

**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

29.05.2021



**Kurs:** TINF18B5

**Dozent:**  Herr Maurice Müller

**Inhaltsverzeichnis**

[Abkürzungsverzeichnis / Glossar IV](#_Toc73274716)

[Verzeichnisse V](#_Toc73274717)

[1 Projektbeschreibung 1](#_Toc73274718)

[1.1 Funktionsumfang 2](#_Toc73274719)

[1.2 Code-Struktur 3](#_Toc73274720)

[1.2.1 LOC 4](#_Toc73274721)

[1.3 Kompilieren, testen & ausführen 5](#_Toc73274722)

[2 Domain Driven Design 6](#_Toc73274723)

[2.1 Ubiquitous Language 6](#_Toc73274724)

[2.2 Kontext 8](#_Toc73274725)

[2.3 Taktisches Domain Driven Design 9](#_Toc73274726)

[2.3.1 Entities 9](#_Toc73274727)

[2.3.2 Value Objects 11](#_Toc73274728)

[2.3.3 Aggregates 12](#_Toc73274729)

[2.3.4 Repositories 12](#_Toc73274730)

[3 Programming Principles 14](#_Toc73274731)

[3.1 SOLID 15](#_Toc73274732)

[3.1.1 Single Responsibility Principle 15](#_Toc73274733)

[3.1.2 Open Closed Principle 16](#_Toc73274734)

[3.1.3 Liskov Substitution Principle 17](#_Toc73274735)

[3.1.4 Interface Segregation Principle 17](#_Toc73274736)

[3.1.5 Dependency Inversion Principle 19](#_Toc73274737)

[3.2 GRASP 20](#_Toc73274738)

[3.2.1 Low Coupling!! 20](#_Toc73274739)

[3.2.2 High Cohesion!! 20](#_Toc73274740)

[3.2.3 Information Expert 20](#_Toc73274741)

[3.2.4 Creator 20](#_Toc73274742)

[3.2.5 Indirection 21](#_Toc73274743)

[3.2.6 Polymorphism 21](#_Toc73274744)

[3.2.7 Controller 22](#_Toc73274745)

[3.2.8 Pure Fabrication 22](#_Toc73274746)

[3.2.9 Protected Variations 23](#_Toc73274747)

[3.3 DRY 24](#_Toc73274748)

[4 Entwurfsmuster 25](#_Toc73274749)

[5 Clean Architecture 30](#_Toc73274750)

[6 Legacy Code 33](#_Toc73274751)

[7 Refactoring 40](#_Toc73274752)

[7.1 Long Method 40](#_Toc73274753)

[7.2 Duplicated Code 44](#_Toc73274754)

[7.3 Weitere Code Smells 48](#_Toc73274755)

[8 Unit Tests 49](#_Toc73274756)

[8.1 Initiale Code Coverage 49](#_Toc73274757)

[8.2 Konzept 50](#_Toc73274758)

[8.3 Mock-Objekte 52](#_Toc73274759)

[8.3.1 MockCoordianteSystem 52](#_Toc73274760)

[8.3.2 MockPlayer 54](#_Toc73274761)

[8.3.3 MockGame 56](#_Toc73274762)

[8.4 ATRIP-Regeln 57](#_Toc73274763)

[9 Zusatz: API-Design 58](#_Toc73274764)

[9.1 Design 58](#_Toc73274765)

[9.2 Analyse 61](#_Toc73274766)

# Abkürzungsverzeichnis / Glossar

**API** Application Programming Interface, Programmierschnittstelle

**CLI** Command-line interface, Kommandozeile

**DIP** Dependency Inversion Principle (D von SOLID)

**DRY** Don’t repeat yourself

**GRASP** General Responsibility Assignment Software Pattern

**ISP** Interface Segregation Principle (I von SOLID)

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

**LSP** Liskov Substitution Principle (L von SOLID)

**OCP** Open / Closed Principle (O von SOLID)

**SRP** Single Responsibility Principle (S von SOLID)

**UL** Ubiquitous Language

# Verzeichnisse

[Abbildung 1: Code-Struktur in UML Notation 3](#_Toc73109921)

[Abbildung 2: Problemdomäne 6](#_Toc73109922)

[Abbildung 3: UML-Klassendiagramm der Entities 10](#_Toc73109923)

[Abbildung 4: Ablauf Computerspieler Initialisierung ohne Erbauer 15](#_Toc73109924)

[Abbildung 5: UML-Klassendiagramm Computerspielererzeugung ohne Erbauer 16](#_Toc73109925)

[Abbildung 6: UML-Klassendiagramm Computerspielererzeugung mit Erbauer 17](#_Toc73109926)

[Abbildung 7: Ablauf Computerspieler Initialisierung mit Erbauer 18](#_Toc73109927)

[Abbildung 8: UML-Klassendiagramm UI-Code abhängig von .NET Framework 27](#_Toc73109928)

[Abbildung 9: UML-Klassendiagramm testbarer UI-Code unabhängig von .NET Framework 28](#_Toc73109929)

[Abbildung 10: UML-Klassendiagramm RandomNotKillingItselfPlayer 30](#_Toc73109930)

[Abbildung 11: Codestruktur UI-Code 34](#_Toc73109931)

[Abbildung 12: Codestruktur UI-Code nach Refactorings 36](#_Toc73109932)

[Abbildung 13: Ergebnis Extract Method Klasse Vector2Extensions 38](#_Toc73109933)

[Abbildung 14: UML-Klassendiagramm CoordinateSystem 42](#_Toc73109934)

[Abbildung 15: UML-Klassendiagramm für MockPlayer 44](#_Toc73109935)

[Abbildung 16: UML Klassendiagramm für MockGame 45](#_Toc73109936)

[Tabelle 1: LOC vor Refactorings 4](#_Toc73109937)

[Tabelle 2: Getting Started 5](#_Toc73109938)

[Tabelle 3: Ubiquitous Language für die Kerndomäne 7](#_Toc73109939)

[Tabelle 4: Ubiquitous Language für die generische Domäne 8](#_Toc73109940)

[Tabelle 5: Beziehungen zwischen Kontexten 8](#_Toc73109941)

[Tabelle 6: Value Objects 11](#_Toc73109942)

[Tabelle 7: Aggregates und Root Entities 12](#_Toc73109943)

[Tabelle 8: Betroffene Methoden für "Sichtbarkeit erhöhen" 26](#_Toc73109944)

[Tabelle 9: Code Coverage (initial) 39](#_Toc73109945)

[Tabelle 10: Code Coverage (mit Testkonzept) 40](#_Toc73109946)

[Tabelle 11: Betrachtung ATRIP-Regeln 46](#_Toc73109947)

[Quellcode 1: Highscore Mapping Code der Adapter Schicht 21](#_Toc73109948)

[Quellcode 2: Adapter-Interface für Client mit TypeScript-Technologie 22](#_Toc73109949)

[Quellcode 3: Aktionsfindung des RandomNotKillingItSelfPlayers 31](#_Toc73109950)

[Quellcode 3: Aktionsfindung des RandomNotKillingItSelfPlayers (Refactored) 32](#_Toc73109951)

[Quellcode 3: UI-Code für die Behandlung von falschen Eingaben 35](#_Toc73109952)

[Quellcode 4: Beispiel: Kontrollierbare Methode eines Mock-Objekts 44](#_Toc73109953)

# 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

Im Folgenden werden die Programmierprinzipen DOLID, GRASP und DRY in Bezug auf den vorliegenden Quellcode betrachtet. Hierbei gilt zu beachten, dass das Projekt ohne die bewusste Einhaltung dieser Prinzipien entwickelt wurde. Die folgende Analyse soll prüfen, ob und in welchem Maß die Prinzipien eingehalten wurden. Die Prinzipien an sich werden nicht explizit erklärt, sondern nur betrachtet, weswegen ein Grundverständnis hierfür vorausgesetzt wird. Ebenfalls würde es den Rahmen sprengen, in dieser Ausarbeitung für alle Prinzipien den gesamten Quellcode zu betrachten. Das gesamte Projekt wurde selbstverständlich analysiert, hier werden dementsprechend repräsentative und markante (positiv oder negativ) Beispiele aufgeführt und wenn möglich begründet.

