Universität Bremen
FB 3 – Informatik
Prof. Dr. Rainer Koschke
Tutor: Karsten Hölscher

Software–Projekt 2 SoSe 2019

VAK 03-BA-901.02

Architekturbeschreibung

Die Grafen

Mark Büßenschütt mark3@tzi.de
Paul Duhr pduhr@tzi.de
Nick Michalek michalek@tzi.de
Jan Neumann jan_neu@tzi.de
Jan Romann s_edhnm5@tzi.de
Lucien Granereau luc_gra@tzi.de

Abgabe: 16. Juni. 2019 — Version 1.0

Inhaltsverzeichnis

Version und Änderungsgeschichte

Version	Datum	Änderungen
0.5	2x.05.2019	Übernahme der Architekturbeschreibung aus dem letzten Semester als initiale Fassung

1 Einführung

Im Rahmen der Veranstaltung »Softwareprojekt II« wurde uns die Aufgabe gestellt, im Auftrag eines Kunden eine Software zu entwickeln und den Entwicklungsprozess entsprechend zu dokumentieren. Die vorliegende Architekturbeschreibung bildet dabei die Struktur sowie die geplante technische Umsetzung ab und legt die Gründe dar, auf deren Grundlage wir die Designentscheidungen für unsere Software getroffen haben.

Bei der zu entwickelenden Software handelt es sich um ein Computerspiel, das dem Genre »Tower Defense« beziehungsweise »Tower Wars« zuzuordnen ist und sowohl im Einzel- wie auch im Mehrspieler*innenmodus spielbar sein soll. Das Spielprinzip eines Tower-Defense-Spiels sieht vor, dass eine Reihe von Einheiten versuchen, auf einer Karte von einem Startpunkt zu einem Zielpunkt zu gelangen, um dem*der Spieler*in Schaden zuzufügen. Der*die Spieler*in kann sich durch das Errichten und Ausbauen von Türmen, die auf die gegnerischen Einheiten feuern und diese durch das Anrichten von einer ausreichenden Menge von Schaden ausschalten können, hiergegen verteidigen. Im Mehrspieler*innen-Modus treten zwei Personen gegeneinander an und können die Wellen an gegnerischen Einheiten, die der*die Gegner*in auszuschalten hat, mit eigenen Einheiten verstärken.

Eine besondere Herausforderung für

Die vorliegende Architekturbeschreibung legt dar, welche konkreten Anforderungen umgesetzt werden sollen, welche Einflussfaktoren und Probleme bei der Erstellung der Softwarearchitektur auftreten können und wie sie zu bewältigen sind und erläutert letztlich unter Zuhilfenahme von einer Reihe von UML-Diagrammen die Archtitektur aus den Blickwinkeln nach Hofmeister et al. [1]. Den Abschluss bildet ein Ausblick auf die Evolution des Produktes und seine Erweiterbarkeit gegeben.

1.1 Zweck

Der Zweck dieser Architekturbeschreibung ist die Erläuterung von Aufbau und Strukturierung unserer Software.

Dieses Dokument richtet sich vor allem an Leser*innen, die mit unserer Software arbeiten, sie testen oder später einmal weiterentwickeln möchten, gibt aber auch uns als Entwicklungsteam eine Vorgabe für die Implementierung von Schnittstellen und das Zusammenwirken der verschiedenen Komponenten. Auch für die Überprüfung der richtigen Umsetzung der Software, etwa mittels Black-Box-Tests, ist diese Archtitekurbeschreibung im Entwicklungsprozess heranzuziehen.

Der Kunde selbst kann mittels der vorliegenden Architekturbeschreibung den Entwicklungsprozess und vor allem die Design-Entscheidungen nachvollziehen, wenngleich es vermutlich nicht möglich sein sollte, die technischen Details der Schnittstellenbeschreibungen sowie die UML-Diagramme vollständig zu dechiffrieren.

1.2 Status

Dieses Dokument beschreibt den ersten Entwurf. Es wurde noch nicht durch ein Architektur-Review freigegeben.

1.3 Definitionen, Akronyme und Abkürzungen

1.3.1 Definitionen

(Relationale) Datenbank System zur Verwaltung von Daten, die als Einträge in Tabellen modelliert werden.

Anwendungsfall Spezifiziert eine Menge von Aktionen, die ein System ausführen muss, damit ein Resultat stattfindet, welches für mindestens einen Akteur von Bedeutung ist..

Apache Maven Ein Programm zur automatischen Durchführung des Erstellungsprozesses von Java-Programmen und der Verwaltung von deren Abhängigkeiten..

Einflussfaktor Ein Faktor, der die Arbeit an der Architektur auf beliebige Art und Weise beeinflusst..

Entwurfsmuster Musterlösung für ein Problem, welches bei Implementierungen immer wieder auftaucht.

Framework Programmiergerüst, welches den Rahmen der Anwendung bildet. Es umfasst Bibliotheken und Komponenten.

Gegner Feindliche Einheit, die auf einem vorbestimmten Pfad versucht, ein Zielfeld zu erreichen. Besitzt je nach Gegner-Typ eine bestimmte Menge Lebenspunkte, einen Rüstungstyp,.

Gradle Ein Programm zur automatischen Durchführung des Erstellungsprozesses von Java-Programmen und der Verwaltung von deren Abhängigkeiten..

JAR-Datei Von einer JVM ausführbare Datei, die aus Java-Code übersetzt wurde.

Java Eine objektorientierte Programmiersprache..

Komponentendiagramm Strukturdiagramm der UML, stellt Komponenten und deren Schnittstellen dar..

Paketdiagramm Strukturdiagramm der UML, stellt die Verbindung zwischen Paketen, Paketimports bzw. Verschmelzungen und deren Abhängigkeiten dar..

Problemkarte Beschreibt ein Problem im Zusammenhang von zuvor zusammengestellten Einflussfaktoren und führen Lösungen bzw. entsprechende Strategien zur Lösung des Problems an.

Sequenzdiagramm Strukturdiagramm der UML, stellt den Austausch von Nachrichten zwischen Objekten mittels einer Lebenslinie dar..

Softwarearchitektur Struktur und der Beziehung der einzelnen Bestandteile einer Software zueinander.

Thread Teil eines Prozesses, der parallel zu anderen Threads ausgeführt werden kann.

Turm Objekt auf dem Spielfeld, das Gegner angreift und ihnen Schaden zufügt, bis sie zerstört sind. Kann von der Spielerin auf dem Spielfeld gegen das Zahlen von Geld platziert und aufgerüstet werden. Hat je nach Turm-Art einen speziellen Angriffstyp, Spezialangriffe und/oder einen Auraeffekt, der einen passiven Effekt auf alle Türme/Gegner in der Umgebung hat..

1.3.2 Akronyme und Abkürzungen

DAO *Database Access Object*, Entwurfsmuster, das die Definition einer Schnittstelle vorsieht, über die Java-Objekte in einer relationalen Datenbank gespeichert, aktualisiert und abgerufen werden können..

GUI *Graphical User Interface*, grafische Schnittstelle, über die ein Mensch mit einer Software interagieren kann.

IDE *Integrated Delevopment Environment*, eine integrierte Entwicklungsumgebung, die Nutzer*innen bei allen Belangen der Softwareentwicklung unterstützt, sei es beim Schreiben von Programmcode, beim Testen oder der Versionskontrolle.

JPA *Java Persistence API*, Framework bzw. Schnittstelle zur Modellierung von Java-Objekten als Datenbankeinträge.

JVM *Java Virtual Machine*, virtuelle Maschine, die es ermöglicht, Java-Code plattformübergreifend auszuführen.

SQL *Structured Query Language*, Sprache zum Definieren, Modifizieren und Abfragen von Datenbankstrukturen in einer relationalen Datenbank..

SWP *Softwareprojekt*, eine in zwei Teile geteilte Veranstaltung im Bachelorstudiengang Informatik an der Universität Bremen, in der Studierende eine eigene Software entwickeln müssen.

UML *Unified Modeling Language*, grafische Modellierungssprache zur Spezifikation, Visualisierung, Konstruktion und Dokumentation von Modellen für Softwaresysteme..

1.4 Referenzen

[1] C. Hofmeister, R. Nord und D. Soni, *Applied Software Architecture*. Boston, MA, USA: Addison-Wesley, 2000.

[2] I. A. W. Group, »IEEE Std 1471-2000, Recommended practice for architectural description of software-intensive systems«, IEEE, Techn. Ber., 2000, S. i–23.

1.5 Übersicht über das Dokument

Dieses Dokument orientiert sich am IEEE-Standard 1471 [2] und folgt den Architekturblickwinkeln von Hoffmeister et al. [1].

