DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

Kleine Spielesammlung von 4-Gewinnt Versionen

Bachelor of Science

Studiengang Informationstechnik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Andreas Schmider

Abgabedatum 30. April 2022

Bearbeitungszeitraum	2 Semester
Kurs	TINF19B3

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Beschreibung der Software

Beschreibung der Funktionalität
Aus einem alten Projekt existiert schon ein Connect6 Spiel, das in der Konsole gespielt wird. Dabei gibt es einige Einstellungen die während des Programms noch gesetzt werden können. Somit kann es von zwei bis vier Spielern gespielt oder die Spielfeldgröße auf 18x18 oder 20x20 festgelegt werden. Eine der größeren Auswirkung hat die Auswahl der Spielregeln. Connect6 ist ähnlich wie Vier gewinnt und Ziel ist es mehrere Steine in eine Reihe zu legen. Connect6 wird dabei aber auf einem Brett gespielt, das eben auf dem Boden liegt. So fallen die Steine nicht bis in die letzte Zeile sondern bleiben an ihrem Platz. Ebenso werden pro Zug zwei Steine pro Spieler gesetzt. Bei der Standart-Spielregel zählen die Steine nur bis zum Spielfeldrand. Es gibt aber auch die Torus-Version, bei dem die Steinreihen über den Rand hinaus gezählt werden. So beginnt das Spielfeld erneut von oben, wenn man aus der untersten Zeile eins nach unten gehen würde.

Zusätzlich soll in Zukunft auch noch Vier Gewinnt mit unterschiedlichen Variationen/Regeln implementiert werden. Bisher wurden nur die Grundsteine dafür gelegt. Bei dieser Spielesammlung soll es möglich sein mit wenig Aufwand neue Spielregeln/Variationen oder sogar ganz neue Spiele (die einen ähnlichen Aufbau haben) zu implementieren.

Bei der Entwicklung dieser Software wurde zwar darauf geachtet einen möglichst strukturierten und einfachen Code zu erstellen, es wurde sich aber nicht direkt an die in dieser Vorlesung besprochenen Themen gehalten. Es wurde zwar versucht die DRY-Regel anzuwenden, da diese mir schon selber in den Sinn gekommen ist, aber wie ich später festgestellt habe wurde diese nicht überall umgesetzt. Ebenso habe ich öfters kleinere Code Reviews vorgenommen und refactored, da mir immer wieder beim erneuten Anschauen aufgefallen ist, wie unverständlich der Code ist. Da der Code ursprünglich von einer Programmieraufgabe am KIT stammt, mussten auch Kommentare verwendet werden. Nachdem ich diesen Code aber nach über 3 Jahren wieder angeschaut habe ist mir klar geworden, dass trotz vieler Kommentare der Code sehr schwierig zu lesen ist.

Beschreibung des Kundennutzens Der Kunde erhält Spaß und Unterhaltung.

Verwendete Technologien - Java - JUnit 5 - IntelliJ

Link zum Repository `https://github.com/a-schmider/ASE_Substrat`

2 Unit tests

Mit den bisher erstellten Tests werden ?% der Programmzeilen abgedeckt. Dies wurde mittels Code Coverage ermittelt. IntelliJ bietet dies standardmäßig schon an und zeigt genau welcher Code ausgeführt wurde und welcher nicht. ?% ist sehr wenig aber auch nicht verwunderlich bei diesen kleinen und wenigen Tests.

Current scope: all classes

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	6,7% (2/30)	3,9% (6/153)	1,9% (8/428)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
dhbw.ase.core.gamerules	0% (0/2)	0% (0/12)	0% (0/52)
dhbw.ase.core.gamerules.connect6	0% (0/5)	0% (0/17)	0% (0/35)
dhbw.ase.core.gamerules.connect_four	0% (0/3)	0% (0/14)	0% (0/14)
dhbw.ase.core.misc	0% (0/6)	0% (0/18)	0% (0/40)
dhbw.ase.core.models	25% (2/8)	17,6% (6/34)	11,6% (8/69)
dhbw.ase.plugin.games	0% (0/4)	0% (0/41)	0% (0/144)
dhbw.ase.plugin.main	0% (0/1)	0% (0/3)	0% (0/21)
dhbw.ase.plugin.userinterface	0% (0/1)	0% (0/14)	0% (0/53)

Abbildung 1: Test-Coverage über das komplette Programm

Wenn man sich aber die die getesteten Klassen ansieht, sieht das schon ganz anders aus. Dort wurde 100% der Zeilen abgedeckt.

