Universität Hildesheim
Institut für Informatik

Die Entwicklung eines rundenbasierten Strategiespiels, das es KI-Spielern ermöglicht gegeneinander anzutreten und KIs gegeneinander zu testen

Bachelor im Studiengang Bachelor of Science (B. Sc.) in Informationsmanagement und Informationstechnologie (IMIT)
Sommersemester 2020

vorgelegt von

Tjark Harjes

Matr.-Nr.: 301249
4. Semester IMIT
E-Mail: harjes@uni-hildesheim.de

Hildesheim, den **07.06.2020**

Erstgutachter: Prof. Dr. Klaus-Dieter Althoff
Zweitgutachter: Pascal Reuss, M.Sc.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcode-Verzeichnis

1 Einleitung

1.1 Aufbau der Arbeit

2 Grundlagen

Da diese Ausarbeitung von der Entwicklung eines Strategiespiels für computergesteuerte Spieler handelt, sollen in diesem Kapitel die Grundlagen eines solchen Spiels erläutert werden. Dieses Kapitel ist insbesondere für das Verständnis wichtig. Der Entwurf und die darauffolgende Implementation des Spiels basieren auf diesen Grundlagen. Außerdem wird die Evaluation des Spiels noch einmal Bezüge zu dem hier Geschriebenen nehmen. Aus diesem Grund sind die folgenden Abschnitte für den Leser von besonderer Relevanz.

Die Entwicklung von computergesteuerten Spielern reicht inzwischen schon über 20 Jahre zurück. So schrieben [Dav99] schon 1999 von ihren Entwicklungen, die beispielsweise in das bekannte Strategiespiel *Civilization: Call To Power* Einzug fanden. Das ist von daher beeindruckend, dass der Sieg von Deep Blue über Garri Kasparow zu diesem Zeitpunkt erst 3 Jahre zurück lag. Seitdem hat sich sowohl für Echtzeitspiel aber auch für rundenbasierte viel getan. Bis heute ist das einprogrammierte “Cheaten” der Computergegner ein Problem, was von [SRGC07] näher beschrieben wurde, wobei ein solches Problem im Kontext dieser Arbeit nicht auftreten sollte, da das Ziel nicht ist, eine möglichst spannende KI zu erstellen. Die Arbeit von [SRGC07] dennoch interessant, da sich speziell mit dem Case-Based-Reasoning auseinander gesetzt haben. Sie präsentieren zudem eine Idee für die Interaktion von Spiel und Spieler mittels einer API. Die API sei in der Lage eine Verbindung zwischen der KI und dem Spiel zu schaffen, sodass die KI Zugriff auf alle Informationen hat und gleichzeitig Anweisungen erteilen kann. Diese API ist vonnöten, da eine Konvertierung, der auf dem Bildschirm erscheinenden Informationen erfolgen muss, um sie für einen Computer wieder verarbeitbar zu machen. Diese Architektur lässt sich wie in 2.1 zu sehen ist, oberflächlich darstellen.

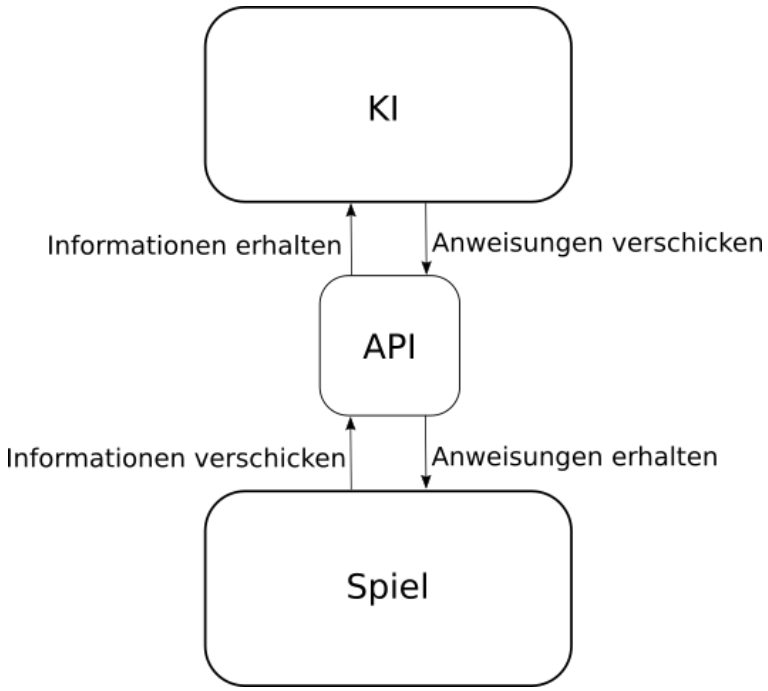

Abbildung 2.1: Architektur KI-Spiel (In Anlehnung an [SRGC07])

2.1 zeigt drei Komponenten über insgesamt vier Pfeile miteinander verbunden. Die oberste Komponente repräsentiert die KI. Sie steht nicht etwa für den Spieler, der im Spiel zu

sehen ist, sondern nur die dahinter stehende Logik. So wie ein menschlicher Spieler auch nicht die Figur im Spiel ist, sondern diese Figur nur eine Repräsentation seiner selbst darstellt. Auf gleiche Weise funktioniert dieses Prinzip auch mit einem computergesteuertem Spieler. Er erhält Informationen und erteilt Anweisungen über eine geeignete Schnittstelle (hier: die API) und wird im Spiel durch eine Spielfigur ohne eine andere Verkörperung verbildlicht.

Die unterste Komponente stellt das Spiel dar. Zwischen der KI- und der Spiel-Komponente befindet sich die API-Komponente. Diese ist die erwähnte Schnittstelle und ermöglicht eine Kommunikation zwischen dem Spiel und den KI-Spielern. Das Spiel kann über die API Informationen an einen Spieler senden und dem Spieler wird so auch das erhalten von Informationen ermöglicht. Äquivalent dazu funktioniert das Senden und Empfangen von Anweisungen. Nur das diesmal der Spieler die Anweisungen absendet und das Spiel diese empfängt und ggf. verarbeitet. So wird dem Spieler die Steuerung seiner Repräsentation im Spiel ermöglicht, egal ob es sich um eine Figur oder ein Reich oder ähnliches handelt.

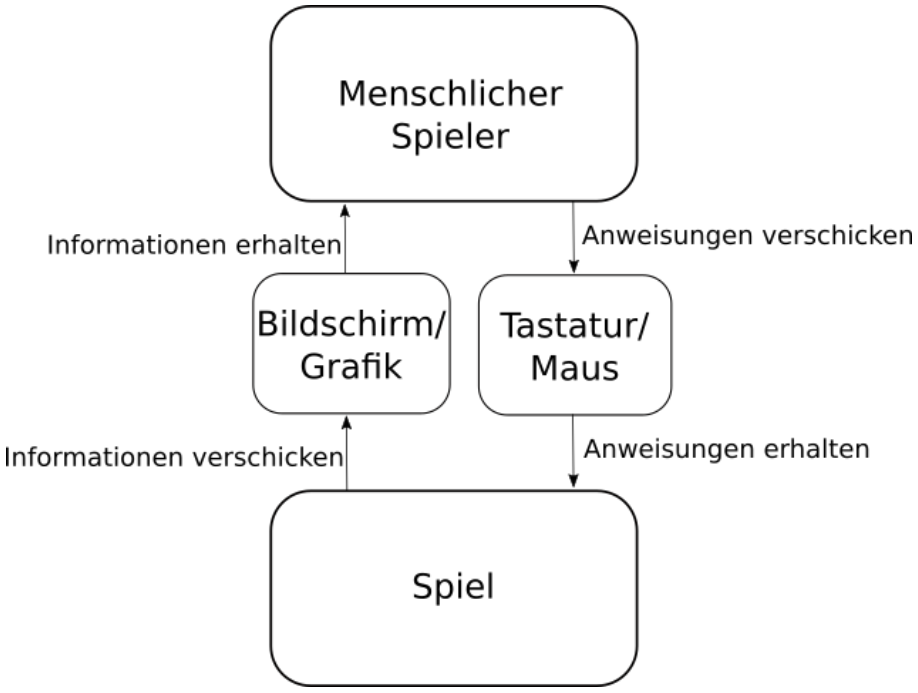

Abbildung 2.2: Architektur menschlicher Spieler

Abbildung 2.2 zeigt eine analoge Darstellung der Komponenten, aber für menschliche Spieler. Diesmal stellt die oberste Komponente nicht etwa den KI-Spieler dar, sondern einen menschlichen. Und auch die mittlere Komponente ist eine andere. Zuvor wurde die Mitte durch die API veranschaulicht. Jetzt sind stattdessen zwei Komponenten zu sehen: *Bildschirm/Grafik* und einmal *Tastatur/Maus*. Nur die untere Komponente *Spiel* ändert sich nicht. Diese Abbildung soll die Parallelen zwischen der Verwendung eines KI-Spielers und der eines menschlichen Spielers weiter verdeutlichen. Denn ob ein Mensch oder eine Maschine spielt ist im Prinzip das Gleiche. Wichtig ist nur die Umsetzung der Schnittstelle und da meist ein Mensch das Spiel spielt, erscheint es uns als intuitiver. Doch die Aufnahme von Informationen über einen Bildschirm und das Erteilen von Anweisungen mit Maus und Tastatur ist nichts anderes als das Verwenden einer Schnittstelle zum Steuern der eigenen Repräsentation im Spiel. Für die KI wird das Gleiche erreicht, wenn sie Infor-

mationen und Anweisungen über eine API erhält bzw. versendet.

Das beschriebene Prinzip gilt also nicht exklusiv für computergesteuerte Gegner. Anhand von 2.2 wird deutlich, dass es sich viel allgemeiner um einen Standardweg der Kommunikation zwischen mit einem Programm und eine, nicht darin enthaltenen, Entität handelt.

2.1 Computergegner in rundenbasierten Strategiespielen

Nachdem im obigen Abschnitt allgemein auf die Interaktion zwischen Spielern und dem Spiel eingegangen wurde. Soll im Folgenden ein möglicher Aufbau und die Konzeption von Computergegnern beschrieben werden.

Ein KI-Spieler kann auf unterschiedliche Weisen konzipiert werden. So ist es möglich feste Strategien von Hand zu implementieren und diese den Spieler ausführen zu lassen. Des Weiteren können KIs durch Neuronale Netze und dem damit verbundenem Deep Learning realisiert werden [May07]. Außerdem sind, wie bereits zuvor erwähnt, KI-Spieler möglich, die mithilfe von Wissensbasierten Systemen ihre Strategien umsetzen [SRGC07]. Alle diese Varianten können gute Ergebnisse erzielen und ihre Einsatz hängt maßgeblich vom gewünschten Ziel ab. So würde eine von Hand programmierte KI dann zum Einsatz kommen, wenn nicht unbedingt eine die bestmögliche Performance gewünscht ist, aber stattdessen ein ganz bestimmtes Verhalten an den Tag gelegt werden soll. Ein Ziel, das durch beispielsweise Neuronale Netze nur schwer zu erreichen wäre. Wenn andererseits die stärkste mögliche KI als Ergebnis herauskommen soll, dann eignen sich Herangehensweisen, die die KI ihre eigenen Methoden finden lassen und durch Lernprozesse auf diesem Wege die besten Strategien ausmachen können. Etwas, das bei hart-kodierten Spielern wiederum äußert schwer, wenn überhaupt zu erreichen ist.

2.1.1 Der KI-Spieler als Wissensbasiertes System

Da das Produkt dieser Arbeit jedoch hauptsächlich für den Einsatz von KI-Spielern nach dem Prinzip der Wissensbasierten Systeme gebraucht werden soll, wird der Fokus der Erläuterungen auch auf diesen liegen. Die Abbildung 2.3 zweigt wie ein Wissensbasiertes System grundlegend aufgebaut ist. Es wird deutlich, dass eine Abgrenzung zwischen dem Interpreter und dem Wissen gezogen wird. Das ist einer der identifizierenden Punkte für ein solches System. Denn die Handlungen des Systems sind nicht nur vom Interpreter an sich abhängig sondern auch von der dahinter liegenden Wissensbasis.
Das Konzept des Systems lässt sich auch anhand der Abbildung erkennen. zunächst erfolgt eine Eingabe, die der Interpreter wahrnimmt. Anschließend greift dieser zur Verarbeitung der Eingabe auf die angeschlossene Wissensbasis zurück und versucht eine passende Lösung mithilfe dieser auf die Problemstellung zu finden. Schlussendlich wird die Lösung als Ausgabe zurückgeliefert. Bezogen auf eine KI im Kontext eines Videospiels, sind die Eingaben als Informationen, die das Spiel liefert zu verstehen. Man denke hierbei an die Abbildung 2.1 zurück, bei der die KI ihre Informationen über die API erhalten hat. Diese Informationen sind demnach die Eingabe für das System. In der gleichen Abbildung wurde auch gezeigt, dass es sich bei der Ausgabe der KI um Anweisungen handelt. Demnach gibt auch das Wissensbasierte System in diesem Fall Anweisungen als Ausgabe zurück. Also muss durch den Interpreter eine Art Konvertierung einer Information in eine Handlung erfolgen, die von der Repräsentation des Spielers im Spiel ausgeführt werden kann. Hierbei

ist dennoch zu beachten, dass nicht jede gesendete Information auch eine direkte Anweisung zur Folge haben muss. Die Informationen diesen vielmehr der Findung einer Strategie durch den Interpreter und der Interpreter kann Aktionen dieser gefundenen Strategie auch dann ausführen lassen, wenn gerade gar kein Input erfolgt ist.
In Bezug auf die Fallbasis ist wichtig zu wissen, dass diese sowohl Wissen über das Spiel und die möglichen Strategien enthält, aber auch Wissen über das eigene System.

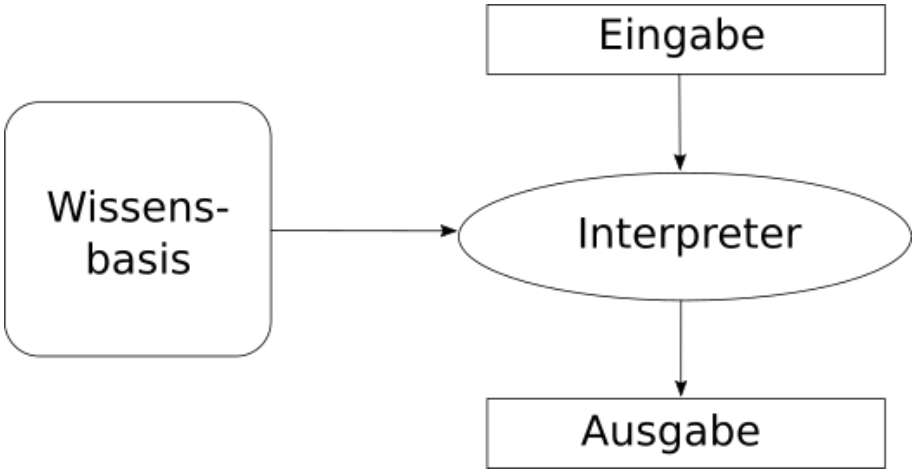

Abbildung 2.3: Wissensbasiertes System

Eine mögliche Umsetzung eines Wissensbasierten Systems ist in Form einer fallbasierten Variante (auch Case Based Reasoning genannt). Bei dieser Herangehensweise wird die Wissensbasis mithilfe einer Fallbasis realisiert. diese Fallbasis enthält, auf ein Videospiel bezogen, alle möglichen Zustände des Spiel, über die das System Bescheid weiß. hieraus ergibt sich auch, dass eine "neue" KI über geringeres Wissen verfügt als eine trainierte, weil sie keine Zeit hatte sich genügend Fälle anzueignen. Was ein Fall enthält muss definiert werden. Hiervon ist abhängig wie gut die KI nach dem Training spielen wird.

[WM09] haben für das Spiel *Wargus* eine eigene KI geschrieben, die mithilfe von Case Based Reasoning funktioniert. Ein Fall wurde bei ihnen durch mehrere Features definiert, die wiederum jeweils mehrere Eigenschaften abbildeten. So beschreibt jeder Fall den Forschungsstand des Gegners und den eigenen, die Anzahl an Kampfeinheiten (nach Typen aufgeteilt) und Arbeitern, sowie die Menge an Produktionsgebäuden und Eigenschaften der Karte. Ein einzelner Fall enthält umfassende Informationen über den gesamten Spielstand und das ist durchaus nötig. Die Strategien werden aufgrund von Ähnlichkeit mit schon bekannten Fällen gebildet.
Angenommen ein Fall würde auf die Eigenschaften der Karte verzichten und nur die übrigen miteinbeziehen. Zusätzlich würden in der Fallbasis bereits zwei Fälle existieren und nun wird ähnlichste dieser beiden zu einem neuen Fall gesucht. Sei nun auch noch angenommen, dass die Eigenschaften des neuen Fall außer die der Karte mit Fall 1 zu komplett übereinstimmen und die von Fall 2 leicht davon abweichen, dafür hat Fall jedoch eine Karte, die sich nicht von der des neuen Falls unterscheidet und die von Fall 1 sieht komplett anders aus. Abhängig davon, ob die Karte mit in die Bewertung der Ähnlichkeit einfließt, könnten nun zwei unterschiedliche ähnlichste Fälle gefunden werden und damit würde eine verschieden gute Strategie gebildet wird. Hieraus lässt sich ableiten, dass zum Finden der bestmöglichen Strategie, die Fälle so genau wie möglich beschrieben werden sollten.
Dennoch muss laut [WM09] ein Fall auch generalisiert werden. Sie wählten als einzige Information über den Gegner den Fortschritt seiner Forschung und bezeichneten dieses

Symptom als wichtigsten Indikator für dessen Strategie zumindest im betrachteten Spiel. Interessanterweise konkurrieren beide Annahmen nicht miteinander, was auf den ersten Blick jedoch paradox wirken kann. Auf der einen Seite sollen Fälle so genau wie möglich die aktuelle Situation beschreiben, andererseits wird aber sich zur Erkennung des Stands des Gegners auf ein einzelnes Symptom beschränkt. Ein solches Vorgehen ist dann sinnvoll, wenn die anderen Symptome deutlich schwerer zu erfassen wären oder nur unzureichend verlässliche Informationen liefern würden. Die Autoren beschrieben diesen Zustand als *unverlässliche Informationsumgebung* [WM09, vgl.]. Sie treffen die Annahme, dass es besser ist, sich auf ein Symptom zu verlassen, welches mit relativ hoher Wahrscheinlichkeit genau bestimmt werden kann, als weitere möglicherweise falsche hinzuzuziehen. Konkret bezeichnen sie die Bestimmung der Anzahl an Arbeitern und Truppen des Gegners als unzuverlässig, weil diese sich bewegen und ihre Anzahl nicht genau bestimmt werden kann. Der Stand der Forschung hingegen sei genauer zu identifizieren.

Da zu jedem Fall einer Wissensbasis auch eine Lösung gehört, muss jedem Fall hier ein Handlungsvorschlag enthalten, der beschreibt, was in welcher Situation zu tun ist. [WM09] fügen jedem Fall hierzu eine Art Liste aus Aktionen hinzu, die die KI vornehmen soll, um zu reagieren. Ein fertiger Fall könnte für dieses Beispiel ungefähr so aussehen wie in Abbildung 2.4 zu sehen ist.

Situation:	
Eigener Forschungsstand	Schmid Festung Mühle
Gegner Forschungsstand	Schmid Mühle
Eigene Truppen	Axtwerfer: 5 Katapulte: 3 Orgs: 19
Eigene Arbeiter	Tagelöhner: 10
Produktions-gebäude	Barracken: 3
Karten-eigenschaften	Distanz zum Gegner: 3 Weg zum Gegner: (2,1),(2,2),(2,3),(3,4)
Lösung:	
Trainiere Truppen:	Axtkämpfer: 3 Katapulte: 1
Baue Prdduktions-gebäude	Barracken: 2

Abbildung 2.4: Beispiel Fall

Wie die Abbildung 2.4 kann ein Fall schnell relativ umfangreich werden, wenn die Symptome nicht wie hier beschränkt werden. Wie zuvor beschrieben wurde auf weitere Symptome, den Gegner betreffend, verzichtet. Das ist neben der Zuverlässigkeit auch noch für die Aussagekraft an sich hilfreich. Selbst wenn zwei Symptome zuverlässig erfasst werden könnten, kann es sein, dass die Erhebung eines dieser unterlassen werden kann, wenn sie im Grunde das Gleiche beschreiben. Eine solche Herangehensweise hilft dann, wenn die Fälle eben nicht zu umfangreich werden sollen. Für die Performance der KI kann dies sowohl Vorteile, aber auch Nachteile bedeuten. Einerseits kann ein solches System durch die Vermeidung von ähnlichen Symptomen die Übergewichtung dieser Bestandteile verhindern, da bei vielen korrelierenden Symptomen möglicherweise ein Fall gefunden wird, der wenig hilfreich ist. Andererseits werden dadurch auch Details vernachlässigt, die durchaus hilfreich für die Strategiefindung sein könnten. Insgesamt muss sehr genau betrachtet werden, welche Eigenschaften des Spiels in einem Fall abgebildet werden sollen und ob diese nötig sind bzw. einen Vorteil für KI bringen.

Da in dem herangezogenen Paper *Wargus* durch die KI gemeistert werden sollte und es sich hierbei um ein Echtzeitspiel handelt, können die Ergebnisse möglicherweise nur teilweise auf rundenbasierte Spiele übertragen werden. Bei Echtzeitspielen ändert sich die Situation in der sich die KI wiederfindet dauerhaft. Das ist für die Tätigung von Aktionen von besonderer Relevanz. Während die KI noch ausführt, kann sich das Spiel schon soweit verändert haben, dass neue Aktionen nötig werden. Das erhöht den Schwierigkeitsgrad für die KI deutlich. Theoretisch kann kontinuierlich ein neuer Fall als Input entstehen und die passende Strategie könnte verändert werden müssen. Das erhöht den Schwierigkeitsgrad zur Konzeptionierung einer solchen KI, da zusätzlich festgelegt werden muss, ob eine Strategie geändert, eingehalten oder gar abgebrochen sollte, wenn sich die Situation im Spiel verändert. Es könnte auch eine Integration dieser Komponente in die Fälle nachgedacht werden. Das erfolgt nach Ausführungen von [WM09] für Ihre KI nicht. Könnte aber Vorteile für Echtzeitstrategiespiele mit sich bringen.

Bei rundenbasierten Strategiespielen gibt es dieses Problem nicht in einem solchen Ausmaß. Da jede KI nur in einem bestimmten Zeitraum Aktionen tätigen darf, müssen auch nur dann Entscheidungen getroffen werden. Jedoch kann sich die Situation eines Spiels auch durch die eigenen Aktionen ändern und in dem betroffenen Zug müssen die weiteren Entscheidungen überdacht werden. Dieses Problem könnte dadurch umgangen werden, dass ein Zug zu Beginn durchgeplant wird. Das bedeutet über alle möglichen Tätigkeiten wird vor der ersten eigenen Aktion schon entschieden, wobei das auch die Performance beeinträchtigen könnte. Komplizierte wird das Problem jedoch, wenn Aktionen über mehrere Runden andauern können. In einem solchen Fall müssten auch bei einem rundenbasierten Spiel zuvor getroffene bzw. anhaltende Entscheidungen, deren Umsetzung noch nicht abgeschlossen ist, in die Findung einer neuen Entscheidung mit einfließen. Gegenüber den Echtzeitspielen bleibt jedoch immer noch der Vorteil, dass keine dauerhafte Neubewertung der Situation erfolgen müsste, da die Rundenbasiertheit vereinfachend auf das zeitliche Gefüge des Spielablaufs wirkt, zumindest aus Sicht der KI-Spieler.

Die obige Ausführung über fallbasierte KIs wirft die Frage auf, ob es Alternativen hierzu gibt, die auf Case Based Reasoning verzichten und stattdessen eine andere Grundlage für die Entscheidungsfindung verwenden. Theoretisch gibt es hierzu sogar gleich mehrere Möglichkeiten. Denn das fallbasierte Lernen stellt nur eine mögliche Art des Lernens für Wissensbasierte Systeme dar. Weitere wären das inkrementelle, das analogiebasierte, das

erklärungsbasierte und das Lernen durch Vergessen. Inwiefern diese sich für die Erstellung von KIs für Videospiele eigenen, bleibt allerdings fraglich und soll im Folgenden etwas näher erläutert werden.

Inkrementelles Lernen:
Vom Namen dieses Lernverfahrens lässt sich bereits auf seine Fähigkeit schließen: Im Grunde werden mit jedem weiteren Lernschritt die Ergebnisse vorangegangener Schritte verbessert. Dieses Verfahren könnte in Verbindung mit dem fallbasierten Schließen eingesetzt werden, um dieses zu verbessern. So könnte durch den Einsatz von inkrementellem Lernen die vorgeschlagene Lösung eines Falles variiert werden. Hierdurch würden bei jeder Anwendung bessere, schlechtere oder kaum veränderte Ergebnisse entstehen. Bei ersterem kann die Veränderung beibehalten oder sogar noch verstärkt werden, bei zweiterem negiert oder umgekehrt und bei letzterem müsste in eine beliebige Richtung verstärkt. Dies könnte dazu führen, dass die KI in ihrer Performance teilweise stark schwankt, aber langfristig eine Verbesserung festzustellen ist. Weiterhin ließe sich inkrementelles Lernen auch als alleiniges Verfahren einsetzen. Dann müsste die Festhaltung des Wissen, aber in einer anderen Form als in Fällen erfolgen. Hier wären Regeln denkbar, die individuell bei eintretenden Ereignisse eine Antwort des Systems erzeugen und durch inkrementelles Lernen abgeändert werden könnten, um auch diese langfristig zu verbessern. Dies scheint auch eine echte Alternative zum Fallbasierten Schließen dazustellen, da wi auch beim analogiebasierten Lernen eine individuellere Betrachtung der Ereignisse stattfinden kann.
Im Grunde ist der Aufbau einer KI mithilfe eines Neuronales Netzes auch nichts anderes als die Anwendung von inkrementellem Lernen. Bei Neuronalen Netzen wird nämlich durch Anpassung der Gewichte zwischen Neuronen eine Veränderung des Verhaltens erreicht nach jedem Lernschritt (Backpropagation). Das wäre dann im eigentlichen Sinne kein WBS mehr, doch könnte ein neuronales Netz die Wissenbasis ersetzen und über einen Interpreter, der wiederum wie bisher Informationen vom Spiel empfängt, angesteuert werden. Er wäre dann für die Vorbereitung der Eingabe für das Neuronale Netz und für die Generierung einer sinnhaften Anweisung aus der Ausgabe heraus zuständig.

