

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

<b>Abkürzungsverzeichnis</b>	<b>iii</b>
<b>Abbildungsverzeichnis</b>	<b>iv</b>
<b>Tabellenverzeichnis</b>	<b>iv</b>
<b>Quellcode-Verzeichnis</b>	<b>v</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Ziel der Arbeit . . . . .	1
1.2 Aufbau der Arbeit . . . . .	1
<b>2 Grundlagen</b>	<b>2</b>
2.1 Computergegner in rundenbasierten Strategiespielen . . . . .	4
2.1.1 Der KI-Spieler als Wissensbasiertes System . . . . .	4
2.1.2 Der KI-Spieler als Neuronales Netz . . . . .	10
<b>3 Konzeption</b>	<b>13</b>
3.1 Problemstellung . . . . .	13
3.2 Entwurf . . . . .	13
3.3 Die Regeln und Anpassungen . . . . .	16
3.4 Konzeption der Schnittstelle . . . . .	19
<b>4 Implementation</b>	<b>24</b>
4.1 Verwendete Technologien . . . . .	24
4.2 Die grundlegende Umsetzung des Spiels . . . . .	25
4.2.1 Die Generierung des Spielfelds . . . . .	26
4.2.2 Der Spieler . . . . .	29
4.2.3 Der Gamemanager . . . . .	36
4.2.4 Die Repräsentation der Bauobjekte . . . . .	43
4.2.5 Der Bau von Siedlungen und Straßen . . . . .	43
4.3 Die Schnittstelle . . . . .	45
4.4 Erweiterung durch den Aufbau von Städten . . . . .	50
4.5 Initiale KIs . . . . .	54
4.6 Bug-Fixes . . . . .	59
4.7 Anpassungen . . . . .	60
4.7.1 Anpassungen am Spiel . . . . .	60
4.7.2 Anpassungen an der Schnittstelle . . . . .	61
<b>5 Evaluation</b>	<b>62</b>
5.1 Muss-Anforderungen . . . . .	62
5.2 Soll- und Kann-Anforderungen . . . . .	64
5.3 Anforderungen an die Schnittstelle . . . . .	65
5.4 Behobene Probleme . . . . .	66
5.5 Tests . . . . .	69
<b>6 Fazit &amp; Ausblick</b>	<b>71</b>
6.1 Fazit . . . . .	71
6.2 Ausblick . . . . .	73
6.2.1 Verbesserungen . . . . .	73

6.2.2 Erweiterungen . . . . .	74
6.2.3 Weitere Möglichkeiten . . . . .	76
<b>Literaturverzeichnis</b>	<b>77</b>
<b>A Anhang</b>	<b>78</b>
A.1 Ergänzungen zum abgebildeten Quellcode . . . . .	78
A.2 Regelwerk: Siedler von Catan . . . . .	80
Selbstständigkeitserklärung . . . . .	91

## Abkürzungsverzeichnis

**API** Application-Program-Interface.

**JSON** JavaScript Object Notation.

**KI** Künstliche Intelligenz.

**WBS** Wissensbasiertes System.

## Abbildungsverzeichnis

2.1	Architektur eines KI-Spiels (In Anlehnung an [SRGC07, vgl. S. 4]) . . . . .	2
2.2	Architektur menschlicher Spieler (eigene Darstellung) . . . . .	3
2.3	Wissensbasiertes System (Aus Vorlesung WBS von Herrn Althoff) . . . . .	5
2.4	Fallbeispiel ([WM09, vgl. S. 108]) . . . . .	7
2.5	Beispiel neuronales Netz . . . . .	10
3.1	Verteilung der Zahlenchips auf dem Spielfeld (siehe Anhang Regelwerk S. 4)	17
4.1	Nutzeroberfläche zu Beginn des Spiels ([But16, vgl.]) . . . . .	26
5.1	Bau von Städten, Siedlungen und Straßen (eigene Darstellung) . . . . .	62
5.2	Ausgabe des Gewinners auf dem Bildschirm (eigene Darstellung) . . . . .	63
5.3	Säulendiagramm zu den Testergebnissen (eigene Darstellung) . . . . .	69

## Tabellenverzeichnis

3.1	Geplante Kommunikation zwischen Schnittstelle und KI-Spieler . . . . .	22
3.2	Geplante Kommunikation zwischen Schnittstelle und KI-Spieler (Fortführung)	23

# Quellcode-Verzeichnis

4.1	Spielfeld-Generator Startmethode . . . . .	26
4.2	Spielfeld-Generator CreateMap-Methode . . . . .	27
4.3	Attribute des Spielers . . . . .	29
4.4	Initialisierung des Spielers . . . . .	30
4.5	Attribute und Initialisierung des Spielers . . . . .	30
4.6	Attribute und Initialisierung des Spielers . . . . .	31
4.7	Attribute und Initialisierung des Spielers . . . . .	31
4.8	Attribute und Initialisierung des Spielers . . . . .	32
4.9	Attribute und Initialisierung des Spielers . . . . .	34
4.10	Initialisierung und Spielerwechsel . . . . .	36
4.11	Update-Methode . . . . .	37
4.12	Update-Methode Aktivierung der Bauplätze für Straßen . . . . .	38
4.13	Würfeln . . . . .	40
4.14	Zug beenden . . . . .	41
4.15	Aktivierung des Baumodus für Siedlungen . . . . .	43
4.16	Bau von Siedlungen . . . . .	44
4.17	Bau von Straßen . . . . .	45
4.18	Hilfsklassen Village und PlaceScript . . . . .	45
4.19	Die relevanten Informationen für eine Situation . . . . .	46
4.20	Die Attribute der Klasse Map . . . . .	46
4.21	Verknüpfung der Erstellung der Karte und den Feldern . . . . .	47
4.22	GameManager: LateStart . . . . .	47
4.23	GameManager: StartAIProcess . . . . .	48
4.24	Ausführen einer Anfrage an den Server . . . . .	48
4.25	PlayerScript: FulfillPlan . . . . .	49
4.26	GameManager: Erstellung des Bauplatzes für Städte . . . . .	51
4.27	Erzeugen einer Stadt im OnClick-Event des Bauplatzes . . . . .	52
4.28	Plan: Übersetzung des Strings in Aktionen über Städte . . . . .	53
4.29	PlayerScript: Umsetzung der Aktionen durch die KI für Städte . . . . .	53
4.30	PlayerScript: Umsetzung der Aktionen durch die KI für Städte . . . . .	54
4.31	Java-Klasse Player: Erstellen eines normalen Zuges zum Testen . . . . .	55
4.32	Java-Klasse DefaultCases: Beispiel . . . . .	56
4.33	Java-Klasse Status: CalculatePreferencePlayer2 . . . . .	58
A.1	Hilfsklasse: Tile . . . . .	78
A.2	Gamemanger: Start- und Awake-Methode . . . . .	78
A.3	PlayerScript-Methode zum Sammeln von Ressourcen für eine bestimmte Siedlung . . . . .	78
A.4	Methode des Status zum Berechnen der Präferenz von Spieler 1 . . . . .	79

# 1 Einleitung

Künstliche Intelligenz (KI) ist heutzutage in Computerspielen weit verbreitet und fand schon Ende des letztes Jahrtausends Einzug in die Spieleindustrie ([Dav99, vgl. S. 1]). Seitdem haben sich die KIs weiterentwickelt und agieren heute besser als damals. Dennoch ist die Entwicklung von starker KI für Strategiespiele aufwendig, weshalb häufig KIs gebaut werden, die durch das “Cheaten” einen Vorteil gegenüber menschlichen Spielern haben (vgl. [SRGC07, vgl. S. 24]).

In dieser Arbeit soll das bekannte Gesellschaftsspiel *Siedler von Catan* als Computerspiel umgesetzt werden und durch KIs spielbar sein. Dabei werden an die KIs die gleichen Maßstäbe angelegt, die auch für menschliche Spieler gelten. Ferner wird ihnen das “Cheaten” nicht erlaubt.

## 1.1 Ziel der Arbeit

Ziel dieser Bachelorarbeit ist es, ein rundenbasiertes Strategiespiel für KI spielbar umzusetzen sowie die Konzeption, Implementation und Evaluation zu dokumentieren. Konkret wurde dabei das bekannte Gemeinschaftsspiel *Siedler von Catan* als Ziel der Implementation gewählt.

In dieser schriftlichen Ausarbeitung werden demnach alle theoretischen Aspekte der Software betrachtet, die dazu umgesetzt wurden. Es soll von den Grundlagen, die zum Verständnis späterer Ausführungen wichtig sind, bis zur Evaluation der entworfenen Software alles abgedeckt werden.

## 1.2 Aufbau der Arbeit

Zunächst werden in Kapitel 2 die nötigen Grundlagen thematisiert, die zum Nachvollziehen späterer Kapitel nötig sind. In Kapitel 3 wird dann auf die Konzeption der Software eingegangen. Hier wird wiederum zwischen einer Problemstellung, welche die Gründe und Ziele der Arbeit konkretisiert, und einem Entwurf für die Software unterschieden.

Die Erläuterung der tatsächlichen Implementation erfolgt anschließend im 4. Kapitel, welches im Detail auf alle wichtigen Bestandteile der angefertigten Software eingeht. In diesem Kapitel soll auch ein Vergleich zwischen dem Entwurf und der tatsächlichen Umsetzung erfolgen.

Im 5. Kapitel soll dann schließlich zur Evaluation übergegangen werden. Diese soll prüfen, ob die gestellten Anforderungen erfüllt werden konnten und eine Bewertung des Implementierten vornehmen. Die Evaluation wird auf Grundlage der erfüllten bzw. nicht erfüllten Anforderungen und durchgeföhrten Tests vorgenommen. Nach der Bewertung werden Probleme der Entwicklung angebracht, sowie deren Ursachen zu beschreiben.

Nach der Evaluation folgt der Schlussteil in dem die Kernpunkte dieser Ausarbeitung zusammengefasst werden und ein Fazit gezogen wird. Außerdem wird ein Ausblick gegeben, indem Lösungen zu den in Kapitel 5 beschriebenen Problemen präsentiert werden. Außerdem werden weitere Ansätze für Verbesserungen der Software untersucht.

## 2 Grundlagen

Da diese Ausarbeitung die Entwicklung eines Strategiespiels für computergesteuerte Spieler behandelt, sollen in diesem Kapitel die Grundlagen eines solchen Spiels erläutert werden. Dieses Kapitel ist insofern von besonderer Bedeutung, als dass der Entwurf und die darauf basierende Implementation des Spieles auf den hier dargestellten Grundlagen basieren. Ferner wird in der Evaluation des Spieles nochmals Bezug auf dieses genommen.

Die Entwicklung computergesteuerter Spieler reicht inzwischen schon über 20 Jahre zurück. So schrieben [Dav99, vgl. S. 24ff] 1999 von ihren Entwicklungen, die beispielsweise in das bekannte Strategiespiel *Civilization: Call To Power* Einzug fanden. Das ist u.a. deshalb beeindruckend, weil der Sieg von Deep Blue über Garri Kasparow in Spiel Schach zu diesem Zeitpunkt erst 3 Jahre zurück lag. Seitdem hat sich sowohl für Echtzeitspiele als auch für rundenbasierte Spiele viel getan. Bis heute ist das einprogrammierte "Cheaten" der Computergegner ein Problem, was von [SRGC07, vgl. S. 1] näher beschrieben wurde, wobei ein dieses Problem im Kontext dieser Arbeit nicht auftreten wird, da das Ziel dieser Arbeit nicht ist, eine möglichst spannende KI zu erstellen. Die Arbeit von [SRGC07] ist dennoch interessant, weil die Autoren sich speziell mit dem Case-Based-Reasoning auseinander gesetzt haben. Sie präsentieren zudem eine Idee für die Interaktion von Spiel und Spieler mittels einer *Application Programming Interface* (API). Die API sei in der Lage, eine Verbindung zwischen der KI und dem Spiel zu schaffen, sodass die KI Zugriff auf alle Informationen habe und gleichzeitig Anweisungen erteilen könne. Diese API ist vornöten, da eine Konvertierung der auf dem Bildschirm erscheinenden Informationen erfolgen muss, um diese für einen Computer wieder verarbeitbar zu machen. Diese entsprechende Architektur lässt sich, wie in Abb. 2.1 zu sehen ist, oberflächlich darstellen.