Class	Class, %	Method, %	Line, %
Compass	0% (0/1)	0% (0/1)	0% (0/2)
Connect6GameBoard	0% (0/1)	0% (0/1)	0% (0/3)
GameBoard	0% (0/1)	0% (0/4)	0% (0/5)
GameField	100% (1/1)	100% (5/5)	100% (7/7)
GameInfo	0% (0/1)	0% (0/9)	0% (0/13)
Player	100% (1/1)	33,3% (1/3)	20% (1/5)
PlayerStone	0% (0/1)	0% (0/2)	0% (0/3)
RectangularGameBoard	0% (0/1)	0% (0/9)	0% (0/31)

Abbildung 2: Test-Coverage für die getesteten Klassen

2.1 ATRIP

2.1.1 Automatic

Damit die Test automatisch ausgeführt werden, wurde in IntelliJ ein Run Configuration erstellt, womit alle Tests ausgeführt werden. Damit sich nach einer Änderung keine Fehler einschleichen, wurde auch in der Normalen Run Konfiguration für das Programm festgelegt, dass die Test im vor hinein ausgeführt werden sollen, bevor das Programm gestartet wird.

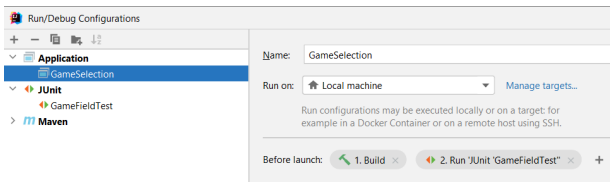

Abbildung 3: Run-Konfiguration des Programmes

Sofern die Projekt-Dateien auch im Git-Repository hinterlegt sind, haben alle Entwickler diese Konfigurationen zur Verfügung und können diese Verwenden.

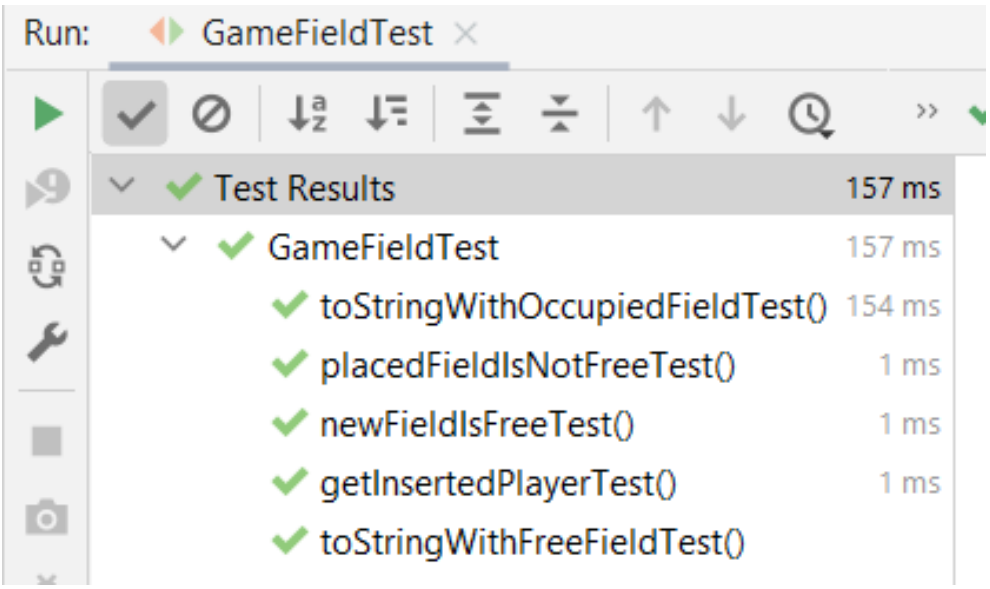

Abbildung 4: Automatisierung der Tests

2.1.2 Thorough

Vieles in meinem Programm arbeitet mit anderen Klassen zusammen. So ist es nicht so einfach möglich mit Unit Tests zu testen, ob die Spielereingaben korrekt sind. Dies würde sich über mehrere Berichte ziehen und es würde ein Integrationstest werden.

Ein grundlegende Funktion, worauf vieles aufbaut, sind die GameFields. Also die einzelnen Felder auf die ein Stein gesetzt werden kann. Deshalb wurden diese als erstes getestet.

2.1.3 Repeatable

Bei den getestet Klassen wurde darauf geachtet, dass die Tests nicht von zufälligen Eingaben oder ähnlichem abhängig sind. Somit sollten die Test beliebig wiederholt werden können und immer das Gleiche Ergebnis liefern.

2.1.4 Independent

Alle Tests wurden so aufgebaut, das alles, was für die Tests benötigt wurde auch nur in diesem einen Test erstellt wird. So können die Tests in beliebiger Reihenfolge ausgeführt werden.