Nach der in diesem Abschnitt erfolgten Einführung beschreiben wir in Abschnitt 2 auf der nächsten Seite zunächst die Anwendungsfälle, die mit Hilfe unserer Software umgesetzt werden sollen. Diese bilden die Grundlage für die Globale Analyse in Abschnitt 3 auf Seite 7, die zunächst in Unterabschnitt 3.1 auf Seite 7 die für unsere Architektur relevanten Einflussfaktoren identifiziert und daraus in Unterabschnitt 3.2 auf Seite 18 eventuell auftretende Probleme sowie entsprechende Lösungsstrategien ableitet. Anschließend folgen die Blickwinkel nach Hofmeister et al. [1]: Zunächst die konzeptionelle Sicht (ab Seite 22), dann die Modul- (Seite 24) und die Ausführungssicht (Seite 27). In Abschnitt 8 auf Seite 28 wird schließlich mittels UML-Sequenzdiagrammen der Zusammenhang zwischen den Anwendungsfällen und der Architektur aufgezeigt, bevor in Abschnitt 9 auf Seite 31 beschrieben wird, wie die Architektur angesichts veränderter Anforderungen weiterentwickelt werden müsste.

2 Anwendungsfälle

3 Globale Analyse

In diesem Abschnitt werden Einflussfaktoren aufgezählt und bewertet (s. Unterabschnitt 3.1) sowie Strategien zum Umgang mit interferierenden Einflussfaktoren entwickelt (s. Unterabschnitt 3.2 ab Seite 18).

3.1 Einflussfaktoren

In der sich über die nächsten Seiten erstreckenden Tabelle 1 werden die von uns identifizierten Einflussfaktoren aufgeführt. In dieser haben wir sowohl eine Bewertung der Flexibilität und Veränderlichkeit als auch der möglichen, architekturrelevanten Auswirkungen vorgenommen. Einflussfaktoren, von denen wir vermutet haben, dass sie nicht archtitekturrelevant sind, haben wir dabei nicht in die Tabelle aufgenommen.[1]

Tabelle 1: Übersicht der Einflussfaktoren

Einflussfaktor	Flexibilität und Veränderlichkeit	Auswirkung
Organisation		
O1: Entwicklungsplan		
O1.1: Time-to-Market		
Die fertige Software muss am 11.08.2019 ausgeliefert werden.	Keine Veränderlichkeit, keine Flexibilität	Die Implementierung aller Funktionen kann unter Umständen nicht umgesetzt werden. Daher sollte die Architektur so aufgebaut werden, dass leicht eine Basisanwendung erstellt werden kann, die um weitere Komponenten erweitert werden kann.
O1.2: Architektur-Abgabe		
Die Architekturbeschreibung muss am 16.06.2019 abgegeben werden.	Keine Veränderlichkeit, keine Flexibilität.	Da nur eine kurze Bearbeitungszeit gegeben ist, sollte für eine schnelle schnelle und fehlerlose Umsetzung auf möglichst einfache Entwurfsmuster bzw. Frameworks sowie eine Modularisierung (um Komponenten später austauschen zu können) zurückgegriffen werden.

[1] S. das Handout zum entsprechenden Teil der »Softwareprojekt I«-Vorlesung für einen umfassenden Überblick über mögliche Einflussfaktoren.

Tabelle 1: Übersicht der Einflussfaktoren (fortgesetzt)

Einflussfaktor	Flexibilität und Veränderlichkeit	Auswirkung
O2: Entwicklungsbudget		
O2.1: Anzahl Entwickler		
An dem Projekt beteiligen sich sechs Entwickler.	Die Zahl kann sich ändern, falls Gruppenmitglieder aussteigen, jedoch können keine neuen hinzukommen.	Dadurch, dass mehrere Entwickler am Projekt beteiligt sind, kann arbeitsteilig vorgegangen werden. Daher sollte die Architektur möglichst modularisiert aufgebaut sein und die Schnittstellen so beschrieben werden, dass die einzelnen Komponenenten unabhängig voneinander implementiert werden können.
O2.2: Finanzielle Mittel		
Es steht kein Entwicklungsbudget zur Verfügung.	Es wird keine Änderungen beim Budget geben.	Es können nur kostenlos nutzbare oder von den Gruppenmitgliedern privat erworbene Mittel (wie Assets) benutzt werden. Dies kann dazu führen, dass nicht die optimale Lösung umgesetzt werden kann bzw. in der Architektur Kompromisse gemacht werden müssen.
O3: Prozesse und Werkzeuge		
O3.1: Systembau (Build)		
Die Software muss mit Gradle oder Maven gebaut werden, worüber auch das sogenannte Dependency Management[2] erfolgt.	Bei diesem Einflussfaktor gibt es keine Veränderlichkeit, in der Wahl eines der beiden Build-Systeme sind wir jedoch flexibel.	Die Nutzung eines Build-Systems erleichtert die Einbindung von Frameworks und anderer Abhängigkeiten und ermöglicht das Bauen von verschiedenen Modulen (z. B. Client und Server oder Desktop und Mobil) zur selben Zeit.

[2]Glossar

Tabelle 1: Übersicht der Einflussfaktoren (fortgesetzt)

Einflussfaktor	Flexibilität und Veränderlichkeit	Auswirkung
O3.2: Test		
Das fertige Programm muss auf verschiedenste Art und Weise getestet werden (z. B. Black-Box-Tests, Komponententests). Die Testergebnisse müssen in einem Testprotokoll festgehalten und abgegeben werden.	Testen ist essentiell, um eine ordnungsgemäße Funktionsweise unserer Software zu garantieren. Zudem ist die Abgabe eines Testprotokolls vorgesehen. In den Anforderungen ist daher keine Veränderlichkeit gegeben, zudem sind die vorzunehmenden Testarten ebenfalls relativ genau vorgegeben, weshalb nur geringe Flexibilität besteht.	Bei der Erstellung der Architektur sollte berücksichtigt werden, dass
O3.3: Release		
Kein Installer, Auslieferung als .jar-Datei	Keine Veränderlichkeit, keine Flexibilität, da vorgegeben.	Dank der JVM muss das Programm nur wenig die verschiedenen Zielsysteme angepasst werden.
O3.4: Client		
Die Entwicklung des Gameclients muss mit einer der zwei Vorgegebenen Bibliotheken (?) erfolgen	Geringe Flexibilität, Änderungen nur in frühen Entwicklungsstadium problemlos möglich	Es muss vor Beginn der Implementierung fest stehen welche Bibliothek verwendet werden soll um Probleme, die durch einen Wechsel auftretten könne, zu vermeiden

Tabelle 1: Übersicht der Einflussfaktoren (fortgesetzt)

Einflussfaktor	Flexibilität und Veränderlichkeit	Auswirkung
Technik		
T1: Hardware		
T1.1Anzahl an Threads		
Eine der Mindestanforderungen an das ausführende System ist, dass min. 3 Threads erzeugt werden können, nämlich der sowieso vorhandene Main-Thread, ein GUI-Thread und für weitere Funktionalitäten wie das Logging (?).	Die Anwendung würde auch bei der Erzeugung von weniger Threads funktionieren, jedoch würde dann die Nebenläufigkeit nur simuliert werden.	Sollten nicht ausreichend viele Threads erzeugt werden können, wirkt sich dies negativ auf die Performanz aus.
T1.2: Netzwerk		
Es soll möglich sein, über eine Netzwerkverbindung gegen eine andere Spielerin zu spielen. Hierbei kann es entweder eine Instanz geben, die nur als Server auftritt, mit dem sich zwei Clients verbinden, oder die Server- und Client-Funktionalität kann in einem Modul kombiniert werden.	Diese Vorgabe ist unveränderlich, bei der Ausgestaltung haben wir jedoch hohe Flexibilität.	
T1.3: Hauptspeicher		
Es gibt keine genauen Hardware-Anforderungen. Die Software wird jedoch auf den Rechnern der Ebene 0 getestet, bei denen es sich nicht um High-End-Geräte handelt.	Da es keine genauen Anforderungen an die Hardware gibt, haben wir eine relativ	Die Anwendung sollte einen möglichst kleinen Memory-Footprint haben.

