Projekt Software-Entwicklung 3

Andrea Feurer - af099
Lena Sophie Musse - lm138
Marco de Jesus António - md131
Maximilian Dolbaum - md127
Simon Geupel - sg184
Timo Waldherr - tw086

Inhaltsverzeichnis

Einführung und Ziele

Im Rahmen des Software-Entwicklung 3 Projekts wurde eine Multiplayer-Desktop-Version des Spiels “Die Siedler von Catan” entwickelt, welche im Heimnetz gespielt werden kann.

Das Ziel des Projekts war sowohl das Kennenlernen der Game-Development-Umgebung “Unity”, die Programmiersprache C# und der Netzwerktechnologie. Dabei stand das Erlernen von Socketprogrammierung und das Versenden von Spieldaten über das Internet im Mittelpunkt. Des Weiteren kamen zum Programmieren auch 3D-Modellierungen und Design hinzu.
Es sollte ein gut spielbares Mehrspieler-Spiel entstehen, mit schön gestalteter grafischer Oberfläche.

Wie kam das Team zustande

Wir haben Kommilitonen, die wir bereits kannten, gefragt ob sie Interesse an einem Spiel hätten. Schnell kamen wir sechs zusammen.

Wie kam die Idee zustande

Klar war, dass wir ein Spiel programmieren möchten, um neue Erfahrungen zu sammeln, da wir in den vorherigen Semestern nichts mit Spielen gemacht hatten.
Zunächst gab es 2 Ideen: Multiplayer Siedler von Catan oder ein auf Musik basiertes Spiel, welches sich dynamisch zu der abgespielten Musik verändert. Durch demokratische Abstimmung entschieden wir uns dann für Siedler von Catan. Dadurch, dass die Mehrheit der Mitglieder dieses Spiel seit der Kindheit begeistert spielten, war das der klare Favorit.

Wie wurden die Aufgaben / Rollen verteilt
Es haben sich direkt Interessenten zu den verschiedenen Bereichen gefunden.

Design - Lena Musse

Lena Musse hat aus Interesse an Design die Modellierung in Blender und das Design des UI's übernommen.

Game Logik - Maximilian Dolbaum / Timo Waldherr / Marco de Jesus António / Lena Sophie Musse

Mit großem Interesse am Umsetzen von Regeln und Funktionen in logischen Programmcode, entschieden sie sich zum Arbeiten an der allgemeinen Game Logik und der Verknüpfung zwischen Logik und UI.

Networking - Andrea Feurer / Simon Geupel

Die beiden Mitglieder hatten in vorangegangenen Projekten schon mit Design und GUI Entwicklung gearbeitet und hatten deshalb großes Interesse am Erlernen der Networking Techniken.

Management - Jeder

Wir haben uns alle bei der Planung mit eingebracht, die Moderation der Meetings wurde jedoch hauptsächlich von Maximilian Dolbaum und Lena Musse übernommen.

Aufgabenstellung

Rahmenbedingungen

In einer Gruppe aus fünf bis sechs Studierenden, soll innerhalb eines Semesters ein neues System geschaffen werden, welches gewissen Anforderungen (siehe Benotungsschema) gerecht wird.

Anforderungen

Neben einer allgemeinen Gesamtqualität mit einer guten Software-Architektur und Clean Code, müssen auch zwei Bereiche des Wahlpflichtbereiches (Parallelisierung – Datenbanken – Networking – UI – Eigene Generics) mit eingebunden werden. Außerdem wird eine gute Organisation und GitLab-Nutzung, sowie Testing gefordert.

Minimum viable product

Mithilfe von Unity wird in C# ein Multiplayer Spiel für zwei bis vier Spieler entwickelt.
Die Kommunikation der einzelnen Clients erfolgt über einen autoritären Server, welcher die Spieldaten und -logik enthält.
Durch die Eingabe der IP-Adresse, können sich Spieler im selben Netz über Peer-to-Peer verbinden.
Die Spielmechaniken beschränken sich auf die nötigen Basics, würfeln, Rohstoffe erhalten, bauen, mit der Bank vier zu eins tauschen und Entwicklungskarten (=Siegpunkte) kaufen.
Der Spieler, der gerade an der Reihe ist, kann durch das UI und das Spielbrett interagieren.
Der Sieger und somit das Ende des Spiels wird automatisch bestimmt.

Qualitätsziele

Erweiterbarkeit

Da unser MVP nur die grundlegenden Spielmechaniken beinhaltet, ist uns eine gute Erweiterbarkeit sehr wichtig, um die restlichen features später mit wenig Aufwand einbauen zu können. Auch in Hinsicht auf Spiel-Erweiterungen ist es unabdingbar die Erweiterbarkeit zu gewährleisten.

Modularität

Server und Client sollen streng getrennt sein. Für die Kommunikation sollen auf beiden Seiten Schnittstellen definiert werden, um die Netzwerkschicht austauschbar gestalten zu können.

Sicherheit

Das Spiel soll gegen einfache Manipulation clientseitig geschützt sein. Aus diesem Grund sollen alle Clients durch einen autoritären Server gesteuert werden. Das bedeutet, jeder Client hat nur die Möglichkeit Funktionen anzufragen, kann sie aber nicht selbst ausführen. Der Server prüft diese Anfragen und handelt entsprechend der ihm vorliegenden Regeln.

Randbedingungen

Technische Randbedingungen

Implementierung in C#
Um das Rad nicht völlig neu erfinden zu müssen, haben wir uns dafür entschieden, mit Unity zu arbeiten, was eine Implementierung in C# voraussetzte. Da die Sprache Java wirklich sehr ähnlich ist, hat das uns aber kaum Probleme bereitet.

Arbeiten mit Unity
Direkt in Unity wurde nur die graphische Oberfläche aufgebaut, und dementsprechend Unity-Skripte verwendet. Die eigentliche Entwicklung des Spiels wurde durch unabhängige Skripte realisiert. Mit dieser Trennung wäre es in Zukunft möglich die Oberfläche durch ein anderes C# kompatibles System auszutauschen.

Networking
Wie auch alle anderen Komponenten wurde das Networking ohne Unity Unterstützung umgesetzt.
Für die Kommunikation wurde das Konzept Client/Server Sockets über TCP gewählt. Die Spieler sollten sich im Heimnetzwerk mit dem Server verbinden können.
Um das Cheaten zu erschweren, wurde der Server autoritär gestaltet und der Client hat die Möglichkeit Funktionen anzufragen.
Eine ausführlichere Beschreibung befindet sich unter dem Punkt “Entwurfsentscheidungen”.

Verschiedene Betriebssysteme
Unity stellt eine Vielfalt an Build Möglichkeiten zur Verfügung. Daher sollte die gebaute Version auf fast allen Betriebssystemen funktionieren.
Das ist auch ein Grund warum wir uns für Unity entschieden haben. Als Game-Engine kommt sie einem mit sehr vielen Features entgegen.

Organisatorische Randbedingungen

Team
Teamgröße von sechs Studierenden, Ansprechpartner in Form eines Tutors.

Zeitplan
Das Projekt wurde Anfang April begonnen, Abgabetermin ist der 18.07.2021.
Da wir uns alle kaum oder gar nicht mit Unity auskannten, reservierten wir die ersten Wochen des Semesters für das Kennenlernen von Unity und für das austesten von Funktionen. Für das Fertigstellen des Netzwerkteils des Spiels setzten wir uns den 10. Mai als Abschlussdatum. Dieses konnten wir nicht ganz einhalten und wurden in der darauffolgenden Woche fertig.
Der Programmcode hatte das Ziel bis zum 30. Mai fertig zu werden, wobei wir die Pfingstferien zum Arbeiten mit einplanten.
Bis zum Abgabetermin sollten Tests und die Dokumentation fertig werden. Zusätzlich sollte der Programmcode etwas aufgeräumt werden.

Versionierung mit GitLab
Wir haben uns für GitLab entschieden, da wir alle bereits Erfahrungen mit diesem Tool gesammelt hatten. Außerdem war es eine Vorgabe, die Versionierung mit GitLab umzusetzen. Nach dem ersten Projekt Setup auf dem master branch haben wir diesen gesperrt. Somit konnte man nicht mehr direkt auf den master pushen und musste alle Änderungen über merge requests anfragen. Der Master gilt seit diesem Zeitpunkt als unser “fertiges Produkt”. Das bedeutet hier befindet sich zu jedem Zeitpunkt des Projektes eine funktionierende Instanz.
Für alle anderen Branches wurde mit prefixes gearbeitet. (siehe Konventionen).