Analogiebasiertes Lernen:
Beim analogiebasierten Lernen werden Schlüsse gezogen. Es wird versucht von einem bereits bekannten Objekt auf ein neues Objekt Wissen zu übertragen und so die richtigen analog angewendeten Schlüsse als gewonnenes Wissen explizit zu speichern (vgl. VL Althoff Wissenbasierte Systeme). Auf ein Videospiel übertragen könnte hierbei die Ähnlichkeit von Situationen genutzt werden, um allgemeine Schlüsse ableiten zu können, die für all diese Arten von Situationen funktionieren. Verglichen mit dem fallbasierten Schließen klingt dieser Ansatz relativ ähnlich. Schließlich wird bei beiden Verfahren mit der Ähnlichkeit der aktuellen Situationen zu vorherigen gearbeitet. Der große Unterschied besteht darin, dass beim analogiebasierten Lernen der Fokus auf dem expliziten Speichern dieses allgemeinen Wissens liegt. Der Vorteil hierin kann in der detaillierteren Anwendung bestehen. So muss nicht der ganze Fall genutzt werden, sondern es kann können ähnliche Aspekte der Situation genutzt werden, um einzelne Strategien zu übertragen und anzuwenden. Fallbasiertes Schließen hingegen würde den Vorteil der besseren Ausnahmebehandlung mit sich bringen, da jede Ausnahmesituation, so in die Fallbasis Einzug findet und nicht auf eine andere Weise repräsentiert werden würde.

Erklärungsbasiertes Lernen:
Erklärungsbasiertes Lernen benötigt zu seiner Anwendung drei Voraussetzungen. Einen

Zielbegriff, ein positives Beispiel dafür und einem Operationalisierungskriterium, dem der Zielbegriff nicht genügt (zu Anfang). Der Lerneffekt besteht darin, das Beispiel so zu generalisieren, dass es operational ist. An sich ist dieses Lernverfahren dann gut geeignet, wenn nachvollziehbares Wissen generiert werden soll in Form von Erklärungen, wie der Name vermuten lässt. Zum erstellen einer KI für ein rundenbasiertes Strategiespiel scheint diese Herangehensweise jedoch wenig geeignet zu sein. Sicher wäre die Konzeption einer solchen KI möglich, aber dieses Verfahren scheint der KI an sich keinen Vorteil zu bringen. Eine Art Implementierung dieser Variante wäre das WBS mit Beispielen von Siegen zu versorgen und es hieraus zu operationalisieren.

Lernen durch Vergessen:
Das Lernen durch Vergessen eignet sich zur Verbesserung von Systemen, indem Wissensinhalte abstrahiert werden und so vereinfacht dargestellt werden. Teilweise werden gespeicherte Informationen auch gelöscht. Auf diese Weise können Fehlinformationen oder zu starke Gewichtungen aufgelöst werden. Bezogen auf Strategiespiele könnte dieses Lernen vielmehr als Ergänzung und nicht als Ersatz eines anderes Verfahren. Angewandt aufs analogiebasierte Lernen könnten einige Schlüsse gelöscht werden, wenn diese keinen großen Vorteil bringen. Normalerweise könnte auch beim fallbasierten Schließen durch das Vergessen oder Abstrahieren von Regeln ein positiver Effekt erzeugt werden. Im Fall von Wargus scheint, dies jedoch nicht nötig zu sein, da der gesamte Fall bereits bekannt ist und nicht weitere Symptomeausprägungen erschlossen werden müssen. Aber je nach KI kann der Einsatz dieses Verfahrens durchaus in Erwägung gezogen werden.

Insgesamt lässt sich festhalten, dass es durchaus sinnhafte Alternativen zum fallbasierten Lernen für Wissensbasierte KI-Spieler gibt. Einige sind besser geeignet als andere und andere würden sogar eine gute Ergänzung darstellen. So eignet sich das erlärungsbasierte Lernen eher weniger für den Einsatz im beschriebenen Kontext und auch für den Anwendungsfall dieser Arbeit (rundenbasierte Strategiespiele) scheint dieses Lernverfahren keine wirklichen Vorteile zu bringen. Besser geeignet ist schon das Lernen durch Vergessen. Dieses könnte für unseren Anwendungsfall die beschriebenen Vorteile mit sich bringen, aber nur als Erweiterung in Kombination mit anderen Lernverfahren. Neben der eher geringen Einsetzbarkeit in Verbindung mit fallbasiertem Schließen (auf [WM09]), kann der Vorteil mit anderen Wissensrepräsentationsarten besser genutzt werden. So wäre der kombinierte Einsatz von inkrementellen Lernen und dem Lernen durch Vergessen möglicherweise eine gute Kombination, die am späteren Produkt dieser Arbeit gegen andere Lernverfahren getestet werden könnte. Das inkrementelle Lernen an sich scheint auch schon eine echte Alternative zum fallbasiertem Lernen darzustellen, wenn auch mit einer anderen Form der Wissensrepräsentation oder sogar als Neuronales Netz. Vor allem die individuelle Anwendbarkeit des Wissen scheint hier Möglichkeiten zu bieten, die Fallbasiertheit an sich nicht gewährleisten kann. Inwiefern sich die Behandlung von Ausnahmen negativ auf inkrementelles lernen auswirkt, bleibt ohne weitere Tests nicht genau identifizierbar. Analogiebasiertes Lernen bildet eine weitere Alternative, die ebenso mit individuellen Antworten aufwarten kann. Hier ist die Form der Speicherung von Wissen ein interessanter Faktor, der Auswirkungen auf die Performance einer späteren KI haben kann.
Abschließend lässt sich zu den verschiedenen Lernstrategien sagen, dass, obwohl [WM09] nur die Strategie des fallbasierten Lernens angewandt haben, einige weitere Verfahren zur Verfügung stehen und gegeneinander getestet werden können, um die bestmögliche KI zu finden. in diesem Kapitel wurden zusätzlich viele verschiedene Vermutungen angestellt, wie sich die unterschiedlichen Strategien auswirken und welche am Ende die besten Ergebnisse bringt. Wirklich festzustellen, ist dies aber nur über ausgiebige Versuche, bei denen die

KI-Spieler mit ihren unterschiedlichen Verfahren gegeneinander antreten.

2.1.2 Der KI-Spieler als Neuronales Netz

Eine Alternative zur, als Wissensbasiertes System konstruierten, KI bildet die Umsetzung durch ein Neuronales Netz. Diese wurde bereits im Abschnitt 2.1.1 kurz angesprochen, soll jedoch hier noch tiefgehender beschrieben werden. Abbildung 2.5 zeigt wie ein neuronales Netzwerk grundlegend aufgebaut ist. Im Detail zeigt die Abbildung ein, aus drei Schichten aufgebautes, Neuronales Netz mit insgesamt fünf Knoten. Zwei Inputknoten in der ersten Schicht, zwei Outputknoten im Hiddenlayer und einem Outputknoten in der Outputschicht. Es wird hier auf die Verwendung des Bias verzichtet.

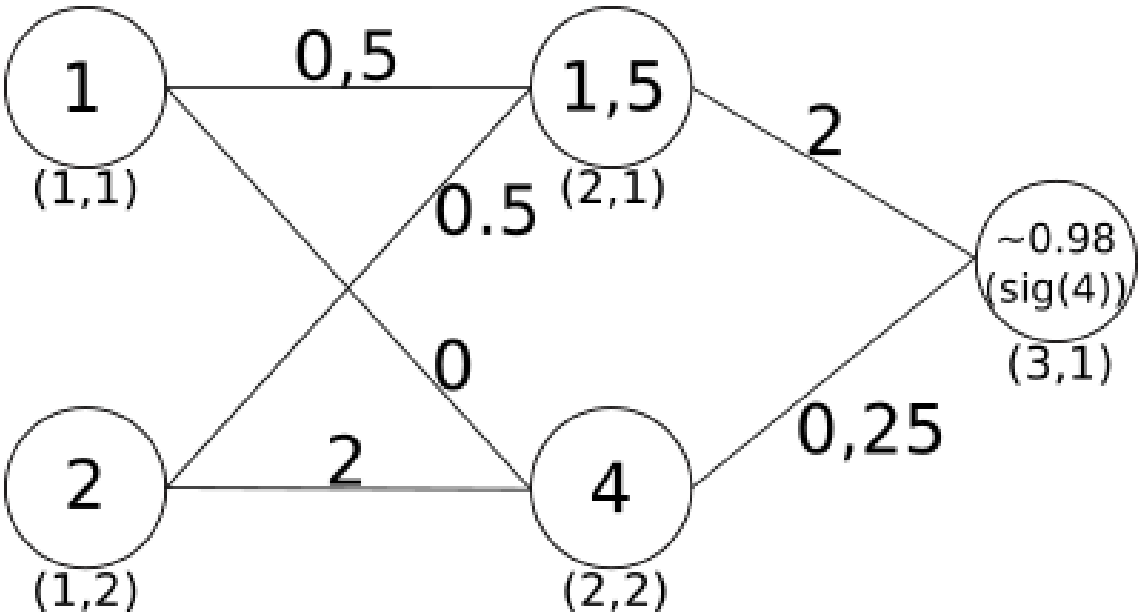

Abbildung 2.5: Beispiel neuronales Netz

Die Schichten sind über gewichtete Verbindungen mit der darauffolgenden Schicht verbunden, wobei jeder Knoten eine Verbindung mit jedem Knoten der nächsten Schicht aufweist. Wenn das Gewicht einer Verbindung auf null gesetzt wird, so wird diese nicht weiter beachtet und fungiert, als würde sie nicht existieren. Der Wert eines Knoten in den folgenden Schichten wird aus allen Knoten der vorherigen Schichten berechnet. So wird der Wert von Knoten (2,1) beispielsweise durch Wert((1,1)) * Gewicht der Verbindung((1,1),(2,1)) + Wert((1,2)) * Gewicht der Verbindung((1,2),(2,2)). Durchgerechnet für das gesamte Beispiel ergibt sich folgender Ablauf:
Wert((1,1)) = 1
Wert((2,1)) = 2
Wert((2,1)) = 1 * 0,5 + 2 * 0,5 = **1,5**
Wert((2,2)) = 1 * 0 + 2 * 2 = **4**
Wert((3,1)) = sig(1,5 * 2 + 4 * 0.25) = sig(4) = **0,98**
Der letzte Knoten stellt einen Sonderfall dar, weil hier eine Aktivierungsfunktion genutzt wird. Solche Funktionen dienen der weiteren Verarbeitung von Inputs. So sorgt die hier verwendete Sigmoidfunktion dafür, dass die Ausgabewerte zwischen 0 und 1 liegen.

Die Sigmoidfunktion stellt nicht die andere mögliche Aktivierungsfunktion dar. Generell sind alle stetigen Funktionen einsetzbar, aber in der Praxis wird sich auf einige wenige beschränkt:

1. **Sigmoidfunktion:** $\frac{1}{1+exp(-ax)}$, mit a $\in \mathbb{R}$
2. **Lineare Funktion:** $\mathrm{linear(x)} = \mathrm{m}^*\mathrm{x} + \mathrm{b}$, mit m $\in \mathbb{R}$ der Steigung und b $\in \mathbb{R}$ dem Bias.
3. **Stückweise lineare Funktion:** $\mathrm{l(x)} = \begin{cases} 1 & \text{falls } x >= \frac{1}{2} \\ a + \frac{1}{2} & \text{falls } -\frac{1}{2} < x < \frac{1}{2} \\ 0 & \text{falls } x <= -\frac{1}{2} \end{cases}$
4. **ReLU:** $\mathrm{ReLU(x)} = \max(0, \mathrm{x})$
5. **Hard limit:** $\mathrm{hard(x)} = \begin{cases} 1 & \text{falls } x >= 0 \\ 0 & \text{falls } x < 0 \end{cases}$

Wobei die Entscheidung über den Einsatz der zu nutzenden Funktion auch vom Ziel abhängig ist, das mit dem neuronalen Netz erreicht werden soll. Bezogen auf Videospiele könnte beispielsweise der Outputlayer eines Netzes, welches über die Anzahl an zu erstellenden Truppen entscheidet, durch eine lineare Aktivierungsfunktion realisiert werden. Ob ein bestimmter taktischer Schritt ausgeführt wird, lässt dagegen eher durch beispielsweise eine Sigmoidfunktion abbilden (1 und 0 als ja und nein).

Bei dem Beispiel aus 2.5 handelt es sich nur um ein rudimentäres Neuronales Netz. Real eingesetzte sind um einiges umfangreicher. Sie unterscheiden sich insbesondere in der Menge an verwendeten Knoten pro Schicht und auch in der Anzahl der Schichten an sich. So wäre es möglich, dass weiterhin eine Input- und eine Outputschicht Teil des Netzes sind, aber 5 hintereinanderliegende Hiddenschichten liegen. Außerdem könnten die Schichten statt zwei Knoten, 20, 30 oder über eine andere beliebige Anzahl verfügen, wobei nicht jede Schicht gleich viele Knoten haben muss. Wie viele Schichten und Knoten in einem Neuronalen Netz genutzt werden, hängt meist von der Komplexität der zu bewältigenden Aufgabe ab. Eine sehr komplexe Tätigkeit braucht häufig ein größeres Netz, um die nötigen Einstellungen vornehmen zu können. Hinzu sollte jedoch erwähnt werden, dass ein größeres Netz auch mehr Training benötigt, um die gewünschte Aufgabe zu erlernen. Dies ist dadurch begründet, dass mehr Gewichte trainiert werden müssen. Dies kann also zum einen ein Vorteil sein, ist aber zum anderen ein Nachteil, wenn nicht genügend Trainingsdaten zur Verfügung stehen.

Das trainieren eines Neuronalen Netzes ist ähnlich zum Finden von Koeffizienten einer Funktion deren Grad gleichzeitig bestimmte werden muss. Im Grunde ist ein Neuronales Netz nichts anderes als eine Funktion. Es erfolgt eine Eingabe und hierzu wird eine eindeutige Ausgabe geliefert. Der Vorteil den uns Neuronale Netze bringen, liegt darin begründet, dass sie ein einfaches Training mittels Backpropagation ermöglichen. Bei Backpropagation handelt es sich um ein Verfahren, dass einen errechneten Fehler zurück durch das Netz führt und damit anteilig jedes Gewicht anpasst, um den Fehler für die Zukunft zu minimieren.

Im Detail funktioniert Backpropagation so, dass zu einem Datensatz das richtige Ergebnis bereits bekannt sein muss. Der Datensatz wird nun als Input in das Neuronale Netz eingegeben und der Output wird mit dem tatsächlichen Ergebnis verglichen. Je nachdem mit wie vielen Datensätzen schon trainiert wurde, wird die Abweichung größer oder kleiner sein. In den meisten Fällen wird es aber eine Abweichung geben. Bei dieser Abweichung handelt es sich um den Fehler, also die Differenz zwischen dem gewünschten Ergebnis und dem errechneten. Es muss nicht immer nur mit der Differenz gearbeitet werden. Es kann auch mit dem quadrierten Fehler oder weiteren Varianten trainiert werden, ganz vom Design des Neuronalen Netzes und dem Lernverfahren abhängig. Für diese Erklärung soll jedoch der einfache Fehler genügen. Dieser wird dann anschließend mittels dem sogenannten Backwardpass durch das Netz geführt und auf diese Weise auf die Gewichte der Verbindungen verteilt, um sie anpassen zu können. Beim Backwardpass fungiert der Output sozusagen als Input.

Am besten lässt sich das Verfahren am Beispiel des gezeigten Netzes verstehen. Hierzu sei angenommen, dass wir statt 0,98 als Ergebnis 0,2 erreichen wollten. Als Lernrate wählen wir $\alpha = 1$, ein Parameter der gleich noch wichtig wird. Die Formel zur Aktualisierung eines Gewichtes wie folgt lautet nämlich:

$W_{i,j} = \alpha * \delta_i * a_j$, mit $\alpha \in \mathbb{R}$ der Lernrate (hier 1), δ_i des Neuron an dem das Gewicht anliegt und a_j dem Input des Neuron von dem das Gewicht kommt.

Die einzige Variable, die hier jetzt zu berechnen ist, ist das δ. Die anderen sind schon gegeben. Dieses wird dann wie folgt für die letzte Schicht berechnet:

$\delta_i = f'(in_i) * (a_i(y) - a_i(y\_hat))$, mit f' der Ableitung der Aktivierungsfunktion, in_i dem Input des Knoten i, $a_i(y)$ dem Sollwert und $a_i(y\_hat)$ dem Istwert.

Umgesetzt für die Outputschicht des Beipielnetzes, ergibt sich folgendes:

3 Problemstellung

Rundenbasierte Strategiespiele sind seit geraumer Zeit ein fundamentaler Bestandteil der Videospiel-Industrie. Sie existieren in vielen Variationen und der Erfolg des Spielers ist maßgeblich von der gewählten Strategie abhängig. Seit den 90er Jahren wurden auch vermehrt computergesteuerte Spieler als Gegner des Menschen eingesetzt. Hierfür lässt sich zum Beispiel die Entwicklung von KI-Spielern im Rundenbasierten Strategiespiel Civilization anführen [Dav99].

Was damals noch eine Herausforderung für die Entwickler dargestellt hat und Teil der aktiven Forschung war, ist heute Alltag geworden und beinahe jedes Strategiespiel, ob rundenbasiert oder kontinuierlich, verfügt über computergesteuerte Spieler. Aufgrund der vermehrten Anwendung solcher KI-Spieler in der Praxis, sollte die effektive Entwicklung dieser Einzug in die Lehre erhalten. Hierbei ist es wichtig, das Wissen möglichst Praxisnah und effizient den Studenten zu vermitteln. Hierfür bietet es sich an, den Studenten eine Möglichkeit zur Verfügung zu stellen, die Ihnen hilft, eigens entwickelte KIs zu testen und gegeneinander antreten zu lassen.

Aus dieser Idee ergeben sich mehrere Anforderungen, deren Erfüllung eine solche Software gewährleisten und gleichzeitig die Problemstellung verdeutlichen:

1. Die Software muss mindestens zwei KI-Spieler zulassen.
2. Die Software muss ein möglichst faires Spiel bereitstellen.
3. Die Software muss eine einfache Integration neuer KIs ermöglichen und einen schnellen Austausch dieser gewährleisten.
4. Die Software muss rundenbasiertes Spielen ermöglichen.
5. Die Software muss strenge Regeln formulieren, sodass KIs kein unerwünschten Verhalten lernen, wobei sie fehlerhafte Regeln ausnutzen würden.
6. Die Software muss einen Sieger feststellen.

Aus der obigen Konkretisierung der Problemstellung lässt sich erkennen, dass die genannte Software sowohl das Spiel an sich umfassen, als auch eine Schnittstelle für die entwickelten KI-Spieler bereitstellen muss. Insgesamt lässt sich die Problemstellung also so zusammenfassen, dass eine Software entwickelt werden muss, die aus zwei Komponenten besteht (rundenbasiertes Strategiespiel + KI-Schnittstelle) und gleichzeitig die genannten Anforderungen erfüllt.

4 Entwurf

Um die in 3 erläuterte Problemstellung und die damit einhergehenden Anforderungen zu erfüllen, wird in diesem Kapitel beschrieben, wie die Problemstellung gelöst und umgesetzt werden kann, wobei zunächst grobe Spezifikationen beschrieben und anschließend tiefgehender Details erläutert werden.

Die beiden Komponenten der Software werden im folgenden als Spiel und Schnittstelle bezeichnet.
Das Spiel Als Grundlage für das Spiel soll das rundenbasierte Brettspiel *Siedler von Catan* genutzt werden. Zu Implementierung des Spiels wird Unity in Verbindung mit der Programmiersprache C# genutzt. Das Endergebnis soll möglichst nah am echten Spiel sein, was eine exakte Umsetzung der Spielregeln erfordert. Auf diese Weise kann jedoch gleichzeitig die Einhaltung mehrerer Anforderungen sichergestellt werden. So werden durch das Umsetzen der Spielregeln bereits mindestens zwei Spieler garantiert (Anforderung eins), das Spiel ist möglichst fair (Anforderung 2), das Spiel ist aufgrund seiner Spielweise rundenbasiert (Anforderung 4) und unerwünschtes Verhalten wird auch weitestgehend ausgeschlossen (Anforderung 5). Wobei der letzte Punkt nicht mit vollkommener Sicherheit erfüllt wird. Das hängt vor allem mit dem Umstand zusammen, dass KI-Spieler sehr gut im Finden von Lücken in den Regeln sein können, was durch eine strenge Implementierung vermindert, jedoch nicht ausgeschlossen werden kann. Insgesamt soll das Spiel neben den in 3 beschriebenen Anforderungen auch noch die folgenden, speziell an das Spiel gerichtete, erfüllen:

1. Das Spiel muss nach den Regeln des Brettspiels ein Spielfeld generieren.
2. Das Spiel muss den Bau von Straßen erlauben.
3. Das Spiel muss den Bau von Siedlungen erlauben.
4. Das Spiel muss den Bau von Städten erlauben.
5. Das Spiel muss am Ende des Spiels einen Gewinner ermitteln.
6. Das Spiel muss jedem Spieler in jeder Runde einen Zug bereitstellen, der mindestens aus dem Rollen von zwei Würfeln und der anschließenden Verteilung von Rohstoffkarten, sowie mit dem optionalen Kaufen von Objekten einhergeht.
7. Das Spiel muss gewährleisten, dass zwischen Siedlungen bzw. Städten mind. 2 Straßen Platz sind.
8. Das Spiel muss Upgrades auf Städte erlauben.
9. Das Programm muss die „Längste Handelsstraße “berechnen.

Neben diesen Mindestanforderungen existieren noch die folgenden Soll bzw. Kann Anforderungen, die durch ihre Umsetzung die Komplexität des Spiels erhöhen und so eine ausgeklügeltere Strategie seitens der KI erfordern. Wobei das Ziel ist, alle Muss-Anforderungen zu implementieren und dann mit ausgewählten Soll- und Kann-Anforderungen zu ergänzen. Es soll jeweils mindestens eine Soll- und eine Kann-Anforderung implementiert werden.

1. Das Spiel soll die Funktion der Ereigniskarten integrieren.
2. Das Spiel soll das Sammeln und Ausspielen von Ereigniskarten ermöglichen.
3. Das Spiel soll ermöglichen, dass reale Spieler das Spiel spielen.

4. Das Spiel soll über einen Dieb verfügen, der einzelne Ressourcenfelder unbrauchbar macht, solange er dieses Feld besetzt und von einem Spieler immer dann versetzt werden kann, wenn er eine Sieben würfelt.

5. Das Spiel kann ermöglichen, dass mehr als zwei Spieler gegeneinander antreten.

6. Das Spiel kann ermöglichen, dass KIs und reale Spieler gegeneinander antreten.

7. Das Spiel kann Handel zwischen den Spielern ermöglichen.

8. Das Spiel kann aktuelle Gewinnwahrscheinlichkeiten ausgeben.

9. Das Spiel kann über initiale KIs verfügen.

Die Schnittstelle: Allgemein ist eine Schnittstelle zur Kommunikation der KI-Spieler mit dem Spiel nichts anderes, als wenn ein menschlicher Spieler per Maus und Tastatur das Spiel steuert bzw. mit seinen Augen und Ohren Informationen, die ausgegeben werden, aufnimmt. Eine Maschine die das Spiel spielt, verfügt nun mal nicht über die gleichen Sinne wie ein Mensch und daher muss ein Weg gefunden werden, wie die Maschine dennoch möglichst effizient Züge ausführen und Informationen erhalten kann.

In diesem Fall wird JSON für die Schnittstelle verwendet. Auf diese Weise können Informationen in einem von der Programmiersprache unabhängigen Format versendet und empfangen werden. Hierdurch können die KI-Spieler in einer beliebigen geeigneten Programmiersprache definiert werden, sofern sie zum Senden und Empfangen von Informationen dasselbe Format nutzen.
Neben der geforderten Plattformunabhängigkeit gibt es noch weitere Anforderungen, die erfüllt werden müssen, damit die Schnittstelle schlussendlich wie gewünscht funktioniert. Im Folgenden sind alle Anforderungen aufgelistet, die erfüllt werden müssen:

1. Die Schnittstelle muss ermöglichen, dass die KIs ohne großen Aufwand ausgetauscht werden können und die Verwendung der Schnittstelle hierfür kein großes Hindernis darstellt.

2. Die Schnittstelle muss ermöglichen, dass die KIs in unterschiedlichen Programmiersprachen geschrieben sein können, sofern sie das richtige Format zur Kommunikation verwenden.

3. Die Schnittstelle muss ermöglichen, dass zwei KIs statt realer Spieler gegeneinander antreten.

4. Die Schnittstelle muss gewährleisten, dass eine KI durch die Verwendung der Schnittstelle gegenüber einem realen Spieler nicht bevorteilt oder benachteiligt wird.

5. Die Schnittstelle muss Informationen vor der Einsicht anderer Spieler geschützt versenden (Kein Broadcast).

6. Die Schnittstelle muss jedem KI-Spieler die gleichen Informationen bereitstellen, auf die ein realer Spieler auch Zugriff hat.

Neben den Muss-Anforderungen gibt es auch für die Schnittstelle einige Anforderungen, die erfüllt werden sollen bzw können. Hierbei ist jedoch zu beachten, dass einige dieser Anforderungen auf die Soll-/Kann-Anforderungen für das Spiel zurückzuführen sind und für eine funktionierendes Feature implementiert werden müssen, sofern die korrespondierende Anforderung auch für das Spiel erfüllt wurde:

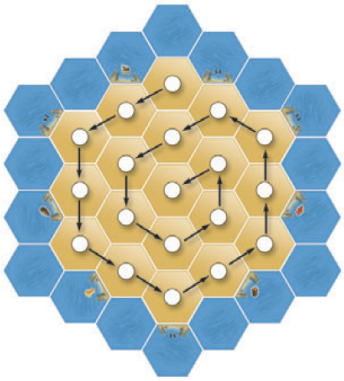

Abbildung 4.1: Verteilung der Zahlenchips auf dem Spielfeld

1. Die Schnittstelle soll die Informationen über Ereigniskarten an KI-Spieler übermitteln und Anweisungen über diese von KI-Spielern annehmen.

2. Die Schnittstelle soll ermöglichen, dass, wenn ein KI-Spieler eine Sieben würfelt, dieser die Funktion des Räubers zunutze machen kann.

3. Die Schnittstelle kann reale und computergesteuerte Spieler gleichzeitig ermöglichen.

4.1 Die Regeln und Anpassungen

In diesem Kapitel wurden bereits allgemeine Anforderungen an die Software erläutert. Dieser Abschnitt soll nun auf die Regeln *Siedler von Catan* betreffend eingehen. Vor allem wird jedoch erklärt, wie und ob einige dieser Regeln umgesetzt werden. Allgemein wird Bezug auf das Regelbuch von *Die Siedler von Catan Catan-Almanach* genommen, welches sich in voller Länge im Anhang befindet.

Der Spielaufbau:
Das Spielfeld besteht aus 19 Landschaftsfeldern, wobei es vier mal Wald, vier mal Weideland, vier mal Ackerland, drei mal Hügelland, drei mal Gebirge und einmal Wüste gibt und 18 Wasserfeldern, wovon 9 über einen Hafen zum Handeln verfügen. Die Landschaftsfelder werden genau so in der späteren Implementation verwendet, die der Wasserfelder ist davon abhängig, inwiefern die Soll-/Kann-Anforderungen umgesetzt werden. Außerdem ist zu beachten, dass der Zufallsfaktor auch in die programmierte Variante des Spiels Einzug erhalten soll, denn die 19 Landschaftsfelder werden zufällig angeordnet, sodass jedes neu begonnene Spiel höchstwahrscheinlich nicht genau gleich zum vorherigen ist.