Tabelle 1: Übersicht der Einflussfaktoren (fortgesetzt)

Einflussfaktor	Flexibilität und Veränderlichkeit	Auswirkung
T1.4: Plattenspeicher		
Unsere Software muss ggf. auf älteren Schulrechnern mit begrenzter Speicherkapazität laufen. Zudem können Logfiles bzw. die entsprechenden Einträge in der Datenbank sehr groß werden.	Durch Optimierung lässt sich nach der Implementierung zusätzlich Speicher einsparen.	Zu Beginn sollte eine Herangehenweise gewählt werden, die das Speicheraufkommen von Logs so stark wie möglich reduziert.
T2: Software		
T2.1: Implementierungssprache		
Die Mindestanforderungen sehen vor, dass entweder Java 8 oder Java 11 für die Implementierung genutzt werden soll.	Dass *eine* der beiden Sprachversionen genutzt werden soll, ist unveränderlich. Bei der Auswahl, welche der beiden Versionen verwendet werden soll, sind wir hingegen flexibel und können Vor- und Nachteile abwägen.	Java 8 hat nach wie vor einen höheren Verbreitungsgrad als Java 11, ist allerdings bereits im Stadium »End of Life« (EOF) angekommen. Beide der fest vorgegebenen Spieleframeworks (siehe T2.3) sind für beide Java-Versionen verfügbar, wobei es gewisse Unterschiede zwischen den beiden Versionen bei FXGL gibt und die Java-11-Version dieses Framework (noch) nicht alle Features der Java-8-Version unterstützt. Es kann daher Sinn ergeben, zunächst auf Java 8 (ggf. unter Nutzung von Verbindung mit LibGDX) zurückzugreifen, um eine möglichst große Verfügbarkeit bei größtmöglicher Möglichkeit zur Weiterentwicklung bereitzustellen.

Tabelle 1: Übersicht der Einflussfaktoren (fortgesetzt)

Einflussfaktor	Flexibilität und Veränderlichkeit	Auswirkung
T2.2: Entwurfsmuster		
Es sind keine speziellen Entwurfsmuster vorgegeben.	Wir sind im Einsatz von Entwurfsmustern nicht beschränkt, weshalb hohe Flexibilität bei geringer Veränderlichkeit gegeben ist.	Entwurfsmuster können (und sollten) dort eingesetzt werden, wo sie die Struktur der Software (z. B. durch Kapselung von Komponenten wie der Persistenz über das DAO-Pattern[3]) verbessern. Stehen mehrere mögliche Entwurfsmuster zur Auswahl, können wir uns für das entscheiden, von der unserer Meinung nach die Software am meisten profitiert. Auch auf die Umsetzung von Entwurfsmustern in Frameworks können wir in diesem Zusammenhang zurückgreifen, sofern sie einen Mehrwert für die Software mit sich bringen.
T2.3Rahmenwerke (Frameworks)		
Die Nutzung von LibGDX oder FXGL ist vorgegeben. Ansonsten kann auf beliebige Frameworks zurückgegriffen werden.	In der Nutzung von einem der beiden genannten Frameworks besteht keine Veränderlichkeit und nur die Flexibilität der Wahl zwischen den beiden. Ansonsten ist bei der Einbindung von Frameworks höchste Flexibilität gegeben.	Die Struktur der Software muss sich an dem gewählten Spiel-Framework orientieren, um deren Funktionalität bestmöglich nutzen zu können. Weitere Funktionalitäten wie die Persistenz oder die Netzwerkkommuniaktion können mithilfe von Frameworks implementiert werden – hier bietet sich eine entsprechende Kapselung an, um bei Nutzung eines Frameworks den Einfluss auf die restlichen Komponenten möglichst gering zu halten.

[3] Glossar-Begriff

Tabelle 1: Übersicht der Einflussfaktoren (fortgesetzt)

Einflussfaktor	Flexibilität und Veränderlichkeit	Auswirkung
T3: Standards		
T3.1: Datenbanken		
Für die Persistierung von Daten muss gemäß Anforderungen eine leichtgewichtige (eingebettete) relationale Datenbank verwendet werden und diese muss SQL-ähnliche Anfragen enthalten.	Da diese Anforderung fest vorgebenen ist, besteht keine Veränderlichkeit. Bei der Umsetzung gibt es jedoch eine hohe Flexibilität. Zum Beispiel kann auf ein Persistenzframework, das die Java Persistence API (JPA) implementiert, zurückgegriffen werden, das den Zugriff auf die Datenbank vom genutzten Datenbanksystem (z. B. Apache Derby oder SQLite) abstrahiert.	Es muss ein Persistenz-Paket innerhalb der Software vorhanden sein, das die Abbildung von Plain Old Java Objects (POJOs) auf Datenbank-Entitäten vornimmt. Durch Kapselung und Verwendung von Entwurfsmustern wie dem DAO-Pattern sollte dabei eine Schnittstelle geschaffen werden, die den Zugriff auf die Datenbank möglichst stark abstrahiert.
T3.2: TCP/IP		
Für die Netzwerkkommunikation muss/sollte entweder mittels TCP- oder UDP-Paketen erfolgen.	Für die genaue Realisierung der Netzwerkkommunikation gibt es keine detaillierten Vorgaben, weshalb wir hier eine hohe Flexibiltät haben. Eine Änderung bzw. Konkretisierung der Vorgaben ist nicht zu erwarten.	Die Architektur sollte spezifizieren, welche Protokolle für welche Arten von Kommunikation angewendet werden. Die Übermittlung von Spielzügen während des Spieles kann bzw. sollte (um Verzögerungen und Unterbrechungen zu vermeiden) mittels UDP erfolgen, für andere Formen der Kommunikation, die die Integrität und zuverlässige Übermittlung der Daten erfordern, kann TCP genutzt werden. Die Netzwerkkomponente sollte gekapselt werden, um von der tatsächlichen Implementierung (etwa durch die Nutzung von Frameworks wie Kryonet) zu abstrahieren.

Tabelle 1: Übersicht der Einflussfaktoren (fortgesetzt)

Einflussfaktor	Flexibilität und Veränderlichkeit	Auswirkung
Produktfaktoren		
P1: Produktfunktionen		
P1.1: Desktop-Anwendung		
Die Software soll als Desktop-Anwendung bereitgestellt werden.	Dies ist eine vom Kunden gestellte Anforderung, die nicht verändert wird. Eine zusätzliche Bereitstellung als Mobil- oder Browser-Anwendung kann jedoch als »begeisternder Faktor« im Sinne des Kano-Modells[4] für eine erhöhte Kundinnenzufriedenheit sorgen, weshalb wir bei der (Über-)Erfüllung dieser Anforderung eine gewisse Flexibilität haben.	Es ist ein Desktop-Modul vorzusehen, das über eine JVM ausgeführt werden kann. Bei Verzicht auf ein separates Server-Modul muss dieses auch die Server-Funktionalität beinhalten. Über die Auslagerung der nicht-plattformspezifischen Eigenschaften in ein eigenes Modul »Core« können später relativ leicht Portierungen auf Android, iOS sowie HTML5 kompatible Browser vorgenommen werden.
P1.2: Verschiedene Schwierigkeitsgrade		
Das Spiel verschiedene Schwierigkeitsgrade aufweisen.	Diese Mindestanforderung sollte sich nicht verändern. Die Anzahl der Schwierigkeitsgrade ist allerdings nicht genau bestimmt – es sollten lediglich mindestens ein »normaler« und ein für Testzwecke nutzbarer leichter Schwierigkeitsgrad vorhanden sein.	Beim Starten des Spieles sollte es möglich sein, mindestens einen der genannten Schwierigkeitsgrade auswählen zu können. Die Berücksichtigung dieses Faktors sollte keinen zu großen Einfluss auf die Architektur haben und kann zum Beispiel durch Definition einer entsprechenden Klasse »Difficulty« oder die direkte Einbettung in die Spiellogik erfolgen.

[4] Zitat

Tabelle 1: Übersicht der Einflussfaktoren (fortgesetzt)

Einflussfaktor	Flexibilität und Veränderlichkeit	Auswirkung
P1.3: Türme/Turmeigenschaften		
Gemäß der Mindestanforderungen des Kunden sollen verschiedene Türme auf dem Spielfeld platzierbar sein, die sich gegen Ressourcen aufrüsten lassen und ihre Eigenschaften (z. B. Angriffsgeschwindigkeit oder Preis) sowie Spezialeigenschaften (wie Flächenschaden oder einen Aura-Effekt) verbessern.	Diese Mindestanforderung sollte sich nicht verändern. Bei der genauen Umsetzung ist jedoch ein hohes Maß an Flexibilität gegeben.	Es muss mindestens eine Turmklasse geben, die entsprechende Attribute für die Eigenschaften und Spezialeigenschaften aufweist, und die z. B. durch Vererbung oder unter Nutzung des Factory-Entwurfsmusters ausdifferenziert werden kann.
P1.4: Verschiedene Gegner-Typen		
Gemäß der Mindestanforderungen soll es möglich sein verschiedene Gegner-Typen in einer Angriffswelle zu schicken. Die Gegner-Typen unterscheiden sich anhand ihrer Attribute (Rüstung, Geschwindigkeit und Lebensenergie). Bei erreichen des Zielpunkts durch einen Gegner soll es negative Auswirkungen für den Spieler haben und sollte ein Gegner zerstört werden soll der Spieler dafür belohnt werden.	Diese Mindestanforderung sollte sich nicht verändern. Bei der genauen Umsetzung ist jedoch ein hohes Maß an Flexibilität gegeben.	Es muss mindestens eine Gegnerklasse geben, die entsprechende Attribute für die Eigenschaften und Spezialeigenschaften aufweist, und die z. B. durch Vererbung oder unter Nutzung des Factory-Entwurfsmusters ausdifferenziert werden kann.

