Advanced Software Engineering

Projektdokumentation

for the examination of Bachelor of Science (B.Sc.)

in **Informatik (Computer Science)**

at Duale Hochschule Baden-Württemberg Karlsruhe

authored by

Lukas Rapp and Simon Reichle

Due date:	May 28th, 2023
Matriculation number, class:	7602924, 7400045, TINF20B2
Period:	04.10.2022 - 28.05.2023
Inspector at Duale Hochschule:	Dr. Lars Briem

Contents

List of Figures

1 Einführung

1.1 Übersicht über die Applikation

Bei dieser Applikation handelt es sich um einen *Single Player Dungeon Crawler*, der *The Binding Of Isaac* in seinen Grundzügen nachempfunden ist. Der Spieler bewegt sich durch prozedural generierte Level. Dabei kann er verschiedene Gegenstände einsammeln wie Rüstung (*Armor*), Waffen (*Weapons*), Münzen (*Coins*) und Herzen (*Hearts*). Diese Gegenstände sollen ihm dabei helfen zu überleben. Die Münzen stellen jedoch den Punktestand dar. Während der Spieler sich durch die Räume bewegt, stößt er auf eine Palette unterschiedlicher Gegner (*Spider*, *Skeleton*, Zombie, *Ogre*). Diese besitzen auch unterschiedliche Fähigkeiten und versuchen den Spieler zu besiegen. Betritt der Spieler einen Raum, so wird dieser gesperrt, bis er alle darin befindlichen Gegner besiegt hat. Erst dann öffnen sich die Türen wieder und der Spieler kann in anliegende Räume laufen. Stirbt der Spieler dabei, so ist das Spiel zu Ende. Bleibt er am Leben, hangelt er sich von Level zu Level. Am Ende eines jeden Levels steht ein Raum mit einer einzigen Falltür in der Mitte, welche den Spieler in das nächste Level bringt.

Das Spiel ist rundenbasiert. In jeder Runde stehen Aktionen zur Verfügung, wie etwa Angreifen (*Space*), Bewegen (*W*, *A*, *S* oder *D*) oder Aufheben (*E*). Auch die Gegner agieren rundenbasiert. Diese können jedoch lediglich angreifen. Sie laufen dem Spieler mit einer Runde Zeitversatz hinterher. Dabei können Gegner sich auch gegenseitig den Weg blockieren.

Damit der Spielstand in Form von Level, Gegnern und Items nachvollziehbar bleibt, gibt es eine einfache Anzeige, welche mithilfe von ANSI- und ASCII-Zeichen den Spielinhalt darstellt. Dabei helfen unterschiedliche Symbole und Farben den Inhalt zu verstehen.

1.2 Wie startet man die Applikation?

Voraussetzung ist **Java 19**. Optimalerweise wird das Programm in einem ANSI-fähigen Terminal (Farbunterstützung und Kontrollzeichen) gestartet. Diese werden bei den allermeisten Linux Distributionen (z.B. Ubuntu) direkt mitgeliefert. Alternativ funktioniert auch die IntelliJ-Konsole.

Ausführung in Konsole:

Bei Vorliegen der JAR-Datei folgenden Befehl ausführen:
`/home/<User>/.jdks/openjdk-19.0.2/bin/java -jar ASE.jar`

Ausführung in IntelliJ:

Im Projekt befindet sich unter `./src/main/java/plugins` die Klasse *Main* mit der obligatorischen *main()*-Methode. Diese Methode lässt sich mittels Knopfdruck ausführen.

Anmerkung:

Um externe Abhängigkeiten zu minimieren, wurde auf eine Key-Input-Library verzichtet. Für die Interaktion benötigen wir allerdings spontane Tasteneingaben, welche nicht durch *Enter* bestätigt werden. Die Lösung in Java ist daher, ein winziges Fenster mit Fokus zu öffnen. Auf diesem ist ein *KeyListener* registriert, welcher die spontanen Tasteneingaben entgegennimmt und an das eigentliche Programm weiterleitet. Damit erhält man auch in der Konsole direkte Interaktivität.

Der Nachteil ist allerdings, dass der Fokus auf dieses Fenster für einen normalen Benutzer nur schwierig wieder zu erlangen ist, sobald eine Maus-Interaktion außerhalb des Programmes getätigt wurde. Dann wird nämlich der Fokus durch das Betriebssystem vom Fenster genommen.

Daher ist darauf zu achten, dass, sobald das Programm gestartet ist, lediglich Tasteneingaben erfolgen. Ansonsten muss das Programm neu gestartet werden, damit das Fenster wieder den Fokus hat. Die Applikation kann in der Konsole dennoch mittels der Tastenkombination *Ctrl + C* mit einem Signal beendet werden.

Im Übrigen sollte das Konsolen-Interface wie folgt aussehen. Sind unpassende Zeichen zu sehen oder fehlen Farben, dann ist das Terminal nicht im richtigen Modus oder unterstützt grundsätzlich nicht die Darstellung. Ein Unix-Terminal oder die IntelliJ-Konsole schaffen dabei Abhilfe.

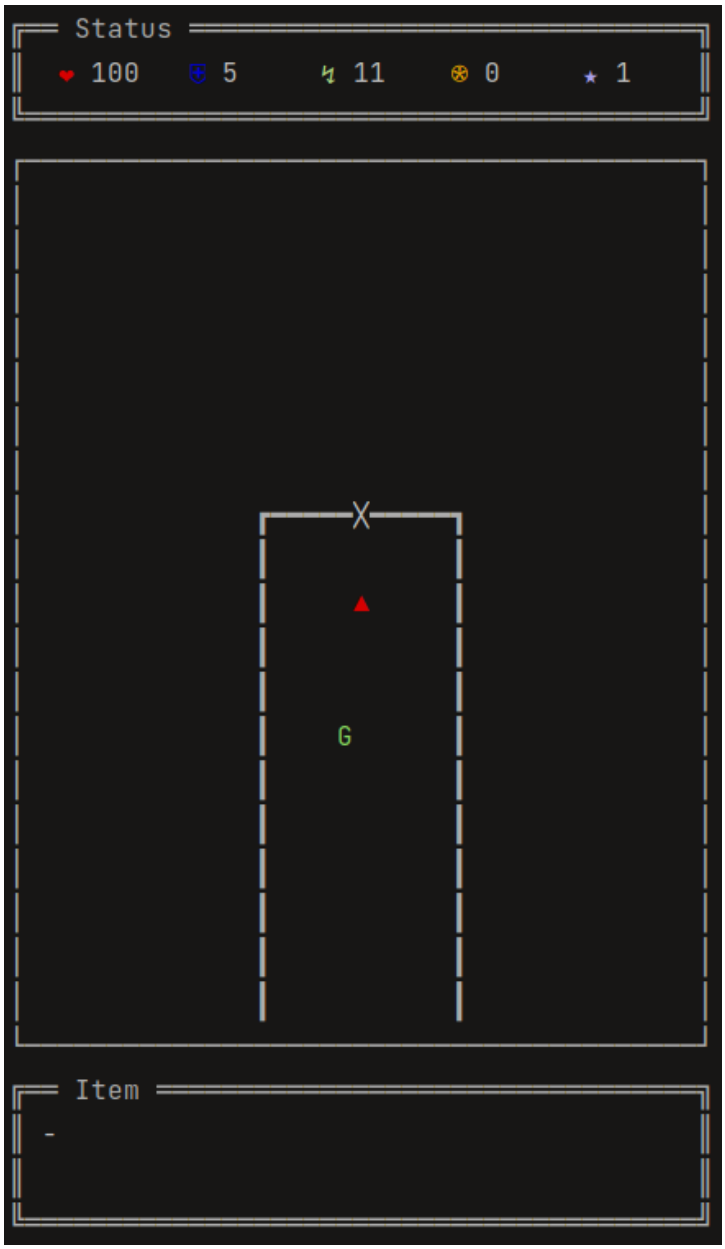

Figure 1.1: Korrekt dargestelltes Konsolen-Interface

1.3 Wie testet man die Applikation?

Alle Tests befinden sich im Ordner `./src/test/java`. Sie sind mittels **JUnit5** umgesetzt. Die vorliegenden Tests können mithilfe der eingebauten IDE-Funktionen einzeln oder im Verbund ausgeführt werden. Es existieren sowohl *Blackbox-Integration-Tests*, als auch normale *Unit Tests*.

2 Clean Architecture

2.1 Was ist Clean Architecture?

Die *Clean Architecture* ist ein Gestaltungskonzept, das darauf abzielt möglichst robuste und wartbare Anwendungen zu entwickeln. Dabei wird eine Software in hierarchisch mehrere Schichten organisiert, was gemeinhin als *Onion-Architektur* bekannt ist. Jede Schicht weist dabei eine spezifische Verantwortung (siehe *domain*, *plugins* etc.).

Die Schichten sind über Schnittstellen verbunden. Hinter den Schnittstellen steckt dann eine tatsächliche Implementation. Ziel ist es dadurch eine Trennung zwischen der Logik einer Schicht und den technischen Details einer anliegenden Schicht zu erzeugen. Statt durch konkrete Implementierungsdetail sind die Schichten (zumindest von innen nach außen) über Funktionsabmachungen in Form der Schnittstellen verbunden. Dadurch lassen sich leicht Änderungen in einer Schicht vornehmen, ohne dabei zwangsläufig Änderungen in einer anderen Schicht vornehmen zu müssen.

Entscheidend für das Prinzip ist die Abhängigkeitsrichtung. Äußere abstraktere Schichten (hierarchisch untergeordnet) können direkt auf innere Schichten zugreifen und von ihnen abhängig sein. Jedoch gilt dies nicht für die Umkehrrichtung. Aufrufe von innen nach außen werden über Schnittstellen und Abstraktionen sichergestellt (*Dependency Inversion* und *Injection*). Tieferliegende Schichten sind zudem am wenigsten abstrakt und damit am langlebigsten.