Des Weiteren müssen die Zahlenchips auf die Landschaftsfelder verteilt werden. Jedes Mal wenn eine Zahl gewürfelt wird und ein Spieler eine Siedlung oder Stadt an einem Landschaftsfeld hat, auf dem ein Zahlenchip liegt dessen Aufschrift der Summe der Augen auf beiden Würfeln entspricht, erhält dieser Spieler den entsprechenden Rohstoff (Ein für jede Siedlung an diesem Feld, zwei für jede Stadt).
Die Zahlenchips werden nach einer bestimmten Reihenfolge verteilt: 5, 2, 6, 3, 8, 10, 9, 12, 11, 4, 8, 10, 9, 4, 5, 6, 3, 11. Der erste Zahlenchip wird auf ein Eckfeld gelegt und die weiteren Chips werden in der beschriebenen Reihenfolge gegen den Uhrzeigersinn ringsum verteilt. Auffällig ist hierbei, dass statt 19 Chips nur 18 existieren, das beruht darauf, dass die Wüste frei bliebt (sie liefert keine Ressourcen). Die Reihenfolge der Verteilung der Zahlenchips wird in Abb. 4.1 verdeutlicht.

Der Spielbeginn:
Die ersten beiden Züge jedes Spielers laufen anders als die restlichen Züge ab. In seinem ersten Zug darf jeder Spieler eine Siedlung und eine Straße platzieren, ohne dafür Ressourcen ausgeben zu müssen. Gleiches gilt für den zweiten Spielzug, wobei nun die Spieler in entgegengesetzter Reihenfolge ihre Gebäude platzieren dürfen. Nach dem zweiten Spielzug jedes Spielers erhält dieser entsprechend Ressourcen für seine zweite Siedlung (jedes angrenzende Landschaftsfeld, bringt ihm eine Ressource). In diesen ersten beiden Spielzügen kann weder gewürfelt, noch können Ereigniskarten erworben werden.

Gewöhnliche Spielzüge:
Die Reihenfolge in der die Spieler ihre Züge machen, entspricht nun wieder der der ersten Baurunde. Die gewöhnlichen Spielzüge laufen ab jetzt bis zum Ende des Spiels auf gleiche Weise ab. Zu Beginn eines Zuges würfelt der Spieler und alle erhalten, wie bereits beschrieben, Ressourcen. Anschließend kann dieser Spieler entweder bauen oder eventuell handeln. Abschließend beendet der aktive Spieler seinen Zug und der nächste ist dran. Im Detail läuft ein gewöhnlicher Spielzug wie folgt ab:

1. Würfeln: Jeder Zug eines Spielers muss immer damit beginnen, dass er mit zwei Würfeln würfelt. Anhand der Summe der Augenzahl auf beiden, werden anschließend die Ressourcen verteilt. Wenn ein Zahlenchip der summierten Augenzahl entspricht, erhält ein Spieler mit einer Siedlung oder einer Stadt an diesem Feld, für jede Siedlung eine passende Ressource und für jede Stadt zwei.
2. Bauen: Nachdem gewürfelt wurde, kann gebaut werden. Zum Bau zur Verfügung stehen Straßen, Siedlungen, Städte und Entwicklungen, die je nach verfügbaren Ressourcen gekauft werden können.
3. (Handeln:) Handeln kann entweder zum Abschluss des Spielzuges erfolgen oder noch vor dem Bauen. Da es sich hierbei jedoch um eine Soll-Anforderung handelt, wird der Begriff hier in Klammern angeführt. Ein Spieler kann entweder an Häfen in den angegebenen Verhältnissen handeln, mit der Bank 4:1 oder mit einem anderen Spieler in einem beliebigen Verhältnis.

Ende des Spiels:
Das Spiel endet, wenn einer der Spieler 10 Siegpunkte erreicht hat. Siegpunkte werden für unterschiedliche Errungenschaften im Spiel vergeben. Zum Beispiel für das Errichten von Siedlungen und Städten, aber auch für die *Längste Handelsstraße* und *Größte Rittermacht*. Für die Implementation kann die Vergabe der Siegpunkte angepasst werden, je nachdem inwiefern die Strategiefindung der KI-Spieler beeinflusst werden soll. Interessant ist hierbei zu sehen, ob bei verschiedenen Siegbedingungen unterschiedliche KIs einen Vorteil haben. Es ist gut möglich, dass eine KI, die das Bauen von Siedlungen favorisiert, deutlich mehr Spiele gewinnt, wenn die Verteilung der Siegpunkte hierfür erhöht wird. Andererseits könnte eine KI, die Wert auf längere Handelsstraßen und mehr Ereigniskarten, bei entsprechender Gewichtung vermehrt siegt.

Baukosten:
Es wurde bereits beschrieben, dass Teil eines gewöhnlichen Spielzugs das Bauen sein kann. Zum Bauen können alle fünf verfügbaren Ressourcen verwendet werden: Lehm, Holz, Schaf, Getreide, Stein. Die Kosten für die einzelnen Gebäude setzen sich wie folgt zusammen:

1. Straße: 1 Lehm, 1 Holz
2. Siedlung: 1 Lehm, 1 Holz, 1 Schaf, 1 Getreide

3. Stadt: 2 Getreide, 2 Stein
4. Entwicklung: 1 Getreide, 1 Schaf, 1 Stein

Der Räuber:
Die Implementation des Räubers ist Teil der Kann-Anforderungen. Sollte er jedoch implementiert werden, so ist zwei Varianten vorstellbar. Entweder der Räuber wird im vollen Umfang implementiert, was bedeutet, dass ein Spieler, der eine Sieben würfelt, den Räuber auf ein Feld setzen darf, das dadurch keine Ressourcen mehr liefert. Außerdem kann der Spieler nun einen gegnerischen Spieler auswählen, der eine Siedlung oder Stadt an diesem Landschaftsfeld hat und ihm eine zufällige Ressource stehlen. Alternativ kann auch nur der erste Teil umgesetzt werden, indem der Räuber zwar Felder blockiert, aber keinen Diebstahl ermöglicht.

Häfen:
In der Sektion *Gewöhnliche Spielzüge* wird bereits das Handeln an Häfen erwähnt. Auch die Häfen sind Teil der Soll-Anforderungen und werden nicht zwingend Teil des Spiels. Sollten sie jedoch implementiert werden, sind hierbei auch verschiedene Stufen der Implementation wie beim Räuber denkbar. Auf jeden Fall gilt, dass ein Spieler nur an einem Hafen handeln kann, wenn er eine Siedlung oder Stadt im Einzugsbereich dieses Hafens besitzt.
Im Brettspiel gibt es mehrere Arten von Häfen. Einige dieser sind an Rohstoffe gebunden und können im Verhältnis 2:1 gehandelt werden. Das bedeutet, ein Handel an einem Holz Hafen würde 2 Einheiten einer anderen Ressource kosten, damit der handelnde Spieler ein Holz erhalten würde. Neben diesen gebundenen Häfen existieren auch ungebundene. Diese bieten ein Verhältnis 3:1 und es können drei Einheiten eines Rohstoffs für eine Einheit eines beliebigen Rohstoffs eingetauscht werden. Als Abstufungen für die Implementation ist eine Umsetzung genau wie beschrieben denkbar. Alternativ könnten aber auch alle Häfen zu ungebundenen Häfen erklärt werden oder es gibt gebundene Häfen, jedoch verfügen alle über das gleiche Eintauschverhältnis.
Betrachtet man den Handel an Häfen im gesamten Kontext des Spiels, lässt sich feststellen, dass die Soll-Anforderung *Handel* zumindest in Teilen umgesetzt werden sollte. Hierbei ist wahrscheinlich der Handel mit der Bank wichtigsten, dann der Handel an den Häfen und schließlich der Handel zwischen Spielern. Eventuell könnte der Handel zwischen Spielern auch über Bank- und Hafenhandel priorisiert werden, da dieser einen interessanten strategischen Aspekt darstellt.
Allgemein sollte jedoch mindestens der Bankhandel für die Spieler zur Verfügung stehen, da es sonst zu Konstellationen kommen kann, in denen ein oder mehrere Spieler nicht vorankommen können. Dieser Zustand würde entstehen, wenn die Anfangssiedlungen ungeschickt platziert würden und kein Zugang zu den für Straßen und Siedlungen nötigen Rohstoffen besteht. Für solch einen Spieler ist der einzige Ausweg der Handel, da es ansonsten keine realistische Chance für ihn gibt, das Spiel noch für sich zu entscheiden.

4.2 Konzeption der Schnittstelle

Über die Schnittstelle wurden in diesem Kapitel bereits Aussagen und Anforderungen an diese gestellt. Doch wie sie realisiert wird, wurde noch nicht erläutert. Denn um die Anforderungen umsetzen, muss die Schnittstelle in einer bestimmten Weise umgesetzt werden. Betrachtet man nochmals die gestellten Anforderungen, so lassen sich die ersten beiden schon dadurch lösen, dass JSON als Kommunikationsmedium genutzt wird. Anforderung 3 hängt eng mit einer korrekten Umsetzung zusammen, was zwar wichtig ist, jedoch das Konzept an sich nicht beeinflusst. Aus Anforderung 4 hingegen lassen sich deutlich mehr

Aspekte extrahieren. Diese besagt, dass eine KI einem realen Spieler gegenüber keine Vorteile oder Nachteile haben darf, allein dadurch, dass sie eine KI ist. Dieser Zusammenhang kann auch als Fairness aufgefasst werden. Konkret bedeutet das für die Umsetzung, dass eine KI nur die gleichen Optionen zu den jeweiligen Zeitpunkten im Spiel. Hieraus geht hervor, dass das Timing einer Rolle für die Anfragen und deren Verarbeitung spielt. Wenn eine KI beispielsweise den Bau einer Siedlung anfragt, während sie nicht an der Reihe ist, muss der Bau einer solchen Siedlung abgelehnt werden. Da das Spiel zu jeder Zeit weiß, wer an der Reihe ist, kann festgelegt werden, dass nur die Anfragen eines Spielers verarbeitet werden, der auch gerade einen Spielzug ausführt. Jedoch spielt das Timing in den Spielzügen selbst eine Rolle, da ein Spielzug immer mit dem Rollen der Würfel beginnen muss. Am besten ließe sich dieses Problem lösen, wenn Anfragen nur in einer bestimmten Reihenfolge erlaubt werden. So würde zunächst gewürfelt und erst anschließend könne gehandelt oder gebaut werden und erneutes würfeln würde verhindert. Beendet die KI jedoch ihren Spielzug verliert sie die Berechtigung Anfragen jeglicher Art zu stellen wieder. Zudem sind in den ersten beiden Zügen jedes Spielers auch andere Dinge erlaubt, als in den restlichen (siehe 4.1 Der Spielbeginn), was auch Einfluss auf das Annehmen einiger Anfragen hat. Optimal ließe sich dieses Problem lösen, wenn im Spiel selbst eine Verzahnung der KIs und menschlichen Spieler erfolgt. Damit ist gemeint, dass die gleichen Bedingungen für KIs genutzt werden, die auch einem realen Spieler Funktionen freischalten oder sperren.
Ein weiterer wichtiger Aspekt, den diese Anforderung aufgreift ist der, Umgang mit Informationen. Denn eine KI muss zu jedem Zeitpunkt im Spiel, auf die gleichen Informationen Zugriff haben, auf die auch ein menschlicher Spieler zugreifen kann. Wichtig ist hierbei, dass die Schnittstelle nicht dauerhaft die Informationen senden muss, sondern die Reaktion auf Anfragen reicht. Dieses Prinzip ist analog zur Interaktion mit einem realen Spieler. Das Spiel hält die Informationen auf der Spielfläche aktuell, zwingt den Spieler jedoch nicht diese auch zu jedem Zeitpunkt wahrzunehmen. Er schaut sie sich bei Bedarf an. Genau so sollte auch mit einem KI-Spieler verfahren werden. Wenn die KI Informationen anfragt, auf die auch ein realer Spieler Zugriff hätte, werden diese übermittelt, aber sie werden nicht dauerhaft gesendet. Eine solch triviale Überlegung könnte dazu führen, dass der Zeitpunkt des Anfragens von Informationen zu einem strategischen Element der KI wird, jedoch nur wenn die Anzahl der Anfragen begrenzt wird, was hier nicht der Fall sein wird. Denn ein realer Spieler ist auch in der Lage ist, Informationen beliebig oft wahrzunehmen.

Anforderung 5 geht in die gleiche Richtung wie schon die eben beschriebene, wobei hier noch ein extra Augenmerk auf das Schützen der versendeten Informationen gelegt wird. Ähnlich zum realen Spieler, der seine Ressourcen vor den Gegnern geheim halten kann, muss es auch einer KI ermöglicht werden, die Menge an Ressourcen und den Inhalt seiner Anfragen vor den Gegnern zu verschleiern. Lediglich das, was auf dem Spielfeld vonstatten geht, erlaubt den Gegnern einen Einblick in die Strategie eines Spielers, aber nicht das mitlesen von gesendeten Anfragen oder das Einsehen von Ressourcen. Daher ist es wichtig die Antwort des Spiels als Einzelnachricht zu realisieren und nicht als Broadcast allen mitzuteilen. Es klingt möglicherweise trivial, soll hier aber dennoch erwähnt werden, weil es einen fundamentalen Beitrag zur Fairness liefert.
Aus den Ausführungen Anforderung vier und fünf betreffend geht auch die Einhaltung der sechsten Anforderung, realen und KI-Spielern jeweils die gleichen Informationen bereitzustellen, hervor.

Durch die Einhaltung der beschriebenen Anforderungen seitens des Spiels, also quasi serverseitig, wird eine hohe Gestaltungsfreiheit für die KIs gelassen. Dadurch dass die KIs

entscheiden können, wann sie Informationen über Ressourcen abfragen, müssen sie ihre Strategie nicht an den Zeitpunkt des Erhalts der Information anpassen. Doch gibt es auch einige Anweisungen die das Spiel an die KI-Spieler aus eigener Initiative senden muss ohne auf eine Anfrage zu warten. Zum Beispiel die Information darüber, ob ein Spieler gerade an der Reihe ist mit seinem Zug oder nicht. Hier würde eine Abfrage durch jeden Spieler keinen Sinn ergeben, weil diese dauerhaft erfolgen müssen. Einfacher ist es für den Ablauf des Spiels, wenn das Spiel nach jedem Spielerwechsel eine Nachricht an die Spieler sendet. Des Weiteren müssen plötzliche Ereignisse sofort an die Spieler übermittelt werden. Das beste Beispiel hierfür ist das aktiv werden des des Räubers. Wenn der Räuber versetzt wird und eine Ressourcenkarte einem Spieler entwendet wird, dann muss dies unmittelbar zumindest den betroffenen Spielern mitgeteilt werden, da dies möglicherweise wichtig für die Strategiebildung ist. Allgemein könnte auch eine Art doppelte Absicherung eingeführt werden, bei denen Änderungen an den Beständen an Ressourcen eines Spielers direkt übermittelt wird, aber zusätzlich Abfragen durch die KI erfolgen können, um auf jeden Fall die aktuellen Zahlen vor dem Bau von Siedlungen, Straßen, etc. zu haben.

Neben den hier offensichtlichen Anfragen und Antworten, gibt es noch weitere fundamentale Informationen, die jede KI haben muss. Die fundamentalste wäre hier der Aufbau des Spielfelds. Jede KI muss Wissen, welches Feld an welcher Stelle liegt, damit sie eine Strategie bilden kann. Da die Initialisierung des Spielfelds jedoch nur zu Beginn erfolgt und danach gleich bleibt, sollte eine Art Initialisierungsnachricht an die KI-Spieler gesendet werden, sobald das Spiel fertig initialisiert ist, um das Finden einer anfänglichen Strategie zu ermöglichen. Um all das zu veranschaulichen folgt eine Tabelle aller Anfragen und Antworten inklusive Zeitpunkt, Absender und Empfänger.

Nachricht	Absender	Empfänger	Zeitpunkt	Aktion/Antwort
Aufbau Spielfeld	Spiel	KI-Spieler	Nach Initialisierung	Empfangs- bestätigung
Siedlungsbaumenü öffnen/ schließen	Spieler	Spiel	Im eigenen Zug nach Würfeln	Menü öffnen, schließen oder ablehnen
Bau Siedlung	Spieler	Spiel	Im eigenen Zug bei geöffneten Baumenü	Bau Siedlung oder ablehnen
Stadtbaumenü öffnen/ schließen	Spieler	Spiel	Im eigenen Zug nach Würfeln	Menü öffnen, schließen oder ablehnen
Bau Stadt	Spieler	Spiel	Im eigenen Zug bei geöffneten Baumenü	Bau oder ablehnen
Straßenbaumenü öffnen/ schließen	Spieler	Spiel	Im eigenen Zug nach Würfeln	Menü öffnen, schließen oder ablehnen
Bau Straße	Spieler	Spiel	Im eigenen Zug bei geöffneten Baumenü Bau oder ablehnen	Bau Straße oder ablehnen
Kauf Ereigniskarte	Spieler	Spiel	Im eigenen Zug nach Würfeln	Ereigniskarte
Einsatz Ereigniskarte	Spieler	Spiel	Im eigenen Zug nach Würfeln	Effekt
Verschiebung Räuber	Spiel	Spieler	Bei gewürfelter Sieben oder Ereigniskarte	Spiel öffnet Verschiebunsmenü für Spieler

Diebstahl Ressource	Spiel	Spieler	Nach Verschiebung des Räubers	Spiel öffnet Diebstahlmenü
Spielende	Spiel	Spieler	Wenn ein Spieler die Mindestanzahl der nötigen Siegespunkte erreicht	Spiel verkündet Sieger als Broadcast
Aktualisierung Ressourcen	Spiel	Spieler	nach jeder Änderung der Ressourcen eines Spielers	Spiel versendet die neuen Mengen der Ressourcen.
Zugende	Spiel	Spieler	Zum Ende eines Zuges nach dem Würfeln	Broadcast: Aktualisierung aktueller Spieler
Würfeln	Spieler	Spiel	Zu Beginn jedes Zuges	Summe der Augen
Ressourcenabfrage	Spieler	Spiel	Bei Bedarf	Spiel antwortet mit Aktualisierung Ressourcen
Handelsanfrage	Spieler	Spieler	Im Spielzug nach dem Würfeln	Schnittstelle ermöglicht eine Antwort durch andere Spieler ja/nein

Die hier gezeigten Nachrichten zeigen detailliert, wie umfangreich die Kommunikation zwischen Spiel, Schnittstelle und computergesteuertem Spieler werden kann. Denn es wird jede Tätigkeit abgebildet, die auch einem menschlichem Spieler zur Verfügung stehen würde, um den Grundsatz der Fairness zu wahren.

5 Implementation

Dieses Kapitel soll sich mit der Umsetzung, der im Entwurf beschriebenen Software befassen, inklusive des Spiels, der Schnittstelle und anfänglichen KI-Spielern. In der Reihenfolge der Beschreibung soll ein roter Faden erhalten bleiben. Aus diesem Grund wird chronologisch vorgegangen und zunächst eine grundlegende Version des Spiels beschrieben, dann die Umsetzung der Schnittstelle, anschließend die Implementation grundlegender KIs. Erst daraufhin wird sich auf die Weiterentwicklung des Spiels und der anderen Bestandteile fokussiert.
Nachdem die Implementation erläutert wurde, werden Anpassungen beschrieben, die zuvor nicht so geplant wurden, aber während der Umsetzung Einzug ins Programm erhielten bzw. weggelassen wurden. Schließlich werden die Ergebnisse erklärt.

5.1 Verwendete Technologien

Zu Beginn von Kapitel 4 wird bereits angesprochen, dass für die Erstellung der Software Unity mit C# und JSON für die Schnittstelle verwendet werden sollen. In diesem Abschnitt soll noch einmal genauer auf die verwendeten Technologien eingegangen werden, sowie erläutert, wieso sich für die Verwendung dieser entschieden wurde.

Unity ist eine Spieleentwicklungsengine, die eine schnelle und einfache Entwicklung relativ komplexer Spieler ermöglicht. Zur Wahl standen entweder die Unity-Engine oder die Unreal-Engine. Im Endeffekt fiel die Wahl jedoch relativ schnell auf Unity, obwohl beide Optionen für dieses Projekt infrage kommen. Unity zeichnet sich vor allem durch den sehr umfangreichen Asset-Store aus, der durch den einfachen Import von vielerlei kostenlosen Grafiken, Animationen, Sounds und sogar Funktionalitäten, Entwicklung beschleunigen kann. Außerdem ist Unity sehr benutzerfreundlich, was zwar lange ein Manko von Unreal war, jedoch inzwischen verbesser wurde.
Auch Unreal bietet einige Vorteile gegenüber Unity, die jedoch für die Zwecke dieser Arbeit keine großen Vorteile bieten. Beispielsweise ist Unreal Unity grafisch überlegen und ermöglicht blueprinting, was den Programmieraufwand in einigen Fällen deutlich reduzieren kann.
Für diese Arbeit wurde sich dennoch für die Verwendung von Unity entschieden, weil der Fokus hauptsächlich auf der Logik des Spiels liegt und nicht auf den grafischen Feinheiten.

Mit der Verwendung von Unity wurden auch die möglichen Programmiersprachen stark eingeschränkt. Bis zur Version 2017.1 war es möglich neben C# auch in JavaScript und Boo zu programmieren. Nach dieser Version wurde dem Programmierer diese Möglichkeit jedoch genommen, weshalb C# seitdem die einzig verwendbare Sprache ist. Aus diesem Grund gab es bei der Wahl der Programmiersprache keinen Spielraum.
C# ist eine plattformunabhängige, objektorientierte Programmiersprache, die starke Ähnlichkeiten zu Java und C++ aufweist. Neben dem objektorientierten Paradigma ist jedoch auch funktionale oder imperative Programmierung möglich.

Die Schnittstelle soll durch JSON (JavaScript Object Notation) realisiert werden, womit gemeint ist, dass JSON das gewählte Format der Informationsübertragung zwischen Spiel und KI-Spieler ist. Am besten lässt sich das Format an einem Beispiel veranschaulichen:

KI-Anfrage:

```
{
  “spieler”: “1”
```

```
  “aktion”: “Wuerfeln”
}
```

Spiel-Antwort:

```
{
  “augenzahl”: 8
  “ressourcen”: [
    {
      “name”: “lehm”
      “anzahl”: 2
    },
    {
      “name”: “holz”
      “anzahl”: 1
    },
    {
      “name”: “schafe”
      “anzahl”: 0
    },
    {
      “name”: “getreide”
      “anzahl”: 3
    },
    {
      “name”: “stein”
      “anzahl”: 2
    }
  ]
}
```

Das Beispiel zeigt mehrere Aspekte des JSON-Formats. Zu sehen sind zwei Dateien, einerseits die gesendete Anfrage durch die KI und andererseits die Antwort des Spiels. Aus dem Beispiel geht hervor, dass innerhalb von geschweiften Klammern Variablen definiert werden, die einen Wert enthalten. Auffällig ist hierbei insbesondere, dass nicht typisiert wird, sondern ohne Auskunft über den Variablentyp verschickt wird. Eine Zeile besteht aus dem Namen der Variable in Anführungszeichen, gefolgt von einem Doppelpunkt und dann dem zugewiesenen Wert. Zeichenketten werden wie der Name in Anführungszeichen gesetzt. Wohingegen Zahlen alleine stehen. Gut zu sehen ist das in der Zeile Spiele der Anfragedatei im Vergleich mit der Zeile “augenzahl” der Antwortdatei. Im ersten Fall handelt es sich um einen Namen, im zweiten um eine Anzahl.
Weiterhin lassen sich Listen per JSON-Format verschicken. So enthält die zweite Datei eine aktualisierte Liste der Ressourcen eines Spielers und schickt ihm diese als Teil der Antwort zu. Listen werden mit eckigen Klammern geöffnet und geschlossen und innerhalb werden einzelne Objekte, durch Kommata getrennt und von geschweiften Klammern umschlossen, definiert.
Ob in diesem Fall die Verwendung einer Liste nötig gewesen werden, ist fraglich, doch dieses Beispiel dient, wie erwähnt, der Veranschaulichung und muss nicht in gleicher Weise später implementiert werden. Weiterhin ist es zu diskutieren, ob das Spiel überhaupt mit den Ressourcen eines Spielers auf das Würfeln antworten muss oder ob hierfür eine weitere Anfrage erfolgen sollte.

5.2 Die grundlegende Umsetzung des Spiels

In 5.1 wurde auf die Verwendung von Unity für die Umsetzung des Spiels eingegangen und die Gründe hierfür. Einer ist, dass mit verhältnismäßig geringem Aufwand eine ansprechende Nutzeroberfläche gestaltet werden kann.

Abbildung 5.1: Nutzeroberfläche zu Beginn des Spiels aus Sicht von Spieler 1

5.1 zeigt eine solch gestaltete Oberfläche. Hierbei fallen mehrere Aspekte ins Auge. Zunächst einmal das Spielfeld an sich. Dieses besteht wie das Original aus insgesamt 19 sechseckigen Einzelteilen. Die Verteilung erfolgt über entsprechend der Verfügbarkeit der Karten des Brettspiels (4 Wald, 4 Getreide, 3 Ziegel, 3 Stein, 3 Schaf, 1 Wüste). Auffällig ist, dass das Wüstenfeld als einziges ohne Nummer bleibt, das liegt daran, dass es keine Ressourcen liefert, wenn ein Spieler eine Siedlung an diesem Feld gründet.
Links ist eine Art Pyramide, die eine Siedlung darstellen soll, Dementsprechend befindet sich im unteren rechten Bereich des Bildes das Abbild einer Straße. Neben den Objekten in der Umgebung sind im Sichtfeld des Nutzers ebenso zwei ausgegraute Buttons zu sehen (Würfeln und Zug beenden) und eine Anzeige, die die verfügbaren Ressourcen des aktuellen Spielers anzeigen.

