

# Programmentwurf [Textventure]

Name: [Eisenstein, Nicolai]  
Matrikelnummer: [7765690]

Abgabedatum: [7.6.24]

# Kapitel 1: Einführung

## Übersicht über die Applikation

Die Anwendung ist ein interaktives textbasiertes Spiel, das ausschließlich über die Konsole bedient wird. Es bietet ein Abenteuererlebnis, bei dem der Spieler eine virtuelle Karte erkundet, um einen Weg nach Hause zu finden und damit das Spielziel zu erreichen. Während des Erkundens und Navigierens durch die Karte wird der Spieler auf verschiedene Herausforderungen stoßen, darunter der Kampf gegen eine Vielzahl von Gegnern.

## Wie startet man die Applikation?

Um die Anwendung zu starten, importiere den Quellcode in deine bevorzugte integrierte Entwicklungsumgebung (IDE). Nachdem du den Code geöffnet hast, führe die Datei Main.java aus, um das Spiel zu starten.

## Wie testet man die Applikation?

Zum Testen der Anwendung alle Test unter src/test/java ausführen..

# Kapitel 2: Clean Architecture

## Was ist Clean Architecture?

Clean Architecture ist ein Architekturmuster der Software Entwicklung. Es zielt darauf ab übersichtlichen, flexiblen und wartbaren Code zu produzieren. Die grundlegenden Prinzipien dabei sind:

- Abhängigkeitsregel: Sollten immer von außen nach innen zeigen und für die inneren Schichten unbekannt sein.
- Interessentrennung: Trennung der technischen Details von den Geschäftsregeln.
- Testbarkeit: Durch Interessentrennung erleichtert.

So eine Architektur besteht aus verschiedenen Schichten, welche eigene Verantwortlichkeiten haben. Die Mitte des ganzen ist der Kern der Anwendung, welcher die Geschäftsregeln und Anwendungslogik enthält.

## Analyse der Dependency Rule

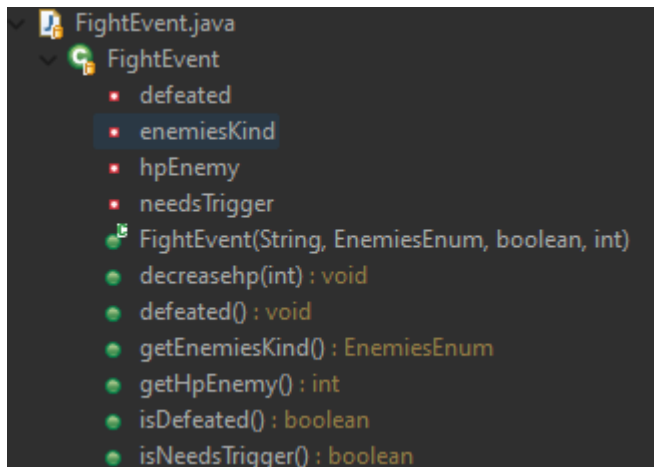
*[(1 Klasse, die die Dependency Rule einhält und eine Klasse, die die Dependency Rule verletzt); jeweils UML der Klasse und Analyse der Abhängigkeiten in beide Richtungen (d.h., von wem hängt die Klasse ab und wer hängt von der Klasse ab) in Bezug auf die Dependency Rule]*

## Positiv-Beispiel: Dependency Rule

Die Klasse FightEvent.

Die Klasse `FightEvent` erbt von der Klasse `BasisEvent`, was bedeutet, dass sie die allgemeinen Eigenschaften und Methoden von `BasisEvent` erbt und somit von einer direkten Abhängigkeit profitiert. Dadurch bleibt die Klasse `FightEvent` weitgehend unabhängig von anderen Klassen im System und fördert die Kohäsion und lockere Kopplung im Code.

UML:



Analyse der Abhängigkeiten:

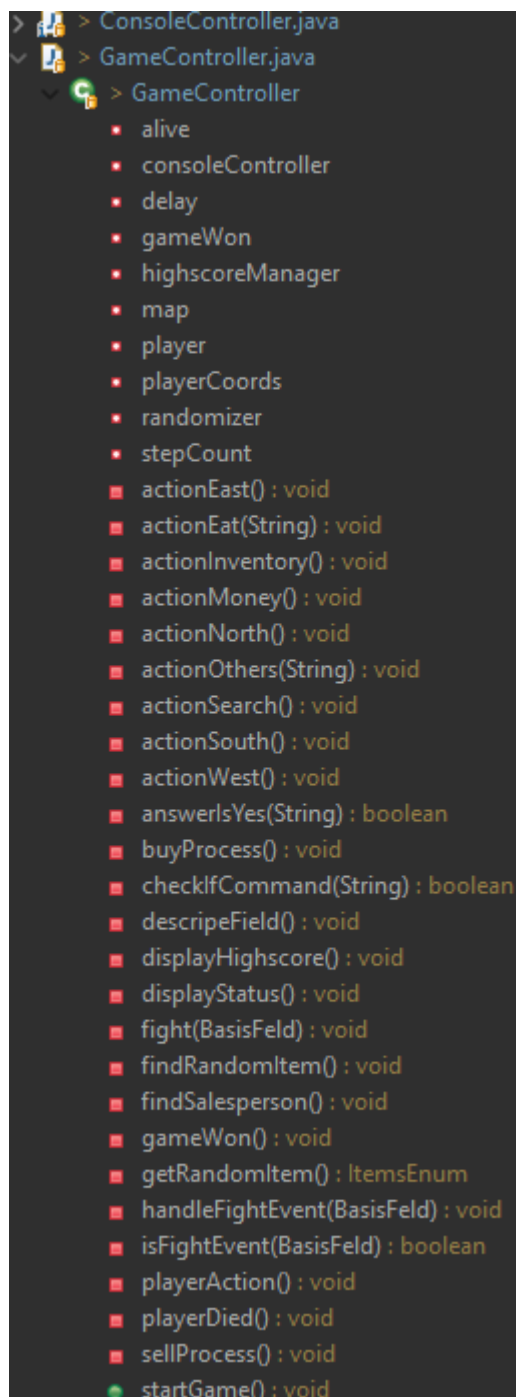
- `EnemiesEnum` (Domain): Die Klasse `FightEvent` verwendet `EnemiesEnum`, um den Typ des Feindes im Kampf zu spezifizieren. Diese Abhängigkeit ist akzeptabel gemäß der Dependency Rule, da `EnemiesEnum` in einer inneren Schicht (Domain) liegt.
- `BasisEvent` (Domain): `FightEvent` erbt von der abstrakten Klasse `BasisEvent` aus der Domänenschicht. Diese Abhängigkeit ist ebenfalls akzeptabel, da `BasisEvent` in einer inneren Schicht liegt.
- `Lombok` (Framework): `FightEvent` verwendet die Lombok-Annotation `@Getter`, um automatisch Getter-Methoden für die Felder zu generieren. Da `Lombok` ein Framework ist und nicht Teil der Anwendungsschicht ist, verstößt diese Abhängigkeit nicht gegen die Dependency Rule.

## ***Negativ-Beispiel: Dependency Rule***

Game-Controller

Der `GameController` hat eine direkte Abhängigkeit von der `ConsoleController`-Klasse. Idealerweise sollte der `GameController` keine direkte Abhängigkeit von einer spezifischen Implementierung einer Controller-Klasse haben, sondern stattdessen mit einem abstrakten Interface oder einer abstrakten Klasse interagieren, um die Abhängigkeit umzukehren.

