

**Praxisprojekt im Rahmen von  
Advanced Software Engineering**

**Technische Dokumentation zum Programmentwurf**

**Weeping Snake**

Duale Hochschule Baden-Württemberg Karlsruhe  
Fakultät Technik

Studiengang Informatik – Informatik

von

Silas Mario Schnurr

26.05.2021



**Kurs:** TINF18B5

**Dozent:**  Herr Maurice Müller

**Inhaltsverzeichnis**

[Abkürzungsverzeichnis / Glossar IV](#_Toc72913410)

[Verzeichnisse V](#_Toc72913411)

[1 Projektbeschreibung 1](#_Toc72913412)

[1.1 Funktionsumfang 2](#_Toc72913413)

[1.2 Code-Struktur 3](#_Toc72913414)

[1.2.1 LOC 4](#_Toc72913415)

[1.3 Kompilieren, testen & ausführen 5](#_Toc72913416)

[2 Domain Driven Design 6](#_Toc72913417)

[2.1 Ubiquitous Language 6](#_Toc72913418)

[2.2 Kontext 8](#_Toc72913419)

[2.3 Taktisches Domain Driven Design 9](#_Toc72913420)

[2.3.1 Entities 9](#_Toc72913421)

[2.3.2 Value Objects 11](#_Toc72913422)

[2.3.3 Aggregates 12](#_Toc72913423)

[2.3.4 Repositories 12](#_Toc72913424)

[3 Programming Principles 14](#_Toc72913425)

[3.1 SOLID 14](#_Toc72913426)

[3.2 GRASP 14](#_Toc72913427)

[3.3 DRY 14](#_Toc72913428)

[4 Entwurfsmuster 15](#_Toc72913429)

[5 Clean Architecture 20](#_Toc72913430)

[6 Legacy Code 23](#_Toc72913431)

[7 Refactoring 24](#_Toc72913432)

[8 Unit Tests 25](#_Toc72913433)

[8.1 Initiale Code Coverage 25](#_Toc72913434)

[8.2 Konzept 26](#_Toc72913435)

[8.3 Mock-Objekte 28](#_Toc72913436)

[8.3.1 MockCoordianteSystem 28](#_Toc72913437)

[8.3.2 MockPlayer 29](#_Toc72913438)

[8.3.3 MockGame 31](#_Toc72913439)

[8.4 ATRIP-Regeln 32](#_Toc72913440)

[9 Zusatz: API-Design 33](#_Toc72913441)

[9.1 API 33](#_Toc72913442)

[9.2 Analyse 33](#_Toc72913443)

# Abkürzungsverzeichnis / Glossar

**API** Application Programming Interface, Programmierschnittstelle

**UL** Ubiquitous Language

**LOC** Lines of Code, die genaue Anzahl von Zeilen im Quellcode. Niedrigere Werte sind besser.

**CLI** Command-line interface, Kommandozeile

# Verzeichnisse

[Abbildung 1: Code-Struktur in UML Notation 3](#_Toc72683982)

[Abbildung 2: Problemdomäne 6](#_Toc72683983)

[Abbildung 3: UML-Klassendiagramm der Entities 10](#_Toc72683984)

[Tabelle 1: LOC vor Refactorings 4](#_Toc72683985)

[Tabelle 2: Getting Started 5](#_Toc72683986)

[Tabelle 3: Ubiquitous Language für die Kerndomäne 7](#_Toc72683987)

[Tabelle 4: Ubiquitous Language für die generische Domäne 8](#_Toc72683988)

[Tabelle 5: Beziehungen zwischen Kontexten 8](#_Toc72683989)

[Tabelle 6: Value Objects 11](#_Toc72683990)

[Tabelle 7: Aggregates und Root Entities 12](#_Toc72683991)

[Quellcode 1: Highscore Mapping Code der Adapter Schicht 17](#_Toc72684386)

[Quellcode 2: Adapter-Interface für Client mit TypeScript-Technologie 18](#_Toc72684387)

# Projektbeschreibung

Das vorliegende Dokument ist die schriftliche technische Dokumentation zu dem Programmentwurf *Weeping Snake* im Rahmen von Advanced Software Engineering an der DHBW Karlsruhe.

Es wird die Programmiersprache C# mit dem .Net 5.0 Framework verwendet. Im Folgenden werden verschiedene Analysen durchgeführt, deren Ergebnisse aufgezeigt sowie daraus resultierende Änderungen begründet. Die Arbeit ist dabei nach verschiedenen Problembereichen untergliedert, welche getrennt voneinander betrachtet werden können.

Bei dem Spiel *Weeping Snake* steuern die Teilnehmer einen *Player* auf einem begrenzten Spielfeld (*Board*). Die Player scheiden aus dem Spiel aus, wenn sie das Spielfeld verlassen. Ziel des Spiels ist es, möglichst viele Punkte zu sammeln. Hierfür bekommen die Teilnehmer Punkte, wenn sie mit ihrem *Player* den *Player* eines gegnerischen Spielers schneiden. Um dies zu ermöglichen, wird die Navigation durch eine Richtungsänderung, Geschwindigkeitsänderungen und Sprünge ermöglicht (Aktionen können über mehrere Runden hinweg vorausgeplant werden). In dem Fall, dass ein *Player* sich selbst schneidet oder geschnitten wird, verliert er Punkte. Die Länge der „Schlange“, welche den *Player* repräsentiert, ist von der Geschwindigkeit abhängig, da der Pfad der letzten 5 Runden für Schnitte relevant ist.

## Funktionsumfang

Der Funktionsumfang des Projekts geht über das eigentliche Spiel hinaus. Neben dem Backend des Spiels, welches als Programmbibliothek (mit API) implementiert wird, existiert eine separate Web-API sowie ein offline Client (CLI), der über die Programmbibliothek Einzelspieler Spiele gegen Bots (mehr oder weniger klug) erlaubt.

Über den CLI-Client ist es möglich, das Spiel zu spielen. Hierbei gibt es die Möglichkeit, als Gast zu spielen oder mit einem Benutzeraccount. Durch die Verwendung eines Benutzeraccounts besteht der Vorteil, dass die Errungenschaften in einer Highscore liste angezeigt werden. Dieser Benutzeraccount lässt sich verwalten, sodass die E-Mail-Adresse oder das Passwort geändert werden können. Die eigentliche Durchführung des Spiels erfolgt ebenfalls in einer Konsole. Hierfür müssen die Aktionen über die Tastatur eingegeben werden. Dieser CLI-Client dient jedoch nur der Demonstration der Grundfunktionalitäten. Das Backend ist so aufgebaut, dass die Spielparameter konfigurierbar sind (Über eine Konfigurationsdatei) und dadurch z. B. Richtungsänderungen nicht immer im 90° Radius erfolgen müssen. Dadurch können sich Spieler nicht nur vertikal und horizontal, sondern in jede beliebige Richtung, abhängig von dem gewählten Radius bewegen. Ein weiterer Bestandteil ist das Logging, welches es ermöglicht Spielinformationen in einer \*.log Datei zu persistieren. Dieses Logging ist standardmäßig deaktiviert und kann über die Datei weepingsnake.config mit dem Eintrag IsLoggingEnabled=true aktiviert werden.

