Programmentwurf
ASE Spieleplattform

Name: Czerkas, Tim
Martrikelnummer: 6835894

Abgabedatum: 14.04.2023

Kapitel 1: Einführung

Übersicht über die Applikation

Die Anwendung stellt eine kleine Spieleplattform dar. Innerhalb dieser kann man die Spiele „Galgenmännchen“, „Zahlenraten“ und „Schere, Stein, Papier“ spielen. Als langfristigen Anreiz, weiterzuspielen, kann man die Statistiken nehmen, die abgespeichert werden können. Man kann sich dadurch mit sich selbst messen, aber natürlich auch mit anderen Spieler*innen. Der User oder die Userin muss dafür selbstständig speichern, dann werden die aktuellen Stats in einer Textdatei abgespeichert. Steuern lässt sich die Anwendung über Befehle in der Konsole, „LOG“ oder „REG“ zum einloggen bzw. registrieren, „GGM“ beispielsweise für ein Spiel Galgenmännchen. Alle weiteren Befehle werden einem beim Starten der Plattform oder des entsprechenden Spiels aufgezeigt.

Durch dieses Programm hat man immer gleich mehrere Spiele an einem Ort. Außerdem werden die Erfolge, die man erzielt auch abgespeichert und sind nicht direkt nach Verlassen der Anwendung wieder „verloren“, wie es oft bei sehr kleinen Plattformen der Fall ist. Damit kann noch mehr Spielspaß erzielt werden und auch längerfristig bleibt dieser erhalten. Vor allem mit dem Vergleich zu anderen Spieler*innen durch die Bestenliste bleibt der Reiz beim Spielen erhalten.

Wie startet man die Applikation?

1. Das Projekt von GitHub klonen: https://github.com/TimCzks/ASE_Spieleplattform
2. Die Klasse „MainMenu“ aufrufen.
3. Die Main-Methode aufrufen und dadurch die Anwendung starten. Weitere Erklärungen zu den einzelnen Befehlen befinden sich dann in der Konsole der IDE.

Wie testet man die Applikation?

Zum Testen der Applikation kann man die Unit-Tests ausführen, die sich in den Test-Packages befinden. Diese sind mit „.test“ gekennzeichnet, z.B. „spiele.test“. Wie man die einzelnen Tests ausführt, variiert je nach Entwicklungsumgebung, bei Eclipse geht es über den Shortcut „F11“.

Möchte man alle Tests auf einmal ausführen, geht das über „Coverage As“ → „Coverage Configurations“. Dann muss die Option „Run all Tests in selected project“ ausgewählt werden, bevor man die Coverage ausführt, wodurch auch die Tests ausgeführt werden.

Kapitel 2: Clean Architecture

Was ist Clean Architecture?

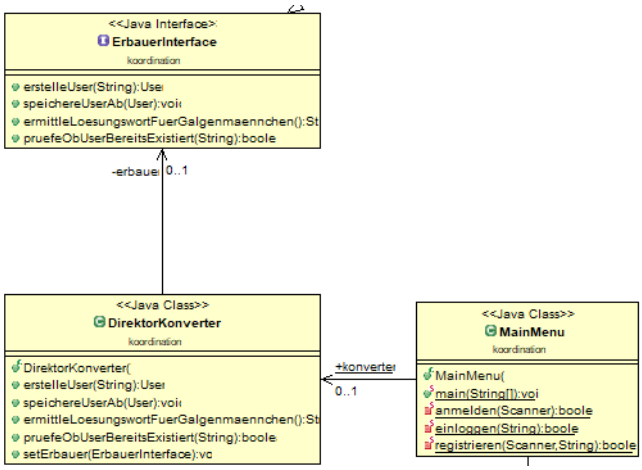
Unter Clean Architecture versteht man grundsätzlich eine nachhaltige Software-Architektur. Ermöglicht wird das durch ein Schichtenmodell, das die Clean Architecture mit sich bringt. Aufgrund dessen wird es auch gerne „Onion Architecture“ genannt. Die Schichten unterscheiden sich im Groben in zwei Kategorien: langlebig und kurzlebig. Der Kern wird dabie von langlebigem Code abgebildet, der dementsprechend auch Technologien enthalten, die zukunftsorientiert sind oder schlichtweg abwärtskompatibel. So kann heutzutage beispielsweise immer noch Java 8 verwendet werden, auch wenn das eigentliche Projekt (der Kern) vor vielen Jahren geschrieben wurde.

Bei genauerer Betrachtung unterscheiden die Schichten sich von innen nach außen in Abstraction Code, Domain Code, Application Code, Adapters und Plugins. Dabei ist wichtig zu beachten, dass die Abhängigkeiten von außen nach innen gehen. Umgesetzt wird das dadurch, dass die inneren Schichten Interfaces festlegen, welche dann in den äußeren Schichten genutzt werden können.

Analyse der Dependency Rule

Positiv-Beispiel: Dependency Rule

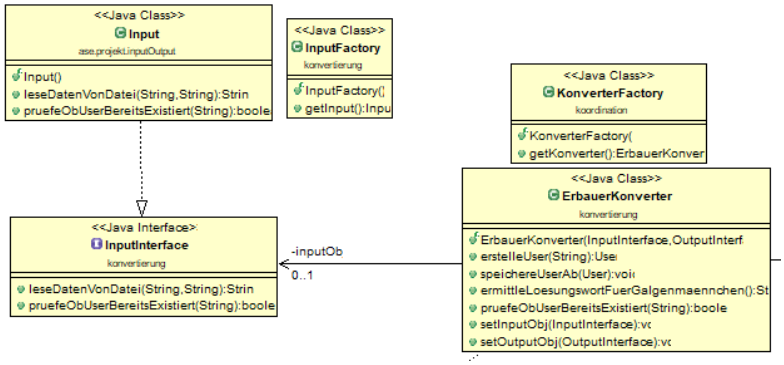
Die Klasse DirektorKonverter hält die Dependency Rule ein.



Die Klasse MainMenu hängt von der Klasse ab, jedoch sind diese beiden Klassen in der selben Schicht. Anders ist das beim ErbauerKonverter, der sich in einer Schicht weiter außen befindet. Darum wird das ErbauerInterface noch in der Schicht des DirektorKonverter definiert, damit es keine Abhängigkeit zu einer Implementierung einer äußeren Schicht gibt. Weitere Verbindungen in Bezug zum DirektorKonverter aus einer äußereren Schicht gibt es nicht.