UML:



Analyse der Abhängigkeiten:

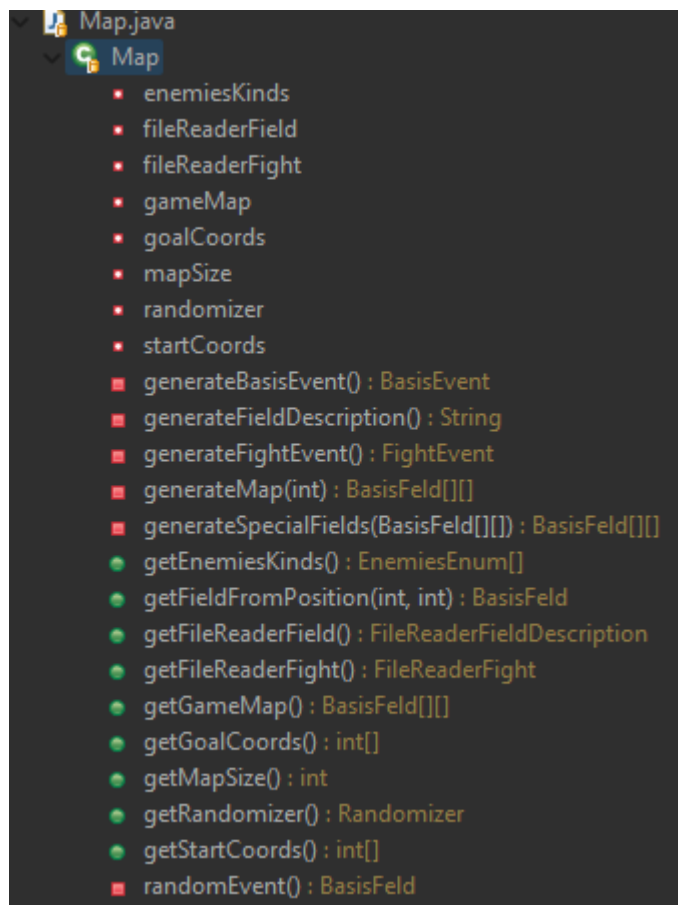
- ConsoleController (Application): Der GameController hat eine direkte Abhängigkeit von der ConsoleController-Klasse, die in der Anwendungsschicht liegt. Diese direkte Abhängigkeit könnte als Verletzung der Dependency Rule angesehen werden, da der GameController idealerweise nicht an eine spezifische Implementierung eines Controllers gebunden sein sollte, sondern mit einem abstrakten Interface oder einer abstrakten Klasse interagieren sollte.
- Player (Domain): Der GameController hat eine direkte Abhängigkeit von der Player-Klasse, die in der Domänenschicht liegt. Diese Abhängigkeit ist akzeptabel gemäß der Dependency Rule, da Player in einer inneren Schicht liegt.

- Map (Application): Der GameController hat eine direkte Abhängigkeit von der Map-Klasse, die in der Anwendungsschicht liegt. Diese direkte Abhängigkeit könnte als Verletzung der Dependency Rule angesehen werden, da der GameController idealerweise nicht an eine spezifische Implementierung der Karte gebunden sein sollte, sondern mit einem abstrakten Interface oder einer abstrakten Klasse interagieren sollte.
- Randomizer (Application): Der GameController verwendet die Randomizer-Klasse, um zufällige Ereignisse im Spiel zu generieren. Obwohl die Randomizer-Klasse nicht Teil eines externen Frameworks ist und speziell für die Anforderungen des GameControllers entwickelt wurde, kann diese direkte Abhängigkeit als akzeptabel betrachtet werden, da sie zur Unterstützung der spezifischen Funktionalität des GameController dient.
- HighscoreManager (Application): Der GameController hat eine direkte Abhängigkeit von der HighscoreManager-Klasse, die in der Anwendungsschicht liegt. Diese direkte Abhängigkeit könnte als Verletzung der Dependency Rule angesehen werden, da der GameController idealerweise nicht an eine spezifische Implementierung des Highscore-Managers gebunden sein sollte, sondern mit einem abstrakten Interface oder einer abstrakten Klasse interagieren sollte.

## Analyse der Schichten

### **Schicht: [Anwendung]**

**Map:** Verwalten und Generieren der Spielkarte. Diese Klasse erstellt und verwaltet das Layout der Spielumgebung.



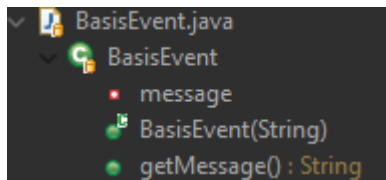
Dabei hat die Klasse Abhängigkeiten zum Randomizer (1-1), BasisFeld(1-n), StartFeld(1-1), Endfeld(1-1) sowie wie StartEvent(1-1), EndEvent(1-1), BasisEvent(1-n) und zwei FileReadern(jeweils 1-1).

Die **Map**-Klasse verwaltet die Spielkarte und deren Struktur. Sie organisiert die Felder auf der Karte und ermöglicht es, Felder basierend auf den Spielerbewegungen zu finden und zu interagieren. Diese Funktionen sind entscheidend für den Ablauf und die Steuerung des Spiels, was typisch für die Application-Schicht ist.

Die **Map**-Klasse wird in der **GameController**-Klasse verwendet, um die aktuellen Positionen und Ereignisse anzuzeigen oder darauf zu reagieren, was ein zentraler Aspekt der Spielsteuerung ist. Der **GameController** ist klar in der Application-Schicht angesiedelt und ruft Methoden der **Map**-Klasse auf, um den Spielfortschritt zu verwalten.

### **Schicht: [Domain]**

**BasisEvent**: Basisklasse für alle Ereignisse im Spiel, die grundlegende Eigenschaften und Verhalten definiert.



Die **BasisEvent**-Klasse gehört zur **Domänenschicht**, weil sie wesentliche Merkmale und Logik der Spielwelt und ihrer Mechaniken abbildet. Sie ist eine abstrakte Klasse, die die grundlegenden Eigenschaften und Verhaltensweisen von Ereignissen im Spiel definiert. Sie modelliert Konzepte wie Begegnungen oder Kämpfe, die zentrale Bestandteile der Spielmechanik sind

## **Kapitel 3: SOLID**

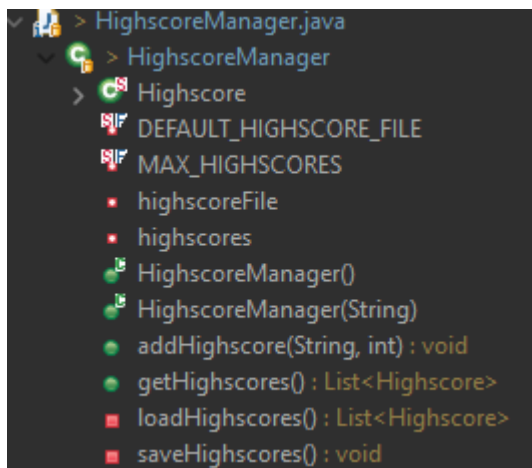
### **Analyse Single-Responsibility-Principle (SRP)**

#### **Positiv-Beispiel**

HighscoreManager

**HighscoreManager** hat eine klar definierte Aufgabe: Verwaltung der Highscores. Er ist dafür verantwortlich, Highscores zu speichern, neue Highscores hinzuzufügen und die Liste der Highscores zurückzugeben.

Alle Methoden und Felder der Klasse sind darauf fokussiert, diese Aufgabe zu erfüllen. Es gibt keine andere Logik oder Funktionalität in dieser Klasse, die nicht direkt mit der Verwaltung von Highscores zu tun hat.



## ***Negativ-Beispiel***

Game-Controller

Diese Klasse hat zahlreiche Verantwortlichkeiten:

- Verwaltung des Spielzustands
- Handhabung der Eingaben und Aktionen
- Verwaltung der Interaktion mit der Spielkarte
- Verarbeitung der Kämpfe und Ereignisse



Dabei hat der GameController zich Abhängigkeiten.

Eine mögliche Lösung wäre:

- Auslagern der Kämpfe, Interaktionen mit dem Spielfeld in eigene handler-Klasse
- Auslagern der Handhabung der Eingaben und Aktionen

## Analyse Open-Closed-Principle (OCP)

### ***Positiv-Beispiel***

BasisFeld



### Offen für Erweiterung:

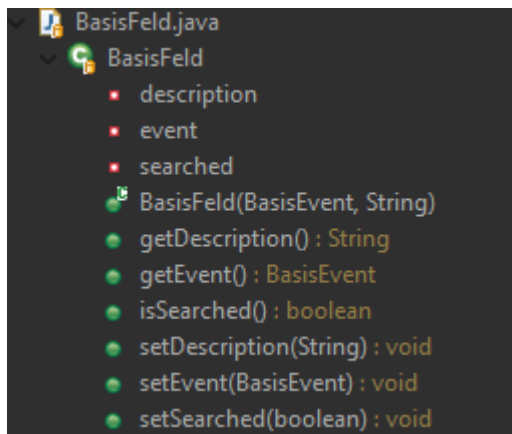
- Die `BasisFeld`-Klasse ist offen für Erweiterungen, da neue Feldtypen einfach durch das Erstellen neuer Unterklassen hinzugefügt werden können. Diese Unterklassen können spezifische Implementierungen von `getDescription()` und anderen Methoden bereitstellen.
- Zum Beispiel könnte ein neues Feld `TrapFeld` hinzugefügt werden, das zusätzliche Logik für Fallen enthält, ohne die bestehenden `BasisFeld`- oder `StartFeld`-Klassen ändern zu müssen.

### Geschlossen für Änderungen:

- Die `BasisFeld`-Klasse und ihre existierenden Unterklassen müssen nicht geändert werden, um neue Feldtypen oder neue Funktionalitäten hinzuzufügen. Das bedeutet, dass der bestehende Code stabil bleibt und nicht durch neue Anforderungen beeinträchtigt wird.
- Wenn neue Felder oder Ereignisse hinzugefügt werden müssen, bleibt der bestehende Code unverändert. Die bestehenden Klassen können weiter genutzt und getestet werden, ohne dass Änderungen vorgenommen werden müssen.