## Code-Struktur

Die 27 Klassen (inklusive geschachtelter Klassen und klassenähnlicher Konstrukte[[1]](#footnote-1)) von dem *Game*-Backend lassen sich in sechs Aufgabenbereiche unterteilen. In dem folgenden UML-Klassen-Diagramm sind die Einteilung der Klassen sowie deren Beziehungen aufgeführt. Der Detailgrad des UML-Klassen-Diagramms fällt in dieser Übersicht jedoch geringer aus, da die grobe Struktur verdeutlicht werden soll. Zusätzlich zu den sechs Aufgabenbereichen gibt es einige unterstützende Klassen, welche unter Utility zusammengefasst werden können. Diese werden an mehreren Stellen verwendet, sodass eine explizite Darstellung der Assoziationen zu Unübersichtlichkeit führen würde.



Abbildung 1: Code-Struktur in UML Notation

### LOC

Die Anzahl der Quellcodezeilen (LOC – Lines of Code) wird von der GitHub Codemetrik mit ca. 4.000 angegeben. Die strengere Analyse mithilfe der IDE Visual Studio Enterprise 2019 ergibt die folgenden Ergebnisse für die Berechnung der Codemetrik der relevanten Projekte.

|  |  |  |
| --- | --- | --- |
| **Projekt** | **Funktion** | **LOC** |
| Game  WeepingSnake.Game | Backend für die Verwaltung und Durchführung von *Game*s | 2219 |
| WebService  WeepingSnake.WebService | Schnittstelle (REST) für den Zugriff über das Internet | 290 |
| ConsoleClient  WeepingSnake.ConsoleClient | Offline-Client (CLI), um im Einzelspielermodus auf das *Game*-Backend zuzugreifen | 526 |

Tabelle 1: LOC vor Refactorings

Die Angabe der LOC bezieht sich auf den Quellcodestand vor der Umsetzung der Refactorings, welche in diesem Dokument beschrieben werden. Dieser Quellcodestand ist auf dem Main-Branch als Tag „1.0“ einsehbar ([github.com/SilasDerProfi/weeping-snake/tree/1.0](https://github.com/SilasDerProfi/weeping-snake/tree/1.0)).

## Kompilieren, testen & ausführen

Das Programm kann begrenzt mit Docker ausgeführt werden, zum Erstellen & Testen und Ausführen mit Interkation ist dotnet (Lauffähig unter allen typischen Systemen) nötig. Im Folgenden ist ein Auszug aus der README-Datei des Repositories, in welcher dieser Prozess beschrieben ist.

|  |
| --- |
| **Getting Started**   1. Make sure that dotnet is installed on your machine. You need It to build, test and run the code.    * To **install dotnet** for example with pacman you can use sudo pacman -S dotnet-sdk    * It is also possible to use **Docker** via the included [Dockerfile](https://github.com/SilasDerProfi/weeping-snake/blob/main/src/WeepingSnake.Game/Dockerfile) to run the program, but no interaction is possible (e.g. controlling your own player) 2. You need the source code. Just load it via the git clone command    * E.g. git clone https://github.com/SilasDerProfi/weeping-snake.git    * Obviously you can specify a specific directory for the clone   **Build** With every commit the code is compiled automatically. You can see if the build was successful by the badge in this readme. To build the code, you must run dotnet build src in the directory of your clone  **Test** With every commit the code is tested automatically. You can see if the test were successful by the badge in this readme. To build the code, you must run dotnet test src in the directory of your clone  **Run** To run the code, you need to run dotnet run --project src/WeepingSnake.ConsoleClient in the directory of your clone |

Tabelle 2: Getting Started

# Domain Driven Design

## Ubiquitous Language

Die Betrachtung der Ubiquitous Language (UL) erfolgt anhand der folgenden Gestaltung der Problemdomäne.

**Accountverwaltung**

(Generische Domäne)

**Highscore liste**

(Unterstützende Domäne)

Abbildung 2: Problemdomäne

Die Einteilung in die drei Subdomänen dient der Eingrenzung der jeweiligen zu betrachtenden Problemdomäne.

Die Kerndomäne „Spiel“ beinhaltet die eigentliche Durchführung eines Spiels, an welchem Spieler teilnehmen. In der Benutzeroberfläche sowie im Quellcode werden daher für diese Subdomäne spezielle Begriffe verwendet. Diese Begriffe sind vor allem auch von Begriffen der anderen Subdomänen abzugrenzen.

|  |  |
| --- | --- |
| **Begriff** | **Beschreibung / Zusammenhang / Abgrenzung** |
| Game | Eine Instanz eines Spieles, an welchem Teilnehmer gegeneinander spielen. |
| Player | Eine Spielfigur, welche von einem Menschen oder einem Computer auf dem *Board* gesteuert wird, mit dem Ziel das Spiel zu gewinnen.   * Zusammenhang mit: *Action*, *PlayerAction* und *Board* * Abgrenzung zu: *Person* * Ist Bestandteil von: *Game* |
| Board | Die Fläche auf welcher ein Spiel stattfindet. Spieler bewegen sich auf dieser Fläche.   * Ist Bestandteil von: *Game* |
| Points | Eine Bewertung innerhalb eines *Game*s, um den Erfolg der *Player* zu messen.   * Ist Bestandteil von: *Player* |
| Action | Eine mögliche Aktion, welche ein *Player* in einem *Game* tätigen kann. (turn left, turn right, change nothing, speed up, slow down, jump)   * Abzugrenzen zu: *PlayerAction* * Ist bestandteil von: *Game* |
| PlayerAction | Eine von einem *Player* getätigte *Action*, welche in der Zukunft ausgeführt wird oder bereits ausgeführt wurde.   * Abzugrenzen zu: *Action* * Ist bestandteil von: *Player* |

Tabelle 3: Ubiquitous Language für die Kerndomäne

Die Subdomäne „Accountverwaltung“ ist eine generische Subdomäne, da diese neben dem eigentlichen Spiel essenziell für den Mehrspielermodus ist. Ohne diese Accountverwaltung ist eine Steuerung des Mehrspielermodus nicht möglich. Im Rahmen der Accountverwaltung ist im Rahmen der UL die Definition und Abgrenzung der folgenden Begriffe erforderlich.

|  |  |
| --- | --- |
| **Begriff** | **Beschreibung / Zusammenhang / Abgrenzung** |
| Person | Ein Mensch, welcher einen Account besitzt. Er kann an mehreren Games mit jeweils einem Player teilnehmen.   * Abgrenzung zu: *Player* |
| Username | Ein von der *Person* gewähltes Pseudonym, welches in der Highscore-Liste verwendet wird. |

Tabelle 4: Ubiquitous Language für die generische Domäne

Die Subdomäne „Highscore-Liste“ umfasst die statistische Übersicht der errungenen *Points* der *Person*s. Während eines *Game*s wissen die *Person*s nicht, von wem die gegnerischen *Player* gesteuert werden. Daher werden diese Informationen über die Highscore-Liste zusammengeführt.

## Kontext

Da es sich bei dem Spiel *Weeping Snake* um ein abgeschlossenes System handelt, wird kein Kontext außerhalb der Problemdomäne betrachtet. Innerhalb der Problemdomäne bestehen die folgenden Abhängigkeiten.

|  |  |
| --- | --- |
| **Abhängige Subdomänen** | **Beziehungsmuster** |
| Accountverwaltung ↔ Spiel | *Shared Kernel* |
| Highscore Liste ↔ Spiel | *Shared Kernel* |

Tabelle 5: Beziehungen zwischen Kontexten

Das Beziehungsmuster *Shared Kernel* wird in diesem Fall verwendet, da das gemeinsame Teilmodell das Spiel an sich ist und sich somit eine zusätzliche Trennung nicht lohnt.