5.2.1 Die Generierung des Spielfelds

```
1  public class MapGeneratorScript : MonoBehaviour
2  {
3
4      void Start()
5      {
6
7          List<Material> resources = new List<Material>()
8          {
9              wheat, wheat, wheat, wheat,
10             wood, wood, wood, wood,
11             sheep, sheep, sheep, sheep,
```

```
12              brick, brick, brick,
13              rock, rock, rock,
14              sand
15          };
16
17          List<GameObject> hexTiles = new List<GameObject>()
18          {
19              hex1, hex2, hex3, hex4, hex5, hex6, hex7, hex8, hex9, hex10,
                    hex11, hex12,
20              hex13, hex14, hex15, hex16, hex17, hex18, hex19
21          };
22
23          List<Material> numbers = new List<Material>()
24          {
25              num5, num2, num6, num3, num8, num10, num9, num12, num11, num4,
                    num8, num10, num9, num4, num5, num6, num3, num11
26          };
27
28          List<int> ints = new List<int>()
29          {
30              5, 2, 6, 3, 8, 10, 9, 12, 11, 4, 8, 10, 9, 4, 5, 6, 3, 11
31          };
32
33          List<int> positions = new List<int>()
34          {
35              7, 3, 0, 1, 2, 6, 11, 15, 18, 17, 16, 12, 8, 4, 5, 10, 14, 13, 9
36          };
37
38          List<GameObject> tiles = new List<GameObject>();
39
40          CreateMap(resources, numbers, tiles, positions, hexTiles, ints);
41      }
```

Listing 5.1: Spielfeld-Generator Startmethode

Listing 5.1 und 5.2 zeigen wie die Erstellung des beschriebenen Spielfelds zustande kommt. In der gezeigten Startmethode werden mehrere Listen deklariert und initialisiert. Die Liste *resources* enthält Materialien, jeweils in der Anzahl, in der sie später auf dem Spielfeld vorkommen dürfen. Neben dieser Liste existieren weitere. Eine speichert alle Sechsecke, sodass diese später angesteuert werden können. Eine andere speichert die Materialien der Nummern auf den Sechsecken, dann noch zwei Listen, die Ganzzahlen speichern. Zum Ende der Startmethode wird die Liste *tiles* ohne Inhalt initialisiert, bevor *CreateMap(List<Material> resources, List<Material> numbers, List<GameObject> tiles, List<int> positions, List<GameObject> hexTiles, List<int> ints)* aufgerufen wird.

```
1  void CreateMap(List<Material> resources, List<Material> numbers,
       List<GameObject> tiles, List<int> positions, List<GameObject> hexTiles,
       List<int> ints)
2      {
3          // Alle Einzelteile des Spielfelds werden durchlaufen
4          foreach (GameObject tile in hexTiles)
5          {
6
7              GameObject hexSurface = tile.transform.GetChild(0).gameObject;
                   //Oberflaeche des Sechsecks
```

```
8             GameObject number = tile.transform.GetChild(1).gameObject;
                  //Nummer auf dem Sechseck
9
10            // Eine zufällige Oberfläche wird dem Sechseck zugewiesen.
11            int randomResource = Random.Range(0, resources.Count);
12            hexSurface.GetComponent<Renderer>().sharedMaterial =
                  resources[randomResource];
13
14            // Die vergebene Oberfläche kann nicht erneut genutzt werden
15            resources.RemoveAt(randomResource);
16
17            // Sofern das Sechseck kein Wüstenfeld ist, erhält es eine Nummer
18            if (hexSurface.GetComponent<Renderer>().sharedMaterial != sand)
19            {
20                number.GetComponent<Renderer>().material = numbers[0];
21                tile.GetComponent<Tile>().number = ints[0];
22                ints.RemoveAt(0);
23                numbers.RemoveAt(0);
24            }
25            else
26            {
27                Destroy(number); //Wenn es sich um ein Sandfeld handelt, wird
                      der Nummer-Blueprint gelöscht
28            }
29        }
30    }
31 }
```

Listing 5.2: Spielfeld-Generator CreateMap-Methode

Listing 5.2 zeigt wie die Oberfläche bei für jedes neue Spiel generiert wird. Zu Beginn wird eine foreach-Schleife geöffnet, die einmal für jedes in hexTiles enthaltene Sechseck (GameObject) durchlaufen wird. Innerhalb dieser Schleife wird dann zunächst die Oberfläche des aktuellen Sechseck als Variable zwischengespeichert, sowie die Nummer auf dem Sechseck. Daraufhin wird zufällig eine Zahl generiert, die zwischen 0 (inklusive) und der Größe der Menge an verfügbaren Landschaften liegt (exklusive). Nun kann im nächsten Schritt das Material an der Stelle dieser zufälligen Zahl der Liste *resources* dem aktuellen Sechseck als Oberfläche zugeordnet werden, wie in Zeile zwölf zu sehen ist. Anschließend wird, um eine Doppelvergabe zu vermeiden, das vergebene Material aus der *resources* Liste entfernt.
In den darauffolgenden Zeilen, wird dafür gesorgt, dass nur Sechsecke, die ein anderes Landschaftsfeld als die Wüste repräsentieren, eine Zahl erhalten. Außerdem werden die Zahlen entsprechend einer bestimmten Reihenfolge vergeben. Diese Reihenfolge ist durch die Listen *numbers* und *ints* vorgegeben. *numbers* enthält die passenden Materialien und *ints* eine Repräsentation dieser Materialien als Ganzzahlen, um anderen Skripten später einen einfachen Zugriff zu ermöglichen.
Wenn eine Zahl, einem Sechseck zugewiesen wurde, dann wird diese diese aus dem beiden Listen entfernt. Sollte es sich bei dem betrachteten Sechseck um die Wüste gehandelt haben, so wird das Plättchen auf diesem Feld zerstört, damit es nicht ohne Zahl auf dem Feld zurückbleibt und die Kenntlichmachung der Wüste verringert.

Diese Zuweisung der Zahlen als Integerwerte, kann nur deshalb erfolgen, weil es eine Hilfsklasse *Tile* gibt, die Teil des Prefabs der Sechsecke ist und über ein Attribut verfügt, welches die Nummer speichert. Jedoch verfügt *Tile* neben der Speicherung eines Integer-

wertes über keinerlei Funktionalität und ist daher nur als Hilfsklasse zu bezeichnen.

Insgesamt garantiert die gewählte Umsetzung eine Variabilität im Aufbau des Spielfelds. Bei jedem Spielstart werden die Felder in einer anderen Reihenfolge verteilt und die gleiche Positionierung der Siedlungen in allen Spielen wird für die Spieler zu teilweise stark unterschiedlichen Ergebnissen führen. Dennoch bietet die Abspeicherung der Zahlen auf den Feldern und die Sicherung des Zugriffs auf die Oberfläche jedes Sechsecks die Möglichkeit, dass weiter Skripte die Informationen verwenden ohne einen Umweg zu gehen.

5.2.2 Der Spieler

Um das Spiel spielen zu können, muss eine Repräsentation des computergesteuerten oder menschlichen Spieler geben, der dessen Anweisungen ausführt. Hierzu wurde ein Skript *PlayerScript* erstellt. Im Spiel selbst besitzt ein leeres GameObject dieses Skript und wird damit zu einem Spieler. Hierdurch können beliebig viele Spieler eingefügt werden.

```
1  public class PlayerScript : MonoBehaviour
2  {
3      public Camera cameraView;
4      public GameObject village;
5      public GameObject road;
6      public float roadRange;
7
8      public Color color;
9
10     public Vector3 playerView;
11
12     public int brick = 0;
13     public int wheat = 0;
14     public int stone = 0;
15     public int wood = 0;
16     public int sheep = 0;
17
18     public Text brickTxt, woodTxt, sheepTxt, stoneTxt, wheatTxt;
19
20     public bool isFirstTurn, isSecondTurn;
21
22     public bool freeBuild;
23     public bool freeBuildRoad;
24
25     public List<GameObject> villages;
26     public List<GameObject> roads;
27
28     public int longestRoad;
29     public int victoryPoints;
30
31     private List<GameObject> roadsModified;
```

Listing 5.3: Attribute des Spielers

In Listing 5.3 ist der Beginn des PlayerScripts zu sehen. Es verfügt über eine Vielzahl an Attributen, die unterschiedlich Funktionen dienen. So braucht es einen eigenen Vector3, der die Perspektive, aus der der Spieler auf das Spielbrett schaut, beschreibt und auf die ebenfalls verfügbare Camera angewendet werden kann. Des Weiteren verfügt kann der

Spieler auf die Vorlage zum Bau von Siedlungen (GameObject village) und Straßen (GameObject road) zugreifen, sowie eine Farbe besitzen, in der später seine Objekte auf dem Spielfeld erscheinen. Zudem müssen seine verfügbaren Ressourcen gespeichert werden und eine Möglichkeit der Ausgabe durch dafür vorgesehene Textfelder gegeben werden. Zusätzlich stellen die Variablen *isFirstTurn* und *isSecondTurn* eine Möglichkeit dar, ob wir uns am Beginn des Spieles befinden. Eine Zeit, in der Spieler kostenlos bauen darf. Dieses kostenlose Bauen kann dann durch *bool freeBuild* bzw. *bool freeBuildRoad* ermittelt werden. Schließlich braucht es neben zwei Listen, die alle Siedlungen (List<GameObject> villages) und alle Straßen (List<GameObject> roads), noch die Anzahl an Siegpunkten und die Länge der längsten Straße des Spielers speichern. roadsModified stellt nur eine Hilfsliste dar.

```
1    private void Awake()
2    {
3        isFirstTurn = false;
4        isSecondTurn = false;
5        longestRoad = 0;
6        victoryPoints = 0;
7        roadRange = 1.1f;
8    }
9
10   void Start()
11   {
12       brickTxt.text = "Lehm: " + brick.ToString();
13       woodTxt.text = "Holz: " + wood.ToString();
14       wheatTxt.text = "Getreide: " + wheat.ToString();
15       sheepTxt.text = "Schafe: " + sheep.ToString();
16       stoneTxt.text = "Stein: " + stone.ToString();
17
18       villages = new List<GameObject>();
19   }
```

Listing 5.4: Initialiserung des Spielers

Die beiden in Listing 5.4 abgebildeten Methode *private void Awake()* initialisiert einige der Attribute auf ihre Anfangswerte, *void Start()* erreicht im Grunde das Gleiche, wird aber nach der Awake-Methode aufgerufen. Dies wird hier benötigt, weil sichergegangen werden muss, dass die Text-Objekte bereits erzeugt wurden, wenn auf diese zugegriffen wird. Außerdem wird die Liste der Siedlungen und die der Straßen leer initialisiert.

```
1  //Sorgt für die richtigen Einstellungen zu Beginn eines Zuges
2    public void FirstTurn()
3    {
4        AdjustCamera();
5        isFirstTurn = true;
6        freeBuild = true;
7        freeBuildRoad = false;
8    }
9
10   public void SecondTurn()
11   {
12       AdjustCamera();
13       isFirstTurn = false;
14       isSecondTurn = true;
15       freeBuild = true;
```

```
16          freeBuildRoad = false;
17      }
18
19      public void Turn()
20      {
21          AdjustCamera();
22          freeBuild = false;
23          freeBuildRoad = false;
24          isFirstTurn = false;
25          isSecondTurn = false;
26      }
27
28      private void AdjustCamera()
29      {
30          cameraView.transform.position = playerView;
31      }
```

Listing 5.5: Attribute und Initialiserung des Spielers

In Listing 5.5 sind die Methoden gezeigt, die zu Beginn eines Zuges des Spielers aufgerufen werden müssen. FirstTurn() wird nur im ersten Zug aufgerufen. SecondTurn() als zweites und Turn() zu Beginn jedes anderen Zuges. Alle drei Methoden beginnen damit, dass sie AdjustCamera() aufrufen. Hierbei handelt es sich um eine weitere Methode, die die Position der gespeicherten Kamera auf die Perspektive des Spielers setzt. Anschließend werden die bool-Werte zur Identifikation des Zuges entsprechend gesetzt, sowie festgelegt, ob in diesem zug kostenlos gebaut werden darf. In den Methoden für die ersten beiden Züge wird hierzu *freeBuild* auf true gesetzt, *freeBuildRoad* jedoch auf *false*. Dies wird erst *true*, sobald die Siedlung im zug gebaut wurde. In *Turn()* darf überhaupt nicht kostenlos gebaut werden, daher werden *freeBuild* und *freeBuildRoad* auf *false* gesetzt und bleiben dies auch.

```
1  public bool HasResourcesForVillage()
2      {
3          if (brick >= 1 && wood >= 1 && sheep >= 1 && wheat >= 1)
4              return true;
5          return false;
6      }
7
8      public bool HasResourcesForRoad()
9      {
10         if (brick >= 1 && wood >= 1)
11             return true;
12         return false;
13     }
```

Listing 5.6: Attribute und Initialiserung des Spielers

Nachdem Listing 5.5 und reflst:PlayerScript1 die Intialisierung des Spielers und seiner Züge zeigten, bringt Listing 5.6 die erste Funktionalität mit sich. Abgebildet sind zwei Methoden, die erste überprüft (HasRessoucesForVillage()), ob der Spieler genug Ressourcen zum Bau einer Siedlung hat und die zweite bewerkstelligt gleiches für die Straßen (HasRessourcesForRoad()). Für eine Siedlung benötigt der Spieler ein Lehm, ein Holz, ein Schaf und einmal Getreide, für eine Straße nur jeweils ein Holz und ein Lehm. Über die Ressourcenvariablen, wird abgefragt, ob diese Rohstoffe vorhanden und falls dies der Fall wird *true* zurückgegeben, ansonsten *false*.

```
1  public void CollectResources(int number)
2  {
3      foreach (GameObject village in villages)
4      {
5          foreach (GameObject tile in village.GetComponent<Village>().tiles)
6          {
7              if (tile.GetComponent<Tile>().number == number)
8              {
9                  Debug.Log(tile.GetComponentInChildren<Renderer>().sharedMaterial.name);
10                 if
                       (tile.GetComponentInChildren<Renderer>().sharedMaterial.name
                       == "wheat 1")
11                 {
12                     wheat++;
13                 }
14                 else if
                       (tile.GetComponentInChildren<Renderer>().sharedMaterial.name
                       == "sheep")
15                 {
16                     sheep++;
17                 }
18                 else if
                       (tile.GetComponentInChildren<Renderer>().sharedMaterial.name
                       == "rock")
19                 {
20                     stone++;
21                 }
22                 else if
                       (tile.GetComponentInChildren<Renderer>().sharedMaterial.name
                       == "brick")
23                 {
24                     brick++;
25                 }
26                 else if
                       (tile.GetComponentInChildren<Renderer>().sharedMaterial.name
                       == "wood")
27                 {
28                     wood++;
29                 }
30             }
31         }
32     }
33     UpdateResources();
34 }
```

Listing 5.7: Attribute und Initialiserung des Spielers

Neben dem Bau von neuen Siedlungen, muss auch der Nutzen einer dieser ermittelt werden. Denn eine jede Siedlung kann, richtig platziert, dem Besitzer, bestimmte Rohstoffe einbringen. Hierzu existieren zwei Methoden *CollectResourcesForVillage(GameObject village)* und *CollectResources(int number)*. Beide erreichen im Grunde das Gleiche, wobei *CollectResourcesForVillage(GameObject village)* die Rohstoffe für eine konkrete Siedlung sammelt und *CollectResources(int number)* abhängig von der gewürfelten Nummer und zusätzlich wird ermittelt, welche Siedlungen an den entsprechenden Feldern mit der passenden Nummer liegen.
Zu jeder Siedlung werden die angrenzenden Felder gespeichert, diese können entweder *null*

oder von einem bestimmten Typ sein, der über deren *Material* abgefragt wird. Auf diese Weise kann nun durch verschachtelte if-Verzweigungen ermittelt werden, ob und welche Rohstoffe dem Spieler zuzusprechen sind. Am Ende beider Methoden wird *UpdateResources()* aufgerufen, eine Methode, die die aktualisierten Mengen an Rohstoffen ausgibt. Exemplarisch ist in Listing 5.7 nur *CollectResources(int number)* aufgeführt. Die andere ist äquivalent und daher nur im Anhang zu finden.

```
1  public GameObject BuildVillage(Vector3 position)
2  {
3      if (freeBuild || this.HasResourcesForVillage())
4      {
5          if (!freeBuild)
6          {
7              brick--;
8              wood--;
9              sheep--;
10             wheat--;
11         }
12         GameObject villageInstance = Instantiate(village);
13         villageInstance.transform.position = position;
14         villageInstance.GetComponent<Renderer>().material.color = color;
15
16         freeBuild = false;
17         freeBuildRoad = true;
18
19         UpdateResources();
20
21         victoryPoints++;
22         return villageInstance;
23     }
24     return null;
25 }
26
27 public GameObject BuildRoad(Vector3 position, Quaternion rotation)
28 {
29     if (freeBuildRoad || this.HasResourcesForRoad())
30     {
31
32         if (!freeBuildRoad)
33         {
34             brick--;
35             wood--;
36         }
37
38         GameObject roadInstance = Instantiate(road);
39         roadInstance.transform.position = position;
40         roadInstance.transform.rotation = rotation;
41         roadInstance.transform.Rotate(-90.0f, 0.0f, 0.0f, Space.Self);
42         roadInstance.GetComponent<Renderer>().material.color = color;
43
44         freeBuildRoad = false;
45
46         UpdateResources();
47
48         return roadInstance;
49     }
50     return null;
```

```
51 }
```

Listing 5.8: Attribute und Initialiserung des Spielers

Nachdem eben erläutert wurde, wie die Ermittlung der nötige Menge an Rohstoffen zum Bau einer Siedlung oder Straße erfolgt und der Ertrag einer Siedlung, stellt Listing 5.8 den tatsächlichen Bau von Siedlungen und Straßen dar. Die Methode *BuildVillage(Vector3 position)* bekommt als Parameter die Position für den Bau der Siedlung übergeben und kann nun dort eine platzieren. Zu Beginn der Methode wird überprüft, ob der Spieler entweder kostenlos bauen darf oder er genug Rohstoffe zum Bau hat. Falls er genug Rohstoffe hat, aber nicht kostenlos bauen darf, verliert der Spieler Rohstoffe in der Menge, die zum Bau einer Siedlung nötig sind. Außerdem wird ihm das Recht auf einen kostenlosen Bau wieder entzogen, gleichzeitig wird ihm aber erlaubt eine kostenlose Straße zu bauen. In jedem Fall wird im Anschluss an diese Prüfung eine neue Siedlung auf Grundlage des GameObjects *village* instanziiert. Bei dem instanziierten Objekt handelt es sich um eine weitere Instanz, des bereits als Repräsentation instanziierten, des dafür vorgesehenen Prefabs. Dieser Instanz wird die zuvor übergebene Position zugeschrieben und es erhält die Farbe des Spielers als seine Oberflächenfarbe. Hierauf folgend werden die Siegpunkte des Spielers um einen erhöht, sowie seine Menge an Rohstoffen aktualisiert und schließlich die Instanz der Siedlung zurückgegeben.
Die Methode zm Bau einer Straße funktioniert äquivalent wie für Siedlungen mit dem Unterschied, dass der Spieler sämtliche kostenlosen Baurechte verliert und zudem eine Rotation an der Straße vorgenommen werden muss. Die Methode erhielt den Grad der Rotation als Parameter und kann diese nun auf die neue Instanz anwenden. Das ist im Gegensatz zur Methode für Siedlungen nötig, weil Straßen auf den Kanten von zwei Sechsecken stehen und diese in eine bestimmte Richtung zeigen. Damit dies auf dem Spielfeld ansehnlicher aussieht, werden die instanziierten Straßen in dieselbe Richtung gedreht.

```
1   public void CalculateRoadLength()
2   {
3       if (roads.Count > 0)
4       {
5           List<int> roadLengths = new List<int>();
6           foreach(GameObject road in roads)
7           {
8               roadsModified = new List<GameObject>(roads);
9               roadLengths.Add(FindRoadNeighbors(road));
10          }
11          longestRoad = roadLengths.Max();
12      }
13      else
14      {
15          longestRoad = 0;
16      }
17  }
18
19  private int FindRoadNeighbors(GameObject road)
20  {
21      int neighbors = 0;
22      List<GameObject> roadsUpdated = new List<GameObject>();
23
24      for (int i = 0; i < this.roadsModified.Count;)
25      {
26          GameObject r = this.roadsModified[i];
27
```

```
28              float distance = (road.transform.position -
                    r.transform.position).magnitude;
29              if (distance < roadRange)
30              {
31                  roadsModified.Remove(r);
32                  roadsUpdated.Add(r);
33                  neighbors++;
34              }
35              else
36              {
37                  i++;
38              }
39          }
40          foreach (GameObject r in roadsUpdated)
41          {
42              neighbors += FindRoadNeighbors(r);
43          }
44          return neighbors;
45      }
```

Listing 5.9: Attribute und Initialiserung des Spielers

5.9 führt uns die Berechnung der längsten Straße eines Spielers ein. Hierfür sind zwei Methoden nötig. Die erste *CalculateRoadLength* prüft zunächst, ob der Spieler überhaupt über Straßen verfügt. Falls dies wahr ist, wird eine neue Liste zur späteren Speicherung der Längen aller Straßen angelegt. Nun werden alle Straßen in der globalen Liste *roads* mittels foreach-Schleife durchlaufen. Für jede Straße wird zu Beginn die globale Liste leer initialisiert und anschließend wird die Menge an Nachbarn für diese Straße ermittelt. Die Anzahl an Nachbarn stellt dann die Länge der Straße dar und durch das Ausfindig machen des Maximums der Liste *roadLengths* kann so die längste Straße des Spielers ermittelt werden. Sollte der Spieler über keine Straßen verfügen, hat die längste Straße eine Länge von 0.
Zum berechnen der Anzahl der Nachbarn einer Straße, wird die zweite Methode *FindRoadNeighbors(GameObject road)* gebraucht. Diese ermittelt viele zusammenhängende Straßen von einer Straße abgehen. Am Anfang der Methode gibt es 0 Nachbarn und eine leere Liste aus GameObjects. Anschließend wird für jedes Objekt in der globalen Liste *roadsModified* die Distanz zur als Parameter übergebenen Straße berechnet. Sollte die Distanz geringer sein, als die Distanz, die einen direkten Nachbarn kennzeichnet, so handelt es sich bei der betrachteten Straße um einen direkten Nachbarn der übergebenen Straße. Eine Straße kann auch Nachbar von sich selbst sein, um auf die Länge eins bei einer Straße zu kommen. Wenn eine Nachbar gefunden wurde, wird dieser aus *roadsModified* entfernt, aber zu *roadsupdated* hinzugefügt. Außerdem wird die Anzahl der Nachbarn um eins erhöht. Wenn andererseits die betrachtete Straße kein direkte Nachbar der übergebenen ist, so wird i um eins erhöht, um zur nächsten Straße überzugehen. Schließlich geschieht der wohl möglich interessanteste Teil dieser Methode. Für jede Straße in *roadsUpdated* wird erneut *FindRoadNeighbors(GameObject road)* aufgerufen. Das Ergebnis dieses Aufrufs wird wiederum zu der Anzahl der aktuellen Nachbarn hinzugezählt. Durch dieses rekursive Vorgehen, kann die komplette Länge aller zusammenhängenden Straßen einfach berechnet werden. Wenn die Rekursion abgeschlossen ist, wird die Anzahl an Nachbarn zurückgegeben.
Da *CalculateRoadLength()* die Anzahl an Nachbarn für jede Straße kalkuliert, ist sichergestellt, dass die tatsächlich längste Straße gefunden wird, weil jede einmal als Startpunkt fungiert.

5.2.3 Der Gamemanager

Der Gamemanager dient der Steuerung des Ablaufs des Spiels. Hierüber wird später auch der Zugriff auf die Schnittstelle gewährleistet, sodass KI-Spieler und Spiel miteinander interagieren können. Dieser Manager sorgt für einen reibungslosen Ablauf des Spiels, steuert wer wann dran ist und überprüft alle Geschehnisse, die während des Spiels vor sich gehen. Ohne ihn wäre kein Zusammenspiel möglich.

```
1  ...
2  void ChangePlayer()
3  {
4      cameraView.transform.Rotate(0.0f, 180.0f, 0.0f, 0);
5      if (activePlayer == player1)
6      {
7          UpdateActivePlayer(player2);
8      }
9      else
10     {
11         UpdateActivePlayer(player1);
12     }
13     activePlayer.Turn();
14 }
15
16 private void UpdateActivePlayer(PlayerScript player)
17 {
18     activePlayer = player;
19
20     activePlayer.UpdateVictoryPoints();
21     if (player1.longestRoad > player2.longestRoad)
22     {
23         player1.victoryPoints += 2;
24     } else if (player1.longestRoad < player2.longestRoad)
25     {
26         player2.victoryPoints += 2;
27     }
28
29   ... //Aktivierung der Bauplätze
30
31     buildVillage[] buildVillages =
           villagePlaces.GetComponentsInChildren<buildVillage>();
32     foreach (buildVillage build in buildVillages)
33     {
34         build.player = activePlayer;
35     }
36
37     ... //Deaktivierung der Bauplätze
38
39     ... //Aktivierung der Bauplätze
40
41     BuildRoad[] buildRoads = roadPlaces.GetComponentsInChildren<BuildRoad>();
42     foreach (BuildRoad build in buildRoads)
43     {
44         build.player = activePlayer;
45     }
46
47     ... //Deaktivierung der Bauplätze
48 }
```

Listing 5.10: Initialisierung und Spielerwechsel

In Listing 5.10 werden zwei Methoden gezeigt. der Methode *ChangePlayer* wären normalerweise die Awake- und Startmethode und die Liste der deklarierten Attribute zu finden. Aus Platz Gründen wurde hier auf die Einbindung dieser Teile verzichtet. Sie können jedoch im Anhang eingesehen werden. Nötige Erklärungen zu betroffenen Variablen sind weiterhin bei der Beschreibung ihrer Verwendung zu finden. Zusätzlich fehlen einige Teile der *UpdateActivePlayer(PlayerScript player)*-Methode, da diese sonst zu lang würde. Entfernte Stellen sind durch jeweils drei Punkte gekennzeichnet.
Neben *UpdateActivePlayer* existiert noch *ChangePlayer()*. Diese rotiert zunächst die Kamera um 180 Grad in der Vertikalen, um den Spielerwechsel für einen menschlichen Betrachter zu signalisieren. anschließend wird der Spieler gewechselt, der gerade am Zug ist. Auch in der *ChangePlayer-Methode* wird zum Setzen des aktuellen Spielers die *UpdateActivePlayer*-Methode genutzt. Diese Methode bekommt als Parameter das Skript des zu setzenden Spielers und setzt die globale Variable *activePlayer* auf diesen Zustand. Anschließend werden die Siegpunkte des Spielers aktualisiert. Zunächst durch den Aufruf der passenden Methode des Spielers und dann durch einen Vergleich der jeweils längsten Straße beider Spieler. Der Spieler mit der längsten Straße erhält zwei zusätzliche Siegpunkte. Die Siegpunkte zuvor zu aktualisieren und die Vernachlässigung der längsten Straße innerhalb dieser Methode ist nötig, weil ansonsten ein Spieler mit der zuvor längsten Straße seine Punkte nicht wieder verlieren würde.
Im nächsten Schritt muss ein potenzieller Bauplatz für eine Siedlung erfahren, welcher Spieler darauf baut, wenn hier eine Siedlung entsteht. Hierzu müssen zunächst die deaktivierten Bauplätze aktiviert werden und dann wird dem verantwortlichen Skript als Attribut jedes Bauplatzes der aktuelle Spieler mitgeteilt. Um dies zu erreichen wird zunächst mithilfe einer Schleife jeder potenzielle Bauplatz durchlaufen und aktiviert. Anschließend wird das Elternelement aktiviert. Nun kann über das Skript jedes Bauplatzes iteriert werden und der aktuelle Spieler gesetzt werden. Anschließend erfolgt die Deaktivierung aller Bauplätze äquivalent zu deren Aktivierung. Dies ist nötig, da ohne den Aktivierungsschritt kein Zugriff auf die Skripte erfolgen könnte. Der ganze Prozess bleibt allerdings vor dem Nutzer verborgen.
Exakt das gleiche vorgehen wird auch auf die potenziellen Bauplätze für die Straßen angewandt. Auch diese benötigen den aktuellen Spieler, um feststellen zu können, wer die angefragte Straße bauen möchte.

```
1  void Update()
2  {
3      if (villageFocus.hasFocus && (activePlayer.isFirstTurn ||
           activePlayer.isSecondTurn))
4      {
5          villagePlaces.SetActive(true);
6          Transform transform = villagePlaces.GetComponent<Transform>();
7          for (int i = villagePlaces.GetComponent<Transform>().childCount - 1;
               i >= 0; i--)
8          {
9              Transform child = transform.GetChild(i);
10             child.gameObject.SetActive(true);
11         }
12         if (buildVillage != null)
13         {
14             DestroyVillagePlacesInRadius();
```

```
15          }
16
17      }
18      ...//Äuqivalent mit HasResourcesForVillage statt isFirstTurn und is
            SecondTurn
19      else
20      {
21          Transform transform = villagePlaces.GetComponent<Transform>();
22          for (int i = villagePlaces.GetComponent<Transform>().childCount - 1;
                i >= 0; i--)
23          {
24              Transform child = transform.GetChild(i);
25              child.gameObject.SetActive(false);
26          }
27          villagePlaces.SetActive(false);
28      }
```