Tabelle 1: Übersicht der Einflussfaktoren (fortgesetzt)

Einflussfaktor	Flexibilität und Veränderlichkeit	Auswirkung
P1.5: Eigenschaften der Spielerin		
Gemäß der Mindestanforderungen soll ein Spieler eine festgesetzte Anzahl an Leben haben. Zudem soll der Spieler über Ressourcen,wovon mindestens eine Geld sein soll. Mit diesen Ressourcen kann der Spieler Türme aufrüsten oder erwerben. Die Türme sollen vom Spieler aufgestellt entfernt und aufgerüstet werden können. Außerdem soll der Spieler im Duell im Multiplayer auch Angriffswellen auslösen können	Diese Mindestanforderung sollte sich nicht verändern. Bei der genauen Umsetzung ist jedoch ein hohes Maß an Flexibilität gegeben.	
P1.6: Spiel-Logik		
Gemäß der Mindestanforderungen soll das Spiel verschiedene Schwierigkeitsstufen haben, wobei die leichteste problemlos durchspielbar ist.	Diese Mindestanforderung sollte sich nicht verändern. Bei der genauen Umsetzung ist jedoch ein hohes Maß an Flexibilität gegeben.	
P1.7: Multiplayer-Modus		
Gemäß der Mindestanforderungen soll es ein Multiplayer-Modus enthalten in dem zwei Spieler parallel im selben Spiel agieren. Zudem ist im Multiplayer-Modus das auslösen von Angriffswellen beim Gegner möglich.	Diese Mindestanforderung sollte sich nicht verändern. Bei der genauen Umsetzung ist jedoch ein hohes Maß an Flexibilität gegeben.	
P2: Benutzerschnittstelle		
P2.1: Interaktionsmodell		

Tabelle 1: Übersicht der Einflussfaktoren (fortgesetzt)

Einflussfaktor	Flexibilität und Veränderlichkeit	Auswirkung
P2.2: Funktionen der Benutzerschnittstelle		
P3: Produktkosten		
P3.1: Hardwarebudget		
Es stehen keine finanziellen Mittel für dieses Projekt zur Verfügung.	Das Budget steht fest und wird nicht verändert.	Das Projekt muss mit kostenlosen oder privaten Mitteln umgesetzt werden.
P3.2: Softwarelizenzbudget (für verwendete Software)		
Es stehen keine finanziellen Mittel für dieses Projekt zur Verfügung.	Das Budget steht fest und wird nicht verändert.	Das Projekt muss mit kostenloser Open-Source-Software bzw. entsprechenden Bibliotheken realisiert werden.

3.2 Probleme und Strategien

In diesem Abschnitt erläutern wir die Probleme, die wir auf Basis der im vorherigen Unterabschnitt zusammengestellten Einflussfaktoren identifiziert haben, auf sogenannten Problemkarten und diskutieren mögliche Lösungsstrategien. Aus diesen werden wir für jedes Problem jeweils eine auswählen, was gleichbedeutend mit einer Designentscheidung für die Architektur unserer Software ist.

Gewählte Strategien

Problem 1: Plattformunabhängige Entwicklung
Einflussfaktoren
Strategien
Strategie 1: Bündelung der Kern-Funktionalität in »Core«-Modul (ausgewählt) Im LibGDX-Framework wird der Großteil eines Spieles in einem Modul »Core« programmiert, auf den weitere Module (wie in unserem Fall das »Desktop«-Modul) zurückgreifen. Diese plattformspezifischen Module weisen jeweils eine eigene Hauptklasse auf und können ggf. auf die jeweiligen Besonderheiten (wie zum Beispiel die Bildschirmgröße oder Unterstützung bestimmter Funktionalitäten wie der Netzwerkkommunikation) angepasst werden. Dieses Vorgehen reduziert Redundanz erleichtert es uns, später die Unterstützung weiterer Zielplattformen hinzuzufügen. **Strategie 2: Programmierung eines eigenen Moduls für jede Plattform (alternativ)** Alternativ könnte die Funktionalität der Software in jedes plattformspezifische Modul integriert werden. Dies schafft jedoch erhebliche Redundanzen, weshalb wir uns gegen diese Strategie entschieden haben.

Problem 2: Umsetzung der Netzwerkkommunikation
Einflussfaktoren P1.3: Protokollierung in Datenbank P1.5: Veränderlichkeit von grafischen Elementen P1.11: Verschiedene Modi
Strategien
Strategie 3: Nutzung des Frameworks kryonet in Verbindung mit TCP und UDP (ausgewählt)

Problem 3: Entscheidung zwischen 2D- und 3D-Grafik

LibGDX unterstützt sowohl die Erstellung von zweidimensionalen als auch von dreidimensionalen Spielen. Die Nutzung von 3D-Grafik ist potentiell ansprechender und kann Nutzerinnen eine intensivere Spielerfahrung bieten. Dreidimensionale Grafik ist allerdings auch anspruchsvoller in der Umsetzung, benötigt mehr zeitliche und unter Umständen auch finanzielle Ressourcen (für den Erwerb passender Assets) und birgt mehr Risiken für eine fehlerfreie Implementierung, was den Erfolg des Projekts gefährden kann. Angesichts der geringen Erfahrung der Projektmitglieder in der Umsetzung von Computerspielen ist es daher anzuraten, sich eher für zweidimensionale Grafik zu entscheiden, die leichter umsetzbar ist und auf das Spielgefühl einer »Tower Defense« keinen zu großen Einfluss hat.

Einflussfaktoren

O1.1: Time-to-Market
O1.1: Architektur-Abgabe
O2.1: Anzahl Entwickler
O2.2: Finanzielle Mittel
T1.3: Hauptspeicher
T1.5: Grafikkarte
T2.3: Rahmenwerke (Frameworks)
P1.1: Desktop-Anwendung
P3.1: Hardwarebudget
P3.2: Softwarelizenzbudget (für verwendete Software)

Strategien

Strategie 4: Nutzung von 2D-Grafik (ausgewählt)
Wir nutzen für die Umsetzung zweidimensionale Grafik. Hierzu ziehen wir die entsprechenden Komponenten aus LibGDX heran, die zweidimensionale Grafikelemente rendern können, sowie passende Assets. Insbesondere greifen wir bei der Umsetzung der Spielwelt auf eine sogenannte »Tiled Map« zurück, mit der sich aus einer geringen Anzahl von Feldern komplexe Spielwelten zusammensetzen lassen, und verwalten alle Elemente über einen Szenengraph, der eine Hierarchie der Teile einer »Szene«, d. h. der Elemente eines zu rendernden Bildes, definiert und somit zum Beispiel das Benutzerinterface logisch vom Spielgeschehen trennt.

Strategie 5: Nutzung von 3D-Grafik (alterantiv)
Wir nutzen für die Umsetzung dreidimensionale Grafik. Hierfür müssen wir auf passende, 3D-modellierte Assets zurückgreifen sowie eine dreidimensionale, freibewegliche Kamera implementieren. Auch hier böte sich die Nutzung eines Szenegraphs an,

Problem 4: Abhängigkeit von Frameworks

Für die Umsetzung der Software werden wir uns auf ein Framework zu Graphvisualisierung verlassen müssen, da es zu aufwändig ist, diese Funktionalität vollständig selbst zu implementieren. Auch für die Realisierung der Interaktionen mit der Datenbank werden wir auf ein Framework (voraussichtlich JPA) zurückgreifen (müssen), um die Protokollierung der Interaktionen gewährleisten zu können.
Bei der Benutzung von Frameworks muss dabei darauf geachtet werden, die Abhängigkeit so groß wie nötig und so gering wie möglich zu halten, um die Anpassbarkeit
Im Voraus lässt sich jedoch noch nicht vollständig abschätzen, ob ein einmal gewähltes Framework unsere Anforderungen wirklich vollständig erfüllen kann oder ob während der Implementierungsphase ein Punkt erreicht wird, an dem das gewählte Framework nicht mehr für unsere Zwecke zu nutzen ist oder z. B. gegen ein anderes ausgetauscht werden muss.
Zudem ist eine zu große Abhängigkeit von Frameworks nicht wünschenswert, da diese die Möglichkeiten und das Weiterentwicklungspotential unser Software u. U. einschränken können und zusätzlichen Overhead, sowohl in Bezug auf die Performance als auch auf die benötigten Ressourcen, verursachen können.