Dieser Sachverhalt führt zu Anwendungen, die wartbarer und robuster sind, weil Änderungen an einer Schicht meist keine Änderungen an anderen Schichten verursachen. Durch clevere Trennung und Abstraktionen lässt sich der Anpassungaufwand durch eine beliebige Änderung an vielen Stellen einsparen.

2.2 Analyse der Dependency Rule

Die Schichten der *Onion-Architektur* sind sortiert von innerster nach äußerster Schicht *abstraction*, *domain*, *application* und *plugin*. Es sind lediglich Abhängigkeiten von außen nach innen erlaubt, aber nicht umgekehrt. Die Schichten sind in einzelne Module gegliedert. Diese haben andere Module, also andere Schichten als fest definierte Import-Möglichkeiten. Dadurch wird sichergestellt, das eine Schicht auch nur von unterliegenden Schichten abhängig sein kann. Sollte eine Abhängigkeit nach außen bestehen, so kann das Programm nicht gebaut werden, weil Imports fehlen.

Positiv-Beispiel I

Die nachfolgende Abbildung zeigt ein vereinfachtes UML-Diagramm der Klasse *GridPosition* und die Abhängigkeiten in beide Richtungen. Die Klasse stammt aus der Schicht *application*. Wie zu sehen, hängt *GridPosition* lediglich von den Klassen *Position* und *IVectorizable* aus den jeweilig unterliegenden Schichten *domain* und *abstraction* ab. Weiterhin ist zu sehen, dass nur Klassen der gleichen Schicht (*application*) von *GridPosition* abhängen, jedoch keine Klasse aus einer tieferen Schicht (weiter innen liegend, z.B. *abstraction* oder *domain*). Dadurch ist die *Dependency Rule* gewahrt.

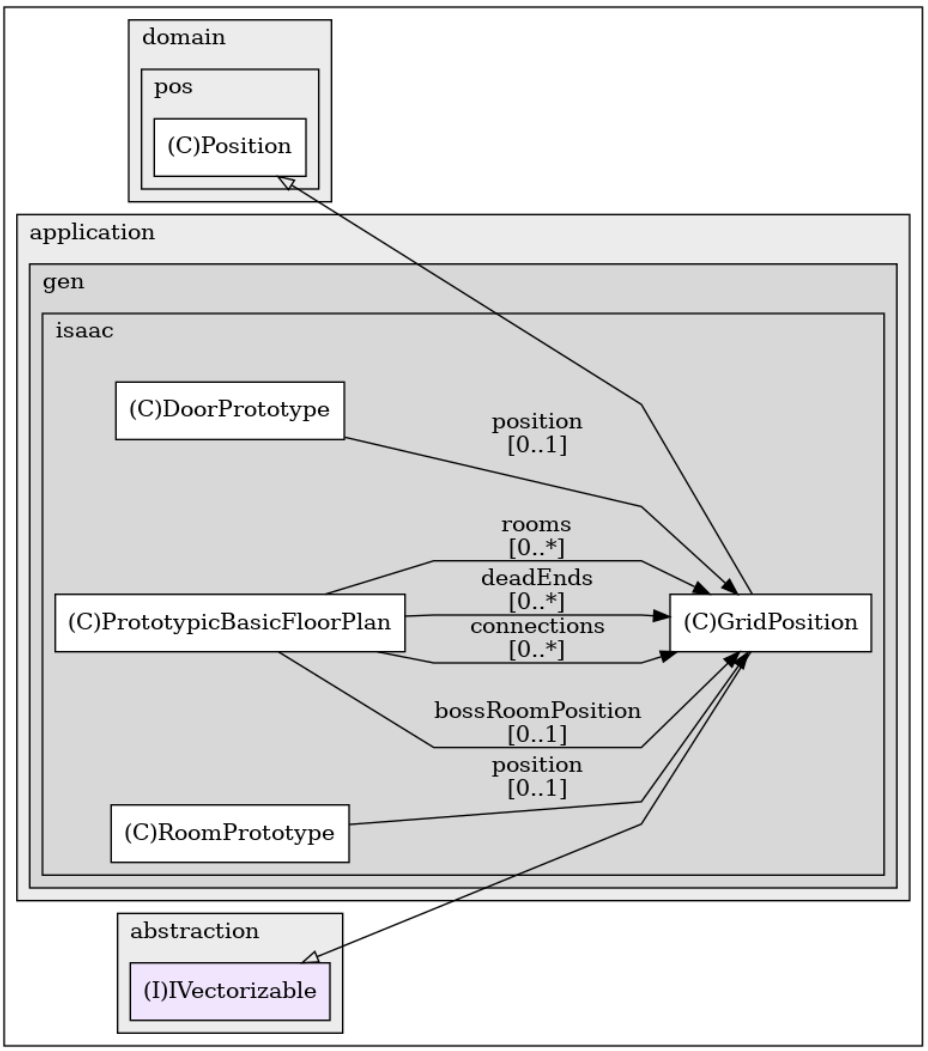

Figure 2.1: Dependency Rule I: GridPosition

Positiv-Beispiel II

Die nachfolgende Abbildung zeigt ein vereinfachtes UML-Diagramm der Klasse *Player* und die Abhängigkeiten in beide Richtungen. Die Klasse stammt aus der Schicht *domain*. Es ist zu sehen, dass *Player* von einigen Klassen der gleichen Schicht wie etwa *LivingEntity*, *Level* und *Item* abhängt. Darüber hinaus hängt die Klasse *Room* von der Klasse *Player* ab. Interessant ist allerdings, dass die Klasse *GameState* aus der höherliegenden Schicht *application* von *Player* abhängt. Die Abhängigkeitsrichtung zeigt von außen nach innen, verletzt also die *Dependency Rule* nicht. Die Klasse *Player* hängt jedoch noch von einem Interface names *IGlobalState* ab, welches von *GameState* implementiert wird. Das Interface stellt daher eine Schnittstelle zwischen *domain* und *application* dar, durch welche der *Player* mit konkreten Implementationen von *IGlobalState* in höherliegenden Schichten, wie etwa *GameState*, kommunizieren kann, ohne von ihnen abhängig zu sein. Dies ist ein wichtiges Merkmal der *Clean Architecture*.

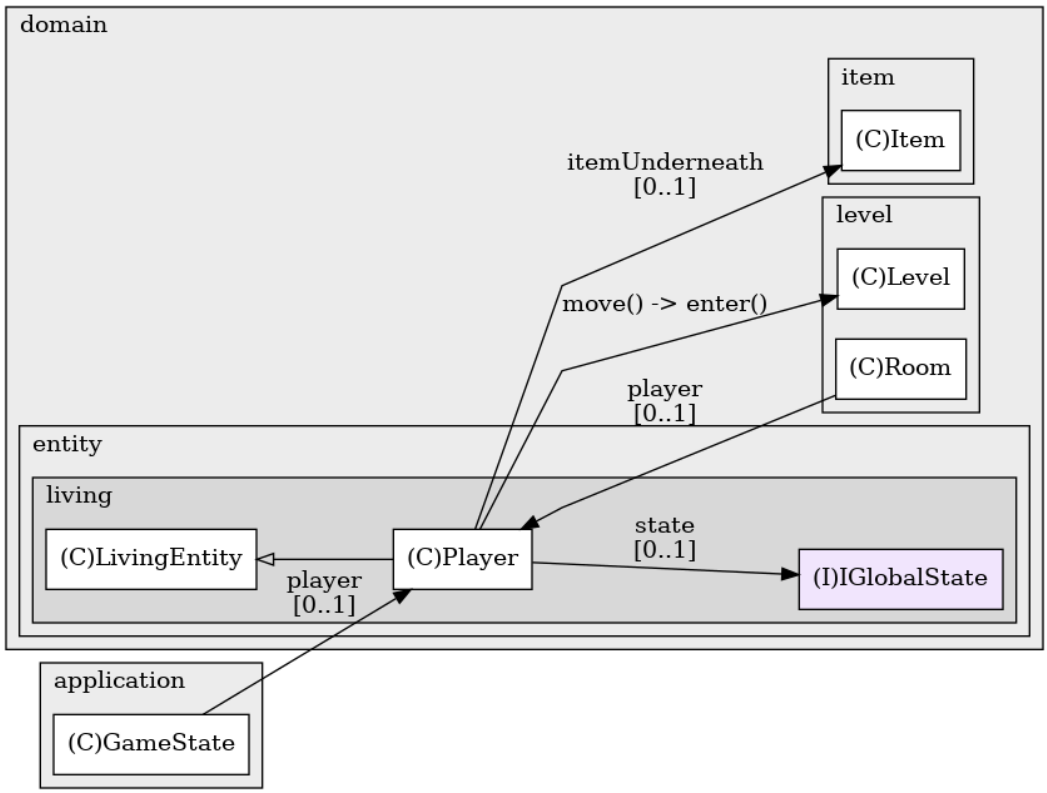

Figure 2.2: Dependency Rule II: Player

2.3 Analyse der Schichten

Domain Schicht

Die Klasse *Room* ist ein integraler Bestandteil des Spiels (Domäne) und ist daher in der *domain* Schicht integriert. Eine Umsiedlung in eine andere Schicht ist nicht sinnvoll. Wie der nachfolgenden Grafik zu entnehmen ist, hängt sie sehr stark mit anderen Klassen der Domäne wie etwa *Item*, *Level*, *Door* oder *Entity* zusammen. Dies unterstreicht die Bedeutung in der Domäne und legitimiert die Einordnung. Ihre Aufgabe ist die modellhafte Abbildung eines Level-Raumes und dessen Verwaltung. Dabei hält es alle enthaltenen Gegner, den Spieler und Items vor. Zudem ist dem Raum auch bewusst, ob er bereits betreten wurde und ob er von Gegnern befreit wurde. Wurde der Raum noch nicht betreten, so werden Gegner beschworen. Ist er noch nicht komplett von Gegner befreit, hindert er den Spieler am Fliehen. Zudem stellt es einige Helferfunktionen bereit, wie etwa *getClosestEnemy()*, da es umfangreiches Wissen über den Zustand des Raumes beinhaltet.