Listing 5.11: Update-Methode

Listing 5.11 zeigt die Methode *void Update()*. Sie kontinuierlich aufgerufen und dauerhaft ausgeführt. Sie beinhaltet daher Anweisungen, die immer wieder durchgeführt werden müssen. Der Aufruf geschieht durch Unity und wird nicht selbstständig gesteuert.
Zu Beginn der Methode wird überprüft, ob potenzielle Bauplätze für Siedlungen aktiviert werden sollen. Hierzu muss der Spieler am Zug auf dem Spielfeld die Repräsentation der Siedlung ausgewählt haben. Dadurch wird villageFocus.hasFocus wahr und die Überprüfung der weiteren Bedingungen kann stattfinden. Sofern es sich um den ersten oder zweiten Zug des Spielers handelt, werden alle potenziellen Bauplätze angezeigt und es kann auf einem gebaut werden. Jedoch schränkt der Bau einer Siedlung die potenziellen Plätze für folgende ein, da zwischen zwei Siedlungen mindestens zwei Straßen sein müssen. Um dies zu gewährleisten, werden die Bauplätze in einem zuvor festgelegtem Umkreis (*DestroyVillagePlacesInRadius()*) gelöscht.
Sollte es sich bei dem aktuellen um einen Standardzug handeln, so wird in der Bedingung eines *else if-Zweiges* geprüft, ob der Spieler genügend Ressourcen hat, falls ja, darf er eine Siedlung bauen, aber nur da, wo sich auch eine Straße direkt an der Siedlung befinden würde. Dadurch und die Löschung der zu nahen Bauplätzen wird die Einhaltung der zwei-Straßen-Abstand-Regeln umgesetzt. Konkret funktioniert die Prüfung, ob eine Straße anliegt so, dass alle gebauten Straßen des Spielers durchlaufen werden und geschaut wird, ob ein Bauplatz an diesen anliegt. Der anliegenden Bauplatz wird daraufhin aktiviert und für den Spieler sichtbar.
Falls sich keine der beiden obigen Bedingungen als wahr herausstellt, so werden die die angezeigten Bauplätze wieder deaktiviert. Durch den kontinuierlichen Aufruf der Update-Methode gibt es keine merkbare Verzögerung für den Spieler und die Deaktivierung der Bauplätze erfolgt, sobald entweder ein erneuter Klick auf die Repräsentation vorgenommen wird, der Spieler den zweiten Zug verlässt oder seine Ressourcen nicht mehr ausreichen, um eine weiter Siedlung zu errichten.

```
1   if (roadFocus.hasFocus && (activePlayer.HasResourcesForRoad() ||
        activePlayer.isFirstTurn || activePlayer.isSecondTurn))
2   {
3       roadPlaces.SetActive(true);
4       foreach (GameObject village in activePlayer.villages)
5       {
6           Transform transform = roadPlaces.GetComponent<Transform>();
```

```
7             for (int i = roadPlaces.GetComponent<Transform>().childCount - 1;
                  i >= 0; i--)
8             {
9                 Transform child = transform.GetChild(i);
10                float distance = (village.transform.position -
                      child.position).magnitude;
11                if (distance < roadRange)
12                {
13                    child.gameObject.SetActive(true);
14                }
15            }
16        }
17        foreach (GameObject road in activePlayer.roads)
18        {
19            Transform transform = roadPlaces.GetComponent<Transform>();
20            for (int i = roadPlaces.GetComponent<Transform>().childCount - 1;
                  i >= 0; i--)
21            {
22                Transform child = transform.GetChild(i);
23                float distance = (road.transform.position -
                      child.position).magnitude;
24                if (distance < roadRange)
25                {
26                    child.gameObject.SetActive(true);
27                }
28            }
29        }
30
31        if (buildRoad != null)
32        {
33            if (!activePlayer.freeBuild && !activePlayer.freeBuildRoad &&
                  (activePlayer.isFirstTurn || activePlayer.isSecondTurn))
34                endTurnBtn.interactable = true;
35        }
36    } else
37    {
38        ... //Deaktivieren wie für die Siedlungen
39    }
40 }
```

Listing 5.12: Update-Methode Aktvierung der Bauplätze für Straßen

Erneut wird das gleiche Prinzip auch auf den Bau von Straßen angewandt (siehe Listing **??**), doch mit wichtigen Anpassungen. Bei den Siedlungen war der Fall, dass im ersten und zweiten Zug jeder ungelöschte Bauplatz infrage kommt. Bei den Straßen ist das etwas anders. Hier darf der Bau nur erfolgen, wenn direkt an ihr entweder bereits eine Siedlung oder eine weitere Straße gebaut wurde. Die Überprüfung erfolgt erneut durch die Iteration über alle Siedlungen in der äußeren Schleife und über alle Bauplätze für Straßen in der inneren. Es werden nur die Bauplätze aktiviert, die sich direkt an einer bereits existierenden Siedlung befinden. Anschließend wird das gleiche Prozedere vorgenommen mit den bereits existierenden Straßen und auch hier werden die anliegenden Bauplätze freigeschaltet. Anders als bei der Aktivierung der Siedlungsbauplätze kann die für die Straßen ohne eine weitere Verzweigung in else if-Form ablaufen, da der Ablauf der Aktivierung für beide Fälle der gleiche ist und somit Zeile 3 und Zeile 17 für die Straßen als Zeile 50 zusammengefasst werden können. Die Deaktivierung wiederum erfolgt exakt äquivalent.

```
1   public void OnRollDice()
2    {
3        // left mouse button clicked so roll random colored dice 2 of each
             dieType
4        Dice.Clear();
5        Dice.Roll("1d6", "d6-" + "red", spawnPoint.transform.position,
             Force());
6        Dice.Roll("1d6", "d6-" + "red", spawnPoint.transform.position,
             Force());
7        rollDiceBtn.interactable = false;
8
9        StartCoroutine(AddResources());
10       StartCoroutine(EnableEndTurn());
11   }
12
13   private Vector3 Force()
14   {
15       Vector3 rollTarget = Vector3.zero + new Vector3(2 + 7 * Random.value,
             .5F + 4 * Random.value, -2 - 3 * Random.value);
16       return Vector3.Lerp(spawnPoint.transform.position, rollTarget,
             1).normalized * (-35 - Random.value * 20);
17   }
18
19   IEnumerator AddResources()
20   {
21       yield return new WaitForSeconds(3f);
22       int number = Dice.GetValue("");
23       player1.CollectResources(number);
24       player2.CollectResources(number);
25       activePlayer.UpdateResources();
26   }
27
28   IEnumerator EnableEndTurn()
29   {
30       yield return new WaitForSeconds(4f);
31       endTurnBtn.interactable = true;
32   }
```

Listing 5.13: Würfeln

Ein weiterer wichtiger Bestandteil des Gamemanagers ist das Würfeln. Listing 5.13 zeigt wie dies bewerkstelligt wird. Es werden hierfür insgesamt zwei Methoden benötigt. Die erste *public void OnRollDice()* löscht zunächst den vorherigen Würfelversuch, um dann anschließend zwei rote sechs-seitige Würfel rollen zulassen. Daraufhin wird das erneute Würfeln verboten und es werden zwei Coroutinen gestartet. Die eine zum Hinzufügen von Ressourcen zum Bestand der Spieler und die andere zum Erlauben des aktuellen Zuges.
Die Würfel stammen aus der Asset-Bibliothek von Unity und wurden nicht selbst implementiert. Es wird hier lediglich auf eine bereits vorhandene Implementation zugegriffen, daher entfällt eine genauere Betrachtung des Vorgangs des Würfels. Dieser ist jedoch auch wenig interessant für den Kernpunkt der Arbeit und hätte keine Relevanz für die Ergebnisse, sondern würde nur besser verwendbaren Platz beanspruchen.
Wenn die Würfeln gerollt werden (*Dice.Roll(... Force())*) wird unter anderem auch die Methode das Ergebnis des Methodenaufrufs von *Force()* übergeben. Diese Methode wiederum liefert einen dreidimensionalen Vektor zurück, der eine zufällige Kraft enthält und den Startpunkt der Würfel teilt. Wenn diese Kraft nun auf die Würfel angewandt wird,

rollen sich diese durch ihre eigenen Physik in zufälliger Weise.
Die besagten Routinen sind ebenfalls abgebildet. Die erste der beiden *IEnumerator AddResources()* dient dem Hinzufügen der Rohstoffe zu den Beständen aller Spieler. Wer welche erhält ist abhängig von der gewürfelten Zahl und ob ein Spieler eine Siedlung an dem zur Zahl korrespondierenden Landschaftsfeld hat. Hierzu wird zu Beginn der Routine für drei Sekunden gewartet, damit das Würfeln auf jeden Fall abgeschlossen ist. In einigen Fällen kann es jedoch dazu kommen, dass der Würfel auf seiner Kante hängen bleibt. Dann können keine Rohstoffe verteilt werden und der Zug läuft weiter ab, als hätte der Spieler nicht gewürfelt. Um solch ein Fehlverhalten auszugleichen, könnte auf die festgeschriebene Wartezeit verzichtet werden und stattdessen die Verteilung vorgenommen werden, sobald das Würfeln abgeschlossen ist. Sollte das Würfeln nicht in einer bestimmten Zeit abgeschlossen werden, weil die Würfel beispielsweise festhängen, so könne der Würfelversuch wiederholt werden. Dieses Verhalten ist wünschenswert, ist aber zum aktuellen Stand der Entwicklung verzichtbar. Daher wird es erst in kommenden Iterationen implementiert.
Nachdem die beiden Spieler ihre Rohstoffe erhalten haben wird noch die Rohstoffanzeige des aktuellen Spielers aktualisiert, um den Bestand und die Darstellung konsistent zu halten.
In der ebenfalls gezeigten Routine *IEnumerator EnableEndTurn()* wird zu Beginn vier Sekunden gewartet. Dass eine Sekunde länger gewartet wird, ist auch auf die Anzeige zurückzuführen. Denn nachdem warten wird lediglich der Button aktiviert, mit dem der Spieler seinen Zug beenden kann. Es soll jedoch zuvor wenigstens kurz angezeigt werden, welche Rohstoffe er nun besitzt. Außerdem soll er den neuen Bestand mit in seine Entscheidung einbeziehen, ob er seinen Zug überhaupt schon beenden will oder zuvor noch eine andere Tätigkeit vornimmt. Für einen menschlichen Spieler stellt eine gleichzeitige Aktualisierung der Rohstoffe und Aktivierung des Zug beenden-Buttons kein Problem dar. Für eine computergesteuerten Gegner allerdings könnte dies zur Verwirrung führen. Es müsste intern implementiert werden, dass er erst wartet, welche Rohstoffe er erhält und dann überlegen, ob der Zug noch immer beendet werden soll. Durch die Verzögerung, die durch die Routinen gewährleistet wird, fällt eine solche Implementation in der KIs weg.
Weiterhin ist die Überlegung anzustellen, warum über Routinen verwendet werden, denn diese sorgen für eine parallele Abarbeitung der Geschehnisse und das wirft die Frage auf, ob nicht im Hauptthread gewartet werden könne. Diese Frage lässt sich einfach beantworten: Damit das Warten auf das Fertigwerden des Würfelns überhaupt einen Sinn hat, muss es auch weiter ablaufen. Würde der Hauptthread warten, so würde das gesamte Spiel sinngemäß für drei Sekunden angehalten werden, wodurch anschließend versucht wird, das gewürfelte Ergebnis auszuwerten, jedoch ohne Erfolg, da die Würfeln noch immer nicht ausgerollt sind. Gleiches gilt für die Aktivierung des Buttons. Er soll auch erst dann aktiv werden, wenn die Augenzahl bereits errechnet und verarbeitet wurde. Hielte man hierfür den Hauptthread an, könnte keine Verarbeitung stattfinden. Durch das Verteilen der Prozesse auf unterschiedliche Threads können die Timer parallel ablaufen und daher den gewünschten Effekt erreichen.

```
1    public void OnEndTurn()
2    {
3        if (activePlayer.victoryPoints > 3)
4        {
5
6        }
7        if (activePlayer.isFirstTurn && activePlayer == player1)
8        {
9            UpdateActivePlayer(player2);
10           cameraView.transform.Rotate(0.0f, 180.0f, 0.0f, 0);
11
```

```
12              activePlayer.FirstTurn();
13              endTurnBtn.interactable = false;
14              rollDiceBtn.interactable = false;
15          }
16          else if (activePlayer.isFirstTurn && activePlayer == player2)
17          {
18              activePlayer.SecondTurn();
19              endTurnBtn.interactable = false;
20              rollDiceBtn.interactable = false;
21          }
22          else if (activePlayer.isSecondTurn && activePlayer == player2)
23          {
24              activePlayer.CollectResourcesForVillage(activePlayer.villages[activePlayer.villag
                    - 1]);
25              UpdateActivePlayer(player1);
26              cameraView.transform.Rotate(0.0f, 180.0f, 0.0f, 0);
27
28              activePlayer.SecondTurn();
29              endTurnBtn.interactable = false;
30              rollDiceBtn.interactable = false;
31          }
32          else if (activePlayer.isSecondTurn && activePlayer == player1)
33          {
34              activePlayer.CollectResourcesForVillage(activePlayer.villages[activePlayer.villag
                    - 1]);
35              activePlayer.Turn();
36              endTurnBtn.interactable = false;
37              rollDiceBtn.interactable = true;
38          }
39          else
40          {
41              ChangePlayer();
42              rollDiceBtn.interactable = true;
43              endTurnBtn.interactable = false;
44          }
45          activePlayer.UpdateResources();
46      }
```

Listing 5.14: Zug beenden

Wird der durch den IEnumerator aktivierte Button schließlich gedrückt, ruft dies die Methode *public void OnEndTurn()* auf. Abbgebildet ist sie als Listing 5.14. Am Anfang der Methode ist eine if-Verzweigung zu finden. Diese dient der Bestimmung des Siegers. Sollte ein Spieler in seinem aktuellen Zug die nötigen Siegpunkte erreicht bzw. überschritten haben, so ist dieser Sieger des Spiels. In diesem Fall wird das Spiel beendet und der aktuelle Spieler als Sieger ausgegeben. Sollte er nicht gewonnen haben, so wird der Rest der Methode ausgeführt. Im ersten Schritt wird geprüft, ob der aktuelle Spieler gerade seinen ersten Zug gemacht hat und ob es sich bei ihm um Spieler 1 handelt. Sollte dies der Fall sein, wird auf Spieler 2 gewechselt, die Kamera dreht sich und dieser darf seinen ersten Zug machen. Zudem müssen die Buttons zum Würfeln und zum Beenden des Zuges wieder deaktiviert werden. Falls diese es sich bei dem aktuellen Zug jedoch bereits um den ersten um Spieler 2 handelt, so wird die Kamera nicht gedreht, da er am Zug bleibt. Er darf nun direkt seinen zweiten Zug ausführen. Wenn nach diesem Zug erneut der Zug beendet wird, erhält Spieler 2 zunächst einmal Rohstoffe für seinen soeben kostenlos gebaute Siedlung. Anschließend wird wieder zu zum zweiten Zug von Spieler 1 gewechselt nach dem bekannten Prozedere. Der letzte durchzuführende zug vor den Standardzügen ist der zweite von

Spieler 1. Wird dieser beendet, so erhält auch er Rohstoffe für die zuletzt gebaute Siedlung. Er darf nun mit dem ersten Standardzug beginnen. Hierzu wird zuvor der Würfelbutton aktiviert und er kann diesen nun betätigen. Die Standardzüge werden in *OnEndTurn()* durch den else-Zweig der bisher beschriebenen Fallabfrage beendet. Sollte also keine der zuvor beschriebenen Bedingungen wahr sein, so muss es sich um einen Standardzug handeln. Dieser wird durch den Aufruf der bereits beschriebenen Methode *ChangePlayer()* (siehe Listing 5.10) und die darauffolgende Aktivierung des Würfelbuttons sowie Deaktivierung des Zug-Beenden-Buttons beendet.
Egal welche der obigen Anweisungen aufgrund der geltenden Bedingungen durchgeführt wurden, in jedem Fall wird am Schluss der Methode der Bestand über die Rohstoffe des aktiven Spielers aktualisiert.

5.2.4 Die Repräsentation der Bauobjekte

In der Beschreibung des Spielfelds wurde erwähnt, dass die abgebildeten Objekte die zu bauenden Siedlungen (links oben) und Straßen (rechts unten) darstellen. In 5.2.3 wurde dann auf die Nutzung dieser eingegangen und dass sie dem Zweck dienen, die den Baumodus für die jeweiligen Objekte an- und ausschalten zu können.

```
1  public class VillageFocus : MonoBehaviour
2  {
3      public bool hasFocus = false;
4      public RoadFocus roadFocus;
5
6      private void OnMouseDown()
7      {
8          hasFocus = !hasFocus;
9          if (hasFocus)
10             roadFocus.hasFocus = false;
11     }
12 }
```

Listing 5.15: Aktivierung des Baumodus für Siedlungen

5.15 zeigt wie dieser Mechanismus für den Baumodus von Siedlungen implementiert ist. Sofern ein Klick auf die angesprochene Repräsentation auf dem Spielfeld erfolgt, wird der Baumodus deaktiviert, wenn er aktiviert ist und aktiviert, wenn er deaktiviert ist. Beachtet werden sollte noch die Variable roadFocus. Diese enthält das äquivalente Skript für die Straßen (Listing 5.16) und über sie kann gewährleistet werden, dass nicht beide Modi gleichzeitig aktiviert sind. Falls der Baumodus für Siedlungen aktiviert wurde, wird im nächsten Schritt der für Straßen deaktiviert. Wird ein Blick auf das Listing 5.16 geworfen, so ist die Ähnlichkeit erkennbar. Beide Skripte erreichen das gleiche, nur auf den jeweiligen Baumodus bezogen. So ist auch das Skript *VillageFocus* als Variable in *RoadFocus* gespeichert, um hier den Baumodus deaktivieren zu können.

```
1  public class RoadFocus : MonoBehaviour
2  {
3      public bool hasFocus = false;
4      public VillageFocus villageFocus;
5
6      private void OnMouseDown()
7      {
```

```
8         hasFocus = !hasFocus;
9         if (hasFocus)
10            villageFocus.hasFocus = false;
11    }
12 }
```

Listing 5.16: Aktivierung des Baumodus für Straßen

5.2.5 Der Bau von Siedlungen und Straßen

Die Klasse *buildVillage* (siehe Listing 5.17)ist das Skript eines jeden Bauplatzes für Siedlungen. Sie verfügt über zwei globale Variablen, die den aktuellen Spieler und den verantwortlichen GameManager repräsentieren. Als einzige Methode verfügt sie über *private void OnMouseDown()*, die als OnClick-Reaktion pro Bauplatz fungiert. Dieser OnClick-Mechanismus funktioniert nur, wenn die Bauplätze auch aktiviert sind, daher wurde im Abschnitt 5.2.3 ausgiebig beschrieben, wann dies der Fall ist.
Wenn nun ein Klick auf einen aktivierten Bauplatz stattfindet, so wird zunächst eine Instanz einer Siedlung erzeugt mithilfe der dafür vorgesehenen Methode *BuildVillage(Vector3 position)*, die in der Klasse PlayerScript 5.8 zu finden ist und durch den aktiven Spieler aufgerufen wird. Wie die Methode eine Instanz für den aufrufenden Spieler erzeugt, wird in 5.2.2 näher beschrieben. Nachdem die Instanz zurückgeliefert wurde, wird auf sein Skript *Village* zugegriffen. Dieses ist unter Listing 5.19 zu sehen. Bei diesem Skript handelt es sich jedoch nur um eine Hilfsklasse, die lediglich ein einziges Attribut in Form einer Liste genannt *tiles* beinhaltet. Ähnlich funktioniert das Skript *PlaceScript*. Dieses enthält drei Attribute, die der Speicherung der anliegenden Landschaftskarten dienen, an die der Bauplatz angrenzt. Mithilfe beider Klassen kann nun dem entstehenden Dorf zugewiesen werden, welche Rohstoffe in seiner Nähe liegen. Diese Zuweisung ist von Zeile 13-18 sichtbar. Die Prüfung auf *null*-Werte erfolgt, da noch keine Wasserfelder existieren und daher einige Bauplätze, die nicht an drei Felder grenzen, teilweise *null*-Werte in den Attributen stehen haben.
Abschließend wird das entstandene Dorf der Liste der Dörfer des Spielers (siehe Ziele 20) hinzugefügt, damit später darauf zugegriffen werden kann. Zudem erhält der Gamemanager Zugriff auf die entstandene Instanz. Diese Setzung der Variable löst die, in Abschnitt 5.2.3 beschriebene Löschung der benachbarten Bauplätze durch die Update-Methode, aus.

```
1  public class buildVillage : MonoBehaviour
2  {
3      public PlayerScript player;
4      public GameManager gameManager;
5
6      private void OnMouseDown()
7      {
8              GameObject villageInstance =
                   player.BuildVillage(this.transform.position);
9
10             Village villageScript = villageInstance.GetComponent<Village>();
11             PlaceScript placeScript = this.GetComponent<PlaceScript>();
12
13             if (placeScript.resourceTile1 != null)
14                 villageScript.tiles.Add(placeScript.resourceTile1);
15             if (placeScript.resourceTile2 != null)
16                 villageScript.tiles.Add(placeScript.resourceTile2);
17             if (placeScript.resourceTile3 != null)
18                 villageScript.tiles.Add(placeScript.resourceTile3);
```

```
19
20              player.villages.Add(villageInstance);
21              gameManager.buildVillage = villageInstance;
22
23      }
24
25  }
```

Listing 5.17: Bau von Siedlungen

In Listing 5.18 ist wie gewohnt der gleiche Ablauf für den Bau von Straßen zu sehen. Hier sind ebenso der aktive Spieler und der verantwortliche Gamemanager gespeichert. Auch für die Straßen wird die Instanziierung über die entsprechende Methode des Spielers vorgenommen. Sofern die dabei entstandene Instanz ungleich *null* ist, wird sie Liste der Straßen des Spielers hinzugefügt, seine längste Straße wird erneut berechnet und der Gamemanager erhält Zugriff auf die Instanz. Abschließend wird noch das aufrufende GameObject zerstört, um den Bauplatz unter der Straße zu entfernen. Ein Schritt, der beim Bau von Siedlungen entfällt, da hier im Nachhinein alle Bauplätze im Nachbarradius zerstört werden.

```
1  public class BuildRoad : MonoBehaviour
2  {
3      public PlayerScript player;
4      public GameManager gameManager;
5
6      private void OnMouseDown()
7      {
8          GameObject roadInstance = player.BuildRoad(this.transform.position,
               this.transform.rotation);
9
10         if (roadInstance != null)
11         {
12             player.roads.Add(roadInstance);
13             player.CalculateRoadLength();
14             gameManager.buildRoad = roadInstance;
15
16             Destroy(this.gameObject);
17         }
18     }
19 }
```

Listing 5.18: Bau von Straßen

```
1  public class Village : MonoBehaviour
2  {
3      public List<GameObject> tiles;
4  }
5
6  public class PlaceScript : MonoBehaviour
7  {
8      public GameObject resourceTile1, resourceTile2, resourceTile3;
9  }
```

Listing 5.19: Hilfsklassen Village und PlaceScript

5.3 Die Schnittstelle

Auch die Schnittstelle wurde bereits im Kapitel Entwurf 4 thematisiert. Auf die konkrete Implementation soll nun hier genauer eingegangen werden. Zunächst wurde die Entscheidung getroffen, dass die KIs auf Java basieren sollen, da hier später eine einfache Nutzung eines Case Based Reasoning-Systems (CBR-System) möglich ist. Grundlegend wurde sich bei der Implementation an der Masterarbeit von Jannis hier Namen einfügen orientiert. Seine Arbeit wendet ein ähnliches Konzept auf einen Ego-Shooter an. Hierbei wurde vor allem seine Java-Klasse CBRSystem.jar genutzt und an das eigene Projekt angepasst.