Einflussfaktoren

O1.1: Time-To-Market
T1.2: Hauptspeicher
T1.2: Plattenspeicher
T2.2: Entwurfsmuster
T2.3: Rahmenwerke (Frameworks)
P1.3: Datenbanken
P1.3: Protokollierung in Datenbank

Strategien

Strategie 6: Kapselung von Funktionalitäten/Definition von Schnittstellen (ausgewählt)
Um zu vermeiden, dass sie einen zu großen Einfluss auf die Gesamtarchtitektur erhalten, werden Frameworks (sofern es möglich ist) möglichst stark gekapselt bzw. ihre Funktionalität durch die Definition von Schnittstellen abstrahiert. So können andere Programmteile ohne genaue Kenntnis des betreffenden Frameworks entwickelt werden (etwa für die Datenbankfunktionalität oder die Netzwerkkommunikation) und ein Framework zu einem späteren Zeitpunkt leichter ausgetauscht werden.

Strategie 7: Abhängigkeit von Frameworks so stark wie möglich reduzieren (alternativ)
Um eine zu starke Abhängigkeit von einem Framework zu vermeiden, werden wir bei der Modularisierung darauf achten, die Kopplung der auf einem Framework basierenden Module zum Rest der Software möglichst gering zu halten. Dies würde im schlimmsten Fall den Austausch des Frameworks ermöglichen und erhöht im Allgemeinen die Möglichkeit, die Software später weiterentwickeln zu können.

Strategie 8: Funktionalitäten selbst implementieren (alternativ)
An den Stellen, an denen wir merken, dass ein Framework nicht ausreichend ist, werden wir die notwendigen Funktionen selbst implementieren müssen. Hierfür ist es wichtig, ein Framework zu verwenden, das zwar einerseits selbst bereits viele Funktionen mitbringt, gleichzeitig jedoch auch die notwendige Anpassungsfähigkeit an unsere Bedürfnisse aufweist.

Problem 5: Vorgegebene Technologien
Wir müssen uns für eines von zwei Frameworks für die Umsetzung unseres Spieles entscheiden: Entweder für LibGDX oder FXGL.
Einflussfaktoren
Strategien
Strategie 9: Nutzung von LibGDX (ausgewählt) Wir nutzen LibGDX als wesentliches Framework für die Erstellung unseres Spieles. Dies hat zur Folge, dass wir unsere Struktur stark an die von LibGDX anpassen werden und einige unserer Klassen direkt von LibGDX-Klassen (wie den Klassen `Game` oder `Screen`) erben. **Strategie 10: Nutzung von FXGL (alternativ)** Wir greifen auf FXGL als Framework zurück.

Problem 6: Umsetzung der Datenbankfunktionalität
Unser Spiel muss die Funktionalität aufweisen, Daten in einer eingebetteten, leichtgewichtigen Datenbank speichern zu können. Hierfür muss mindestens zwei Anfragen an die Datenbank geben, die zudem SQL-ähnlich sein müssen. Zur Umsetzung der Datenbankfunktionalität bietet es sich an, auf ein Framework zurückzugreifen, das die Java Persistence API (JPA) implementiert, wie zum Beispiel Hibernate. Über eine Konfigurationsdatei kann sodann ein SQL-»Dialekt« sowie eine entsprechende (in unserem Fall eingebettete) Datenbank angegeben werden, die mit den passenden Einstellungen automatisch die passenden Tabellen initialisiert. Neben dem Zugriff über einen sogenannten Session-Manager auf die Datenbank, über den Java-Objekte ohne weiteres Zutun auf Datenbank-Entitäten abgebildet werden, bietet die JPA auch die Möglichkeit, sogenannte »Named Queries« zu definieren, mit denen sich die Mindestanforderung der Nutzung von SQL-ähnlichen Befehlen erfüllen lässt.
Einflussfaktoren
Strategien
Strategie 11: Nutzung von Hibernate und in Verbindung mit Derby DB (ausgewählt) **Strategie 12: ???**

4 Konzeptionelle Sicht

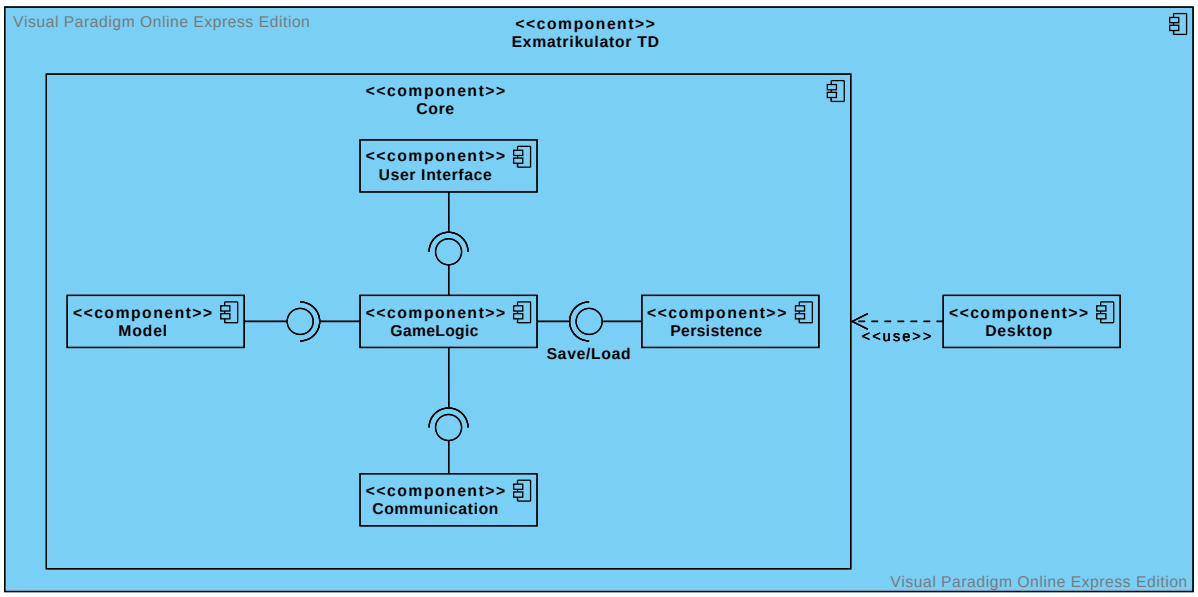

Abbildung 1: Konzeptionelle Sicht des Spiels »Exmatrikulator TD«

Wie Abbildung 1 zu entnehmen ist, ziehen wir zur Realisierung unseres Spiels »Exmatrikulator TD« wie in Strategie 1 beschrieben zwei Komponenten heran: Eine Komponente »Core«, die das eigentliche Spiel umsetzt, und eine Komponente »Desktop«, die das Implementieren von plattformspezifischen Funktionen ermöglicht, im Wesentlichen jedoch der Bereitstellung eines entsprechenden »Launchers« dient. Diese Struktur der Software ermöglicht es uns, das Spiel später auf zusätzliche Plattformen wie Android oder iOS zu portieren, indem wir entsprechende Komponenten nach dem Vorbild des Desktop-Moduls hinzufügen (siehe hierzu auch Abschnitt 9 zur Evolution der Software). Diese Struktur ergibt sich auch zu einem wesentliche Teil aus der Strategie XYZ »Nutzung von LibGDX«, da dies die Standard-Struktur eines Projektes ist, das mit diesem Framework erstellt wird.

Die »Core«-Komponente ist folglich das Herz unserer Software und ist in weitere Komponenten ausdifferenziert, die jeweils eine bestimmte Aufgabe innerhalb des Spieles übernehmen:

- Die Komponente **User Interface** stellt die grafische Benutzeroberfläche zur Verfügung, mit der die Benutzerin interagiert und verwaltet zudem die verschiedenen »Bildschirme« (Screens), die der Benutzerin angezeigt werden, wie zum Beispiel Menüs oder den eigentlichen Spielbeschirm, nimmt Eingaben und Spielzüge entgegen und reicht sie an das **Game Logic**-Modul weiter, dessen Zustand es zur Darstellung des Spieles rendert. Die Realisierung als eigene Komponente ergibt sich aus der Strategie XYZ »Kapselung der Benutzerschnittstelle«.

- Die Komponente **Game Logic** dient der Verarbeitung von Spielzügen und zur Berechnung des aktuellen Spielzustandes. Es greift auf die Komponente »Model« zu, das die Spielwelt sowie die verschiedenen Turm- und Einheiten-Typen beinhaltet. In einem Multiplayer-Spiel kann unsere Software sowohl als Client als auch als Server auftreten (Strategie XYZ). Im ersteren Fall sendet die Game-Logic-Komponente Spielzüge über die Komponente »Communication« an den Server, um diese zunächst von diesem überprüfen zu lassen. Tritt die Instanz von »Exmatrikulator TD« als Server auf, wird diese Überprüfung ebenfalls durch die Game-Logic-Komponente übernommen, da diese den Zustand des Spieles kennt und somit verifizieren kann, ob ein Spielzug »legal« ist. Unter Nutzung der Komponente »Persistence« kann der aktuelle Spielzustand abgespeichert und wieder geladen werden.
- Die Komponente **Communication** dient der Realisierung der Netzwerkkommunikation stellt sowohl Client- als auch Server-Funktionalität zur Verfügung (vgl. Strategie 3). Die Realisierung als eigene Komponente dient der Abstraktion der Netzwerkschnittstelle und ermöglicht es uns, diese Komponente weitgehend unabhängig von anderen Teilen des Programmes zu realisieren (Strategie XYZ). Dies erleichtert nicht nur ggf. einen späteren Austausch des verwendeten Frameworks, sondern verringert auch dessen Einfluss auf den Rest der Architektur (Strategie XYZ). Zudem kann durch eine Modularisierung der Software nicht nur bei dieser Komponente arbeitsteilig vorgegangen werden (Strategie Blah).
- Die Komponente **Persistence** schließlich dient der Realisierung der Datenhaltung und stellt eine Schnittstelle zu einer eingebetteten Datenbank zur Verfügung. Die Persistence-Komponente abstrahiert dabei vom verwendeten Datenbank-System und erleichtert die Interaktion mit dieser Komponente.