## SOLID

### Single Responsibility Principle

Das SRP wird zunächst auf ebene der Namespaces betrachtet. In C# ist ein Namespace mit einem Package in java zu vergleichen. Die folgende Tabelle enthält die Namespaces des Projekts.

|  |
| --- |
| **Namespace** |
| WeepingSnake.ConsoleClient.Navigation |
| WeepingSnake.WebService.Controller |
| WeepingSnake.Game |
| WeepingSnake.Game.Geometry |
| WeepingSnake.Game.Person |
| WeepingSnake.Game.Player |
| WeepingSnake.Game.Player.ComputerPlayer |
| WeepingSnake.Game.Structs |
| WeepingSnake.Game.Utility.Extensions |
| WeepingSnake.Game.Utility.Logging |

Tabelle 8: Namespaces

Durch die Unterteilung in verschiedene Namespaces kann das SRP auf dieser Ebene zum Großteil eingehalten werden. Hierbei ist jedoch die „Responsibility“ sehr weit gefasst, sodass auch die tieferen Ebenen untersucht werden müssen. Innerhalb des Namespaces WeepingSnake.ConsoleClient.Navigation wird neben der Navigation auch die Aufgabe „Darstellung“ und „Benutzereingaben“ übernommen, sodass hier eine Verletzung des SRP deutlich wird.

Auf der Ebene der Klassen wird das Prinzip ebenfalls an den meisten stellen umgesetzt, indem oft Aufgaben von ausgelagerten Klassen oder Methoden übernommen werden. Ein Negativbeispiel ist jedoch die Klasse Game. Hier werden sowohl aktiv die Spieler verwaltet wie auch das Spielbrett an sich. Der Grund hierfür ist, dass die Spieler und das Spielfeld beide Teile eines Spiels sind und die Funktionalitäten bei der Entwicklung als zusammengehörig aufgefasst wurden. Hier ist jedoch eine Trennung möglich, indem sich die Klasse Game z. B. einen „GameController“ und einem „BoardController“ verwendet, welche die jeweiligen Verantwortlichkeiten übernehmen.

Auf der Ebene der Methoden wird SRP teilweise umgesetzt, jedoch lassen einige (vor allem lange) Methoden erkennen, dass hier zu viel Verantwortlichkeit an einer Stelle vereint ist. Ein Beispiel ist hierfür das zufällige Erzeugen von Computerspielern. Unter anderem dieser Fall wird in Kapitel *7.1 Long Method* genauer betrachtet.

### Open Closed Principle

Das OCP wird durch die Verfügbarkeit der sog. Erweiterungsmethoden in C# unterstützt. Diese Technik ermöglicht es, an bestehende Klassen (auch von externen Frameworks) Methoden hinzuzufügen, welche wie herkömmliche Methoden verwendet werden können. Damit diese Erweiterungsmethoden jedoch effizient eingesetzt werden können, ist es notwendig, dass sie auf einer höheren Abstraktionsebene arbeiten. Im Fall der Computerspieler ist dies gegeben, da die verschiedenen Computerspieler mithilfe des Interfaces IComputerPlayer abstrahiert werden. Ein weiteres positiv-Beispiel ist das Spielbrett, bei welchem das abstrakte Koordinatensystem als Ansatzpunkt für Erweiterungen verwendet werden kann.

Diese beiden beschriebenen Stellen erfüllen jedoch als einzige das OCP. Bei anderen Klassen fehlt die Abstraktion (weil sie bisher nicht nötig war). Beispiele hierfür sind die Klassen Game oder Player. Vor allem für die Klasse Player ist es möglich, dass in Zukunft verschiedene Typen existieren, sodass hier eine Abstraktion sinnvoll ist, sodass das OCP in Zukunft eingehalten wird.

### Liskov Substitution Principle

Auf dem Quellcodestand vor den Refactorings wird wenig Vererbung eingesetzt. Die einzige stelle ist das Spielbrett (Board), welches von der abstrakten Klasse CoordinateSystem erbt. Hierbei wird das LSP zwangsläufig eingehalten, da Board die einzige Unterklasse von CoordinateSystem ist. Nach den Refactorings wird die Vererbung intensiver eingesetzt. Somit werden z. B. die Pages für die CLI-Darstellung mithilfe von Vererbung abstrahiert. Hierbei wurde dementsprechend explizit auf die Einhaltung von LSP geachtet.

### Interface Segregation Principle

Die Einhaltung des ISP ist nicht auf den ersten Blick ersichtlich, weswegen alle Interfaces genauer betrachtet werden (siehe hierbei jeweils zugehöriges UML-Klassendiagramm).

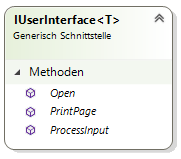
Das IUserInterface gibt das Verhalten für eine Page in der UI vor und besteht hierzu aus Open, ProcessInput und PrintPage. Dieses Interface könnte zwar noch einmal separat in die Bereiche User-Input (ProcessInput) und User-Output(Open, PrintPage) unterteilt werden, diese Feingranularität wäre jedoch wieder grenzwertig, da die Interfaces zwar spezifisch wären, es jedoch auch die Möglichkeit einer Page gäbe, welche keinen Input ermöglicht.

Abbildung 4: UMl-Klassendiagramm IUserInterface

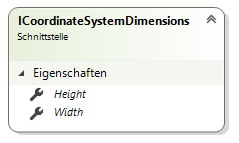
Das Interface ICoordinateSystemDimestions gibt lediglich die Dimensionen eines Koordinatensystems (Höhe und Breite) an, sodass hier das Interface Segregation Principle als erfüllt betrachtet werden kann.

Abbildung 5: UML-Klassendiagramm ICoordinateSystemDimensions

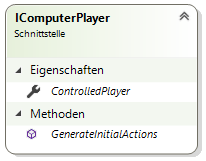
Zuletzt wird das Interface IComputerplayer betrachtet. Hier wird ein Getter und Setter vorgegeben, um einen zu kontrollierenden Spieler festzulegen und eine Methode, um initiale Aktionen abzufragen (falls Aktionen bereits vor Spielstart feststehen. Z. B. Für einen Spieler der sich dauerhaft nach rechts dreht). Hierbei wurde das ISP umgesetzt, da eine weitere Unterteilung nicht sinnvoll wäre.

Abbildung 6: UML-Klassendiagramm IComputerPlayer

Für das gesamte Projekt kann somit behauptet werden, dass das ISP eingehalten wurde. Wenn auch an keiner Stelle mehr als ein einzelnes Interface implementiert wurde, sind die verfügbaren Interfaces nicht zu groß. In diesem Dokument entstehen aufgrund von einigen Refactorings weitere Interfaces, für welche bei der Umsetzung das ISP separat betrachtet / begründet wird (siehe *8.3.2 MockPlayer*).

### Dependency Inversion Principle

Die Einhaltung des DIP ist nicht gegeben. Hierfür werden explizit die Klassen Game und die Klasse Player betrachtet.

Game hängt von Player und von GameBoard ab. Das Problem hierbei ist, dass Game die verschiedenen Player verwaltet und es so eine starke Kopplung zu der Implementierung von Player gibt. Des Weiteren hängt Player von Person und von Game ab. Vor allem die zirkuläre Abhängigkeit zwischen Player und Game stellt ein großes Problem dar. Somit wird DIP nicht umgesetzt. Der Grund für diese Nichteinhaltung von DIP ist vermutlich auf Unwissenheit bei der Entwicklung zurückzuführen. In den folgenden Kapiteln wird durch verschiedene Refactorings die Einhaltung DIP an mehreren Stellen sichergestellt. Beispiele hierfür sind, dass Game nicht GameBoard abhängt, sondern von dessen abstrakter Oberklasse CoordinateSystem. Ebenfalls werden die Abhängigkeiten von Player und Game aufgelöst, indem interfaces eingesetzt werden.