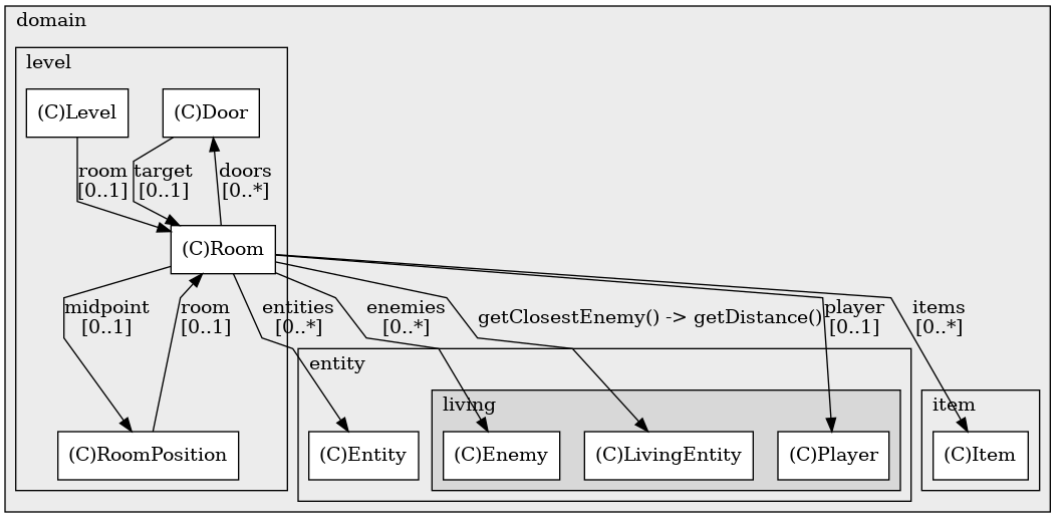

Figure 2.3: Analyse der Schichten: Domain

Plugin Schicht

Die Klasse *TerminalInterface* ist ein integraler Bestandteil der Anzeige bzw. Ausgabe und ist daher in der *plugin* Schicht integriert. Eine Umsiedlung in eine andere Schicht ist nicht sinnvoll. Die Klasse *TerminalInterface* ist ein Kompositum, welches sich aus mehreren visuellen Komponenten (*ItemView*, *StateView* und *LevelView*) zusammensetzt. Diese lesen Spiel-Informationen aus und wandeln sie in eine passende Darstellung um. In diesem Fall, wird in Zeichen-Buffer (siehe Klasse *Buffer* und *CompositeBuffer*) gerendert. Komposita sind üblich für die Erstellung von hierarchischen Anzeigen, welche sich üblicherweise in der äußersten Schicht befinden. Dies untermauert die Einordnung in die *plugin* Schicht. Zudem hängt keine andere Klasse, nicht einmal über ein Interface, von *TerminalInterface* ab. Das *TerminalInterface* selbst hängt wiederum von einem Interface namens *IInterface* ab, welches lediglich zur Orchestrierung einer Anzeige dient und *render()*-Befehle absetzt. Es ist daher sehr leicht auszutauschen durch eine andere Anzeige-Mechanik. Dies ist typisch für *Plugins*.

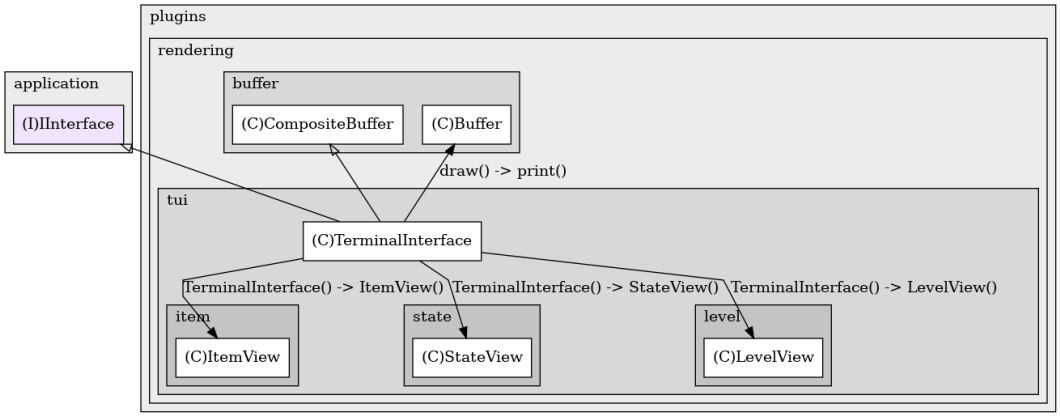

Figure 2.4: Analyse der Schichten: Plugin

3 SOLID

3.1 Analyse Single-Responsibility-Principle (SRP)

3.1.1 Positiv-Beispiel

Die nachfolgende Abbildung zeigt das gekürzte UML-Diagramm der Klasse *CollectAction*. Die ist in der Schicht *plugins* angesiedelt. Sie implementiert das *IAction*-Interface, welche vom Spieler initiierte Aktionen darstellt. Die einzige Aufgabe der *CollectAction* ist es, zu prüfen, ob ein Item unter dem Spieler liegt und dieses aufzunehmen bzw. mit dem bisherigen Item gleichen Typ (z.B. Rüstung) auszutauschen. Dafür bekommt es ein *Game*-Objekt übergeben, aus welchem die nötigen Informationen gezogen und die nötigen Funktionen aufgerufen werden können.

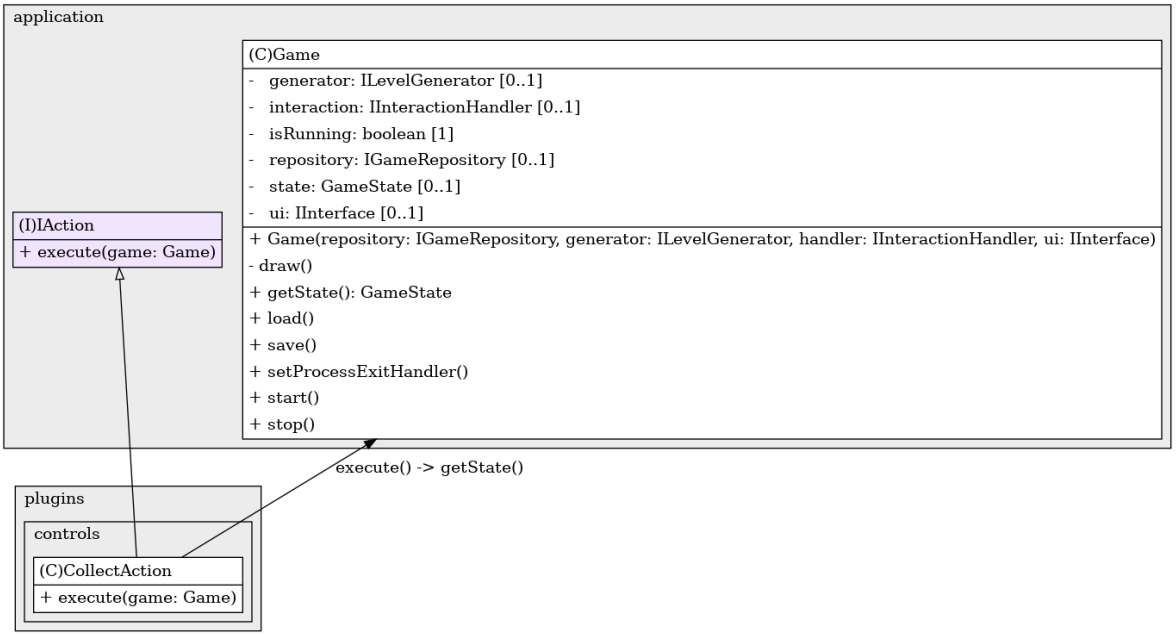

Figure 3.1: Analyse Single-Responsibility-Principle: Positiv

3.1.2 Negativ-Beispiel

3.2 Analyse Open-Closed-Principle (OCP)

3.2.1 Positiv-Beispiel

Die nachfolgende Abbildung zeigt das gekürzte UML-Diagramm des Interfaces *IAction*. Dieses ist in der Schicht *application* angesiedelt. Die *Main*-Klasse mit ihrem *Game Loop* kennt lediglich das Aktions-Interface *IAction*. Es bezieht die Aktionen aus einem *IInteractionHandler* (hier zur Vereinfachung nicht mit dargestellt). Diese werden dann mittels ihrer *execute()*-Methode ausgeführt.

Diese Struktur ermöglicht die Einhaltung des *Open-Closed-Principles*. Um eine neue Aktion zu implementieren, muss lediglich eine weitere Klasse hinzugefügt werden, welche das *IAction*-Interface implementiert. Alle anderen Implementierungen des *IAction*-Interface bleiben dabei unberührt, ebenso wie die *Main*-Klasse.

Die Verwendung dieses Schemas ist sehr sinnvoll, da vor allem in den frühen Phasen der Spielentwicklung viele Änderungen bezüglich der Spielerfahrung gemacht werden. Davon sind auch Aktionen und Tasteneingaben betroffen. Mit dem *IAction*-Interface lassen sich somit schnell Änderungen mit minimalem Eingriff umsetzen. Es müssen keine schnell unübersichtlichen *switch*-Statements benutzt werden.