## Taktisches Domain Driven Design

### Entities

Die verwendeten Entities sind Player, Game und Person. Diese Klassen werden jeweils über eine Surrogate-ID (UUID) identifiziert und verglichen und sind veränderbar (bei Person: E-Mail-Adresse & Passwort; bei Game: Teilnehmer, Runde, …; Bei Player: Position, gewählte Aktionen, …).

Mit Commit [2c8242f](https://github.com/SilasDerProfi/weeping-snake/commit/2c8242f1e48e880d0dc09979f62273fedac269a1) erfolgte eine Korrektur, sodass Person und Player als verschieden betrachtet werden, wenn sie verschiedene IDs haben (Überschreiben von Equals & GetHashCode).

Die Identifikation über eine ID ist für Person nötig, da diese Entity persistiert wird. Für Player und Game ist die ID relevant, damit bei der Kommunikation mit einem Client (Web oder CLI) festgestellt werden kann, um welche *Game*-Instanz und welchen teilnehmenden *Player* es sich handelt. Aus diesem Grund wird auch die UUID verwendet, da es hier zusätzlich den Sicherheitsaspekt gibt, dass das herausfinden einer Gültigen ID durch ausprobieren nahezu unmöglich ist.

Die Einhaltung von Domänenregeln und allgemeinen Regeln ist teilweise gegeben. Ungültige Werte und Ungültige Zustände werden zwar durch das Werfen von Ausnahmefehlern verhindert und es werden Value Objects wenn möglich verwendet, die Gestaltung der Entitys bezüglich der öffentlichen Methoden beschreiben das Verhalten jedoch nicht korrekt. Die folgende Abbildung zeigt für die drei beschriebenen Entities jeweils das UML-Klassendiagramm ohne Assoziationen. Das Herz-Symbol steht für den Zugriffsmodifizierer internal. Das Schloss steht für private.



Abbildung 3: UML-Klassendiagramm der Entities

Unter anderem dieser unstrukturierte Aufbau der Entities wird im Rahmen dieser Arbeit in den folgenden Kapiteln durch das Anwenden verschiedener Techniken korrigiert.

### Value Objects

Durch die Implementierung mit der Programmiersprache C# gibt es für den Einsatz von Value Objects neben der Klasse (class) die Struktur (struct). Dieser Typ erzwingt die Einhaltung der Eigenschaften (unveränderlich, gleich bei selbem Wert, …) der Value Objects.

In der folgenden Tabelle sind die verwendeten Value Objects aufgeführt.

|  |  |
| --- | --- |
| **Name** | **Funktion** |
| **PlayerRange** | Bestimmt wie viele Spieler (min, max) an einem *Game* teilnehmen können / müssen. |
| * Validiert, dass min >= max | |
| **BoardDimensions** | Bestimmt die Ausmaße eines *Board*s (Breite / Höhe) |
| * Validierung (Größe je Seite mindestens 5) mit Commit [a2e14b6](https://github.com/SilasDerProfi/weeping-snake/commit/a2e14b6dfdb7a158e34ccab802baeeca0c18f8eb) hinzugefügt | |
| **GameCoordinate** | Eine Position auf dem *Board* für eine Spezifische Runde (x, y, z) |
| * Unveränderlich, da eine Position immer die gleiche Position bleibt * Kapselung, da es an mehreren Stelle verwendet wird | |
| **PlayerDirection** | Eine Richtung in welche sich ein *Player* bewegen kann |
| * Unveränderlich, da eine Richtung immer die gleiche Reichtung bleibt * Kapselung, da es an mehreren Stelle verwendet wird | |
| **PlayerOrientation** | Vereint GameCoordinate und PlayerDirection |
| * Kapseln der Ausrichtung des *Player*s auf dem *Board* von dem *Player* an sich. Für eine Änderung der Ausrichtung wird die aktuelle überschrieben, da die Ausrichtung des Spielers sich nicht verändert, sondern der Spieler die Ausrichtung wechseln muss. | |
| **GameDistance** | Eine mögliche Bewegung auf einem *Board* mit Start- und Endpunkt. |

Tabelle 6: Value Objects

### Aggregates

Die Aggregates werden in Anlehnung an das in Abbildung 1: Code-Struktur in UML Notation (Seite 3) dargestellte UML-Diagramm betrachtet. Die Zusammenfassung in Funktionseinheiten stellt hier schon die Bildung der Aggregates dar. In der folgenden Tabelle sind die Aggregate Root Entities aufgeführt.

|  |  |
| --- | --- |
| **Aggregate** | **Aggregate Root Entity** |
| Accountverwaltung | PersonDatabase |
| Spieler | Player |
| Spieleverwaltung | GameController |
| Spiel | Game |
| Spielfeld | Game.Board |
| Computerspieler | ComputerPlayer |

Tabelle 7: Aggregates und Root Entities

Anhand des UML-Diagramms ist zu sehen, dass nicht immer der Zugriff über das Aggregate Root erfolgt. Dieses Problem besteht in den Aggregates Accountverwaltung und Spieleverwaltung und wird in den folgenden Kapiteln genauer betrachtet und behoben.

### Repositories

In der Anwendung *WeepingSnake* werden lediglich *Person*s persisitiert, sodass nur für das Aggregat Accountverwaltung das Repository betrachtet wird. Der Zugriff auf persistierte Entities erfolgt über die Klasse PersonDatabase sowie über die Entity Person direkt.[[2]](#footnote-2) Somit ist hier das Repository nicht korrekt implementiert, was auch darauf zurückzuführen ist, dass das Aggregate nicht korrekt ist. In den folgenden Kapiteln wird daher das Accountverwaltung-Aggregate korrigiert, sodass der zugriff nur über das Aggregate Root erfolgt und dieses auf ein einzelnes Repository zugreift, um persistierte Daten abzufragen.

# Programming Principles

## SOLID

Analyse und Begründung

## GRASP

Analyse und Begründung

(insbes. Kopplung und Kohäsion)

## DRY

Analyse und Begründung

# Entwurfsmuster

Im Folgenden wird das Objektbasierte Erzeugungsmuster „Erbauer“ auf die Erzeugung Computergesteuerter *Player* angewendet.

Es gibt verschiedene Arten von Computerspielern, welche über das Interface IComputerPlayer generalisiert werden. Die Klasse Game übernimmt die Aufgabe, zufällig verschiedene Computerspieler zu erzeugen und einem *Game* zuzuweisen, wenn nicht genügend echte Teilnehmer verfügbar sind. Die Zuweisung erfolgt über die Klasse Player. Innerhalb der Klasse Player wird die Steuerung des Computerspielers initialisiert. Dieser Ablauf ist in dem folgenden Diagramm zu sehen.



Abbildung 4: Ablauf Computerspieler Initialisierung ohne Erbauer

Die Verantwortlichkeit für das Erzeugen eines Computerspielers ist somit zwischen der Klasse Game und der Klasse Player aufgeteilt. In dem folgenden UML-Diagramm sind die Abhängigkeiten dargestellt. Durch die Generalisierung der verschiedenen Computerspielerklassen als IComputerPlayer muss nur die Klasse Game die einzelnen ComputerPlayer-Klassen kennen. Die Player-Klasse kann über die Schnittstelle IComputerPlayer auf die ControlledPlayer-Eigenschaft zugreifen.