2.1.5 Professional

Damit die Tests einfach zu verstehen sind, wurden die AAA Regeln bei allen Tests angewandt. Jedoch waren bei einigen Tests nicht alle Schritte notwendig und wurden deshalb weggelassen . Zu sehen ist das zum Beispiel bei dem Test mit Mocks in Abbildung 5.

2.2 Mocking

Um die Klasse GameField zu testen wurden Mocks verwendet. Genauer gesagt, bei der toString-Methode. Diese Methode gibt den Wert, von dem Spielstein als Text zurück, der auf diesem Feld liegt. Dafür wird natürlich eine Instanz der Klasse Spieler verwendet. Dementsprechend wird dieses Objekt gemocked. Eigentlich will die GameField-Klasse aber den Spielstein, der auch noch in dem Spieler liegt. Deshalb wird der Spielstein auch gemocked. Die Player-Instanz verwendet dann die gemockte PlayerStone-Klasse. Schluss endlich kann das GameField getestet werden ohne die anderen Klassen zu verwenden.

```
@Test
public void toStringWithOccupiedFieldTest() {
    //Arrange
    GameField field = new GameField();
    PlayerStone stone = Mockito.mock(PlayerStone.class);
    Mockito.when(stone.getLabel()).thenReturn("P1");
    Player player = Mockito.mock(Player.class);
    Mockito.when(player.getPlayerStone()).thenReturn(stone);

    field.placeStone(player);

    //Act
    String fieldRepresantation = field.toString();

    //Assert
    assertEquals( expected: "P1", fieldRepresantation);
}
```

Abbildung 5: Verwendung von Mocks bei Unit-Tests

3 Programming Principles

3.1 SOLID

3.1.1 Single-Responsibility-Principle

Im Code wurde das bei den Spielregeln angewendet. Für jede Spielversion gibt es eine Klasse die nur überprüft, ob ein Zug gemacht werden darf und ob ein Gewinner vorliegt. So gibt es bisher zwei Spielvarianten für Connect6 die in unterschiedlichen Klassen implementiert sind und nicht auf eine Helper-Klasse o.ä. angewiesen sind und dort die Spielvariante abfragen müssen, um zu entscheiden, ob ein Zug erlaubt ist oder nicht. Definiert wurde das über Interfaces und der abstrakten BoardGameRule-Klasse. Die davon abgeleiteten Klassen implementieren dafür weitere Methoden, die das besser strukturieren, aber keine, die zusätzliche , ungewollte Funktionen implementieren.

```
public abstract class BoardGameRule implements Winnable, PlaceableCheck {

    @Override
    public boolean checkAllowedPlacement(RectangularGameBoard board, int width, int height) { return false; }

    public abstract Player checkWin(RectangularGameBoard board, Command command);

    /**
     * Should return the name of the GameBoardRule
     *
     * @return the name of the GameBoardRule
     */
    public abstract String toString();
}
```

Abbildung 6: Die abstrakte BoardGameRule-Klasse

3.1.2 Open/Closed-Principle

Angewendet wurde das auf die Spielregeln. Zu Beginn gab es nur zwei Klassen für Spielregeln, die BoardGameRule- und die ConnectGameRule-Klassen. In ConnectGameRule wurde der grundsätzliche Ablauf für diese Art von Spielregeln erstellt. Durch die Unterklassen können aber neue Varianten implementiert werden, im Code mit den Connect6StandardGameRule- und Connect6TorusGameRule-Klassen. So könnten in Zukunft noch viele weitere Spielregel erstellt werden, welche diese Regeln anpassen ohne bestehenden Code zu ändern.

```java
public class Connect6TorusGameRule extends Connect6GameRule {

    @Override
    protected boolean checkOnBoard(RectangularGameBoard board, int width, int height) { return true; }

    @Override
    protected int getNextWidth(RectangularGameBoard board, int width, Compass direction) throws NoSuchFieldException {...}

    @Override
    protected int getNextHeight(RectangularGameBoard board, int height, Compass direction) throws NoSuchFieldException {...}

    @Override
    public boolean checkAllowedPlacement(RectangularGameBoard board, int width, int height) {...}

    @Override
    public String toString() { return "Torus"; }

}
```

Abbildung 7: Die Connect6TorusGameRule-Klasse

3.1.3 Liskov-Substitution-Principle

Dieses Prinzip wurde bei den Spielbrettern verletzt. Es wurde nämlich ein quadratisches Spielfeld als Unterklasse eines Rechteckigen Spielfelds implementiert. Somit verhält sich das quadratische Spielfeld aber nicht nicht mehr wie ein Rechteckiges Spielfeld und schränkt die Funktionalität sogar ein, anstatt sie zu erweitern. Es gibt zwar keine Methoden um die Länge und Breite zu setzen aber einen Konstruktor, der diese setzt. Dort könnte zwar ein Fehler geworfen werden aber das führt wieder zu „schlechtem “Code. Deshalb sollte dort ein neues Konstrukt angestrebt werden.