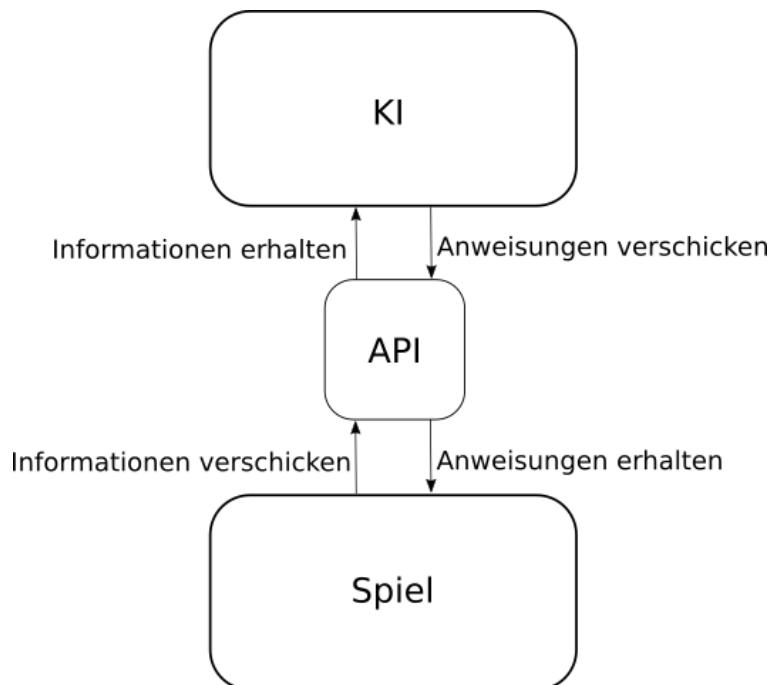


Abbildung 2.1: Architektur eines KI-Spiels (In Anlehnung an [SRGC07, vgl. S. 4])

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

zu sehen ist, sondern nur für 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 ohne eine andere Verkörperung durch eine Spielfigur verbildlicht.

Die unterste Komponente stellt das Spiel dar. Zwischen der KI- und der Spielkomponente 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 das Erhalten von Informationen ermöglicht. Äquivalent dazu funktioniert das Senden und Empfangen von Anweisungen. Diesmal sendet jedoch der Spieler die Anweisungen ab, während das Spiel diese empfängt und ggf. verarbeitet. So wird dem Spieler die Steuerung seiner Repräsentation im Spiel ermöglicht, unabhängig davon, ob es sich um eine Figur oder ein Reich oder ähnliches handelt.

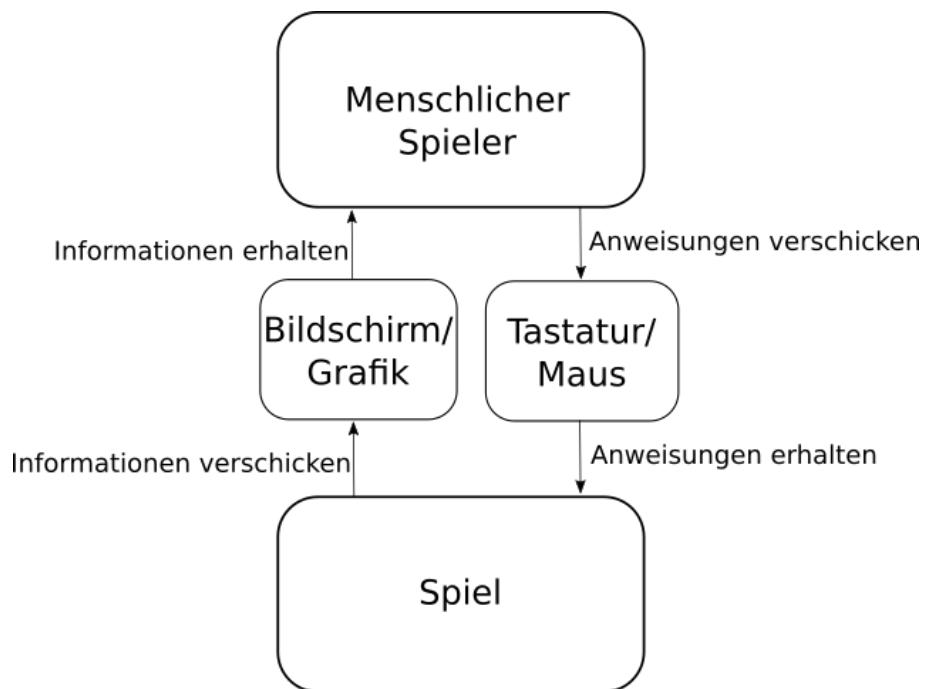


Abbildung 2.2: Architektur menschlicher Spieler (eigene Darstellung)

Abbildung 2.2 zeigt eine analoge Darstellung der Komponenten für menschliche Spieler. Diesmal stellt die oberste Komponente nicht etwa den KI-Spieler dar, sondern einen menschlichen Spieler. 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 *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, erscheinen Maus, Tastatur und Bildschirm als Schnittstelle intuitiver. Doch die Aufnahme von Informationen über einen Bildschirm und das Erteilen von Anweisungen mit Maus und Tastatur ist nichts anderes als das Ver-

wenden einer Schnittstelle zum Steuern der eigenen Repräsentation im Spiel. Für die KI wird das Gleiche erreicht, wenn sie Informationen und Anweisungen über eine API erhält bzw. versendet.

Das beschriebene Prinzip gilt also nicht exklusiv für computergesteuerte Gegner. Anhand von Abb. 2.2 wird deutlich, dass es sich viel allgemeiner um einen Standardweg der Kommunikation zwischen einem Programm und einer 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, sollen 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, vgl. S. 1]. Außerdem ist es möglich, KI-Spieler zu entwickeln, die ihre Strategien mithilfe Wissensbasierter Systeme (WBS) umsetzen. [SRGC07, vgl. S.5]. 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 die bestmögliche Performance gewünscht ist, aber stattdessen ein ganz bestimmtes Verhalten erzeugt werden soll. Ein Ziel, das beispielsweise durch Neuronale Netze nur schwer zu erreichen wäre. Wenn andererseits die stärkste mögliche KI als Ergebnis herauskommen soll, dann eignen sich Herangehensweisen, die die KIs ihre eigenen Methoden finden lassen und durch Lernprozesse auf diesem Wege die besten Strategien ausmachen kann. Dies ist etwas, was 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 WBS gebraucht werden soll, wird der Fokus der Erläuterungen auf diesen liegen. Abbildung 2.3 zeigt, wie ein WBS grundlegend aufgebaut ist. Es wird deutlich, dass eine Abgrenzung zwischen dem Interpreter und dem Wissen gezogen wird. Das ist einer der identifizierenden Punkte eines solchen Systems, da 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 dieser Interpreter wahrnimmt. Anschließend greift dieser zur Verarbeitung der Eingabe auf die angeschlossene Wissensbasis zurück und versucht, mit Hilfe dieser eine passende Lösung 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 Abb. 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 WBS 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 dienen vielmehr der Findung einer Strategie durch den Interpreter und der Interpreter, wobei dieser Aktionen der gefundenen Strategie auch dann ausführen lassen kann, 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, als auch Wissen über das eigene System enthält.

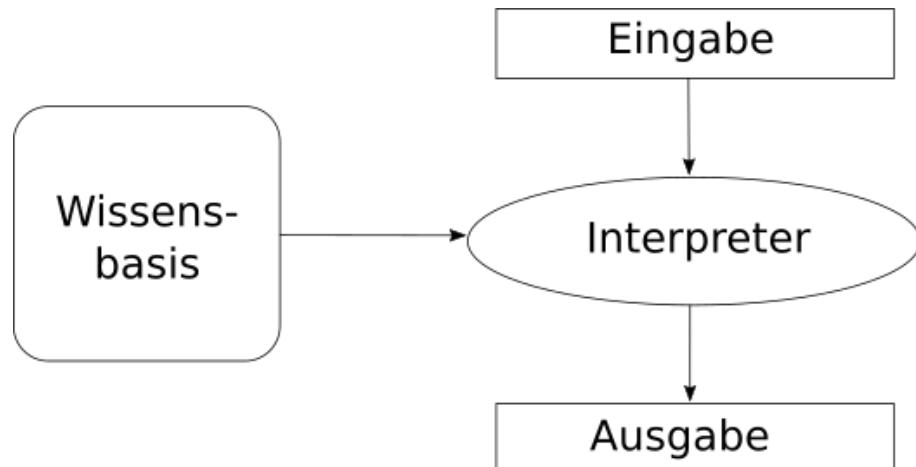


Abbildung 2.3: Wissensbasiertes System (Aus Vorlesung WBS von Herrn Althoff)

Eine mögliche Umsetzung eines WBS 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 Spiels, über die das System Bescheid weiß. Hieraus ergibt sich auch, dass eine "neue" KI über geringeres Wissen verfügt als eine trainierte, weil sie keine dazu Zeit hatte, sich genügend Fälle anzueignen. Was ein Fall enthält, muss definiert werden. Hiervon ist abhängig, wie gut die neuere KI nach dem Training spielen wird.

[WM09, vgl. S. 106] 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 abbilden. So beschreibt jeder Fall den Forschungsstand des Gegners, den eigenen, die Anzahl an Kampfseinheiten (nach Typen aufgeteilt) und Arbeitern sowie die Menge an Produktionsgebäuden und Eigenschaften der Karte. Ein einzelner Fall enthält umfassende Informationen über den gesamten Spielstand, was durchaus nötig ist. Die Strategien werden aufgrund von Ähnlichkeiten 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 wird der ähnlichste dieser beiden zu einem neuen Fall gesucht. Sei nun außerdem angenommen, dass die Eigenschaften des neuen Fall - außer die der Karte - mit Fall 1 übereinstimmen und Fall 2 leicht davon abweicht, dafür hat Fall jedoch eine Karte, die sich nicht von der des neuen Falls unterscheidet. 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 werden. Hieraus lässt sich ableiten, dass es zum Finden der bestmöglichen Strategie nötig ist, die Fälle so genau wie möglich zu beschreiben.

Dennoch muss laut [WM09, vgl. S. 108] 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 diesen beiden Annahmen nicht miteinander, was auf den ersten Blick jedoch paradox wirken kann. Einerseits sollen Fälle die aktuelle Situation so genau wie möglich beschreiben, andererseits wird sich jedoch zur Erkennung des Stands des Gegners auf ein einzelnes Merkmal bezogen. 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. S. 107f]. Sie treffen die Annahme, dass es besser ist, sich auf ein Symptom zu verlassen, welches mit relativ hoher Wahrscheinlichkeit genau bestimmt werden kann, statt weitere, möglicherweise falsche, Merkmale 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.

Da zu jedem Fall einer Wissensbasis auch eine Lösung gehört, muss jeder Fall einen Handlungsvorschlag enthalten, der beschreibt, was in welcher Situation zu tun ist. [WM09, vgl. S. 108f] fügen jedem Fall hierzu eine Art Liste aus Aktionen hinzu, welche die KI vornehmen soll, um zu reagieren. Ein fertiger Fall könnte für dieses Beispiel ungefähr so aussehen wie Abbildung 2.4 darstellt.

Wie die Abbildung 2.4 kann ein Fall schnell relativ umfangreich werden, wenn die Symptome nicht beschränkt sind. Wie zuvor beschrieben wurde auf weitere Symptome, den Gegner betreffend, verzichtet. Das ist nicht nur für die Zuverlässigkeit, sondern auch für die Aussagekraft selbst hilfreich. Selbst, wenn zwei Symptome zuverlässig erfasst werden können, 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 als auch Nachteile bedeuten: Einerseits kann ein solches System durch die Vermeidung ähnlicher Symptome 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 gilt es sehr genau zu betrachten, welche Eigenschaften des Spiels in einem Fall abzubilden sind und ob diese nötig sind bzw. einen Vorteil für die KI bringen.

Da in dem herangezogenen Paper *Wargus* durch die KI gemeistert werden sollte und es sich hierbei um ein Echtzeitspiel handelt, sind die Ergebnisse möglicherweise nur teilweise auf rundenbasierte Spiele übertragbar. 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 Anweisungen gibt, 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, sodass die passende Strategie verändert werden müsste. Das erhöhe den Schwierigkeitsgrad der Konzeptionierung einer solchen KI, da zusätzlich festzulegen wäre, ob eine Strategie geändert, eingehalten oder gar abgebrochen werden soll, wenn sich die Situation im Spiel verändert. Es könnte auch eine Integration dieser Komponente in die Fälle stattfinden. Das erfolgt laut den Ausführungen von [WM09, vgl. S. 109f] für Ihre KI nicht, könnte aber Vorteile

Situation:	
Eigener Forschungsstand	Schmid Festung Mühle
Gegner Forschungsstand	Schmid Mühle
Eigene Truppen	Axtwerfer: 5 Katapulte: 3 Orgs: 19
Eigene Arbeiter	Tagelöhner: 10
Produktionsgebäude	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 Prdduktionsgebäude	Barracken: 2

Abbildung 2.4: Fallbeispiel ([WM09, vgl. S. 108])

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 innerhalb dieses Zeitraums Entscheidungen getroffen werden. Jedoch kann sich die Situation eines Spiels auch durch die eigenen Aktionen ändern und in dem betroffenen Zug sind die weiteren Entscheidungen dementsprechend zu überdenken. Dieses Problem könnte dadurch umgangen werden, dass ein Zug zu Beginn durchgeplant wird. Das bedeutet, dass über alle möglichen Tätigkeiten bereits vor der ersten eigenen Aktion schon entschieden wird, wobei das die Performanz beeinträchtigen könnte. Komplizierter wird das Problem jedoch, wenn Aktionen über mehrere Runden andauern. 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 miteinfließen. Gegenüber den Echtzeitspielen bleibt jedoch immer noch der Vorteil, dass keine dauerhafte Neubewertung der Situation erfolgen muss, da die Rundenbasiertheit vereinfachend auf das zeitliche Gefüge des Spielablaufs einwirkt - zumindest aus Sicht der KI-Spieler.