Die Klasse CBRSystem.jar aktiviert im Grunde einen Server, der vom Client (dem Spiel) angesteuert werden kann und die Anfragen dann durch das Ansteuern der KIs Lösungen für gegebene Situationen erhält und diese wiederum an das Spiel weiterleiten kann. Hierbei findet die Kommunikation zwischen Server und Client mittels JSON wie geplant statt. Die Anfragen, die das Spiel an den Server richtet bestehen immer aus einer ganzen Situation. Das bedeutet, über die Schnittstelle werden die KIs jedes Mal über den Spieler informiert, der am Zug ist, sie enthalten die aktuelle Karte und der Status des Spielers (alle seine relevanten Attribute) werden ebenfalls übermittelt.

```
1 [DataMember]
2 public Map map { get; set; }
3
4 [DataMember]
5 public string player { get; set; }
6
7 [DataMember]
8 public Status playerStatus { get; set; }
```

Listing 5.20: Die relevanten Informationen für eine Situation

5.20 zeigt die Attribute einer Situation. Auffällig ist, dass der Name des Spielers als String gespeichert wird, die Karte und der Status jeweils durch eigene Klassen repräsentiert werden. Die Klasse *Status* bedarf keiner weiteren Erläuterung, da dort lediglich die Attribute des betroffenen Spielers zur Verarbeitung durch JSON vorgehalten werden. Bei der Klasse Map ist dies ähnlich, nur muss ein Weg gefunden werden, die sichtbare Karte als abstrakten Objekt zu formulieren. Zu sehen sind die tatsächlichen Attribute in Listing **??**. Einerseits wird hier ein Enum genutzt, um die Ressourcen der einzelnen Felder verwalten zu können. Andererseits werden drei Listen gespeichert, die die Felder, die Siedlungsbauplätze und die Straßenbauplätze enthalten. Hierbei ist zu beachten, dass es sich bei den gespeicherten Objekten nicht um die GameObjects auf dem Spielfeld handelt. Stattdessen wurden jeweils eigene Klassen definiert, die die nötigen Informationen über jedes Objekt enthalten.

```
1
2 public enum Material
3 {
4     brick,
5     wheat,
6     rock,
7     wood,
8     sheep,
9     sand,
10    ocean
11 }
12
13 [DataMember]
```

```
14 public List<Tile> tiles;
15
16
17 [DataMember]
18 public List<VillageBuildPlace> villageBuildPlaces;
19
20 [DataMember]
21 public List<RoadBuildPlace> roadBuildPlaces;
```

Listing 5.21: Die Attribute der Klasse Map

Die beiden Klassen zur Speicherung von Bauplätzen verfügen dabei über ein Attribut, um die Aktivierung des Bauplatzes abzufragen und lassen sich mittels Methoden aktivieren bzw. deaktivieren. Die Klasse Tile speichert Informationen über die Sechsecke auf dem Spielfeld und braucht hierfür Informationen über die Nummer auf einem Sechseck und die Art des Felder, wofür das Enum genutzt wird.

Durch die angeführten Klassen und Methoden wird das Speichern des sichtbaren Spielfeldes zwar möglich, jedoch muss auch eine Verbindung zwischen den abstrakten Modellen und dem Spiel an sich aufgebaut werden, damit die Informationen gewonnen werden können. Um dies zu gewährleisten, wird bereits bei der Generierung des Spielfelds der erste Schritt unternommen.

```
1 Assets.Scripts.Model.Tile modelTile = new Assets.Scripts.Model.Tile(ints[0],
      (Map.Material)System.Enum.Parse(typeof(Map.Material),
      resources[randomResource].name, true));
2 map.tiles.Add(modelTile);
```

Listing 5.22: Verknüpfung der Erstellung der Karte und den Feldern

Lisiting 5.22 zeigt die Erstellung eines abstrakten Feldes. Dies geschieht für jedes sichtbare Feld, dass kreiert wird. Es verfügt über die gleiche Nummer und die Information über die Art des Feldes. Nach der Erstellung wird es der entsprechenden Liste in der Karte hinzugefügt.

Zur Verlinkung der Bauplätze muss der *GameManager* wiederum näher betrachtet werden. Listing 5.23 zeigt die Methode LateStart(), diese wird nur einmal zu Beginn des ersten Aufrufs der *Update()*-Methode aufgerufen. Hierdurch wird sichergestellt, dass die Start-Methode aller anderen GameObject bereits ausgeführt wurde und es zu keinen Ausnahmen kommt. Konkret muss auf die Erstellung der Karte im *MapGenerator* und auf die physischen Bauplätze gewartet werden. Denn diese Methode übernimmt zunächst die Kontrolle über die Karte des *MapGenerators* und lädt dann aus jedem Bauplatz den enthaltenen abstrakten Bauplatz und speichert diesen in der passenden Liste der Karte. Der abstrakte Bauplatz wird in jedem physischen Bauplatz zur Laufzeit in der Startmethode erstellt, sodass auch hier eine Zuordnung möglich ist. Nachdem dies geschehen ist, darf der erste Zug des ersten Spielers ausgeführt werden, da nun alle Vorbereitungen abgeschlossen wurden. Hierzu wird nun der KI Prozess gestartet und auch der erste Zug für die erste KI.

```
1 private void LateStart()
2 {
3     //Die Karte kann erst jetzt geholt werden, da sie vorher noch nicht
          fertig war.
4     map = mapGenerator.map;
5
6     Transform transform = villagePlaces.GetComponent<Transform>();
```

```
7     for (int i = villagePlaces.GetComponent<Transform>().childCount - 1; i
          >= 0; i--)
8     {
9         Transform child = transform.GetChild(i);
10        map.villageBuildPlaces.Add(child.gameObject.GetComponent<buildVillage>().villagePlace
11    }
12
13    transform = roadPlaces.GetComponent<Transform>();
14    for (int i = roadPlaces.GetComponent<Transform>().childCount - 1; i >=
          0; i--)
15    {
16        Transform child = transform.GetChild(i);
17        map.roadBuildPlaces.Add(child.gameObject.GetComponent<BuildRoad>().roadPlace);
18    }
19
20    //Nun kann der erste Zug von Spieler 1 ausgefÃ¼hrt werden
21    activePlayer.FirstTurn();
22    StartAIProcess();
23    MakeAiTurn(4f);
24 }
```

Listing 5.23: GameManager: LateStart

```
1 private void StartAIProcess()
2 {
3     //String path = "Assets\\HalloWelt.jar";
4     Process foo = new Process();
5     foo.StartInfo.FileName = @"C:\Users\tjark\Desktop\CBRSystem.jar";
6     //foo.StartInfo.FileName = Environment.CurrentDirectory +
          @"\Assets\HalloWelt.jar";
7     foo.StartInfo.Arguments = "" + Constants.PORT;
8     foo.Start();
9     connection = new Connection();
10 }
```

Listing 5.24: GameManager: StartAIProcess

Die Methode *StartAIProcess()* dient dem Starten des Servers, zu dem sie auch die Verbindung aufbaut. Sie ist in Listing 5.24 zu sehen. Die Methode *MakeAITurn* (Listing 5.25) wird anschließend aufgerufen, damit die erste Anfrage an den Server geschickt werden kann. Die Methode selber startet fünf Co-Routinen die jeweils um das angegebene Delay eine Anfrage an den Server schicken (siehe ActivateAi Listing 5.25). Als Antwort wird ein Plan erwartet, der für den aktiven Spieler ausgeführt werden kann. Hierzu muss erwähnt werden, dass Methode und IEnumerator nur für die ersten beiden Züge jedes Spielers verwendet werden. Anschließend werden MakeAiMainTurn und ActivateAiMainTurn verwendet, die ebenso im gleichen Listing **??** zu sehen sind. Der Unterschied besteht darin, dass der IEnumerator zunächst nur einmal gestartet wird und dann rekursiv nur dann erneut anläuft, falls der empfangene Plan keine Anweisung zum Beenden des Zuges enthält. Der Aufruf dieses Konstrukts wird nicht in den ersten beiden Zügen ausgeführt, dafür aber in jedem normalen Zug.

```
1 private void MakeAiTurn(float delay)
2 {
3     StartCoroutine(ActivateAi(delay));
4     StartCoroutine(ActivateAi(delay*2));
5     StartCoroutine(ActivateAi(delay*3));
```

```
6       StartCoroutine(ActivateAi(delay*4));
7       StartCoroutine(ActivateAi(delay*5));
8   }
9
10
11  IEnumerator ActivateAi(float wait)
12  {
13      yield return new WaitForSeconds(wait);
14      Response response = SendToAI(endTurnBtn.interactable,
            rollDiceBtn.interactable);
15      Plan plan = response.plan;
16      plan.StringToActions();
17      UnityEngine.Debug.Log("Plan " + plan.ToString());
18      activePlayer.FulfillPlan(plan);
19  }
20
21  private void MakeAiMainTurn()
22  {
23      StartCoroutine(ActivateAiMainTurn(4f));
24  }
25
26  IEnumerator ActivateAiMainTurn(float wait)
27  {
28      yield return new WaitForSeconds(wait);
29      Response response = SendToAI(endTurnBtn.interactable,
            rollDiceBtn.interactable);
30      Plan plan = response.plan;
31      plan.StringToActions();
32      UnityEngine.Debug.Log("Plan " + plan.ToString());
33
34      bool wantsToEndTurn = false;
35      for (int i = 0; i < plan.actions.Count; i++) {
36          if (plan.actions[i].GetType() == typeof(EndTurn))
37          {
38              wantsToEndTurn = true;
39          }
40      }
41      activePlayer.FulfillPlan(plan);
42      yield return new WaitForSeconds(wait);
43      if (!wantsToEndTurn)
44      {
45          StartCoroutine(ActivateAiMainTurn(5f));
46      }
47  }
```

Listing 5.25: Ausführen einer Anfrage an den Server

Die Verarbeitung der Pläne erfolgt dann in dem PlayerScript eines Spielers selbst. Hierzu muss das Script folgendermaßen erweitert werden (siehe Listing **??**). Die angefügte Methode erlaubt die Ausführungen von geforderten Aktionen eines Plans. Hierzu wird eine for-Schleife genutzt, die den kompletten Plan eines Spielers durchläuft und nach einander jede Aktion versucht durchzuführen. Ob eine Aktion durchgeführt werden kann, hängt davon ab, ob die nötigen Bedingungen erfüllt sind und ob es die Aktion überhaupt gibt. Jede Aktion wird durch eine eigene Klasse repräsentiert, die von der abstrakten Klasse *Aktion* erbt. Bereits verfügbar, ist das Aktivieren und deaktivieren von Bauplätzen, dass Bauen von Straßen und Siedlungen (auf zufälligen Bauplätzen) und das Beenden eines Zuges so-

wie das Würfeln. Welche Aktion ausgeführt wird, hängt davon ab, welcher Klassenname an der entsprechenden Stelle in der Liste des Plans steht.

```
1  public bool FulfillPlan(Plan plan)
2  {
3      for (int i = 0; i < plan.GetActionCount(); i++)
4      {
5          if (plan.actions[i].GetType() == typeof(ActivateVillagePlaces))
6          {
7              gm.villageFocus.hasFocus = !gm.villageFocus.hasFocus;
8              if (gm.villageFocus.hasFocus)
9              {
10                 gm.roadFocus.hasFocus = false;
11             }
12             gm.Update(); //sicher gehen, dass Update mindestens einmal
                   aufgerufen wurde.
13         }
14         else if (plan.actions[i].GetType() == typeof(ActivateRoadPlaces))
15         {
16             ...//Äquivalent zu ActivateVillagePlaces
17         }
18         else if (plan.actions[i].GetType() == typeof(BuildVillage))
19         {
20             //Zurzeit noch zufälliger Bauplatz
21             Transform transform = gm.villagePlaces.GetComponent<Transform>();
22             int rand = Random.Range(0,
                   gm.villagePlaces.GetComponent<Transform>().childCount - 1);
23             Transform child = transform.GetChild(rand);
24             child.gameObject.GetComponent<buildVillage>().Instantiate();
25         }
26         else if (plan.actions[i].GetType() ==
               typeof(Assets.Scripts.CBR.Plan.BuildRoad))
27         {
28
29             ... //Weitesgehend äquivalent zu BuildVillage
30         }
31         else if (plan.actions[i].GetType() == typeof(EndTurn))
32         {
33             if (gm.endTurnBtn.interactable)
34                 gm.OnEndTurn();
35         }
36         else if (plan.actions[i].GetType() == typeof(RollDice))
37         {
38             if (gm.rollDiceBtn.interactable)
39                 gm.OnRollDice();
40         }
41     }
42     return true;
43 }
```

Listing 5.26: PlayerScript: FulfillPlan

Der Plan wird auf Seiten des Servers zusammengestellt. Hierfür sind vor allem drei Klassen relevant: *CBRSystem, PlayerManager* und *Player*. CBRSystem ist für die Verbindung verantwortlich. Sie empfängt Anfragen und sendet dann die Antworten wieder ab. Die Verarbeitung der Anfragen geschieht allerdings in den anderen beiden genannten Klassen. Hier kümmert sich *PlayerManager* um das Aufrufen des richtigen Spielers und entscheidet

wer auf Grund der Anfrage, welcher Spieler angesteuert wird. Außerdem sorgt diese Klasse dafür, dass die Informationen, die der KI zur Verfügung stehen aktuell sind. Die Spieler an sich bilden auf Grundlage dieser Informationen einen Plan, der im PlayerManager zu einer Antwort in Form eines Response-Objektes geformt wird, an die aufrufende Klasse *CBRSystem* zurückgegeben wird und von da aus wie genannt an das Spiel geschickt wird.

5.4 Erweiterung durch den Aufbau von Städten

Beim Einpflegen von Städten in das Spiel handelt es sich um einer Erweiterung, die zwar zu den Muss-Anforderungen gehört, aber nicht in der initialen Beschreibung der Implementation enthalten war. Hier wurde sich zunächst nur auf den fundamentalen Aufbau des Spiels fokussiert. Darin inbegriffen waren dementsprechend nur Straßen, Siedlungen, Spielzüge, das Würfeln und die Ermittlung des Gewinners. Durch die Einführung der Städte in das Spiel erhöht sich die Komplexität für den Spieler, egal ob computergesteuert oder menschlich. Hiermit erhält das Spiel eine interessante strategische Erweiterung, denn für den Fall, dass sowohl Siedlungen, Straßen und Städte gebaut werden können, aber nicht alle, muss sich eine Auswahl durch den Spieler getroffen werden.

Konkret ist die Implementierung der Städte sehr ähnlich zu der der Siedlungen. Einige Aspekte für Siedlungen wurden so für Städte umgesetzt, weshalb eine erneute Auflistung des Quellcodes unsinnig wäre. Dementsprechend wird an geeigneter Stelle auf den abgebildeten Quellcode in den Abschnitten 5.2 und 5.3.

Städte können auf Siedlungen errichtet werden. Sie stellen gewissermaßen ein Upgrade für diese dar. Für Straßen und Siedlungen wurden geeignete Bauplätze durch GameObjects auf der Spielfläche zur Verfügung gestellt. Diese konnten nach Aktivierung des entsprechenden Baumenüs durch den Spieler ausgewählt werden. Wurde ein Klick auf einen Bauplatz ausgeführt, so entstand dort eine Straße oder eine Siedlung. Für die KI wurde dies durch geeignete Anfragen über die Schnittstelle realisiert. Auf gleiche Weise sollte dies auch für die Städte bewerkstelligt werden. Der große Unterschied für die Bauplätze besteht in dem Zeitpunkt ihrer Erstellung. Solche für Straßen und Siedlungen wurden bereits vor Beginn in das Spiel eingefügt, aber deaktiviert gelassen, bis sie zum Einsatz kamen. Für Städte geht dies nicht. Ihre Bauplätze entstehen erst, wenn eine Siedlung gebaut wird. Sie werden an den gleichen stellen errichtet, deshalb bietet sich dies an. Hierzu wird wie bisher der Siedlungsbauplatz unter einer neuen Siedlung gelöscht und aus dem Spiel entfernt. Nun wird zusätzlich ein neuer Bauplatz für Städte erstellt und unter der Siedlung platziert. Dieser neue Bauplatz verfügt über die gleichen Eigenschaften wie solche für eine Siedlung, nur dass er für Städte gilt. Listing 5.27 zeigt, wie die Erzeugung eines Bauplatzes für Städte in den Gamemanager integriert wird. In Listing 5.11 ist in Zeile 12-15 und 33-36 zu sehen, welcher Code konkret ersetzt bzw. erweitert wurde.

```
1  //Siedlungen muessen zwei Strassen von der naechsten entfernt werden, die zu
       nahen Bauplaetze werden zerstoert
2  if (buildVillage != null)
3  {
4      GameObject cityPlaceInstance = Instantiate(cityPlace);
5
6      cityPlaceInstance.transform.position = buildVillage.transform.position;
7      cityPlaceInstance.GetComponent<BuildCity>().player = activePlayer;
8      cityPlaceInstance.GetComponent<BuildCity>().gameManager = this;
9      cityPlaceInstance.GetComponent<BuildCity>().row = tempRow;
10     cityPlaceInstance.GetComponent<BuildCity>().column = tempColumn;
11     cityPlaceInstance.GetComponent<BuildCity>().UpdatePositionForAI();
12     cityPlaceInstance.GetComponent<BuildCity>().tiles = new
           List<GameObject>(buildVillage.GetComponent<Village>().tiles);
```

```
13
14      cityPlaceInstances.Add(cityPlaceInstance);
15      map.cityBuildPlaces.Add(cityPlaceInstance.GetComponent<BuildCity>().cityPlace);
16
17      DestroyVillagePlacesInRadius();
18      buildVillage = null;
19  }
```

Listing 5.27: GameManager: Erstellung des Bauplatzes für Städte

Im nächsten Schritt muss der Zugriff auf den Bauplatz abgespeichert werden, damit dieser auch ansteuerbar bleibt. Dies geschieht in Form einer Liste direkt im GameManager. Außerdem wird eine Liste der Bauplätze für Städte in der Klasse Map aktualisiert und erhält ebenfalls Zugriff. Dieser gilt im GameManager jedoch für das GameObject, während die Map nur auf die abstrakte Repräsentation des Bauplatzes zugreifen kann. Zu sehen ist dieser Vorgang ebenfalls in Listing 5.27. Der Zugriff erfolgt äquivalent zu dem auf Straßen oder Siedlungen.

Neben dem Zugriff muss auch die Aktivierung und der Bau einer Stadt ermöglicht werden. Hierzu hat auch der Städtebauplatz ein passendes OnClick-Event, dass ähnlich wie für den ursprünglichen Bau von Siedlungen funktioniert. Das OnClick-Event lässt sich also auch durch eine geeignete Anweisung der KI aufrufen, was der der Schnittstelle hinzugefügt wurde. Beim Erstellen des Städtebauplatzes erhielt dieser neben einer Position auch Informationen über die Felder, an denen er sich befindet. Diese gehen beim Bau der Stadt auf sie über und anhand dessen kann nach dem Würfeln auch die Ressourcenverteilung für Städte bestimmt werden. Hierzu ist zu erwähnen, dass eine Stadt doppelt so viele Ressourcen bringt wie eine Siedlung. Sie erwirtschaftet in gewisser Weise die Erträge der Siedlung und zusätzlich noch ihre eigenen. Die Siedlung an sich wird beim Bau der Stadt aus dem Spiel entfernt, wie es auch für den Bauplatz gilt. Dies geschieht in der CityBuild-Klasse, die an sich komplett äquivalent zur Version für Siedlungen aufgebaut ist, aber die Löschung des Bauplatzes aus dem Spiel und den nötigen Listen umsetzt (siehe Listing **??**)

```
1  private void OnMouseDown()
2  {
3      cityPlace.row = row;
4      cityPlace.column = column;
5      //Liefert nur eine Siedlung zurueck, wenn die Bedingungen fÃ¼r den
           Spieler erfuellt sind
6      GameObject cityInstance = player.BuildCity(this.transform.position);
7
8      //Die Informationen Ã¼ber anliegende Ressourcenfelder werden aus dem
           Bauplatz auf die Siedlung Ã¼bertragen
9      City cityScript = cityInstance.GetComponent<City>();
10     cityScript.tiles = new List<GameObject>(this.tiles);
11
12     player.cities.Add(cityInstance);
13
14     gameManager.cityPlaceInstances.Remove(this.gameObject);
15     gameManager.map.cityBuildPlaces.Remove(cityPlace);
16     Destroy(this.gameObject);
17     cityPlace = null;
18 }
```

Listing 5.28: Erzeugen einer Stadt im OnClick-Event des Bauplatzes

Abgesehen vom höheren Ressourcenertrag verfügen Städte über fast die gleichen Eigenschaften wie auch Siedlungen. Zwischen ihnen müssen jeweils mindestens zwei Straßen sein, was schon durch den Aufbau auf ehemaligen Siedlungen garantiert wird. Weiterhin können um sie herum Straßen errichtet werden. Allerdings gibt es für sie kein weiteres Upgrade. Eine einmal errichtete Stadt bleibt also an dieser Position bis das Spiel beendet ist. Außerdem erhält der Besitzer der Stadt zwei Siegpunkte für jede Stadt unter seinem Einfluss.

Neben dem Spiel an sich musste die Schnittstelle wie bereits angedeutet auch angepasst werden. Wichtig war, dass es eine zusätzliche Aktion gibt, die die Anweisung des Aktivierens des Baumodus für Städte und für den Bau an sich gibt. Hierzu wurden die gleichen Mechanismen wie für Straßen und Siedlungen erneut für Städte implementiert. Auf Seiten der KI, kann diese jetzt eine Aktion auswählen, die die Bauplätze sichtbar macht und zusätzlich einen Bauplatz als Ziel auswählen. Auf dieser wird dann eine Stadt errichtet. Gleiches wurde auf Seiten des Spielers aufgesetzt, damit solche Anfragen auch verstanden werden. Dementsprechend wurde die Klasse *Plan* erweitert (siehe Listing 5.29). Sie verfügt kann jetzt die genannten Aktionen erkennen und der Liste an Aktionen hinzufügen. Im PlayerScript erfolgt dann die Verarbeitung dieser Liste. Hier wird, wenn die Voraussetzungen erfüllt sind, entweder der Baumodus aktiviert oder eine Stadt errichtet (siehe Listing 5.31).

```
1  if (action.Contains("ActivateCityPlaces"))
2  {
3      this.actions.Add(new ActivateCityPlaces());
4  }
5
6  if (action.Contains("BuildCity"))
7  {
8      string[] splits = action.Split(':');
9      this.actions.Add(new BuildCity(int.Parse(splits[1]),
           int.Parse(splits[2])));
10 }
```

Listing 5.29: Plan: Übersetzung des Strings in Aktionen über Städte

```
1  else if (plan.actions[i].GetType() == typeof(ActivateCityPlaces))
2  {
3      gm.cityFocus.OnMouseDown();
4      gm.Update(); //sicher gehen, dass Update mindestens einmal aufgerufen
           wurde.
5  }
6
7  else if (plan.actions[i].GetType() ==
       typeof(Assets.Scripts.CBR.Plan.BuildCity))
8  {
9
10     Assets.Scripts.CBR.Plan.BuildCity buildcity =
           (Assets.Scripts.CBR.Plan.BuildCity) plan.actions[i];
11     if (gm.map.getCityPlaceByPosition(buildcity.row,
           buildcity.column).gameObject.activeSelf)
12     {
13         gm.map.getCityPlaceByPosition(buildcity.row,
               buildcity.column).gameObject.GetComponent<BuildCity>().Instantiate();
14     }
15 }
```

Listing 5.30: PlayerScript: Umsetzung der Aktionen durch die KI für Städte

Durch die erhöhte Komplexität sind beim Testen auch einige Fehler aufgefallen. Beispielsweise wurde es zum Problem, dass die verwendete KI für einige Situationen keine Lösung wusste. Wenn so ein Fall auftritt, wurde eine leere Antwort verschickt. Mit dieser leeren Antwort konnte das Spiel jedoch nichts anfangen, weshalb eine Exception geworfen wurde. Diese wird nun abgefangen. Die KI erhält in solch einem Fall drei Versuche eine sinnvolle Antwort zu liefern, ansonsten wechselt der Spieler und der Zug des aktuellen Spielers gilt als beendet. Listing **??** zeigt eine Erweiterung der SendToAI-Methode aus der Klasse GameManager, die am Ende Methode eingefügt wird und gewährleistet, dass der KI drei Versuche zur Verfügung stehen. Zusätzlich erfolgt das Abfangen der geworfenen Ausnahme über einen try-catch-Block im JsonParser, der die Bearbeitung einer leeren Antwort verhindert. Dieser liefert lediglich ein leeres Objekt T zurück, sollte der Fall eintreten. Ein weiteres Problem war die Auswahl des Bauplatzes. Die KI konnte nicht wirklich auswählen, wo eine Stadt, Siedlung oder Straße errichtet werden soll. Auch dies ist jetzt gelöst. Die Situation verfügt nun Informationen der Position eines jeden Bauplatzes, sodass die KI eine Auswahl treffen kann. Im String, der die durchzuführenden Aktionen enthält, sind Zeile und Spalte der Bauplatzes durch Doppelpunkt getrennt angegeben und werden im Plan für alle drei Arten von Gebäuden äquivalent verarbeitet. Für Städte ist dies in Listing 5.29 zu sehen. Für Straßen und Siedlungen erfolgt die Verarbeitung auf gleiche Weise.

```
1  else if (plan.actions[i].GetType() == typeof(ActivateCityPlaces))
2  {
3      gm.cityFocus.OnMouseDown();
4      gm.Update(); //sicher gehen, dass Update mindestens einmal aufgerufen
           wurde.
5  }
6
7  else if (plan.actions[i].GetType() ==
       typeof(Assets.Scripts.CBR.Plan.BuildCity))
8  {
9
10     Assets.Scripts.CBR.Plan.BuildCity buildcity =
           (Assets.Scripts.CBR.Plan.BuildCity) plan.actions[i];
11     if (gm.map.getCityPlaceByPosition(buildcity.row,
           buildcity.column).gameObject.activeSelf)
12     {
13         gm.map.getCityPlaceByPosition(buildcity.row,
               buildcity.column).gameObject.GetComponent<BuildCity>().Instantiate();
14     }
15 }
```

Listing 5.31: PlayerScript: Umsetzung der Aktionen durch die KI für Städte

5.5 Initiale KIs

Die initialen KIs wurden auf Seite der Schnittstelle implementiert und dementsprechend in Java umgesetzt. Das hat den Hintergrund, dass im späteren Verlauf die KIs fallbasiert vorgehen sollen und so aufgrund von bereits bekannten Fällen Lösungen für Situationen auf dem Spielfeld finden sollen.