### Einfache Erweiterbarkeit:

- Das Hinzufügen neuer Feldtypen ist einfach und intuitiv, da es durch das Erstellen neuer Unterklassen geschieht. Diese neuen Klassen erben die allgemeinen Eigenschaften und Methoden von `BasisFeld`, was die Implementierung neuer Logik erleichtert.
- Diese Struktur fördert eine modulare und gut organisierte Codebasis, in der neue Funktionen durch Ergänzung und nicht durch Änderung eingeführt werden.



### Negativ-Beispiel

Das Open/Closed Principle (OCP) besagt, dass Softwarekomponenten so entworfen werden sollten, dass sie erweiterbar sind, ohne ihren bestehenden Code zu ändern. In deinem Projekt gibt es Beispiele, die gegen dieses Prinzip verstoßen. Ein solches Beispiel ist die Klasse `ConsoleController`. Diese Klasse muss verändert werden, wenn man die Art und Weise ändern will, wie Nachrichten angezeigt werden oder wie Eingaben vom Benutzer verarbeitet werden. Solche Änderungen erfordern eine direkte Bearbeitung des bestehenden Codes, anstatt durch Erweiterungen neue Funktionalitäten hinzuzufügen.

### Geschlossen für Erweiterung:

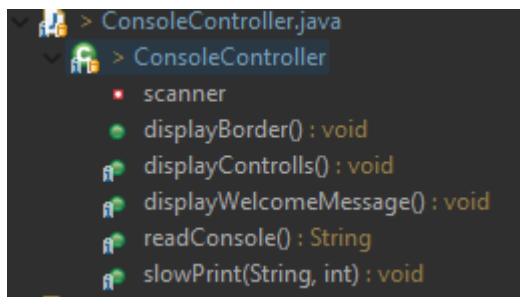
- Wenn die Klasse um neue Arten der Ausgabe erweitert werden soll (zum Beispiel eine grafische Oberfläche anstelle der Konsole oder zusätzliche Formatierungen), müsste der bestehende Code verändert werden. Es gibt keine Möglichkeit, die Ausgabe zu erweitern, ohne die Methoden `slowPrint`, `displayControlls`, `displayBorder` und `displayWelcomeMessage` zu ändern.
- Jede Änderung der Art und Weise, wie Nachrichten gedruckt oder Eingaben gelesen werden, erfordert eine Modifikation des bestehenden Codes. Das bedeutet, dass die Klasse nicht für Erweiterungen ohne Modifikation offen ist.

### Fehlen von Abstraktion:

- Es gibt keine Abstraktion oder Schnittstelle, die es ermöglicht, das Verhalten der Methoden `slowPrint`, `readConsole`, `displayControlls` und `displayWelcomeMessage` zu ändern, ohne den `ConsoleController` selbst zu modifizieren.
- Zum Beispiel könnte die Implementierung von `slowPrint` durch eine flexiblere Struktur ersetzt werden, wie eine Schnittstelle `Printer` mit einer Methode `print(String message, int delay)`, die dann von verschiedenen Klassen implementiert werden könnte, die unterschiedliche Druckverhalten bieten (z.B. `ConsolePrinter`, `FilePrinter`, `GraphicPrinter`).

### Keine Erweiterungsmöglichkeiten:

- Der `ConsoleController` ist stark gebunden an spezifische Implementierungen, wie das Lesen von Dateien aus einem bestimmten Pfad oder das Verwenden eines Scanners für die Konsoleneingabe. Dies macht es schwierig, das Verhalten der Klasse zu erweitern oder zu verändern, ohne den Quellcode selbst zu modifizieren.



### • Verbesserung:

- Einführung von Schnittstellen oder abstrakten Basisklassen, um die verschiedenen Aufgaben wie Drucken und Lesen zu abstrahieren. Zum Beispiel könnte eine `MessageDisplay`-Schnittstelle verwendet werden, um das Drucken von Nachrichten zu kapseln.
- Anstatt die Implementierungsdetails in der `ConsoleController`-Klasse festzulegen, könnten verschiedene Implementierungen von Schnittstellen erstellt werden, die bei Bedarf ausgetauscht werden können. Dies würde ermöglichen, neue Arten der Nachrichtenanzeige hinzuzufügen, ohne den bestehenden Code zu ändern.

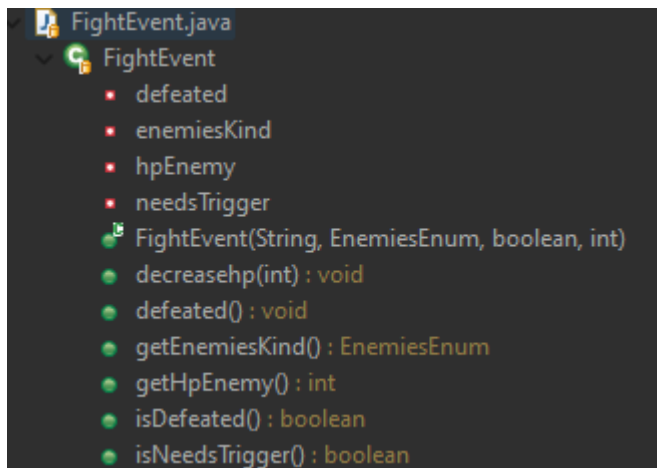
- Anstatt konkrete Klassen innerhalb des `ConsoleController` zu instanziiieren, könnte man Abhängigkeiten von außen injizieren, was es einfacher macht, diese durch verschiedene Implementierungen zu ersetzen und die Klasse erweiterbar zu machen.

## Dependency-Inversion-Principle (DIP)

### *Positiv-Beispiel*

#### FightEvent

Die `FightEvent`-Klasse nutzt Vererbung von `BasisEvent` und dabei wird das erwartete Verhalten nicht verändert. Sie erweitert lediglich die Funktionalität um spezifische Eigenschaften und Methoden, die für einen Kampfevent relevant sind, wie z. B. das Festlegen des Feindtyps und die Behandlung der Feind-HP. Die Verwendung einer `FightEvent`-Instanz anstelle einer `BasisEvent`-Instanz beeinträchtigt nicht das erwartete Verhalten, was dem LSP entspricht.



### *Negativ-Beispiel*

Ein negativ Beispiel wäre wenn die Klasse `FightEvent` die `getMessage`-Methode der `BasisEvent` Klasse überschreiben würde um eine andere Nachricht zurückzugeben. Diese Änderung des Verhaltens könnte unerwartete Auswirkungen haben, wenn Code vorhanden ist, der `EndEvent`-Objekte verwendet und auf das erwartete Verhalten basiert.