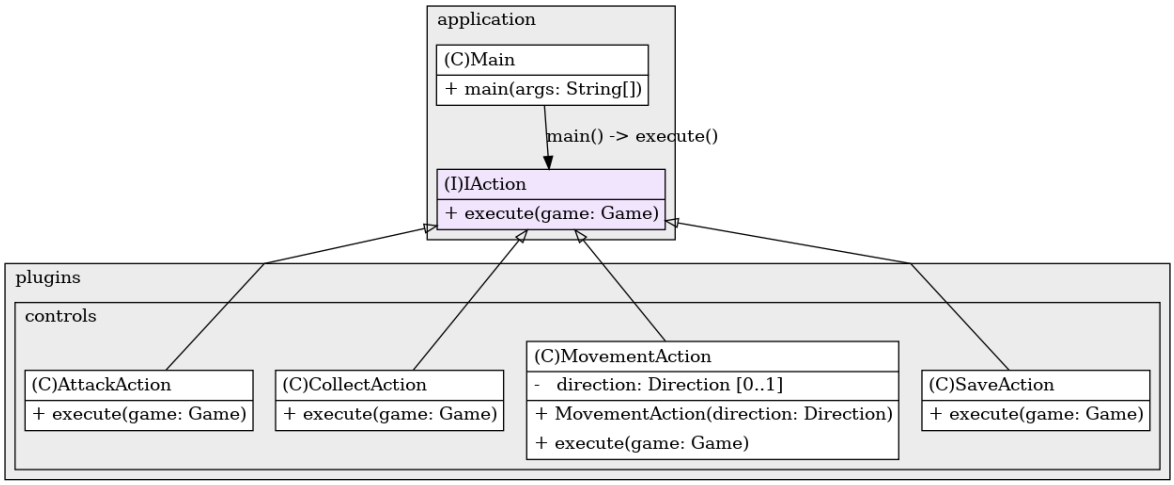

Figure 3.2: Analyse Open-Closed-Principle: Positiv

3.2.2 Negativ-Beispiel

3.3 Analyse Dependency-Inversion-Principle (DIP)

3.3.1 Positiv-Beispiel

Die nachfolgende Abbildung zeigt das UML-Diagramm der Klasse *IGameRepository*. Dieses ist in der Schicht *plugins* angesiedelt. Wie an den Pfeilen zu sehen ist, arbeiten die Schichten *application* und *plugins* über das Interface namens *IGameRepository* als Vermittler miteinander. Beide Pfeile laufen auf das Interface zu.

Das *Dependency-Inversion-Principle* ist erfüllt, da eine gemeinsame abstrakte Abmachung zwischen den Schichten besteht. Die Klasse *Game* hat keine Abhängigkeit zur Funktionalität der Klasse *FileGameRepository*, lediglich zu einer Abstraktion, mit welcher Funktionalität versichert wird. Stattdessen ist die Abhängigkeit umgekehrt (*Dependency Inversion*). Die Klasse *FileGameRepository* hat eine Abhängigkeit nach innen, nämlich zum Interface *IGameRepository*. Somit lassen sich leicht Änderungen am *FileGameRepository* vornehmen oder dieser gar durch eine ganz andere Persistenz-Methode ausgetauscht werden, ohne dass die Klasse *Game* davon direkt betroffen ist.