3.1.4 Interface-Segregation-Principle

Bisher gibt es in diesem Projekt nur drei Interfaces. Eins für die GUI und zwei für Spielregeln. Bei den Interfaces für Spielregeln gibt es das Winnable Interface, das die Methode checkWin(...) und das PlaceableCheck Interface, das eine checkAllowedPlacement(...) definiert. Bisher werden diese Zwei nur von der Klasse BoardGameRule verwendet. In Zukunft könnte es aber ein Spiel geben, z.B. Monopoly, bei dem es zwar einen Gewinner gibt, aber jeder Zug erlaubt ist sodass nur Winnable und nicht PlaceableCheck implementiert werden müsste.

3.1.5 Dependency-Inversion-Principle

Normalerweise wird Dependency-Inversion verwendet um die Abhängigkeiten innerhalb des Programms umzudrehen. In diesem Programm wurden nur zwei Schichten der Clean-Architecture erstellt. Diese wurden auch erst sehr spät implementiert. Deshalb wurde nur versucht die Abhängigkeiten zwischen den Schichten zu regeln, damit nur die Äußeren Schichten von den Inneren Schichten abhängig sind. Dafür waren aber alle Abhängigkeiten schon in die richtige Richtung. Der erste Schritt zur

Dependency-Inversion ist die Dependency Injection. Zumindest diese wurde an einigen Stellen umgesetzt. Zum Beispiel bei der Connect6-Klasse aus der Plugin-Schicht und der ConnectGameRule-Klasse aus dem Kern verwendet. Die Connect6-Klasse besitzt drei Listen. Eine für die möglichen Spielregeln, eine für die möglichen Spielbrettgrößen und eine für mögliche Anzahl an Spielern. Alle drei Listen werden über den Konstruktor übergeben. Somit wird an dieser Stelle Dependency Injection angewandt.

```
public class Connect6 extends TurnBasedGame {

    private final List<ConnectGameRule> possibleGameRules;
    private final List<Integer> possibleBoardSizes;
    private final List<Integer> possiblePlayers;

    private GameInfo gameInfo;
    private RectangularGameBoard gameboard;


    public Connect6(List<ConnectGameRule> possibleGameRules, List<
        this.possibleGameRules = possibleGameRules;
        this.possibleBoardSizes = possibleBoardSizes;
        this.possiblePlayers = possiblePlayersCount;
        gui = new ConsoleGUI();
    }
```

Abbildung 8: Die Dependency Injection bei der Connect6-Klasse

3.2 GRASP

Im laufe der Vorlesung wurden neun Faktoren von General Responsibility Assignment Software Patterns/Principles (GRASP) besprochen. Davon wurden aber nur zwei (Low Coupling und High Cohesion) genauer besprochen und deshalb sollen in diesem Dokument auch nur diese beiden thematisiert werden.

3.2.1 Low Coupling

Ein gutes Beispiel dafür ist das GuiInterface. Dieses besitzt eine Methode printList(Iterable<?>). Für diese Methode wird nur vorausgesetzt, dass eine Element übergeben werden muss über das iteriert werden kann. Es wäre auch möglich eine List<?> anstatt des Iterable<?> zu nehmen, da eine Liste auch das Iterable Interface implementiert und das in der Methode genutzt wird. Wenn aber eine Liste verwendet wird schränkt man die verwendbaren Klassen ein, obwohl das gar nicht nötig wäre.

```java
public void printList(Iterable<?> list) {
    System.out.println(TextRepository.SELECT_ONE);
    int i = 1;
    for (Object item : list) {
        System.out.println(i++ + ". " + item.toString());
    }
    System.out.println();
}
```

Abbildung 9: Die printList-Methoden Implementierung

Das GuiInterfaces enthält ebenfalls eine rowPrint(RectangularGameBoard)-Methode. RectangularGameBoard ist diesmal aber kein Interface sondern eine normale Klasse. Indem davon Unterklassen erstellt werden, wie Connect6GameBoard, können auch diese für Low Coupling verwendet werden.

```java
public void rowPrint(RectangularGameBoard board, int wantedRow) {
    if (wantedRow >= 0 && wantedRow < board.getCountOfRows()) {
        StringBuilder line = new StringBuilder();
        for (int i = 0; i < board.getCountOfColumns(); i++) {
            line.append(board.getFieldAsString(wantedRow, i)).append(" ");
        }
        line = new StringBuilder(line.substring(0, line.length() - 1));
        printLine(line.toString());
    } else {
        printError("invalid row");
    }
}
```

Abbildung 10: Die rowPrint-Methoden Implementierung

Die rowPrint Methode verwendet folgende drei Methoden der übergebende Klassen.