## GRASP

### High Cohesion

Der Zusammenhalt der Elemente ist im vergleich zur geringen Kopplung stärker gegeben. Der Zusammenhang besteht hierbei zum einen innerhalb eines Namespaces (siehe *3.1.1 Single Responsibility Principle*) sowie innerhalb von den jeweils enthaltenen Klassen. Entsprechend der Betrachtung von SRP ist auch der Zusammenhalt der Komponenten gegeben. Hier gilt es wiederum die Klasse Player positiv und die Klasse Game (aufgrund der Verwaltung von Spielern und Spielfeld) negativ hervorzuheben. Die Kohäsion in der Klasse Game ist jedoch trotz der Verantwortlichkeit für Player und GameBord hoch, da Game diese beiden getrennten Bereiche nicht nur verwaltet, sondern auch zusammenführt. Somit ist die hohe Kohäsion für Game trotz der Verantwortlichkeit für mehrere Dinge gegeben, da die beiden getrennten Verantwortlichkeiten miteinander verknüpft sind. Abgesehen von diesen beiden Klassen ist aufgrund der guten Umsetzung von SRP auch die hohe Kohäsion an den anderen Stellen (insbesondere sind GameBoard und GameDistance repräsentativ) im Quellcode gegeben.

Trotz der (mit Ausnahmen) hohen Kohäsion, wird Low Coupling nicht ausreichend unterstützt. Das Problem hierbei ist gleichermaßen, dass innerhalb der Klassen konkrete Implementierungen und nicht deren Abstraktionen verwendet werden.

### Low Coupling

Die Kopplung ist nicht gering. Dies ist wie bei der Betrachtung von DIP vor allem an den Klassen Player & Game zu erkennen. Diese beiden Klassen sind stark aneinander und auch an andere Klassen (GameDistance, PlayerDirection, GameCoordinate, …) gekoppelt und somit direkt von deren Implementierungen abhängig. Der Computerplayer ist ein Beispiel für Low Coupling, da hier über das interface auf die tatsächlichen Implementierungen zugegriffen wird.

Für die UI-Pages wäre eine geringe Kopplung möglich, da hier alle Pages ein Interface implementieren. So wie die Navigation jedoch derzeit implementiert ist (die Pages rufen die folgenden Pages selbst auf), ist die geringe Koppelung nicht gegeben, da jede Page auf die Spezielle Implementierung der folgenden Page verweist.

Wie bereits betrachtet, wird an anderen Stellen im Quellcode nicht auf die Verwendung von Interfaces / abstrakten Oberklassen gesetzt, sodass hier die Elemente stark an Implementierungen gekoppelt sind.

Diese Abhängigkeiten werden durch das Refactoring Extract Interface für Player und Game aufgelöst, sodass hier von Low Coupling gesprochen werden kann (die Refactorings erfolgen mit dem Ziel, dass der Quellcode besser testbar ist).

### Information Expert

Information Expert wird in dem gesamten Projekt korrekt und gut umgesetzt (z. B. GameBoard, PlayerAction, PlayerOrientation, PlayerDirection führen jeweils Operationen aus, welche die gehaltenen Daten betreffen), sodass konkrete Beispiele nicht erläutert werden. Die durchgehende Einhaltung liegt vermutlich darin begründet, dass dieses Prinzip durch das Klassen-Konzept der Objektorientierten Programmierung vorgegeben ist und eine Nichteinhaltung oft mit einem höheren Entwicklungsaufwand einhergeht.

### Creator

Creator wird ebenfalls durchgehend eingehalten. Dies liegt vermutlich daran, dass die erlaubten Zustände für die Initialisierung von Objekten einen Großteil der Möglichkeiten abdeckt und die übrigen Möglichkeiten bei der Entwicklung nicht nahe liegen. Im Folgenden sind einige Beispiele hierfür aufgeführt.

* GameController erzeugt Game, Player und Person, da diese hier verarbeitet werden.
* Game erzeugt ComputerPlayer, da CopmuterPlayer von Game abhängt.
* Player erzeugt PlayerAction, da Player der Informationsexperte (für die Erzeugung) ist.

### Indirection

Indirection wird an mehreren Stellen eingesetzt. Ein Beispiel ist, dass von Game über einen Eventhandler (Vermittler) delegiert wird. Delegiert wird hierbei die Aufgabe, die Clients oder ComputerPlayer über einen neuen Rundenstatus zu informieren und eine durchzuführende Aktion als Antwort zu verarbeiten. Weitere Anwendungen sind zum Großteil Delegationen an verschiedene Strukturtypen (PlayerDirection, PlayerAction, …), welche auch Informationexpert für die delegierte Aufgabe sind.

### Polymorphism

Für Polymorphism gilt wie für das betrachtete LSP, dass wenig Vererbung eingesetzt wurde, das aufgeführte Beispiel ist jedoch auch hierfür repräsentativ. Allgemein ist jedoch ersichtlich, dass Polymorphie nicht wo möglich eingesetzt wurde. Es werden zwei Switch-Expressions verwendet, welche zwar einige Vorteile gegenüber der klassischen Switch-Statements bieten, jedoch trotzdem mithilfe von Vererbung ersetzt werden könnten.

### Controller

Die Klasse GameController ist ein typisches Beispiel für einen Controller. Auf diese Klasse wird von der UI (OfflineConsoleClient) und der WebAPI zugegriffen. Der Controller delegiert die Aufrufe daraufhin an das GameBackend weiter, sodass die grundlegenden Bedingungen für einen Controller eingehalten werden.

Des Weiteren besteht die WebApi selbst aus Controllern, welche Aufrufe an GameController weiterleiten. Hierbei handelt es sich bei den einzelnen Controllern um Use Case Controller, welche im Gegensatz zu dem GameController für eine zusammengehörige Gruppe von Use Cases verantwortlich sind (Account, Game, Highscore). Die genaue Funktionsweiße der WebApi und deren Controller ist in Abschnitt *9 Zusatz: API-Design* beschrieben.

### Pure Fabrication

Die Hilfsklassen werden im Projekt unter dem Namespace Utility zusammengefasst. Hier wird die Mechanik der Erweiterungsmethoden von C# intensiv eingesetzt. Die folgenden Hilfsklassen sind nach der Definition in *2.1 Ubiquitous Language* nicht teil der Problemdomäne:

* EnumerableExtensions
* ListExtensions
* Vector2Extensions
* GameControllerLogger
* GameInformationLogger

Neben diesen Utility-Klassen gibt es jedoch auch Klassen der Problemdomäne, welche Methoden haben, die in eine Hilfsklasse extrahiert werden könnten. Ein Beispiel hierfür ist die Klasse CoordinateSystem, welche den Bresenham-Algorithmus durchführt. Hierbei kann die Klasse CoordinateSystem jedoch auch als „nicht teil der Problemdomäne“ betrachtet werden, da je nach Auslegung lediglich das GameBoard teil der Problemdomäne ist.

### Protected Variations

Mit Interfaces wurde nicht intensiv gearbeitet (siehe *3.1.4 Interface Segregation Principle* und *3.2.1 Low Coupling!!*), weswegen Protected Variations nicht eingehalten wird. Wie bereits beschrieben, wird diese Abhängigkeit von konkreten Implementierungen durch Refactorings behoben, welche in diesem Dokument beschrieben werden. Aus diesem Grund wird in diesem Unterkapitel nicht weiter auf die Abhängigkeit von Implementierungen eingegangen.