## Kapitel 4: Weitere Prinzipien

### Analyse GRASP: Geringe Kopplung

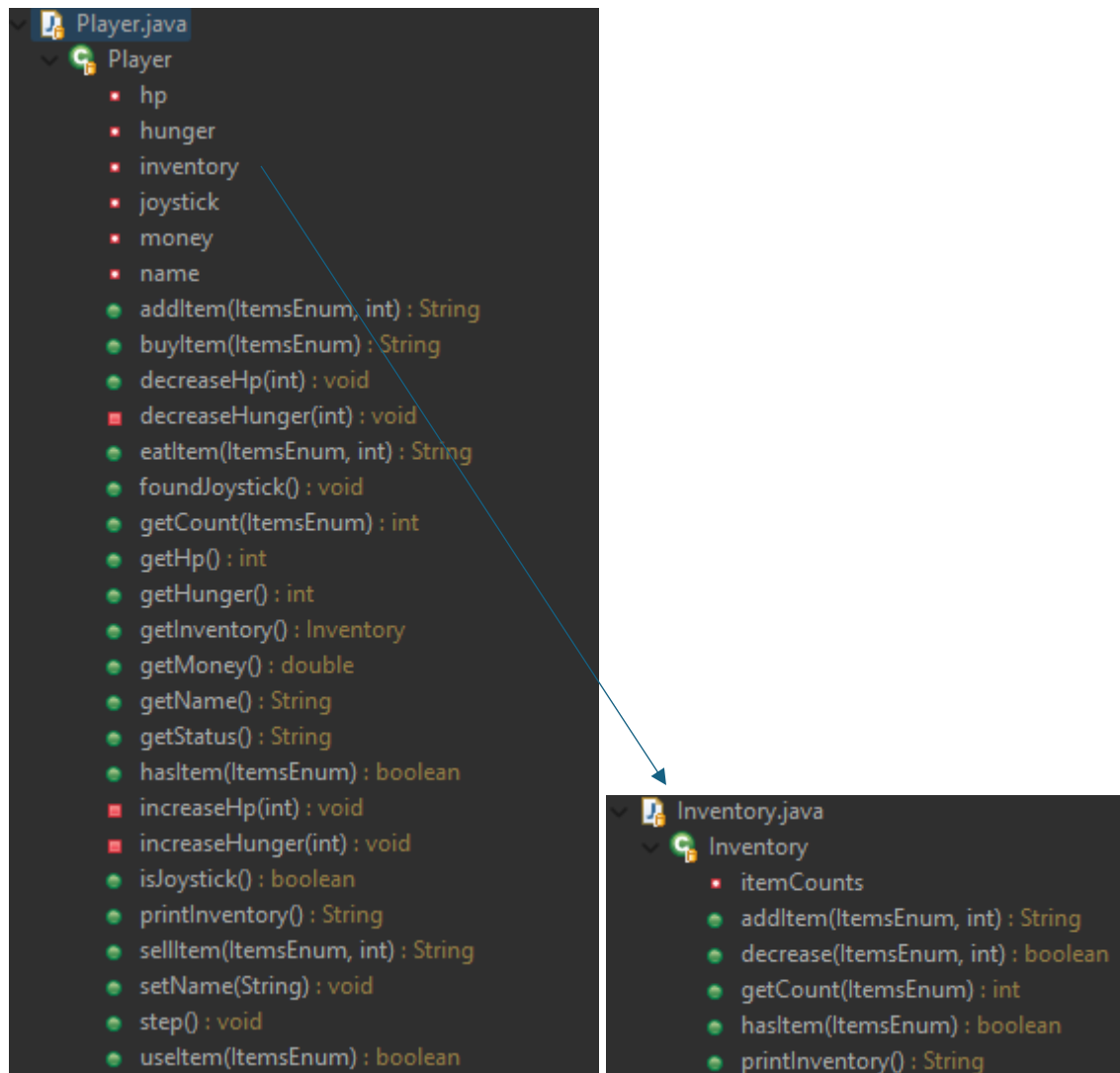
#### *Positiv-Beispiel*

#### Player

Die Klasse `Player` übernimmt klar definierte Aufgaben, indem sie die Verwaltung eines Spielers im Spiel übernimmt. Sie stellt Methoden bereit, um die Gesundheitspunkte (HP) des Spielers zu verringern und das Geld des Spielers zu erhöhen. Durch diese Fokussierung auf eine einzige Verantwortlichkeit ist

die Klasse gut kohäsiv, da alle ihre Methoden und Attribute dem Zweck dienen, einen Spieler zu repräsentieren und zu verwalten. Diese klare Strukturierung zeigt das Prinzip des "Low Coupling" (niedrige Kopplung) und "High Cohesion" (hohe Kohäsion).

Die Klasse `Player` wird ausschließlich vom `GameController` verwendet und interagiert selbst nur mit der `Inventory`-Klasse. Diese Beschränkung der Abhängigkeiten trägt dazu bei, die Kopplung zwischen den Klassen zu minimieren und die Modularität des Codes zu verbessern.



Um diese Kopplung aufzulösen könnte ein Interface verwendet werden. Dadurch können Klassen austauschbar werden, ohne die Implementierung zu ändern.

## Negativ-Beispiel

In der `GameController`-Klasse ist eine zu hohe Kopplung mit anderen Klassen wie `Map`, `Player` und `ConsoleController` vorhanden. Wenn die `GameController`-Klasse für zu viele unterschiedliche Aufgaben zuständig ist, wie beispielsweise das Verwalten des Spielzustands, das Aktualisieren der Spielkarte, die Interaktion mit dem Spieler und die Anzeige von Informationen auf der Konsole, würde dies gegen das Prinzip der klaren Verantwortlichkeiten verstoßen. Dies würde zu einer unklaren Zuweisung von Verantwortlichkeiten führen und die Wartbarkeit des Codes beeinträchtigen.

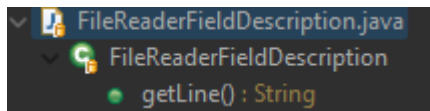
Um dieses Problem zu lösen, könnten Sie die Verantwortlichkeiten in kleinere und spezifischere Klassen aufteilen, um eine bessere Kohäsion und eine geringere Kopplung zu erreichen. Zum Beispiel könnten Sie separate Klassen für die Verwaltung des Spielzustands, die Aktualisierung der Spielkarte, die Spielerinteraktion und die Konsolenausgabe erstellen, um die Verantwortlichkeiten klarer zuzuweisen und die Codequalität zu verbessern.

## Analyse GRASP: Hohe Kohäsion

### FileReaderDescription

Die `FileReaderFieldDescription`-Klasse hat eine klare Verantwortung, nämlich das Lesen von Zeilen aus einer Datei mit Feldbeschreibungen. Sie hat eine Methode `getLine()`, die eine zufällige Zeile aus der Datei zurückgibt. Die Klasse ist gut kohäsiv, da alle Methoden und Attribute dem Zweck dienen, Feldbeschreibungen aus einer Datei zu lesen und eine zufällige Zeile zurückzugeben.

Ein Beispiel für hohe Kohäsion in dieser Klasse ist die Verwendung des `BufferedReader` und die Verarbeitung der Datei nur innerhalb der `getLine()`-Methode. Alle Methoden und Attribute sind eng miteinander verbunden und dienen einem klaren Zweck: das Lesen von Zeilen aus der Datei. Dies zeigt, dass die Klasse gut strukturiert ist und die Prinzipien von hoher Kohäsion und niedriger Kopplung beachtet werden.



## Don't Repeat Yourself (DRY)

<https://github.com/N-Eisen/ASWE/commit/778994fbf8769459695b8f1734dd84de0f43fe9a>

Es gab keine größeres Aufkommen von dupliziertem Code.

Um die Wartbarkeit und Lesbarkeit des Codes zu verbessern, wurde die Methode `answerIsYes` extrahiert. Diese Methode überprüft, ob die Antwort des Spielers "ja" ist und wird an verschiedenen Stellen im Code verwendet, wo diese Überprüfung erforderlich ist.

Durch Extrahieren dieser Methode wird der Code sauberer und leichter zu verstehen. Anstatt die Bedingung für "ja" an verschiedenen Stellen im Code zu wiederholen, wird sie jetzt an einer einzigen Stelle definiert, was die Wartbarkeit erhöht. Wenn sich die Bedingung für "ja" in Zukunft ändern sollte, muss sie nur an einer Stelle im Code aktualisiert werden.

Diese Änderung hat keine Auswirkungen auf das Verhalten des Codes. Sie verbessert jedoch die Struktur und Lesbarkeit des Codes, indem sie Duplikate beseitigt und die Bedeutung der Bedingung klarer macht.

```
private boolean answerIsYes(String answer) {  
    return answer.equals("yes");  
}
```

statt Mehrfach

```
answer.equals("yes");
```

## Kapitel 5: Unit Tests

### 10 Unit Tests

*[Nennung von 10 Unit-Tests und Beschreibung, was getestet wird]*

Unit Test	Beschreibung
testRandomizerInteger# testGetRandomIntFromRange	Testet ob der Randomizer eine zufällige Zahl in einem gegebenen Zahlenbereich zurückgibt. Es werden 100 Zahlen Generiert.
testRandomizerInteger# testGetBoolean50	Testet ob der Randomizer bei einer 50% Chance auf True, auch ca 50% True zurückgibt. Wird 1000 mal getestet mit einer Toleranz.
testRandomizerInteger# testGetBoolean99	Testet ob der Randomizer bei einer 99% Chance auf True, auch ca 99% True zurückgibt. Wird 1000 mal getestet mit einer Toleranz.
testPlayer# testEatNonEatableItem	Testet ob der Spieler nicht essbare Items auch nicht essen kann.
testPlayer# testUseItem	Testet ob der Player auch ein Item verwenden kann.
testPlayer# testBuyItemWithoutMoney	Testet ob der Spieler ein Item kaufen kann, wenn er zu wenig Geld dafür hat.
testInventory# testAddItem	Testet ob das Inventory auch Items hinzufügen kann.
testInventory# testPrintInventory_FlexibleComparison	Testet ob die print Methode des Inventorys auch das richtige Zurückliefert. Egal welche Reihenfolge, da es eine Map ist.
testConsoleController# testSlowPrint	Testet ob die Methode slowPrint auch das Richtige als Konsolenausgabe ausgibt.
testHighscoreManager# testGetHighscores	Testet mit einer testDatei ob der Highscoremanager auch die Daten lesen und zurückgeben kann.