Zweites Positiv-Beispiel: Dependency Rule

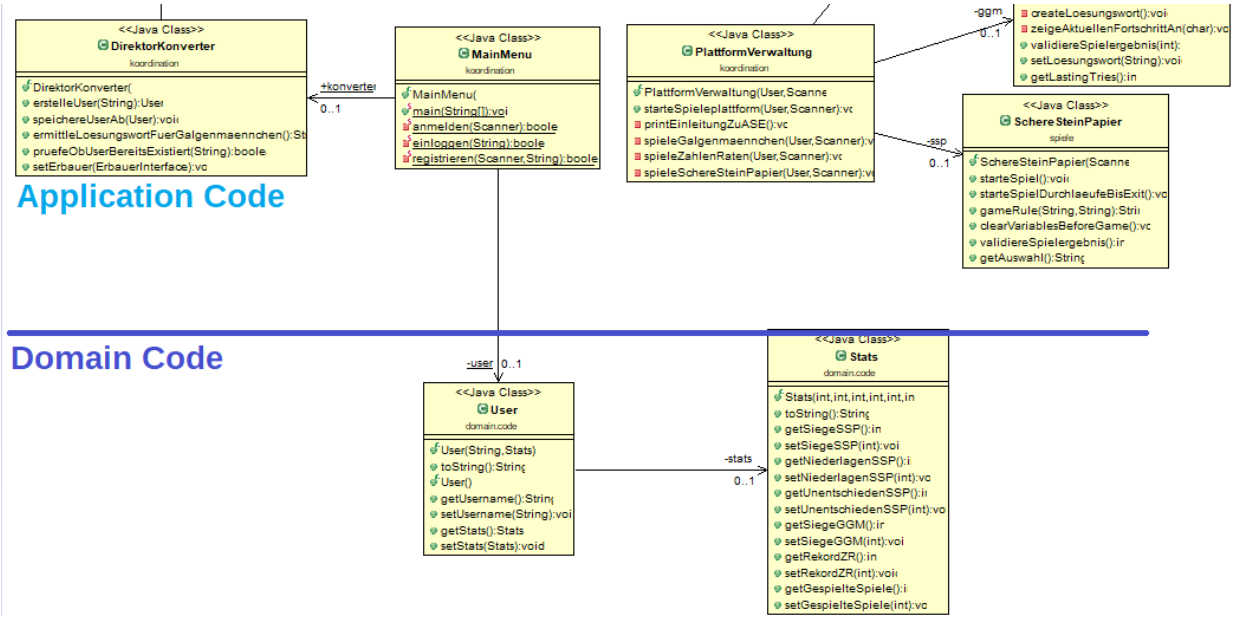
Die Klasse ErbauerKonverter erhält ebenfalls die Dependency Rule, indem von hier aus auf Interfaces referenziert wird, die sich in der selben Schicht befinden, aber in einer äußeren Schicht implementiert werden.



Analyse der Schichten

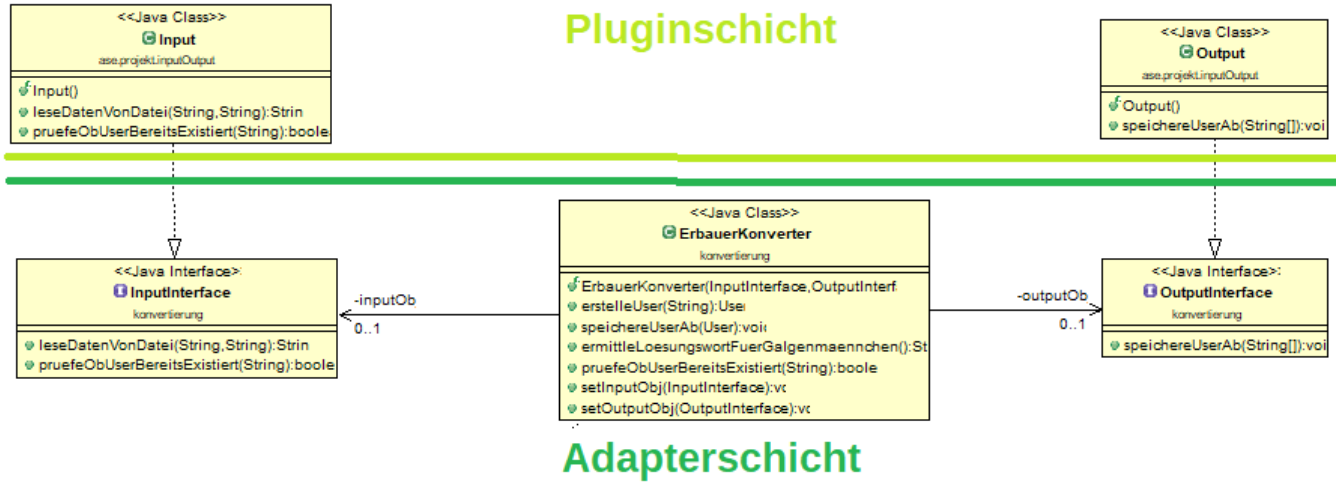
Schicht: Domain Code Layer

Die Domaincode-Schicht umfasst bei diesem Projekt die Entitäten des Users und der Statistik eines Users. Das **User-Objekt** ist dafür da, um die erspielte Statistik einem*r angemeldeten Nutzer*in zuordnen zu können. Die einzelnen Nutzer*innen werden ihrem Account über den Benutzernamen zugeordnet, der für alle User*innen einzigartig ist. Diese Objektklasse muss in nahezu keinem Fall geändert werden, egal wie sehr sich der Rest der Anwendung über die Zeit verändert oder erweitert. Da der User eine essentielle Entität ist und nicht mehr verändert werden muss, ist die Domaincode-Schicht hierfür gut geeignet.



Schicht: Plugin Layer

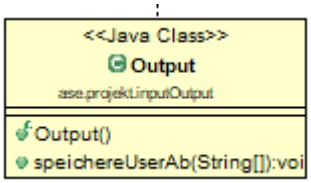
In der Pluginschicht befinden sich das Input- und Output-Objekt. Die Aufgabe von der **Inputklasse** ist es, Daten vom System einzulesen und für die weitere Verarbeitung bereitzustellen. Beispiele dafür sind das Einlesen der Nutzerdaten aus einer Textdatei über den Usernamen und das Überprüfen, ob ein*e Nutzer*in bereits existiert, indem überprüft wird, ob eine Datei mit dem Namen des*r Nutzer*in bereits vorhanden ist. Durch die einfache Aufgabe der Kommunikation zum Datenspeicherort (oft eine Datenbank, hier lediglich ein Ordner mit allen Dateien), die dann unverarbeitet für die Anwendung bereitgestellt werden, ist die Pluginschicht hierfür gut geeignet. Das Zusammenspiel mit der Adapterschicht ist in der unteren Abbildung zu sehen.



Kapitel 3: SOLID

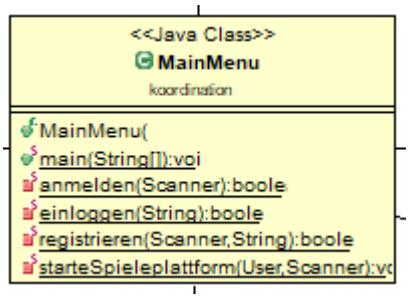
Analyse Single-Responsibility-Principle (SRP)

Positiv-Beispiel



Bei dieser Klasse findet das SRP Anwendung. Die einzige Aufgabe der Klasse, sowie seiner Methode, ist es, einen User abzuspeichern. Dabei bekommt es als Input einen String-Array, der bereits alle wichtigen Userdaten zusammengesammelt enthält. Dadurch muss wirklich nur noch der Schritt des Abspeicherns in einer Datei vorgenommen werden.

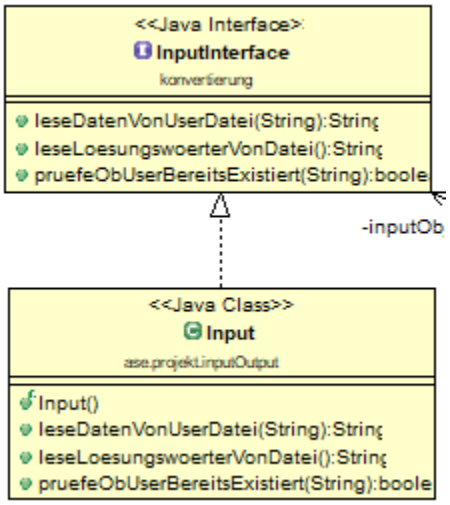
Negativ-Beispiel



Die MainMenu-Klasse erfüllt das SRP nicht. Neben den logischen Methoden rund um die Anmeldung wird hier zusätzlich auch die gesamte Plattform gestartet. Durch diese Methode wiederum können jegliche Spiele gestartet werden, d.h. auch hier hat die Methode mehr als eine Aufgabe. Eine Lösung kann sein, das Starten der Plattform in eine seperate Klasse auszulagern und die Methode ebenfalls in mehrere Methoden aufzuteilen. Außerdem kann auch die Anmeldung in eine eigene Klasse gezogen werden, damit die einzige Aufgabe des MainMenus das Starten der Anwendung über die main-Methode ist.