5 Modulsicht

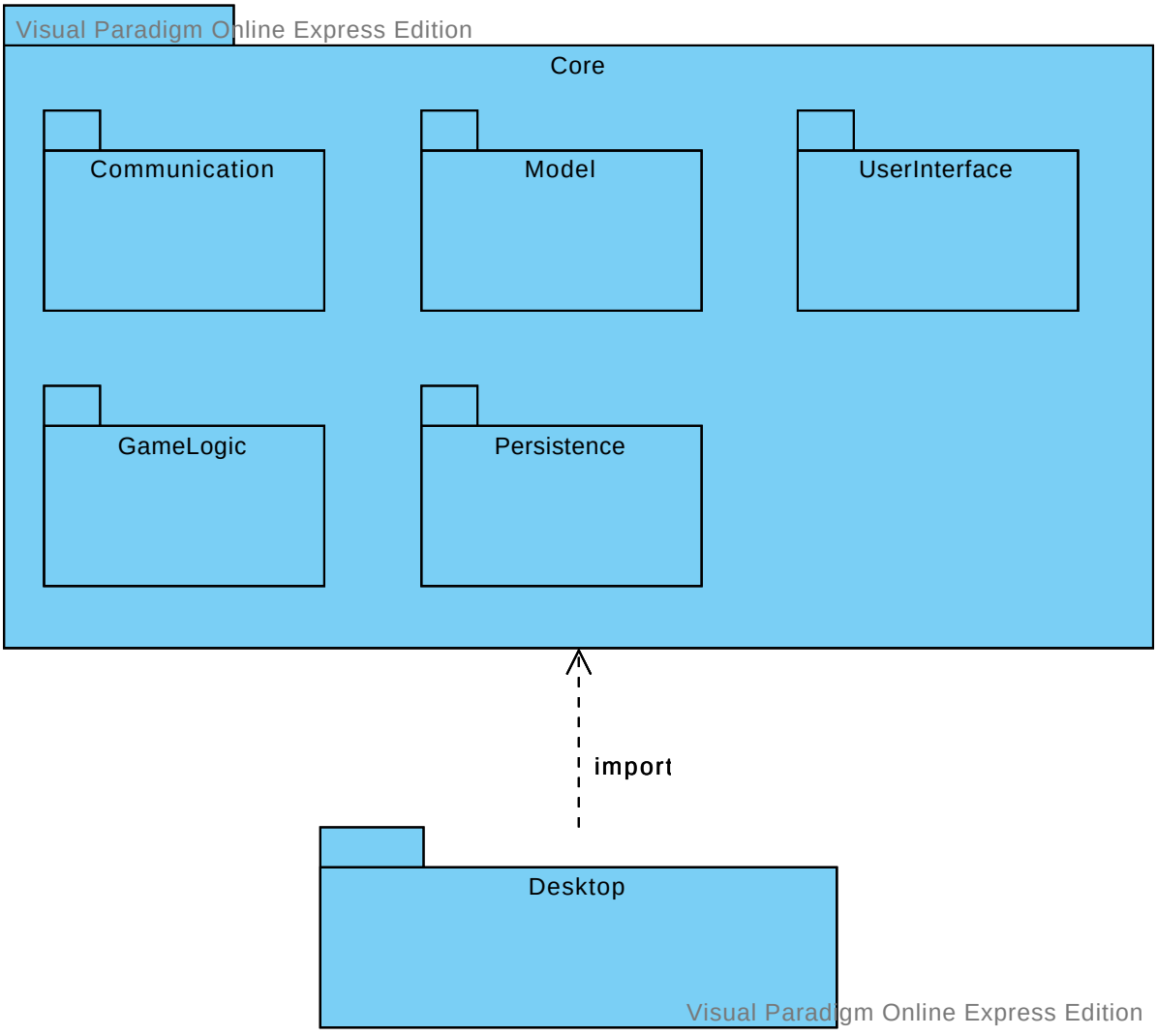

Abbildung 2: Modulsicht des Spiels »Exmatrikulator TD«

5.1 Konkretisierung Desktop

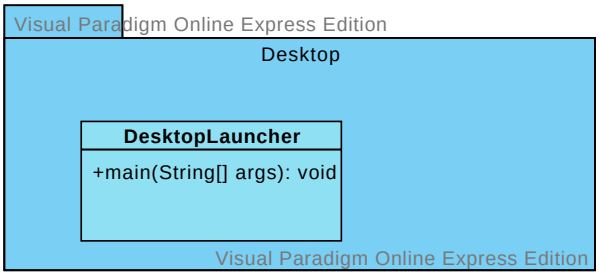

Abbildung 3: Kokretisierung des Desktop-Moduls

5.2 Konkretisierung Core

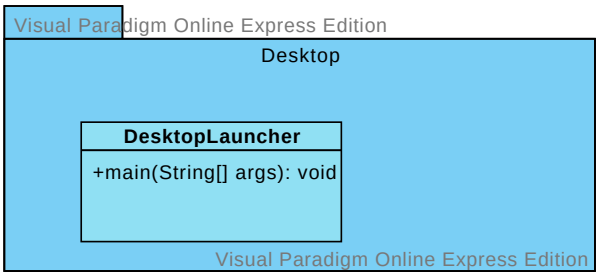

Abbildung 4: Kokretisierung des Core-Moduls

Das Core-Modul besteht aus den Paketen **Communication**, **Model**, **UserInterface**, **GameLogic** und **Persistence**, die im Folgenden näher erläutert werden.