- getCountOfRows
- getCountOfColumns
- getFieldAsString

Es wäre auch möglich ein Interface zu erstellen, dass diese drei Methoden unterstützt. So wäre es nicht notwendig, dass das übergebende Objekt eine Unterklasse oder selber vom Typ RectangularGameBoard ist. Zum Beispiel könnten dann auch Tabellen unterstützt werden, die gar nichts mit Spielbrettern zu tun haben. Dies ist für diese Spielesammmlung aber nicht interessant, da nur Spielbretter verwendet werden. Somit könnte aber noch mehr Unabhängigkeit erzielt werden.

3.2.2 High Cohesion

Um eine hohe Kohäsion zu erhalten, sollte versucht werden, Klassen oder Programmteile die zusammen arbeiten auch möglichst nahe beieinander zu halten z.B. in Packages oder Modulen. Ebenso sollte jede Klasse nur für eine Aufgabe zuständig sein (s. Single-Responsibility-Principle). Damit lässt sich schneller erkennen in welchem Bereich Änderungen vorgenommen werden müssen und welche andere Klassen davon betroffen sein könnten. Zu Kohäsion zählt aber auch wie gut die Daten und die Methoden einer Klasse zusammen passen und arbeiten.

Um eine hohe Kohäsion zu erreichen werden in diesem Projekt hauptsächlich Packages verwendet. Das Package (gamerules), dass für die Spielregeln verantwortlich ist, zeigt dies ganz gut. In diesem Package liegen die abstrakten Klassen und die Interfaces welche diese implementieren. Darin gibt es weitere Packages für die genauen Spielregeln der einzelnen Spiele (connect6 und connect_four). Sobald etwas an einem Interface oder der abstrakten Klassen geändert wird, kann erkannt werden, dass alle anderen Klassen in diesem Package davon betroffen sein könnten.

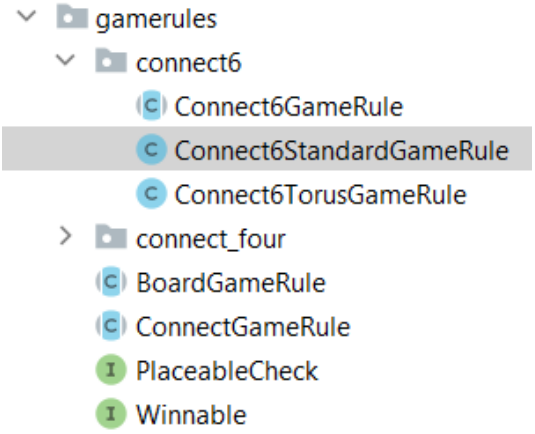

Abbildung 11: Das gamerule-Package

3.3 DRY

DRY steht für „Don't repeat yourself ". So soll möglichst kein Code mehrfach vorhanden sein und damit verhindern, dass der Quellcode unnötig groß und es möglicherweise zwei unterschiedliche Verfahren für die gleiche Sache verwendet wird. Natürlich können grundlegende Befehle nicht nur einmal verwendet werden aber es sollte darauf geachtet werden, das falls zwei Abschnitte eine ähnliche Funktion implementieren und nur minimale Unterschiede aufweisen, diese so anzupassen, dass die Unterschiede durch Parameter oder Verzweigungen generiert werden. Somit muss nur an genau einer Stelle etwas geändert werden und alle Funktionen die darauf aufbauen erhalten diese Änderung und müssen nicht manuell angepasst werden.

4 Clean Architecture

Dieses Programm wurde in zwei Schichten aufgeteilt. Dem Kern-Modul, in dem die ganzen Strukturen liegen, und dem Plugin-Modul, in dem momentan die Ausgabe/-GUI realisiert wird. Um sicherzustellen, dass die Klassen im Kern die Plugins nicht kennen wurden Java Module verwendet und es wurde nur dem Plugin Modul gestattet die Klassen des Kerns zu verwenden und nicht andersherum. Dies ist wichtig, damit die Plugins ausgetauscht werden können ohne den Domain Code anpassen zu müssen.