Regelmäßige Team besprechungen und Absprachen mit dem Tutor
Um uns gegenseitig auf dem Laufenden zu halten und den Projektfortschritt zu überwachen, vereinbarten wir einen festen Termin (Mittwoch 16:00 Uhr) für Gruppenbesprechungen. Hier planten wir die Prioritäten bis nächsten Mittwoch und die nächsten Schritte. Außerdem informierten wir uns über den Fortschritt der einzelnen Teams und wichtige Neuerungen.

Konventionen

Dokumentation
Die Dokumentation hält sich an den Aufbau des deutschsprachigen arc42-Templates.

Sprache
Mit Ausnahme der Dokumentation ist alles auf Englisch geschrieben. Quellcode, Kommentare, Issue Tracking, sowie das Ingame-UI.

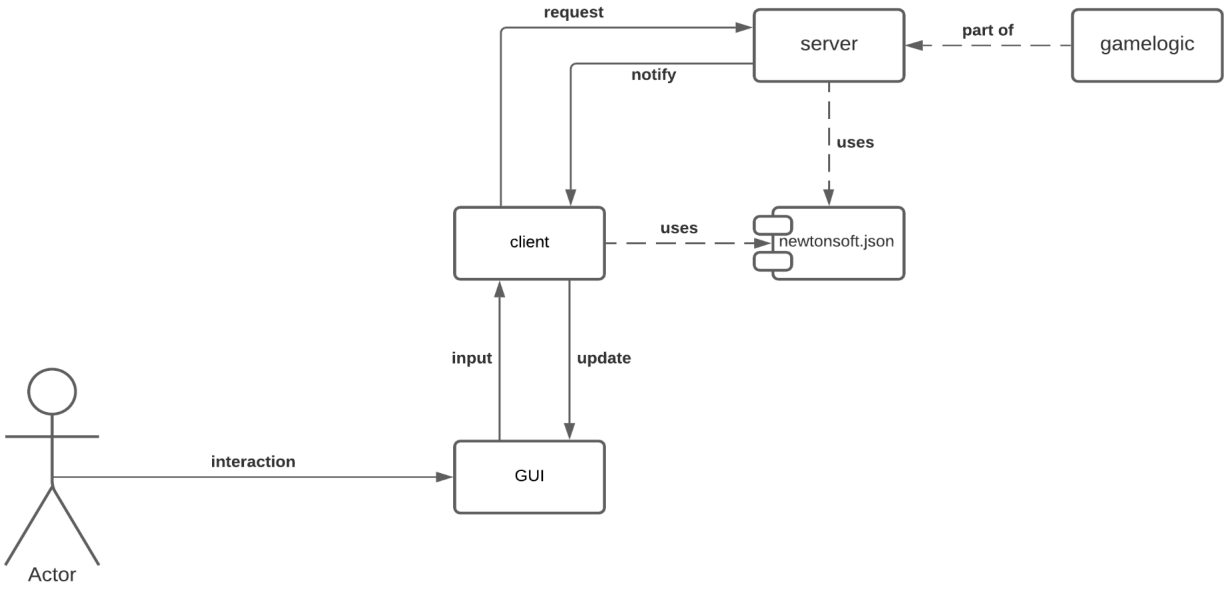
Coding Conventions
C# Konventionen waren etwas gewöhnungsbedürftig und haben uns nicht so richtig zugesagt. Deswegen entschieden wir uns dazu, stattdessen die JAVA Konventionen anzuwenden. Damit konnten wir auch sicherstellen, dass sich jeder daran hält, denn alle haben Vorkenntnisse in JAVA. Variablen und Methoden sollten immer nach ihrer Funktion benannt sein.

Benennung der Branches
Bei der Benennung der Branches einigten wir uns auf das Muster “Zweck/Beschreibung”. War ein Branch einem Issue in GitLab zugeordnet, begann die Beschreibung des Branches mit der Issue Nummer. Implementierte jemand von uns beispielsweise eine neue Handelsfunktion, die dem Issue #36 zugeordnet wäre, so könnte der zugehörige Branch “feature/36-trading” heißen.

Benennung der commit messages
https://gitmoji.dev/ Anhand der hier beschriebenen Emojis wurden auch die commits “gelabelt”. Das bedeutet wenn man einen Bug fixt bekommt der entsprechende commit ein 🐛 vorangestellt. Das macht es später einfacher im Projekt ältere commits wieder zu finden. So kann man, wenn man weiß, dass man nach einem Bugfix sucht, nur auf dieses Emoji achten muss.

Kontextabgrenzung

Fachlicher Kontext



Spieler (Benutzer)
Der Spieler kann über die Benutzeroberfläche mit dem Spiel interagieren. Der Benutzer hat die Möglichkeit das Spiel zu hosten, in diesem Fall wird bei ihm der Server gestartet. Gleichzeitig joint er der Lobby aber als Client. Alternativ kann er einer bereits bestehenden Lobby als Client beitreten, der Server liegt dabei bei einem anderen Benutzer.

Client
Der Client ist bei jedem Spieler lokal angelegt, baut eigenständig eine Verbindung zum angegebenen Server auf und nimmt die Anweisungen des Benutzers entgegen. Anhand der aufgerufenen Methode wird ein entsprechendes Paket an den Server gesendet. Als Antwort erhält er wiederum ein Paket vom Server. Durch die angegebenen Methode im Paket, werden die Daten in der richtigen Methode verarbeitet und an das GUI weitergeleitet.

Server
Der Server wird auf dem Host Computer unter dessen IP Adresse und dem Port 50042 gestartet. Sobald der Server gestartet ist, lauscht dieser auf Verbindungsanfragen und verarbeitet diese. Er hält sich eine Liste von verbundenen Clients und kann so zu jedem Zeitpunkt die eingehende Kommunikation einem Spieler zuordnen.
Während des laufenden Spiels erhält der Server in den gesendeten Paketen die Information, welche Methode aufzurufen ist. Von den aufgerufenen Methoden wird alles an die Gamelogic weitergeleitet, welche die Daten verarbeitet und über bereitgestellte Server-Methoden zurück an den Client leitet.

Socket
Für die Kommunikation zwischen Client und Server verwenden wir Sockets über TCP. Anhand dieser ist der Server immer in der Lage seinen Gegenüber zu erkennen.

newtonsoft.json
Die Bibliothek stellt uns zwei Methoden zur Serialisierung und Deserialisierung von Objekten, beziehungsweise Strings, zur Verfügung. Damit lassen sich die zu versendenden Daten in einem Objekt speichern. Erst wenn diese versendet werden müssen, wird das Objekt serialisiert, sodass ein String entsteht. Diesen kann der Empfänger wieder deserialisieren und in ein Objekt verwandeln, mit dem er arbeiten kann.

Paket
Server und Client kommunizieren über Pakete, welche mit Hilfe der newtonsoft.json-Bibliothek (de-)serialisiert werden. Ein Paket enthält den Methodennamen, welche der Server bzw. der Client ausführen soll, sowie alle benötigten Spieldaten. Der Empfänger findet über einen Switch-Case die auszuführende Methode.

Lösungsstrategie

Technologieentscheidungen

Unity

Anfangs war die Idee, für das Projekt eine einfache, eigene Gameengine zu programmieren. Doch da wir schnell merkten, dass der Aufwand hierfür viel zu groß wäre und somit den Rahmen des Projekts sprengen würde, entschieden wir uns dazu, Unity als Gameengine zu verwenden.

Unity stellt Spieleentwicklern umfassende Funktionen zur Spieleentwicklung bereit. Für unser Projekt haben wir davon aber hauptsächlich die Renderengine, das UI Framework “Text Mesh Pro” und das Testframework genutzt.
Um Unity nutzen zu können, mussten wir uns die Programmiersprache C# aneignen, da diese fest in Unity integriert ist.

Server

Um die Server-Client-Kommunikation einfach zu gestalten, entschieden wir uns dazu, die Pakete als JSON zu verschicken. So können die Daten einfach ver- und entpackt werden. Die Idee war, ein Objekt zu haben, in dem man alle zu versendenden Daten speichert, dieses dann in einen String umwandelt und dann versenden kann.