6 Datensicht

6.1 Konkretisiertes Datenmodell

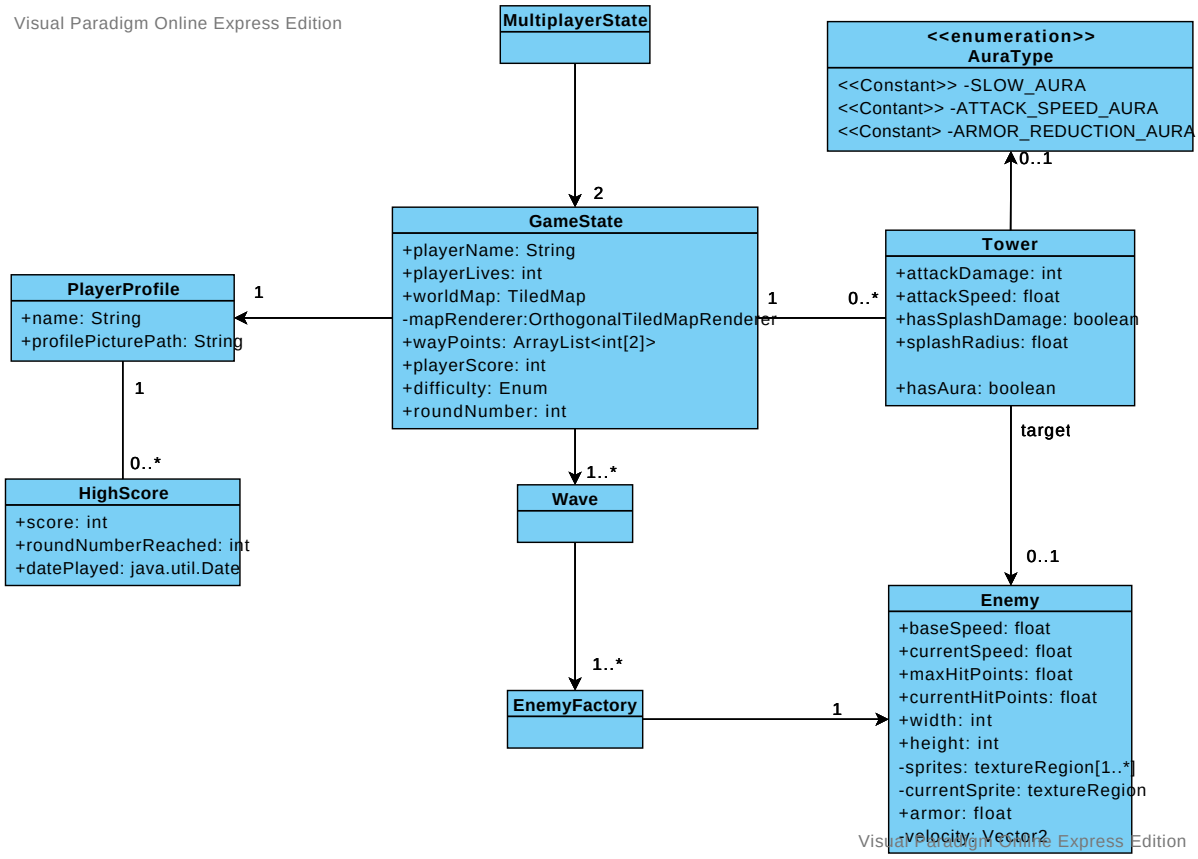

Abbildung 5: Darstellung unseres Datenmodells als UML-Klassendiagramm

7 Ausführungssicht

In Abbildung 6 ist unsere Software in der Ausführugssicht dargestellt. Da es sich um eine Desktop-Anwendung handelt und nicht um ein verteiltes System, ähnelt die Darstellung sehr stark der konzeptionellen Sicht (vgl. Abbildung 1). Wie sich Abbildung 6 entnehmen lässt, laufen auf dem Ausführungssystem zwei Prozesse parallel: Ein `main`-Prozess, der die Funktionen aus den in der Paketsicht umsetzt, und ein Datenbank-Prozess, über den die Speicherung des Protokolls verwirklicht wird. Zwischen diesen beiden Prozessen findet ein lokaler Datenaustausch statt, bei dem das `Persistance`-Paket die Schnittstelle zur Datenbank bereitstellt.

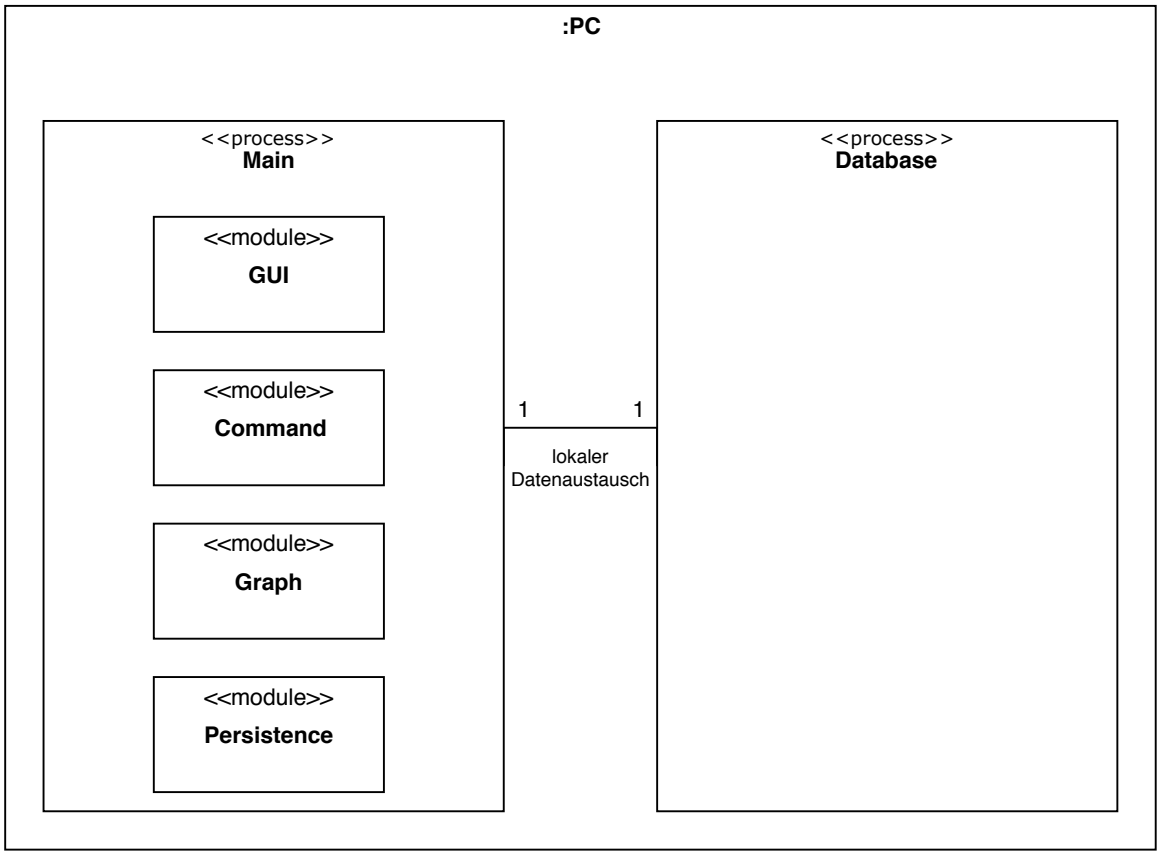

Abbildung 6: Ausführungssicht

8 Zusammenhänge zwischen Anwendungsfällen und Architektur

In diesem Abschnitt erläutern wir anhand von UML-Squenzdiagrammen, wie die einzelnen Elemente unserer Software bei deren Ausführung zusammenwirken. Hierzu betrachten wir die zwei wesentlichen Anwendungsfälle unserer Software: Das Starten und Spielen eines Single- sowie eines Multiplayer-Spiels, letzteres inklusive des Hostens einer Session als Server. Diese Anwendungsfälle sind in Abbildung 7 als UML-Usecase-Diagramm dargestellt.

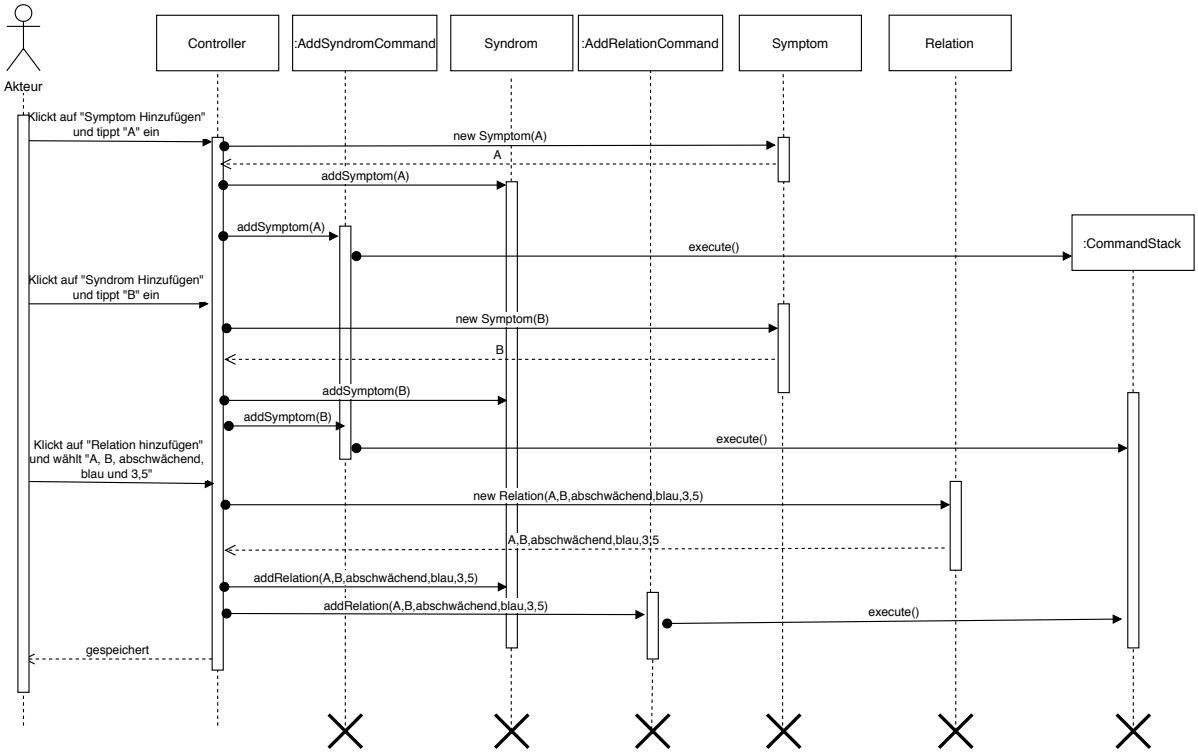

Abbildung 7: UML-Anwendungsfall-Diagramm (noch durch ein tatsächliches Diagramm zu ersetzen)