Analyse Open-Closed-Principle (OCP)

Positiv-Beispiel

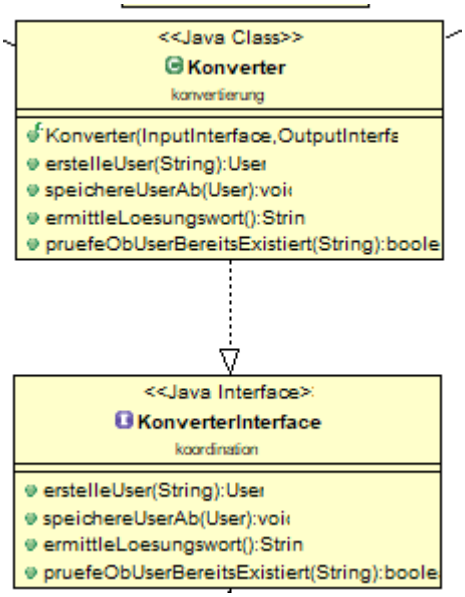


Diese Klasse, sowie deren Methoden, sind offen für Erweiterungen, aber geschlossen für Veränderungen und erfüllen somit das OCP. Jede der Methoden ist unabhängig von anderen Veränderungen im Projekt und müssen nicht angepasst werden. Der folgende Codeausschnitt enthält den Kern der Methode „leseDatenVonUserDatei“. Auch wenn die eingelesene Datei sich verändert und beispielsweise zusätzliche Daten des Users abgespeichert sind, werden diese hier weiterhin gleich gelesen und als Array zurückgegeben, sodass diese weiter verwendet werden können. So ist man an dieser Stelle unabhängiger von Änderungen anderere Komponenten.

```
BufferedReader br = new BufferedReader(new FileReader(f));
while (br.ready()) {
    s = br.readLine().split(",");
}
br.close();
```

Abbildung 1: Kern der Methode "leseDatenVonUserDatei"

Negativ-Beispiel

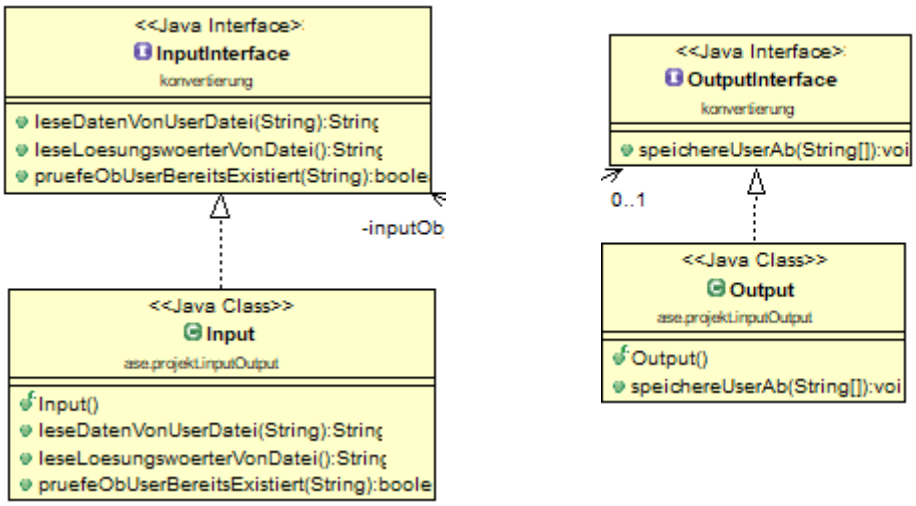


Die Konverter-Klasse implementiert zwar das entsprechende Interface und die Methoden haben jeweils nur eine Aufgabe. Dennoch könnte die Klasse verändert werden müssen, wenn sich beim Abspeichern des Users die abzuspeichernden Statistiken verändern. Diese müssten hier explizit in der Methode zusätzlich angegeben werden.

Man kann dieses Problem eventuell dadurch lösen, dass nur eine Methode aufgerufen wird, die beispielsweise vom User-Objekt stammt. Durch diese Methode sollten dann alle nötigen Stats gesammelt werden und das entsprechende Array zurückgegeben werden.

Interface-Segreggation- (ISP)

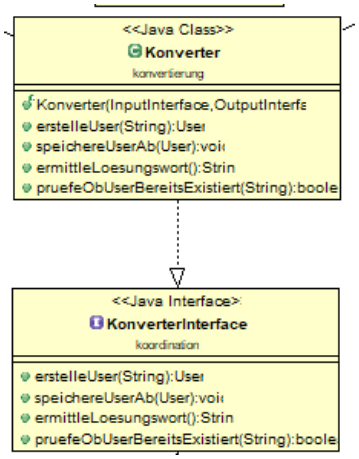
Positiv-Beispiel



Beide Klassen haben die Aufgabe, mit dem Dateisystem und Textdateien zu interagieren, wurden allerdings dennoch voneinander getrennt. Durch die Unterscheidung von „Input“ (Einlesen von Dateien) und „Output“ (Herausschreiben von Dateien) konnten zwei kleinere Interfaces erstellt werden, die handlicher sind. Außerdem erkennt man dadurch auf den ersten Blick, was die hier ausgeführte Aufgabe der Klasse und/oder der Methoden ist.

Negativ-Beispiel

Als negatives Beispiel dient wieder der Konverter. Auch wenn durch das Interface eine Abstraktion stattfindet und dieses recht überschaubar ist, ist es dennoch größer, als notwendig. Auch hier könnte



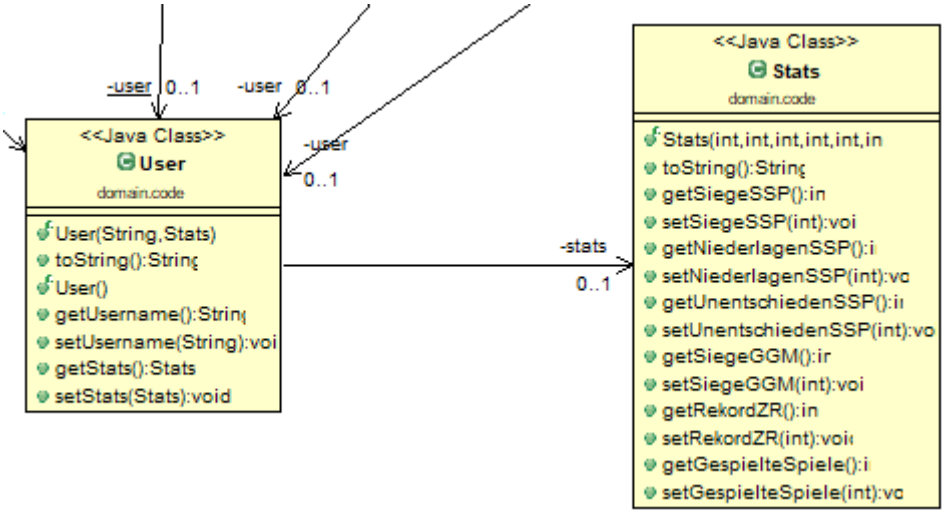
man zwischen Input und Output unterscheiden. Aktuell sammelt der Konverter alle Userdaten zusammen und übergibt sie dem Output, zusätzlich nimmt er die ausgelesenen Werte vom Input entgegen und gibt diese an das MainMenu weiter. Spätestens, wenn das Interface noch größer wird, wird ersichtlich, weshalb man hier einmal mehr separieren sollte.