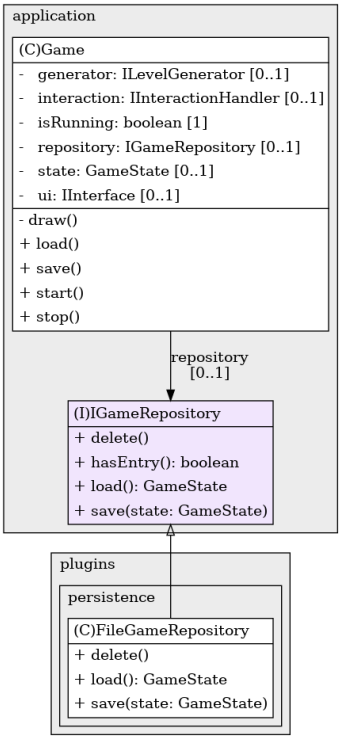

Figure 3.3: Analyse Dependency-Inversion-Principle: Positiv

3.3.2 Negativ-Beispiel

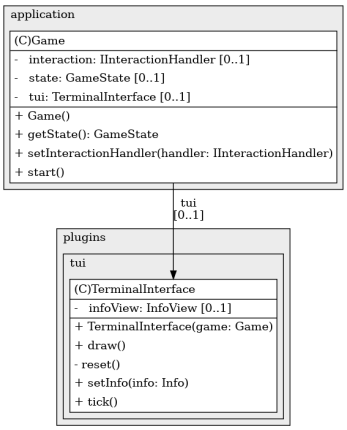

Figure 3.4: Analyse Dependency-Inversion-Principle: Negativ

4 Weitere Prinzipien

4.1 Analyse GRASP: Geringe Kopplung

Positiv-Beispiel

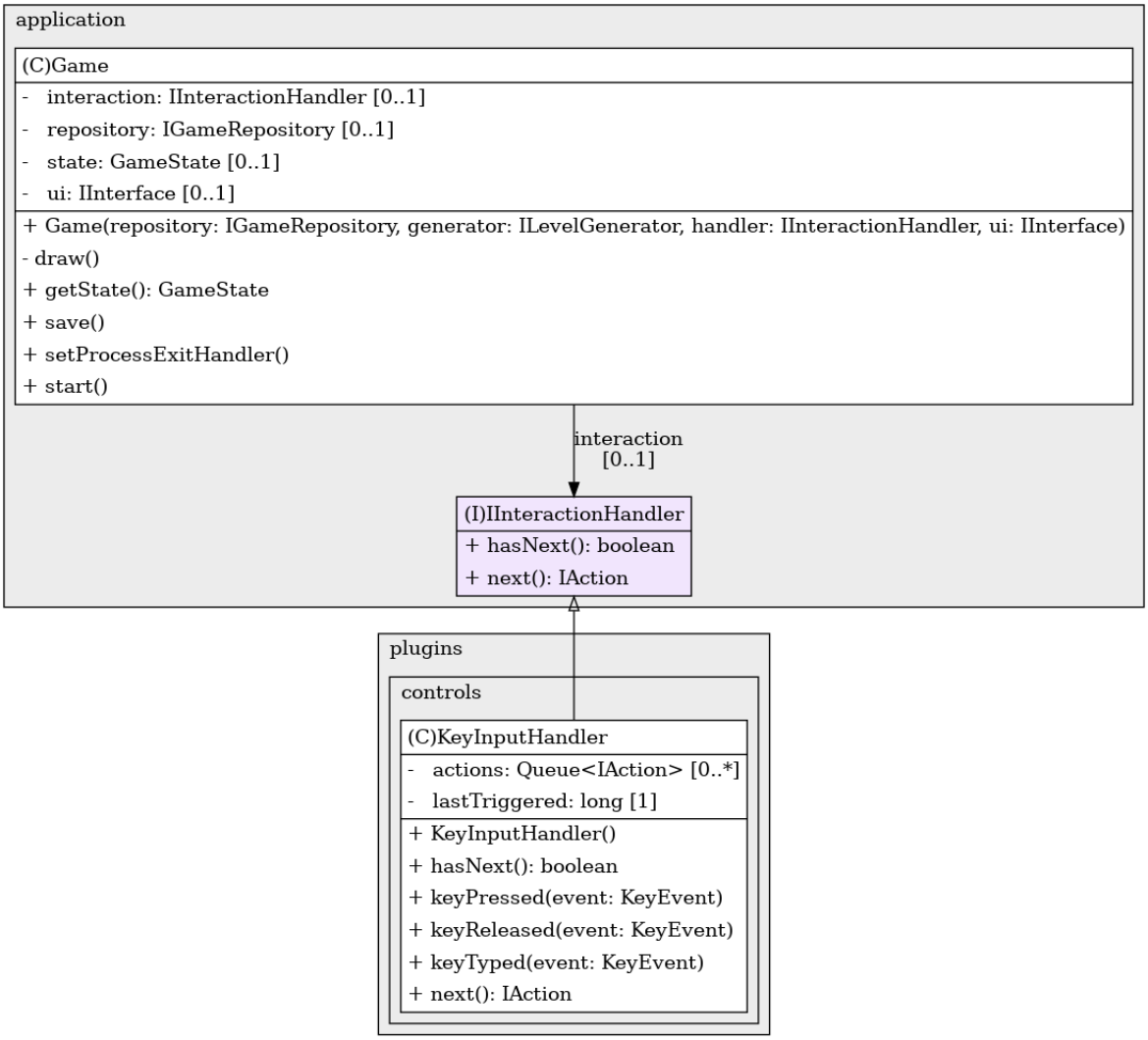

Figure 4.1: Analyse GRASP: Geringe Kopplung / Positiv

Negativ-Beispiel

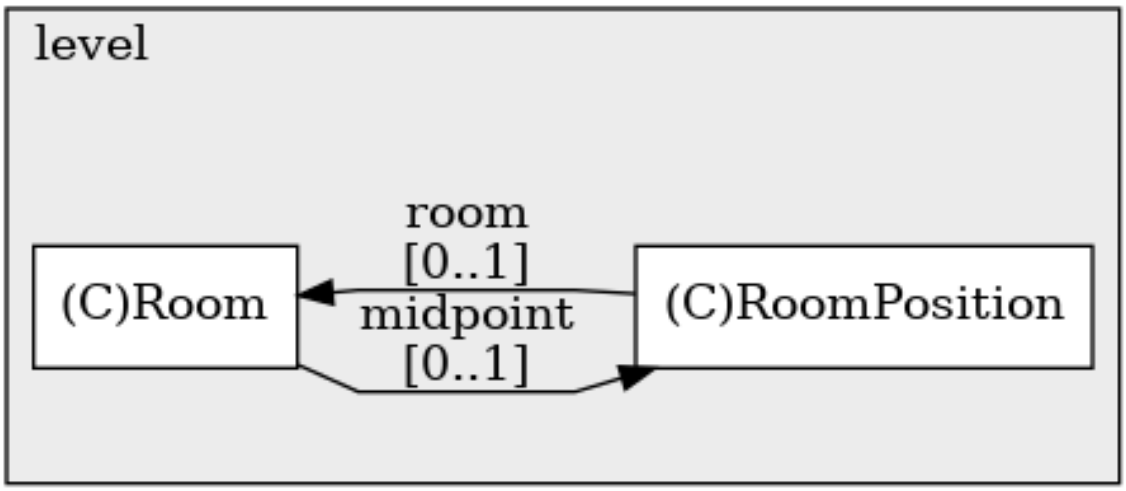

Figure 4.2: Analyse GRASP: Geringe Kopplung / Negativ

4.2 Analyse GRASP: Hohe Kohäsion

4.3 Don't Repeat Yourself (DRY)

5 Unit Tests

5.1 10 Unit Tests

Unit Test	**Beschreibung**
1. VectorTest#dotTest	Testet *dotTest*-Funktion der Klasse *Vector* darauf, ob sie korrekt Skalarprodukte berechnet
2. VectorTest#lengthTest	Testet *length*-Funktion der Klasse *Vector* darauf, ob sie korrekt Vektorlängen berechnet
3. VectorTest#getClockwiseAngleFromTest	Testet *getClockwiseAngleFrom*-Funktion der Klasse *Vector* darauf, ob sie korrekt Winkel zwischen zwei Vektoren berechnet
4. NumericTest#clampTest	Testet *clamp*-Funktion der Klasse *Numeric* darauf, ob sie korrekt Werte in einem gegebenen Intervall einschließt
5. EntityTest#getPreferredMovement DirectionTest	Testet *getPreferredMovementDirection*-Funktion der Klasse *Enemy* darauf, ob sie korrekt eine Bewegung aufgrund der Position des Spielers wählt
6. RoomGridTest#fitsTest	Testet *fits*-Funktion der Klasse *RoomGrid* darauf, ob sie korrekt feststellen kann, ob ein Raum noch auf die Spielkarte passt
7. RoomPositionTest#getMaxDistanceAlongAnyAxisTest	Testet *getMaxDistanceAlongAnyAxis*-Funktion der Klasse *RoomPosition* darauf, ob sie korrekt die maximale Entfernung entlang einer von beiden Axen von einem Punkt zum anderen berechnen kann
8. CollectionSelectorTest#randomSubset Test	Testet *random*-Funktion der Klasse *CollectionSelector* darauf, ob sie korrekt Teilmengen der Auswahl selektiert

Table 5.1: Unit Tests mit Beschreibung I

Unit Test	**Beschreibung**
9. GameStateTest#movementTest	Testet die Klasse *Game* mithilfe des *KeyInputHandlerFake*, ob der Spielzustand nach beliebigen Eingabesequenzen korrekt ist
10. RepositoryTest#loadAndSaveRobustnessTest	Testet die Robustheit, insbesondere das *Exception*-Verhalten der *load()*- und *save()*-Funktionen der Klasse *Game* mithilfe des *GameRepositoryFake*

Table 5.2: Unit Tests mit Beschreibung II

5.2 ATRIP: Automatic

Der Begriff *Automatic* bezieht sich auf die automatische Ausführung von Unit Tests. Dies kann mit herkömmlichen Test-Frameworks wie etwa *JUnit5* erreicht werden.

Dabei können Methoden eigens zum Testen erstellt werden, welche mit *@Test* annotiert werden. Innerhalb der IDE (z.B. IntelliJ) können sie dann über einen Befehl oder Knopfdruck einzeln oder im Gesamten ausgeführt werden.

Die Tests sind so konzipiert, dass sie nicht andere Durchläufe beeinflussen. Sie sind also unabhängig von einander (*Independence*) und wiederholbar (*Repeatability*). Das führt zu nachvollziehbaren Ergebnissen. Zudem besitzen die Tests keine externen Abhängigkeiten.

Zum Schluss gibt JUnit eine Zusammenfassung aus. Diese beschreibt die erfolgreichen und fehlgeschlagenen Tests.

5.3 ATRIP: Thorough

Positiv-Beispiel

Der nachfolgende Code-Ausschnitt zeigt das Positiv-Beispiel zum *Thorough*-Aspekt. Hierbei wird die *getClockwiseAngleFromTest()*-Funktion der Klasse *Vector* getestet, welche den Winkel im Uhrzeigersinn zwischen zwei Vektoren berechnet. Dabei gibt es einige verschiedene Möglichkeiten. Diese hängen unter anderem von der Definitionsreihenfolge der beiden involvierten Vektoren ab. Dabei beginnt die Winkelmessung beim zweiten angegebenen Vektor und läuft bis zum ersten Vektor im Uhrzeigersinn. Es gibt zudem auch Fälle, in denen die Berechnung fehlschlägt, etwa dann wenn ein Vektor der Nullvektor ist.

Der Test ist *Thorough*, da er als *ParameterizedTest* umgesetzt ist und die Eingabe-Quelle *getClockwiseAngleFromArguments()* eine gründliche Stichprobe möglicher Fälle bereitstellt. Dabei werden etwa Standardfälle in einigen Ausführungen und Definitionsreihenfolgen ausprobiert, sowie unübliche Fälle, Edge Cases und pathologische Fälle.

```
1     @ParameterizedTest
2     @MethodSource("getClockwiseAngleFromArguments")
3     void getClockwiseAngleFromTest(Vector to, Vector from, double ↙
          ↳ angle) {
4         assertEquals(angle, to.getClockwiseAngleFrom(from));
5     }
6
7     private static List<Arguments> getClockwiseAngleFromArguments() {
8         return List.of(
9                 Arguments.of(Vector.UP, Vector.UP, 0D),
10                Arguments.of(Vector.RIGHT, Vector.UP, 90D),
11                Arguments.of(Vector.UP, Vector.RIGHT, 270D),
12                Arguments.of(Vector.LEFT, Vector.UP, 270D),
13                Arguments.of(Vector.UP, Vector.LEFT, 90D),
14                Arguments.of(Vector.DOWN, Vector.UP, 180D),
15                Arguments.of(Vector.UP, Vector.DOWN, 180D),
16
17                Arguments.of(new Vector(1, 2), new Vector(1, 2), 0D),
18                Arguments.of(new Vector(1, 1), new Vector(1, -1), 90D↙
                      ↳ ),
```

```
19                  Arguments.of(new Vector(-1, 1), new Vector(1, -1), ↙
                        ↳ 180D),
20
21                  Arguments.of(new Vector(1, 1), Vector.RIGHT, 45D),
22                  Arguments.of(new Vector(10000, 0), Vector.UP, 90D),
23
24                  Arguments.of(new Vector(0, 0), Vector.UP, Double.↙
                        ↳ NEGATIVE_INFINITY),
25                  Arguments.of(new Vector(0, 0), Vector.RIGHT, Double.↙
                        ↳ NEGATIVE_INFINITY)
26          );
27      }
```

Code listing 5.1: ATRIP: Thorough / Positiv

Negativ-Beispiel

Der nachfolgende Code-Ausschnitt zeigt das Negativ-Beispiel zum *Thorough*-Aspekt. Hierbei wird die *clamp()*-Funktion der Klasse *Numeric* getestet, welche einen Wert in einem gegebenen Intervall einschließt. Der Wert liegt entweder im Intervall oder nimmt den Wert der jeweilig überschrittenen Intervallgrenze an.

Der Test ist nicht *Thorough*, da er nur einen einzigen Fall prüft. Dieser ist zudem trivial. Der Test hat keine Aussagekraft über die Robustheit der *clamp()*-Funktion.

```
1       @Test
2       void clampTest() {
3           assertEquals(5, Numeric.clamp(0, 10, 5));
4       }
```

Code listing 5.2: ATRIP: Thorough / Negativ

5.4 ATRIP: Professional

Positiv-Beispiel

Der nachfolgende Code-Ausschnitt zeigt das Positiv-Beispiel zum *Professional*-Aspekt. Hierbei wird die *random()*-Funktion der Klasse *CollectionSelector* getestet, welche aus einer vorgegebenen Collection versucht *amount* Elemente zufällig zu wählen. Dabei sind die gewählten Elemente unterschiedlich. Hat die Collection zu wenige Elemente, so wird die komplette Collection zurückgegeben.

```
1   private static final Random CONTROLLED_RANDOMNESS = new Random(69420);
2   private static final List<Integer> COLLECTION = Arrays.asList(1, 2, 3, 4, 5);
3
4   @ParameterizedTest
5   @MethodSource("randomSubsetArguments")
6   void randomSubsetTest(CollectionSelector<Integer> selector, int amount, int expectedSubsetSize) {
7       Set<Integer> selection = new HashSet<>(selector.random(amount).get());
8       assertEquals(expectedSubsetSize, selection.size());
9       assertTrue(selector.get().containsAll(selection));
10  }
11
12  private static List<Arguments> randomSubsetArguments() {
13      return List.of(
14              Arguments.of(new CollectionSelector<>(COLLECTION, CONTROLLED_RANDOMNESS), -1, 0),
15              Arguments.of(new CollectionSelector<>(COLLECTION, CONTROLLED_RANDOMNESS), 0, 0),
16              Arguments.of(new CollectionSelector<>(COLLECTION, CONTROLLED_RANDOMNESS), 1, 1),
17              Arguments.of(new CollectionSelector<>(COLLECTION, CONTROLLED_RANDOMNESS), 2, 2),
18              Arguments.of(new CollectionSelector<>(COLLECTION, CONTROLLED_RANDOMNESS), 3, 3),
19              Arguments.of(new CollectionSelector<>(COLLECTION, CONTROLLED_RANDOMNESS), 4, 4),
```

```
20                  Arguments.of(new CollectionSelector<>(COLLECTION, ↙
                        ↳ CONTROLLED_RANDOMNESS), 5, 5),
21                  Arguments.of(new CollectionSelector<>(COLLECTION, ↙
                        ↳ CONTROLLED_RANDOMNESS), 6, 5),
22                  Arguments.of(new CollectionSelector<>(COLLECTION, ↙
                        ↳ CONTROLLED_RANDOMNESS), 10000, 5)
23          );
24      }
```

Code listing 5.3: ATRIP: Professional / Positiv

Sie ist aus folgenden Gründen professionell:

1. Es existieren keine Code-Duplikationen im Test selbst. Alle Fälle sind in einer Eingabe-Quelle namens *randomSubsetArguments* zusammengefasst. Dadurch kann nachvollzogen werden, an welchen Stellen der Test fehlschlägt. Zudem ist die Logik sehr viel einfacher zu erkennen und nachzuvollziehen.