Abbildung 5: UML-Klassendiagramm Computerspielererzeugung ohne Erbauer

Durch den Einsatz des Erbauer-Patterns, soll dieser Konstruktionsprozess aufgelöst werden, sodass die Klasse Game zwar weiterhin als Direktor fungiert, die Initialisierung soll jedoch von einem separaten Erbauer übernommen werden. Dieser Separate Erbauer ist die Klasse CreateComputerPlayer. Durch deren Methode CreateForGame(Game game) wird die Zufallsauswahl und Initialisierung von Computerspieler und Player übernommen. An dem UML-Klassendiagramm ist somit der tatsächliche Vorteil (Initialisierung ist nichtmehr über mehrere Klassen verteilt) nicht direkt erkennbar.



Abbildung 6: UML-Klassendiagramm Computerspielererzeugung mit Erbauer

Es ist zu sehen, dass sich die Verweise auf die Implementierungen der IComputerPlayer-Schnittstelle verschoben haben, sodass diese nur noch dem Erbauer bekannt sind. In dieser Umstrukturierung fungiert die Klasse Game als Direktor und verwendet CreateComputerPlayer als Erbauer. Dieser Erbauer wiederum übernimmt die Verantwortung, dass der Player und der IComputerPlayer korrekt initialisiert und dem Game korrekt zugewiesen werden. Dieser Ablauf ist in dem folgenden Diagramm dargestellt.

Die Änderungen erfolgten mit Commit [06c8851d](https://github.com/SilasDerProfi/weeping-snake/commit/06c8851d6f948675f4367153e81275225686bf80) & [73a3bb46](https://github.com/SilasDerProfi/weeping-snake/commit/73a3bb46d376317719253bc31d1f7b900f7381b6) & [809b2ef](https://github.com/SilasDerProfi/weeping-snake/commit/809b2eff3eec3bf2df85cd5b44c5955cbd4c5847) & [f3748cf](https://github.com/SilasDerProfi/weeping-snake/commit/f3748cf0bbb7eaa122d103d885bf2c0e42f202f2) & [fcd0705](https://github.com/SilasDerProfi/weeping-snake/commit/fcd0705570598ca92c61670481b1c5f636aac04e)



Abbildung 7: Ablauf Computerspieler Initialisierung mit Erbauer

Ein weiteres zu betrachtendes Entwurfsmuster ist das Adapter-Pattern (Wrapper). Hierbei wird eine Klasse zu einer für den Client leicht zu verarbeitenden Struktur konvertiert. Zusätzlich bietet es den Vorteil einer gewissen Unabhängigkeit von der Zugrundeliegenden Datenstruktur, sodass Änderungen hinter (*Application*-Schicht) oder vor (*Plugin*-Schicht) dem Adapter auf die jeweilige andere Schicht keine Auswirkungen haben. Im folgenden Kapitel *Clean Architecture* wird dieses Entwurfsmuster betrachtet und angewendet.

# Clean Architecture

Als Klasse der Adapterschicht wird der GameController des WebService (WeepingSnake.WebService) betrachtet. Diese Klasse erlaubt es, dass über WebRequests auf das Game-Backend zugegriffen werden kann. Damit die Klasse den Anforderungen der Adapterschicht gerecht wird, wurden Mappings der internen Datentypen auf einfache Strukturen abgebildet. Hierfür werden die folgenden beiden Methoden betrachtet:

Methode: GetGameState

1. /// <summary>
2. /// Requests the game state for a game in which a specific player is
3. /// participating.
4. /// </summary>
5. /// <param name="playerId">The id of the player for which the game state is
6. /// requested.</param>
7. /// <returns>The state of the game, if the specified player exists and is
8. /// involved in a game.</returns>
9. [HttpGet("{id}")]
10. public object GetGameState(Guid playerId)

Die Änderungen erfolgten mit Commit [37f57e9](https://github.com/SilasDerProfi/weeping-snake/commit/37f57e94fdb68995d0bf3871571adc1aef01cc68) und [06c8851](https://github.com/SilasDerProfi/weeping-snake/commit/06c8851d6f948675f4367153e81275225686bf80)

Durch das Mapping soll zum einen sichergestellt werden, dass an den Client nur die relevanten Informationen einer Runde gesendet werden, welche benötigt werden, um das Spiel zu visualisieren. Zum andren werden explizit Informationen entfernt, welche dem Client einen unfairen Vorteil verschaffen würden (z. B. welche Aktionen die Gegner für die kommenden Runden geplant haben). Somit wird der aktuelle Status des Spielers (erreichte Punkte und IsAlive), die Pfade der letzten fünf Runden (für die Visualisierung) und die Größe des Boards extrahiert und in ein Objekt zusammengefasst. Die Umsetzung des Mappings erfolgt entsprechend der C#-Best-Practices mit einem anonymen Objekt.

Methode: GetHighscores

1. /// <summary>
2. /// Returns the complete Highscore-List
3. /// </summary>
4. /// <returns>List of Highscore entries</returns>
5. [HttpGet]
6. public IEnumerable<object> GetHighscores()

Die Änderungen erfolgten mit Commit [9bb0e48](https://github.com/SilasDerProfi/weeping-snake/commit/9bb0e48409aceec12cd9770209d2fd0d02e901c1)

Damit die Highscores direkt in der Client-Oberfläche angezeigt werden können, übernimmt das Mapping die Sortierung nach den Highscore-Punkten (absteigend) und generiert einen String um die Platzierung darzustellen. Die Umsetzung des Mappings erfolgt entsprechend der C#-Best-Practices mit einem anonymen Objekt. Im Folgenden ist der Mapping-Code abgebildet. Hierbei gilt zu beachten, dass diese Methode im Rahmen des API-Designs weiter angepasst wird.

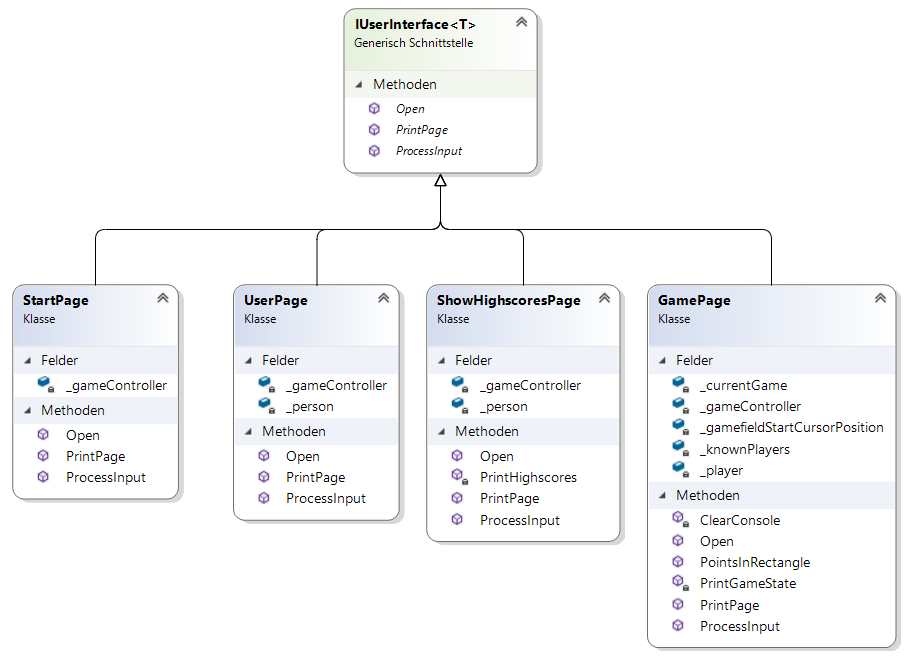
1. public IEnumerable<object> GetHighscores()
2. {
3. var highscores = HighscoreEntry.GetHighscoreEntries();
4. highscores = highscores.OrderByDescending(h => MaximumPointsInGame)
5. .ToList();
6. for(int placement = 1; placement <= highscores.Count; placement++)
7. {
8. var highscoreEntry = highscores[placement];
9. var mappedHighscore = new
10. {
11. Placement = $"#{placement}",
12. UserName = highscoreEntry.Username,
13. Highscore = highscoreEntry.MaximumPointsInGame,
14. NumerOfPlayedGames = highscoreEntry.PlayedGames,
15. HighscoreSum = highscoreEntry.TotalPoints
16. };
17. yield return mappedHighscore;
18. }
19. }