Die obigen Ausführungen über fallbasierte KIs werfen die Frage auf, ob es Alternativen 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 Arten des Lernen sind das inkrementelle, das analogiebasierte, das erklärbungsorientierte sowie das Lernen durch Vergessen. Inwiefern diese sich für die Erstellung von KIs für Videospiele eignen, bleibt allerdings fraglich und soll im Folgenden näher erläutert werden.

### **Inkrementelles Lernen:**

Vom Namen dieses Lernverfahrens lässt sich bereits auf seine Eigenschaft 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 inkrementellen Lernens 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, bei zweiterem negiert oder umgekehrt und bei letzterem müsste in eine beliebige Richtung verstärkt werden. Dies könnte dazu führen, dass die KI in ihrer Performanz teilweise stark schwankt, aber langfristig eine Verbesserung hinsichtlich ihrer Performanz festzustellen ist. Weiterhin ließe sich inkrementelles Lernen auch als alleiniges Verfahren einsetzen. Dazu müsste die Festhaltung des Wissen, aber in einer anderen Form als in Fällen erfolgen. Hier wären Regeln denkbar, die individuell bei eintretenden Ereignissen eine Antwort des Systems erzeugen und durch inkrementelles Lernen abgeändert werden können, um auch diese langfristig zu verbessern. Dies scheint auch eine Alternative zum fallbasierten Schließen darzustellen, da auch beim analogiebasierten Lernen eine individuellere Betrachtung der Ereignisse stattfinden kann. [PBG96, vgl. S. 207f], [Alt19, vgl. V. 5 S. 35]

Im Grunde ist der Aufbau einer KI mithilfe eines Neuronalen Netzes auch nichts anderes als die Anwendung inkrementellen Lernens. Bei Neuronalen Netzen wird durch Anpassung der Gewichte zwischen Neuronen nach jedem Lernschritt (Backpropagation) eine Veränderung des Verhaltens erreicht. Das wäre dann im eigentlichen Sinne kein WBS mehr, jedoch könnte ein Neuronales Netz die Wissensbasis ersetzen und über einen Interpreter, der wiederum wie bisher Informationen vom Spiel empfängt, angesteuert werden. Dieser 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, Wissen von einem bereits bekannten Objekt auf ein neues Objekt zu übertragen und so die richtigen, analog angewendeten Schlüsse als gewonnenes Wissen explizit zu speichern (vgl. VL Althoff Wissensbasierte Systeme). Im Hinblick auf eine mögliche Übertragung auf Videospiele könnte hierbei die Ähnlichkeit von Situationen genutzt werden, um allgemeine Schlüsse ableiten zu können, die für all diese Arten von Situationen funktionieren. Dieser Ansatz scheint dem fallbasierten Schließen zu ähneln, da bei beiden mit der aktuellen Ähnlichkeit aktueller Situationen zur vorherigen gearbeitet wird. Der 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 können ähnliche Aspekte der Situation genutzt werden, um einzelne Strategien zu übertragen und anzuwenden. Fallba-

siertes Schließen hingegen würde den Vorteil der besseren Ausnahmebehandlung mit sich bringen, da so jede Ausnahmesituation in die Fallbasis Einzug findet und nicht auf eine andere Weise repräsentiert werden würde. [Alt19, vgl. V. 5 S. 12, 25, 28]

### **Erklärungsbasiertes Lernen:**

Zur Anwendung erkläzungsbasierten Lernens benötigt es drei Voraussetzungen: einen Zielbegriff, ein positives Beispiel und ein Operationalisierungskriterium, dem der Zielbegriff zu Anfang nicht genügt. Der Lerneffekt besteht darin, das Beispiel so zu generalisieren, dass es operational ist. Dieses Lernverfahren dann gut geeignet, wenn nachvollziehbares Wissen generiert werden soll in Form von Erklärungen. Zum Erstellen einer KI für ein rundenbasiertes Strategiespiel scheint diese Herangehensweise jedoch wenig geeignet zu sein: Die Konzeption einer solchen KI wäre möglich, allerdings scheint dieses Verfahren der KI keinen Vorteil zu bringen. Eine Art Implementierung dieser Variante wäre das WBS mit Beispielen von Siegen zu versorgen und es hieraus zu operationalisieren. [Alt19, vgl. V. 5 S. 38]

### **Lernen durch Vergessen:**

Das Lernen durch Vergessen eignet sich zur Verbesserung von Systemen, indem Wissensinhalte abstrahiert 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 Lernverfahren vielmehr als Ergänzung und nicht als Ersatz eines anderen fungieren. Angewandt aufs analogebasiertes Lernen könnten einige Schlüsse gelöscht werden, wenn diese nicht vorteilig sind. 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 keine weiteren Symptomausprägungen erschlossen werden müssen. Je nach KI kann der Einsatz dieses Verfahrens durchaus in Erwägung gezogen werden. [Alt19, vgl. V. 5 S. 39]

Insgesamt lässt sich festhalten, dass es durchaus sinnvolle Alternativen zum fallbasierten Lernen für wissensbasierte KI-Spieler gibt. Einige sind dabei besser geeignet als andere und einige würden sogar eine gute Ergänzung darstellen. So eignet sich das erkläzungsbasierte 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 hingegen das Lernen durch Vergessen. Dieses könnte für den hier beschriebenen Anwendungsfall die oben genannten Vorteile mit sich bringen, allerdings nur als Erweiterung in Kombination mit anderen Lernverfahren. Neben der eher geringen Einsetzbarkeit, in Verbindung mit den fallbasierten Schließen ([WM09, vgl. S. 107f]), können die Vorteile anderer Wissensrepräsentationsarten besser genutzt werden. So wäre der kombinierte Einsatz inkrementellen Lernens und des Lernens durch Vergessen möglicherweise eine gute Kombination, die an der später erläuterten Software gegen andere Lernverfahren getestet werden könnte. Das inkrementelle Lernen stellt bereits eine Alternative zum fallbasierten Lernen dar, wenn auch mit einer anderen Form der Wissensrepräsentation oder sogar als Neuronales Netz. Vor allem die individuelle Anwendbarkeit des Wissens scheint hier Möglichkeiten dazu zu bieten, die Fallbasiertheit 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 Performanz einer späteren KI haben kann.

Abschließend lässt sich zu den verschiedenen Lernstrategien festhalten, dass, obwohl [WM09] nur die Strategie des fallbasierten Lernens angewandt wurde, einige weitere Verfahren zur Verfügung stehen und gegeneinander getestet werden können, um die bestmögliche KI zu entwickeln. In diesem Kapitel wurden zusätzlich verschiedene Vermutungen dazu ange stellt, wie sich die unterschiedlichen Strategien auswirken und welche Strategie am Ende zu den besten Ergebnisse führt. Wirklich festzustellen ist dies aber nur durch 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 tiefgehender beschrieben werden. Abbildung 2.5 zeigt wie ein Neuronales Netzwerk grundlegend aufgebaut ist. Im Detail zeigt die Abbildung ein Neuronales Netz, welches aus drei Schichten aufgebaut ist und insgesamt fünf Knoten beinhaltet: Zwei Inputknoten in der ersten Schicht, zwei Outputknoten im Hiddenlayer und ein Outputknoten in der Outputlayer. Es wird in dieser Darstellung (2.5) auf die Verwendung des Bias verzichtet. [Zup94, vgl. S. 2ff]

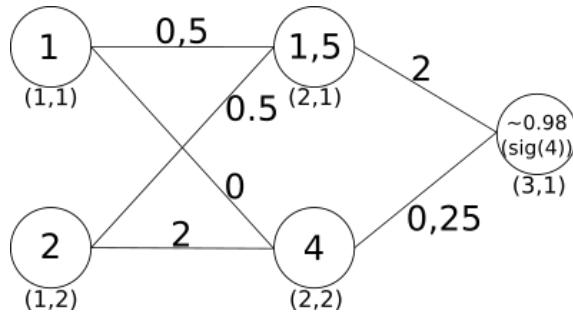


Abbildung 2.5: Beispiel neuronales Netz

Die Schichten sind über gewichtete Verbindungen mit der darauffolgenden Schicht verbun den, 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 Knotens in den folgenden Schichten wird auf Basis aller Knoten der vorherigen Schicht berechnet. So wird der Wert des Knotens (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 daraus folgender Ablauf:

$$Wert((1,1)) = 1$$

$$Wert((1,2)) = 2$$

$$Wert((2,1)) = 1 * 0,5 + 2 * 0,5 = 1,5$$

$$Wert((2,2)) = 1 * 0 + 2 * 2 = 4$$

$$Wert((3,1)) = \text{sig}(1,5 * 2 + 4 * 0,25) = \text{sig}(4) = 0,98$$

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 einzige mögliche Aktivierungsfunktion dar. Generell sind alle stetigen Funktionen einsetzbar, aber hier sollen nur einige, wichtige genannt werden:

1. **Sigmoidfunktion** ([NIGM18, vgl. S.5ff]):  $\frac{1}{1+exp(-ax)}$ , mit  $a \in \mathbb{R}$
2. **Lineare Funktion** ([AHSB14, vgl. S. 1ff]):  $linear(x) = m^*x + b$ , mit  $m \in \mathbb{R}$  der Steigung und  $b \in \mathbb{R}$  dem Bias.
3. **Stückweise lineare Funktion** ([AHSB14, vgl. S. 1ff]):  

$$l(x) = \begin{cases} 1 & \text{falls } x \geq \frac{1}{2} \\ a + \frac{1}{2} & \text{falls } -\frac{1}{2} < x < \frac{1}{2} \\ 0 & \text{falls } x \leq -\frac{1}{2} \end{cases}$$
4. **ReLU** ([NIGM18, vgl. S.8f], [RZL17, vgl. S.3ff]):  $ReLU(x) = \max(0, x)$
5. **Hard limit** ([BB17, vgl. S.1040f]):  $hard(x) = \begin{cases} 1 & \text{falls } x \geq 0 \\ 0 & \text{falls } x < 0 \end{cases}$

Die Entscheidung über den Einsatz der zu nutzenden Funktion ist u.a. vom Ziel abhängig, das mit dem Neuronalen Netz erreicht werden soll. Bezogen auf Videospiele wie *Wargus* könnte beispielsweise die 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 sich dagegen eher durch eine Sigmoidfunktion abbilden (1 als ja und 0 als nein).

Bei dem Beispiel aus Abbildung 2.5 handelt es sich nur um ein rudimentäres Neuronales Netz. Real eingesetzte Neuronale Netze sind deutlich umfangreicher. Sie unterscheiden sich in der Menge verwendeter Knoten pro Schicht sowie hinsichtlich der Anzahl der Schichten. So bestehen Neuronale Netze i.d.R. aus einer Input- und einer Outputlayer sowie beliebig viele hintereinanderliegenden Hiddenlayers. Außerdem können die Schichten abweichend vom Beispiel über eine beliebige Anzahl an Knoten verfügen (siehe Abb. 2.5), wobei nicht jede Schicht gleich viele Knoten haben muss. Wie viele Schichten und Knoten in einem Neuronalen Netz genutzt werden, hängt i.d.R. von der Komplexität der zu bewältigenden Aufgabe ab. Hierzu soll jedoch erwähnt werden, dass ein größeres Netz auch mehr Training benötigt, um die gewünschte Aufgabe zu erlernen. Dies hängt damit zusammen, dass mehr Verbindungen trainiert werden müssen. Einerseits stellt die Größe des Netzes einen Vorteil dar, andererseits kann dieser sich jedoch auch nachteilig auswirken, wenn nicht genügend Trainingsdaten verfügbar sind.

Das Trainieren eines Neuronalen Netzes ist ähnlich zum Finden von Koeffizienten einer Funktion deren Grad gleichzeitig bestimmt 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 Neuronaler Netze, 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 [Zup94, vgl. S. 5f].