Kapitel 4: Weitere Prinzipien

Analyse GRASP: Geringe Kopplung

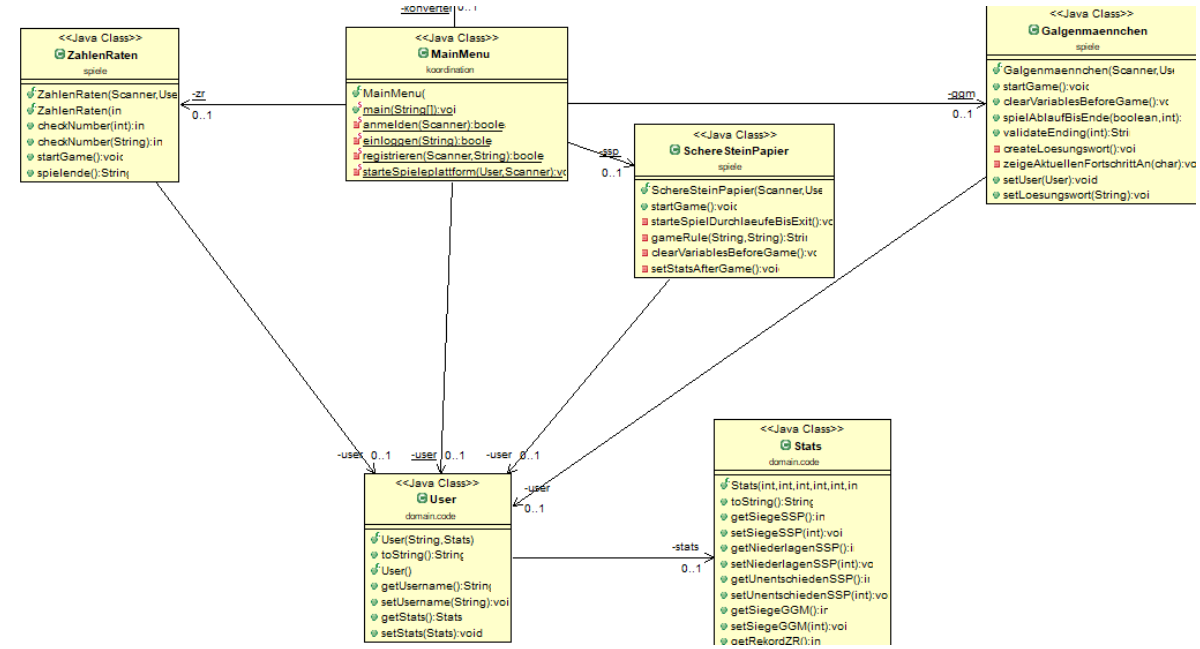
Positiv-Beispiel

Bei der User- und Stats-Klasse ist lediglich eine Abhängigkeit vom User zu seinen entsprechenden Stats. Keine der anderen Klassen greift direkt auf die Stats zu, wodurch das Prinzip erfüllt ist. Das macht auch Sinn, da nur der User seine Stats sehen muss, für die anderen Klassen sind die Stats Teil des Users und werden nur über ihn erreicht.



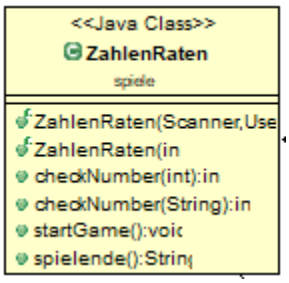
Negativ-Beispiel – behoben ab Commit

https://github.com/TimCzks/ASE_Spieleplattform/commit/6ffc146a11d4a9d299b5709a8577e6f7e9c49671



Nicht erfüllt ist dieses Prinzip jedoch bei den vielen Zugriffen auf das Userobjekt. Jedes der drei Spiele und das MainMenu greifen direkt auf den User zu. Die Spiele ändern dessen Stats nach einer gespielten Runde. Dadurch entsteht eine hohe Abhängigkeit zum User. Ändern kann man das, indem nach einer Partie eines Spiels nur eine Rückgabe an das MainMenu stattfindet und dort die Stats des Users angepasst werden. Damit hätte man nur noch eine Abhängigkeit zur User-Klasse.

Analyse GRASP: Hohe Kohäsion



Beim ZahlenRaten besteht eine hohe Kohäsion, da alle Methoden ausschließlich zur Umsetzung des Spiels beitragen und so auch eng miteinander verknüpft sind, vom Starten des Spiels über dessen Logik und einschließlich des Spielendes.

Don’t Repeat Yourself (DRY)

Commit https://github.com/TimCzks/ASE_Spieleplattform/commit/ca1a541a2a78aee2f34f0b029aa722bef549e955 bei

der Klasse MainMenu. Bei diesem Beispiel wurde eine Dopplung bei der Anmeldung eines Users aufgelöst. Die Methoden „einloggen“ und „registrieren“ haben beide den User durch ein System.out.println() dazu aufgefordert, seinen Nutzernamen angegeben und anschließend durch einen Scanner auf dessen Input gewartet.

```
private static boolean einloggen(Scanner sc) {
    System.out.println("Bitte gib deinen Benutzernamen an: ");
    String username = sc.next();
    user = konverter.erstelleUser(username);
    return true;
}
```

```
private static boolean registrieren(Scanner sc) {
    System.out.println("Bitte gib deinen zuk�nftigen Benutzernamen an:");
    String username = sc.next();
    user = new User(username, new Stats(0, 100, 0, 0, 0, 0));
    konverter.speichereUser(user);
    return true;
}
```

Diese Duplizierung wurde gelöst, indem die Aufforderung und die Eingabe in die übergeordnete Methode „anmelden“ überführt wurden. Dadurch sind die Methoden jetzt noch eindeutiger und z.T. schmaler als zuvor. Nach dem Refactoring sehen die Methoden wie folgt aus:

```
private static boolean einloggen(String username) {
        user = konverter.erstelleUser(username);
        return true;
}

private static boolean registrieren(Scanner sc, String username) {
        while (konverter.pruefeObUserBereitsExistiert(username)) {
                System.out.println("Der Benutzername '" + username + "' ist bereits
                                                vergeben, bitte wähle einen Neuen.");
                username = sc.next();
        }
        user = new User(username, new Stats(0, 100, 0, 0, 0, 0));
        konverter.speichereUserAb(user);
        return true;
}
```

Die Methode „registrieren“ hat allerdings noch die Funktionalität erhalten, zu prüfen, ob ein Username bereits vergeben ist, weshalb sie hier größer wirkt, auch wenn die Dopplung entfernt wurde.