Pipeline

Um zu vermeiden, dass man bei jedem Merge auf den Master das Projekt neu builden und testen muss, haben wir eine Pipeline eingerichtet, dass dies nicht vergessen werden kann. Sie soll bei jedem gestellten Merge-Request durchlaufen. Durchlaufen bedeutet, dass Unity das Projekt einmal vollständig builden soll. Damit stellen wir sicher, dass beim Mergen der neue Code immer noch lauffähig ist und vor allem beim Mergen auf den Master unser “fertiges” Spiel nicht kaputt macht. In der Konfiguration befinden sich auch noch die Möglichkeiten, nur Tests zu triggern oder das Spiel für WebGL zu builden. Damit wären wir bei Fertigstellung des Spiels sogar in der Lage eine GitLab Seite zu Hosten, auf der das Spiel laufen kann.

Entwurfs- und Architekturmuster

Server-Client-Kommunikation

Zur Umsetzung unseres Ziels, ein Spiel mit Netzwerkanbindung zu entwickeln, realisierten wir eine Client-Server Architektur. Genaueres hierzu ist in dem Abschnitt Entwurfsentscheidungen beschrieben.

Ordnerstruktur

Die Ordner sind nach logischen Aufgabenbereiche sortiert.
Es gibt Ordner für das Networking, das Board, die Player und das UI, welche fast alle in weitere Unterordner aufgeteilt sind (siehe Entwurfsentscheidungen).
Zusätzlich gibt es einen Ordner für alle Enums und die Prefab Factory.

Organisatorische Entscheidungen zur Erreichung der wichtigsten Qualitätsanforderungen

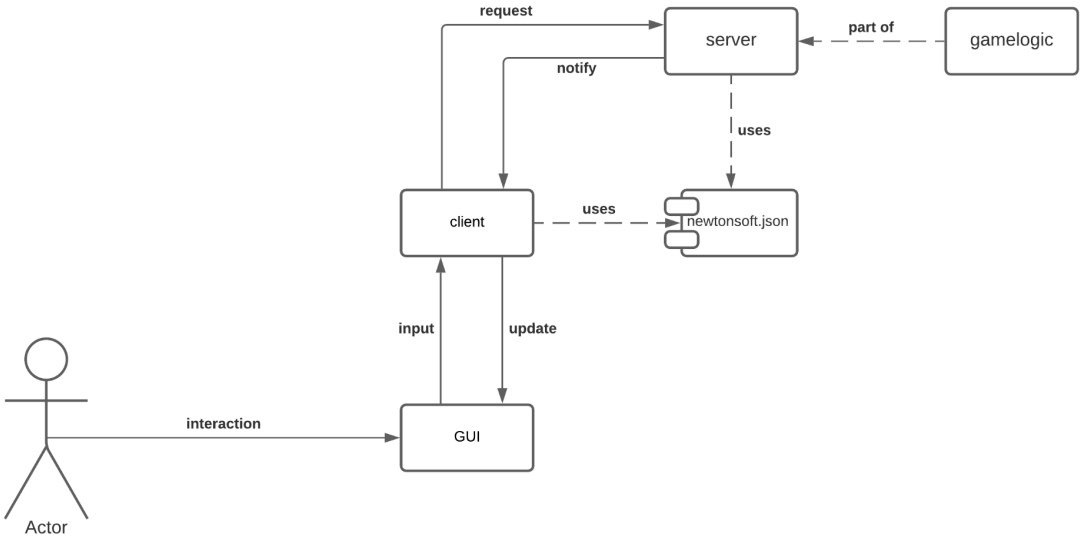
Zu Beginn des Projekts wollten wir zunächst planen, welche Funktionen implementiert werden sollen und wie die Architektur des Projekts aussehen soll. Jedoch stellten wir schnell fest, dass uns dies wegen unseres fehlenden Wissens über die Funktionsweise von Unity und der noch nicht vorhandenen Client-Server Architektur nicht möglich war.
Aufgrund dessen beschlossen wir zunächst mit einer einfachen Implementation zu beginnen, um Unity kennen zu lernen.
Nachdem wir einen Prototyp programmiert hatten, trafen wir uns um die weitere Architektur des Projekts zu besprechen. Die getroffenen Entscheidungen sind unter dem Punkt “Entwurfsentscheidungen” aufgeführt. Zu diesem Zeitpunkt legten wir auch als unseren Meilenstein unser MVP fest, welches wir bis zu den Pfingstferien fertigstellen wollten. Nach dem erreichen des MVP sollten optionale Funktionen des Spiels implementiert werden, sofern dafür noch Zeit bleibt.

Über das Semester hinweg haben wir uns einmal wöchentlich getroffen, um uns auf den jeweils aktuellen Stand zu bringen, Probleme zu besprechen und das weitere Vorgehen zu besprechen.
Die Arbeit fand in kleineren Gruppen statt, welche sich nach der Architektur richteten. Dabei programmierten wir entweder in Zweiergruppen, oder alleine. So konnten auch Absprachen getroffen werden, ohne dass direkt die ganze Gruppe involviert war.
Um Probleme und anstehende Aufgaben festzuhalten, nutzten wir Issues auf Gitlab. Aus den Issues wurde ein Branch erzeugt, auf dem das spezifische Problem bearbeitet wurde. Das anschließende Mergen wurde meistens von einem Gruppenmitglied übernommen und dabei auf die Funktionalität überprüft. Somit landeten nur fertige Features auf dem Main Branch. Dieser wurde gesperrt, sodass nur auf Branches gearbeitet werden konnte um die Lauffähigkeit des Main zu gewährleisten.

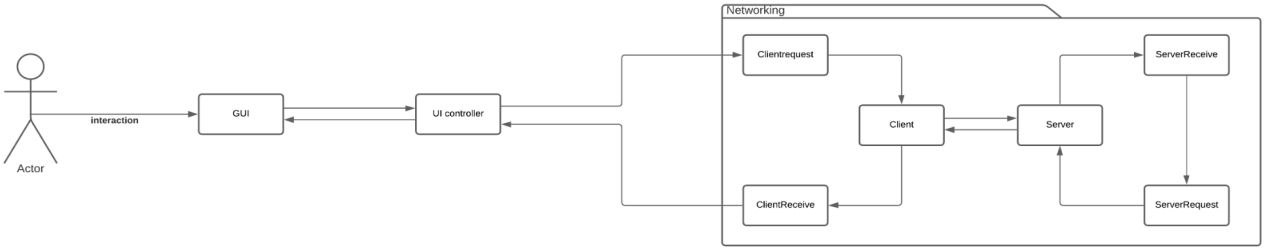
Ursprünglich wollten wir nur Programmcode auf den Main zulassen, der vorher durch Unit Tests getestet wurde. Jedoch hatten wir relativ lange Probleme mit dem Testing Framework, weshalb die Tests erst gegen Ende des Projekts implementiert werden konnten. Näheres dazu wird unter dem Punkt “Risiken und technische Schulden” erläutert.