Die Voraussetzung für Backpropagation ist es, dass zu einem Datensatz bereits das richtige Ergebnis bekannt ist. Dieser Datensatz wird als Input in das Neuronale Netz eingegeben und der Output mit dem tatsächlichen Ergebnis verglichen. Je nachdem mit wie vielen Datensätzen schon trainiert wurde, ist die Abweichung größer oder kleiner. Bei dieser Abweichung handelt es sich um den Fehler, also um die Differenz zwischen dem erwarteten und dem errechneten Ergebnis. Es muss nicht immer nur mit der Differenz gearbeitet werden. Abhängig vom Design des Neuronalen Netzes und von dem angewandten Lernverfahren kann nicht nur mit der Differenz, sondern auch mit dem quadrierten Fehler oder anderen Varianten trainiert werden. Anschließend an seine Berechnung wird der Fehler mittels des sogenannten *Backwardpass* durch das Netz geführt und auf diese Weise auf die Gewichte der Verbindungen verteilt, um diese anzupassen. Entsprechend lässt sich festhalten, dass der Output beim *Backwardpass* quasi als Input fungiert. [Zup94, vgl. S. 5ff]

### 3 Konzeption

Nachdem zuvor die Grundlagen erläutert wurden, soll in diesem Kapitel die Problemstellung konkretisiert und ein Entwurf zu ihrer Lösung präsentiert werden. Dabei wird zunächst allgemein beschrieben, was eine entsprechende Softwarelösung umsetzen müsste, bevor anschließend im Entwurf konkrete Anforderungen dazu formuliert werden.

#### 3.1 Problemstellung

Rundenbasierte Strategiespiele sind seit geraumer Zeit ein fundamentaler Bestandteil der Videospielindustrie. Sie existieren in unterschiedlichen Variationen und der Erfolg des Spielers ist bei diesen Spielen maßgeblich von der gewählten Strategie abhängig. Seit den 1990er-Jahren werden vermehrt computergesteuerte Spieler als Gegner für den Menschen eingesetzt. Als Beispiel hierfür lässt sich u.a. die Entwicklung von KI-Spielern im rundenbasierten Strategiespiel *Civilization: Call To Power* anführen ([Dav99, vgl. S. 1]).

Die Entwicklung von KI-Spielern, die in den 1990er-Jahren eine Herausforderung für Industrie und Forschung darstellte, ist heute Alltag geworden und beinahe jedes Strategiespiel, ob rundenbasiert oder kontinuierlich, verfügt über computergesteuerte Spieler. Um die Entwicklung solcher KIs auch in der universitären Lehre behandeln zu können, erscheint es sinnvoll, den Studenten eine Möglichkeit dazu zur Verfügung zu stellen, ihre eigens entwickelte KI zu testen und gegeneinander antreten zu lassen.

Aus dieser Idee ergeben sich mehrere Anforderungen, deren Erfüllung eine entsprechende Software gewährleisten sollte, und die 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.
6. Die Software muss einen Sieger feststellen.

Aus der obigen Konkretisierung der Problemstellung lässt sich erkennen, dass die genannte Software sowohl das Spiel selbst umfassen als auch eine Schnittstelle für die entwickelten KI-Spieler bereitstellen sollte. Insgesamt lässt sich die Problemstellung zusammenfassen, indem man festhält, dass eine Software entwickelt werden muss, die aus zwei Komponenten besteht (rundenbasiertes Strategiespiel + KI-Schnittstelle) und die oben genannten Aspekte berücksichtigt.

#### 3.2 Entwurf

Im Hinblick auf die in Kapitel 3.1 erläuterte Problemstellung und die damit einhergehenden Anforderungen, wird nachfolgend beschrieben, wie diese gelöst und umgesetzt werden kann bzw. können. Im ersten Schritt wird dazu eine grobe Spezifikation erstellt, ausgehend von welcher anschließend weiterführende Details erläutert werden. Die beiden Komponenten der Software werden im Folgenden als “Spiel” und “Schnittstelle” bezeichnet.

## Das Spiel:

Als Grundlage für das Spiel wird das rundenbasierte Brettspiel *Siedler von Catan* genutzt. Zur 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 gleichzeitig die Einhaltung mehrerer Anforderungen sichergestellt werden. Durch das Umsetzen der Spielregeln werden bereits mindestens zwei Spieler garantiert. Zusätzlich sorgt dies für ein möglichst faires Spiel, welches zudem rundenbasiert ist und unerwünschtes Verhalten weitestgehend ausschließt. Da KI-Spieler je nach Lernverfahren sehr stark darin sein können, Lücken in den Regeln zu finden, kann allerdings auch durch eine strenge Implementierung nicht vollständig verhindert werden, dass es zu einem solchen unerwünschtem Verhalten kommt. . Insgesamt lassen sich auf Grundlage der in Kapitel 3.1 genannten Aspekte folgende konkrete Anforderungen für das Spiel formulieren:

### 1. Titel: Generierung

**Beschreibung:** Das Spiel muss nach den Regeln des Brettspiels ein Spielfeld generieren.

### 2. Titel: Straßenbau

**Beschreibung:** Das Spiel muss den Bau von Straßen erlauben.

### 3. Titel: Siedlungsbau

**Beschreibung:** Das Spiel muss den Bau von Siedlungen erlauben.

### 4. Titel: Städtebau

**Beschreibung:** Das Spiel muss den Bau von Städten erlauben.

### 5. Titel: Gewinnermittlung

**Beschreibung:** Das Spiel muss am Ende des Spiels einen Gewinner ermitteln.

### 6. Titel: Zug

**Beschreibung:** Das Spiel muss jedem Spieler in jeder Runde einen Zug bereitstellen, der mindestens das Rollen zweier Würfel und die anschließende Verteilung von Rohstoffkarten sowie optional den Kauf von Objekten beinhaltet.

### 7. Titel: Mindestabstand

**Beschreibung:** Das Spiel muss gewährleisten, dass zwischen Siedlungen bzw. Städten mind. 2 Straßen Platz ist.

### 8. Titel: Upgrades auf Städte

**Beschreibung:** Das Spiel muss Upgrades auf Städte erlauben.

### 9. Titel: Längste Handelsstraße

**Beschreibung:** Das Programm muss die „Längste Handelsstraße“ berechnen.

Neben den genannten Mindestanforderungen lassen sich die folgenden Soll- bzw. Kann-Anforderungen formulieren, die durch ihre Umsetzung die Komplexität des Spiels erhöhen und so ausgeklügeltere Strategien seitens der KI erfordern. Das Ziel der Softwareentwicklung ist es, alle Muss-Anforderungen zu implementieren und dann ausgewählten Soll- und Kann-Anforderungen zu ergänzen. Hierbei wird auf die Erfüllung mindestens zweier Kann-Anforderungen abgezielt.

### 1. Titel: Reale Spieler

**Beschreibung:** Das Spiel soll es ermöglichen, dass reale Spieler das Spiel spielen können.

**2. Titel:** Gemischte Duelle

**Beschreibung:** Das Spiel soll es ermöglichen, dass KI und reale Spieler gegeneinander antreten können.

**3. Titel:** Funktion der Ereigniskarten

**Beschreibung:** Das Spiel kann die Funktion der Ereigniskarten integrieren.

**4. Titel:** Ausspielen und Sammeln von Ereigniskarten

**Beschreibung:** Das Spiel kann das Sammeln und Ausspielen von Ereigniskarten ermöglichen.

**5. Titel:** Der Räuber

**Beschreibung:** Das Spiel kann über einen Dieb verfügen, der einzelne Ressourcenfelder unbrauchbar macht, solange er das jeweilige Feld besetzt und immer dann von einem Spieler versetzt werden kann, wenn er eine Sieben würfelt.

**6. Titel:** Mehr als zwei Spieler

**Beschreibung:** Das Spiel kann es ermöglichen, dass mehr als zwei Spieler gegeneinander antreten können.

**7. Titel:** Handel

**Beschreibung:** Das Spiel kann Handel zwischen den Spielern ermöglichen.

**8. Titel:** Gewinnwahrscheinlichkeiten

**Beschreibung:** Das Spiel kann aktuelle Gewinnwahrscheinlichkeiten ausgeben.

**9. Titel:** Initiale KIs

**Beschreibung:** Das Spiel kann über initiale KIs verfügen.

**Die Schnittstelle:**

Die Schnittstelle zur Kommunikation der KI(s) mit dem Spiel ist äquivalent zu Maus, Tastatur und Bildschirm für die Kommunikation zwischen Mensch und Spiel anzusehen. Eine Maschine, die das Spiel spielt, verfügt nicht über die gleichen Sinne wie ein Mensch, weshalb ein Weg gefunden werden muss, der es der Maschine ermöglicht, dennoch möglichst effizient Züge ausführen und Informationen erhalten zu können.

In diesem Fall wird die Sprache JSON für die Schnittstelle verwendet. Dementsprechend 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 zu erfüllen sind, damit die Schnittstelle wie angedacht funktioniert. Im Folgenden sind alle Anforderungen aufgelistet:

**1. Titel:** Austausch der KIs

**Beschreibung:** Die Schnittstelle muss es ermöglichen, dass die KIs ohne großen Aufwand ausgetauscht werden können und die Verwendung der Schnittstelle hierbei kein großes Hindernis darstellt.

**2. Titel:** Programmiersprachen

**Beschreibung:** Die Schnittstelle muss es ermöglichen, dass die KIs in unterschiedlichen Programmiersprachen geschrieben werden können, sofern sie das richtige Format zur Kommunikation verwenden.

**3. Titel:** Funktion von KIs

**Beschreibung:** Die Schnittstelle muss es ermöglichen, dass statt realer Spieler zwei KIs gegeneinander antreten.

**4. Titel:** Grundsatz der Fairness

**Beschreibung:** Die Schnittstelle muss gewährleisten, dass eine KI durch die Verwendung der Schnittstelle gegenüber einem realen Spieler im Grunde die gleichen Möglichkeiten hat.

**5. Titel:** Zugänglichkeit von Informationen

**Beschreibung:** Die Schnittstelle muss Informationen vor der Einsicht anderer Spieler geschützt versenden. Damit ist gemeint, dass Informationen selektiv an einen bestimmten Spieler zu vergeben sind.

**6. Titel:** Bereitstellung von gleichen Informationen

**Beschreibung:** Die Schnittstelle muss grundlegend dazu fähig sein, jedem KI-Spieler die gleichen Informationen bereitzustellen, auf die auch ein menschlicher Spieler Zugriff hätte. Das bedeutet, das gewählte Kommunikationsformat muss dazu geeignet sein, diese Informationen zu übermitteln.

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

**1. Titel:** Ereigniskarten für KI-Spieler

**Beschreibung:** Die Schnittstelle soll die Informationen über Ereigniskarten an KI-Spieler übermitteln und Anweisungen über diese von KI-Spielern annehmen. Dies gilt, sofern Anforderung 3 und 4 des Spiels erfüllt werden.

**2. Titel:** Räuber für KI-Spieler

**Beschreibung:** Die Schnittstelle soll es ermöglichen, dass sich ein KI-Spieler die Funktion des Räubers zunutze machen kann, wenn er eine Sieben würfelt. Dies gilt, wenn Anforderung 5 an das Spiel erfüllt wird.

**3. Titel:** Anwendung der Schnittstelle auf einzelne Spieler

**Beschreibung:** Die Schnittstelle kann es realen und computergesteuerten Spielern gleichzeitig ermöglichen, das Spiel zu spielen. Dies gilt, wenn Anforderung 9 an das Spiel erfüllt wird.

### 3.3 Die Regeln und Anpassungen

In diesem Kapitel wurden bereits die allgemeinen Anforderungen an die zu entwickelnde Software erläutert. Dieser Abschnitt soll nun aufbauend darauf auf die Regeln des Spiels *Siedler von Catan* eingehen und erklären, ob und wie einige dieser Regeln umgesetzt werden können. Allgemein wird Bezug auf das Regelbuch von Die Siedler von *Die Siedler von Catan Almanach* (s. Anhang) genommen, welches in voller Länge im Anhang zu finden ist.