Quellcode 1: Highscore Mapping Code der Adapter Schicht

Da die beiden beschriebenen Methoden Anonyme Objekte verwenden, welche über JSON übertragen werden, wird hierbei nicht über Interfaces kommuniziert. Dies ist jedoch grundsätzlich möglich, indem für die entsprechende Client-Technologie (z. B. TypeScript) ein Interface im Rahmen einer Quellcodebibliothek bereitgestellt werden. Ein solches Interface könnte z. B für einen Eintrag in der Highscore Liste folgendermaßen aussehen.

1. interface HighscoreListEntry {
2. readonly placement: string;
3. readonly userName: string;
4. readonly highscore: number;
5. readonly numberOfPlayedGames: number;
6. readonly highscoreSum: number;
7. }

Quellcode 2: Adapter-Interface für Client mit TypeScript-Technologie



# Legacy Code

* >= 2 Abhängigkeiten mit Techniken aus der Vorlesung brechen
* ausgewählte Stellen und Techniken begründe

Person & Highscore entry (interface?)

# Refactoring

Im Folgenden werden die Code Smells in dem Quellcode betrachtet und durch geeignete Refactorings gelöst.

## Long Method

* Code Smells identifizieren
  + Game Configuration constructor
  + Random not killing itself player
  + Create copmuterplayer
  + GameDistance Konstruktor
  + Game Join
  + Game Leave
  + Game Apply action per player
  + Game board apply action
  + Game board randomstart coordinate
* Refactoring anwenden und begründen
* UML vorher & nachher
* Extract Method (Auftrennen, Name finden,)

## Duplicated Code

Im Folgenden wird der Code Smell Duplicated Code an mehreren Stellen betrachtet

**UI Code**

Da es sich bei dem Offline-Client um eine Konsolenanwendung handelt, liegt die Kontrolle der Oberfläche hierbei komplett beim Entwickler, da kein Framework verwendet wird, welches konkrete Strukturen vorgibt. Der aktuelle Stand ist an dem folgenden UML-Klassendiagramm zu sehen.

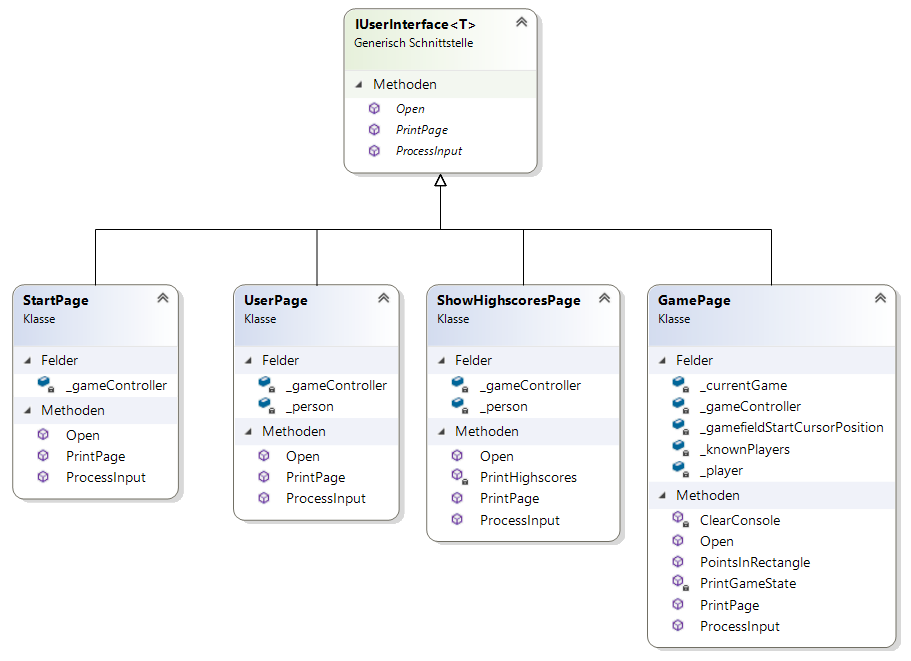


Abbildung : Codestruktur UI-Code

Die verschiedenen Page-Klassen sind für die Darstellung und Navigation zuständlich und implementieren das Interface IUserInterface. Durch diesen ähnlichen Aufgabenbereich kommt der folgenden Quellcode-Ausschnitt mehrmals in jeder Klasse vor.

1. Console.WriteLine("There was an error. Press any key.");
2. Console.ReadKey();
3. new StartPage().Open(\_gameController);

Quellcode 3: UI-Code für die Behandlung von falschen Eingaben

Außerdem wird auch innerhalb der Klassen bei korrekten Eingaben an verschiedenen Stellen der gleiche Code ausgeführt. Um die Wiederverwendung des Codes zu erhöhen, wird daher der Code für den Fehlerfall in eine separate Methode extrahiert. Hierfür wird statt eines Interfaces eine Abstrakte Klasse verwendet.

Durch die Abstrakte Klasse ist es nun ebenfalls möglich, dass die von der Page geforderten Daten direkt über den Konstruktor definiert werden können. Somit werden die Daten nun ebenfalls in der abstrakten Oberklasse gehalten und die Unterklassen greifen nur noch auf diese Daten zu. Hierbei wurden zusätzlich Rename Method angewandt (vgl. UML Diagramme). Zusätzlich werden häufig verwendete Code-Ausschnitte in Methoden extrahiert.

Das folgende UML-Diagramm zeigt die neue Struktur der Klassen. Vor allem in Bezug auf die im Quellcode enthaltenen Benutzerausgaben, ist somit die Wartbarkeit sichergestellt. Hierbei würde es sich anbieten die Ausgabetexte in eine Ressourcendatei auszulagern und darauf über einen Ressource-Manager zuzugreifen, was jedoch in diesem Zuge nicht gemacht wird.