### ATRIP: Automatic

Durch die Verwendung von JUnit und der Eclipse IDE können alle Tests mit einem einfachen Mausklick ausgeführt werden. Eclipse kümmert sich um den Startprozess und führt automatisch alle Tests aus, ohne dass sie manuell gestartet werden müssen.

Innerhalb der Testfälle werden Assertions verwendet, um sicherzustellen, dass die erwarteten Ergebnisse mit den tatsächlichen Ergebnissen übereinstimmen. Diese Assertions sind entscheidend für die Fehlererkennung, da sie anzeigen, wenn ein Test fehlschlägt und somit helfen, potenzielle Probleme im Code frühzeitig zu identifizieren.

## ATRIP: Thorough

Positiv:

testPrintInventory\_FlexibleComparison

```
@Test
public void testPrintInventory_FlexibleComparison() {
    inventory.addItem(ItemsEnum.BERRY, 3);
    inventory.addItem(ItemsEnum.BULLET, 2);

    String expectedOutput = "Inventory:\n" + "Berry: 3 | Price: $0.5 | Eatable: Yes | Hunger Value: 5\n"
        + "Bullet: 2 | Price: $0.5 | Eatable: No";

    String inventoryOutput = inventory.printInventory().trim();
    String[] expectedLines = expectedOutput.trim().split("\n");
    String[] actualLines = inventoryOutput.split("\n");

    assertEquals(expectedLines.length, actualLines.length);

    for (String expectedLine : expectedLines) {
        boolean found = false;
        for (String actualLine : actualLines) {
            if (expectedLine.trim().equals(actualLine.trim())) {
                found = true;
                break;
            }
        }
        assertTrue("Die erwartete Zeile '" + expectedLine + "' wurde nicht in der Ausgabe gefunden.", found);
    }
}
```

### Analyse:

Dieser Test überprüft die `printInventory()`-Methode gründlich, indem er jede einzelne Zeile der erwarteten Ausgabe mit der tatsächlichen Ausgabe vergleicht. Zuerst werden die Artikel zur Inventarliste hinzugefügt, dann wird die erwartete Ausgabe definiert und mit der tatsächlichen Ausgabe verglichen. Dabei wird überprüft, ob die Anzahl der Zeilen in beiden Ausgaben übereinstimmt. Anschließend wird jede Zeile der erwarteten Ausgabe mit jeder Zeile der tatsächlichen Ausgabe verglichen, um sicherzustellen, dass alle erwarteten Zeilen in der tatsächlichen Ausgabe vorhanden sind.

### Begründung:

Dieser Test ist ein gutes Beispiel für das Prinzip der Gründlichkeit (Thorough) in der Testfallerstellung. Durch die detaillierte Überprüfung jeder einzelnen Zeile der erwarteten Ausgabe wird sichergestellt, dass kein Detail übersehen wird. Die Verwendung einer flexiblen Vergleichsmethode macht den Test widerstandsfähig gegenüber kleinen Änderungen in der Ausgabe und bietet eine feingranulare Überprüfung der Funktionalität der `printInventory()`-Methode.

Negativ:

testGetFieldFromPosition

```

@Test
public void testGetFieldFromPosition() {
    Map map = new Map();
    int[] startCoords = map.getStartCoords();

    BasisFeld startField = map.getFieldFromPosition(startCoords[0], startCoords[1]);
    assertNotNull(startField);
    assertEquals(StartFeld.class, startField.getClass());
}

```

### Analyse:

Dieser Test überprüft die `getFieldFromPosition()`-Methode des `Map`-Objekts. Jedoch nur auf korrekte Eingabekoordinaten, nicht auf ungültige.

### Begründung:

Dieser Test ist mangelhaft, da er nur positive Szenarien abdeckt und nicht die Reaktion der `getFieldFromPosition()`-Methode auf ungültige Eingaben testet. Es ist wichtig, auch ungültige Eingaben zu überprüfen, um sicherzustellen, dass die Methode robust gegenüber fehlerhaften oder unerwarteten Eingaben ist. Durch das Testen ungültiger Eingaben können potenzielle Fehlerquellen entdeckt und behoben werden, was zur Verbesserung der Zuverlässigkeit und Robustheit der Anwendung beiträgt.

## ATRIP: Professional

Positiv:

```

@Test
public void testStartAndEndFields() {
    Map map = new Map();
    BasisFeld[][] generatedMap = map.getGameMap();
    int[] startCoords = map.getStartCoords();
    int[] goalCoords = map.getGoalCoords();

    assertNotNull(generatedMap[startCoords[0]][startCoords[1]]);
    assertNotNull(generatedMap[goalCoords[0]][goalCoords[1]]);
}

```

### Analyse:

Dieser Test überprüft, ob das Startfeld und das Endfeld auf der generierten Karte nicht null sind.

### Begründung:

Dieser Test ist professionell, weil er sicherstellt, dass die wichtigsten Elemente der Karte, nämlich das Startfeld und das Endfeld, ordnungsgemäß initialisiert und nicht null sind. Durch diese Überprüfung



wird sichergestellt, dass die grundlegende Funktionalität der Karte, insbesondere der Start- und Endpunkte, vorhanden ist, was für das Funktionieren des Spiels entscheidend ist. Das Testen dieser essenziellen Eigenschaften trägt zur Sicherstellung der Qualität und Zuverlässigkeit des Codes bei.

Negativ:

testPrintInventory\_FlexibleComparison

```
@Test
public void testPrintInventory_FlexibleComparison() {
    inventory.addItem(ItemsEnum.BERRY, 3);
    inventory.addItem(ItemsEnum.BULLET, 2);

    String expectedOutput = "Inventory:\n" + "Berry: 3 | Price: $0.5 | Eatable: Yes | Hunger Value: 5\n"
        + "Bullet: 2 | Price: $0.5 | Eatable: No";

    String inventoryOutput = inventory.printInventory().trim();
    String[] expectedLines = expectedOutput.trim().split("\n");
    String[] actualLines = inventoryOutput.split("\n");

    assertEquals(expectedLines.length, actualLines.length);

    for (String expectedLine : expectedLines) {
        boolean found = false;
        for (String actualLine : actualLines) {
            if (expectedLine.trim().equals(actualLine.trim())) {
                found = true;
                break;
            }
        }
        assertTrue("Die erwartete Zeile '" + expectedLine + "' wurde nicht in der Ausgabe gefunden.", found);
    }
}
```

### Analyse:

Dieser Test vergleicht die erwartete Ausgabe des Inventars mit der tatsächlichen Ausgabe, indem er die Zeilen der Ausgaben einzeln vergleicht. Dies führt zu einer starren Testimplementierung, die anfällig für fehleranfällige und wartungsintensive Änderungen ist. Wenn sich beispielsweise das Format der Ausgabe ändert, müsste der Test entsprechend aktualisiert werden.

### Begründung:

Das ist eine schlechte Umsetzung des Professionalitätsprinzips, da der Test zu stark an die konkrete Implementierung gebunden ist und somit nicht flexibel auf Änderungen reagieren kann. Tests sollten möglichst unabhängig von der internen Implementierung sein und sich stattdessen auf das erwartete Verhalten konzentrieren. In diesem Fall wäre es besser, die Funktionalität des Inventar-Drucks auf eine abstraktere Ebene zu testen, um eine flexible und robuste Testsuite zu gewährleisten.









## Code Coverage

Die aktuelle Codeabdeckung von 46 % spiegelt eine teilweise durchdachte Teststrategie wider, die darauf ausgerichtet ist, die Qualität und Zuverlässigkeit unserer Software sicherzustellen. Ich habe angestrebt, sowohl eine hohe Methoden- und Klassenabdeckung zu erreichen als auch eine Zeilenabdeckung von 80 %. Diese Entscheidung wurde nach einer gründlichen Abwägung von Kosten-Nutzen und Zeit-Nutzen sowie nach Berücksichtigung verschiedener Quellen getroffen.

Es ist wichtig anzumerken, dass die Anwendung nicht in einem Umfeld kritischer Anwendungen wie der medizinischen oder militärischen Nutzung arbeitet. In solchen Fällen sind höhere Abdeckungsraten möglicherweise ratsam. Dennoch war das Ziel, eine umfassende Testabdeckung zu erreichen, um sicherzustellen, dass die Robustheit, Fehlerfreiheit und Wartungsfreundlichkeit des Codes gewährleistet sind.

Trotz meiner Bemühungen hat sich gezeigt, dass die Klasse `GameController` aufgrund ihrer Komplexität und Größe schwer zu testen ist. Die Implementierung dieser Klasse ist sehr umfangreich und verlangt eine spezifische Herangehensweise an das Testen. Ein Refactoring dieser Klasse wäre sinnvoll.

Folgend ist unser Coverage Report zu sehen.