#### Der Spieldatenbau:

Das Spielfeld besteht aus 19 Landschaftsfeldern, die wie folgt unterteilt sind: Viermal Wald, viermal Weideland, viermal Ackerland, dreimal Hügelland, dreimal Gebirge und einmal Wüste. Zusätzlich besteht das Spielfeld aus 18 Wasserfeldern, wovon neun über

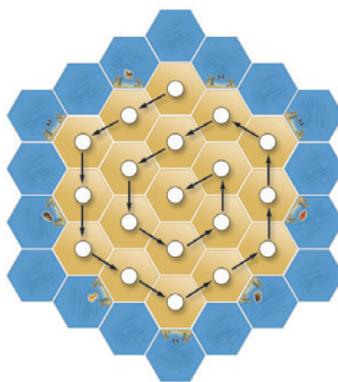


Abbildung 3.1: Verteilung der Zahlencips auf dem Spielfeld (siehe Anhang Regelwerk S. 4)

einen Hafen zum Handeln verfügen. Während die Landschaftsfelder in der späteren Implementation genauso verwendet werden, ist die Implementation der Wasserfelder davon abhängig, inwiefern die Soll-/Kann-Anforderungen umgesetzt werden. Außerdem soll auch der sogenannte Zufallsfaktor in der programmierten Variante des Spiels enthalten sein, sodass ? wie im Originalspiel - die 19 Landschaftsfelder zu Beginn zufällig angeordnet werden, wodurch neu begonnene Spiele sich mit hoher Wahrscheinlichkeit von dem vorherigen abheben.

Des Weiteren müssen die Zahlencips auf die Landschaftsfelder verteilt werden. Jedes Mal, wenn eine Zahl gewürfelt wird und ein Spieler eine Siedlung oder eine Stadt auf einem Landschaftsfeld besitzt, auf dem ein Zahlenchip liegt, dessen Aufschrift der Summe der Augen beider Würfel entspricht, erhält dieser Spieler den entsprechenden Rohstoff: eine Einheit des jeweiligen Rohstoffes für jede Siedlung, die er auf diesem Feld besitzt; zwei Einheiten für jede Stadt.

Die Zahlencips 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 ringsherum gegen den Uhrzeigersinn verteilt. Auffällig ist hierbei, dass statt 19 Chips nur 18 existieren, was darauf zurückzuführen ist, dass die Wüste frei bleibt, da diese keine Ressourcen liefert. Die Reihenfolge der Verteilung der Zahlenchips wird in Abb. 3.1 verdeutlicht.

### **Der Spielbeginn:**

Die ersten beiden Züge jedes Spielers laufen anders als die restlichen Züge des Spiels 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 die Spieler ihre Gebäude nun in entgegengesetzter Reihenfolge platzieren dürfen. Nach dem zweiten Spielzug jedes Spielers erhält der jeweilige Spieler entsprechend Ressourcen für seine zweite Siedlung (jedes angrenzende Landschaftsfeld außer der Wüste bringt ihm eine Ressource). In diesen ersten beiden Spielzügen wird weder gewürfelt, noch werden Ereigniskarten erworben.

### **Gewöhnliche Spielzüge:**

Die Reihenfolge, in der die Spieler ihre Züge machen, entspricht nun wieder der der ersten Spielrunde. Die gewöhnlichen Spielzüge laufen von nun an bis zum Ende des Spiels auf gleiche Weise ab: Zu Beginn seines Zuges würfelt der Spieler, woraufhin alle Spieler , wie bereits beschrieben, Ressourcen erhalten. Anschließend kann der Spieler, der jeweils am Zug ist, entweder etwas bauen oder eventuell handeln. Abschließend beendet der Spieler

seinen Zug, sodass der Nächste dran ist. Im Detail läuft ein gewöhnlicher Spielzug wie folgt ab:

1. **Würfeln:** Jeder Zug eines Spielers muss damit beginnen, dass er mit zwei Würfeln würfelt. Anhand der Summe der Augen beider Würfel, 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 die entsprechenden Ressourcen bzw. für jede Stadt jeweils zwei der entsprechenden Ressourcen.
2. **Bauen:** Nachdem gewürfelt wurde, kann gebaut werden. Zum Bau stehen Straßen, Siedlungen, Städte und Entwicklungen (wenn diese Anforderung umgesetzt wird) zur Verfügung, die mit den vorhandenen Ressourcen gekauft werden können.
3. **(Handeln:)** Handeln kann entweder zum Abschluss des Spielzuges erfolgen oder bevor der Spieler baut. Da es sich hierbei jedoch um eine Kann-Anforderung handelt, wird dieser Spielbestandteil nur in Klammern angeführt. Ein Spieler kann entweder an den Häfen zu den jeweils angegebenen Verhältnissen,, mit der Bank im Verhältnis 4:1 oder mit einem anderen Spieler in einem beliebigen Verhältnis tauschen.

#### **Ende des Spiels:**

Das Spiel endet, wenn einer der Spieler 10 Siegpunkte erreicht hat. Siegpunkte werden für unterschiedliche Errungenschaften im Spiel vergeben, so zum Beispiel für das Errichten von Siedlungen und Städten, aber auch für die *Längste Handelsstraße* und die *Größte Rittermacht*. Für die Implementation kann die Vergabe der Siegpunkte danach angepasst werden, wie die Strategiefindung der KI-Spieler beeinflusst werden soll. Interessant wäre hierbei zu sehen, ob bei verschiedenen Siegbedingungen unterschiedliche KIs einen Vorteil haben. Es ist möglich, dass eine KI, die das Bauen von Siedlungen favorisiert, mehr Spiele gewinnt, wenn dies mit mehr Siegpunkten belohnt wird. Stattdessen würde eine KI, die Wert auf längere Handelsstraßen und mehr Ereigniskarten legt, bei einer dieser Strategie begünstigenden Gewichtung vermehrt siegen.

#### **Baukosten:**

Es wurde bereits beschrieben, dass das Bauen Teil eines gewöhnlichen Spielzugs ist. Zum Bauen können alle fünf verfügbaren Ressourcen verwendet werden: Lehm, Holz, Schafe, Getreide und 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, 3 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 sind zwei Varianten vorstellbar: Entweder wird der Räuber im vollen Umfang, was bedeutet, dass ein Spieler, der eine Sieben würfelt, den Räuber auf ein Feld setzen darf, welches dadurch keine Ressourcen mehr liefert. Außerdem könnte der jeweilige Spieler in dieser Variante einen gegnerischen Spieler auswählen, der eine Siedlung oder eine Stadt an diesem Landschaftsfeld hat und ihm eine zufällig ausgewählte Ressource stehlen. Alternativ hierzu kann 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. Häfen sind Teil der Kann-Anforderungen und somit nicht zwingend Teil des Spiels. Sollten sie jedoch implementiert werden, sind verschiedene Stufen der Implementation denkbar: Die Mindestanforderung wäre, dass ein Spieler nur an einem Hafen handeln kann, wenn dieser eine Siedlung oder eine Stadt im Einzugsbereich dieses Hafens besitzt.

Im Brettspiel gibt es mehrere Arten von Häfen. Einige dieser Häfen sind an Rohstoffe gebunden und können im Verhältnis 2:1 für den Handel genutzt werden. Das bedeutet, beim Handel an einem Holzhafen würde eine Einheit Holz zwei Einheiten eines anderen Ressourcen kosten. Neben diesen gebundenen Häfen existieren auch ungebundene. Diese bieten ein Verhältnis von 3:1, sodass drei Einheiten eines Rohstoffes für eine Einheit eines beliebigen anderen Rohstoffes eingetauscht werden können. Als Abstufungen für die Implementation ist es denkbar, zunächst nur die ungebundenen und anschließend auch die gebundenen Häfen umzusetzen.

Betrachtet man den Handel an Häfen im gesamten Kontext des Spiels, lässt sich feststellen, dass die Kann-Anforderung *Handel* zumindest als spätere Weiterentwicklung der Software zu empfehlen ist. Hierbei ist wahrscheinlich der Handel mit der Bank am wichtigsten, gefolgt vom Handel an den Häfen und dem Handel zwischen den Spielern. Eventuell könnte der Handel zwischen den Spielern auch höher als der Bank- und Hafenhandel priorisiert werden, da dieser einen interessanten strategischen Aspekt darstellt.

Er erscheint jedoch sinnvoll, den Bankhandel als erste Erweiterung der Software zu Verfügung zu stellen, da es ohne diesen zu Konstellationen kommen kann, in denen ein oder mehrere Spieler im Spiel nicht mehr vorankommen können. Dieser Zustand kann entstehen, wenn die Anfangssiedlungen ungeschickt platziert werden und kein Zugang zu den - für Straßen und Siedlungen nötigen - Rohstoffen besteht. Für einen Spieler, der sich in einer solchen Situation befindet, ist Handel der einzige Ausweg, da es ansonsten keine realistische Chance für ihn gibt, das Spiel noch für sich zu entscheiden. Eine genauere Erläuterung dieser Erweiterung ist in Kapitel 6.2 zu finden.

### 3.4 Konzeption der Schnittstelle

In diesem Kapitel wurden zwar bereits Aussagen über die Schnittstelle getroffen und Anforderungen an diese gestellt, jedoch wurde noch nicht erläutert, wie diese realisiert werden kann. Entsprechend soll letzteres hier nachgeholt werden, wobei zu beachten ist, dass die Umsetzung der Schnittstelle zur Erfüllung der Anforderungen in einer bestimmten Weise erfolgen muss: Betrachtet man nochmals die gestellten Anforderungen, so lassen sich die ersten beiden bereits dadurch lösen, dass JSON als Kommunikationsmedium genutzt wird. Anforderung 3 (s.o.) hängt eng mit einer korrekten Umsetzung zusammen, was zwar wichtig ist, jedoch das Konzept an sich nicht beeinflusst. Aus Anforderung 4 (s.o.) hingegen lassen sich deutlich mehr Aspekte extrahieren. Diese besagt, dass eine KI einem realen Spieler gegenüber keine Vor- oder Nachteile haben darf, die allein dadurch entstehen, 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 wie ein menschlicher Spieler hat. 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 des Spielers verarbeitet werden, der auch gerade einen Spielzug ausführt. Das Zustand eines Zuges selbst, ist jedoch auch von Relevanz, da ein Spielzug immer mit dem Rollen der Würfel beginnen muss. Am besten ließe sich dieses Problem dadurch lösen, Anfragen nur in einer bestimmten

Reihenfolge zu erlauben: Zunächst würde gewürfelt und erst anschließend daran könnte gehandelt oder gebaut werden, ein erneutes Würfeln würde dabei verhindert. Beendet die KI ihren Spielzug, verlöre sie die Berechtigung dazu, Anweisungen jeglicher Art ausführen zu lassen, wieder. Zudem sind in den ersten beiden Zügen jedes Spielers Aktionen erlaubt, die in den restlichen Aktionen nicht zugelassen werden würden (siehe Kapitel 3.3), was auch einen Einfluss auf das Annehmen einiger Anfragen hat. Optimal ließe sich dieses Problem lösen, indem im Spiel selbst eine Verzahnung der KIs und der menschlichen Spieler erfolge. Damit ist gemeint, dass die gleichen Bedingungen für KIs genutzt werden, die auch einem realen Spieler Funktionen freischalten oder sperren.

Ein weiterer wichtiger Aspekt, der durch die Anforderungen aufgegriffen wird, ist der Umgang mit Informationen. Anforderung 4 (s.o.) beschreibt zusätzlich, dass eine KI zu keinem Zeitpunkt im Spiel prinzipiell bevorteilt oder benachteiligt werden darf. Wichtig ist hierbei, dass die Schnittstelle nicht dauerhaft alle Informationen senden muss, sondern dass das grundlegende Vorhandensein der Möglichkeit ausreicht. Wahrscheinlich werden die initialen KIs nicht alle Informationen benötigen bzw. verarbeiten können, die ein Mensch nutzen würde. Allerdings sollte das Format grundlegend auch umfangreichere Informationen übermitteln können. 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. Genauso sollte auch mit einem KI-Spieler verfahren werden. Wenn die KI Informationen anfragt, auf die auch ein realer Spieler Zugriff hat, werden diese übermittelt, aber sie werden nicht dauerhaft gesendet. Diese Überlegung könnte dazu führen, dass der Zeitpunkt des Anfragens zu einem strategischen Element wird - allerdings nur dann, wenn die Anzahl der Anfragen begrenzt wird, was hier nicht der Fall ist, da ein realer Spieler auch in der Lage ist, Informationen beliebig oft wahrzunehmen.

Anforderung 5 (s.o.) geht in die gleiche Richtung wie die eben beschriebene, wobei hier der Fokus auf dem Schützen der versendeten Informationen liegt. 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, sollte den Gegnern einen Einblick in die Strategie eines Spielers ermöglichen. Daher ist es wichtig, die Antwort des Spiels als Einzelnachricht zu realisieren und nicht allen in Form eines Broadcasts mitzuteilen. Es klingt möglicherweise trivial, soll hier aber erwähnt werden, weil es einen fundamentalen Beitrag zur Fairness liefert.

Aus den Ausführungen Anforderung 4 und 5 (s.o.) geht zudem die Einhaltung der Anforderung 6 (s.o.), realen und KI-Spielern grundlegend die gleichen Informationen bereitzustellen zu können wie realen Spielern, hervor.