2. Durch den *ParameterizedTest* ist zudem einfacher zu erkennen, ob alle wichtigen Fälle abgedeckt sind und die Funktion gründlich untersucht wurde.

3. Durch fehlende Duplikation kann die Test-Funktion sich auf die grundlegende Mechanik beschränken. Variablennamen bleiben dadurch verständlich, weil keine Alternativnamen für Objekte gleicher Art gefunden werden müssen.

4. Kurze aber effektive Tests vermindern das Risiko für *Code Smells* oder Fehler im Code, wodurch die Robustheit und Wartbarkeit des Tests steigt.

5. Es wird eine nicht-triviale Kernfunktion der Domäne getestet. Der Test sorgt damit für eine sinnvolle Erhöhung der Coverage und stellt sicher, dass wichtige Bestandteile für das Funktionieren der Gesamtanwendung selbst funktional sind.

6. Der *ParameterizedTest* erfüllt die Eigenschaft *Thorough*.

7. Die zu prüfende Funktion benötigt *Randomness*. Für nachvollziehbare Ergebnisse wird der entsprechende Generator mit einem Seed festgesetzt.

Negativ-Beispiel

Der nachfolgende Code-Ausschnitt zeigt das Negativ-Beispiel zum *Professional*-Aspekt. Hierbei wird die *dot()*-Funktion der Klasse *Vector* getestet, welche das Skalarprodukt zwischen zwei Vektoren berechnet.

```
1    @Test
2    void dotTest() {
3        Vector posA = new Vector(1, 2);
4        Vector posB = new Vector(3, 4);
5        Vector negC = new Vector(-1, -2);
6        Vector negD = new Vector(-3, -4);
7        double dot = posA.dot(posB);
8
9        assertEquals(5D, posA.dot(new Vector(1, 2)));
10       assertEquals(dot, posA.dot(posB));
11
12       Vector e = new Vector(-1, -2);
13       assertEquals(11, e.dot(negD));
14
15       // [Edge Cases]
16       Vector zero = new Vector(0, 0);
17       assertEquals(0, posA.dot(zero));
18       assertEquals(0, posB.dot(zero));
19       assertEquals(0, negD.dot(zero));
20   }
```

Code listing 5.4: ATRIP: Professional / Negativ

Sie ist aus folgenden Gründen nicht professionell:

1. Alle Test-Fälle sind in der *dotTest()*-Methode beschrieben. Das erschwert zum einen die Wartung, zum anderen ist dadurch schlecht zu erkennen, an welcher Stelle genau ein Test fehlgeschlagen ist. Die Beispiele sollten daher stattdessen über einen *ParameterizedTest* umgesetzt werden.

2. Die Instruktionsblöcke greifen teilweise ineinander. Beispielsweise wird im ersten Block eine Variable benutzt, welche in den folgenden Blöcken genutzt wird, obwohl die Aufteilung eine logische Trennung suggeriert. Dadurch wird der Code unübersichtlicher.

3. Die Variablenbenennung ist nicht konsequent. Anfangs werden Vektoren bezüglich ihrer Ausrichtung benannt, letztlich nicht mehr.

4. Es existiert Code-Duplikation. Einmal wird *posA.dot(posB)* in die Variable *dot* geschrieben, danach in einem Test aber nochmals komplett ausgeschrieben. Das führt zu Unübersichtlichkeit. Vermutlich hat das auch zur Prüfung, ob *dot* sich selbst entspricht, geführt - ein trivialer Fall, welcher dem Test keine Aussagekraft liefert.

5. Zudem sind Randfälle (siehe unten) mehrfach behandelt, obwohl die entsprechenden Stichproben logisch keinen Unterschied aufweisen sollten. Das spricht entweder für fehlendes Wissen oder Nachlässigkeit. Letztlich bringen die überflüssigen Assertions dem Test keine zusätzliche Aussagekraft.

5.5 Code Coverage

Class	**Method**	**Line**
41.8% (28/67)	29.1% (118/405)	29.2% (317/1084)

Table 5.3: Overall Coverage Summary

Die Tabelle zeigt die Code-Coverage aller Tests über das gesamte Projekt, aufgeschlüsselt nach Klassen, Methoden und Zeilen. Die dargestellten Werte zeichnen eine schlechte Coverage ab. Um diese zu erhöhen, muss noch viel Arbeit investiert werden. Bisher hat sich die Arbeit aber hauptsächlich auf die Logik und die Design-Prinzipien beschränkt. Über die geforderten Unit Tests ist bis dato nicht viel passiert. Dabei ist eine hohe Coverage (häufig $> 90\%$) das Ziel. Sie verspricht, dass durch entsprechende Tests schnell Bugs und neue Fehler identifiziert werden. Zudem untermauert eine sehr gute Coverage bei gleichzeitig sehr guter Test-Qualität entsprechend auch die hohe Qualität des Produktes.

In diesem Fall gibt es einzelne Bereiche, die recht gute Coverage erhalten haben. Dazu zählen die *abstraction* Schicht, sowie einige Teile in der *domain* und der *application* Schicht. Aber auch hier gibt es merklich Lücken. Die *plugin* Schicht fällt der Gesamt-Coverage jedoch besonders schwer in's Gewicht. Hier sind lediglich die Controls und die Persistenz mäßig abgedeckt. Dabei existieren in dieser Schicht allerdings noch viele Klassen und

Funktionen, welche sich mit der Anzeige des Spiels in der Konsole beschäftigen. Diese haben gar keine Coverage abbekommen.

5.6 Fakes und Mocks

5.6.1 GameRepositoryFake

Die nachfolgende Abbildung zeigt das UML-Diagramm des Fakes *GameRepositoryFake*. Er ist realisiert als eine weitere Implementation des *IGameRepository*-Interfaces. Dieses wird von der Klasse *Game* genutzt, um über eine Schnittstelle mit einer Persistenz zu interagieren. Dabei können prinzipiell unerwünschte Fehler auftreten, vor allem bei der Interaktion mit einem Dateisystem. Diese müssen zwangsläufig behandelt werden, um ein sicheres Programm zu gewährleisten.

Es ist allerdings nicht zwangsläufig einfach gezielt Fehlermeldungen zu erzeugen, um die Abdeckung der Fehlerbehandlung festzustellen. Daher wird auch einen Fake zurückgegriffen, welcher entsprechend seiner Konfiguration Fehlermeldungen erzeugt und so eine fehlerhafte Interaktion, z.B. mit einem Dateisystem simuliert.

So wurden beispielsweise die *load()*- und *save()*- Funktion ausgewählt, um gefaked zu werden. Dafür gibt man entsprechend im Test an, ob jeweilig *load()* und *save()* fehlschlagen sollen, um die Robustheit der Benutzung sicherzustellen.

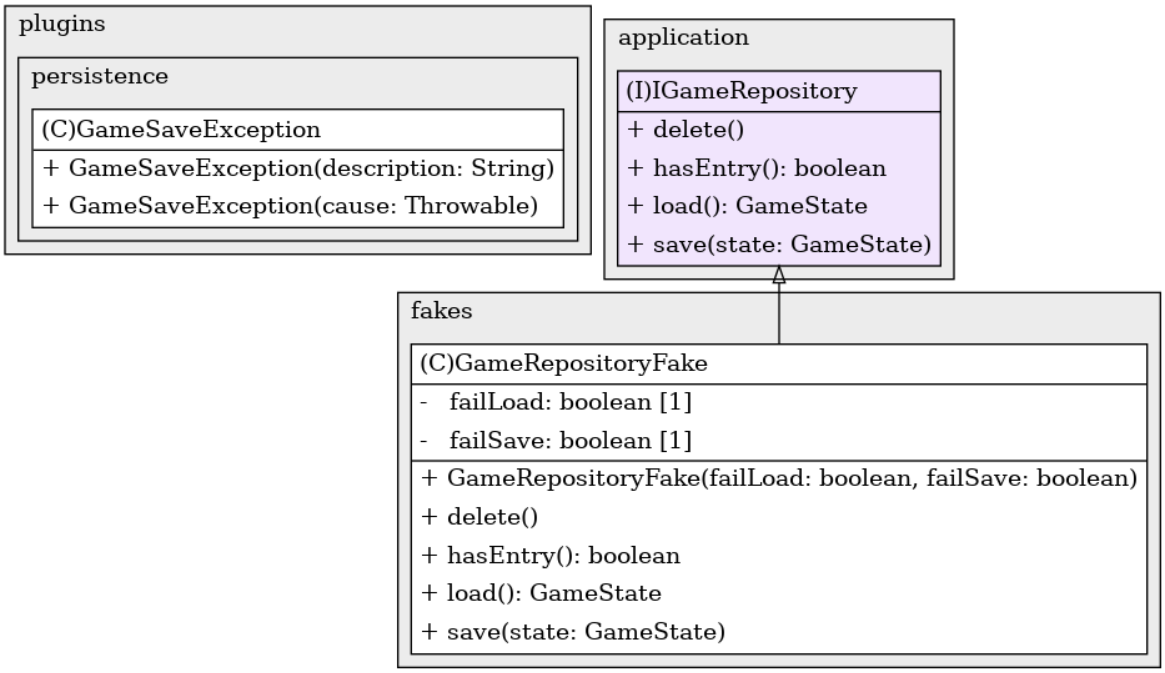

Figure 5.1: KeyInputHandlerFake

5.6.2 KeyInputHandlerFake

Die nachfolgende Abbildung zeigt das UML-Diagramm des Fakes *KeyInputHandler-Fake*. Er ist realisiert als eine weitere Implementation des *IInteractionHandler*-Interfaces. Dieses wird von der Klasse *Game* genutzt, um über eine Schnittstelle beliebige Aktionen entgegenzunehmen und zu verarbeiten. Um die korrekte Verarbeitung der Aktionen sicherzustellen, wird der Spielzustand nach einer Aktionssequenz geprüft. Dies passiert konkret im Test *GameStateTest#movementTest*, indem die Auswirkung von verschiedenen Aktionen auf die Bewegung und Position des Spielers geprüft werden.