Element	Coverage
src/main/java - textventure	 46,4 %
> textventure.controllers	 3,0 %
> textventure.enums	 100,0 %
> textventure.events	 86,2 %
> textventure.filereader	 86,1 %
> textventure.inventory	 98,7 %
> textventure.main	0,0 %
> textventure.map	 91,3 %
> textventure.player	 92,3 %
> textventure.randomizer	100,0 %

## Fakes und Mocks

Die Verwendung von Mocks ist bei der Klasse `GameController` besonders sinnvoll, da diese Klasse sehr komplex ist und viele Abhängigkeiten hat. Durch das Mocken können wir isolierte Tests schreiben, die nur das Verhalten der `GameController`-Klasse überprüfen, ohne auf die korrekte Funktionalität der anderen Klassen angewiesen zu sein. Hier sind die relevanten Mock-Objekte für die Klasse `GameController`:

- `ConsoleController`: Dieser Controller ist für die Interaktion mit der Benutzeroberfläche verantwortlich. Durch das Mocken können wir sicherstellen, dass die richtigen Nachrichten angezeigt werden, ohne tatsächlich auf die Konsole zugreifen zu müssen.
- `Player`: Der Spieler ist eine wichtige Entität in unserem Spiel. Durch das Mocken können wir das Verhalten des Spielers steuern und verschiedene Szenarien testen, ohne dass ein echter Spieler benötigt wird.

- `Map`: Die Karte ist für die Spielwelt verantwortlich. Durch das Mocken können wir verschiedene Kartenkonfigurationen simulieren und sicherstellen, dass der `GameController` korrekt mit verschiedenen Karten umgehen kann.
- `HighscoreManager`: Dieser Manager ist für das Speichern und Laden von Highscores zuständig. Durch das Mocken können wir sicherstellen, dass der `GameController` korrekt mit dem Highscore-System interagiert, ohne tatsächlich Dateioperationen durchführen zu müssen.

Die Tests für die `GameController`-Klasse wurden nicht umgesetzt, da diese Klasse äußerst komplex ist. Das Mocken der Abhängigkeiten wäre jedoch der beste Ansatz, um isolierte Tests für den `GameController` zu schreiben. Damit könnten wir das Verhalten der Klasse testen, ohne tatsächlich die gesamte Spiellogik ausführen zu müssen

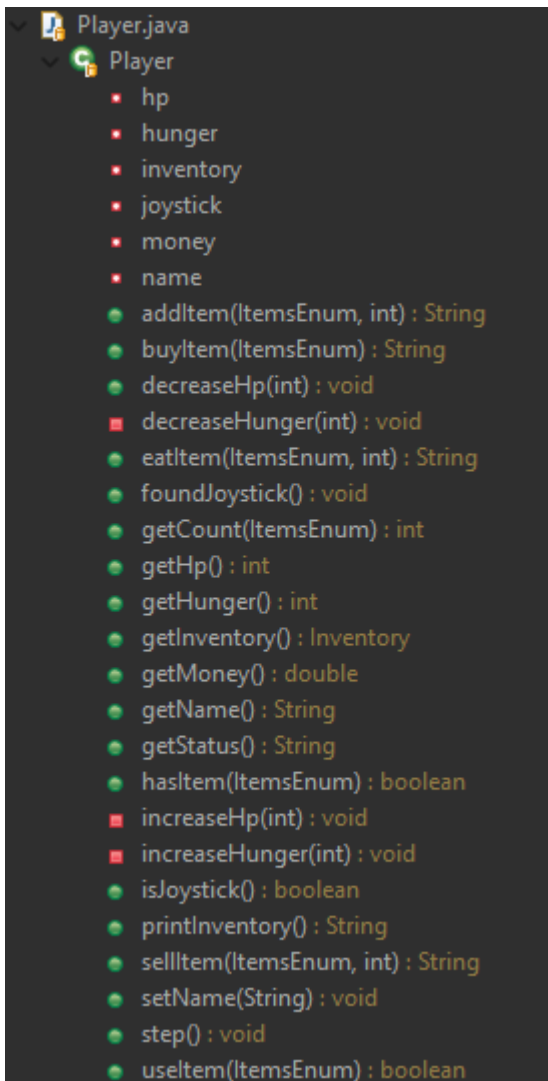
## Kapitel 6: Domain Driven Design

### Ubiquitous Language

Bezeichnung	Bedeutung	Begründung
Player	Ein Spieler im Spiel	Der Begriff Player ist allgegenwärtig in der Spiellogik und beschreibt eine spielinterne Entität.
Inventory	Das Inventory eines Spielers	Es ist ein zentrales Konzept im Spiel und wird häufig verwendet, um den besitz und die Gegenstände eines Spielers zu repräsentieren.
Highscore	Die Rangliste der besten Spieler	Die Highscores sind ein wichtiges Element im Spiel, um die Leistung der Spieler zu verfolgen und zu vergleichen.
Map	Die Spielkarte	Die Map ist ein wesentlicher Bestandteil im Spiel, das die Spielwelt repräsentiert und von Teilen der Spiellogik verwendet wird.

### Entities

Player



Die Player-Entität ist eine wichtige Komponente in einem Textabenteuerspiel. Sie repräsentiert einen Spieler innerhalb des Spiels und trägt alle relevanten Informationen über den Spieler. Hier ist eine ausführliche Beschreibung und Begründung für den Einsatz dieser Entität:

**Beschreibung:** Die Player-Entität ist dafür verantwortlich, alle relevanten Informationen über den Spieler im Spiel zu speichern und zu verwalten. Dazu gehören:

1. **Name des Spielers:** Der Name des Spielers wird benötigt, um den Spieler zu identifizieren und ihm eine persönliche Identität im Spiel zu geben.
2. **Position auf der Karte:** Die Position des Spielers auf der Spielkarte ist von entscheidender Bedeutung, da sie bestimmt, wo sich der Spieler im Spiel befindet und welche Aktionen er ausführen kann.
3. **Aktueller Zustand des Spielers:** Dies umfasst Informationen wie die Gesundheit des Spielers, den Zustand seines Inventars und möglicherweise andere Attribute wie Erfahrungsstufen oder Punktestände.

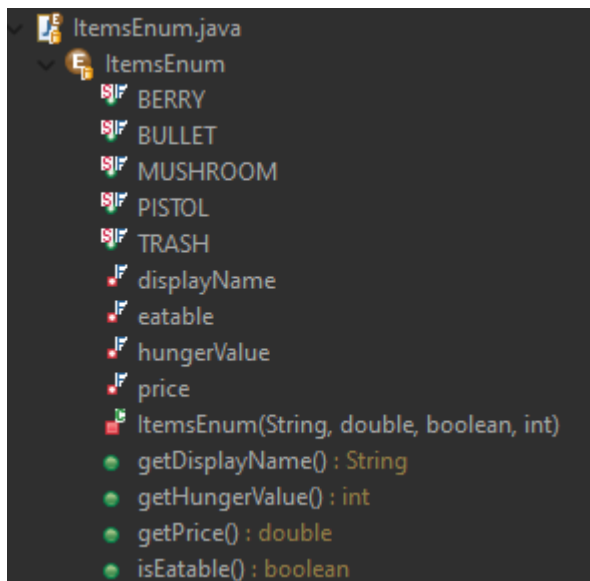
**Begründung für den Einsatz:** Die Player-Entität ist entscheidend für die Modellierung des Spielers im Spiel. Hier sind einige Gründe, warum diese Entität wichtig ist:

1. **Identifikation und Personalisierung:** Der Spieler sollte im Spiel eine eindeutige Identität haben, die es ihm ermöglicht, sich mit dem Spiel zu identifizieren. Durch die Speicherung des Spielerzeichens können wir sicherstellen, dass der Spieler eine personalisierte Erfahrung im Spiel hat.
2. **Positionsbasierte Aktionen:** Die Position des Spielers auf der Karte bestimmt, welche Aktionen er ausführen kann und welche Ereignisse ihn beeinflussen können. Durch die Speicherung der Spielerposition können wir sicherstellen, dass der Spieler korrekt mit der Spielwelt interagieren kann.
3. **Verwaltung des Spielerzustands:** Der Zustand des Spielers ändert sich im Laufe des Spiels, z. B. durch Kämpfe, Interaktionen mit der Umgebung oder den Konsum von Gegenständen. Durch die Speicherung des Spielerzustands können wir sicherstellen, dass der Spieler immer über die aktuellsten Informationen verfügt und das Spiel konsistent bleibt.

Insgesamt ist die Player-Entität unverzichtbar für die Modellierung eines Spielercharakters in einem Textabenteuerspiel. Sie ermöglicht es, den Spieler zu identifizieren, seine Aktionen zu steuern und seinen Zustand zu verwalten, was wesentlich zur Spielerfahrung und zum Spielspaß beiträgt.