Durch die Einhaltung der beschriebenen Anforderungen seitens des Spiels wird den KIs eine hohe Gestaltungsfreiheit gelassen. Dadurch, dass die KIs entscheiden können, wann sie Informationen über Ressourcen abfragen, müssen sie ihre Strategie nicht an den Zeitpunkt des Erhaltens der Information anpassen. Dennoch gibt es einige Anweisungen, die das Spiel an die KI-Spieler senden muss, ohne auf Anfragen von diesen zu warten: Das betrifft zum Beispiel die Information darüber, ob ein Spieler gerade an der Reihe ist. Hier würde eine Abfrage durch jeden Spieler keinen Sinn ergeben, weil diese dauerhaft erfolgen müssten. Einfacher ist es für den Ablauf des Spiels, wenn das Spiel nach jedem Spielerwechsel eine Nachricht an die Spieler sendet. Ferner müssen plötzliche Ereignisse sofort an die Spieler übermittelt werden. Das beste Beispiel hierfür ist das Nutzen des Räubers. Wenn der Räuber versetzt wird und einem Spieler eine Ressourcenkarte entwendet wird, muss dies ? zumindest den betroffenen Spielern - unmittelbar mitgeteilt werden, da dies mög-

licherweise wichtig für die Strategiebildung ist. Allgemein könnte auch eine Art doppelte Absicherung eingeführt werden, bei denen Änderungen an den Beständen von Ressourcen eines Spielers direkt übermittelt werden, aber zusätzlich Abfragen durch die KI erfolgen können, um es dieser zu ermöglichen, vor dem Bau von Siedlungen, Straßen, etc. auf jeden Fall Informationen über die aktuelle Zahl an verfügbaren Ressourcen zu haben.

Neben den hier offensichtlichen Anfragen und Antworten gibt es weitere fundamentale Informationen, über die jede KI verfügen muss. Die fundamentalste ist hierbei die über den 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 das Spielfeld danach gleichbleibend ist, sollte eine Art Initialisierungsnachricht an die KI-Spieler gesendet werden, sobald das Spiel fertig initialisiert ist, um das Entwickeln einer anfänglichen Strategie zu ermöglichen. Zur Veranschaulichung folgt eine Tabelle aller Anfragen und Antworten, inklusive der jeweiligen Zeitpunkte, Absender und Empfänger. Die gezeigten Nachrichten verdeutlichen, wie umfangreich die Kommunikation zwischen Spiel, Schnittstelle und dem computergesteuerten Spieler werden kann. Es wird jede Tätigkeit abgebildet, die auch einem menschlichen Spieler zur Verfügung stehen würde. Dies würde der Implementierung aller Anforderungen gleichkommen.

Tabelle 3.1: Geplante Kommunikation zwischen Schnittstelle und KI-Spieler

Nachricht	Absender	Empfänger	Zeitpunkt	Aktion/Antwort
Aufbau Spiel-feld	Spiel	Spieler	Nach Initiali-sierung	Empfangs-be-stätigung
Siedlungsbau-menü 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 Ereignis-karte	Spieler	Spiel	Im eigenen Zug nach Würfeln	Ereigniskarte
Einsatz Ereig-niskarte	Spieler	Spiel	Im eigenen Zug nach Würfeln	Effekt
Verschiebung Räuber	Spiel	Spieler	Bei gewürfelter Sieben oder Ereigniskarte	Spiel öffnet Verschiebungs-menü für Spieler

Tabelle 3.2: Geplante Kommunikation zwischen Schnittstelle und KI-Spieler (Fortführung)

Diebstahl Res- source	Spiel	Spieler	Nach Verschie- bung des Räu- bers	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
Ressourcen- abfrage	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

## 4 Implementation

Dieses Kapitel soll sich mit der Umsetzung der im Entwurf beschriebenen Software befassen. Es werden das Spiel, die Schnittstelle und die initialen KI-Spieler erläutert, wobei chronologisch vorgegangen wird, sodass zunächst eine grundlegende Version des Spiels beschrieben und anschließend auf die Umsetzung der Schnittstelle eingegangen wird. Anschließend wird die Weiterentwicklung des Spiels und seiner anderen Bestandteile fokussiert. Die Implementation der initialen KIs wird zuletzt betrachtet.

Nach der Erläuterung der Implementation werden Anpassungen beschrieben, die zuvor zwar nicht so geplant wurden, während der Umsetzung jedoch trotzdem Einzug ins Programm erhielten bzw. ausgelassen wurden.

### 4.1 Verwendete Technologien

Zu Beginn des Kapitels 3.2 wird bereits thematisiert, dass für die Erstellung der Software *Unity*, *C#* und *JSON* für die Schnittstelle verwendet werden sollen. In diesem Abschnitt soll noch einmal genauer auf die verwendeten Technologien eingegangen und die Entscheidung für diese Technologien begründet werden.

*Unity* ist eine Spieleanwendungsengine, die eine schnelle und einfache Entwicklung komplexer Spieler ermöglicht. Zur Wahl standen entweder die *Unity*-Engine oder die *Unreal*-Engine. Obwohl für dieses Projekt beide Optionen infrage gekommen wären, fiel die Entscheidung auf *Unity*, welches sich vor allem durch den sehr umfangreichen *Asset-Store* und somit durch die Möglichkeit, einfacher Importe vielerlei kostenloser Grafiken, Animationen und Sounds sowie die Entwicklung beschleunigenden Funktionalitäten, aus. Außerdem ist *Unity* subjektiv gesehen benutzerfreundlicher als die *Unreal*-Engine.

Auch *Unreal* bietet einige Vorteile gegenüber *Unity*, die jedoch für diese Arbeit keinen nennenswerten Mehrwert bieten, weshalb die Entscheidung im Hinblick auf die Ziele dieser Arbeit auf die Verwendung von *Unity* fiel, zumal 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, seit der Veröffentlichung dieser Version ist *C#* jedoch die einzige verwendbare Programmiersprache. Entsprechend fiel die Wahl der Programmiersprache für dieses Projekt auf *C#*.

*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. *JSON* soll durch das folgende Beispiel veranschaulicht werden:

#### KI-Anfrage:

```
{  
  "spieler": "1"  
  "aktion": "Wuerfeln"  
}
```

#### Spiel-Antwort:

```
{
```

```

    "augenzahl": 8
    "ressourcen": [
        {
            "name": "lehm"
            "anzahl": 2
        },
        {
            "name": "holz"
            "anzahl": 1
        }
    ]
}

```

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 der geschweiften Klammern Variablen definiert werden, die einen Wert enthalten. Auffällig ist insbesondere, dass nicht typisiert wird, sondern ohne Auskunft über den Variablentyp verschickt wird.

Eine Zeile, die ein Attribut beschreibt, besteht aus dem Namen der Variable in Anführungszeichen, gefolgt von einem Doppelpunkt und dem zugewiesenen Wert. Zeichenketten wie der Name eines Attributes werden in Anführungszeichen gesetzt, wohingegen Zahlen ohne dastehen. Deutlich wird dies bei den Attributen *spieler* (KI-Anfrage) und *augenzahl* (Spiel-Antwort). Im ersten Fall handelt es sich um einen Namen, im zweiten um eine Anzahl.

Des Weiteren lassen sich Listen per *JSON*-Format verschicken. So enthält die zweite Datei eine Liste der Ressourcen eines Spielers und schickt ihm diese als Teil der Antwort zu. Es werden nachfolgend aus Platzgründen nur zwei Ressourcen beispielhaft angegeben. Listen werden mit eckigen Klammern geöffnet und geschlossen. Innerhalb einer solchen Sektion werden einzelne Objekte durch Kommata getrennt und von geschweiften Klammern umschlossen definiert.

Ob in diesem Fall die Verwendung einer Liste nötig wird, ist fraglich, doch dieses Beispiel dient, wie erwähnt, lediglich der Veranschaulichung und muss daher nicht zwingend in gleicher Weise später implementiert werden. Weiterhin ist zu diskutieren, ob das Spiel überhaupt mit den Ressourcen eines Spielers auf das Würfeln antworten muss oder ob hierfür eine weitere Anfrage erfolgen sollte.

## 4.2 Die grundlegende Umsetzung des Spiels

In Abschnitt 4.1 wurde auf die Gründe der Verwendung von *Unity* für die Umsetzung des Spiels eingegangen. Einer ist, dass mit verhältnismäßig geringem Aufwand eine ansprechende Nutzeroberfläche gestaltet werden kann. Die Grafiken und die Formen der Objekte, die im Spiel verwendet werden, stammen aus einer anderen Implementation von *Siedler von Catan*, die auf Github unter <https://github.com/buttsj/unity-settlers-of-catan> zu finden ist ([But16]).

Abbildung 4.1 zeigt eine so gestaltete Oberfläche. Hierbei fallen mehrere Aspekte ins Auge. Zunächst einmal das Spielfeld selbst. Dieses besteht wie das Original aus insgesamt 19 sechseckigen Einzelteilen. Die Verteilung der Sechsecke erfolgt nach folgender Verfügbarkeit: viermal Wald, viermal Getreide, dreimal Ziegel, dreimal Stein, dreimal Schaf, einmal Wüste. Auffällig ist, dass das Wüstenfeld als einziges ohne Nummer bleibt, was daran liegt,

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

---



Abbildung 4.1: Nutzeroberfläche zu Beginn des Spiels ([But16, vgl.])

dass es keine Ressourcen liefert, wenn ein Spieler eine Siedlung an diesem Feld gründet. Links ist eine Art Pyramide, die eine Siedlung darstellen soll, zu sehen. 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 (Würfeln und Zug beenden) und eine Anzeige, die die verfügbaren Ressourcen des aktuellen Spielers darstellt, zu sehen.

#### 4.2.1 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,
20            hex11, hex12,
21            hex13, hex14, hex15, hex16, hex17, hex18, hex19
22        };
23    }
```

```

22     List<Material> numbers = new List<Material>()
23     {
24         num5, num2, num6, num3, num8, num10, num9, num12, num11, num4,
25             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 4.1: Spielfeld-Generator Startmethode

Listing 4.1 und 4.2 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 beispielsweise alle Sechsecke, sodass diese später angesteuert werden können. Eine andere speichert die Materialien der Nummern auf den Sechsecken. Zudem stehen zwei Listen zur Verfügung, 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
28         //der Nummer-Blueprint gelöscht
29     }
30 }
31 }
```

---

Listing 4.2: Spielfeld-Generator CreateMap-Methode

Listing 4.2 zeigt wie die Oberfläche 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 zunächst die Oberfläche des aktuellen Sechsecks als Variable sowie die Nummer auf dem Sechseck zwischengespeichert. 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 aus dem beiden Listen entfernt. Sollte es sich bei dem betrachteten Sechseck um die Wüste gehandelt haben, so wird das Plättchen auf diesem Feld gelöscht, damit es nicht ohne Zahl auf dem Feld zurückbleibt und die Sichtbarkeit der Wüste verringert wird.

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 Integerwertes über keinerlei Funktionalität und ist daher nur als Hilfsklasse zu bezeichnen, weshalb sie nur im Anhang abgebildet ist..

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

#### 4.2.2 Der Spieler

Um das Spiel spielen zu können, muss es eine Repräsentation des computergesteuerten oder menschlichen Spielers 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 theoretisch 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 4.3: Attribute des Spielers

In Listing 4.3 ist der Beginn des *PlayerScripts* zu sehen. Es verfügt über eine Vielzahl an Attributen, die unterschiedlichen Funktionen dienen. So braucht es einen eigenen Vector3, der die Perspektive, aus der der Spieler auf das Spielbrett schaut, beschreibt und auf die die ebenfalls als Attribut gespeicherte *Camera* angewendet werden kann. Des Weiteren verfügt der Spieler über die Vorlage (den Prefab) zum Bau von Siedlungen (*GameObject village*) und Straßen (*GameObject road*), sowie über eine Farbe, in der später seine Objekte auf dem Spielfeld erscheinen.

Zudem müssen seine verfügbaren Ressourcen gespeichert und eine Möglichkeit der Ausgabe durch Textfelder gegeben werden. Zusätzlich stellen die Variablen *isFirstTurn* und *isSecondTurn* eine Möglichkeit dar, zu prüfen, ob der Spieler sich in einem der ersten beiden Züge befindet - eine Zeit, in der Spieler kostenlos bauen darf. Dieses kostenlose

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

---

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`) sowie die Anzahl an Siegpunkten und die Länge der längsten Straße des Spielers speichern.

---

```
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 4.4: Initialisierung des Spielers

Die in Listing 4.4 abgebildete Methode `private void Awake()` initialisiert einige der Attribute auf ihre Anfangswerte, die Methode `void Start()` (ebenfalls in Listing 4.4 abgebildet) erreicht im Grunde das Gleiche, wird aber nach der Awake-Methode aufgerufen. Dies ist an dieser Stelle notwendig, um sichergehen zu können, dass die Text-Objekte bereits erzeugt wurden, wenn auf diese zugegriffen wird. Außerdem werden die Liste der Siedlungen und die der Straßen jeweils als leere Listen 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 4.5: Attribute und Initialisierung des Spielers

In Listing 4.5 sind die Methoden gezeigt, die zu Beginn eines Zuges aufgerufen werden müssen. *FirstTurn()* wird nur im ersten Zug aufgerufen, *SecondTurn()* nur im zweiten und *Turn()* zu Beginn jedes anderen Zuges. Alle drei Methoden beginnen damit, dass sie die Methode *AdjustCamera()* aufrufen. Hierbei handelt es sich um eine 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 daher *freeBuild* auf *true* gesetzt, *freeBuildRoad* jedoch auf *false*. *freeBuildRoad* wird erst *true*, sobald die kostenlose Siedlung des aktuellen Zuges gebaut wurde. In *Turn()* darf nicht kostenlos gebaut werden, weshalb *freeBuild* und *freeBuildRoad* auf *false* gesetzt werden und bis zum Ende des Spiels auf diesem Wert bleiben.

