
10     moeglicheZuege.addAll(checkZugMoeglich(1, 2,figuren, schachbrett, spalte, zeile, springer));
11     moeglicheZuege.addAll(checkZugMoeglich(-1,2, figuren, schachbrett, spalte, zeile, springer));
12     moeglicheZuege.addAll(checkZugMoeglich(-2,1, figuren, schachbrett, spalte, zeile, springer));
13     moeglicheZuege.addAll(checkZugMoeglich(-2,-1, figuren,schachbrett, spalte, zeile, springer));
14     moeglicheZuege.addAll(checkZugMoeglich(-1, -2, figuren, schachbrett, spalte, zeile, springer));
15     moeglicheZuege.addAll(checkZugMoeglich(1, -2, figuren,schachbrett, spalte, zeile, springer));
16
17     for (Feld eintrag : moeglicheZuege) {
18         System.out.println("(" + eintrag.getSpalte() + ", " + eintrag.getZeile() + ")");
19     }
20
21     return moeglicheZuege;
```

```
22  }
23  private ArrayList<Feld> checkZugMoeglich(int zeilenBewegung, int spaltenBewegung, ArrayList<Figur> figuren, ↙
        ↳ Schachbrett schachbrett, int spalte, int zeile, Springer springer){
24      ArrayList<Feld> moeglicheZuege = new ArrayList<>();
25      boolean kollisionGefunden = false;
26      Figur kollidierteFigur = null;
27
28      if(zeile+ zeilenBewegung > 8 || zeile + zeilenBewegung < 1 || spalte +spaltenBewegung > 8 || spalte + ↙
            ↳ spaltenBewegung < 1){
29          return moeglicheZuege;
30      }
31
32      for(Figur f : figuren){
33          if(f.getPosition().getZeile() == (zeile + zeilenBewegung) && f.getPosition().getSpalte() == spalte + ↙
                ↳ spaltenBewegung){
34              kollisionGefunden = true;
35              kollidierteFigur = f;
36              break;
37          }
38      }
39      if(kollisionGefunden){
40          System.out.println("Auf diesem Feld wurde eine Kollision festgestellt");
41
42          if(springer.getFarbe() == kollidierteFigur.getFarbe()){
43              System.out.println("Die erkannte Kollision ist mit einer Figur der gleichen Farbe");
44          } else{
45              System.out.println("Die erkannte Kollision ist mit einer Figur der anderen Farbe");
46              moeglicheZuege.add(schachbrett.getFeld(spalte + spaltenBewegung, zeile + zeilenBewegung));
47          }
48      } else {
49          moeglicheZuege.add(schachbrett.getFeld(spalte + spaltenBewegung, zeile + zeilenBewegung));
50      }
51
52      return moeglicheZuege;
53  }
```

Listing 6.4: Code Duplication behoben

6.2.2 Long Method

Um eine Long Method zu beheben kann ebenfalls eine Extract Method angewendet werden. Alternativ können aber auch die Berechnungen von lokalen Variablen ausgelagert werden. Durch die Anwendung dieser Techniken konnte der Quellcode aus 6.2 von der Long Method befreit werden. Der neue Quellcode ist in Abbildung 6.5 zu sehen.