Kapitel 5: Unit Tests

10 Unit Tests

Unit Test	Beschreibung

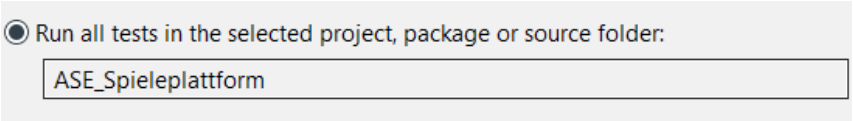
`KonverterTest#testErstelleUser`	Es wird getestet, ob das Erstellen eines User-Objekts richtig funktioniert. Dafür wird der Username des durch die Methode zurückgegebenen Users mit einem manuell erstellten User verglichen. Zur Überprüfung wird dabei der Username benutzt, da die User-Klasse keine equals-Methode implementiert.
`KonverterTest#testPruefeObUserBereitsExistiert`	Hierbei wird überprüft, ob der richtige Wert zurückgegeben wird, wenn vom Input-Objekt und dessen Methode `pruefeObUserBereitsExistiert` ein vorgegebener Wert kommt. In dem Fall sollte der erwartete Wert „true“ sein.
`GalgenmaennchenTest#testValidateEndingNiederlage`	Es wird getestet, dass beim erreichen einer Niederlage in GGM die erwartete Zahl mit der Angabe einer Niederlage (0) zurückgegeben wird.
`GalgenmaennchenTest#testStartGame`	Bei diesem Test wird die startGame-Methode dadurch getestet, dass verifiziert wird, dass sie beim aufrufen einmal ausgeführt wird.
`GalgenmaennchenTest#testSpielAblaufBisEnde`	Hier wird getestet, dass beim „spielAblaufBisEnde“ der erwartete Wert zurückgegeben wird, der die Anzahl an verbleibenden Rateversuchen nach Ende des Spiels darstellt.
`ZahlenRatenTest#testZahlenRaten`	Es wird getestet, dass die Logik zur Überprüfung der geratenen Zahl erwartungsgemäß funktioniert, indem die gesuchte Zahl manuell festgelegt wird. Erwartet wird die Prüfzahl 0 (geratene Zahl = gesuchte Zahl).
`ZahlenRatenTest#testException`	Es wird erwartet, dass beim Überprüfen der eingegebenen Zahl eine Exception geworfen wird, die durch die Eingabe eines Buchstabens provoziert wird. Dabei wird die erwartete Exceptionmessage überprüft.
`SchereSteinPapierTest#testGameruleUnentschieden`	Bei diesem Test wird die „gameRule“ der Klasse SchereSteinPapier geprüft. Es soll als Ergebnis der entsprechende String für ein Unentschieden zurückgegeben werden, was durch das Setzen des Symbols (z.B. „SCHERE“) des COM-Gegners kontrolliert erzielt werden kann.
`SchereSteinPapierTest#testStartGame`	Hier wird verifiziert, dass die Methode „startGame“ bei einem Aufruf auch dementsprechend genau einmal aufgerufen wird.
`SchereSteinPapierTest#testValidiereSpielergebnis`	Das Validieren des Spielergebnis wird hier dadurch getestet, dass zuerst alle Variablen des Spiels zurückgesetzt werden (was normalerweise vor jedem Spielstart passiert) und danach der Wert 0 erwartet wird, wenn man „validiereSpielergebnis“ aufruft und den 0. Wert des Arrays nimmt.

ATRIP: Automatic

Solange man sich bei dem Projekt in einer Testklasse befindet, kann man in der Entwicklungsumgebung ganz einfach über einen Knopfdruck (Bei Eclipse ‚F11‘) alle dort befindlichen

Tests ausführen. Alternativ geht das auch über einen Rechtsklick im Test und dem Auswählen der Option „Run as JUnit Test“.

Möchte man alle Tests auf einmal ausführen, kann man über einen Rechtsklick die Option „Coverage As“ und dann „Coverage Configurations“, um so im nächsten Schritt anzugeben, dass alle Tests des Projekts ausgeführt werden sollen.



Auch wenn so die Coverage angezeigt wird, werden dennoch alle Tests ausgeführt, um diese zu ermitteln und in Folge dessen würde man auch Einsicht darin haben, ob die Tests fehlschlagen oder nicht.

Somit gibt es verschiedene Möglichkeiten, alle Tests automatisch ausführen zu können, ohne einzelne Testmethoden manuell ausführen zu müssen.

ATRIP: Thorough

Der KonverterTest ist hierfür ein positives Beispiel, da er alle Methoden der ErbauerKonverterKlasse mit Tests abdeckt. Hier hat auch jeder Test seine Daseinsberechtigung, da alle Methoden essentielle Funktionen der Klasse abbilden. Ein Beispieltest, der ausreichend die Funktionen der Methode des

```
@Test
void testErstelleUser() {
Mockito.when(input.leseDatenVonDatei(Mockito.isA(String.class),Mockito.isA(String.class))
                        .thenReturn(new String[] { "0", "0", "0", "0", "0", "0" });
Mockito.doCallRealMethod().when(classUnderTest).erstelleUser(Mockito.isA(String.class));
             User userByMethod = classUnderTest.erstelleUser("username");
             User userToCompare = new User("username", new Stats(0, 0, 0, 0, 0, 0));
             assertEquals(userToCompare.getUsername(), userByMethod.getUsername());
```

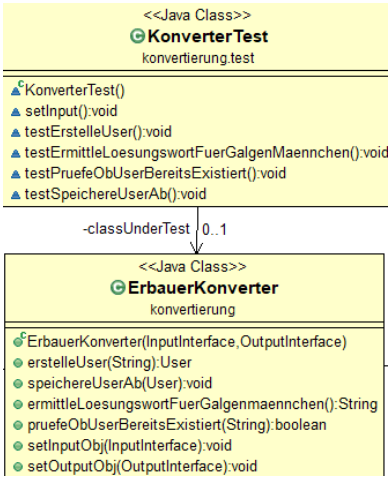


Abbildung 2: Testmethoden für alle entsprechenden Funktionen des ErbauerKonverters

Erstellens eines Users testet, ist hier abgebildet. Dabei werden alle anderen Tests mit dem gleichen Prinzip aufgebaut, dass die echte Methode über die Bibliothek Mockito aufgerufen wird. Die benötigten Zwischenschritte anderer Klassen werden dabei simuliert, um die echte Methode aufrufen zu können, ohne einen großen Mehraufwand zu haben. Bei dem Beispiel der Methode ist der simulierte Zwischenschritt vom Input-Objekt, der wie folgt aussieht:

```
String[] s = inputObj.leseDatenVonDatei(username, ",");
```

Negativ-Beispiel: SchereSteinPapierTest. Hier sind fast alle Funktionen des Spiels SchereSteinPapier vom Test abgedeckt, bis auf eine Essentielle. Die Methode `starteSpielDurchlaeufeBisExit()` ist nicht von der Testklasse überprüft. Hier wird eine Schleife ausgeführt, welche jede neue Eingabe des Users überprüft und je nachdem eine Spielregel darauf anwendet, oder den Nutzer*innen Feedback gibt, dass die Eingabe nicht erkannt wurde. Somit ist die hier abgebildete Funktion fundamental für das Spielprinzip und verleitet dabei immer wieder auf andere Methoden (die `gameRule(String equal, String lose)`). Durch die Komplexität, die mit der Schleife einhergeht, wurde an dieser Stelle von einem Test abgesehen, auch wenn dadurch das Konzept von „Thorough“ verletzt wird.

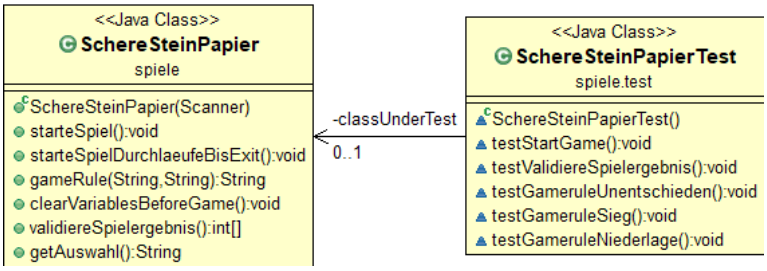


Abbildung 3: Spielklasse und zugehörige Testklasse

ATRIP: Professional

Positiv-Beispiel: KonverterTest. Auch hier ist der KonverterTest ein gutes Beispiel für das Prinzip. Wie bereits bei „Thorough“ angegeben, werden alle grundlegenden Methoden des ErbauerKonverters getestet. Jede dieser Methoden hat genau eine Aufgabe, dessen Erfüllen vom Konverter an anderen essentiellen Stellen im Projekt erwartet wird. Darüber hinaus wurde in den Tests selbst das Hauptmerkmal auf die zu testende Methode gelegt, Zwischenschritte anderer Objekte werden durch Mocks vorbestimmt, um keine realen Aufrufe anderer Methoden zu haben, deren Funktionieren sowieso im Test vorausgesetzt wird. Zu sehen ist das u.a. in folgender Methode:

```
@Test
void testPruefeObUserBereitsExistiert() {
Mockito.when(input.pruefeObUserBereitsExistiert(Mockito.isA(String.class))).thenReturn(true);
Mockito.doCallRealMethod().when(classUnderTest).pruefeObUserBereitsExistiert(Mockito.isA(String.clas
s));
assertEquals(true, classUnderTest.pruefeObUserBereitsExistiert("username"));
}
```

Die Testklasse besitzt außerdem noch die Methode `setInput()`, die vor jedem Test ausgeführt wird (gekennzeichnet mit).

```
Mockito.doCallRealMethod().when(classUnderTest).setInputObj(Mockito.isA(Input.class));
classUnderTest.setInputObj(input);
```

Hierbei wird das benötigte Input-/Output-Objekt in der zu testenden Klasse durch einen Mock ersetzt, dessen Verhalten bestimmt werden kann. Die dafür vorgesehenen Setter wurden jedoch nicht getestet, was ebenfalls dem Prinzip der Professionalität entspricht. Setter, wie auch Getter, besitzen keine Komplexität, die zu testen wäre.