Bausteinsicht

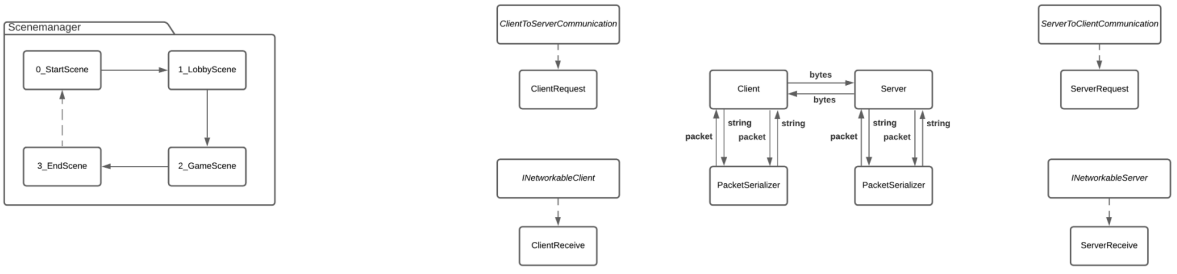
Ebene 1 - Kontextabgrenzung



Ebene 2

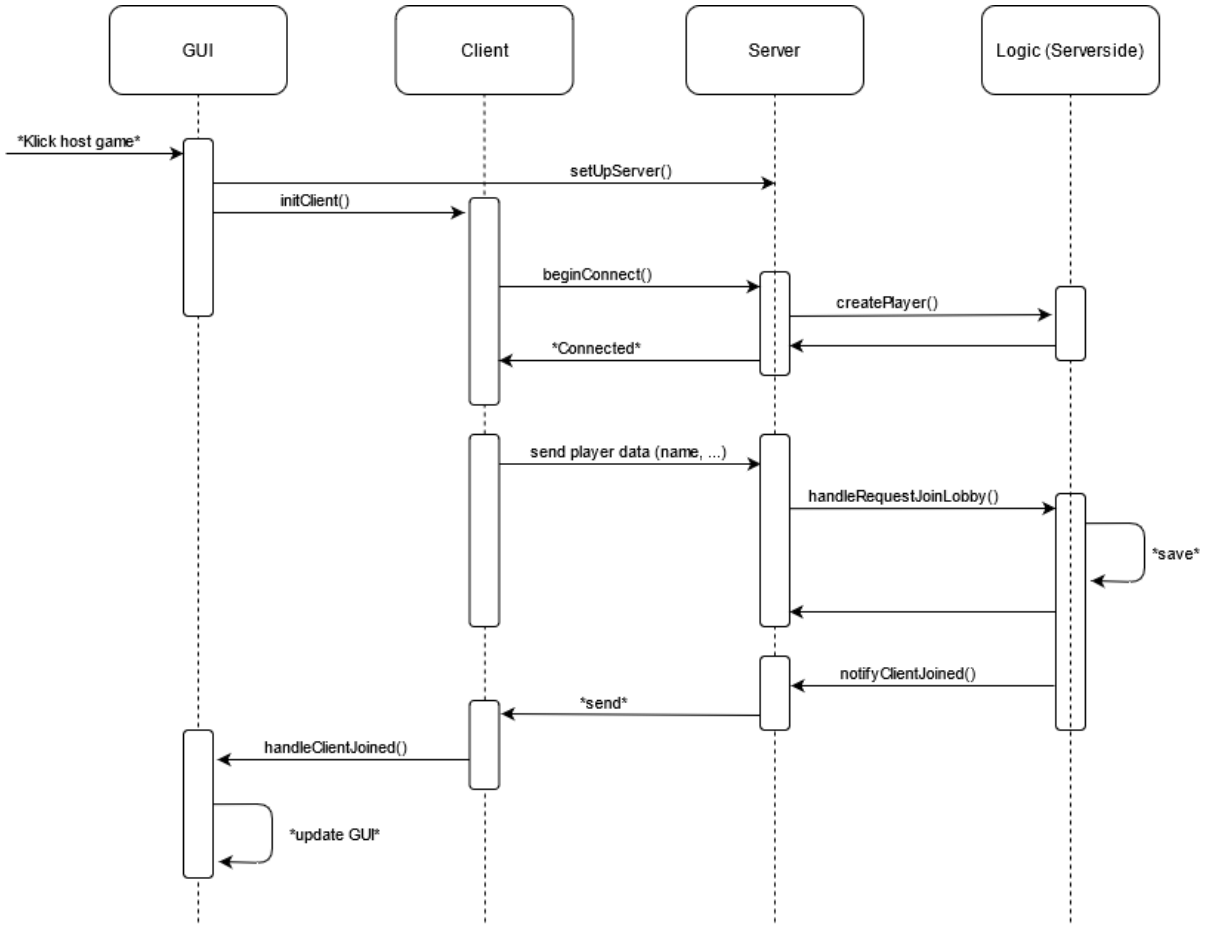


Ebene 3



Laufzeitsicht

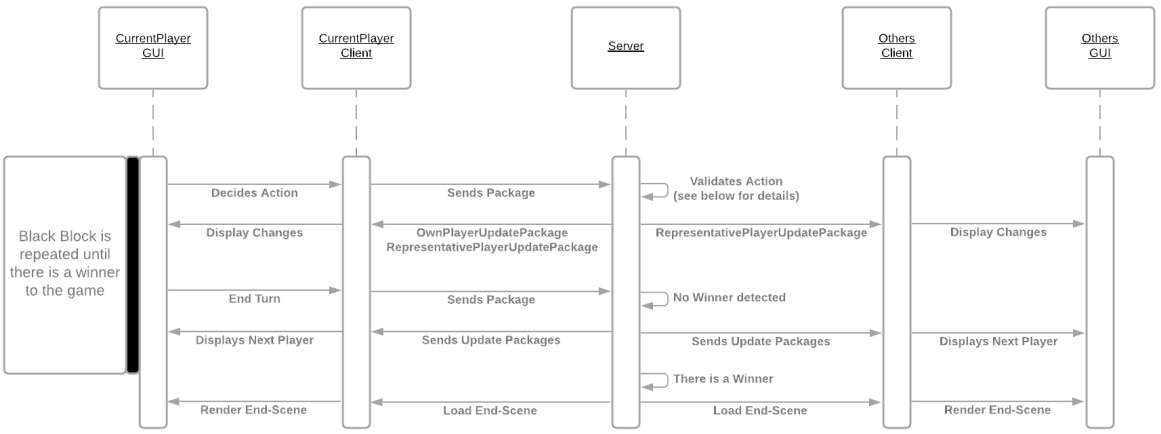
Szenario 0: Verbindungsaufbau



Szenario 1: Spielablauf

Nach dem Verbindungsaufbau kommt man in die Startup-Phase, welche Laufzeit-Technisch gleich aussieht, wie die restlichen Runden des Spiels. Lediglich gibt es bei der Validierung der Aktion einen Unterschied.

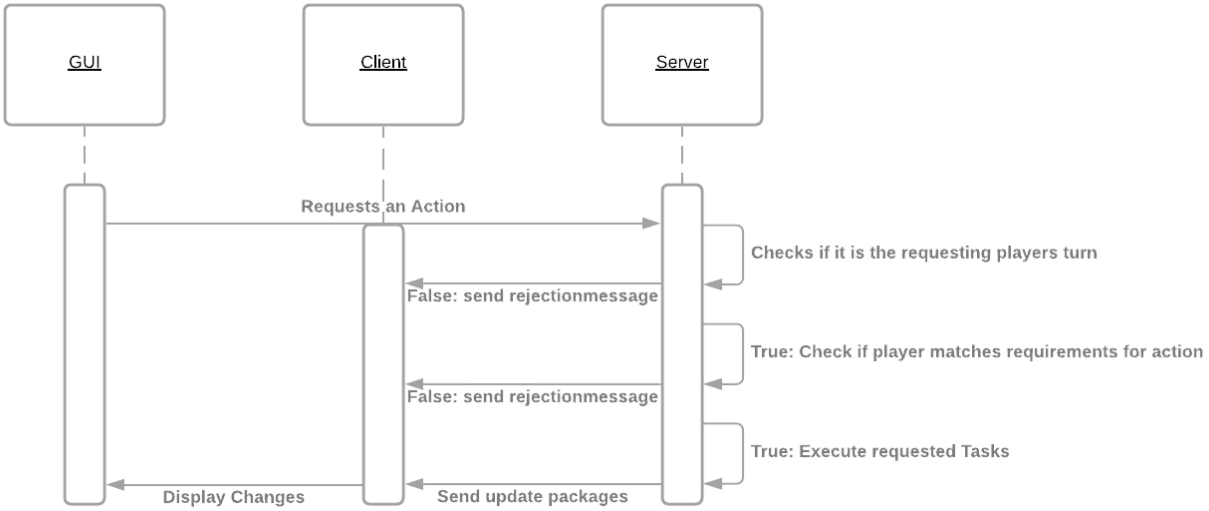
Der Schwarze Balken (loop) wird solange ausgeführt, bis ein Spieler das Spiel gewinnt. Daraufhin kommt man in die End-Szene.



Szenario 2: Unerlaubte Aktionen

Jegliche Serveranfrage wird ignoriert, wenn der Sender nicht dem Spieler, der am Zug ist, entspricht.
UI Anzeigen sind aber noch möglich, wie z.B. die Baukosten oder das Trade-Menü.

Um Aktionen auszuführen, wenn ein Spieler am Zug ist, müssen auch bestimmte Kriterien erfüllt sein. So wird bei jeder erlaubten Aktion auf dem Server überprüft, ob der Spieler alle Anforderungen wie genügend Ressourcen oder die richtige Position zum Bauen hat.