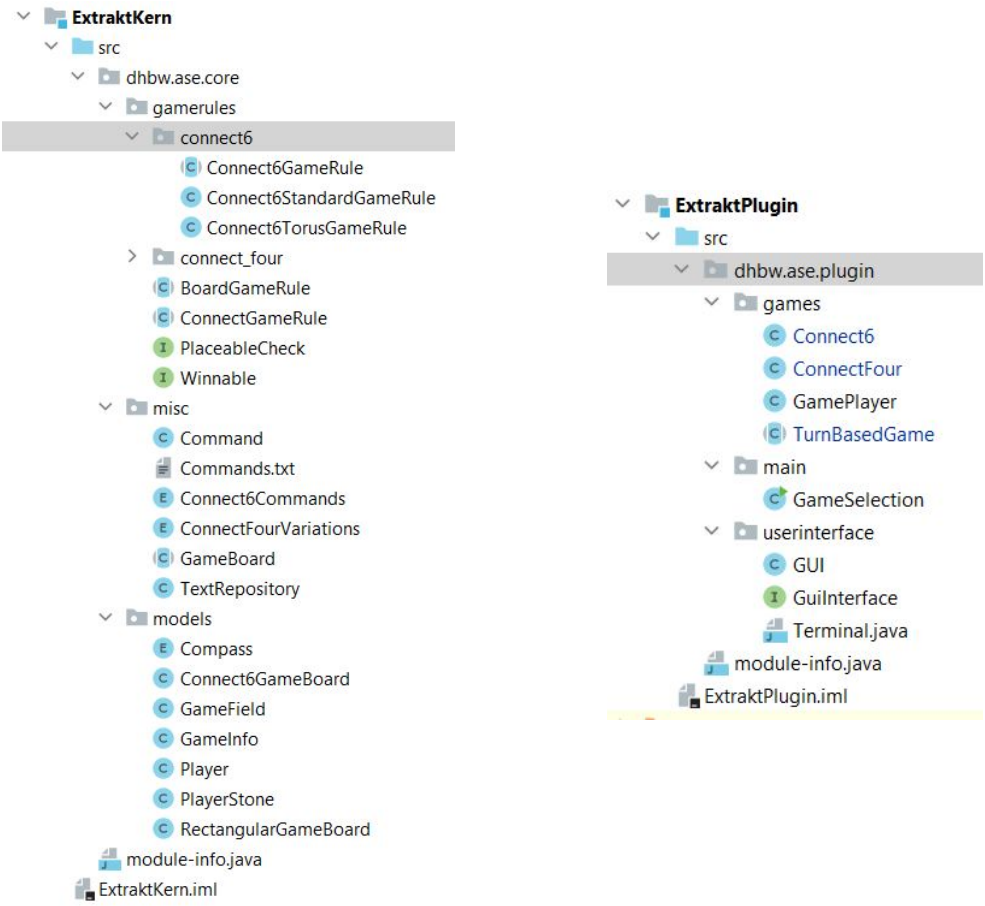

Abbildung 12: Kern- und Plugin-Module

5 Entwurfsmuster

Entwurfsmuster werden verwendet um Konstrukte, die häufig im Code vorkommen, generell zu beschreiben und stellen damit einen allgemeinen Lösungsansatz dar. Über die Jahre wurden dies immer mehr. Im folgenden wird aber nur auf zwei eingegangen.

5.1 Brücken-Entwurfsmuster

Eines der Entwurfsmuster, das eingesetzt wurde ist das Brücken-Muster. Dies hat sich von selber eingeschlichen, als versucht wurde eine hohe Abstraktion zu erreichen. Genau dafür ist dieses Muster auch vorgesehen. Es soll die genaue Implementierung von der Abstraktion trennen. In diesem Fall wurde ein Interface erstellt, das die Benutzeroberfläche verwenden soll (GuiInterface). Dies ist der Implementierer in dem Design-Pattern. Bisher gibt es nur eine Klasse, die ConsoleGUI-Klasse, die dieses Interface implementiert und Ausgaben in der Konsole macht. Es sollte aber in Zukunft auch möglich sein eine weitere Klasse zu erstellen, die dann z.B. eine Graphische Benutzeroberfläche verwendet. Diese Klassen (ConsoleGUI und die folgenden) sind die Konkreten Implementierer des Interfaces. Aufgerufen werden die Implementierungen des GuiInterfaces von der TurnBasedGame-Klasse bzw. den Unterklassen, Connect6 und ConnectFour, da diese eine abstrakte Klasse ist. Dabei ist die TurnBasedGame-Klasse die Abstraktion und Connect6 und ConnectFour sind die speziellen Abstraktionen.