## DRY

DRY steht für *Don’t Repeat Yourself* und ist nicht nur auf den Quellcode an sich, sondern auch auf die Logik dahinter anwendbar. Dies bedeutet, dass z. B. mehrere Unit-Tests nicht genau das gleiche Verhalten testen sollten. Solche Duplikate sind schwer zu finden, da diese nicht mithilfe von statischer Codeanalyse aufgedeckt werden können. Es wird ein umfassendes Verständnis des Sourcecodes benötigt. Bei der Durchsicht des Projekts konnten solche ausschließlich logischen Duplikate nicht entdeckt werden. Mit dem PMD Sourcecode Analyzer konnten mehrere Duplikate in Quellcode-Dateien ermittelt werden. Die folgende Tabelle führt die betroffenen Dateien auf

|  |  |
| --- | --- |
| **Datei** | **Anzahl duplizierte Zeilen** |
| RandomPlayer.cs & RandomNotKillingItselfPlayer.cs | 36 |
| HighscoreEntry.cs & Person.cs | 35 |
| UserPage.cs | 17 |
| StartPage.cs & UserPage.cs | 17 |
| ShowHighscoresPage.cs | 17 |
| GameControllerLogger.cs & GameInformationLogger.cs | 14 |
| GameControllerLogger.cs | 10 |
| Vector2Extensions.cs | 7 |
| GameControllerLogger.cs | 6 |
| CoordinateSystemTests.cs | 5 |
| CoordinateSystemTests.cs | 5 |

Tabelle 9: Quellcode Duplikationen

Die Duplikate werden im Abschnitt *7.2 Duplicated Code* genauer betrachtet und ggfs. aufgelöst. Zum Großteil handelt es sich hierbei um versehentliche (inadvertent) Duplikate, da bei der Entwicklung nicht auf das Einhalten von Programmierprinzipen geachtet wurde.

# 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 7: 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 8: 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 9: 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 10: 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

In diesem Kapitel sind Abhängigkeiten im Quellcode dargestellt, welche Änderungen am System problematischer gestalten und Probleme bei Unit Tests darstellen. Da keine konkreten Änderungen an dem Quellcode vorgenommen werden, werden diese Änderungen hypothetisiert. D. h., dass die potenzielle Änderung beschrieben und Testpunkte theoretisch aufgezeigt werden. Daraufhin werden die Abhängigkeiten identifiziert und gebrochen. Das Schreiben der Tests für die hypothetische Änderung sowie die hypothetische Änderung an sich werden daraufhin nicht betrachtet, da sich dieses Kapitel auf das Brechen der Abhängigkeiten fokussiert[[3]](#footnote-3).

**Extract Interface für die Klasse Game**

Die Klasse Game wird von mehreren Methoden benötigt, da hier die Steuerung eines einzelnen Spiels stattfindet. Um die Methoden, welche die Klasse Game verwenden unabhängig von der tatsächlichen Implementierung testen zu können, wird hierzu ein Mock- oder Fake-Objekt benötigt, dessen Erstellung jedoch sehr aufwendig ist.

Dieser hohe Aufwand liegt darin begründet, dass es sich bei Game um eine sog. „Schwierige Klasse“ handelt. Die abhängigen Objekte (Board, Player) sind nur schwer zu erzeugen. Außerdem hat der Konstruktor den Seiteneffekt, dass direkt mehrere Computerspieler initialisiert werden. Die Problematik betrifft somit alle stellen, an welchen die Klasse Game verwendet wird und aufgrund von potenziellen Änderungen Unittests relevant sind. Die Testpunkte sind diesbezüglich bei den einzelnen Methoden anzusiedeln, welche die Klasse Game verwenden.

Um diese Abhängigkeit von der Klasse Game zu brechen, wird das Interface IGame extrahiert und verweise im Quellcode wurden auf dieses Interface geändert. In diesem interface sind die notwendigen Methoden definiert, welche von den zu testenden externen Methoden verwendet werden. Da die Klasse Game jedoch oft referenziert wird, sind hierbei alle public Methoden (bis auf Überschreibungen wie GetHashCode, Equals, …) betroffen. Es wird trotzdem das Verfahren Extract Interface verwendet, da hierdurch eine starke Kontrolle über Mock-Objekte möglich ist, sodass bei Tests gezielter das tatsächlich zu testenden Verhalten überprüft werden kann (Detailliertere Beschreibung hierzu ist in Abschnitt *8.3.3 MockGame* enthalten)

Die Änderung erfolgte mit Commit [f9eb1c5](https://github.com/SilasDerProfi/weeping-snake/commit/f9eb1c533640128b8e92d431fc1a242225323397)

**Sichtbarkeit bei Struktur-Typen erhöhen**

Der in C# verfügbare Werttyp struct unterscheidet sich von einer Klasse, indem er immer Parameterlos initialisierbar ist, niemals null sein kann und unveränderlich ist. Einige dieser Strukturtypen verwenden für Methoden den internal-Zugriffsmodifizierer, sodass auf diese Methoden nicht von einem separaten Test-Projekt zugegriffen werden kann. Bei diesem Problem handelt es sich zwar nicht per se um eine zu lösende Abhängigkeit, die Änderung wird jedoch durchgeführt, um das Erstellen von Unit Tests zu ermöglichen.

Damit diese Methoden auch als Testpunkte verwendet werden können, wird die Sichtbarkeit auf public erhöht. Diese Änderung der Sichtbarkeit bringt üblicherweise einige Fallstricke mit sich. Da es sich bei den betroffenen typen jedoch nicht um Klassen, sondern um structs handelt, bietet sich die Erhöhung der Sichtbarkeit an. Aufgrund der Eigenschaft Unveränderlichkeit sind Nebeneffekte innerhalb des structs ausgeschlossen. Aus diesem Grund ist die Funktion der Methoden immer, dass ein neues struct mit einer gewissen Veränderung erstellt und zurückgegeben wird. Diese Funktionalitäten können auch in anderen Projekten (z. B. für die UI oder intelligente Computerspieler) sinnvoll verwendet werden, da es sich hierbei um Berechnungen handelt, welche für die Darstellung / Entscheidungsfindung verwendet werden können. In der folgenden Tabelle sind die betroffenen Methoden aufgelistet.

|  |  |  |
| --- | --- | --- |
| **Methode** | **Anmerkung** | **Commit** |
| PlayerOrientation(  position,  direction)  (Parametrisierter Konstruktor) | Der parameterfreie Konstruktor ist bei einem struct sowieso immer public, daher kann der parametrisierte Konstruktor auch public sein. | [a274b33](https://github.com/SilasDerProfi/weeping-snake/commit/a274b330d91d1a4099bab7820538a4cdfb3a8920) |
| PlayerOrientation  .ApplyAndMove(action) | Berechnet die Position, welche beim Ausführen einer Aktion erreicht wird. | [a274b33](https://github.com/SilasDerProfi/weeping-snake/commit/a274b330d91d1a4099bab7820538a4cdfb3a8920) |
| PlayerDirection  .Apply(action) | Berechnet die Blickrichtung, welche beim Ausführen einer Aktion resultiert. | [a274b33](https://github.com/SilasDerProfi/weeping-snake/commit/a274b330d91d1a4099bab7820538a4cdfb3a8920) |
| GameCoordinate  .Translate(direction) | Berechnet die Position, welche bei einem Tick mit gegebener Richtung erreicht wird. | [a274b33](https://github.com/SilasDerProfi/weeping-snake/commit/a274b330d91d1a4099bab7820538a4cdfb3a8920) |

Tabelle 10: Betroffene Methoden für "Sichtbarkeit erhöhen"

**Construction BLOB für die Testbarkeit von Ein- und Ausgaben mithilfe von Adapt Parameter**

Die Ein- und Ausgaben von dem ConsoleClient werden derzeit über die statische Klasse Console des .NET Frameworks abgehandelt. Diese Klasse bietet keine Nahtstellen, sodass Benutzereingaben nicht über Unittests gesteuert werden können. Das folgende UML-Klassendiagramm zeigt die Ausgangssituation.

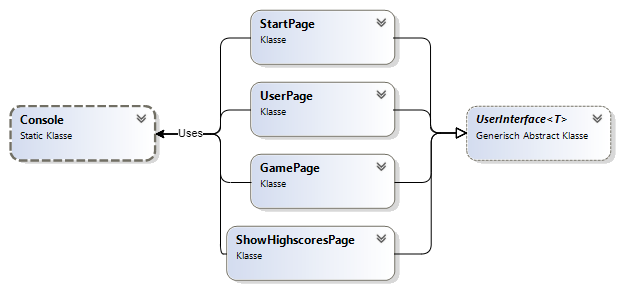


Abbildung 11: UML-Klassendiagramm UI-Code abhängig von .NET Framework

Durch eine Wrapper-Klasse werden diese Abhängigkeiten gebrochen. Zusätzlich wird für diesen Wrapper *Extract Interface* angewendet, um das Mocking des Wrappers zu ermöglichen. Im letzten Schritt muss der Legacy-Code angepasst werden, sodass dieser für die Ein- und Ausgaben auf eine beliebige Implementierung des extrahierten Interfaces zugreift. Hierfür wird *Adapt Parameter* angewendet, sodass die verschiedenen Page-Klassen direkt im Konstruktor die Instanz für CLI-IO-Operationen erhalten.