```
1   public ArrayList<Feld> getMoeglicheZuege(ArrayList<Figur> figuren, Schachbrett schachbrett, Bauer bauer) {
2
3       ArrayList<Feld> moeglicheZuege = new ArrayList<>();
4
5       Feld aktuellePosition = bauer.getPosition();
6       System.out.println("Aktuelles Feld: " + aktuellePosition);
7       if (aktuellePosition != bauer.getStartposition()) {
8           bauer.setDoppelschrittMoeglich(false);
9       }
10      int aktuelleZeile = aktuellePosition.getZeile();
11      int aktuelleSpalte = aktuellePosition.getSpalte();
12      int richtung = bauer.getFarbe()==0?-1:1;
13
14      boolean einzelschritt;
15      boolean doppelschritt = bauer.getDoppelschrittMoeglich();
```

```
16
17      if (berechneSchrittMoeglich(bauer, figuren, 1)) {
18          einzelschritt = true;
19          if (berechneSchrittMoeglich(bauer, figuren, 2) && doppelschritt) {
20              doppelschritt = true;
21          } else {
22              doppelschritt = false;
23          }
24      } else {
25          einzelschritt = false;
26      }
27
28      for (Figur f : figuren) {
29          if (istFigurSchlagbar(bauer, f, richtung)) moeglicheZuege.add(f.getPosition());
30      }
31
32      if (einzelschritt) {
33          moeglicheZuege.add(schachbrett.getFeld(aktuelleSpalte, aktuelleZeile + 1 * richtung));
34      }
35      if (doppelschritt) {
36          moeglicheZuege.add(schachbrett.getFeld(aktuelleSpalte, aktuelleZeile + 2 * richtung));
37      }
38
39      for (Feld eintrag : moeglicheZuege) {
40          System.out.println("(" + eintrag.getSpalte() + ", " + eintrag.getZeile() + ")");
41      }
42
43      return moeglicheZuege;
44  }
```

Listing 6.5: Long Method behoben

Um die Methode getMoeglicheZuege zu verkürzen, wurde die Berechnung von Einzel- bzw. Doppelschritten des Bauern ausgelagert. Somit wird nur noch eine Methode aufgerufen, um zu bestimmen, ob der gewünschte Schritt möglich ist. Weiterhin wurde die Berechnung, ob der Bauer eine andere Figur schlagen kann, in eine neue Methode ausgelagert. Die ausgelagerten Methoden sind in Quellcode 6.6 zu sehen.

```
1   private Boolean istFigurSchlagbar(Bauer bauer, Figur figur, int richtung) {
2       int bauerSpalte = bauer.getPosition().getSpalte();
3       int bauerZeile = bauer.getPosition().getZeile();
4
5       int figurSpalte = figur.getPosition().getSpalte();
6       int figurZeile = figur.getPosition().getZeile();
7
8       if (bauerSpalte - 1 == figurSpalte || bauerSpalte + 1 == figurSpalte) {
9           if (bauerZeile == figurZeile - 1 * richtung) {
10              if (bauer.getFarbe() != figur.getFarbe()) {
11                  return true;
12              }
13          }
14      }
15      return false;
16  }
17
18  public Boolean berechneSchrittMoeglich(Bauer bauer, ArrayList<Figur> figuren, int schrittWeite) {
19      int bauerSpalte = bauer.getPosition().getSpalte();
20      int bauerZeile = bauer.getPosition().getZeile();
21      int schritt = bauer.getFarbe()==1? schrittWeite *-1 : schrittWeite;
22      for (Figur f : figuren) {
23          Feld positionFigur = f.getPosition();
24          int zeileFigur = positionFigur.getZeile();
25          int spalteFigur = positionFigur.getSpalte();
```

```
26          if (bauerSpalte == spalteFigur) {
27              if (bauerZeile == zeileFigur + schritt) {
28                  return false;
29              }
30          }
31      }
32      return true;
33  }
```

Listing 6.6: Long Method augelagerte Methoden

Mit der Methode berechneSchrittMoeglich wurde weiterhin eine Code Duplication behoben. Zuvor gab es bei der Berechnung der Möglichen Schritte eine Code Duplication (siehe Quellcode 6.2 Zeile 18-47 und Zeile 50-79).

Durch das Refactoring konnte die Long-Method um 43 Zeilen (nahezu 50 Prozent) verkürzt werden und der Code-Abschnitt im Allgemeinen konnte ebenfalls um 9 Zeilen durch die Entfernung der Code Duplication verkürzt werden. Somit konnte der Quellcode verständlicher und wartbarer gemacht.

7 Programming Principles

Dieses Kapitel befasst sich mit der Anwendung der in der Vorlesung behandelten Programming Principles und beschreibt die Anwendung dieser im Quellcode.