Querschnittliche Konzepte

Authentifizierung und Autorisierung

Für ein mehrspieler Spiel ist ein Server essentiell. Dieser muss die verschiedenen Clients zuverlässig erkennen können, um einen funktionierenden Spielfluss zu gewährleisten.

Der Spielbeitritt

Ein Client startet lokal einen Server, sobald dieser auf die Schaltfläche “Host Game” im Hauptmenü drückt. Gleichzeitig wird auch ein Client gestartet, der über den Localhost dem Server beitritt. Andere Spieler können dem Server über die IP-Adresse des Host Computers, welche in der Lobby Szene angezeigt wird, beitreten. Der offene Port des Servers ist im Programmcode fest codiert.

Client Identifizierung

Beim Beitritt eines Clients wird diesem serverseitig eine einzigartige zufällige Identifikationsnummer zugewiesen. Bei jedem weiteren Paketaustausch wird serverseitig diese ID über den verbundenen Socket bestimmt. Umgekehrt ist über die ID auch der Socket bestimmbar, über den der jeweilige Client erreichbar ist.

Autorisierung

Bei jeder Anfrage eines Clients wird serverseitig geprüft, ob es diesem erlaubt ist, die angefragte Aktion auszuführen. Dazu werden die ID des anfragenden Clients und die des Clients, der gerade an der Reihe ist, abgeglichen. Ist es dem Client erlaubt, die Aktion auszuführen, so wird diese serverseitig kalkuliert und das Ergebnis zum Aktualisieren des UI an den Client gesendet. Ist des dem Client nicht erlaubt die Aktion auszuführen, wird eine Fehlermeldung zurück gesendet.

Graphical User Interface

Für ein optimales Spielerlebnis ist die graphische Benutzeroberfläche von großer Bedeutung und deshalb auch ein wichtiger Punkt für uns.

Das Spielbrett

Da das Spielbrett in jeder Runde neu angeordnet wird, besteht es wie das Brettspiel-original aus einzelnen Landschafts- hexagons, welche in Blender erstellt wurden. Dabei versuchten wir, ein Gleichgewicht zwischen Übersichtlichkeit und künstlerischen Details zu finden. Dies wurde durch den detail-einschränkenden “low-poly-style” und verhältnismäßig großen Elementen umgesetzt.

Das UI

Auch das UI-Overlay sollte nicht nur übersichtlich, sondern auch thematisch, passend und spannend sein. Wir entschieden uns für einfache Holzbretter und ein in Holz gerahmtes Pergament, was eine alte und landwirtschaftliche Atmosphäre vermitteln soll. Bei der Schrift entschieden wir uns für “deutsch”, da diese alt wirkt, aber immer noch gut leserlich ist. In allen Szenen sind die gleichen Elemente zu finden. Auch Elemente des Spielfelds (z.B. bei der Rohstoffanzeige) tauchen in der UI wieder auf und verknüpfen stilistisch beide Ebenen miteinander.

Die Interaktion

Die Interaktion läuft fast ausschließlich durch Buttonclicks ab. Nur in der Startscene muss der Name und eventuell die IP-Adresse eingegeben werden. In der Gamescene sind alle Aktionen, welche der Spieler, der gerade an der Reihe ist, ausführen kann, unten links aufgelistet. So wird zum Beispiel auch der “Baumodus” über einen Button gestartet. Zum Gebäude/Straßen platzieren, wird dann allerdings mit der Karte interagiert.

Das Layout

Generell sollen alle wichtigen Informationen jederzeit abrufbar sein, ohne dass die Oberfläche zu voll wird. Dies geschieht zum Beispiel durch die Nutzung von Pop-ups in der Startscene, sowie in der Gamescene (Trademenu, Developmentcards und Buildingcosts). Wie oben bereits erwähnt, sind alle Actionbuttons an einer Stelle zu finden, was es dem Spieler leicht macht, zu verstehen, was für Möglichkeiten er im Spiel hat. Die eigene Rohstoffanzeige ist unten mittig angebracht, da sie ein sehr zentrales Element ist. Die Anzeigen der anderen Spieler hingegen werden etwas kleiner am oberen Bildschirmrand von links angeordnet. Sie sind jederzeit gut zu erkennen, ziehen aber keinen zu großen Fokus auf sich.

Debugging

Hierzu muss man einen wichtigen Punkt betrachten. Die UI und alle Unity-Skripte laufen in einem sogenannten Mainthread. Da wir aber z.B. die komplette Client/Server Struktur außerhalb von Unity-Skripten geschrieben haben, was auch die Game Logik betrifft, kann ein auftretender Fehler nicht erkannt werden. Hierfür sind extra Try/Catch Blöcke notwendig, um eventuell auftretende Fehler abzufangen. Dieses Thema ist erst mitte des Projektes aufgetreten, als bei der vermeintlichen Kommunikation zwischen Server und Client das ganze Spiel abstürzte, aber kein Fehler in der Konsole zu sehen war.

Entwurfsentscheidungen

Frameworks und Bibliotheken

Json.NET von Newtonsoft

Für die Kommunikation zwischen Client und Server wurden JSON Pakete verwendet. Um diese serialisieren und deserialisieren zu können, wurde die Bibliothek Json.NET von Newtonsoft benutzt. In der Klasse “Packet” sind alle Felder für die Kommunikation vordefiniert. Wenn diese befüllt wurden, kann das gesamte Objekt mit dem Framework serialisiert werden. Sobald ein JSON-String empfangen wurde, kann dieser über das Framework in ein Packet-Object deserialisiert werden.

TextMeshPro

Für den Text in der UI wurde die von Unity zur Verfügung gestellte Bibliothek verwendet. Im Gegensatz zum normalen Text gibt es hier viele weitere Funktionen. Dabei wurde hauptsächlich das Character Spacing benutzt, um die Schrift besser leserlich zu machen. Auch die Color Gradient Funktion hat die Wirkung der Schrift an manchen Stellen aufgewertet.

Architekturprinzipien und Design Patterns

Client-Server Architektur

Für die Kommunikation über das Netzwerk wurde eine Client-Server Architektur verwendet. Es sollte eine möglichst starke Trennung zwischen beiden Seiten realisiert werden. Aus diesem Grund wurde Client- und Serverseitig jeweils ein Interface für das Versenden und Empfangen von Daten entworfen. Diese wurden dann später implementiert, um sowohl eine starke Kapselung, als auch eine Austauschbarkeit des Netzwerkteils zu erreichen.

Zu Beginn hatten wir den Ansatz eines **intelligenten Clients** verfolgt. Es erschien einfacher zu implementieren und war die erste Idee, die wir hatten. Wenn der Client selbst alle Informationen verwalten würde, müsste der Server nur grundlegende Funktionalität haben und lediglich die Informationen, die er erhält, an alle bestehenden Verbindungen weiterleiten.
Mit der Zeit kristallisierten sich immer mehr Nachteile dieses Ansatzes heraus. Einerseits verwirrten wir uns bei Überlegungen immer wieder selbst und bei fortschreitender Implementierung entstanden immer mehr “Hirnknoten”. Das große Problem bei diesem Ansatz ist das Cheaten. Wenn ein Client alles weiß und alles tun kann, was hält ihn davon ab, nicht hier und da ein paar Parameter zu verändern?
Und ist es überhaupt schlau jegliche Spiellogik auf jedem Client zu haben?

Nach einiger Recherche kamen wir zu dem Schluss, dass ein autoritärer Server bei Multiplayer Spielen der Standard ist. Dies würde dann auch Probleme mit Cheaten sehr einschränken. Aus diesen Gründen machten wir mitten im Projekt eine 180° Kehrtwende und verwarfen den gesamten Code, um dieses neue Konzept zu verfolgen.
Bei der Entwicklung des **autoritären Servers** befolgten wir das Motto “never trust a client”. So entwickelten wir einen Server, der alle Anfragen der Clients auf deren Validität überprüft.
Serverseitig werden alle Berechnungen ausgeführt. Die Ergebnisse dieser Berechnungen werden an einen oder mehrere Clients gesendet, die diese lediglich in der graphischen Oberfläche anzeigen müssen. Zur Überwachung der Verbindungen wird von dem Server ausgehend regelmäßig ein Ping an alle Clients gesendet, auf den diese antworten. Beidseitig wird die Zeit zwischen den einzelnen Pings überwacht, sodass ein Verbindungsverlust zuverlässig festgestellt werden kann.