Zunächst wurde der Aspekt der Fallbasiertheit jedoch vernachlässigt und festgelegten Reaktionen seitens Computerspieler gearbeitet werden. Auf diese Weise wurden die Funktionalitäten geprüft und etwaige Fehler konnten behoben werden. Dies erwies sich besonders

während des Einpflegens neuer Funktionalitäten ins Spiel als hilfreich, da so quasi ein Testcase zur Verfügung stand. Zudem musste nicht jeder Test manuell durchgeführt werden, was darüber hinaus Zeit sparte. Die festgelegten Züge wurden den ersten, zweiten und übrige Züge unterteilt, wobei erster und zweiter Zug ähnlich bewältigt wurden und sich so nur die übrigen Züge unterschieden. Wie ein normaler Zug durchgeführt wurde, ist in Listing (hier einfügen) zu sehen. Zu erkennen ist, dass jede auszuführende Aktion durch eine passend benannte Aktion repräsentiert wird. Jeder dieser Methoden fügt dem Plan, der zurück ans Spiel geschickt wird, eine geeignete Aktion in Form eines Strings hinzu. Dieser String kann dann auf Seiten des Spiels wie in Abschnitt 5.3 und 5.4 interpretiert werden.

```
1  if (allowedToRollDice) {
2    RollDice();
3    first = true;
4  } else if (HasResourcesForCity() || HasResourcesForVillage() ||
       HasResourcesForRoad()){
5    if (HasResourcesForCity() && first && map.cityPlacesExist()) {
6      triesCity++;
7      first = false;
8      ActivateCityPlaces();
9    } else if (HasResourcesForCity() && !first && map.cityPlacesExist() &&
         triesCity <= 3) {
10     triesRoad = 0;
11     triesCity = 0;
12     triesVillage = 0;
13     first = true;
14     BuildCity(map.getRandomCityPlace());
15     ActivateCityPlaces();
16   } else if (HasResourcesForVillage() && (first || triesVillage == 0)) {
17     triesVillage++;
18     first = false;
19     ActivateVillagePlaces();
20   } else if (HasResourcesForVillage() && !first && map.villagePlacesActive()
         && triesVillage <= 3) {
21     triesRoad = 0;
22     triesCity = 0;
23     triesVillage = 0;
24     first = true;
25     BuildVillage(map.getRandomVillagePlace());
26     ActivateVillagePlaces();
27   } else if (HasResourcesForRoad() && (first || triesRoad == 0)) {
28     triesRoad++;
29     first = false;
30     ActivateRoadPlaces();
31   } else if (HasResourcesForRoad() && !first && triesRoad <= 3) {
32     triesRoad = 0;
33     triesCity = 0;
34     triesVillage = 0;
35     first = true;
36     BuildRoad(map.getRandomRoadPlace());
37     ActivateRoadPlaces();
38   }
39 } else {
40   EndTurn();
41 }
```

Listing 5.32: Java-Klasse Player: Erstellen eines normalen Zuges zum Testen

Neben dem Hinzufügen der Aktionen zum Plan zeigt Listing (5.32) ebenfalls, wie verschiedene Aktionen zum Testen priorisiert werden. Sollte der Bau einer Stadt möglich sein, so wird immer diese gebaut. Wenn keine Stadt, aber eine Siedlung gebaut werden kann, so wählt die KI diese Option. Falls auch dies nicht funktioniert, weil entweder zu wenige Ressourcen gesammelt wurden oder kein passender Bauplatz verfügbar ist, so versucht die KI eine Straße zu bauen. Der Bau einer Straße ist wie bereits bekannt auch nicht unter allen Umständen möglich, weshalb in solchen Fällen der Zug beendet werden soll.

Es kann jedoch auch vorkommen, dass die KI auf eine vorliegende Situation keine Antwort findet. Dann schickt sie einen leeren Plan an das Spiel zurück, mit dem es nichts anzufangen weiß, weshalb nach drei ungültigen Antworten durch die KI automatisch der nächste Spieler am Zug ist.

Soviel zu den festgelegten Aktionen zum Testen der Funktionalitäten des Spiels. Darüber hinaus soll auch die Fallbasiertheit Einzug in die Entwicklung erhalten, was die KIs aufwertet und gleichzeitig für die Erfüllung der Kann-Anforderung *Das Spiel kann über initiale KIs verfügen.* sorgt. Hierzu muss eine Umstellung der Reaktionen der KIs erfolgen. Es reicht nicht mehr über Verzweigungen Reaktionen hervorzurufen. Stattdessen muss es eines Wissensbasis geben, auf die zurückgegriffen werden kann.

```
1  //Konkreter Fall fÃ¼r den ersten Zug, wenn ein kostenloses Dorf gebaut
       werden darf
2  Status firstTurnVillage = new Status();
3
4  //Wichtig
5  firstTurnVillage.villagePlacesActive = true;
6  firstTurnVillage.isFirstTurn = true;
7  firstTurnVillage.isSecondTurn = false;
8  firstTurnVillage.freeBuild = true;
9  firstTurnVillage.freeBuildRoad = false;
10
11
12 //Unwichtig
13 firstTurnVillage.victoryPoints = 0;
14 firstTurnVillage.longestRoad = 0;
15 firstTurnVillage.hasLongestRoad = false;
16
17 firstTurnVillage.bricks = 0;
18 firstTurnVillage.wheat = 0;
19 firstTurnVillage.stone = 0;
20 firstTurnVillage.wood = 0;
21 firstTurnVillage.sheep= 0;
22
23 //firstTurnVillage.villages = new ArrayList<~>();
24 //firstTurnVillage.roads = new ArrayList<~>();
25
26 firstTurnVillage.isAbledToEndTurn = false;
27 firstTurnVillage.allowedToRollDice = false;
28
29 plan = "BuildVillage;ActivateVillagePlaces";
30 status.put(firstTurnVillage, plan);
```

Listing 5.33: Java-Klasse DefaultCases: Beispiel

Für die Erstellung der Wissensbasis dient zunächst die Klasse *DefaultCases*, die in Java geschrieben wurde und grundlegende Fälle enthält, um der KI etwas Intelligenz zu verleihen. Hierbei wird auf den Status eines Spielers zugegriffen, der alle wichtigen Attribute

enthält. Es werden Fälle für jede grundlegende Situation erstellt. Ein Beispielfall ist in Listing (5.33) zu sehen. Dieser Fall beschreibt, wie vorgegangen wird, wenn im ersten Zug bereits die Bauplätze für Siedlungen aktiviert wurden und nun der Bau einer solchen erfolgen soll. Dieser Fall steht stellvertretend für alle anderen, weil alle Fälle auf gleiche Weise aufgebaut sind und sich nur in der Ausprägung der Attribute unterscheiden.

Die initialen Fälle dienen dazu, die Voraussetzungen für einen Plan festzuhalten. Das Ziel besteht darin, einen Plan nur auszuführen, wenn das Spiel dies auch zulässt. An sich wird durch die Standardfälle die bereits festgelegte Folge an Reaktionen aus Listing (5.32) simuliert. Der Unterschied besteht allerdings darin, dass nun ganz einfach neue Fälle hinzukommen können, die die Wissensbasis erweitern und so die Performanz der KIs erhöhen.

Zurzeit würde das Hinzufügen speziellerer Fälle allerdings keinen Effekt erzielen, da zunächst einmal auf die Wissensbasis zugegriffen werden muss. Hierzu müssen die tatsächlichen Situationen mit den Fällen abgeglichen werden und anschließend kann anhand der wichtigen Attributen der Plan ausgewählt werden. Dieser Prozess ist wie folgt implementiert, wobei sich eng an der Vorgehensweise von Jannis Hillmann, Jobst-Julius Bartels und Marcel Kolbe orientiert wurde.

In der Klasse *CBREngine* erhält jedes Attribut, abhängig von der aktuellen Situation, eine Gewichtung. Diese ist für die Ermittlung des passenden Plans insofern relevant, indem abhängig von dieser Gewichtung unterschiedlich viel Wert auf die Attribute von Status gelegt wird. So ist es beispielsweise sinnhaft, dass im ersten Zug eines Spielers kein Wert auf die Menge an verfügbaren Rohstoffen gelegt wird. Andererseits, ist es in normalen Zügen irrelevant, ob ein kostenlose Siedlung oder Straße gebaut werden darf, weil diese Option gar nicht zur Verfügung stehen kann. Hier wäre es wiederum wichtig auf die die verfügbaren Rohstoffe zu schauen.

In diesem Schritt kann auch die Intelligenz einer KI beeinflusst werden. Neben einer ausreichend großen Wissensbasis, die möglichst viele Situationen umfasst und gute Pläne für diese bereithält, ist wichtig, wie überhaupt eine Auswahl getroffen wird. Beispielsweise kann hier entschieden werden, ob Siedlungen über Städte und gar Straßen vorrangig gebaut werden sollen.

Eine mögliche Strategie wäre es besetzte Ressourcenfelder möglichst auszureizen und mit wenig Platz auf der Karte, viele Siegpunkte zu erwirtschaften. Hierzu sollten möglichst wenige Straßen zwischen zwei Städten bzw. Siedlungen existieren und vorhandene Siedlungen sollten schnellstmöglich zu Städten ausgebaut werden. Bei dieser Strategie ist das Priorisieren von Holz und Lehm zu Beginn des Spiels wichtig, um Momentum im Spiel aufzubauen. Allerdings muss nun auch viel Wert auf Getreide und Stein gelegt werden, aufgrund der hohen Kosten für die Städte. Die Implementation ist in Listing 5.34 zu sehen.

```
1  public void updateAttributeWeight(Status status) {
2
3    if (status.isFirstTurn) {
4      isFirstTurn = 1000;
5      isSecondTurn = 0;
6      hasFreeBuild = 900;
7      hasFreeBuildRoad = 900;
8      villagePlacesAct = 800;
9      roadPlacesAct = 800;
10
11   } else if (status.isSecondTurn) {
```

```
12      ... //Äquivalent zu isFirstTurn
13    } else if(status.allowedToRollDice) {
14
15      isAllowedToRollDice = 1000;
16
17    } else if(!status.allowedToRollDice) {
18
19      flush();
20      if (status.hasResourcesForCity()) {
21        if (!status.getMap().cityPlacesActive()) {
22          isAbledToBuildCity = 100;
23          cityPlacesAct = 1000;
24
25        } else if (status.getMap().cityPlacesActive()) {
26          isAbledToBuildCity = 1000;
27        }
28
29      } else if (status.hasResourcesForVillage()) {
30        if (!status.getMap().villagePlacesActive()) {
31          isAbledToBuildVillage = 100;
32          villagePlacesAct = 1000;
33
34        } else if (status.getMap().villagePlacesActive()) {
35          isAbledToBuildVillage = 1000;
36        }
37
38      } else if (status.hasResourcesForRoad()) {
39        if (!status.getMap().roadPlacesActive()) {
40          isAbledToBuildRoad = 100;
41          roadPlacesAct = 1000;
42
43        } else if (status.getMap().roadPlacesActive()) {
44          isAbledToBuildRoad = 1000;
45          roadPlacesAct = 100;
46        }
47
48      } else {
49        isAbledToEndTurn = 1000;
50        ... //Restliche auf 0 setzen
51      }
52    }
53  }
```

Listing 5.34: Java-Klasse CBREngine: Konfiguration KI-Spieler1

Eine konträre Strategie für Spieler 2 zu finden, sollte für einen besonders interessanten Vergleich sorgen. Daher soll Spieler 2 sich darauf konzentrieren, möglichst viel Kontrolle über die Karte zu erhalten. Hierzu muss die KI viele Straßen und Siedlungen bauen, damit sie einen großen Anteil der theoretisch verfügbaren Bauplätze einnehmen kann. Städte sind bei solch einer Strategie eher zu vernachlässigen. Es sei denn, die KI kann dadurch ihr Ressourceneinkommen signifikant steigern. Außerdem sollte sich vor allem auf Lehm und Holz als Ressourcen konzentriert werden, da beide sowohl für Straßen als auch Siedlungen benötigt werden. Allerdings muss besonders zu Beginn des Spiels ebenfalls auf den Zugang zu Getreide und Schafen geachtet werden. Ansonsten wäre eine Expansion auf der Spielfläche, die den Gegner einschränkt gar nicht möglich. Diese Strategie soll Spieler 2 verfolgen. Die Konfiguration dieses Spielers ist nicht hier im Text sondern aus Platz Gründen im

Anhang abgedruckt. Allgemein ist der Aufbau äquivalent. Allerdings werden die Gewichtungen anders initialisiert. Nachdem für die ersten beiden Züge die gleiche Initialisierung erfolgte, wird für die normalen Züge anders vorgegangen. Wenn nicht gewürfelt werden darf, kommt die Strategie des Spielers zum Tragen. Statt zunächst auf die Bauerlaubnis für Städte, wird stattdessen auf Dörfer geprüft. Denn jedes neue Dorf vergrößert den Einflussbereich des Spielers, was der gewünschten Strategie entspricht. Zusätzlich wird auch der Bau von Straßen über den von Städten gestellt. Dies gewährleistet, dass möglichst viele Objekte auf dem Spielfeld errichtet werden.

Eine interessante Erweiterung für eine weitere Strategie wäre ein Kombination beider Herangehensweisen. Statt einer einfachen Priorisierung könnte einbezogen werden, welcher Schritt die größten Ressourcenertrag in kommenden Runden verspricht. Die auf Städte konzentrierte KI würde Steine und Weizen priorisieren, die auf Expansion gerichtete eher Lehm und Holz. Eine KI die keine klare Präferenz an Gebäuden hat, könnte möglicherweise mehr Siegpunkte erlangen.

Damit die beschriebene Strategie auch umgesetzt werden kann, müssen priorisierte Ressourcen auch Einzug in die Implementation erhalten. Hierzu werden die Fälle um das Attribut *preference* erweitert. Dieses Attribut erlaubt es, eine priorisierte Ressource mit in das Retrieval einfließen zu lassen. Dadurch kann schlussendlich eine strategischere Entscheidung getroffen werden. Die Priorisierung von Ressourcen ist nämlich überall dort Teil der Standardfälle, die als Plan den Bau einer neuen Siedlungen enthalten.

Zur Umsetzung zweier verschiedener Strategien, muss selbstverständlich auch eine getrennte Berechnung der Präferenz erfolgen. Diese wirkt direkt in der Java-Klasse *Status* vorgenommen. Diese wurde um zwei Methoden mit den Namen *CalculatePreferencePlayer1* und *CalculatePreferencePlayer2* erweitert. In Listing **??** ist allerdings nur die erstere abgebildet. Die übrige ist im Anhang zu finden. Jedoch reicht zum Verständnis die Betrachtung der einen. Denn beide gehen ähnlich vor: Es wird aufgrund der vorhandenen Mengen an Ressourcen eine Präferenz berechnet. Das Listing (hier einfügen) zeigt speziell, dass zunächst einmal zwischen erstem, zweitem und den übrigen Zügen unterschieden wird. Denn für beide Spieler gilt: Sie bauen die kostenlose Siedlung aus dem ersten Zug an einem Ressourcenfeld, dass Holz liefert und im zweiten Zug bauen sie an einem Feld mit Lehm. Dies ist so entscheidend, weil zum aktuellen Stand des Spiels kein späteres tauschen möglich ist. Demnach sollten die KIs versuchen, schnellst möglich Zugang zu wichtigen Ressourcen zu erhalten. Da Holz sowie Lehm für Siedlungen und Straßen benötigt werden, wird sich zunächst auf diese fokussiert.

Anschließend beginnt sich jedoch die Strategie heraus zu kristallisieren. Die gezeigte Methode setzt immer, wenn der Bau einer Siedlung möglich ist, die Ressource als Präferenz, die am wenigsten zur Verfügung steht. Sollten allerdings zwei Ressourcen gleich häufig verfügbar sein, so wird die wichtigere (meist Holz oder Lehm) gewählt. Wenn die Entscheidung zwischen Getreide und Schafen getroffen werden muss, dann hat das Getreide die Oberhand. Stein wird nie präferiert, da es zur Expansion nicht entscheidend ist und weder zum Bau von Siedlungen noch zu dem von Straßen gebraucht wird.

```
1 public String calculatePreferencePlayer2() {
2   String preference = "";
3   if (isFirstTurn && freeBuild) {
4     preference = "wood";
5   } else if (isSecondTurn && freeBuild) {
6     preference = "brick";
7   } else if (isAbledToBuildCity) {
8     preference = "";
```

```
9   } else if (isAbledToBuildVillage) {
10    if (bricks < wood || bricks == wood) {
11      if (bricks < sheep || bricks == sheep) {
12        if (bricks < wheat || bricks == wheat) {
13          preference = "brick";
14        } else {
15          if (wheat < sheep || wheat == sheep) {
16            preference = "wheat";
17          } else {
18            preference = "sheep";
19          }
20        }
21      } else {
22        if (sheep < wheat) {
23          preference = "sheep";
24        } else {
25          preference = "wheat";
26        }
27      }
28    } else {
29      if (wood < sheep || wood == sheep) {
30        preference = "wood";
31      } else {
32        if (wheat < sheep || wheat == sheep) {
33          preference = "wheat";
34        } else {
35          preference = "sheep";
36        }
37      }
38    }
39  } else {
40    preference = "";
41  }
42  this.preference = preference;
43  return preference;
44 }
```

Listing 5.35: Java-Klasse Status: CalculatePreferencePlayer2

Durch die obige Erklärung, sollte die Berechnung der Präferenz klar geworden sein. Diese findet auf ähnliche Weise für Spieler 1 statt. Dieser präferiert in normalen Zügen allerdings immer Getreide oder Stein, was ihn sehr anfällig für Fehler macht.

5.6 Anpassungen

5.7 Umsetzung der Implementation

5.8 Ergebnisse

6 Ergebnisse

7 Evaluation

Zu Beginn dieser Bachelorarbeit wurden Anforderungen an das Produkt gestellt, die in diesem Kapitel überprüft werden sollen. Es wurde zwischen Muss-, Soll- und Kann-Anforderungen unterschieden, wobei die Muss-Anforderungen zwingend Teil der Umsetzung sein müssen. Die beiden anderen Varianten dienen als Erweiterung, um das Produkt aufzuwerten. Daher sollen zunächst die Muss-Anforderungen betrachtet und analysiert werden. Erst im Anschluss daran, wird über die Erweiterungen gesprochen. Neben den Anforderungen sollen in diesem Kapitel auch Probleme behandelt werden, die während der Umsetzung auftraten. Abschließend wird noch auf Verbesserungen/Erweiterungen und Strategien für eine intelligentere KI eingegangen.

7.1 Muss-Anforderungen

Die erste lautete, dass das Spielfeld nach den Regeln des Originals aufgebaut sein muss. Diese Anforderung ist erfüllt worden. Die sechseckigen Felder werden zufällig verteilt, den Regeln entsprechend angeordnet und die darauf liegenden Zahlen werden in einer vorgegeben Reihe auf diesen Feldern verteilt. Ebenfalls gilt als erfüllt, dass die Wüste als einziges Feld keine Zahl erhält. Im echten Brettspiel ist das Spielfeld von Wasser umrandet, auf diese Erweiterung wurde in der Implementation verzichtet, da diese nur einen Mehrwert bietet, wenn auch gleichzeitig der Handel an Häfen implementiert würde.

Anforderung zwei bis vier aus dem Kapitel 4 zum Spiel fordern das Erlauben vom Bau von Straßen, Siedlungen und Städten. All diese Objekte stehen zum Bau zur Verfügung, wie die Abbildung 7.1 zeigt. Die Abbildung zeigt speziell einen Spielstand bei dem bereits alle drei Objekte auf dem Spielfeld errichtet wurden. Damit sind auch diese drei Anforderungen erfüllt.

Abbildung 7.1: Bau von Städten, Siedlungen und Straßen

Die nächste Anforderung handelt von der Ermittlung eines Gewinners am Ende des Spiels. Diese wurde umgesetzt, indem jeder Spieler über eine Anzahl an Siegpunkten verfügt. Wenn diese einen bestimmten Wert (Standard: 10) erreicht haben, so gilt das Spiel als

beendet und der Spieler, der diesen Wert zuerst erreicht hat, wird auf dem GameOver-Bildschirm als Gewinner ausgegeben. Außerdem ist auf diesem Bildschirm ein Button zu sehen, der den Start eines neuen Spiels ermöglicht. Im Hintergrund werden zudem alle Handlungen gestoppt und zurückgesetzt, sodass bei einem Neustart auch wirklich ein neues Spiel beginnt. Der GameOver-Bildschirm ist in Abbildung 7.2 zu sehen.

Abbildung 7.2: Ausgabe des Gewinners auf dem Bildschirm

Da Siedler von Catan ein rundenbasiertes Spiel ist, muss im Spiel jedem Spieler ein Zug pro Runde zugeschrieben werden. In den Anforderungen wurde unter sechstens gefordert, dass ein solcher Zug immer mit dem Würfeln beginnt, nachdem Ressourcen verteilt werden und anschließend soll der Spieler bauen können. Diese Anforderung wurde genau so im Spiel umgesetzt. Zunächst muss der Spieler den Würfelbutton betätigen, ansonsten kann er keine andere Aktion ausführen. Erst danach ist es ihm möglich weitere Aktionen auszuführen. Bezogen auf das Bauen, ist dies nur möglich, wenn der Baumodus durch den Spieler aktiviert wird und dies wird wiederum nur erlaubt, wenn er genügend Ressourcen für den gewählten Baumodus hat. So muss der Spieler zum Bau von Straßen beispielsweise mindestens ein Lehm und ein Holz besitzen. Diese beiden Abfragen sind relativ deutlich, doch kann auch der Fall auftreten, dass die Voraussetzungen zwar erfüllt sind, aber der Spieler dennoch keinen Bau ausführen kann. Dieser Fall kann zum Beispiel auftreten, wenn kein geeigneter Bauplatz zur Verfügung steht. Dies könnte beispielsweise auftreten, wenn bereits alle Siedlungen zu Städten aufgewertet wurden und der Spieler dennoch den entsprechenden Modus aktiviert. In solch einem Fall, darf wird natürlich keine weitere Stadt errichtet, denn der Baumodus aktiviert lediglich, die Bauplätze und diese sind bereits alle gelöscht, weshalb es hier zu keinem Problem kommt. Somit ist auch diese Anforderung komplett erfüllt.

Neben den notwendigen Anforderungen, die das Spiel erst ermöglichen, gibt es auch Regeln, die zwar zum Spiel gehören, aber deren Nichtumsetzung das Spiel nicht unspielbar machen würde. Eine dieser Anforderungen ist, ist die siebte. Diese besagt, dass zwischen Städten bzw. Siedlungen mindestens zwei Straßen liegen müssen. Implementiert wurde dies, indem die direkt, an eine Siedlung oder Stadt angrenzenden, Bauplätze für weitere Siedlungen gelöscht werden. In der Praxis erfolgt dies, indem ein Vektor um eine Siedlung oder Stadt gelegt wird und jeder Bauplatz für Siedlungen im Umkreis entfernt wird. Dies

erfüllt jedoch nur einen Teil des Problems. Abgesehen von den ersten beiden Zügen, darf der Spieler auch nur eine bereits gebaute Straße gebaut werden. Hierzu macht sich das Spiel erneut Entfernungen durch Vektoren zunutze. Denn wenn ein Baumodus aktiviert wird, aktiviert dieser nicht zwangsläufig alle Bauplätze, die nicht gelöscht wurden. Stattdessen aktiviert der entsprechende Baumodus für Straßen oder Siedlungen nur solche Plätze, die direkt an eine bereits bestehende Straße angrenzen. Beide Varianten kombiniert, erfüllen insgesamt die Anforderung, weil keine Bauplätze nah genug an bestehenden Siedlungen existieren und gleichzeitig Straßen sowie Siedlungen nur direkt an Straßen gebaut werden können. Zudem gilt für Städte das Gleiche wie für Siedlungen, da ein Bauplatz für eine Stadt nur dort errichtet wird, wo schon eine Siedlung steht. Dadurch wird übrigens auch die Anforderung acht erfüllt, die fordert, dass das Aufwerten von Siedlungen von Siedlungen auf Städte im Spiel enthalten sein muss.

Die Umsetzung der letzten Muss-Anforderung stellt auch gleichzeitig die komplizierteste Implementation dar. Denn hier wird gefordert, dass das Spiel berechnet, welcher Spieler die längste Straße errichtet hat und dass dieser entsprechend mit Siegpunkten belohnt wird. Eine Straße in Siedler von Catan ist so definiert, dass einzelne aneinander liegende Straßen eine längere Straße bilden. Eine Straße aus einem Stück hat somit die Länge eins. Jedes weitere Stück der Straße, dass direkt angrenzt erhöht die Länge um ebenfalls um eins. Nach dieser Regel ist es somit auch möglich eine Kreuzung zu errichten. Eine solche Kreuzung hat demnach die Länge von drei und nicht von zwei. Die Umsetzung dessen erscheint zunächst schwieriger als die der anderen Anforderungen zu werden, lässt sich aber mit bereits bekannten Methoden implementieren. Denn die Vektoren aus Unity sind hier wieder von Nutzen. Konkret verfügt jede Straße auf dem Spielfeld über einen Punkt im Raum, dessen Vektor zur Bestimmung der Entfernung zu anderen Straßen (wie bei den Bauplätzen) genutzt werden kann. Hierzu muss für jeden Spieler eine Prüfung aller seiner Straßen erfolgen. Im Quelltext dienen hierzu zwei Schleifen, die äußere durchläuft alle Straßen des Spielers und in der inneren wird jede Iteration geprüft, ob eine Straße im Einzugsgebiet liegt. Gleichzeitig sorgt ein rekursives Vorgehen dafür, dass diese Straße ebenfalls als Ausgangspunkt genutzt wird. Jede Rekursion wiederum liefert die Länge der Straße bis jetzt zurück. Somit ergibt die Tiefe der Rekursion die Länge einer Straße. Da durch das Durchlaufen aller Straßen zum Schluss jede Straße als Beginn genutzt wurde, ist sichergestellt, dass eine der bestimmten Längen die längste Straße beschreibt und wenn das Maximum dieser Längen nun mit dem der anderen Spieler verglichen wird, kann die längste Handelsstraße dem passenden Spieler zugewiesen werden, wodurch die Anforderung erfüllt ist.

Aufgrund der obigen Evaluation lässt sich feststellen, dass alle Muss-Anforderungen, die gestellt wurden, erfüllt sind. Dadurch sind im folgenden Abschnitt die Soll- und Kann-Anforderungen zu bewerten. Hier war es nicht das Ziel alle zu Erfüllen, sondern nur eine oder einige, um das Produkt über die Mindestanforderungen hinaus aufzuwerten.

7.2 Soll- und Kann-Anforderungen

Die nicht erfüllten Anforderungen lassen sich schnell aussortieren. So wurde die Funktion von Ereigniskarten und das Ausspielen dieser Karten nicht implementiert. Auch der Dieb, der Ressourcen blockiert und deren Diebstahl ermöglicht erhielt keinen Einzug ins Spiel. Zudem werden auch keine Gewinnwahrscheinlichkeiten ausgeben, auch der Handel zwischen Spielern ist nicht möglich und es dürfen auch maximal zwei Spieler gleichzeitig das Spiel spielen. Die damit ganz oder teilweise erfüllten Bedingungen belaufen sich damit auf die übrigen, die wie folgt umgesetzt wurden.

Es ist möglich, dass reale Spieler das Spiel spielen. Hierzu muss nur eine Einstellung zu Beginn des Spiels im Startmenü vorgenommen werden. Diese ist schon so implementiert, dass auch die nächste Anforderung, die einen KI-Spieler und einen realen Spieler ermöglichen soll, erfüllt wird. Denn für Spieler 1 und Spieler 2 kann jeweils ein Haken (KI an) gesetzt werden. Ist das Kästchen angehakt, so wird der entsprechende Spieler durch die KI gesteuert, ansonsten muss ein menschlicher Spieler die Züge ausführen.

Darüber hinaus gilt eine weitere Kann-Anforderung als erfüllt. Es werden nämlich initiale KIs bereitgestellt. In Abschnitt 5.5 wird darauf näher eingegangen. Daher dieser Punkte hier verkürzt behandelt werden. Es bleibt lediglich zu auszusagen, dass diese KIs grundlegende Strategien verfolgen und diese auch über das Spiel hinweg gleich bleiben. So ändert die KI beispielsweise nicht ihre Strategie, wenn sie nur noch wenige Siegpunkte braucht, wobei dies natürlich durch eine Erweiterung der Fallbasis möglich wäre. Durch die Bereitstellung der initialer Fälle und der Gewichtung von Attributen sind jedoch erweiterbare KIs Teil des Produktes geworden, bei denen nun die Fallbasis ausgetauscht oder erweitert werden kann, was für intelligenteres Verhalten sorgen kann. Sie unterscheiden sich damit fundamental von den in Abschnitt 5.5 beschriebenen Testreaktionen der KI.