7.1 SOLID

Die SOLID-Regeln wurden Anfang der Nullerjahre von Michael Feathers und Robert C. Martin gesammelt und formuliert. Die Regeln haben die Ziele Software wartbar, Systeme erweiterbar und Codebasen langlebiger zu machen.

7.1.1 Single responsibility principle

Das Single responsibility principle ist das Prinzip der einzigen Zuständigkeit. Somit sollte eine Klasse nur einen einzigen Grund haben sich zu ändern. Die Zuständigkeiten einer Klasse können mithilfe von Change dimensions dargestellt werden. Dabei spannt jede Zuständigkeit eine zusätzliche Achse auf. Entlang dieser Achsen werden Änderungen am Code dargestellt. Im Idealfall sind die Achsen daher orthogonal, da sich die Änderungen dadurch nicht gegenseitig beeinflussen.

Dieses Prinzip wird im gesamten Quellcode angewendet. Die Klassen auf Domain-Ebene haben jeweils nur den Zweck das Entsprechende Objekt aus der Domain abzubilden. In der Application-Schicht hat jede Klasse Bezug zu einer Klasse aus der Domain-Ebene und nur die Aufgabe die entsprechenden Objekte zu Verwalten. Die einzige Ausnahme dabei ist der SchachspielKontrollierer, welcher die Koordination der anderen Klassen in der Application-Ebene übernimmt, um die Figuren auf dem Spielfeld miteinander interagieren zu lassen.

7.1.2 Open/Closed principle

Das Open/Closed principle besagt, dass Software-Entitäten offen für Erweiterungen sein sollen, aber gleichzeit geschlossen bezüglich Veränderungen sein sollen. Erweiterungen können zum Beispiel durch Unterklassen erschaffen werden, da nur die Unterklasse ihr Verhalten ändert, aber nicht die bereits existierende Klasse. Eine Veränderung stellt in diesem Kontext eine Modifikation des Codes durch geänderte Anforderungen dar. Zusammengefasst sollte bestehender Code also nicht mehr geändert werden müssen.

Als Beispiel für das Open/Closed principle im Code können die Figuren verwendet werden. Die Figuren sind so implementiert, dass beliebig neue Figuren hinzugefügt werden können oder Figuren ersetzt werden können.

7.1.3 Liskov substitution principle

Das LSP besagt, dass Objekte durch Instanzen ihrer Subtypen ersetzbar sein sollten, ohne die Korrektheit des Programms zu ändern. Somit gibt es strikte Regeln für Vererbungshierarchien. Subtypen dürfen dabei die Funktion der Oberklasse nicht einschränken sondern nur erweitern.

Als Beispiel für LSP können erneut die Figuren genommen werden. Zwischen der Figur-Klasse und den einzelnen Unterklassen tritt eine Kovarianz auf.

7.1.4 Interface segregation principle

Mithilfe von ISP sollen Schnittstellen mindestens in Nutzergruppen aufgeteilt werden. Dies wird umgesetzt indem anstelle eines großen Interfaces mehrere kleine Interfaces erstellt werden. Dies führt zu einer hohen Kohäsion und unterstützt auch das SRP.

ISP wurde im Quellcode nicht angewendet, da keine Interfaces für das Spiel implementiert wurden. Die einzige sinnvolle Stelle für die verwendung eines Interfaces wäre bei den Diensten für die Figuren, da diese alle eine Methode getMoeglicheZuege implementieren. Allerdings würde dieses Interface nur eine einzelne Methode umfassen, weshalb die Anwendung von ISP hier nicht sinnvoll ist.

7.1.5 Dependency inversion principle

Das DIP besagt, dass Klassen höherer Ebenen nicht von Klassen niederer Ebe abhängig sein sollen, sondern beide im Idealfall von einem Interface abhängig sind. Dies verhindert, dass Änderungen aus einem niedrigerem Modul zu Änderungen in höheren Modulen führen. Umgesetzt wird das indem das hohe Modul eine Schnittstelle definiert, welche vom niedrigen Modul implementiert wird. Die Referenz auf die Konkrete Instanz wird dem höheren Modul dann per Dependency Injection übergeben.