Für Unit Tests kann die UI nun mit dem CLI-IO-Mock-Objekt aufgerufen werden, um die verschiedenen Testpunkte zu erreichen.

Der Nachteil hierbei ist, dass der Legacy-Code angepasst werden muss, sodass nichtmehr die Klasse Console, sondern die Wrapper-Klasse (bzw. das extrahierte Interface) verwendet wird. Dieser Nachteil ist jedoch durch die entstehende Kontrolle über die Ein- und Ausgaben zu vernachlässigen. Ein weiterer Nachteil ist, dass sich die Signatur ändert, dieser ist jedoch ebenfalls vertretbar, da der UI Code nicht von außerhalb aufgerufen wird und somit keine anderen Projekte betroffen sind. Das folgende UML-Klassendiagramm zeigt das Resultat der Änderungen.[[4]](#footnote-4)

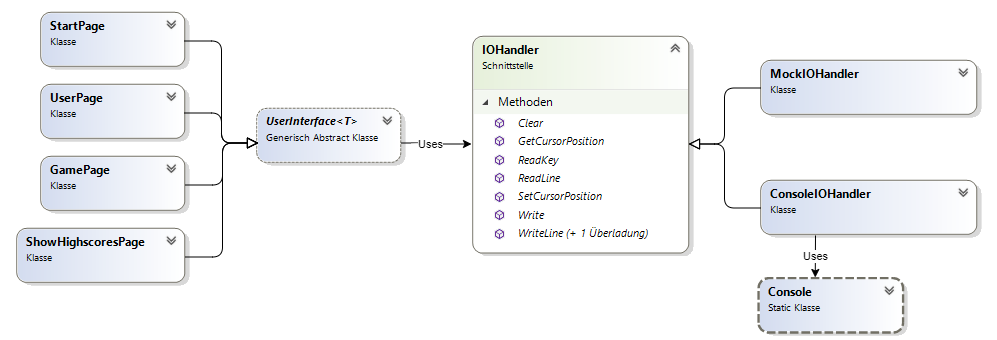


Abbildung 12: UML-Klassendiagramm testbarer UI-Code unabhängig von .NET Framework

Die Änderung erfolgte mit Commit [2256df6](https://github.com/SilasDerProfi/weeping-snake/commit/2256df610c1526217d21fb35f3c08f60e84c4a14)

**EVTL:**

* Extract interface player
* Configurationentry -> public (<https://github.com/SilasDerProfi/weeping-snake/commit/58fb81ad84c875805c20a13b6b26694e07fb9991>, Ist ok, weil das ein struct sein könnte -> muss gemacht werden.).
* Computerplayer klasse war internal. (evtl internal / protected und subclass furs testen…)
* PlayerAction.apply (ist zwar struct, hat aber Auswirkungen auf Player. Fragwürdig ob das ok ist, hier wird zunächst versucht die abhängigkeit zu lösen), verwendung von MockPlayer (das geht schon…)

# Refactoring

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

## Long Method

Im Folgenden wird der Code Smell Long Method an mehreren Stellen betrachtet. Die Problemstellung hierbei ist, dass eine lange Methode zu viele Aufgaben übernimmt und darunter die Lesbarkeit und Wartbarkeit des Quellcodes leidet.

**RandomNotKillingItselfPlayer**

Bei dieser Klasse handelt es sich um eine Implementierung für einen Computergesteuerten Spieler, welcher in jeder Runde zufällig eine Aktion wählt, welche ihn nicht aus dem Spiel ausscheiden lässt (wenn möglich).

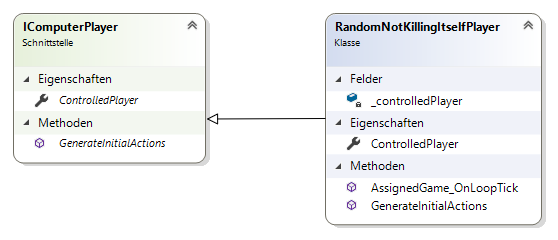


Abbildung 13: UML-Klassendiagramm RandomNotKillingItselfPlayer

An dem zunächst schwerfälligem Namen der Klasse wird keine Änderung vorgenommen, da dieser die Funktion der Klasse sehr gut beschreibt. Da jedoch mit dem beschriebenen Verhalten dieser Klasse eine gewisse Komplexität ausgeht, ist es hierbei nicht sinnvoll die gesamte Funktionalität innerhalb der Eventhandler-Methode AssignedGame\_OnLoopTick abzuhandeln. Hierfür wird zunächst anhand des Quellcodes ermittelt, was es für Teilfunktionalitäten gibt.

1. public void AssignedGame\_OnLoopTick(List<Geometry.GameDistance> newPaths)
2. {
3. var board = \_controlledPlayer?.AssignedGame?.GameBoard;
4. if (board != null)
5. {
6. var randomAction = Enum.GetValues<PlayerAction.Action>().Random();
7. int retries;
8. for (retries = 0; retries < 100; retries++)
9. {
10. var newOrientation = \_controlledPlayer.Orientation
11. .AplyAndMove(randomAction);
12. if (newOrientation.Position.X >= 0
13. && newOrientation.Position.Y >= 0
14. && newOrientation.Position.X < board.Width
15. && newOrientation.Position.Y < board.Height)
16. {
17. break;
18. }
19. else
20. {
21. randomAction = Enum.GetValues<PlayerAction.Action>()
22. .Random();
23. }
24. }
25. \_controlledPlayer.AddAction(randomAction);
26. }
27. }

Quellcode 3: Aktionsfindung des RandomNotKillingItSelfPlayers

Zu Beginn wird der korrekte Status des Spiels sichergestellt (Zeile 8). In Zeile 10 wird eine zufällige Aktion bestimmt. In Zeile 13 steht der Kopf einer for-Schleife, welche bis zu 100 mal durchläuft. In jedem durchlauf wird geprüft, ob die derzeit gesetzte Aktion den Spieler tötet (Zeile 18). Falls ja wird eine neue zufällige Aktion generiert. Falls der Spieler den Zug überleben würde, wird die Aktion beibehalten und als nächste Aktion gesetzt. Das hier eingesetzte „Trial and error“-Verfahren um eine Aktion zu finden ist sehr ineffizient, sodass dieses ebenfalls angepasst wird. Bei Refactorings muss das Gesamtverhalten gleich bleiben, in diesem Rahmen wird jedoch ein anderes Verfahren für die Bestimmung der zufälligen Aktion eingesetzt. In diesem Rahmen bleibt die Funktionalität (eine zufällige, gültige Aktion wählen) jedoch gleich, auch wenn sich die Wahrscheinlichkeitsverteilung für die jeweiligen Aktionen ändern kann. In dem folgenden Quellcodeausschnitt sind die beiden extrahierten Methoden GetActionsInRandomOrder und IsActionValidForBoardDimensions enthalten. Hierbei werden nun die möglichen Aktionen in eine zufällige Reihenfolge gesetzt um diese Liste anschließend zu iterieren, bis eine gültige Aktion gefunden wurde. Das UML-Klassendiagramm ist in diesem Fall trivial und daher nicht separat abgebildet (zwei zusätzliche Methoden)