---

```

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 4.6: Attribute und Initialisierung des Spielers

Nachdem Listing 4.5 und Listing 4.3 die Initialisierung des Spielers und seiner Züge zeigten, bringt Listing 4.6 die erste Funktionalität mit sich: Abgebildet sind zwei Methoden. Erstere (*HasResourcesForVillage()*) überprüft, ob der Spieler genug Ressourcen zum Bau einer Siedlung hat, die zweite Methode (*HasResourcesForRoad()*) bewerkstelligt dies analog für die Straßen. Für eine Siedlung benötigt der Spieler jeweils eine Einheit Lehm, Holz, Schafe und Getreide. Für eine Straße nur jeweils eine Einheit Holz und Lehm. Über die Ressourcenvariablen wird abgefragt, ob diese Rohstoffe vorhanden sind. Dementsprechend wird ein bool-Wert zurückgegeben.

---

```

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)
```

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

---

```
8         {
9             Debug.Log(tile.GetComponentInChildren<Renderer>().sharedMaterial.name);
10            if
11                (tile.GetComponentInChildren<Renderer>().sharedMaterial.name
12                 == "wheat 1")
13                {
14                    wheat++;
15                }
16            else if
17                (tile.GetComponentInChildren<Renderer>().sharedMaterial.name
18                 == "sheep")
19                {
20                    sheep++;
21                }
22            else if
23                (tile.GetComponentInChildren<Renderer>().sharedMaterial.name
24                 == "rock")
25                {
26                    stone++;
27                }
28            else if
29                (tile.GetComponentInChildren<Renderer>().sharedMaterial.name
30                 == "brick")
31                {
32                    brick++;
33                }
34            else if
35                (tile.GetComponentInChildren<Renderer>().sharedMaterial.name
36                 == "wood")
37                {
38                    wood++;
39                }
40            }
41        }
42    }
43    UpdateResources();
44 }
```

---

Listing 4.7: Attribute und Initialisierung des Spielers

Neben dem Bau von neuen Siedlungen, muss auch der Nutzen einer solchen ermittelt werden. Jede Siedlung kann 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 die Rohstoffe verteilt. Exemplarisch ist in Listing 4.7 nur *CollectResources(int number)* aufgeführt. Die andere ist äquivalent und daher nur im Anhang zu finden.

Zu jeder Siedlung werden die angrenzenden Felder gespeichert. Diese können entweder *null* oder von einem bestimmten Typus sein, der über dessen *Material* abgefragt wird. Auf diese Weise kann 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.

---

```
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 4.8: Attribute und Initialisierung des Spielers

Nachdem oben erläutert wurde, wie die Ermittlung der nötigen Menge an Rohstoffen zum Bau einer Siedlung oder Straße erfolgt und der Ertrag einer Siedlung berechnet wird, stellt Listing 4.8 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 diese dort platzieren. Zu Beginn der Methode wird überprüft, ob

der Spieler kostenlos bauen darf bzw. ob er genug Rohstoffe zum Bauen hat. Falls er genug Rohstoffe hat, aber nicht kostenlos bauen darf, bezahlt der Spieler entsprechend mit den Rohstoffen. Wenn allerdings kostenlos gebaut werden darf, wird die Möglichkeit des kostenlosen Bauens genutzt. Anschließend würde dann der kostenlose Bau einer Straße erlaubt werden.

In jedem Fall wird im Anschluss an diese Prüfung eine neue Siedlung auf Grundlage des *GameObjects village* instanziert. Bei dem instanziierten Objekt handelt es sich um eine weitere Instanz, des bereits als Repräsentation instanziierten und dafür vorgesehenen *Prefabs*. Dieser Instanz wird die zuvor übergebene Position zugeschrieben und sie erhält die Farbe des Spielers als Oberflächenfarbe. Nachfolgend werden die Siegpunkte des Spielers um eins erhöht sowie seine Menge an Rohstoffen aktualisiert und schließlich wird die Instanz der Siedlung zurückgegeben.

Die Methode zum Bau einer Straße funktioniert äquivalent zum Bau von Siedlungen, wobei insofern ein Unterschied besteht, als dass der Spieler sämtliche kostenlosen Baurechte verliert und eine Rotation an der Straße vorgenommen werden muss. Die Methode erhält 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 zweier Sechsecke stehen und diese in eine bestimmte Richtung zeigen. Damit dies auf dem Spielfeld ansehnlicher aussieht, werden die instanziierten Straßen in dieselbe Richtung gedreht.

In den Spielregeln (s.Anhang) ist angegeben, dass die kostenlosen Straßen im ersten bzw. zweiten Zug eines Spielers nur an die Siedlung gebaut werden dürfen, die im jeweiligen Zug errichtet wurden. Im Zuge der Implementation wird auf diese Regel verzichtet. Der Bau der zweiten Straße kann dadurch auch an die bestehende Straße erfolgen. Dies kann als weiteres strategischen Element aufgefasst werden.

---

```
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];
```

```

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

Listing 4.9: Attribute und Initialisierung des Spielers

Listing 4.9 führt die Berechnung der längsten Straße eines Spielers ein. Hierfür sind zwei Methoden nötig. Die erste Methode *CalculateRoadLength* prüft zunächst, ob der Spieler überhaupt Straßen besitzt. 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 ohne Inhalt initialisiert und anschließend wird die Menge an Nachbarn für diese Straße ermittelt. Die Anzahl der Nachbarn stellt die Länge der Straße dar und durch das Ausfindigmachen des Maximums der Liste *roadLengths* kann 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)* genutzt. Diese ermittelt, wie viele Straßen von der betrachteten Straße abgehen. Am Anfang der Methode gibt es 0 Nachbarn und eine leere Liste. 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. Damit für eine alleinstehende Straße eine Länge von eins errechnet wird, wird sie als Nachbar von sich selbst angesehen. Wird ein Nachbar gefunden, wird dieser aus *roadsModified* entfernt und zu *roadsupdated* hinzugefügt. Außerdem wird die Anzahl der Nachbarn um eins erhöht. Wenn die betrachtete Straße allerdings kein direkter Nachbar der übergebenen Straße ist, so wird *i* um eins erhöht, um zur nächsten Straße überzugehen.

Anschließend wird für jede Straße in *roadsUpdated* 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 berechnet werden. Ist die Rekursion abgeschlossen, wird die Anzahl der Nachbarn zurückgegeben.

Da *CalculateRoadLength()* die Anzahl der Nachbarn für jede Straße kalkuliert, ist insofern sichergestellt, dass tatsächlich die längste Straße gefunden wird, als dass jede Straße einmal als Startpunkt fungiert.

#### 4.2.3 Der Gamemanager

Der Gamemanager dient der Steuerung des Ablaufs des Spiels. Hierüber wird später der Zugriff auf die Schnittstelle gewährleistet, sodass KI-Spieler und Spiel miteinander interagieren können. Dieser Manager sorgt für einen reibungslosen Ablauf des Spiels, weißt zu, welcher Spieler am Zug ist und überprüft alle Geschehnisse, die während des Spiels vor sich gehen. Ohne ihn wäre kein Zusammenspiel der Spieler bzw. überhaupt kein Spiel 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 =
32         villagePlaces.GetComponentsInChildren<buildVillage>();
33     foreach (buildVillage build in buildVillages)
34     {
35         build.player = activePlayer;
36     }
37
38     ... //Deaktivierung der Bauplätze
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 4.10: Initialisierung und Spielerwechsel

In Listing 4.10 werden zwei Methoden gezeigt. Vor der Methode *ChangePlayer* wären normalerweise die Awake- und die Startmethode sowie die Liste der deklarierten Attribute zu finden. Aus Platzgründen wurde hier auf die Einbindung dieser Abschnitte verzichtet, diese können jedoch im Anhang eingesehen werden. Erklärungen zu betroffenen Variablen sind weiterhin in der Beschreibung ihrer Verwendung zu finden. Zusätzlich fehlen einige Teile der *UpdateActivePlayer(PlayerScript player)*-Methode, da diese sonst zu lang gewesen wäre. Entfernte Stellen sind jeweils durch drei Punkte gekennzeichnet.

Neben *UpdateActivePlayer* existiert auch *ChangePlayer()*. Diese rotiert zunächst die Kamera um 180 Grad in der Vertikalen, um den Spielerwechsel für einen menschlichen Betrachter zu signalisieren. Anschließend wird der nächste Spieler aktiv, wodurch er seinen Zug beginnt.

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 aktualisiert damit die globale Variable *activePlayer*. Anschließend werden die Siegpunkte des Spielers aktualisiert - zunächst durch den Aufruf der passenden Methode des aktuellen Spielers, 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 sowie die Vernachlässigung der längsten Straße sind innerhalb dieser Methode nötig, weil ansonsten der Spieler, der zuvor die längsten Straße besessen hat, seine Punkte nicht wieder verlieren würde.

Im nächsten Schritt muss ein potenzieller Siedlungsbauplatz erfahren, welchem Spieler die Errichtung einer Siedlung zugeordnet werden würde. Hierzu müssen zunächst die deaktivierten Bauplätze aktiviert werden, anschließend wird dann dem verantwortlichen Skript jedes Bauplatzes der aktuelle Spieler als Attribut mitgeteilt. Um dies zu erreichen, wird zunächst mithilfe einer Schleife jeder potenzielle Bauplatz durchlaufen und aktiviert. Anschließend wird das Elternelement ebenfalls aktiviert. Nun kann über das Skript jedes Bauplatzes iteriert und der aktuelle Spieler gesetzt werden. Anschließend erfolgt die Deaktivierung aller Bauplätze sowie des Elternelementes. Dies ist nötig, da ohne den Aktivierungsschritt kein Zugriff auf die Skripte erfolgen könnte. Dieser Prozess ist für den Nutzer allerdings unsichtbar.

Das gleiche Vorgehen wird auch auf die potenziellen Bauplätze von 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 ||
4         activePlayer.isSecondTurn))
5     {
6         villagePlaces.SetActive(true);
7         Transform transform = villagePlaces.GetComponent<Transform>();
8         for (int i = villagePlaces.GetComponent<Transform>().childCount - 1;
9              i >= 0; i--)
10        {
11            Transform child = transform.GetChild(i);
12            child.gameObject.SetActive(true);
13        }
14    }
15}
```

```
12     if (buildVillage != null)
13     {
14         DestroyVillagePlacesInRadius();
15     }
16
17 }
18 ...//Äquivalent mit HasResourcesForVillage statt isFirstTurn und is
19     SecondTurn
20 else
21 {
22     Transform transform = villagePlaces.GetComponent<Transform>();
23     for (int i = villagePlaces.GetComponent<Transform>().childCount - 1;
24         i >= 0; i--)
25     {
26         Transform child = transform.GetChild(i);
27         child.gameObject.SetActive(false);
28     }
29     villagePlaces.SetActive(false);
30 }
```

---

Listing 4.11: Update-Methode

Listing 4.11 zeigt die Methode *Update()*. Sie wird kontinuierlich aufgerufen und daher dauerhaft ausgeführt. Sie beinhaltet 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, der am Zug ist, auf dem Spielfeld die Repräsentation der Siedlung ausgewählt haben. In solch einem Fall wird *villageFocus.hasFocus* wäre auf *true* gesetzt und die Überprüfung der weiteren Bedingungen findet statt. Sofern es sich um den ersten oder zweiten Zug des Spielers handelt, werden alle potenziellen Bauplätze angezeigt und es kann auf einem dieser Plätze gebaut werden. Jedoch schränkt der Bau einer Siedlung die Plätze für potenziell folgende Siedlungen ein, da sich zwischen zwei Siedlungen mindestens Platz für zwei Straßen befinden muss. Um dies zu gewährleisten, werden die Bauplätze in einem zuvor festgelegtem Umkreis durch die Methode *DestroyVillagePlacesInRadius()* gelöscht.