Negativ-Beispiel: DirektorKonverterTest. Bei dieser Testklasse sind etliche Tests ohne wichtige Bedeutung. Da die zu testende Klasse DirektorKonverter in seinen Methodenaufrufen sowieso nur weiterdirigiert und somit auf die Methoden der Klasse „ErbauerKonverter“ verweist, sind diese bereits in dessen Testumfang enthalten. Der Aufruf des Erbauers ist dabei die einzige, notwendige Funktion, die zu testen wäre. Da dies gegeben ist, müsste man nicht jede Methode einzeln überprüfen. Durch die geringe Komplexität der Methoden widerspricht das Testen dieser der Professionalität der Testklasse. Man könnte das beheben, indem man alle Tests, bis auf einen, entfernt und somit nur noch einmal den korrekten Verweis auf den Erbauer überprüft.

Code Coverage

Die Code Coverage über das gesamte Projekt beträgt 54,2%. Dieser Wert ergibt sich unter anderem aus 50,4% der Application-Layer, 55,7% der Plugin-Layer, 46,4% des DomainCodes und 85,7% der Adapter-Layer. Darin befindlich sind u.a. die Spiele, die Bestenliste, das Input-Objekt, sowie der ErbauerKonverter, welcher vollständig getestet ist (bis auf Setter-Methoden). Die Spiele ergeben dabei eine Coverage von 29,5%, der ErbauerKonverter beispielsweise ist dagegen bei 94,3%. Bei den Spielen wurde trotz des vergleichsweise geringen Werts maßgeblich die wichtigsten Funktionen getestet. Das MainMenu und die PlattformVerwaltung, welche ebenfalls in der Application-Layer liegen, sind zu 0% getestet. Das ist damit zu begründen, dass beide Klassen Funktionen anbieten, die sich meistens in Schleifenaufrufen befinden oder nur einmal zum Start der Plattform aufgerufen werden. Somit wurde keine Notwendigkeit darin gesehen, hierfür Testklassen anzulegen. Durch den einmaligen Aufruf direkt zu Beginn der Programmausführung würde ohnehin direkt ein Fehler geworfen werden, wenn etwas nicht funktioniert. Die Schleife wiederum ist, wie im Fall des Spiels „SchereSteinPapier“ schwer zu testen und wurde deshalb vernachlässigt. Besser sieht es dagegen wieder bei der Pluginschicht aus. Hier ist die gesamte Coverage bei 55,7%, wobei der Input zu 77% getestet ist und der Output zu 0%. Das hat den Grund, dass beim Input auf Textdateien zugegriffen wird und deren ausgelesener Inhalt zurückgegeben wird, was sich gut überprüfen lässt. Das Erstellen einer Datei über die Output-Klasse ist dagegen schwierig zu testen. Weitere Abstriche wurden bei den Observern der Bestenliste gemacht. Hier wurde nur der „BestenlisteObserverGGM“ getestet, da alle anderen Observer ohnehin sehr ähnlich aussehen und dadurch mit dem einen Test als getestet angesehen werden. Die Bestenliste wiederum hat einen eigenen Unit-Test, da hier essentielle Funktionen für das Bereistellen dieser vorliegen.

Fakes und Mocks

Mock-Objekte kamen bei fast allen Tests zum Einsatz, zwei Beispiele dazu sind der KonverterTest und der GalgenmaennchenTest. Dabei wurde fast immer das Test-Framework Mockito zur Unterstützung

genommen. Hierüber kann man simpel Klassen mocken und als Mocks dann deren Verhalten steuern. Ein erstes Beispiel, bei dem das Mockobjekt manuell erstellt wurde (Ohne Hilfe von Mockito), findet sich beim KonverterTest wieder. Für das Testen aller Methoden, welche Ergebnisse des Inputobjekts erwarten, wurde der Rückgabewert durch eine Mockklasse „InputMock“ vorgegeben, welche die Input-Klasse erweitert. Beispielsweise bei der Prüfung des Erstellens eines Users wird beim Aufruf des InputMocks lediglich das erwartete Ergebnis der Inputklasse, nämlich ein String-Array mit Werten, direkt zurückgegeben. Konkret sehen der Test, sowie die Methode des InputMocks, wie folgt aus:

```
@Test
void testErstelleUser() {
	User userByMethod = classUnderTest.erstelleUser("username");
	User userToCompare = new User("username", new Stats(0, 0, 0, 0, 0, 0));
	assertEquals(userToCompare.getUsername(), userByMethod.getUsername());
}
```

Auszug des InputMocks:

```
@Override
public String[] leseDatenVonDatei(String dateiname, String seperator) {
	if (dateiname.equals("GalgenmaennchenWoerter")) {
		return new String[] { "Loesungswort" };
	}
	return new String[] { "0", "0", "0", "0", "0", "0" };
}
```

Die einzige Unterscheidung, die der Mock macht, ist bei der Eingabe „GalgenmaennchenWoerter“ , da hier ein anderes Ergebnis erwartet wird. Ansonsten reicht standardmäßig der Rückgabewert des Arrays.

Ein weiteres, komplexeres Beispiel findet sich bei dem GalgenmaennchenTest wieder. Für das Testen einer void-Methode wird hierbei dessen Funktionsaufruf verifiziert. Damit dieser Aufruf reibungslos abläuft, wird das Verhalten aller anderen Methoden innerhalb der zu testenden Methode mit realistischen Werten vorgegeben. Die void-Methode „clearVariablesBeforeGame“ beeinflusst dabei keinerlei die Funktion, weshalb hier bestimmt werden kann, dass bei diesem Aufruf nichts passieren soll.

```
@Test
void testStartGame() {
Mockito.when(classUnderTest.spielAblaufBisEnde(Mockito.isA(Boolean.class))).thenReturn(5);
Mockito.doNothing().when(classUnderTest).clearVariablesBeforeGame();
Mockito.when(classUnderTest.validiereSpielergebnis(Mockito.isA(Integer.class))).thenReturn
(1);
Mockito.doCallRealMethod().when(classUnderTest).starteSpiel();
classUnderTest.starteSpiel();
Mockito.verify(classUnderTest, Mockito.times(1)).starteSpiel();
}
```

Kapitel 6: Domain Driven Design

Ubiquitous Language

Bezeichung	Bedeutung	Begründung

User	Die Nutzer der Anwendung.	Ein zentraler Bestandteil, die Nutzer der Anwendung, werden im Programmcode als User dargestellt und auch in der Anwendung selbst als „User“ angesprochen. Der englische, aber im deutschen sehr verbreitete Begriff, eignet sich somit gut als Begriff der UL.
Spiel	Ein allgemeiner Begriff, der die möglichen Spiele der Plattform beschreibt.	Die Spiele sind essentiell für die Spieleplattform und dieser übergeordnete Begriff eignet sich sehr gut sowohl für die Entwicklung, als auch für das Projektmanagement.
Stats	Die Statistik eines Nutzers, die seine gesamten Erfolge der einzelnen Spiele erfasst.	Die Stats sind essentiell für den User. Auch ‚Stats‘ ist ein sehr viel im deutschen verwendeter Begriff, unter dem man die Statistik in einem Spiel versteht.
Lösungswort	Ein gesuchter Begriff, der vom User erraten werden muss, unabhängig vom Spiel.	Das Lösungswort ist ein zentraler Bestandteil des Spiels Galgenmännchen und könnte auch bei neuen Spielen relevant sein, weshalb es zur UL gehört.