1. public void AssignedGame\_OnLoopTick(List<Geometry.GameDistance> newPaths)
2. {
3. var actions = GetActionsInRandomOrder();
4. var action = actions.FirstOrDefault(IsActionValidForBoardDimensions);
5. \_controlledPlayer.AddAction(action);
6. }
7. private PlayerAction.Action[] GetActionsInRandomOrder()
8. {
9. var actions = Enum.GetValues<PlayerAction.Action>();
10. var random = new Random();
11. return actions.OrderBy(a => random.Next()).ToArray();
12. }
13. private bool IsActionValidForBoardDimensions(PlayerAction.Action action)
14. {
15. var board = \_controlledPlayer.AssignedGame?.GameBoard;
16. var position = \_controlledPlayer.Orientation.ApplyAndMove(action)
17. .Position;
18. return board != null && position.X >= 0 && position.Y >= 0
19. && position.X < board.Width && position.Y < board.Height;
20. }

Quellcode 3: Aktionsfindung des RandomNotKillingItSelfPlayers (Refactored)

Die Änderung erfolgte mit Commit [807b3f7](https://github.com/SilasDerProfi/weeping-snake/commit/807b3f750482971f3c88a7c30f7a90d7a6500e68)

**Game.ApplyOneActionPerPlayer**

Die Methode Game.ApplyOneActionPerPlayer ist dafür verantwortlich, dass die von den Spielern gewählte Aktionen durchgeführt werden. Zusätzlich werden jedoch auch nachträglich alle Spieler über den neuen Spielstatus informiert. Dieses Problem wird als Long Method Code Smell aufgefasst, da die Methode mehr aufgaben übernimmt als ihr Name preisgibt. Aus diesem Grund wird hierfür das Refactoring Rename Method eingesetzt, da in der Methode an sich nicht zu viele Programmcode-Zeilen, sondern zu viel Funktionalität ist. Der neue Name der Methode ist länger, indem „AndNotify“ angehängt wurde (ApplyOneActionPerPlayerAndNotify), deutet jedoch direkt auf die zugrundeliegende Funktionalität hin, sodass nicht „versehentlich“ alle Spieler benachrichtigt werden, obwohl dies nicht gewünscht ist.

Die Änderung erfolgte mit Commit [4be5813](https://github.com/SilasDerProfi/weeping-snake/commit/4be5813c7a93a3e739be6d820356ec65eb2215b4)

* Code Smells identifizieren
  + Game Configuration constructor
  + Create computerplayer
  + GameDistance Konstruktor
  + Game Join
  + Game Leave
  + Game board randomstart coordinate

## 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.

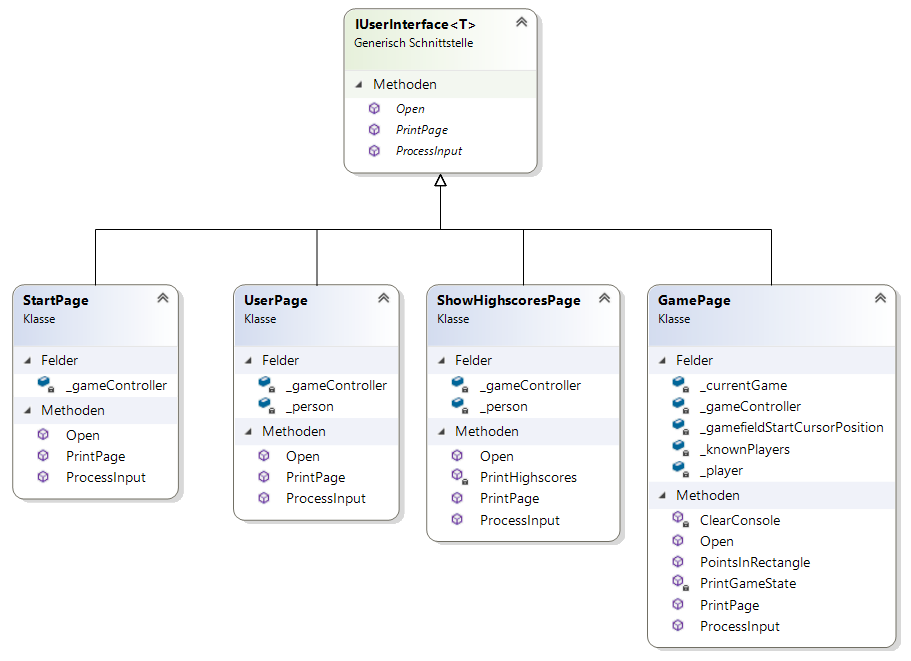


Abbildung 14: Codestruktur UI-Code

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.

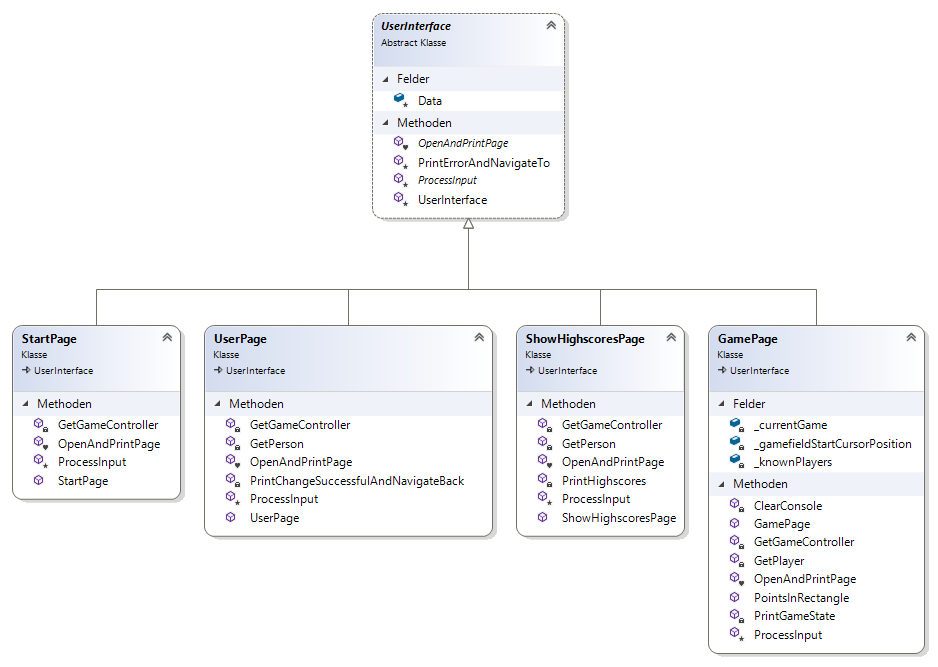


Abbildung 15: Codestruktur UI-Code nach Refactorings

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. OpenAndPrintPage ist dabei das Ergebnis eines Rename Method-Refactorings, da die Methode Open durch den Konstruktor der Oberklasse obsolet ist und PrintPage somit auch die Aufgabe Open übernimmt . 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).

**Vector2Extensions**

Die Klasse Vector2Extensions bietet Erweiterungsmethoden für die Klasse Vector2 des .net Frameworks an, welche im Kontext von WeepingSnake benötigt werden.

Bereitgestellt werden unter anderem die Methoden Decrease und Increase, welche die Länge des Vektors verändern, sodass sich der Spieler schneller oder langsamer bewegt (Richtung und Geschwindigkeit wird durch einen zweidimensionalen Vektor angegeben). Da der Code sich bis auf die Rechenoperation (Subtraktion / Addition) nicht unterscheidet wird die Erweiterungsmethode Add(Vector2 vector, int lengthSummand) extrahiert. Somit kann diese Methode von Increase mit einem positiven Wert oder Decrease mit einem negativen Wert aufgerufen werden.

Des Weiteren wird die Berechnung, um einen Vektor auf eine gewünschte Länge zu bringen, in mehreren Methoden durchgeführt (Add, UnitVector, DefaultDistanceVector). Da es sich hierbei um eine mathematische Formel handelt, welche auch in Zukunft nicht geändert wird, hat dieser duplizierte Code nur in geringem Ausmaß negativen Auswirkungen auf die Wartbarkeit. Um dennoch die Wiederverwendung des Quellcodes zu erhöhen und die Komplexität zu verringern wird hierfür die Erweiterungsmethode ChangeLengthTo(float length) hinzugefügt und in den entsprechenden Methoden referenziert. Die folgende Abbildung in Anlehnung an den UML-Klassendiagramm-Standard visualisiert die beschriebenen Änderungen.