7.3 Anforderungen an die Schnittstelle

Neben direkten Anforderungen an das Spiel an sich wurden auch spezielle an die Schnittstelle gestellt, die die Qualität weiter steigern sollen.

Bei der ersten wird gefordert, dass die KIs ohne großen Aufwand ausgetauscht werden können müssen. Dies ist durch die fallbasierte Umsetzung der initialen KIs ermöglicht. Die Fallbasis kann einfach in der Projektdatei angepasst werden. Lediglich die Standardfälle sind durch den Quelltext abgedeckt. Hier an also beliebig angepasst und ausgetauscht werden, was die Anforderung erfüllt.

Die zweite Anforderung sagt aus, dass die KIs in unterschiedlichen Programmiersprachen umgesetzt sein dürfen. Dieser Punkt ist kein Teil der Standardumsetzung und wurde auch nicht im Detail verfolgt. Allerdings kann er dennoch als erfüllt betrachtet werden, denn als Kommunikationsmittel wird JSON benutzt. Wenn nun zusätzlich im Quellcode des Spiels das aufgerufene Programm ausgetauscht wird, so kann auch eine andere Programmiersprache die Rolle von *CBRSystem.jar* übernehmen.

Die nächste Anforderung wurde schon beim Beschreiben der Anforderungen fürs Spiel mitbetrachtet. Da zum Start der Spiels ausgewählt werden kann, ob Mensch-Mensch, Mensch-KI oder KI-KI gegeneinander spielen,muss die Schnittstelle zwangsläufig das Antreten von zwei computergesteuerten Spielern ermöglichen.

Die letzte Anforderung erfordert erneut den größten Erläuterungsbedarf. Diese möchte nämlich, dass die KI einem menschlichen Spieler gegenüber weder bevorteilt oder benachteiligt wird, was durchaus vorkommen könnte. Hierfür ist nämlich entscheidend, dass Mensch und Maschine die gleichen Informationen erhalten, obwohl sie nicht über die gleiche Wahrnehmung verfügen. Die KI muss also eine Situationsbeschreibung erhalten, die dem gleicht, was ein menschlicher Spieler auf dem Spielfeld sehen würde, wenn er dran ist. Außerdem ist der Zeitpunkt der Informationsübermittlung relevant, weil kein konstanter Fluss erfolgen kann, wie bei der Betrachtung eines Bildschirms durch einen Menschen. Somit sollte die KI immer dann eine neue Situationsbeschreibung erhalten, wenn sich am Zug ist und Situation sich so signifikant geändert hat, dass davon auszugehen ist, dass Sie andere Aktionen ausführen möchte, als Sie vorher ausgewählt hat. In der Implementation wird dies umgesetzt, indem zum Beginn eines Zuges eine Anfrage, die eine Situationsbeschreibung enthält, an die KI geschickt wird. Auf diese kann Sie mit einem Plan reagieren.

Dieser besteht aus einer Liste an Anweisungen, die die KI ausführen möchte. Das Spiel führt die Aktionen nach und nach im Namen des Spielers aus, sofern dies möglich ist. Wenn alle Aktionen ausgeführt wurden, wird die neue Situation der KI erneut übermittelt und sie kann weitere Aktionen durchführen. Dies geschieht so lange, bis im Plan der Wunsch nach dem Beenden des Zuges auftritt. Hierdurch endet der Zug der KI sofort und der nächste Spieler ist an der Reihe. Einen Ausnahmefall stellt ein leerer Plan dar. Wenn die KI auf eine gegebene Situation keine Antwort weiß, so kann es vorkommen, dass sie eine leere Antwort sendet. Dieser Fall wurde in Abschnitt 5.4 schon im Detail beschrieben. Das dort beschriebene Vorgehen gewährleistet, dass durch einen Fehler der KI, ihr Zug nicht abrupt beendet wird. Stattdessen erhält Sie erneut die Chance auf die gegebene Situation angemessen zu reagieren. Antwortet sie allerdings weiterhin unsinnig, so wird ihr Zug dennoch beendet und der nächste Spieler ist an der Reihe.

Allgemein kann durch dieses Vorgehen die gleiche Informationsverfügbarkeit simuliert werden, die auch einem Menschen zugänglich wäre. Zwar kann der Mensch dauerhaft auf seine Rohstoffe schauen, das Spielgeschehen auf der Spielfeld im Blick behalten und seine Strategie ändern und die KI ist dazu nicht in der Lage, doch muss dies auch nicht der Fall sein. Es reicht wenn die KI ihre Handlungen bei einer Situationsänderung überdenken darf. Denn der Mensch wird seine Strategie wahrscheinlich auch nur dann überdenken, wenn sich etwas an seiner Situation ändert. Für die Ki wurde dies äquivalent umgesetzt. Zu Beginn eines Zuges darf die KI ihren Zug planen und nach Ausführung des Plans eventuell einen neuen Umsetzen. Hierdurch lässt sich insgesamt schließlich, dass die genannte Anforderung an die Schnittstelle erfüllt wurde und ein Grundsatz von Fairness zwischen dem menschlichen und dem computergesteuerten Spieler etabliert wurde.

Abschließend kann aufgrund der obigen Analyse die Aussage getroffen werden, dass auch für die Schnittstelle alle nötigen Anforderungen hinreichend umgesetzt wurden. Lediglich an der Verwendung anderer Programmiersprachen außer Java und C# zur Implementierung weiterer KIs Bedarf es an mehr Aufwand. Hierbei bleibt aber dennoch zu sagen, dass es möglich ist, sofern die Schnittstelle JSON genutzt wird das angesteuerte Programm im Quellcode des Spiels vermerkt wird.

7.4 Aufgetretene Probleme

Viele Softwareprojekte laufen nicht so ab, wie es zuvor geplant wurde. Häufig treten Probleme auf, die vorher nicht bedacht wurden und dann die Entwicklung verzögern oder sogar zu einer angepassten Funktionsweise führen. Dies war auch im Rahmen der Entwicklung der Software zu dieser schriftlichen Ausarbeitung der Fall. Einige Hindernisse traten auf, die zunächst einmal bewältigt werden mussten, bevor andere Funktionalitäten in Betracht gezogen wurden. In diesem Abschnitt sollen diese Probleme behandelt und auch deren Lösung kurz erläutert werden. Dabei wird selbstverständlich nicht auf jeden einzelnen Fehler eingegangen, der während der Entwicklung auftrat, aber auf alle größeren Hindernisse. Zunächst sollen solche behandelt werden, die gelöst wurden. Anschließend auch auf die, die bestehen bleiben.

7.4.1 Behobene Probleme

Die Integration der KI und insbesondere der Schnittstelle erwies sich aufgrund des eigenen Vorgehens als schwierig. Hier hätten rückblickend wahrscheinlich mehrere Stunden eingespart werden können. Denn zunächst konzentrierte ich mich nur auf eine laufende Version des Spiels. Es wurde völlig vernachlässigt, wie die Schnittstelle und damit auch die KIs später noch eingebunden werden können. Hierdurch entstanden vor allem konzeptionelle

Hindernisse. Da das Spiel anfangs nur auf menschliche Spieler ausgelegt war, konnten diese zwar spielen, aber der eigentliche Sinn, KIs gegeneinander antreten zu lassen, rückte in den Hintergrund. Daraus resultierend wuchsen Klassen von MonoBehaviour erbend. Sie übernahmen auch Funktionalitäten, die auch in eine andere Klasse hätten ausgelagert werden können. Hier hätte sich eine höher gelagerte Klasse *Spieler* angeboten. Diese hätte alle allgemeinen Eigenschaften des Spielers übernommen und in der Klasse *PlayerScript* hätten lediglich Funktionen, die Unity direkt betreffen, vorhanden sein müssen. Stattdessen sind übergeordnete Funktionen jetzt auch in die PlayerScript-Klasse integriert, was während der Programmierung für kleinere Probleme sorgte. An sich bietet es jedoch auch Vorteile. So erfolgt die Verwaltung des Spielers an einer zentralen Stelle im Spiel und Änderungen an den Attributen der Entität eines Spielers durch die KI wirkt sich genau so aus, als wären die Änderungen durch einen Menschen vorgenommen worden. Dies kommt dem Aspekt der Fairness wiederum zugute.

Insgesamt zieht sich dieses Vorgehen durch die gesamte Schnittstelle. Die Klasse *PlayerScript* dient hierbei nur als größtes Beispiel. Andere Fälle in denen dieses Vorgehen deutlich wird, sind die abstrakten Bauplätze, die teilweise direkt in den Entitäten des Spiels erzeugt werden, um sie dann an die KI zu schicken. Auch dies scheint konzeptionell fragwürdig, liefert aber eine enge Verzahnung beider Spielmodi (Mensch vs. KI).

Durch eine längere bzw. bessere Konzeptionsphase hätte ich das ungewollte auftreten solcher Verzahnungen wohl möglich verhindern oder zumindest den Einsatz auf gewollte Male beschränken können. Wie groß der Einfluss auf die Dauer des Projektes im Endeffekt genau war, lässt sich nur schwer beziffern. Auf der einen Seite ist der Quellcode auf diese Weise möglicherweise schlechter nachzuvollziehen, was bei Änderungen zu einer Verzögerung führen kann. Andererseits hätte die Konzeption auch mehr Zeit in Anspruch nehmen müssen. Diese beiden Aspekte gilt es also gegeneinander abzuwägen. Langfristig kann jedoch ausgesagt werden, dass dann ein Zeitersparnis durch die Konzeption entsteht.

Auch die Implementierung von Testreaktionen hätte besser konzipiert werden können und hängt engt mit dem obigen Aspekt zusammen. Damit die KI sinnvoll agieren kann, muss zunächst einmal eine geeignete Situationsbeschreibung erstellt werden, die dann mittels JSON über die Schnittstelle der KI zugänglich gemacht wird. Durch das Implementieren der Schnittstelle, nachdem bereits eine laufende Version für menschliche Spieler erstellt wurde, erwies sich auch das als Hindernis. Zur Lösung wurde zusätzlich zu jedem relevanten Unity-Objekt eine abstrakte Darstellung in Form einer Klasse erzeugt, die alle relevanten Informationen enthält. Die Zusammenfassung dieser Klassen in Form der Klasse *Situation* ermöglichte schließlich eine geeignete Übermittlung der Daten, sodass die KI antworten kann.

Zu diesem Zeitpunkt unterstützte die KI jedoch noch keine wirklich intelligente Verwaltung von Wissen. Stattdessen musste im Quellcode mittels Verzweigungen festgehalten werden, wie auf bestimmte Eigenschaften einer Situation reagiert wird. Dies vereinfachte das Testen des Spiels und half dabei, die Schnittstelle richtig zu konfigurieren. Dennoch erschwerte dies das Einführen der fertigen bzw. initialen KI. Diese arbeitet fallbasiert mit dem MyCBR zusammen und hierfür mussten wieder fundamentale Änderungen am Quellcode vorgenommen werden. Die Anpassungen an der Schnittstelle waren minimal. Lediglich die Java-Klasse *CBRSystem* musste etwas umgebaut werden. Doch war die Überführung der Situation in Form einer Klasse in Fälle, die die Fallbasis versteht relativ aufwendig. Außerdem änderte ich die Fallbasis zur Erhöhung der Intelligenz der KI dauerhaft. Auch dies war mit hohem Anpassungsaufwand verbunden.

Rückblickend würde ich auch dies anders lösen. Ich würde zunächst genau festlegen, welche Reaktionen eine initiale KI zeigen soll und dann Fälle definieren, mit denen diese Reaktionen erreicht werden kann. Somit würde die Implementation von Testreaktionen ohne Fallbasis wegfallen und es wäre mehr Zeit zur Verbesserung der initialen KIs geblieben. Kombinieren würde ich dieses Vorgehen mit dem oben beschriebenen Punkt, der besagt, dass mehr Zeit auf die Konzeption hätte verwendet werden sollen. Dann wäre das Spiel zwar langsamer und mit zunächst weniger Features, aber dafür insgesamt schneller entstanden und die weitere Funktionalitäten hätten mit weniger Aufwand Einzug ins Spiel erhalten können.

Das Stoppen des Spiels war bevor die KIs eingeführt werden auch kein Problem. Der Gamemanager wurde disabled, was automatisch dazu führte, dass das Spiel nicht weitergeführt werden konnte. Die Anfrage durch das Spiel an die Schnittstelle erfolgt allerdings unabhängig von der Update-Methode des Gamemanagers, was zu einem fortführen des Spiels führt, selbst wenn schon der *Game Over Screen* angezeigt wird. Um dieses Problem zu beheben wurde vor dem Starten einer Anfrage an die KI getestet, ob nicht schon ein Spieler gewonnen hat. Sollte dies der Fall sein, wird keiner weitere Anfrage gesendet und das Spiel stoppt. Eigentlich handelt es sich hierbei mehr um einen einfachen Bug, als um ein Problem, das die Umsetzung betrifft. Allerdings geht es in die gleiche Richtung wie die oben aufgeführten und wird daher auch in diesem Zusammenhang behandelt.

Abschließend bleibt zu den behobenen Problemen zu sagen, dass diese zwar einen Einfluss auf die Umsetzung des Projektes hatten und durch eine bessere Konzeption möglicherweise Zeit hätte gespart werden können. Jedoch hielten die aufgeführten Hindernisse das Fortschreiten der Entwicklung nicht solch einem Ausmaß zurück, dass der Erfolg der Implementierung zu einem Zeitpunkt gefährdet gewesen wäre. Denn an sich wurde viel Zeit in die Konzeption gesteckt, was im Kapitel 4 Entwurf nachzulesen ist. In der tatsächlichen Umsetzung mussten dann aber Anpassungen vorgenommen werden, die einige Aspekte des Entwurfs hinfällig machten. Eventuell wäre es daher eine Option gewesen, wenn deutlich wird, dass ein Aspekt nicht so umgesetzt werden kann, zurück zum Entwurf zu gehen und diesen zu überarbeiten. Hierdurch hätten die genannten unerwünschten Effekte vermieden werden können.

7.4.2 Andauernde Probleme

Neben den behobenen Problemen, die während der Umsetzung auftraten, gibt es auch einige wenige, die nicht gelöst werden konnten. Diese bestehen demnach weiterhin. Sie sollen hier erläutert und theoretisch behandelt werden.

Das vorherbestimmen der Antworten der fertigen KI erwies sich als größeres Problem, dass sich nicht abschließend lösen ließ. Hierbei wurden zwar Standardfälle in die Fallbasis eingefügt, die für ein bestimmtes Verhalten der KIs sorgen sollten. Die KIs verhielten sich allerdings nicht immer so wie erwartet. Sodass schließlich einige Einschränkungen in das Retrieval eingeführt werden mussten. Inzwischen ist daher das Problem teilweise behoben. Zwar werden teilweise Pläne durch die KI geliefert, die nicht komplett auf die Situation passen, aber dadurch sind Schwachstellen im Programm an sich aufgefallen.

Ein Beispiel für einen solchen Fall ist das Einbeziehen des vorherigen Plans der KI in die aktuelle Entscheidung. Dies wäre nötig wenn Bauplätze für Siedlungen aktiviert werden sollen. Es kann vorkommen, dass die KI den Plan, Bauplätze zu aktivieren entsendet, aber keiner aktiviert wird. Dann hätte sich die Situation für die KI nicht geändert und sie würde diesen Plan immer wieder entsenden. In der Implementation wird dieses Problem nun gelöst, indem der vorherige Plan zwischengespeichert wird und mit dem aktuellen abgeglichen wird. Somit kann ein Fehlverhalten verhindert und ein alternatives Vorgehen

in die KI integriert werden. Allerdings wäre es angebracht auch dies fallbasiert zu lösen. Hierzu könnte entweder der vorherige Plan Einzug in die Fallbasis erhalten oder es wird die Situation an sich verändert. Die erste Lösung erscheint zwar wirksam, aber den vorherigen Plan in die Fälle einzufügen, würde die Fallbasis unnötig aufblasen. Stattdessen ist die Situation an sich zu verändern die bessere Option. Hierfür müsste pro Aktivierungsaktion eine Methode zur Verfügung gestellt werden, die einbezieht, ob überhaupt Plätze aktiviert werden würden. Diese Information würde in Form eines weiteren Symptoms in den Fällen gespeichert werden. Konkret bräuchte es drei Symptome, die mit Wahrheitswerten gefüllt werden. Pro Bauplatzart bräuchte es ein Symptom.

Am simpelsten ist die Überprüfung von Städtebauplätzen. Hierfür muss die entsprechende Methode lediglich prüfen, ob der aktuelle Spieler über Siedlungen verfügt, denn jede Siedlung ist auch gleichzeitig ein Bauplatz für eine Stadt. Wenn also unausgebaute Siedlungen auf dem Spielfeld vorhanden sind, so können auch Städtebauplätze aktiviert werden.

Schwieriger gestaltet sich die Überprüfung für Straßen und Siedlungen. Bei beiden sind zwei mögliche Vorgehen denkbar. Entweder auf Seiten der KI oder das Spiel fügt die benötigte Information direkt mit in die Situation ein. Wenn die KI diese Information liefert, müsste eine Methode gestaltet werden, die eine normale Aktivierung der Plätze simuliert und speichert, wie viele Aktivierungen vorgenommen wurden. Im Fall von null Aktivierungen würde ein *false* ansonsten ein *true* übermittelt werden. Dies ist für Straßen und Siedlungen gleichermaßen einsetzbar. Auf Seiten der KI allerdings könnten die Vektoren für die Abstandsmessung nicht genutzt werden. Hier wäre es nötig anhand der Zeilen und Spalten, die einem Bauplatz zugewiesen worden sind, zu rechnen. Die Methode an sich würde also überprüfen, ob die Bedingungen im Umfeld eines Bauplatzes gegeben sind, damit dieser aktiviert würde. Für Straßenbauplätze müsste eine weitere Straße des gleichen Spielers unmittelbar benachbart liegen und für Siedlungen müsste dies auch erfüllt und zusätzlich müsste die angrenzende Straße wieder einen Nachbarn in Form einer Straße haben. Zu beachten ist, dass die erste Möglichkeit der Lösung auf Seiten des Spiels eine Leistung erbringt, die eigentlich durch die KI selber erbracht werden müsste. Schließlich muss ein menschlicher Spieler auch selbst feststellen, dass ein erneutes Drücken des Buttons zum Aktivieren von Bauplätzen unsinnig ist, wenn beim ersten Versuch schon keine Plätze aktiviert wurden. Allerdings würde sich die Funktionalität gewissermaßen doppeln, wenn die Aktivierung sowieso durch das Spiel überprüft wird und eine zusätzliche Implementation durch die KI stattfinden müsste. In diesem Fall bietet es sich daher an die Implementierung auf Seiten des Spiels vorzunehmen. Der Grundsatz der Fairness könnte hierdurch zwar als verletzt betrachtet werden, jedoch nur theoretisch. Denn ein Mensch, der in der Lage ist das Spiel zu spielen, dem wird es auch möglich sein, zu erkennen, dass ein erneutes Drücken des gleichen Buttons bei identischer Situation zu keinem anderen Ergebnis führt.

Aus Zeitgründen wurden diese Anpassungen nicht im Spiel umgesetzt und hier nur theoretisch beschrieben. Durch diese Erklärung ließe sich eine Umsetzung relativ unkompliziert vorzunehmen.

7.5 Mögliche Verbesserungen / Erweiterungen

Wie jedes Produkt, lässt sich auch die hier erstellte Software verbessern, was sich schon daran erkennen lässt, dass nicht alle Kann- und Soll-Anforderungen umgesetzt wurden. Als große Punkte sind hier der Handel zwischen Spielern, der Dieb und das Verwenden von Ereigniskarten nennen. Durch die Erweiterung des Spiels durch diese Funktionen, würde das Spielen an sich eine komplexere Strategie erfordern. Die Situationsbeschreibung für die KIs müsste umfangreicher werden, wodurch die Komplexität für diese steigt und mehr

Wissen in der Wissensbasis nötig wird. Auf diese Weise würde dann insgesamt das Produkt verbessert werden. Wenn eine dieser Anforderungen implementiert würde, dann sollte am ehesten der Handel gewählt werden, da dieser die interessantesten strategischen Optionen bietet, indem er eine direkte Interaktion zwischen zwei Spielern ermöglicht. Hier müsste die KI miteinbeziehen, dass der andere möglicherweise auch profitiert und es muss abgewogen werden, ob die KI selbst mehr vom Handel profitiert als der Gegner dies tun würde. Diese Frage stellt sich sowohl als Initiator eines Handels als auch als sein Partner.

Eine weitere interessante Erweiterung wäre das Erlauben von mehr als zwei Spielern. Hierzu müsste die Anzahl der Spieler Teil der Situationsbeschreibung werden, da diese Anzahl nun relevant für die Strategie der Spieler sein kann. Die Erhöhung der Spieleranzahl würde dazu führen, dass strategisch günstige Bauplätze für Siedlungen schneller vergriffen wären und ein computergesteuerter Spieler sich möglichst früh gut aufstellen muss. Außerdem könnte es viel eher dazu kommen, dass ein Spieler quasi handlungsunfähig wird, weil alle Plätze für den Bau von Objekten um ihn herum belegt sind. Dies stellt einen Zustand dar, der bisher nicht betrachtet wurde und das Einbauen von Spielern, könnte sogar zur Strategieoption werden. Ein KI könnte dann anhand der Situation erkennen, dass die Möglichkeit besteht einen Konkurrenten auszustechen, indem an der Stelle eine Siedlung oder Straße errichtet wird, die die einzige Expansionsoption für den anderen Spieler darstellt.

7.5.1 Intelligentere KI

Die Verbesserung der KI wurde bereits in der der Einleitung dieses Kapitels angerissen. Jedoch kann diese noch deutlich weiterentwickelt werden. Daher soll in diesem Abschnitt auf die einzelnen Verbesserungsmöglichkeiten eingegangen werden. Außerdem sollen hier weitere interessante Strategien betrachtet werden, die eine KI verwenden könnte.

Strategie 1: Sammeln eines Rohstoffs
Wenn der Tausch von Rohstoffen mit der Bank oder anderen Spielern im implementiert würde, ließe sich das Sammeln eines einzelnen Rohstoffs als valide Strategie einsetzten. Hierbei würde die KI beim Bau einer neuen Siedlung immer den gleichen Rohstoff priorisieren. Durch dieses vorgehen, würde sie enorme Mengen dieses Rohstoffs anhäufen. Wenn Sie diese nun gegen benötigte Rohstoffe tauscht, kann sie so schnelle Fortschritte im Spiel machen und die Partie möglicherweise für sich entscheiden. Für diese Strategie spricht, dass normalerweise mehrere Siedlungen nötig sind, um an genügend Rohstoffe zu kommen. Durch die Tausch-Strategie kann sich mit wenigen Siedlungen bzw. Städte konzentriert werden und die Siedlungen müssen weniger weit verteilt aufgeschlagen werden.

Strategie 2: Würfelwahrscheinlichkeit Derzeit enthalten die Fälle keinerlei Informationen darüber, welche Zahlen auf den Rohstoffkarten liegen. Diese Information könnte für eine verbesserte KI aber durch aus Wert haben. Einige Zahlen werden häufiger gewürfelt als andere, wenn zwei Würfel gleichzeitig geworfen werden. Die Wahrscheinlichkeit für eine Sieben beispielsweise beträgt 16,67%, während eine Zwei bzw. Zwölf nur mit einer Wahrscheinlichkeit von 2,78% gewürfelt wird. Leider ist die Sieben für den Räuber reserviert und findet sich daher nicht auf Ressourcenfeldern. Allerdings werden Sechs und Acht auch besonders häufig gewürfelt. Beide mit 13,89%. Die KI könnte also versuchen besonders solche Felder zu bebauen, die beim Würfeln einer Sechs oder Acht Ressourcen bringen. Kombiniert mit der Abwägung, ob sich das Streben nach einem bestimmten Rohstoff, der gerade benötigt wird, mehr lohnt. Hierdurch könnte eine Art Wettbewerbsvorteil entstehen, denn die genutzten Informationen sind umfangreicher.

Strategie 3: Fokus auf den Räuber An sich soll der Räuber aktiviert werden, wenn eine Sieben durch den Spieler am Zug gewürfelt wird. Dies ist selbstverständlich nicht zu beeinflussen. Dennoch kann nach Integrierung der Ereigniskarten, der Räuber auch

bewusst angesteuert werden. Die KI müsste hier allerdings vermehrt abwägen. Sie könnte nicht einfach nur Ereigniskarten kaufen und auf den Bau von Objekten auf dem Spielfeld verzichten. Sie müsste einen Ausgleich zwischen der Einnahme neuer Ressourcen und der Ausgabe für Ereigniskarten finden. Hierfür wäre es insbesondere vorteilhaft, wenn die Situation der anderen Spieler mit in die eigene Strategie einbezogen wird. Dies hängt damit zusammen, dass das Verschieben des Räubers dann besonders gewinnbringend ist, wenn dieser A) auf einem eigenen Ressourcenfeld steht oder B) die Besetzung eines gegnerischen Feldes, dem Gegner ausreichend großen Schaden zugefügt wird. Demnach scheint es nicht sinnvoll einfach nur Karten zu sammeln, die das Verschieben des Räubers erlauben. Die bessere Alternative scheint es zu sein, einen Ereigniskarten zu sammeln, bis die passende Karte kommt und diese solange aufzubewahren, bis erwartete Nutzen groß genug ist.

8 Alternative Ansätze

9 Bewertung

9.1 Relevanz der Forschung für andere Bereiche

9.2 Mögliche Verbesserungen/Weiterentwicklungen

10 Fazit & Ausblick

Literaturverzeichnis

[Dav99] Davis, Ian L.: Strategies for Strategy Game AI / ActivisionI, Inc. 1999. – Forschungsbericht

[May07] Mayer, Helmut A.: Board Representations for Neural Go Players Learning by Temporal Difference. In: *2007 IEEE Symposium on Computational Intelligence and Games*, IEEE, apr 2007

[SRGC07] Sánchez-Ruiz, Antonio C. ; González-Calero, Pedro A.: Game AI for a Turn-based Strategy Game with Plan Adaptation and Ontology-based retrieval, 2007

[WM09] Weber, Ben ; Mateas, Michael: Case-Based Reasoning for Build Order in Real-Time Strategy Games., 2009

A Anhang

Selbstständigkeitserklärung

Hiermit erkläre ich, **Tjark Harjes**, dass ich diese Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Stellen der Arbeit, die wörtlich oder sinngemäß aus Veröffentlichungen oder aus anderweitigen fremden äußerungen entnommen wurden, sind als solche kenntlich gemacht. Ferner erkläre ich, dass die Arbeit noch nicht in einem anderen Studiengang als Prüfungsleistung verwendet wurde.