## Value Objects

ItemsEnum



Jeder Eintrag in der Enumeration enthält Informationen über den Gegenstand, einschließlich seines Anzeigenamens, seines Preises, seiner Essbarkeit und des Hungerwerts, den er dem Spieler gibt, wenn er gegessen wird.

### Begründung des Einsatzes von Value Objects:

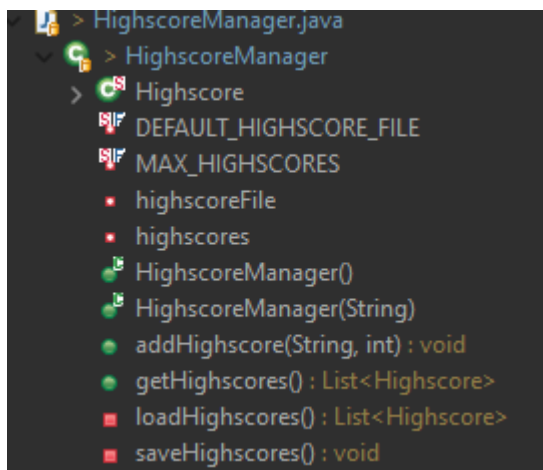
1. **Klar definierte und feste Attribute:** Jeder Eintrag in der Enumeration repräsentiert einen eindeutigen Spielgegenstand mit fest definierten Attributen wie Anzeigenamen, Preis, Essbarkeit und Hungerwert. Diese Attribute sind unveränderlich und bieten eine klare und konsistente Repräsentation der Spielgegenstände.

2. **Wiederverwendbarkeit und Konsistenz:** Durch die Verwendung einer Enumeration können die verschiedenen Spielgegenstände zentral definiert und verwaltet werden. Dies fördert die Wiederverwendbarkeit des Codes und stellt sicher, dass alle Teile des Spiels konsistent auf die gleichen Spielgegenstände zugreifen.
3. **Typsicherheit und Lesbarkeit:** Da die Enumeration eine begrenzte Anzahl von vordefinierten Einträgen enthält, verbessert sie die Typsicherheit des Codes. Dies erleichtert das Schreiben von Code und reduziert die Wahrscheinlichkeit von Fehlern. Außerdem verbessert die Verwendung aussagekräftiger Namen wie "BERRY" und "PISTOL" die Lesbarkeit des Codes und erleichtert das Verständnis der verschiedenen Spielgegenstände.
4. **Einfache Erweiterbarkeit:** Wenn neue Spielgegenstände hinzugefügt werden müssen, kann dies einfach durch das Hinzufügen eines neuen Enum-Eintrags erfolgen. Dies erleichtert die Wartung des Codes und ermöglicht eine einfache Erweiterung des Spielinhalts, ohne bestehenden Code umfangreich ändern zu müssen.

Insgesamt bietet die Verwendung der Enumeration `ItemsEnum` eine einfache und effektive Möglichkeit, Spielgegenstände im Textabenteuerspiel zu repräsentieren und zu verwalten. Sie trägt zur Klarheit, Konsistenz, Typsicherheit und Erweiterbarkeit des Codes bei und erleichtert die Entwicklung und Wartung des Spiels.

## Repositories

HighscoreManager



Der HighscoreManager kann als ein Repository betrachtet werden, da er für das Speichern und Abrufen von Highscores verantwortlich ist.

### HighscoreManager als Repository:

1. **Verwaltung von Highscores:** Der HighscoreManager ist dafür zuständig, Highscores zu speichern, zu aktualisieren und abzurufen. Ähnlich wie ein Repository in einem klassischen Datenbankkontext verwaltet der HighscoreManager die persistenten Daten des Spiels, nämlich die Highscore-Einträge.
2. **Abstraktion des Datenzugriffs:** Der HighscoreManager abstrahiert den Zugriff auf die Highscore-Daten, indem er Methoden wie `addHighscore` und `getHighscores` bereitstellt. Dadurch wird der GameController und andere Klassen von der direkten Interaktion mit der Speicherimplementierung entkoppelt, was die Wartbarkeit und Flexibilität des Codes verbessert.

3. **Einfache Austauschbarkeit der Speicherimplementierung:** Durch die Verwendung des HighscoreManagers als Repository kann die konkrete Speicherimplementierung leicht ausgetauscht werden, ohne dass Änderungen an anderen Teilen des Codes vorgenommen werden müssen. Wenn zum Beispiel die Highscores später in eine Datenbank gespeichert werden sollen, könnte eine entsprechende Datenbankimplementierung des HighscoreManagers erstellt werden, während die Schnittstelle für den GameController und andere Klassen unverändert bleibt.
4. **Einheitlicher Zugriff auf Highscore-Daten:** Durch die Verwendung des HighscoreManagers als Repository wird ein einheitlicher Zugriff auf Highscore-Daten in der gesamten Anwendung gewährleistet. Alle Klassen, die auf Highscores zugreifen müssen, können dies über den HighscoreManager tun, ohne auf die spezifische Implementierungsdetails des Datenzugriffs achten zu müssen.

## Aggregates

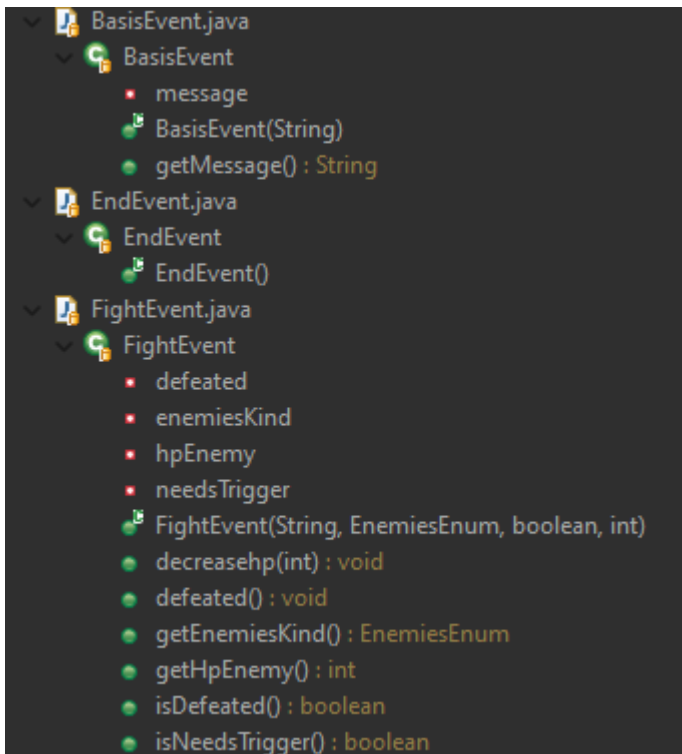
Ein Beispiel für ein Aggregat in unserem Projekt ist die Kombination von BasisEvent und den spezifischen Eventtypen wie FightEvent, StartEvent und EndEvent. Diese Ereignisse sind eng miteinander verbunden und bilden zusammen ein Aggregat.

### Begründung des Einsatzes dieses Aggregats:

**Strukturelle Klarheit:** Die verschiedenen Eventtypen wie FightEvent, StartEvent und EndEvent sind strukturell eng miteinander verbunden, da sie alle auf BasisEvent basieren. Durch die Aggregation dieser Ereignisse wird die strukturelle Klarheit verbessert, da ihre Beziehung zueinander klar dargestellt wird.

**Benutzerinteraktion:** In unserer Textadventure-Anwendung ist es wichtig, dass Benutzer die verschiedenen Eventtypen manipulieren können. Durch die Aggregation der verschiedenen Ereignisse können Benutzer alle Ereignisse gleichzeitig bearbeiten und verwalten, was die Benutzerinteraktion vereinfacht und verbessert.

**Klare Modellierung:** Ein Aggregat ermöglicht eine klare Modellierung der Beziehung zwischen BasisEvent und den spezifischen Eventtypen. Es verdeutlicht, dass die spezifischen Eventtypen Teil des übergeordneten BasisEvents sind, jedoch jeweils ihre eigenen spezifischen Eigenschaften und Funktionen haben können. Dies erleichtert die Modellierung des Systems und verbessert das Verständnis der Systemstruktur.