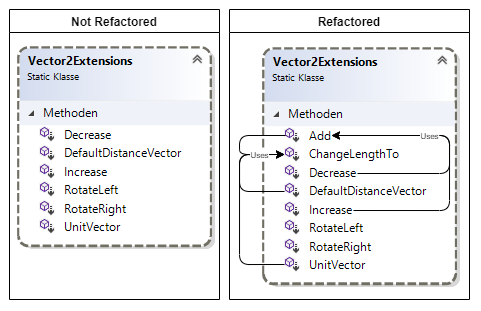


Abbildung 16: Ergebnis Extract Method Klasse Vector2Extensions

Die Änderungen erfolgten mit dem Commit [72962a0](https://github.com/SilasDerProfi/weeping-snake/commit/72962a0eec432195a20e72a94ae313e5537999e9).

## 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 11: 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 12: 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.

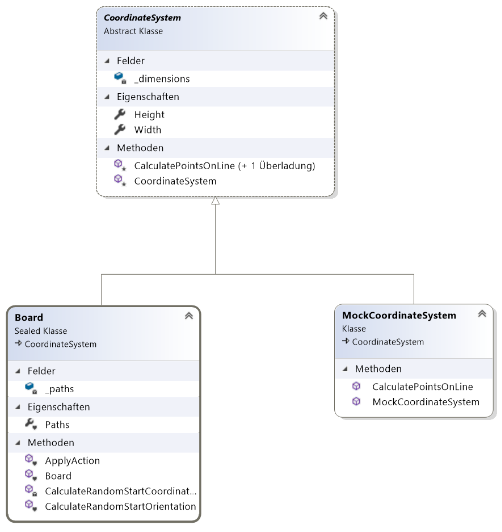


Abbildung 17: UML-Klassendiagramm CoordinateSystem

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[[5]](#footnote-5)). 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.

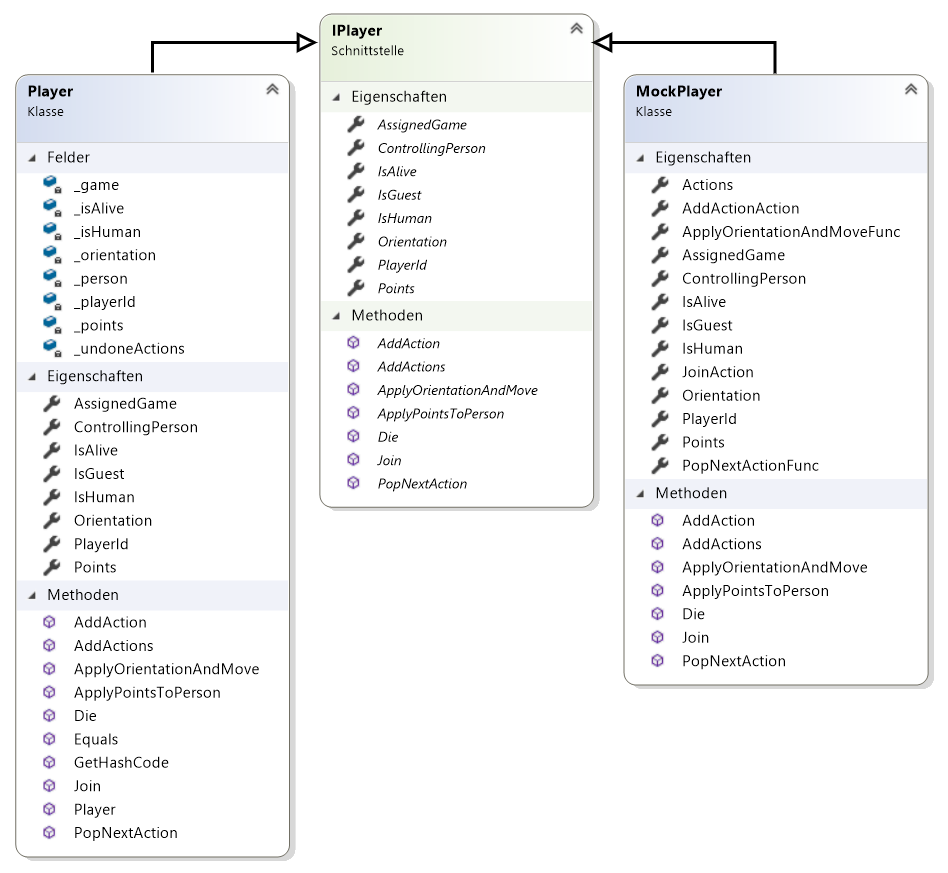


Abbildung 18: UML-Klassendiagramm für MockPlayer

### 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 (Hier wird ebenfalls die bewusste Verletzung des ISP in Kauf genommen). In dem folgenden UML-Diagramm ist das Ergebnis dargestellt.

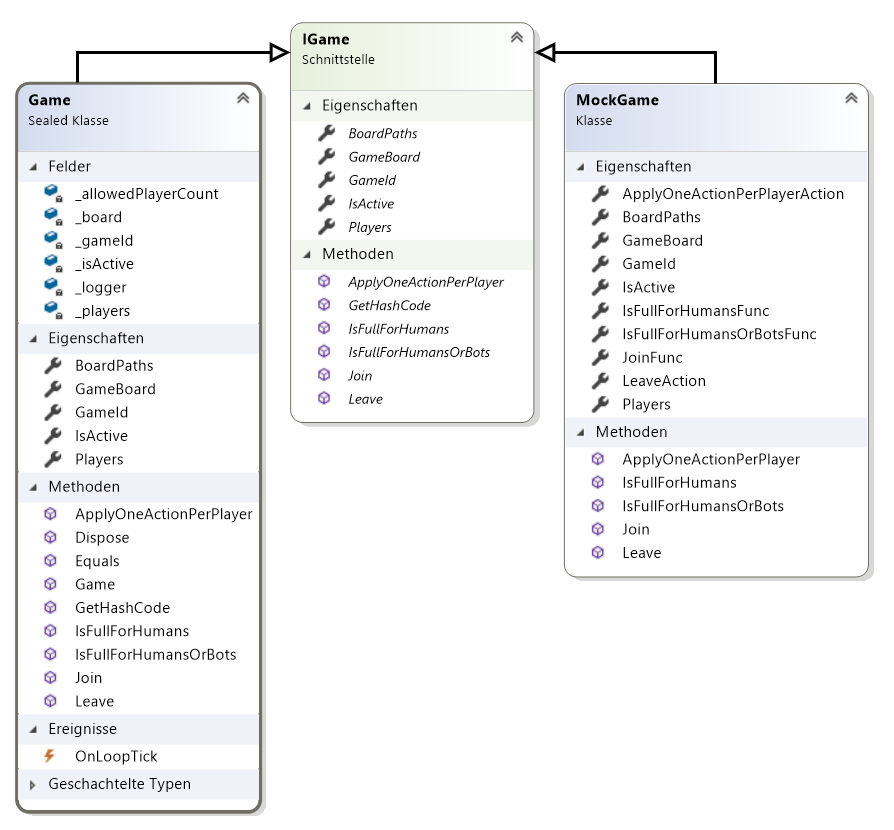


Abbildung 19: UML Klassendiagramm für MockGame

## 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 13: Betrachtung ATRIP-Regeln

# Zusatz: API-Design

Im Folgenden wird die Web-Schnittstelle betrachtet. Diese besteht aus 10 HTTP-Endpunkten und erlaubt die Kommunikation mit dem Game-Backend. Gegenstand der Übertragung sind dabei Informationen über die Spiele und die Benutzeraccounts (incl. Highscores).