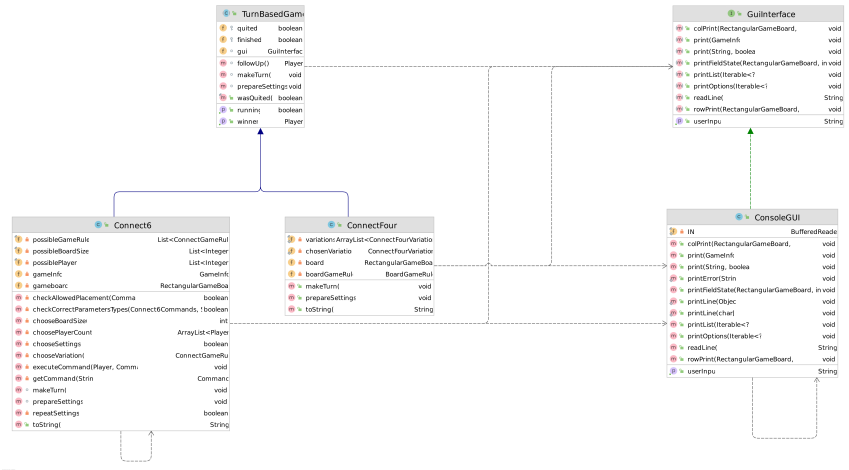

Abbildung 13: Das Brücken-Entwurfsmuster

Damit die Abstraktionen das Interface verwendet können, besitzen sie dieses als Attribut. In TurnBasedGame wird dieses jedoch noch nicht instanziiert, da es dort

nicht verwendet wird. Erst in den Unterklassen werden diese in dem Konstruktor instanziiert. Erst damit ist es möglich die Methoden des Interfaces zu verwenden.

5.2 Schablonen-Entwurfsmuster

Dieses Entwurfsmuster wird verwendet um eine Reihenfolge der auszuführenden Methoden notwendig ist. Häufig wird das verwendet um z.B. erst Speicher zu belegen und später dann wieder freizugeben. Der Entwickler kann dann nur sagen, was zwischen diesen Schritten geschehen soll und kann nicht vergessen den Speicher wieder freizugeben. Ich habe das verwendet, damit jedes Spiel den gleichen Spielverlauf hat. Somit wird zu Beginn immer erst die Einstellungen festgelegt. Danach wird solange das Spiel nicht beendet wurde ein Zug eines Spielers gemacht. Und zum Schluss wird immer noch der Gewinner bekanntgegeben. Dieser Ablauf muss von jedem verwendet werden, sofern von der TurnBasedGame-Klasse geerbt wird und das Muster nicht umgangen wird. Was die Spiele in den einzelnen Methoden machen kann der Entwickler selber entscheiden. Aber mit diesem Muster kann ein Ablauf vorgegeben werden.

```
public final Player play() {
    prepareSettings();

    while (!wasQuited()) {
        makeTurn();
    }

    return followUp();
}
```

Abbildung 14: Die Template-Methode play()

6 Refactoring

Refactoring wird dazu verwendet den Code durch Code Reviews übersichtlicher und/oder lesbarer zu machen. Dabei wird das Gesamtverhalten des bestehendem Codes nicht verändert. Nachdem Refactoring-Schritt sollte die Vorgehensweise des Programms immer noch die gleiche sein wie zuvor. Dabei können sich aber Fehler einschleichen, da möglicherweise etwas vergessen oder übersehen wurde. Diese Stellen, die auf „schlechten "Code hinweisen, werden als Code Smells bezeichnet.

6.1 Duplicated Code

Wie der Name schon sagt, gibt es an mehreren Stellen Code, der für das Gleiche zuständig ist. Dies sollte vermieden werden, da zum einen der Code unnötig größer wird und, falls sich etwas an dieser Stelle ändern soll, muss das an zwei oder mehr Stellen getan werden. Um einen Gewinner nach einem Spielzug zu berechnen werden Spielregeln verwendet.

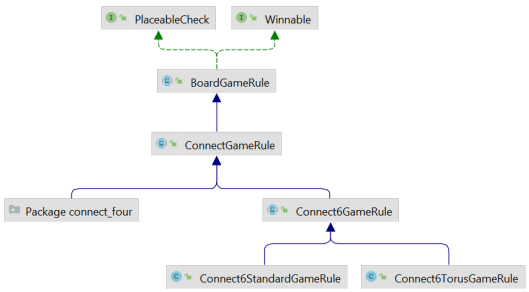

Abbildung 15: Klassendiagramm der Spielregeln

Ursprünglich wurden sowohl in der Connect6StandardGameRule und der Connect6-TorusGameRule berechnet ob ein Sieger vorliegt. Im Grundaufbau machen die beiden Klassen genau das Gleiche. Nur dass bei der Torus-Variante die Felder auch über den Rand hinaus gezählt werden. Zusätzlich wurde bei der Torus-Variante der Algorithmus auch sehr kompliziert programmiert. Aus diesen zwei verschiedene Implementierungen wurde dann eine in der Oberklasse Connect6GameRule. Somit kann diese Implementierung dann auch bei den Connect_four Spielregeln verwendet werden. In ConnectGameRules wurde diese Methode dann wie bei der Connect6StandardGameRule programmiert und somit eine Method-Extraktion angewendet. Es wurde nur eine kleine Anpassung gemacht, sodass das nächste Feld, das betrachtet werden soll, in einer eigenen Methode bestimmt wird. Dies wird dann erst in den beiden Unterklassen definiert. Somit gibt es jetzt nur noch eine Implementierung die einen Gewinner berechnet.