Ein genauer Ablauf des allgemeinen Networking Flusses befindet sich unter dem Punkt **Szenario 0: Verbindungsaufbau.**

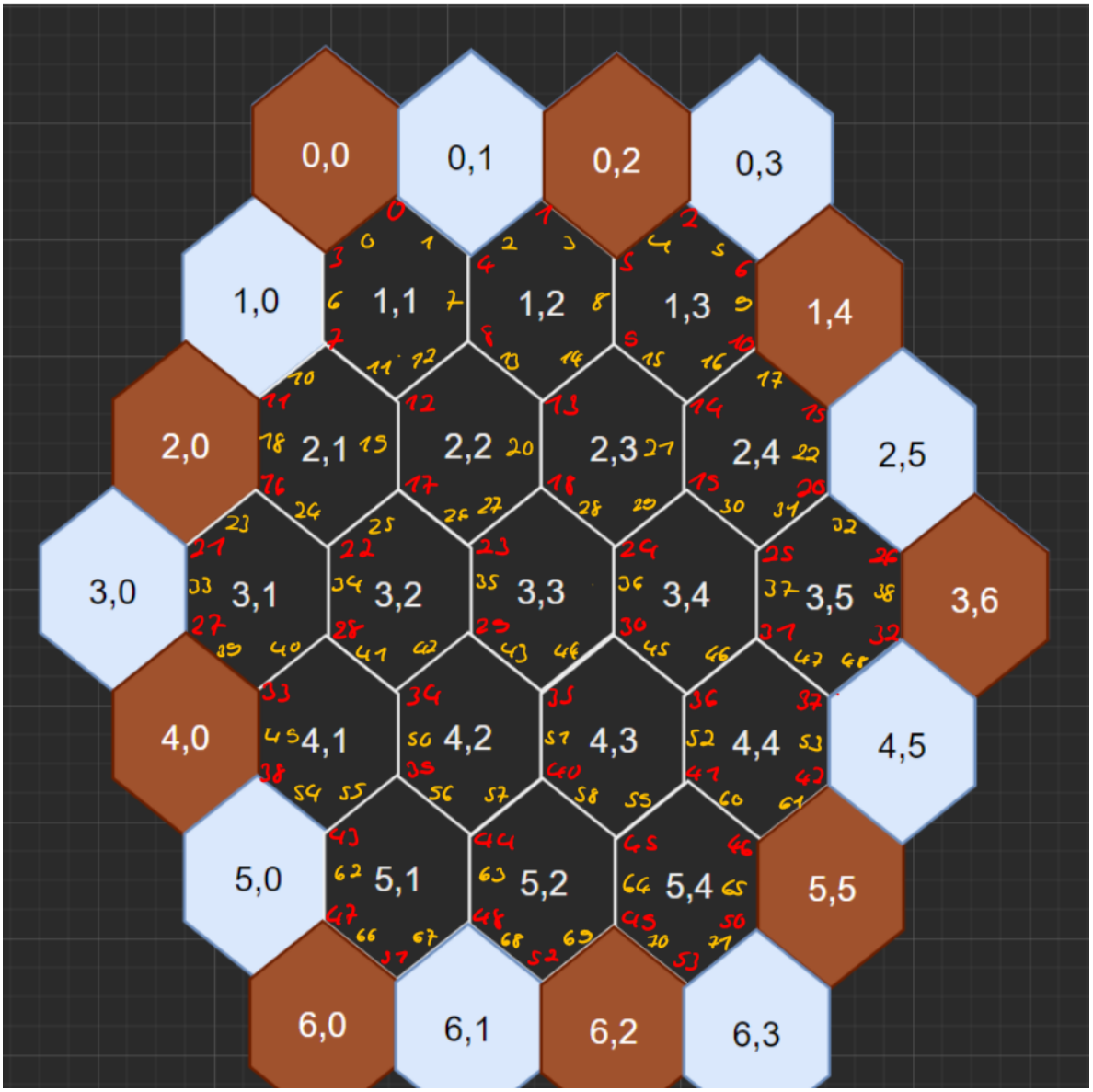
Singleton

Die Klassen Server und Client sind von dem Aufbau an das Entwurfsmuster Singleton angelehnt. Es kann pro Ausführungsumgebung immer nur ein Server und ein Client gestartet werden. Wird ein zweites Mal versucht, einen Server oder Client zu starten, wird geprüft, ob diese jeweils schon gestartet sind. Ist dies der Fall, passiert nichts.
Dies hat den Vorteil, dass auf diese Weise Konflikte bei der Adress- und Portvergabe vermieden werden können.

Aufbau des Spielbretts

Bei dem Brettspiel “Die Siedler von Catan” besteht das Spielbrett aus 37 Hexagons, welche zufällig auf dem Spielbrett in einem Hexagon angeordnet werden. Dabei hat jedes Hexagon 6 angrenzende Ecken und 6 angrenzende Kanten. Diese Ecken und Kanten überschneiden sich mit den Nachbarfeldern, weshalb sich mehrere Hexagons angrenzende Ecken und Kanten teilen. Dabei war für uns die Schwierigkeit, dies im Programmcode abzubilden, da sichergestellt werden musste, dass die benachbarten Hexagons die gleiche Referenz auf die Ecken und Kanten halten.

Speicherung der Hexagons



Unser ursprünglicher Ansatz war, die Hexagons in einem zweidimensionalen Jagged-Array zu speichern. Dabei waren die Arrays in zweiter Dimension von unterschiedlicher Länge, um das Spielbrett genau darzustellen. Um die Nachbarn im Array zu finden, nutzen wir “offset” Arrays, welche die Werte zur Berechnung deren Koordinaten enthalten. Werden diese Werte auf die Koordinaten, des aktuellen Hexagons addiert, erhält man die Koordinaten eines Nachbarn.
Hierbei war das Problem, dass es keine allgemein gültigen Offset-Werte für das komplette Board gab. In der Mitte und der unteren Hälfte unterscheiden sich die Offset-Werte von denen in der oberen Hälfte.

```
---[obere Hälfte]--- | -------[Mitte]-----|---[untere Hälfte]---
y |-1,-1, 0, 1, 1, 0 | -1,-1, 0, 1, 1, 0  | -1,-1, 0, 1, 1, 0
x | 0,-1,-1, 0, 1, 1 |  0,-1,-1,-1, 0, 1  |  1, 0,-1,-1, 0, 1
```

Um dies zu umgehen, entschieden wir uns, das Board so zu speichern, dass nur ein Offset-Array nötig ist. Hierzu haben wir allen Arrays zweiter Dimension dieselbe Länge gegeben, und die Positionen an passender Stelle mit Nullen aufgefüllt:

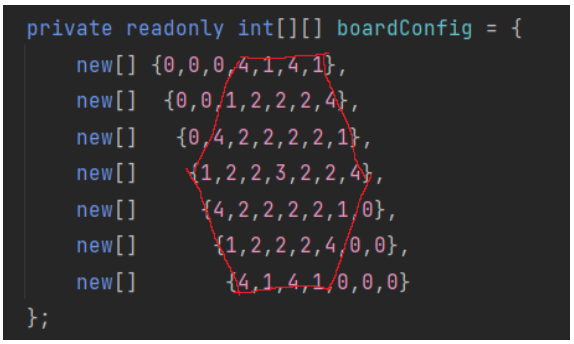
	q = 0	q = 1	q = 2	q = 3	q = 4	q = 5	q = 6
r = 0	(null)	(null)	(null)	3, 0	4, 0	5, 0	6, 0
r = 1	(null)	(null)	2, 1	3, 1	4, 1	5, 1	6, 1
r = 2	(null)	1, 2	2, 2	3, 2	4, 2	5, 2	6, 2
r = 3	0, 3	1, 3	2, 3	3, 3	4, 3	5, 3	6, 3
r = 4	0, 4	1, 4	2, 4	3, 4	4, 4	5, 4	(null)
r = 5	0, 5	1, 5	2, 5	3, 5	4, 5	(null)	(null)
r = 6	0, 6	1, 6	2, 6	3, 6	(null)	(null)	(null)

Danach ergibt sich das nachfolgende Offset-Array:

```
--------[new]--------
x|  1, 0,-1,-1, 0, 1
y| -1,-1, 0, 1, 1, 0
```

Board-Config

Da die Grundlegende Struktur des Boards in jedem Spiel die gleiche ist (Wasser- und Hafenfelder sind immer in abwechselnder Reihenfolge am Rand), entschieden wir uns dazu, ein boardConfig-Array anzulegen. Dieses Array hat die gleichen Dimensionen wie das spätere Array der Hexagons, und enthält Nummern, welche die unterschiedlichen Feldtypen codieren.



```
private readonly int[][] boardConfig = {
    new[] {0,0,0,4,1,4,1},
    new[]  {0,0,1,2,2,2,4},
    new[]   {0,4,2,2,2,2,1},
    new[]    {1,2,2,3,2,2,4},
    new[]     {4,2,2,2,2,1,0},
    new[]      {1,2,2,2,4,0,0},
    new[]       {4,1,4,1,0,0,0}
};
```

Config-Files

Unser ursprünglicher Lösungsansatz um die richtigen Nachbarn zuzuweisen, war bei Initialisierung des Boards über das Array mit den Hexagons zu iterieren, und jedem Hexagon die angrenzenden Nachbarn zuzuweisen.

