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

Erklärung

Wir versichern hiermit, dass wir unsere Studienarbeit mit dem Thema:

Kleine Spielesammlung von 4-Gewinnt Versionen

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben. Wir versichern zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Karlsruhe, 30. April 2022

Ort, Datum Unterschrift

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 dazu soll jetzt auch noch Vier Gewinnt mit unterschiedlichen Variationen/Regeln implementiert werden. Bei einer Variation soll es möglich sein, die Spielsteine nur von oben herunterfallen zu lassen. Bei einer anderen z.B. Drop Four ist es aber auch möglich den untersten Stein einer Spalte zu entfernen. 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

3 Programming Principles

3.1 SOLID

3.1.1 Single-Responsibility-Principle

Das Single-Responsibility-Principle soll dafür sorgen, dass jede Klasse nur eine Aufgabe erfüllt. Damit entstehen viele kleinere, unabhängigere Klassen. Sobald sich bei einer großen Klasse etwas ändert muss möglicherweise vieles auch in anderen Methoden angepasst werden. Wenn es viele kleine Klassen gibt, die nur für eine Sache zuständig sind, ist es nach der kleinen Änderung in einer Klasse getan und es müssen sonst nirgends Anpassungen gemacht werden (je nach Ausprägung der Änderung).

Im Code wurde das zum Beispiel 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 1: Die abstrakte BoardGameRule-Klasse

3.1.2 Open/Closed-Principle

Programme sollten offen für Erweiterungen und geschlossen für Änderungen sein. So sollte im nachhinein keine Änderung mehr an bestehendem Code vorgenommen werden. Wenn die Software um Funktionen erweitert werden soll, soll das nur durch Erweiterung z.B. Unterklassen gemacht werden.