6.2 Shotgun Surgery & Switch-Statement

Dieses Problem tritt auf, wenn eigentlich nur eine Kleinigkeit angepasst werden soll aber dann tatsächlich an vielen Stellen Code angepasst werden muss. In meinem Fall kam noch hinzu, dass es sich um ein Enum gehandelt hat, dass auch in mehreren Switch-Statements verwendet wurde und die möglichen Befehle darstellen sollte. Wenn bisher aber ein neuer Befehl hinzukommen soll, musste an mindestens vier Stellen Code angepasst werden.

```
switch (command.getCommand()) {
    case print -> executePrint();
    case rowprint -> executeRowprint(command)
    case colprint -> executeColprint(command)
    case quit -> executeQuit();
    case reset -> executeReset();
    case place -> executePlace(player, commar
    case state -> executeState(command);
    case help -> executeHelp();
    default -> {}
}
```

```
Connect6Commands command = switch (arrayString[0]) {
    case "print" -> Connect6Commands.print;
    case "rowprint" -> Connect6Commands.rowprint;
    case "colprint" -> Connect6Commands.colprint;
    case "quit" -> Connect6Commands.quit;
    case "reset" -> Connect6Commands.reset;
    case "place" -> Connect6Commands.place;
    case "state" -> Connect6Commands.state;
    case "help" -> Connect6Commands.help;
    default -> throw new IOException("Command not recognized");
};
```

Abbildung 16: Mehrere Swtich-Statements für Befehle

Jeder Befehl braucht folgende Informationen:

- einen Text, der in der Konsole eingegeben werden muss um den Befehl zu adressieren
- einen Enum, um den Befehl unabhängig der Eingabe zu machen z.B. case-insensitiv
- eine Anzahl an korrekten Parametern
- welche Methode angewendet werden soll

Daraus wurde dann eine neue Klasse gemacht, die alle diese Punkte in einem Objekt speichert. Zusätzlich dazu wurde noch eine Klasse erstellt, die auf Anfrage die richtigen Werte zurückgibt und die zuvor verwendeten Switch-Konstrukte ablöst. Damit wurde Replace Conditional with Polymorphism angewendet. Somit muss nur noch an zwei Stellen Änderungen gemacht werden. Zum Einen muss ein neues Enum definiert werden und zum Anderen müssen alle anderen Werte für diesen Befehl definiert werden. Damit sind die Befehle auch direkt im Spiel verwendbar werden.

```
private void initalizeCommandMap() {
    Connect6CommandMap.addCommand( text: "print", Connect6CommandEnum.print,  parameter: 0,
            (ignored, ignored2) -> executePrint());

    Connect6CommandMap.addCommand( text: "rowprint", Connect6CommandEnum.rowprint,  parameter: 1,
            (BiConsumer<Player, Command>) (ignored, command) -> executeRowprint(command));

    Connect6CommandMap.addCommand( text: "colprint", Connect6CommandEnum.colprint,  parameter: 1,
            (BiConsumer<Player, Command>) (ignored, command) -> executeColprint(command));
```

Abbildung 17: Definition der Eigenschaften der Befehle

6.3 Long Method

Dieses Problem taucht an mehreren Stellen im Code auf. Zwar gibt es so gut wie keine Methode, die länger als 20 Zeilen ist aber bei manchen Methoden ist selbst das zu viel und sorgt für Probleme beim Lesen der Klasse. Je nach Person ist die akzeptable Länge auch unterschiedlich. Bei mir sind das ungefähr 10 Zeilen pro Methode. Dies halte ich auch an vielen Stellen ein aber an vielen Stellen eben auch nicht, wie in der Connect6-Klasse.

```
256 @    private ArrayList<Player> choosePlayerCount() {
257          while (true) {
258              try {
259                  gui.print(TextRepository.CHOOSE_PLAYER_COUNT,  emptyEndingLine: false);
260                  gui.printOptions(possiblePlayers);
261                  String input = gui.getUserInput();
262
263                  int playerCount = Integer.parseInt(input);
264                  ArrayList<Player> players = new ArrayList<>();
265                  players.add(new Player());
266
267                  if (possiblePlayers.contains(playerCount)) {
268                      while (players.size() < playerCount) {
269                          players.add(new Player());
270                      }
271                      return players;
272                  } else {
273                      gui.print(TextRepository.INPUT_ERROR_MSG,  emptyEndingLine: false);
274                  }
275              } catch (IOException ignored) {
276                  gui.print(TextRepository.INPUT_ERROR_MSG,  emptyEndingLine: false);
277              }
278          }
279      }
```

Abbildung 18: Beispiel für lange Methoden