DIP wurde im Quellcode nicht angewendet. Die einzige Möglichkeit DIP anzuwenden besteht erneut bei den Figur-Diensten, welche für dieses Beispiel die niedrigeren Module darstellen. Das höhere Modul wäre der SchachspielKontrollierer. Der SchachspielKontrollierer könnte die verschiedenen Figur-Dienste als Interface definieren und die Figur-Dienste würden dann ein Interface mit der Methode getMoeglicheZuege definieren. Die Abhängigkeiten würden wie bisher per Dependency Injection übergeben werden. Jedoch wurde sich dazu entschieden dies nicht umzusetzen, da die Verständlichkeit des Codes dadurch sinkt (unserem empfinden nach). Durch die klare Typisierung der verschiedenen Figur-Dienste ist der Code strukturierter und man kann zusätzlich zum Name der Variable auch Anhand des Typs erkennen, welche Aufgabe der entsprechende Dienst hat.

7.2 GRASP

GRASP steht für General Responsibility Assignment Software Patterns/Principles und stellt Standardlösungen für Typische Fragestellungen bei der Softwareentwicklung dar. Insgesamt stellt GRASP neun Prinzipien/Werkzeuge bereit:

- Low Coupling
- High Cohesion
- Indirection
- Polymorphism
- Pure Fabrication
- Protected Variations

- Controller
- Information Expert
- Creator

In den folgenden Abschnitten wird kurz das Konzept der Kopplung und das Konzept der Kohäsion erläutert und die Anwendung im Quellcode beschrieben.

7.2.1 Kopplung

Als Kopplung wird das Maß für die Abhängigkeit einer Klasse von ihrer Umgebung bezeichnet. Durch geringe Kopplung werden viele Vorteile ermöglicht. So lässt sich Code mit geringer Kopplung leicht anpassen und gut Testen. Weiterhin ist der Code leichter verständlich und kann besser wiederverwendet werden.

Der Quellcode ist eher ein negativ-Beispiel für geringe Kopplung. Dies kann daran erkannt werden, dass zum größten Teil statische Methodenaufrufe ausgeführt werden, keine Interfaces verwendet werden bis auf den Beobachter, und auch keine Events auf einem Eventbus versendet werden. Als größtes Negativ-Beispiel im Quellcode kann man den Schachspielkontrollierer nehmen, welcher an eine Vielzahl von anderen Klassen und Methoden gekoppelt ist.

7.2.2 Kohäsion

Bei Kohäsion handelt es sich um das Maß für den inneren Zusammenhalt einer Klasse. Dabei ist Kohäsion ein semantisches Maß - also abhängig von der menschlichen Einschätzung.

Die Kohäsion des Quellcodes ist relativ hoch. In den Klassen der Figur-Dienste rufen sich die Methoden zum größten Teil untereinander auf. Die Klassen der Figuren-Dienste können allgemein als positives Beispiel für eine hohe Kohäsion genommen werden.

7.3 DRY

DRY steht für Don't Repeat Yourself und besagt, dass man alles einmal machen soll und nur einmal machen soll. Das Motto von Dry kann wiefolgt definiert werden: „ Jeder Wissensaspekt darf nur eine einzige, nicht zweideutige verbindliche Repräsentation in einem System besitzen". Mechanische Duplikation ist dabei jedoch erlaubt, solange die Originalquelle klar definiert ist. Somit ändert eine Modifikation alle verknüpften Elemente in gleicher Weise, aber ändert keine nicht verknüpften Elemente.

Im Quellcode ist dieses Prinzip zum Beispiel in der Klasse SpringerDienst zu erkennen. Die Klasse definiert wiederverwendbare und universell anwendbare Methoden ohne sich zu wiederholen.