## Design

Die Restful-API bietet die folgenden http-Endpunkte. Bei diesem Design werden wie für Restful-Services üblich die Ressourcen über eine ID identifiziert und daten werden über den Body gesendet. Für die verschiedenen Bereiche stehen separate Controller zur Verfügung, um das Routing innerhalb der API möglichst intuitiv zu gestalten. Die Antworten auf Anfragen werden mithilfe der angegebenen HTTP-Response-Codes klassifiziert, hierbei ist jedoch zu beachten, dass die Responsecodes 5xx: Server Error aufgrund von unerwarteten Fehlern immer auftreten können. Die API ist jedoch nicht für die Passwort-Sicherheit zuständlich. Die Verantwortlichkeit, dass Passwörter als Hash, mit Salt, etc. übertragen werden, liegt bei dem System an sich und ist von der API unabhängig. Die Kommunikation beruht darauf, dass die personId (Identifiziert einen Account) nur dem Account-Besitzer bekannt ist. Um eine Dokumentation für die API automatisch generieren zu lassen, wurden in diesem Programmbestandteil die API-Methoden mit Doc-Kommentaren ausgestattet. Die folgende Tabelle zeigt die HTTP-Endpunkte auf.

|  |  |  |
| --- | --- | --- |
| **API** | | **Request body** |
| **GET /api/time**  (Aktuelle Serverzeit, falls für den Client eine Synchronisierung dem nötig ist) | | n. a. |
| Response Body | 200: {aktuelle Serverzeit} | |
| **PUT /api/account**  (Registrieren eines neuen Benutzers) | | {  eMailAddress,  userName  password,  retypedPassword  } |
| Response Body | 200: {personId}  400: Email address is already known, or the retyped password is wrong | |
| **GET /api/account**  (Einloggen eines registrierten Benutzers) | | {  eMailAddress,  password  } |
| Response Body | 200: {personId}  400: Invalid combination of Email address and password | |
| **POST /api/account/{personId}/email**  (E-Mail-Adresse eines Registrieren Benutzers ändern) | | Neue E-Mail Adresse |
| Response Body | 200: -  400: The given Email address is invalid or already in use  404: Can't find the Person with the given ID | |
| **POST /api/account/{personId}/password**  (Passwort eines Registrieren Benutzers ändern) | | {  oldPassword,  newPassword,  retypedNewPassword  } |
| Response Body | 200: -  400: Old password is invalid  400: The retyped password is wrong  404: Can't find the Person with the given ID | |
| **PUT /api/game**  (Als Gast einem Spiel beitreten) | | n. a. |
| Response Body | 200: {playerId} | |
| **PUT /api/game/{personId}**  (Als registrierter Benutzer einem Spiel beitreten) | | n. a. |
| Response Body | 200: {playerId}  400: Invalid personId | |
| **API** | | **Request body** |
| **GET /api/game/{playerId}**  (Den Status eines Spiels abfragen, an welchem ein gegebener Spieler teilnimmt) | | n. a. |
| Response Body | 200: { Gamestate JSON notated }  400: Invalid playerId | |
| **POST /api/game/{playerId}**  (Eine Aktion für einen Spieler senden) | | Gewünschte Aktion: (CHANGE\_NOTHING, JUMP, TURN\_LEFT, SLOW\_DOWN, TURN\_RIGHT, SPEED\_UP) |
| Response Body | 200: -  400: Invalid playerId | |
| **GET /api/highscores/{pageNumber}**  (Die Highscores abfragen. In Pakete mit je 20 Einträgen gebündelt, beginnt bei Seite 0) | | n. a. |
| Response Body | 200: { 20 Highscore entries JSON notated }  400: The searched Page is out of range. | |

Tabelle 14: Beschreibung der API

Die Änderungen erfolgten mit dem Commit [84fd3e2](https://github.com/SilasDerProfi/weeping-snake/commit/84fd3e23d4049327a2b7cf2431caa7f27a8a0cbd).

## Analyse

Im Folgenden wird das Design der API anhand der Qualitätsmerkmale nach ISO 9126 analysiert.

**Benutzbarkeit**

Die Benutzbarkeit ist das Zentrale Zeil des API-Designs. Da die API nur 10 http-Endpunkte (minimal) anbietet, ist diese sehr übersichtlich und die Konsistenz (vor allem beim Routing) ist gegeben. Aus diesem Grund ist die API ebenfalls intuitiv verständlich (und damit auch leicht zu lernen), da sehr Stark mit den verschiedenen http-Verben gearbeitet wird, um die Verschachtelung beim Routing möglichst gering zu halten (sodass Client-Code ebenfalls kurz sein kann).

Als Dokumentation steht jedoch nur dieses Dokument zur Verfügung. Hierbei wäre es sinnvoll, eine Dokumentation als GitHub Wiki oder eigene Webseite zu veröffentlichen und auch Beispiele für die Responses (vor allem das Format der JSON-Daten) anzugeben.

Die Fehlerfälle bei falscher Benutzung wird im folgenden Abschnitt Zuverälssigkeit betrachtet. Die Erweiterbarkeit wird daraufhin unter Vollständig und korrekt analysiert.

**Effizienz**

Bei dem Design der API wurde darauf geachtet, dass die Datenübertragung auf ein Minimum begrenzt wird. Somit werden keine Daten doppelt übertragen, wenn dies vermeidbar ist. Eine Ausnahme stellt hierbei das Anfragen des Game-States dar. Hierbei werden immer die Informationen der letzten fünf Runden gesendet, sodass auch neue Teilnehmer direkt auf den aktuellen Status des Spiels zugreifen können. Ein gutes Beispiel für die Skalierbarkeit ist das Abfragen der Highscores. Hier wird die gesamte Liste in kleine Pakete unterteilt, sodass nicht unnötig große Datenmengen übertragen werden müssen und die Anfrage zügig beantwortet werden kann (gutes Zeitverhalten). Die Testbarkeit ist ebenfalls gegeben, da es sich um eine leichtgewichtige API handelt, welche die Anfragen delegiert. Als letzter Teilaspekt der Effizienz wird die Konformität betrachtet. Diese ist gegeben, da durch das eingesetzte Routing die intuitive Benutzbarkeit gegeben ist.

**Zuverlässigkeit**

Die API stellt lediglich eine Schnittstelle zu dem Game-Backend dar, sodass die interne Fehlerbehandlung nicht von der API übernommen wird. Ebenfalls werden Requests, welche im falschen Format sind bereits vom dotnet-Framework abgefangen und nicht an die entwickelten http-Endpunkte weitergeleitet. Inhaltlich falsche anfragen werden korrekt mithilfe der http-Responsecodes behandelt, sodass für den Client eine Fehlerbehandlung möglich ist. Hierdurch ist die Fehlertoleranz zwar eingeschränkt, es kann jedoch von einem Client erwartet werden, dass nur gültige Anfragen gesendet werden.

**Vollständig und korrekt**

Über die API können die meisten Funktionen von dem Game-Backend angesteuert werden. Die Ausnahme stellt hierbei die Teilnahme an einem konkreten Spiel dar. Technisch ist es möglich, dass ein registrierter Spieler einem bestimmten Spiel beitritt, dies wird derzeit jedoch nicht durch die API unterstütz. Der Grund hierfür ist, dass kein Client existiert oder in Planung ist, welcher diese Option ermöglicht. Aufgrund der bereits beschriebenen Skalierbarkeit ist es jedoch problemlos möglich, diese Funktion zu erweitern.

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)
3. Im Kapitel *8 Unit Tests* wird auf die nachträglich erstellten Tests eingegangen, sodass diese beiden Kapitel in einem geringen zusammenhängen, beziehungsweiße sich überschneiden. [↑](#footnote-ref-3)
4. Die vier verschiedenen Pages verwenden zwar auch den IOHandler, greifen hierauf jedoch nur über die abstrakte Oberklasse zu. [↑](#footnote-ref-4)
5. Im Sinne des ISP wäre es hier sinnvoll mehrere Interfaces zu erstellen (zum Beispiel

   IGameParticipant für Die() & Join() und IPlayerActionController für die restlichen Methoden. Dies wurde bewusst nicht gemacht, da hierfür zu viel Quellcode umgebaut werden müsste, was weitere Fallstricke mit sich bringt. Wenn an dem Projekt weitergearbeitet wird, ist es jedoch empfehlenswert, diese Änderung vor weiteren umfangreichen Änderungen vorzunehmen. [↑](#footnote-ref-5)