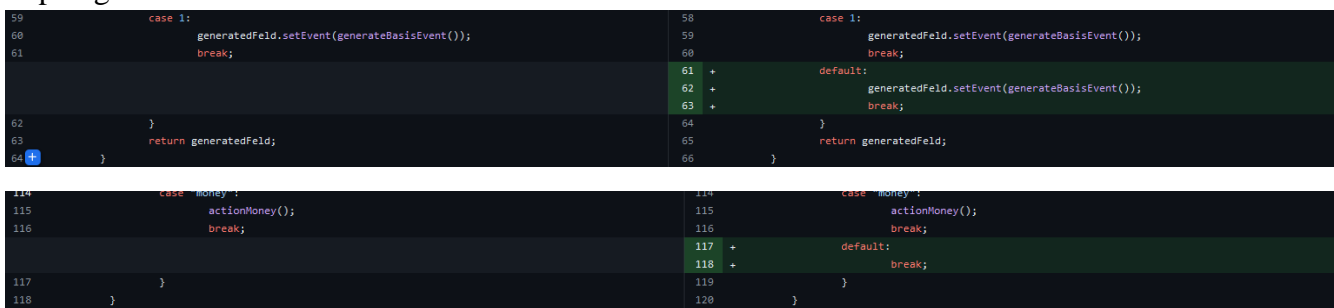
## Kapitel 7: Refactoring

### Code Smells

#### Switch Statements

Im Zuge der Codeanalyse fielen zwei Switch-Statements auf, die anfänglich ohne einen default case auskamen. Das stellte ein potenzielles Risiko dar, da unvorhergesehene Eingaben nicht richtig behandelt wurden. Als Lösung wurde im Zuge des Refactorings ein default case zu diesen Switch-Statements hinzugefügt. Dadurch wurde sichergestellt, dass das Programm robuster gegen unerwartete Eingaben ist und potenzielle Fehlerquellen minimiert wurden.

<https://github.com/N-Eisen/ASWE/commit/7b2ca005372a79b8fb64eefbf6368a8cfc78c15b>



#### Long Class

Aufgrund der Länge der Klasse GameController wurden drei Methoden – displayBorder, displayControls und displayWelcomeMessage – in den ConsoleController verschoben, um die Klasse zu entlasten und die Lesbarkeit des Codes zu verbessern. Diese Umstrukturierung erleichtert die



Wartung und Erweiterung des Codes, da die Verantwortlichkeiten klarer verteilt sind und die Codebasis insgesamt schlanker wird.

<https://github.com/N-Eisen/ASWE/commit/82c00e1ce6266b8666e9d7ec03f4579fa0aa348a>

```
21 - e.printStackTrace();
22 }
23
24 + e.printStackTrace();
25 + }
26 + }
27 +
28 + public void displayControls() {
29 +     try (BufferedReader reader = new BufferedReader(
30 +         new FileReader("src/main/java/textventure/filereader/Controls"))) {
31 +         String line;
32 +         while ((line = reader.readLine()) != null) {
33 +             slowPrint(line, 50);
34 +         }
35 +     } catch (IOException e) {
36 +         System.out.println("Error reading welcome message file: " +
37 +             e.getMessage());
38 +     }
39 + }
40 +
41 + public void displayBorder() {
42 +     slowPrint("-----", 50);
43 + }
44 +
45 + public void displayWelcomeMessage() {
46 +     try (BufferedReader reader = new BufferedReader(
47 +         new FileReader("src/main/java/textventure/filereader/
48 +             WelcomeMessage"))) {
49 +         String line;
50 +         while ((line = reader.readLine()) != null) {
51 +             slowPrint(line, 50);
52 +         }
53 +     } catch (IOException e) {
54 +         System.out.println("Error reading welcome message file: " +
55 +             e.getMessage());
56 +     }
57 + }
```

## 2 Refactorings

### Name Conventions

Im Zuge der Codeüberprüfung wurden die Namenskonventionen gründlich überarbeitet, da festgestellt wurde, dass einige Variablen und Methoden nicht den üblichen Konventionen entsprachen. Durch diese Anpassungen wurde die Konsistenz im Code verbessert und die Lesbarkeit erhöht, was letztendlich zu einer besseren Wartbarkeit und Verständlichkeit des Programms beiträgt.

<https://github.com/N-Eisen/ASWE/commit/8fbd25b3068e7613fb36acfb437835f69954b827>

### Extract Method

Im Zuge der Codeanalyse wurde identifiziert, dass eine Methode namens `answerIsYes` extrahiert werden konnte. Dies wurde notwendig, da an verschiedenen Stellen im Code abgefragt wurde, ob die Antwort des Spielers "ja" ist.

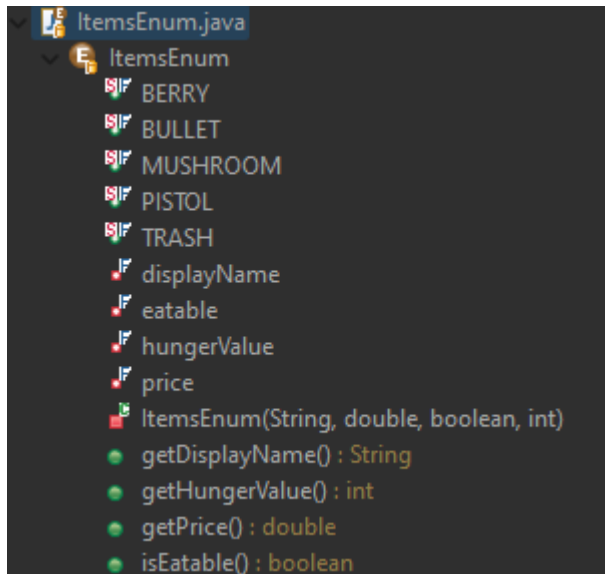
<https://github.com/N-Eisen/ASWE/commit/778994fbf8769459695b8f1734dd84de0f43fe9a>

## Kapitel 8: Entwurfsmuster

### Entwurfsmuster: [Erbauer]

Im Code wird das Builder-Muster verwendet, um Instanzen von `ItemsEnum` schrittweise aufzubauen. Dies ermöglicht die Erstellung von Objekten mit verschiedenen Eigenschaften wie Name, Preis, Essbarkeit und Hungerwert. Die `ItemsEnum`-Klasse definiert einen Enum-Konstruktor, der die notwendigen Parameter entgegennimmt, um eine Instanz zu erstellen.

`ItemEnum` ist eine Aufzählungsklasse, die verschiedene Gegenstände im Spiel repräsentiert. Jedes Element der Aufzählung hat eine Reihe von Eigenschaften wie Namen, Preis, Essbarkeit und Hungerwert. Durch den Enum-Konstruktor werden diese Eigenschaften beim Erstellen einer neuen Instanz von `ItemEnum` festgelegt. Die Verwendung des Builder-Musters ermöglicht es, die Erstellung von Objekten zu vereinfachen und die Flexibilität bei der Konfiguration zu erhöhen.



## Entwurfsmuster: [Komposition]

### ***Klasse Map:***

Die Klasse `Map` repräsentiert die Gesamtstruktur der Spielkarte und dient als Kompositum. Sie enthält eine Sammlung von Spielfeldern und implementiert Methoden zum Generieren und Durchsuchen von Spielfeldern.

### ***Klassen BasisFeld, StartFeld und EndFeld:***

Die Klassen `BasisFeld`, `StartFeld` und `EndFeld` stellen spezifische Arten von Spielfeldern dar und können als Blattkomponenten betrachtet werden. Sie implementieren die gemeinsame Schnittstelle für alle Spielfelder und enthalten die spezifischen Details und Funktionen für jedes Feld.

## Begründung des Einsatzes:

1. **Hierarchische Struktur:** Durch die Verwendung des Kompositum-Musters wird eine hierarchische Struktur der Spielkarte geschaffen, wobei die Klasse `Map` als Wurzel fungiert und einzelne Spielfelder als Komponenten enthält.
2. **Gleichartige Behandlung:** Alle Spielfelder werden über eine gemeinsame Schnittstelle behandelt, unabhängig von ihrer konkreten Implementierung als `BasisFeld`, `StartFeld` oder `EndFeld`. Dies erleichtert die Interaktion mit den Spielfeldern und fördert die Wiederverwendbarkeit von Code.
3. **Einfache Erweiterbarkeit:** Das Muster ermöglicht es, neue Arten von Spielfeldern einfach hinzuzufügen, indem neue Klassen für spezifische Feldtypen erstellt werden. Diese neuen Klassen können nahtlos in die bestehende Struktur der Spielkarte integriert werden.

4. **Flexibilität:** Die Verwendung des Kompositum-Musters bietet Flexibilität bei der Strukturierung der Spielkarte, da einzelne Spielfelder oder Gruppen von Spielfeldern je nach Bedarf hinzugefügt, entfernt oder modifiziert werden können.

Indem das Kompositum-Muster auf die Klasse `Map` angewendet wird, wird die Strukturierung der Spielkarte verbessert und die Flexibilität bei der Handhabung von Spielfeldern erhöht. Dies trägt zu einer klareren und wartungsfreundlicheren Codebasis bei.