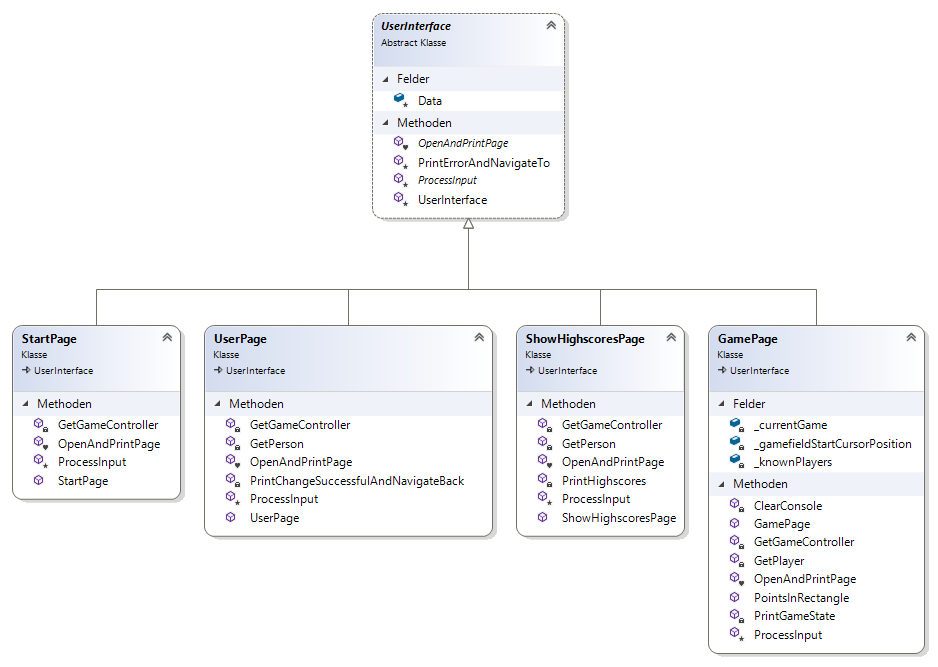


Abbildung 8: Codestruktur UI-Code nach Refactorings

Hierbei ist zu sehen, dass die Privaten Eigenschaften durch das Feld Data in der abstrakten Klasse UserInterface abgelöst wurden (Hierauf wird innerhalb der Kind-Klassen über Getter zugegriffen). Diese Abstrakte Klasse enthält neben dem Konstruktor die zwei abstrakten Methoden OpenAndPrintPage sowie ProcessInput, welche für die Darstellung und Navigation verantwortlich sind. Über PrintErrorAndNavigateTo können die geerbten Klassen den Fehlerfall über die Oberklasse abbilden. Zusätzlich wird in UserPage die Methode PrintChangeSuccessfulAndNavigateBack eingesetzt, um die Wiederverwendung des enthaltenen Codes zu ermöglichen.

Die Änderungen erfolgten mit dem Commit [198156a](https://github.com/SilasDerProfi/weeping-snake/commit/198156a9de1f44396d75eb93922a88d880af80fa).

* Code Smells identifizieren
  + CoordinateSystemTests (new list…)
  + Logger (das drum herum)
  + Person & Highscore entry (interface?)
  + Vector2 extensions
* Refactoring anwenden und begründen
* UML vorher & nachher
* (innerhalb einer Klasse oder ähnlicher code in mehreren klassen/methoden)

## Weitere Code Smells

* Code Smells identifizieren
* Refactoring anwenden und begründen
* UML vorher & nachher
* Evtl ist klasse game zu groß (player , logging, controller evtl auch)

# Unit Tests

## Initiale Code Coverage

Im Folgenden wird die Code Coverage von dem Game-Backend (WeepingSnake.Game) und dem Offline-Client (WeepingSnake.ConsoleClient) betrachtet. Die aufgeführte Tabelle enthält die Code Coverage-Ergebnisse (Line Coverage) für diese beiden Projekte.

|  |  |  |
| --- | --- | --- |
| **Projekt / Namespace** | | **Code Coverage** |
| **WeepingSnake.ConsoleClient** | | **0,00 %** |
|  | Navigation | 0,00 % |
| **WeepingSnake.Game** | | **5,80 %** |
|  | Game | 1,15 % |
| Geometry | 14,04 % |
| Person | 0,00 % |
| Player | 0,00 % |
| Player.ComputerPlayer | 0,00 % |
| Structs | 0,00 % |
| Utility.Extensions | 79,03 % |
| Utility.Logging | 0,00 % |

Tabelle 8: Code Coverage (initial)

Diese Code Coverage ergibt sich aus 17 verschiedenen Unit-Tests, welche primär die Utility.Extensions Funktionalitäten abdecken. In diesem Kapitel wird beschrieben, wie die Code Coverage durch ein Testkonzept erhöht wird.

## Konzept

Um eine Angemessene Code Coverage (Line Coverage) zu erreichen wurden weitere Tests geschrieben. Insgesamt stehen nun 56 Unit Tests zur Verfügung, welche die bereits beschriebenen Teile der Software folgendermaßen abdecken.

Die Änderungen erfolgten mit den Commits: [dd2298e](https://github.com/SilasDerProfi/weeping-snake/commit/dd2298e4b866489ce17c2bcf6937820c65b7bda8), [d4ecd13](https://github.com/SilasDerProfi/weeping-snake/commit/d4ecd136a899ae5a0f55dff8f0de925b62347c6b), [a274b33](https://github.com/SilasDerProfi/weeping-snake/commit/a274b330d91d1a4099bab7820538a4cdfb3a8920), [bb31c68](https://github.com/SilasDerProfi/weeping-snake/commit/bb31c6842bc3a1e40abb6938f75ef7924b6e820e), [bd1f07d](https://github.com/SilasDerProfi/weeping-snake/commit/bd1f07dc810572febe3d17b52f888bafbad1814c), [f9eb1c5](https://github.com/SilasDerProfi/weeping-snake/commit/f9eb1c533640128b8e92d431fc1a242225323397), [a14e1e7](https://github.com/SilasDerProfi/weeping-snake/commit/a14e1e7c287fca0e8ad6dac76ce3ca6958c030b9), [5a3736b](https://github.com/SilasDerProfi/weeping-snake/commit/5a3736bb48316c1c981776f2537adc3373ac4aa4), [0bc0443](https://github.com/SilasDerProfi/weeping-snake/commit/0bc044348b22f943a83ce6b2756812041e3ea841), [1fa0fa9](https://github.com/SilasDerProfi/weeping-snake/commit/1fa0fa912445c598d4a1ac635ac747f1ff2d868d), [150db9f](https://github.com/SilasDerProfi/weeping-snake/commit/150db9f01019476fb942b030663e1dcd71020602), [f8f05ce](https://github.com/SilasDerProfi/weeping-snake/commit/f8f05cee1764d3f4182fc04d90ef288b096c9c49), [c3fec83](https://github.com/SilasDerProfi/weeping-snake/commit/c3fec833193deccf261565c7b311c8650b6ad584), [201ae9e](https://github.com/SilasDerProfi/weeping-snake/commit/201ae9e7854cf6202c3b8d933f4d2c12c522b5cf), [18e714c](https://github.com/SilasDerProfi/weeping-snake/commit/18e714cde27bdaa7a751dd146ad4cc2de42a5bd5), [58fb81a](https://github.com/SilasDerProfi/weeping-snake/commit/58fb81ad84c875805c20a13b6b26694e07fb9991), [655f2c3](https://github.com/SilasDerProfi/weeping-snake/commit/655f2c320e24540c137841c2a1f1d28df9dfeac6), [b6abb25](https://github.com/SilasDerProfi/weeping-snake/commit/b6abb25a686e2dab5aa407c6c3107aec056d9994)

|  |  |  |
| --- | --- | --- |
| **Projekt / Namespace** | | **Code Coverage** |
| **WeepingSnake.ConsoleClient** | | **3,08 %** |
|  | Navigation | 3,08 % |
| **WeepingSnake.Game** | | **78,60 %** |
|  | Game | 75,68 % |
| Geometry | 96,07 % |
| Person | 91,23 % |
| Player | 86,87 % |
| Player.ComputerPlayer | 89,72 % |
| Structs | 100,00 % |
| Utility.Extensions | 98,39 % |
| Utility.Logging | 17,22 % |

Tabelle 9: Code Coverage (mit Testkonzept)

Die Test Coverage des ConsoleClient fällt sehr gering aus, da das Projekt hauptsächlich aus UI besteht. Da die UI nicht sehr umfangreich ist und komplett in wenigen Minuten überprüft werden kann, sind hierfür Unit Tests nicht sinnvoll und nötig. Die stichprobenartige Überprüfung nach einer Änderung am UI-Code reicht aus, um die gesamte Benutzeroberfläche zu prüfen.