Entities

Bei diesem Projekt bietet der Einsatz von Entities keinen zusätzlichen Vorteil, dennoch müsste man zum Implementieren viel Aufwand betreiben. Da diese meist eine zentrale Stelle zur Verwaltung von relevanten Daten sind, die zumeist als Value Objects dargestellt werden, ist deren Nutzen sehr gering, da auch die Komplexität der Funktionen und Anforderungen an die Spieleplattform sehr gering ist.

Value Objects

Es gibt zudem keine Notwendigkeit, Value Objects zu verwenden. Die Komplexität der genutzten Datenstrukturen ist zu gering, als das durch VO’s ein Mehrwert entstehen würde. Der Aufwand zum Implementieren von Value Objects wäre deshalb für den daraus resultierenden Nutzen zu hoch.

Repositories

Da es keine Aggregates gibt, ist der Einsatz von Repositories ebenso sinnlos, da es keine abgegrenzten Objekte gibt, die durch ein Repository verwaltet werden müssten.

Aggregates

Ohne Entities ist es nicht notwendig, Aggregates zu verwenden. Normalerweise gliedern diese mehrere Entities voneinander ab und vereinfachen somit die Übersicht über alle Beziehungen zwischen Entities. Durch den Wegfall der Entities erübrigt sich der Einsatz jedoch.

Kapitel 7: Refactoring

Code Smells

Code Smell 1: Duplicated Code. In der Klasse „Input“ gibt es zwei Methoden, die beide eine Textdatei auslesen und den ausgelesenen Inhalt als String-Array zurückgeben. Unterschieden haben sich die Methoden nur in der Angabe des Dateipfades (Dateiname). Ganze Methode:

```
public String[] leseLoesungswoerterVonDatei() {
	File f = new File(PATH + "GalgenmaennchenWoerter.txt");
	String[] s = null;
	try {
		BufferedReader br = new BufferedReader(new FileReader(f));
		while (br.ready()) {
			s = br.readLine().split(",");
		}
		br.close();
	} catch (FileNotFoundException e) {
		e.printStackTrace();
	} catch (IOException e) {
		e.printStackTrace();
	}
	return s;
```

Unterschied bei der Methode „leseUserDatenVonDatei“:

```
public String[] leseDatenVonDatei(String username) {
	File f = new File(PATH + username + ".txt");
```

Behoben wurde der Code Smell, indem eine Methode „`leseDatenVonDatei`“ geschrieben wurde, die einen String „dateiname“ als Parameter bekommt. An der Stelle, wo zuvor „`leseLoesungswoerterVonDatei`“ aufgerufen wurde, ist dann die neue Methode eingefügt worden, die als Parameter den Dateinamen „Galgenmaennchen“ bekommt. Die zuvor parallel existierenden Methoden wurden gelöscht. Nachzusehen ist das im Commit https://github.com/TimCzks/ASE_Spieleplattform/commit/d53e241d8726991c38a85f1c8e4d51567c32ab1e

Code Smell 2: Large Class. Die Klasse „MainMenu“ war bis zum Commit https://github.com/TimCzks/ASE_Spieleplattform/commit/1f9bd967cbfea82ca7b8a5ce345df74f9902fce0 deutlich zu groß für die Aufgabe dieser Klasse. Hier sollte lediglich die Anmeldung durchgeführt werden, was in kürzeren Methoden simpel umzusetzen ist. Zusätzlich wurde die Methode „starteSpieleplattform“ ausgeführt, die sehr lang ist. Dadurch wurde die gesamte Klasse unnötig unübersichtlich. Die Klasse hatte zuvor 99 Codezeilen, nach dem Refactoring sind es noch 54, wodurch ersichtlich wird, die groß die Klasse durch diese Methode war. Als Lösungsmaßnahme wurde eine neue Klasse „PlattformVerwaltung“ erstellt, die sich nur um das Starten und Ausführen der Spieleplattform kümmert. Aufgerufen wird die Methode über den Konstrukter dann im MainMenu wie folgt:

```
new PlattformVerwaltung(user, sc);
```

2 Refactorings

Refactoring 1: Extract Method, siehe commit https://github.com/TimCzks/ASE_Spieleplattform/commit/558d324e09ce541831b16ef4f4be9877831bd4fd. Die in der extrahierten Klasse „PlattformVerwaltung“ befindliche Methode „starteSpieleplattform“ ist sehr lang und soll deshalb in mehrere Unterteilt werden. Der vorher/ nachher-Vergleich sieht dabei wie folgt aus:

Abbildung 5: UML vor extract method

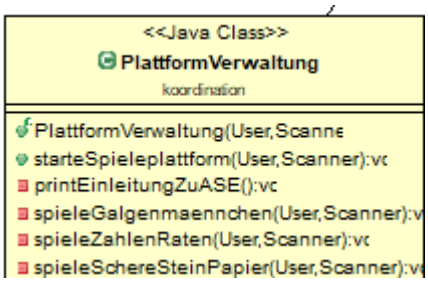


Abbildung 4: UML nach extract method

Dafür wurde jeder Spielaufruf, wie auch der sehr lange System.out.println-Aufruf in einzelne Methoden ausgelagert, was sich somit auch logisch gut erklären lässt.

Refactoring 2: Rename method. Die Methode `ermittleLoesungswort` war nicht eindeutig genug, da es sich dabei um das Ermitteln eines Lösungswortes für das Spiel Galgenmännchen handelt. Sollte die Plattform allerdings irgendwann noch um ein anderes Spiel ergänzt werden, bei dem ebenfalls ein Lösungswort existiert (z.B. Montagsmaler o.ä.), dann wäre nicht mehr zuzuordnen, für welches Spiel hier ein Lösungswort erstellt wird. Nachzusehen im Commit https://github.com/TimCzks/ASE_Spieleplattform/commit/6179f1262b9b68dcd7fdb8e31dd7472165cf5c11.