Es stellte sich jedoch heraus, dass der dafür Notwendige Code so aufwändig war, dass es einfacher ist, die angrenzenden Ecken und Kanten in Config-Dateien zu spezifizieren und über diese zuzuweisen. Das ist nur möglich, da sich diese im laufe des Spiels und Spielübergreifend nicht ändern.

UI Skripte

Alles was die Repräsentation und Interaktion im UI betrifft, befindet sich in einem separaten UI Ordner. Dabei hatten wir zwei Ideen zur Strukturierung. Die erste Möglichkeit wäre, nach Input und Output zu trennen, also eine Klasse/Ordner, welche alle Buttons und EventListener hält, und eine weitere, die das UI aktualisiert.
Möglichkeit Nummer zwei: Themenorientiert, sodass jede Klasse für eine Aktion (z.B. Development Cards) oder Representation zuständig ist.
Da die verschiedenen Aktionen ohnehin von verschiedenen Teammitgliedern geschrieben und strukturiert wurden, entschieden wir uns für die zweite Möglichkeit. Außerdem führte dies zu weitaus übersichtlicheren Klassen.

Dabei gibt es erstmal eine weitere Unterteilung in Dicerendering, Interaction und Representation, da es Teile gibt, die nur repräsentiert werden müssen (eigene Ressourcen und die Übersicht der Spieler), in anderen Bereichen dagegen ist der User-input gefragt. Diese sind im Interaction-Ordner abgelegt.

Player

Nachdem wir unsere Server-Client-Struktur geändert haben, mussten wir uns nochmal Gedanken über die Player-Objekte machen, welche alle Informationen zu den Spielern halten (Name, Farbe, Ressourcen usw.). Schließlich sollte der Server alle diese Informationen und benötigten Methoden halten, andererseits müssen die Clients ständig Zugriff auf einige, aber nicht alle, Parameter haben, um diese zu repräsentieren. Wir entschieden uns dafür, diese Informationen nicht direkt an den das UI zu schicken, sondern noch Player-Klassen mit allen nötigen Parametern clientseitig zu speichern, und nach bestimmten Aktionen zu aktualisieren. Dadurch bleibt der Player ein abgegrenzter Kontext.
Wir unterteilten den Player also in den “ServerPlayer”, “RepresentativePlayer” und “OwnClientPlayer”.

Der ServerPlayer

Wird vom Server gehalten und enthält alle Parameter und Methoden, auf welche die Gamelogic dann zugreifen kann.

Der RepresentativePlayer

Dient jedem Client dazu, alle Mitspieler anzuzeigen. Dabei darf er neben Name, Farbe und ID nur die Anzahl der Siegpunkte, Ressourcen und Entwicklungskarten enthalten, nicht aber welche Karten genau die anderen Spieler auf der Hand halten. Außerdem gibt es keine Möglichkeit diese Daten zu manipulieren, außer über die update Methode, welche vom Server getriggert und mit den richtigen Werten befüllt wird.

Der OwnClientPlayer

Wird auch vom Client gehalten, dabei hat jeder Client aber nur seine eigenen Daten. Auch hier können die Parameter nur durch die update Methode geändert werden.

Zuerst wollten wir die Klassen hierarchisch aufbauen, da der “ServerPlayer” alles hat, was die Clientseitigen Player haben. Allerdings können die “ClientPlayer” nicht hierarchisch strukturiert werden. Da in C#, wie in JAVA, keine Mehrfachvererbung möglich ist, hätte mit unnötig komplizierten zusätzlichen abstrakten Klassen und Interfaces gearbeitet werden müssen, weshalb wir uns dagegen entschieden.

Qualitätsanforderungen

Erweiterbarkeit
Da unser MVP nur die grundlegenden Spielmechaniken beinhaltet, ist uns eine gute Erweiterbarkeit sehr wichtig, um die restlichen features später mit wenig Aufwand einbauen zu können. Auch im Hinblick auf Spiel-Erweiterungen ist es unabdingbar die Erweiterbarkeit zu gewährleisten.

Modularität
Server und Client sollten streng getrennt sein. Für die Kommunikation beider sollten Schnittstellen definiert werden, um die Netzwerkschicht austauschbar gestalten zu können.

Sicherheit
Das Spiel sollte gegen einfache Manipulation clientseitig geschützt sein. Aus diesem Grund sollten alle Clients durch einen autoritären Server gesteuert werden. Das bedeutet, jeder Client hat nur die Möglichkeit Funktionen anzufragen, aber kann sie nicht selbst ausführen. Der Server prüft die Anfragen und handelt entsprechend der ihm vorliegenden Regeln.

Performance
Das Spiel sollte performant laufen. Darunter verstehen wir, dass die Reaktion auf die Eingaben der Spielenden in kürzestmöglicher Zeit erfolgen. Ein langes Warten der Spielenden auf ein Ergebnis ihrer Eingabe soll so vermieden werden. Auch während dem Senden einer Anfrage an den Server bleibt das Spiel bedienbar und nimmt Eingaben entgegen.

Zuverlässigkeit
Client und Server sollten Zuverlässig, ohne Probleme und Verbindungsabbrüche zusammenarbeiten. Dazu wird unter anderem ein Keepalive-Ping von den Clients an den Server gesendet, um die Sitzung aufrecht zu erhalten.

Testing
Durch hinreichendes Testing wird sichergestellt, dass das Spiel zu jeder Zeit lauffähig bleibt und auf alle möglichen Arten von Eingaben reagieren kann, ohne abzustürzen. Ebenso wird durch Negativ-Tests sichergestellt, dass auch unerwartete Eingaben nicht zum Beenden des Programms führen.

Logging
Das Logging soll eine Lauffähigkeit und eine einfache Fehleranalyse, durch Dokumentieren des Programmstatus zu unterschiedlichen Laufzeiten, ermöglichen.

Coding Conventions
Im Programmcode sollen stets die Namenskonventionen der Programmiersprache JAVA umgesetzt werden. Außerdem sollen if-Statements nicht in lediglich einer einzelnen Zeile stehen und statische Variablen, so weit möglich, vermieden werden.

Attraktivität
Das Erscheinungsbild des Spiels soll in ansprechender Art und Weise dargestellt werden und einen hohen Wiedererkennungswert bieten. Auch die Eingaben sollen für die Spielenden einfach umzusetzen und nachvollziehbar sein.

Detailgetreue
Das Spiel soll möglichst nahe der Brettspielvorlage im Spielablauf sein. Dafür sollen zunächst grundlegende Elemente und Regeln von “Die Siedler von Catan”, wie im MVP beschrieben, implementiert, und später fehlende oder weiterführende Regeln ergänzt werden

Qualitätsszenarien

1) Spieler*in baut ein Dorf
Die Spielerin hat die notwendigen Ressourcen um ein Dorf zu bauen und klickt auf einen geeigneten Platz, welcher den Voraussetzungen entsprechend den Regeln erfüllt. Das Spiel schickt die Anfrage mit dem gewünschten Platz und dem gewünschten Objekt an den Server welcher die Anfrage in kürzester Zeit verarbeitet und entsprechend reagiert.