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.

```
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 2: Die Connect6TorusGameRule-Klasse

3.1.3 Liskov-Substitution-Principle

Dieses Prinzip wird eingehalten, wenn für jedes Oberklassenobjekt ein Unterklassenobjekt verwenden werden kann, ohne dass dies zu Fehlern oder unbeabsichtigten Verhalten führt.

Verletzt wurde dieses Prinzip bei den Spielbrettern. 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

Mit diesem Prinzip sollen, ähnlich wie beim Single-Responsibility-Principle, kleine kompakte Einheiten generiert werden. In diesem Fall aber mit Interfaces und nicht mit Klassen. So sollen Klassen immer alle Methoden etc. des Interfaces sinnvoll verwenden ohne diese nur zu implementieren, damit diese implementiert sind. Wenn es dazu kommt sollten lieber mehrere kleine Interfaces erstellt werden. Die Klasse

müsste dann nicht mehr alle Interfaces implementieren, sondern nur noch die, die tatsächlich benötigt werden.

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

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

Um eine geringe Kopplung zu erhalten muss versucht werden die Klassen so unabhängig wie möglich voneinander zu machen. Dies kann zum Beispiel damit erreicht werden, dass keine spezielle Klasse sondern ein Interface als Parameter/Attribut verwendet wird. Dies kann auch mit Oberklassen gemacht werden, mit Interfaces sind die Klassen aber noch unabhängiger voneinander. Damit wird dann nur spezifiziert welche Methoden mit welchen Parametern unterstützt werden müssen. Wie genau die Klasse das dann umsetzt ist egal. So muss zu Beginn nicht gewusst werden welche Klassen genau verwendet werden.

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 werde 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 3: 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 diese auch dafür 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 4: 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 siehe 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.

So existiert in diesem Projekt ein Package (gamerules), dass für die Spielregeln verantwortlich ist. 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). So bald etwas an einem Interface oder der abstrakten Klassen geändert wird, erkennt man, dass alle anderen Klassen in diesem Package davon betroffen sein könnten.

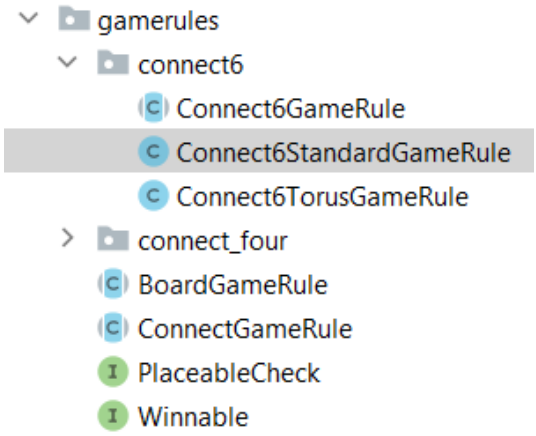

Abbildung 5: Das gamerule-Package

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.

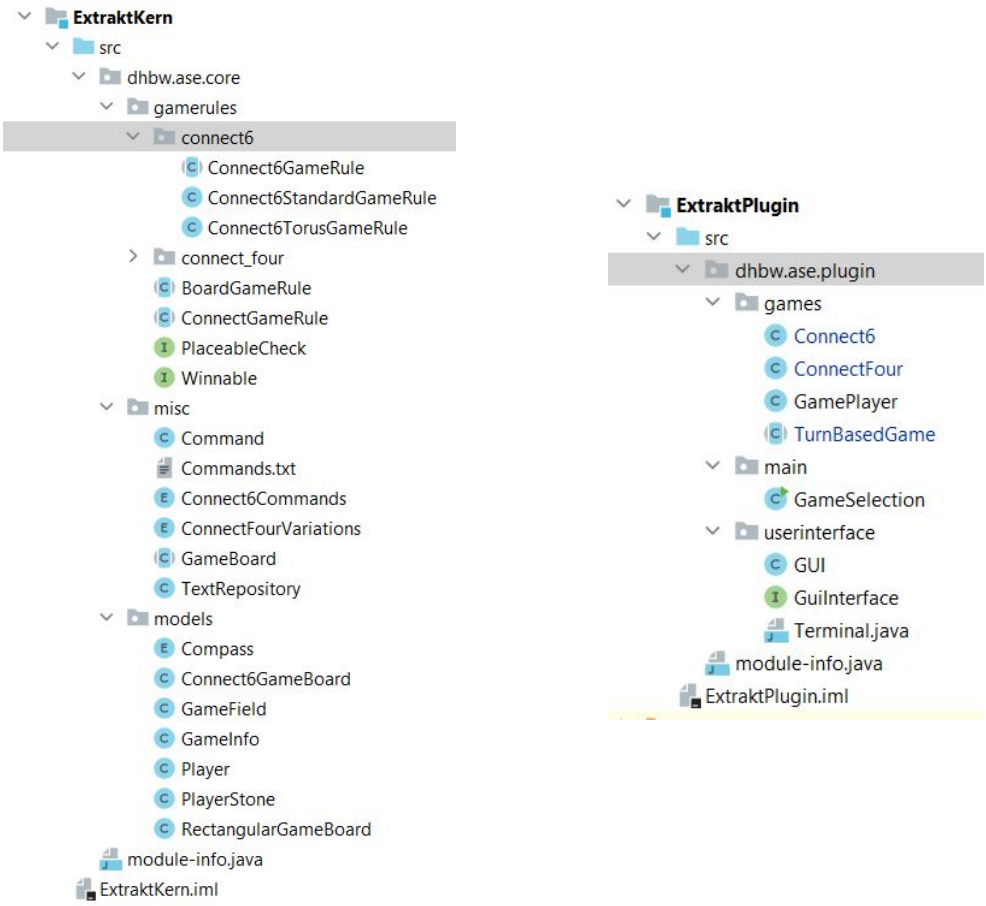

Abbildung 6: Kern- und Plugin-Module

5 Entwurfsmuster evtl. Noch Template/Schablonen pattern

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.

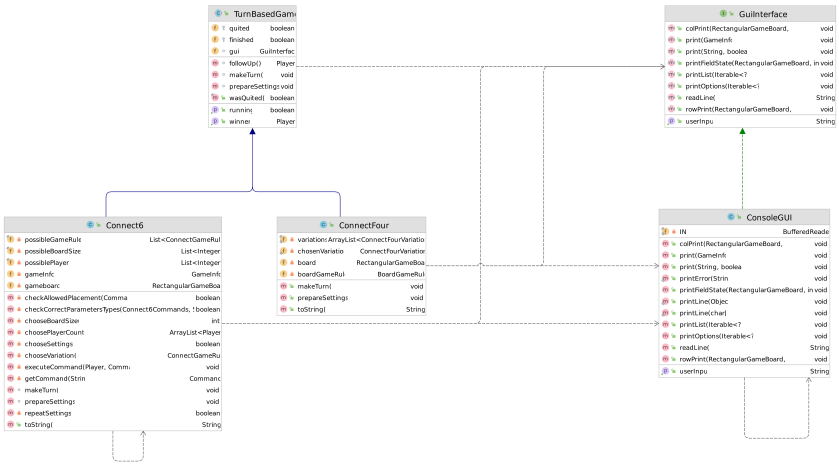

Abbildung 7: Das Brücken-Entwurfsmuster

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.

6 Refactoring