Die Test Coverage für das Game Backend liegt bei insgesamt 78,6 %. Diese Testabdeckung wird jedoch durch den Bereich Logging nach unten verfälscht, da dieser Code primär für die Fehlersuche während der Entwicklung gedacht ist. Das Logging ist standardmäßig deaktiviert und sollte in einer Produktivumgebung ebenfalls nur für die Fehlersuche verwendet werden. Des Weiteren handelt es sich hierbei lediglich um das schreiben einer Logdatei, sodass Unit-Tests hierfür nur begrenzt sinnvoll sind, da sich das Programmverhalten hier auf den verschiedenen System unterscheiden kann (Wenn der Test unter Windows laufen würde, ist das keine Referenz für Linux, MacOs, …).

Für die anderen Projektbestandteile liegt die Code Coverage über 85% (außer für Game, da hier das Laden der Konfigurationsdatei enthalten ist – Betrifft IO wie beim Logging). Mit den erstellten Tests wird beinahe die Gesamte Logik getestet. Der nicht abgedeckte Teil sind hauptsächlich Getter oder Setter, welche zwar vom ConsoleClient verwendet werden, jedoch nicht explizit über Unit-Tests abgeprüft werden.

## Mock-Objekte

Bei der Implementierung der Unittests musste mit Mock-Objekten gearbeitet werden. Hierfür wurden Mock-Klassen für CoordinateSystem, Player, Game und erstellt.

### MockCoordianteSystem

Bei dem CoordianteSystem handelt es sich um eine abstrakte Generalisierung des Spielfelds (*Board*), welches Teil von einem *Game* ist. Damit die Tests, welche nicht die Funktionalitäten von Board betreffen, sondern nur die Oberklasse CoordinateSystem, von der Komponente Board unabhängig sind, wird hierfür ein MockCoordiateSystem verwendet.

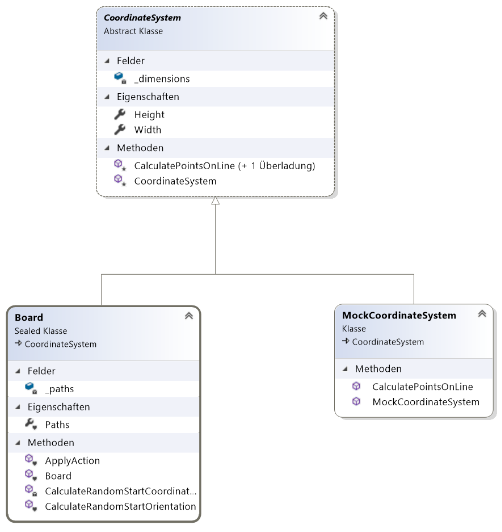


Abbildung 9: UML-Klassendiagramm CoordinateSystem

Das MockCoordianteSystem erbt auch von der abstrakten Klasse, sodass ein größerer Umbau nicht erforderlich ist. Hierbei werden Eigenschaften, welche das Board besitzt, jedoch nicht implementiert, da diese für die betroffenen Tests nicht relevant sind.

Neben dem Vorteil, dass die Abstrakte Klasse unabhängig von ihrer Implementierung getestet werden kann, kann das MockCoordinateSystem auch für andere Tests als eine Art „MockBoard“ verwendet werden (sofern hierfür der implementierte Umfang reicht, z. B. für [GameDistanceTests](https://github.com/SilasDerProfi/weeping-snake/blob/main/src/tests/WeepingSnake.Game.Tests/Geometry/GameDistanceTests.cs)).

### MockPlayer

Die Klasse Player ist zentraler Bestandteil und wird daher oft verwendet. Um die Methoden, welche die Player-Klasse verwenden unabhängig von der Implementierung der Klasse Player zu testen, wird hierfür ein Mock-Objekt eingesetzt. Um ein Fake-Objekt für die klasse Player zu erzeugen, musste zuerst Extract Interface angewendet werden. Hierfür werden die Methoden, Getter und Setter in das Interface übernommen, welche für die Tests relevant sind (dies sind beinahe alle, da das Mock-Objekt von mehreren Tests verwendet wird). Zusätzlich werden Referenzen auf die Player-Klasse im Quellcode auf das neue Interface IPlayer geändert (wo möglich und sinnvoll), damit sowohl Tests wie auch Quellcode primär mit dem Interface arbeiten. Damit jeder Test das Mock-Objekt möglichst spezifisch manipulieren kann, werden Functions und Actions eingesetzt (gemäß C#-Best-Practices / unit test). Diese ermöglichen es, das Verhalten des Mock-Objekts von außen zu steuern, indem vorgegeben wird, wie auf bestimmte Methodenaufrufe reagiert werden soll (z. B. bei [CreateComputerPlayerTests](https://github.com/SilasDerProfi/weeping-snake/blob/main/src/tests/WeepingSnake.Game.Tests/Player/ComputerPlayer/CreateComputerPlayerTests.cs) wird die Methode Join(game) über die Eigenschaft JoinAction definiert).

1. public Action<IGame> JoinAction { get; set; }
2. public void Join(IGame game)
3. {
4. JoinAction?.Invoke(game);
5. }

Quellcode 4: Beispiel: Kontrollierbare Methode eines Mock-Objekts

In dem folgenden UML-Diagramm ist das Ergebnis dargestellt. Die Bezeichnung „Action“ oder „Func“ hinter dem Namen einer Eigenschaft deutet darauf hin, dass es sich um einen Delegaten handelt, welcher für die Steuerung des Mock-Objekts verwendet wird.