Im Normalbetrieb werden die Aktionen durch den KeyInputHandler als Reaktion auf Tasteneingaben erzeugt. Dies ist jedoch beim Testen nicht möglich. Der *KeyInputHandler-Fake* wird daher benötigt, um reale Tasteneingaben zu simulieren und Änderungen im Spielzustand hervorzurufen. Dafür wird für jeden Testlauf eine feste Aktionssequenz definiert.

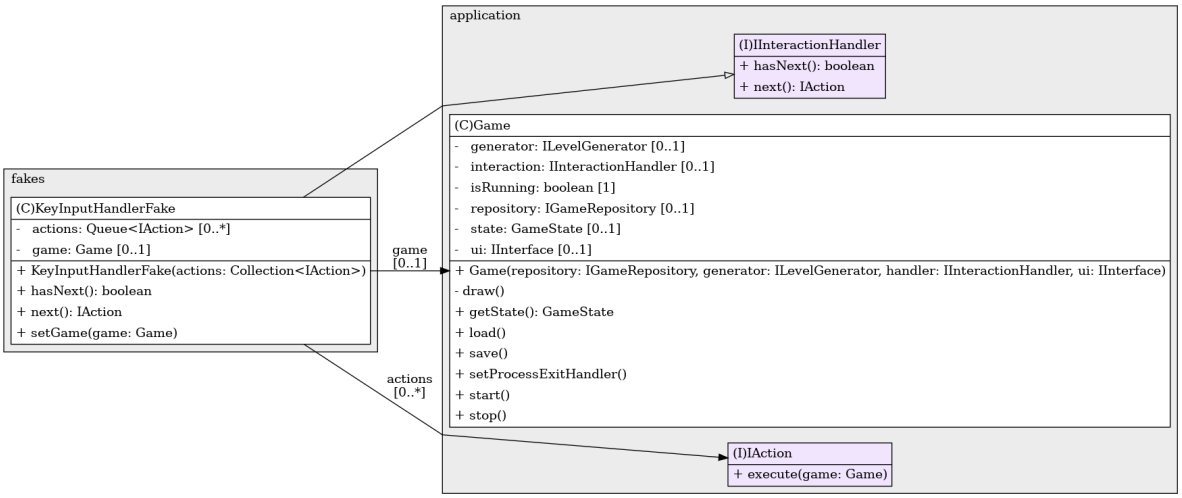

Figure 5.2: KeyInputHandlerFake

6 Domain Driven Design

6.1 Ubiquitous Language (UL)

Item

Bedeutung: Ein *Item* ist ein nicht-belebtes Objekt, welches einen konkreten Gegendstand im Spiel wie etwa Waffen, Rüstungen oder Münzen repräsentiert. Mit ihm kann interagiert werden. Ein Item erfüllt verschiedene Funktionen abhängig vom Typ, wie etwa Heilung, Stärkung des Spielers oder Beeinflussung des Spielstandes.

Begründung: Die Verwendung des Begriffes in der UL erlaubt eine einheitliche Kommunikation zwischen den Entwicklern und Designern. Hinzu kommt, dass der Begriff grundlegend in der Spieleentwicklung ist. Somit lässt sich leichter festlegen, wie Items designed und logisch in das Spielgeschehen integriert werden und wie diese die Spielerfahrung formen.

Room

Bedeutung: Der Begriff *Room* definiert einen abgegrenzten Bereich der Spielwelt. Jeder Raum stellt für den Spieler eine neue Herausforderung dar. In ihm befinden sich neue Gegner und neue Items. Besonders ersichtlich wird die Abgrenzung der Räume, wenn sie abgeschlossen sind, weil in ihnen noch Gegner leben.

Begründung: Der Begriff Room in der UL ist essentiell wichtig, da Entwickler und Designer sich absprechen müssen, um den technischen und stilistischen Entwurf für abgegrenze Spielbereiche zu schaffen. Für Entwickler ist der Begriff besonders prägend im Bereich der Welt- Generierung. Für Designer ist er wichtig, um zwischen verschiedenen Typen zu unterscheiden, welche unterschiedlich ausgebildet werden.

Level

Bedeutung: Das *Level* repräsentiert eine abgegrenzte Spielstufe. Es gilt dieses komplett zu absolvieren, bis man den End-Raum des Levels gefunden hat. Es stellt eine Einheit einzelner mit einander verbundener Räume dar. Levels haben Fortschrittsmechanismen bezogen auf die Schwierigkeit der Räume.

Begründung: Der Begriff *Level* ist in der UL sehr wichtig, da es einen Strukturbegriff darstellt. Er gliedert die Spielwelt weiter als übergeordneter Einheit über den Räumen. Daher ist der Begriff relevant für Entwickler im Bezug auf die Welt-Generierung und für Designer im Bezug auf die Schwierigkeitsentwicklung und den Stil.

Player

Bedeutung: Der *Player* stellt die Spielfigur dar, durch welche der Spielende mit der virtuellen Umgebung interagieren kann. Er stellt ein sterbliches Lebewesen dar, welches den Umgang mit der Spielwelt maßgeblich beeinflusst: Das oberste Ziel ist überleben. Der Spieler kann Items aufsammeln, Gegnern schaden zufügen, von ihnen Schaden nehmen und durch die Räume gehen.

Begründung: Die Verwendung des Begriffs *Player* schafft Klarheit in der Abgrenzung zu anderen virtuellen Lebewesen wie den Gegnern. Der Spieler hat eine gesonderte Rolle und benötigt daher wesentlichen Mehraufwand in der Entwicklung und Design. Seine Interaktionen prägen das Spielerlebnis. Technisch gesehen teilt sich ein Spieler Eigenschaften mit anderen Lebewesen. Daher ist eine klare Definition der Spielfigur unerlässlich für die Entwicklung selbst, um die herausstellenden Merkmale klar abzugrenzen.

6.2 Entities

6.3 Value Objects

Die nachfolgende Abbildung zeigt das UML-Diagram zur Klasse *Position*, welche ein *Value Object* darstellt. Instanzen dieser Klasse sind *immutable*. Die Felder sind *final* und

es existieren lediglich die Getter *x()* und *y()*. Damit lässt sich ein Objekt dieser Klasse einmalig mit Positions-Werten initialisieren.

Der Vorteil ist, dass Positionen - welche einen wichtigen Bestandteil der Domänenlogik darstellen - nicht einfach geändert werden können. Es muss bewusst eine Ersetzung vorgenommen werden. Es entstehen somit keine unerwarteten Seiteneffekte.

Die Gleichheit zweiter Instanzen wird über die überschriebenen *equals()*- und *hashCode()*-Funktionen festgestellt.

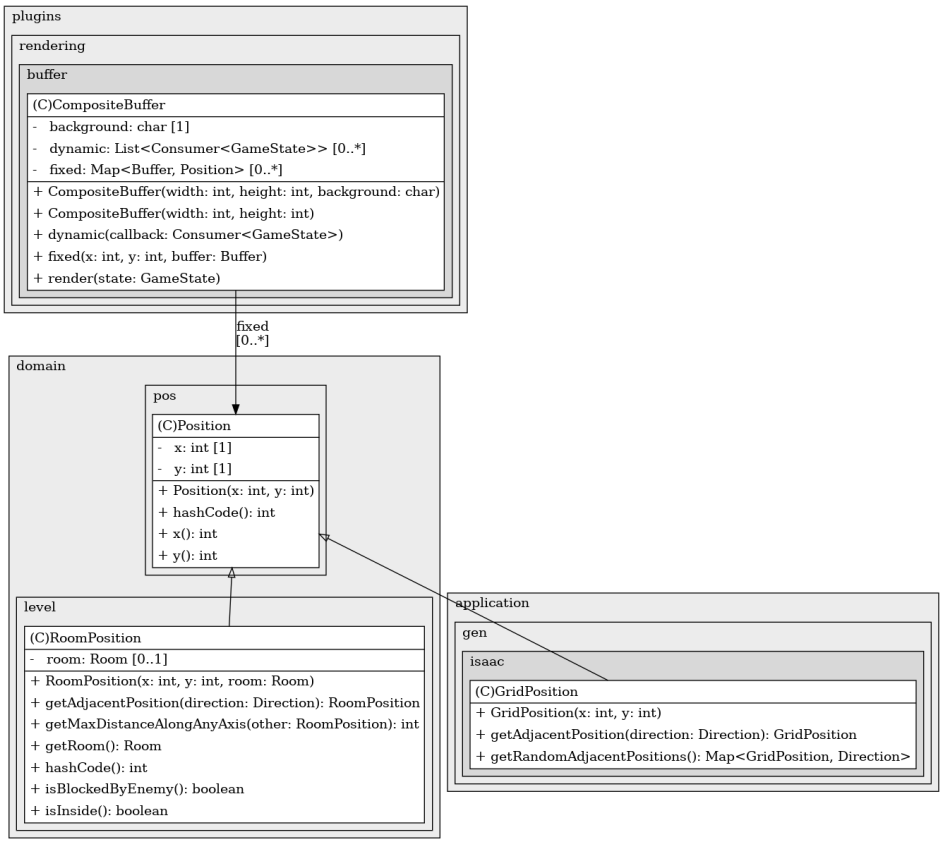

Figure 6.1: Value Object *Position*

6.4 Repositories

6.5 Aggregates

7 Refactoring

7.1 Code Smells

```
1   @Override
2   public void move(Direction direction) {
3       RoomPosition target = this.getPosition().getAdjacentPosition(↙
            ↳ direction);
4       if (target.isValid() && target.isFree()) {
5           if (this.getRoom() instanceof BossRoom boss) {
6               RoomPosition trapdoor = boss.getTrapdoorPosition();
7
8               if (target.equals(trapdoor) && this.getRoom().↙
                    ↳ isCleared()) {
9                   this.game.nextLevel();
10              }
11          }
12
13          this.setPosition(target);
14
15          Item underneath = this.getRoom().getItemUnderneath(this.↙
                ↳ getPosition());
16          // this.game.setItemUnderneath(underneath);
17      } else {
18          if (!this.getRoom().isCleared()) {
19              return;
20          }
21
22          this.getRoom().getDoors().forEach(door -> {
23              if (door.getRoomPosition().equals(target)) {
24                  this.game.getLevel().enter(door);
25              }
26          });
```

```
27          }
28      }
```