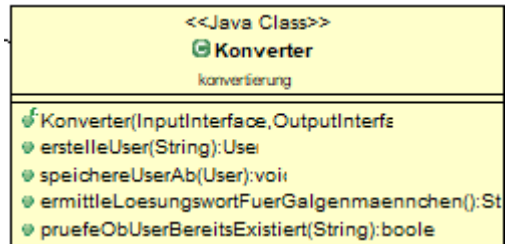


Abbildung 7: Methodenname nach dem Refactoring

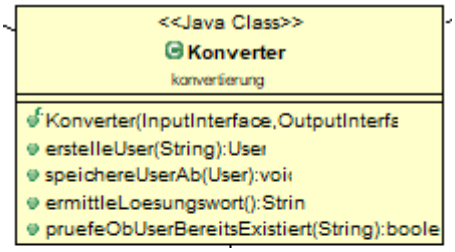


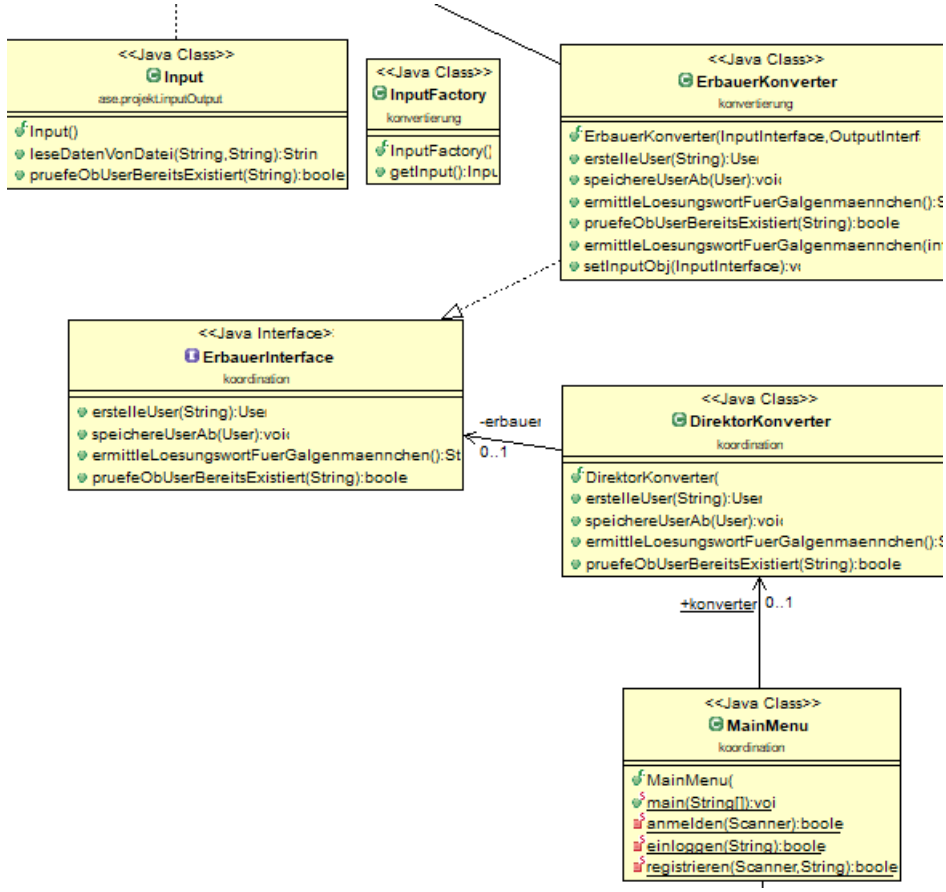
Abbildung 6: Methodenname vor dem Refactoring

Kapitel 8: Entwurfsmuster

Entwurfsmuster: Erbauer

Der Erbauer wurde an der Stelle des Konverters eingesetzt (Commit https://github.com/TimCzks/ASE_Spieleplattform/commit/e2d0c352600f92dedcd0adfab2d0b21880419d12) und ist durch das „ErbauerInterface“ definiert. Für die Konstruktion beispielsweise eines

Users, oder dem Speicherobjekt des Users, wird nun der Erbauer verwendet, bzw dessen konkrete Implementierung. Dadurch ergibt sich der Vorteil, dass man alle Funktionen über den Direktor abruft und dieser die konkreten Methoden nicht sehen kann, weshalb eventuell neu hinzufügende Erbauerklassen unabhängig vom Direktor eingebunden werden können. Falls beispielsweise der Input eines bestimmten Datentyps in eine neue Erbauerklasse ausgelagert werden soll, um mehr Struktur zu schaffen, dann ermöglicht dieses Modell des Erbauers ein simples Erweitern. Auch die Änderung einer Funktion durch eine neue Erbauerklasse ist so einfach zu gestalten.



Entwurfsmuster: Beobachter

Der Beobachter kommt zum Einsatz, um die Bestenliste nach Ende eines Spiels zu aktualisieren. Dafür wird zu Beginn der Spieleplattform jeder Observer zur Observerliste des Observables hinzugefügt. Konkret ist das Observable die Klasse „BestenlisteObserverVerwaltung“. Nach dem Ende eines Spiels wird bei dieser Klasse die Methode „setNewStat“ aufgerufen, die wie folgt aussieht.

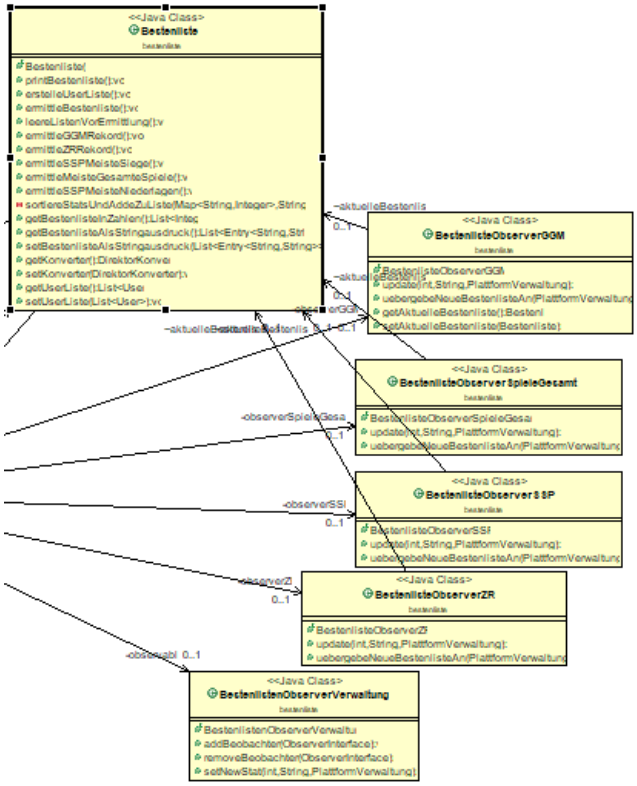
```
public void setNewStat(int newStat, String spiel, PlattformVerwaltung spielePlattform) {
	this.newStat = newStat;
	this.spiel = spiel;
	for (ObserverInterface observer : this.observerListe) {
		observer.update(this.newStat, this.spiel, spielePlattform);
	}
}
```

Hier wird dann für jeden Beobachter dessen Update-Methode ausgeführt. Innerhalb dieser Funktion wird überprüft, ob der Beobachter für das angegebene Spiel verantwortlich ist und ob durch das neuste Ergebnis ein neuer Rekord in der Bestenliste erzielt wurde. Ist das nicht der Fall, passiert nichts weiter. Ansonsten wird allerdings der Eintrag der Bestenliste an der Stelle des Spiels durch den aktuellen Nutzer mit dessen neuem Rekord überschrieben. Implementiert sieht diese Methode folgendermaßen aus.

```
@Override
public void update(int newStat, String spiel, PlattformVerwaltung spielePlattform) {
aktuelleBestenliste.ermittleBestenliste();
if (spiel.equals("'Galgenmännchen'") && newStat >
       aktuelleBestenliste.getBestenlisteInZahlen().get(0)) {
       aktuelleBestenliste.getBestenlisteAlsStringausdruck().set(0,
                                  Map.entry(spiel, "Beste*r Spieler*in in " + spiel + " ist
       " + spielePlattform.getAktuellerUser().getUsername() + " mit " + newStat + "
       Siegen."));

       uebergebeNeueBestenlisteAn(spielePlattform);
       }
}
```

Im Klassendiagramm ist noch einmal ersichtlich, dass es für jeden Rekordfall der Bestenliste einen eigenen Observer gibt. Jedes dieser Objekte hat dabei eine Referenz auf die Bestenliste und auf die PlattformVerwaltung, welche sich beide in der selben Schicht der Clean Architecture verwenden.



Durch die Varietät an Einträgen in der Bestenliste ist die Implementierung eines Beobachters an dieser am Sinnvollsten. Durch die einzelnen Observer hat man die Verantwortlichkeiten für die Statistik des aktuellen Nutzers klar durch die Spiele, beziehungsweise Anwendungsfälle getrennt.