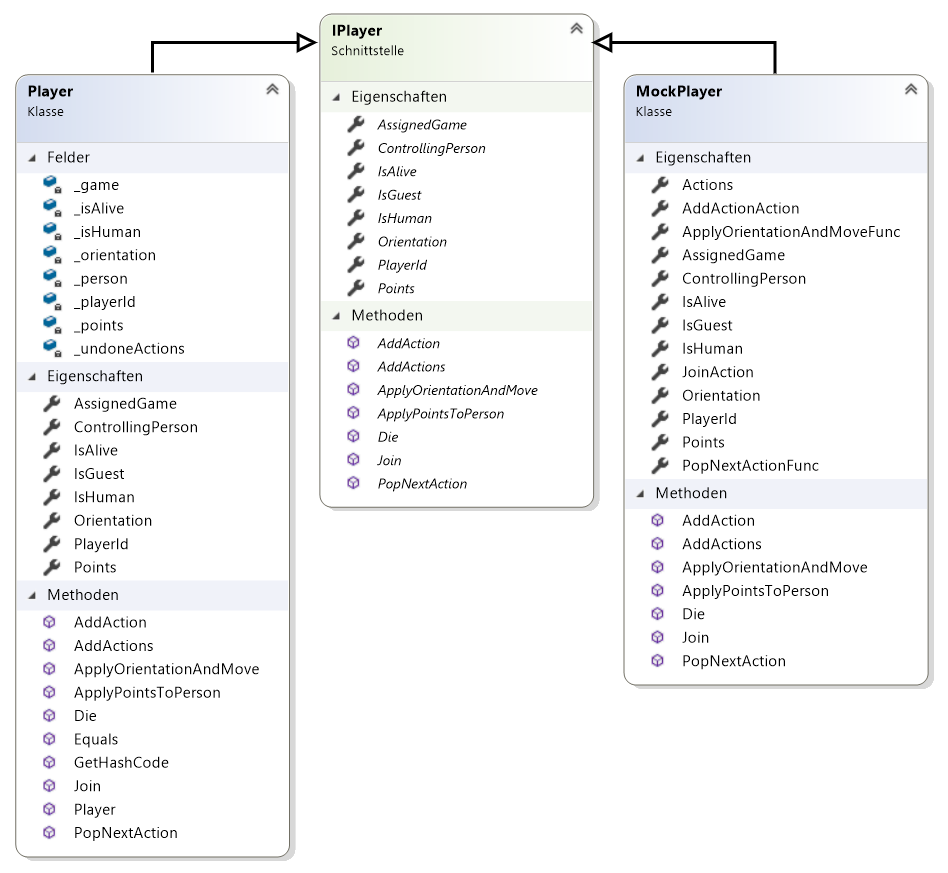


Abbildung 10: UML-Klassendiagramm für MockPlayer

### MockGame

Die Klasse Game ist zentraler Bestandteil und wird daher oft verwendet. Um die Methoden, welche die Game-Klasse verwenden unabhängig von der Implementierung der Klasse Game zu testen, wird hierfür ein Mock-Objekt eingesetzt. Um ein Fake-Objekt für die Klasse Game zu erzeugen, musste zuerst Extract Interface angewendet werden. Das Verfahren hierfür ist analog zu dem bereits beschriebenen MockPlayer. In dem folgenden UML-Diagramm ist das Ergebnis dargestellt.

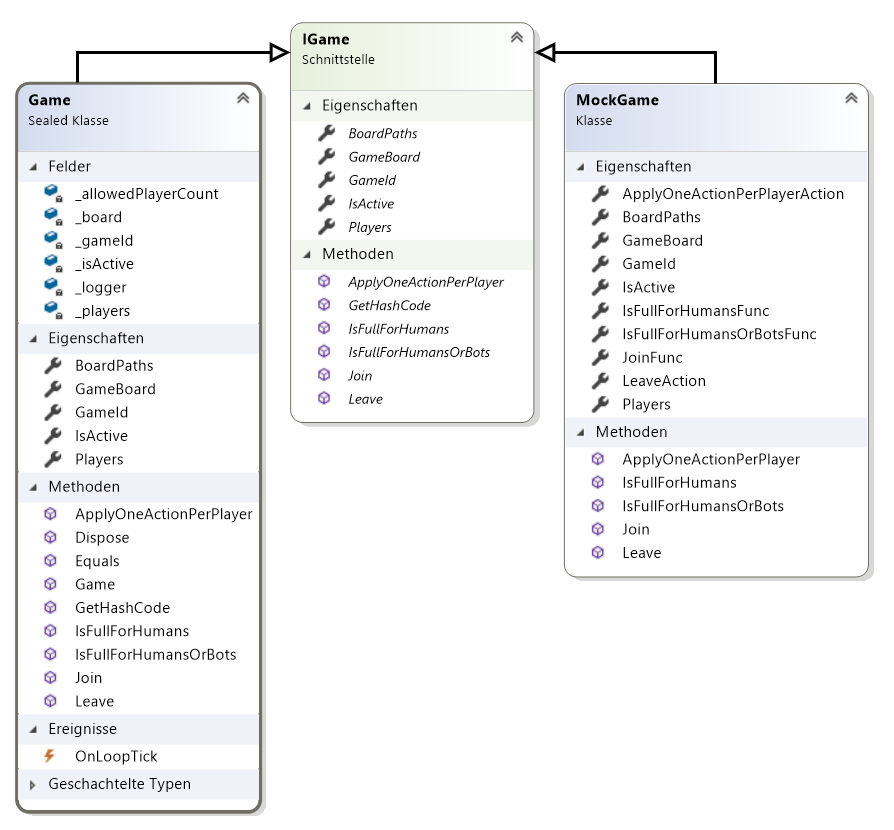


Abbildung 11: UML Klassendiagramm für MockGame

## ATRIP-Regeln

Die Einhaltung der ATRIP Regeln ist für alle Unit-Tests Pflicht. Da kein Ansatz wie *Test Driven Development* oder *Test First* verwendet wurde, konnte die Einhaltung direkt beim (nachträglichen) programmieren der Tests sichergestellt werden. Im Folgenden wird die Umsetzung dieser Regeln für alle Tests für das Game-Backend betrachtet. Die Tests stehen auf [GitHub](https://github.com/SilasDerProfi/weeping-snake/tree/main/src/tests/WeepingSnake.Game.Tests) zur Verfügung.

|  |  |
| --- | --- |
| **Regel** | **Umsetzung / Sicherstellung** |
| **A**utomatic | * Durch die Verwendung von XUnit und Mockobjekten * Continuous Integration über GitHub Actions: Tests werden nach jedem Commit automatisch ausgeführt |
| **T**hrough | * Potenzielle Fehler durch Tests abgedeckt (subjektiv) * Aufgetretene Fehler und deren Umgebung * Hohe Test Coverage, vor allem in kritischen / oft verwendeten Bereichen * Es wird auch geprüft, dass Exceptions bei Fehlverhalten geworfen werden. |
| **R**epeatable | * Durch die Verwendung von XUnit und Mockobjekten * Continuous Integration über GitHub Actions: Tests werden nach jedem Commit und täglich um 3:00 Uhr automatisch ausgeführt |
| **I**ndependet | * Beim Ausführen der Tests werden immer alle Tests ausgeführt (Über die IDE oder GitHub Actions) 🡪 Hier ist die Reihenfolge verschieden. * Tests beziehen sich immer auf kleine Bestandteile. * Neben Arrange, Act und Assert wird auch ein Annihilate-Teil implementiert, wenn nach dem Test aufgeräumt werden muss |
| **P**rofessional | * Getter & Setter werden nicht (explizit) getestet * Der Arrange-Teil der Tests wird (wenn möglich und sinnvoll) ausgelagert * Keine Tests um die Test Coverage künstlich nach oben zu drücken |

Tabelle 10: Betrachtung ATRIP-Regeln

# Zusatz: API-Design

Im Folgenden wird die Web-Schnittstelle betrachtet. Diese besteht aus 11 HTTP-Endpunkten und erlaubt die Kommunikation mit dem Game-Backend. Gegenstand der Übertragung sind dabei Informationen über die Spiele und die Benutzeraccounts (incl. Highscores).

## API

Sinnvollen API

* + http (Highscores noch anpassen)
  + 11 Methoden / HTTP Endpunkte
  + Responsecodes korrekt verwenden

## Analyse

* Analyse der API anhand der Qualitätsmerkmale und Begründung für das Design

1. struct in C#: Wie eine Klasse, mit gewissen Einschränkungen (hat immer Standardwerte / ist nie null und kann nicht erben oder vererbt werden) [↑](#footnote-ref-1)
2. Da laut Aufgabenstellung eine Datenbankanbindung etc. nicht erforderlich / gewünscht ist, wird für das Persistieren eine private Liste verwendet. [↑](#footnote-ref-2)