Code listing 7.1: Code Smell I (Vorher)

```
1       @Override
2       public void move(Direction direction) {
3           RoomPosition target = this.getPosition().getAdjacentPosition(↙
                ↳ direction);
4           if (target.isValid() && target.isFree()) {
5               if (this.getRoom().isBossRoom()) {
6                   RoomPosition trapdoor = this.getRoom().↙
                        ↳ getTrapdoorPosition();
7                   if (target.equals(trapdoor) && this.getRoom().↙
                        ↳ isCleared()) {
8                       this.game.nextLevel();
9                   }
10              }
11
12              this.setPosition(target);
13
14              Item underneath = this.getRoom().getItemAt(this.↙
                    ↳ getPosition());
15              // this.game.setItemUnderneath(underneath);
16          } else {
17              if (!this.getRoom().isCleared()) {
18                  return;
19              }
20
21              Door door = this.getRoom().getDoorAt(target);
22              if (door != null) {
23                  this.game.getLevel().enter(door);
24              }
25          }
26      }
```

Code listing 7.2: Code Smell I (Nachher)

7.2 Refactorings

8 Entwurfsmuster

8.1 Entwurfsmuster: Erbauer

Die nachfolgende Abbildung zeigt ein vereinfachtes UML-Diagramm der Klasse *IsaacLevelGenerator*, welches das Interface *ILevelGenerator* implementiert. Das Interface dient als Abstraktion eines beliebigen Level-Generators, der *IsaacLevelGenerator* ist eine konkrete Implementation.

Es handelt sich hierbei um das Entwurfsmuster **Erbauer**. In diesem Fall ist *IsaacLevelGenerator* ein konkreter Erbauer. Durch die Verwendung des *ILevelGenerator* Interfaces kann die Konstruktionsmechanik für ein Level jedoch beliebig ausgetauscht werden.

Die *next()*-Funktion des *IsaacLevelGenerator* beschreibt eine mehrschrittige komplexe Logik zur Erzeugung von *Level*-Objekten. Sie kann genutzt werden, um immer wieder gleiche Objekte zu generieren. Die Verwendung eines *Erbauers* erlaubt die Trennung zwischen der komplexen Erstellung eines Objektes und seiner Klassenrepräsentation. Zudem versteckt der Generator die interne Repräsentation während der Level-Erstellung. Letztlich wird nur das fertige Produkt zur Verfügung gestellt.

Die Verwendung des *Erbauer*-Musters erhöht die Modularität durch Austauschbarkeit und die Verwendung eines Interfaces. Zudem vermindert es die Komplexität, da unterschiedliche komplexe Erzeugungsprozesse jeweilig in einem *Erbauer* gebunden sind. Alles zusammen erlaubt auch einfache Erweiterbarkeit, da immer neue *Erbauer* anstelle des Interfaces gesetzt werden können.

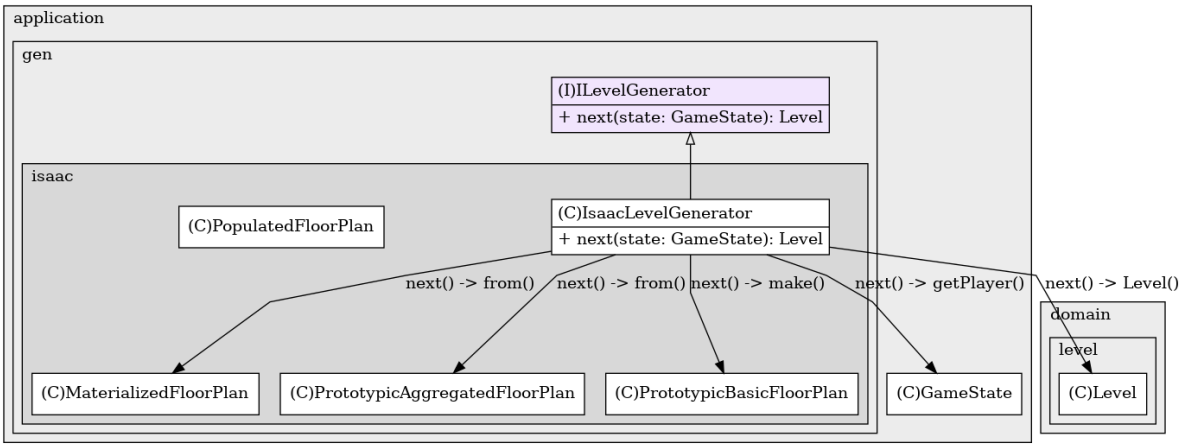

Figure 8.1: Entwurfsmuster: Erbauer

8.2 Entwurfsmuster: Kompositum

Die nachfolgende Abbildung zeigt das UML-Diagramm der Klassen *Buffer* und *CompositeBuffer*, wobei letztere von ersterer erbt. Beide Klassen finden Verwendung in der Spielanzeige auf der Konsole. Dabei werden statische *Sprites* mithilfe der Klasse *Buffer* beschrieben, komplexere Anzeigen, bei denen dynamische Änderungen auftreten können mithilfe der Klasse *CompositeBuffer*.

Es handelt sich hierbei um das Entwurfsmuster **Kompositum**. Objekte der beiden Klassen lassen sich gemeinsam zu einer Baum-Struktur zusammensetzen. Beide Typen sind gleichwertige Komponenten. Die *Buffer* sind dabei Blätter bzw. einfache Elemente, wohingegen *CompositeBuffer* wie Container funktionieren und die Kinder verwalten.

In der Gesamtheit verhalten sie sich wie eine Einheit. Sie teilen sich alle Funktionen der zugrundeliegenden Klasse *Buffer*. So etwa die Funktion *toString()*, welche den Buffer-Inhalt in einen String für die Konsole überführt oder die Funktion *render()*, welche rekursiv den Baum vom Wurzel-Element ausgehend traversiert und entsprechende Updates veranlasst.

In der Abbildung ist zu sehen, dass die Klasse *CompositeBuffer* genutzt wird, um Anzeigen (*Views*) zu erstellen. Die bedeutenste Klasse ist dabei *TerminalInterface*, da sie die Hauptanzeige modelliert. Die Klasse *CompositeBuffer* bietet jedoch, im Gegensatz

zum *Buffer*, zur Einfachheit und Effizienz die Möglichkeit zwischen fixierten *fixed()* und dynamischen *dynamic()* Elementen zu unterscheiden.

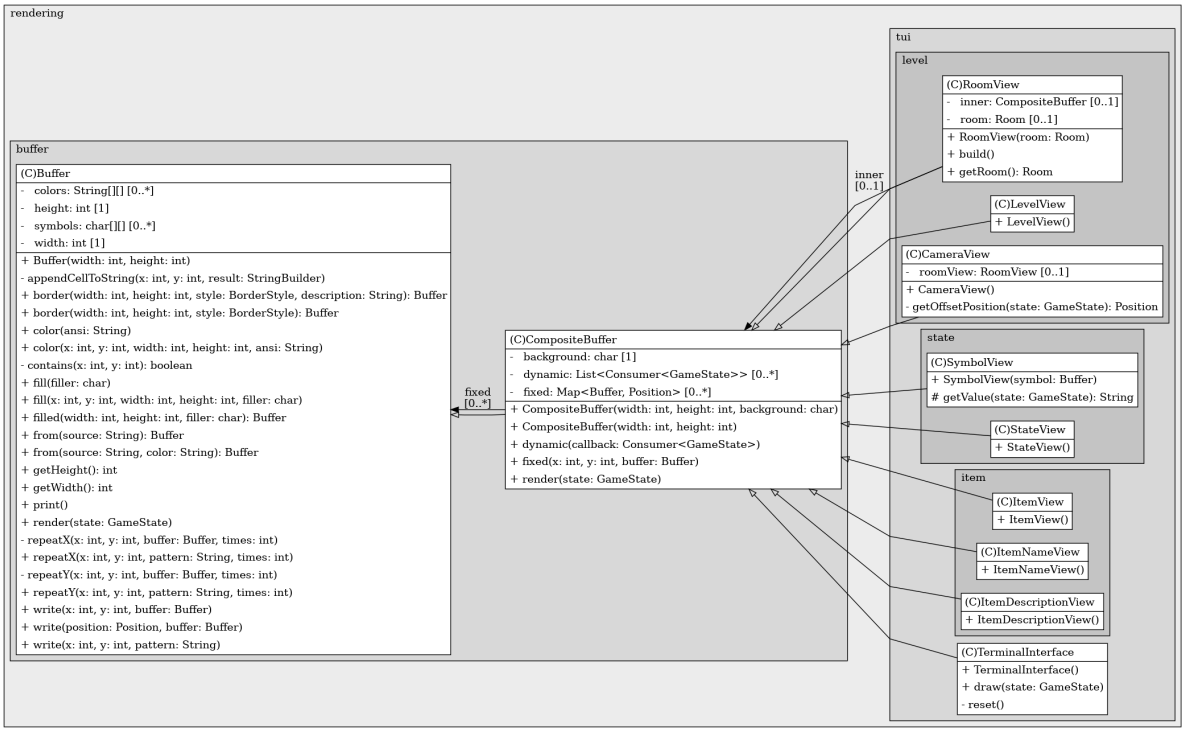

Figure 8.2: Entwurfsmuster: Kompositum