8.1 Hinzufügen von zwei oder mehr Knoten und Verbindung durch eine Kante

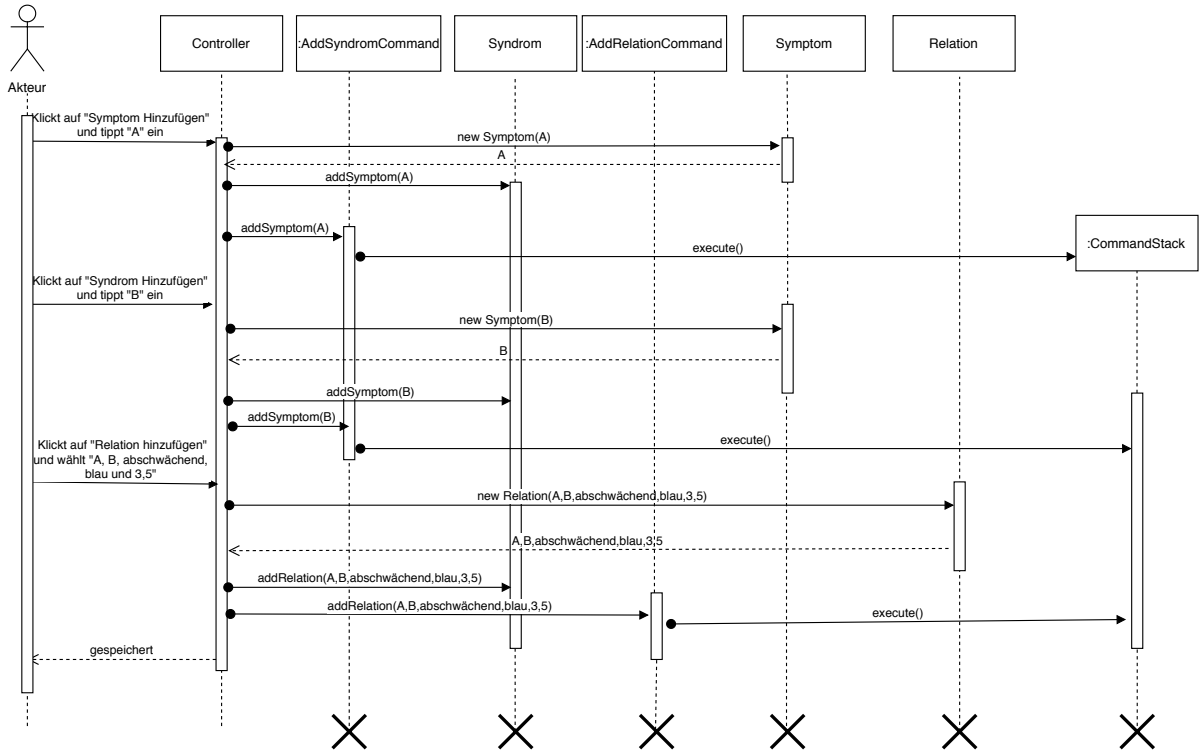

Abbildung 8: Sequenzdiagramm 1

Der Benutzer befindet sich bereits im Ersteller-Modus und möchte zunächst ein Symptom in eine bereits erstellte Vorlage einfügen. Hierfür klickt der Benutzer auf den Button »Symptom hinzufügen«, dann muss nur noch eine Bezeichnung für das Symptom vom Benutzer eingegeben werden. Anschließend wiederholt der Benutzer den Prozess, da er insgesamt zwei Symptome einfügen möchte. Nachdem es dem Benutzer gelungen ist, zwei Symptome einzufügen, möchte er die beiden Symptome mittels einer Relation verbinden. Dafür muss er auf den Button »Relation hinzufügen« klicken und dazu angeben, welche der beiden Symptome er verbinden möchte. Zudem muss er noch die Farbe und Stärke angeben und um welche Art von Relation es sich handelt. Im Anschluss wird das Dokument gespeichert.

8.2 Ändern der Farbe einer Kante sowie Löschen eines Knotens

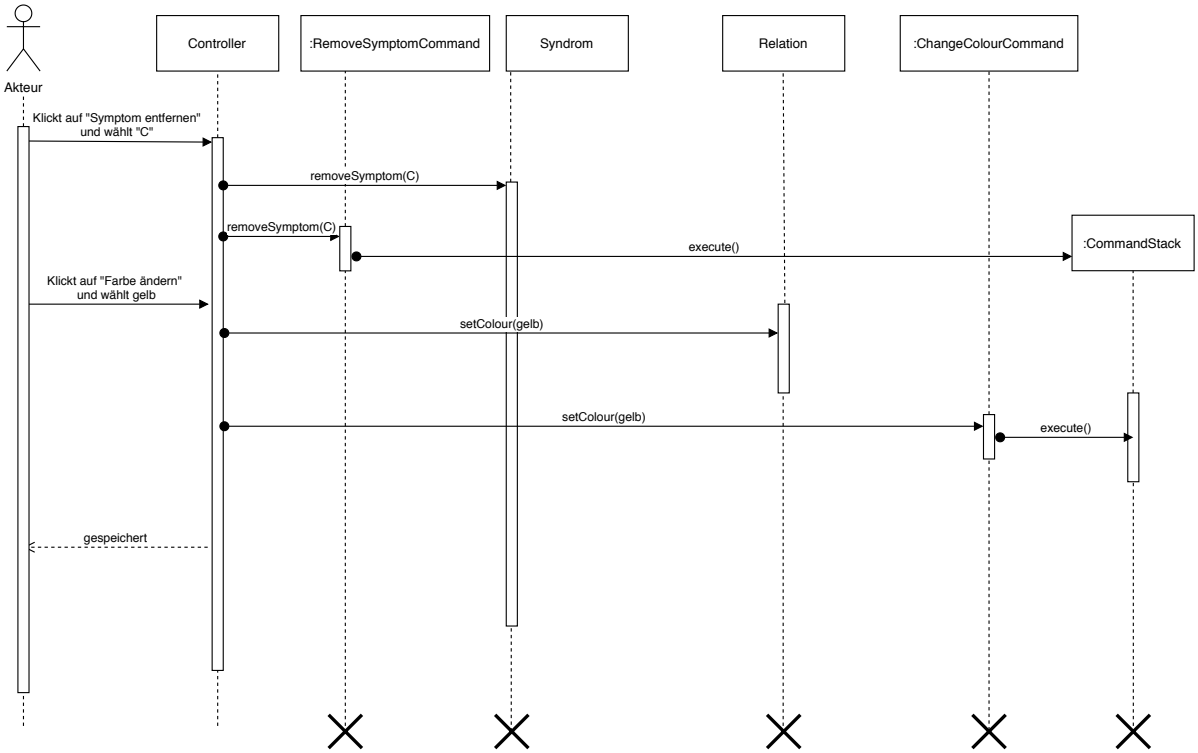

Abbildung 9: Sequenzdiagramm 2

Wieder befindet sich der Benutzer im Ersteller-Modus und möchte erst ein Symptom löschen und danach die Farbe einer Relation ändern. Nun muss der Benutzer den Button »Symptom entfernen« drücken und das nicht mehr benötigte Symptom auswählen. Um die Farbe einer Relation zu ändern, muss der Benutzer zunächst mit der rechten Maustaste auf die Relation klicken und »Farbe ändern« wählen. Jetzt muss der Benutzer nur noch die gewünschte Farbe auswählen. Danach wird das Dokument erneut gespeichert.

9 Evolution

In diesem Abschnitt wird beschrieben, welche Änderungen an der Architektur vorzunehmen sind, wenn sich eine Reihe von Anforderungen ändern bzw. zu den bereits berücksichtigten hinzukommen. In diesem Rahmen zeigen wir weiterhin auf, welche Entwicklungspotentiale unser Spiel hat.

9.1 Bereitstellung für als Android-, iOS- und HTML5-Version

Durch Nuzung von LibGdx lässt sich unser Spiel relativ leicht auf andere Plattformen portieren: Ein jeweiliges plattformspezifisches Modul (in dieser Archtitekturbeschreibung das Modul »Desktop«) greift im Wesentlichen auf den Inhalt des Core-Moduls zu und setzt das dort implementierte Spiel mit plattformspezifisch um.

Die bestehende Architektur sollte sich folglich relativ leicht um weitere Plattformen, namentlich Android, iOS und HTML5, erweitern lassen, wobei hierbei auf die Spezifika der Zielplattformen (wie etwa die unterschiedlichen Bildschirmgrößen) Rücksicht genommen werden muss. Da die Kryonet-Bibliothek jedoch nicht mit iOS kompatibel ist, muss für diese Plattform auf eine andere Netzwerk-Bibliothek zurückgegriffen werden.

Gerade bei der browserbasierten HTML5-Version sollte zudem eine andere Client-Server-Architektur diskutiert werden, die die Funktionalität des Servers auf einen dezidierten Server verlagert.

9.2 Mehrsprachigkeit der GUI

9.3 Bereitstellung eines Map-Editors