2) Verbindungsaufbau zum Host
Der Spieler möchte einer bereits bestehenden Lobby beitreten, welche sich noch im Lobby-Screen befindet. Nach dem eingeben der IP-Adresse des Hosts stellt der Client schnellstmöglich eine Verbindung zum Server her.

3) Verbindungsabbruch eines Clients
Ein Client verliert, aufgrund eines unerwarteten Ausfalls seiner Internetverbindung, die Verbindung zum Server. Die anderen Clients in der Lobby sind davon nicht betroffen und können das Spiel ohne Einschränkungen fortsetzen. Ein Wiedereintritt des getrennten Clients soll zu einem späteren Zeitpunkt der Entwicklung, außerhalb des MVPs implementiert werden.

4) Spieler*in ist nicht am Zug aber versucht zu handeln
Eine Spielerin, welche aktuell nicht an der Reihe ist, versucht mit der Bank zu handeln, um fehlende Rohstoffe zu erwerben. Die Anfrage des Clients wird zwar an den Server geschickt und von diesem verarbeitet, jedoch mit einer “Rejection” beantwortet - also abgelehnt.

5) Spieler*in gewinnt das Spiel
Eine Mitspielerin hat genügend Siegpunkte erhalten, dass sie das Spiel gewinnt. Am Ende ihres Zuges werden alle Mitspielenden benachrichtigt und das Spiel beendet. Daraufhin öffnet sich die Endszene. Von dort aus kann man entweder zurück zum Startscreen, um ein neues Spiel zu starten, oder das Spiel beenden.

Risiken und technische Schulden

Anfälligkeit des Servers gegenüber (D)DoS Angriffen oder Ähnlichem
Verbundenen Clients ist es möglich, beliebig viele Anfragen an den Server zu schicken. Dieser antwortet aktuell auf jede abgelehnte Anfrage mit einer Fehlermeldung. Dies birgt das Potential, durch zu viele Anfragen eine zu hohe Rechenlast auf dem Server zu verursachen.
Des Weiteren lauscht der Server permanent auf eingehende Verbindungen. Das macht ihn anfällig für SYN-Flooding oder ähnliche Angriffe, bei denen der Speicher der eingehenden Verbindungen aufgefüllt wird.
Diese Möglichkeiten sind uns bekannt, allerdings war es nicht Ziel des MVPs den Server vollständig gegen Angriffe abzusichern, weshalb wir uns entschieden, diese Möglichkeiten erst einmal außen vor zu lassen.

Empfangsbuffer des Server und Client
Es kann vorkommen, dass in schneller Abfolge Pakete ankommen. Dies geschieht vor allem auf Clientseite, da der Server immer wieder schnell hintereinander verschiedene Pakete zum Aktualisieren des GUI versendet. Kommen diese zu schnell an, kann es sein, dass ein anderes Paket noch in dem Empfangsbuffer steht, das nicht verarbeitet wurde. In diesem Fall, wird das nächste Paket einfach an das vorherige gehängt. Das hat zur Folge, dass der JSON Paket Serializer das Paket nicht in ein Packet Objekt serialisieren kann.

Fragmentierung der Pakete
Ein weiteres Problem war, dass die Pakete, welche verschickt werden, eine bestimmte Größe nicht überschreiten dürfen. Wird diese Größe überschritten, wird das Paket zerteilt und in Fragmenten verschickt. Da dieses Problem nur bei dem Paket, welches das vom Server erzeugte Spielbrett an die Spieler übermittelt, auftrat, konnte das Problem umgangen werden, indem dieses eine Paket sehr gekürzt wurde.
Sollte das Projekt weitergeführt werden und noch größere Pakete auftreten, wäre es sinnvoll, das Problem mit der Fragmentierung nachhaltig zu lösen. Zudem könnte die Codierung der Pakete von JSON zu einer binären Form geändert werden, um die Effizienz der Übertragung zu erhöhen.

Testing

Die Tests konnten erst am Ende des Projekts implementiert werden, da das Testframework und das UI-Framework “Text Mesh Pro” nicht miteinander harmonierten. Auch nach langer Recherche konnte keine Lösung gefunden werden. Erst durch Ausprobieren mit sogenannten “Assemblies” konnte der Fehler behoben werden.
Die Assemblies dienen in Unity zur Strukturierung und Abgrenzung von Code in Projekten. Um das Testing Framework nutzen zu können, bedarf es einer Assembly, welche auf die zu testenden Klassen verweist, um diese in den Tests instanziieren zu können. Sobald jedoch die Assembly erstellt wird, ist das UI-Framework nicht für die Klassen in der Assembly sichtbar.
Um das Problem zu beheben, muss für die Klassen im UI-Framework ebenfalls eine Assembly erstellt werden, die es vorher nicht brauchte. Diese Assembly muss dann jeweils mit den Assemblies des Testframeworks und des restlichen Projekts verknüpft werden.
Generell ist die Kombination von Testing und Unity teilweise etwas gewöhnungsbedürftig, insbesondere, wenn es um das Testen von UI-Elementen, bzw. damit verknüpften MonoBehavior-Klassen geht. Dafür müssten alle Objekte nochmals als gesonderte Testobjekte erzeugt werden. Wir entschieden uns dafür, beim Testen den Hauptfokus auf die serverseitige Gamelogic und die Kommunikation zwischen Server und Client zu legen.

Lessons learned

Neue Wege

Wir hatten uns zu Beginn für gegen Java und eine Umsetzung mit Unity und C# entschieden. Keiner von uns hatte zu diesem Zeitpunkt Erfahrung in diesem Bereich gesammelt. Da zu Beginn die Motivation groß war, die neue Umgebung aus zu testen wurde munter ausprobiert und drauf los entwickelt. Die genauen Zuständigkeiten waren zu diesem Zeitpunkt noch nicht definiert gewesen. Daraus haben wir gelernt, dass man vor allem bei neuen Technologien und Sprachen enorm darauf achten sollte, wer für welchen Teil der Entwicklung zuständig ist, damit nichts doppelt gemacht wird und jeder etwas zu tun hat. Auch das gewohnte Erstellen von UML Diagrammen erwies sich als kompliziert. Ohne das nötige Grundwissen über Unity war es uns quasi unmöglich präzise Diagramme anzufertigen. Hier haben wir gelernt, dass ein grobes Konzept zu Beginn ausreichend ist und man die fehlenden Details mit fortschreitendem Wissen gut ergänzen kann.

Wissen und Wissen verbreiten

Nicht jeder hat den gleichen Wissensstand oder die gleichen Anforderungen. Das ist mit der Zeit aufgefallen. Wer schon einmal in einer Firma mit git gearbeitet hat, hat andere Vorstellungen von Benennungen als andere. Hier muss man unbedingt vor Beginn des Projektes Regeln aufstellen, sich auf Guidelines einigen und auch Coding Konventionen festlegen. Wenn sich alle über die Regeln einig sind und jeder bescheid weiß, findet man sich auch im ganzen Projekt zurecht.

Pipelines Fluch und Segen

wenn man das Konzept einer Pipeline verstanden hat und die Möglichkeiten erkennt, will man natürlich unbedingt auch eine haben. Dies hat sich nach der ersten Euphorie leider schnell gelegt. Das Einrichten einer Pipeline die ein Unity Projekt bauen soll, hat sich als eine riesen Aufgabe herausgestellt. Das Wissen über die Konfiguration sollte hierbei nicht bei nur einer Person liegen. Wenn einmal etwas schief geht, gibt es nur einen Ansprechpartner. Für ein Projekt dieser Größe ist das vielleicht noch tragbar, aber für zukünftige größere Projekte ein absolutes no go.

Zu hoher Datenverkehr

Je weniger man übers Internet sendet, in Sachen Dateigröße, desto besser. Anstelle eines Objekt zu JSON Vorgehens, sollte man eine Methode einfügen, welche die Werte möglichst kompakt übertragen, um unnötige Strings herauszufiltern und Informationen eventuell zu verschlüsseln.
Pakete binär statt als JSON zu übertragen, wäre auch ein effizienterer Ansatz.

Beispiel:
CLIENT: Incoming Data:
{"type":11,"playerColor":0,"myPlayerID":5,"currentPlayerID":0,"previousPlayerID":2,"isReady":false}
VS
CLIENT: Incoming Data:
{11,0,5,0,0,0,}