Sollte es sich bei dem aktuellen Zug um einen Standardzug handeln, so wird in der Bedingung einer *else if-Abzweigung* geprüft, ob der Spieler genügend Ressourcen hat. Falls dem so ist, darf er eine Siedlung bauen - allerdings nur dort, wo ein Siedlungsbauplatz an einer bestehenden Straße liegt. Dadurch und durch die Löschung der zu nah an anderen Siedlungen gelegenen Bauplätze wird die Einhaltung der Zwei-Straßen-Abstand-Regel umgesetzt. Konkret funktioniert die Prüfung, ob eine Straße anliegt, indem alle gebauten Straßen des Spielers durchlaufen werden und somit überprüft wird, ob ein Bauplatz an diesen anliegt. Der anliegende Bauplatz wird daraufhin aktiviert und für den Spieler sichtbar.

Falls sich keine der beiden obigen Bedingungen als wahr herausstellt, so werden 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 ein erneuter Klick auf die Repräsentation vorgenommen wird, der Spieler nicht mehr kostenlos bauen darf oder seine Ressourcen nicht mehr ausreichen, um eine weiter Siedlung zu errichten.

---

```
1 if (roadFocus.hasFocus && (activePlayer.HasResourcesForRoad() ||
2     activePlayer.isFirstTurn || activePlayer.isSecondTurn))
3 {
```

```

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;
8             i >= 0; i--)
9         {
10            Transform child = transform.GetChild(i);
11            float distance = (village.transform.position -
12                child.position).magnitude;
13            if (distance < roadRange)
14            {
15                child.gameObject.SetActive(true);
16            }
17        }
18    }
19    foreach (GameObject road in activePlayer.roads)
20    {
21        Transform transform = roadPlaces.GetComponent<Transform>();
22        for (int i = roadPlaces.GetComponent<Transform>().childCount - 1;
23             i >= 0; i--)
24        {
25            Transform child = transform.GetChild(i);
26            float distance = (road.transform.position -
27                child.position).magnitude;
28            if (distance < roadRange)
29            {
30                child.gameObject.SetActive(true);
31            }
32        }
33        if (buildRoad != null)
34        {
35            if (!activePlayer.freeBuild && !activePlayer.freeBuildRoad &&
36                (activePlayer.isFirstTurn || activePlayer.isSecondTurn))
37                endTurnBtn.interactable = true;
38        } else
39        {
40            ... //Deaktivieren wie für die Siedlungen
41        }
42    }

```

Listing 4.12: Update-Methode Aktivierung der Bauplätze für Straßen

Das gleiche Prinzip wird mit wichtigen Anpassungen auch auf den Bau von Straßen angewandt (siehe Listing 4.12). Bei den Siedlungen gilt, dass im ersten und zweiten Zug jeder ungelöschte Bauplatz für den Bau einer Siedlung infrage kommt. Bei den Straßen ist dies anders: Hier darf der Bau einer Straße nur dann erfolgen, wenn direkt an am Bauplatz entweder bereits eine Siedlung oder eine weitere Straße gebaut wurde. Die Überprüfung erfolgt erneut durch die Iteration über alle Siedlungen in der äußeren Schleife sowie über alle Bauplätze für Straßen in der inneren Schleife. Es werden nur die Bauplätze aktiviert, die sich direkt an einer bereits existierenden Siedlung befinden. Anschließend wird das gleiche Prozedere für die bereits existierenden Straßen vorgenommen und auch hier werden die anliegenden Bauplätze freigeschaltet. Anders als bei der Aktivierung der Siedlungsbau-

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

---

plätze, kann die für die Straßen ohne weitere Verzweigung in else if-Form ablaufen, da der Ablauf der Aktivierung in beiden Fällen der gleiche ist. Die Deaktivierung der Bauplätze wiederum erfolgt äquivalent.

```
1  public void OnRollDice()
2  {
3      // left mouse button clicked so roll random colored dice 2 of each
4      // dieType
5      Dice.Clear();
6      Dice.Roll("1d6", "d6-" + "red", spawnPoint.transform.position,
7          Force());
8      Dice.Roll("1d6", "d6-" + "red", spawnPoint.transform.position,
9          Force());
10     rollDiceBtn.interactable = false;
11 }
12
13 private Vector3 Force()
14 {
15     Vector3 rollTarget = Vector3.zero + new Vector3(2 + 7 * Random.value,
16         .5F + 4 * Random.value, -2 - 3 * Random.value);
17     return Vector3.Lerp(spawnPoint.transform.position, rollTarget,
18         1).normalized * (-35 - Random.value * 20);
19 }
20
21 IEnumerator AddResources()
22 {
23     yield return new WaitForSeconds(3f);
24     int number = Dice.GetValue("");
25     player1.CollectResources(number);
26     player2.CollectResources(number);
27     activePlayer.UpdateResources();
28 }
29
30 IEnumerator EnableEndTurn()
31 {
32     yield return new WaitForSeconds(4f);
33     endTurnBtn.interactable = true;
34 }
```

---

Listing 4.13: Würfeln

Ein weiterer wichtiger Bestandteil des *Gamemangers* ist das Würfeln. Listing 4.13 zeigt die Implementierung dieser Funktionalität. Es werden hierfür zwei Methoden benötigt: Die erste Methode, *OnRollDice()*, löscht zunächst den vorherigen Würfelversuch, um dann anschließend zwei rote sechsseitige Würfel zu werfen. Daraufhin wird das erneute Würfeln verboten und es werden zwei Coroutinen gestartet. Die eine dieser *Coroutinen* dient dem Hinzufügen von Ressourcen zum Bestand der Spieler, die andere dem Erlauben des Bauens und Beendens des 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.

Die besagten Routinen sind ebenfalls abgebildet. Die erste der beiden *IEnumerator* - AddResources() - dient dem Hinzufügen der Rohstoffe zu den Beständen aller Spieler. Welcher Spieler Rohstoffe erhält, ist von der gewürfelten Zahl sowie davon abhängig, ob ein Spieler eine Siedlung an dem mit der Zahl korrespondierenden Landschaftsfeld hat. Dementsprechend 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 so weiter ab als hätte der Spieler nicht gewürfelt. Um ein solches Fehlverhalten auszugleichen, könnte auf die festgeschriebene Wartezeit verzichtet werden und die Verteilung stattdessen erst dann vorgenommen werden, wenn das Würfeln abgeschlossen ist. Würde das Würfeln nicht in einer bestimmten Zeit abgeschlossen, weil die Würfel beispielsweise festhängen, so könnte der Würfelversuch wiederholt werden. Dieses Verhalten ist wünschenswert, zum aktuellen Stand der Entwicklung aber verzichtbar.

Nachdem die beiden Spieler ihre Rohstoffe erhalten haben, wird die Rohstoffanzeige des aktuellen Spielers aktualisiert, um den Bestand und die Darstellung konsistent zu halten.

In der ebenfalls gezeigten Routine - EnableEndTurn() - wird zu Beginn vier Sekunden gewartet. Die längere Wartezeit ist darauf zurückzuführen, dass die Darstellung des aktualisierten Rohstoffbestands abgeschlossen sein soll. Nachdem gewartet wurde, wird der Button aktiviert, mit dem der Spieler seinen Zug beenden kann. Es soll jedoch zuvor wenigstens kurz angezeigt werden, welche Rohstoffe er aktuell besitzt. Dadurch kann er den neuen Bestand in seine Entscheidung miteinbeziehen und prüfen, ob er seinen Zug überhaupt schon beenden oder ob er zuvor weitere Aktionen vornehmen will. Für einen menschlichen Spieler stellt eine gleichzeitige Aktualisierung der Rohstoffe und Aktivierung des Buttons zur Zugbedeutung kein Problem dar. Bei einem computergesteuerten Gegner könnte dies jedoch zu Fehlern führen. Es müsste intern implementiert werden, dass der Computer abwartet, um festzustellen, welche Rohstoffe er erhält und dann überlegen zu können, ob der Zug tatsächlich beendet werden soll. Durch die Verzögerung kann auf eine solche Implementation allerdings verzichtet werden.

Die Verwendung von Routinen bietet sich für die beiden oben genannten Anwendungsfälle an, da sie für eine parallele Abarbeitung der Geschehnisse sorgen. Dies wirft allerdings die Frage auf, ob nicht auch der Hauptthread pausiert werden könnte. Es gilt jedoch: Das Warten ist nur deshalb sinnvoll, weil im Hintergrund weitere Prozesse ablaufen. Würde der Hauptthread warten, so würde das gesamte Spiel angehalten werden, wodurch das Auswerten der Würfel keinen Effekt mehr hätte, da diese noch gar nicht geworfen worden wären. Gleiches gilt für die Aktivierung des Buttons. Dieser soll erst dann aktiv werden, wenn die Augenzahl bereits errechnet und verarbeitet wurde. Hielte man dafür den Hauptthread an, könnte keine Verarbeitung stattfinden. Durch das Verteilen der Prozesse auf unterschiedliche Threads können die Timer parallel ablaufen und so 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
25         (activePlayer.villages[activePlayer.villages.Count - 1]);
26         UpdateActivePlayer(player1);
27         cameraView.transform.Rotate(0.0f, 180.0f, 0.0f, 0);
28
29         activePlayer.SecondTurn();
30         endTurnBtn.interactable = false;
31         rollDiceBtn.interactable = false;
32     }
33     else if (activePlayer.isSecondTurn && activePlayer == player1)
34     {
35         activePlayer.CollectResourcesForVillage
36         (activePlayer.villages[activePlayer.villages.Count - 1]);
37         activePlayer.Turn();
38         endTurnBtn.interactable = false;
39         rollDiceBtn.interactable = true;
40     }
41     else
42     {
43         ChangePlayer();
44         rollDiceBtn.interactable = true;
45         endTurnBtn.interactable = false;
46     }
47     activePlayer.UpdateResources();
48 }
```

---

Listing 4.14: Zug beenden

Wird der Button zum Beenden des Zuges betätigt, ruft dies die Methode *OnEndTurn()* auf. Abgebildet ist sie als Listing 4.14. Am Anfang der Methode ist eine if-Verzweigung zu sehen. Diese dient der Bestimmung des Siegers. Sollte ein Spieler in seinem aktuellen Zug die nötigen Siegpunkte erreicht bzw. überschritten haben, so ist dieser der Sieger des Spiels. In diesem Fall wird das Spiel beendet und der aktuelle Spieler als Sieger ausgegeben. Erreicht der aktuelle Spieler die nötige Anzahl der Siegpunkt nicht bzw. hat er nicht gewonnen, so wird der Rest der Methode ausgeführt: Im ersten Schritt wird dann 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, sodass die Kamera sich dreht und *Spieler 2* seinen ersten Zug machen darf. Zudem müssen die Buttons zum Würfeln und zum Beenden des Zuges wieder deaktiviert werden. Falls es sich bei dem aktuellen Zug jedoch bereits um den ersten Zug von *Spieler 2* handelt, so wird die Kamera nicht gedreht, da er am Zug bleibt. Er darf direkt seinen zweiten Zug ausführen. Wenn er seinen Zug beendet, erhält *Spieler 2* die Rohstoffe für seine in Zug 2 kostenlos gebaute Siedlung. Anschließend wird zum zweiten Zug von *Spieler 1* gewechselt. Dieser Zug ist der letzte vor den gewöhnlichen Spielzügen. Wird dieser beendet, so

erhält auch *Spieler 1* Rohstoffe für die zuletzt gebaute Siedlung. Er darf anschließend mit dem ersten Standardzug beginnen. Zunächst wird allerdings der Würfelbutton aktiviert, damit der Spieler diesen zu Beginn seines Zuges betätigen kann. Die Standardzüge werden in *OnEndTurn()* durch den *else*-Zweig der bisher beschriebenen Fallabfrage bearbeitet. Sollte 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 4.10) und die darauffolgende Aktivierung des Würfelbuttons sowie die Deaktivierung des Buttons zur Zugbeendung beendet.

Unabhängig davon, welche der obigen Anweisungen aufgrund der geltenden Bedingungen durchgeführt wurden, in jedem Fall wird am Schluss der Methode der Bestand der Rohstoffe des aktiven Spielers aktualisiert.

#### 4.2.4 Die Repräsentation der Bauobjekte

In der Beschreibung des Spielfelds wurde erwähnt, dass die abgebildeten Objekte, die zu bauenden Siedlungen (siehe Abb. 3.1, links oben) und Straßen (3.1, rechts unten) darstellen. In Kapitel 4.2.3 wurde auf die Nutzung dieser Objekte eingegangen und festgehalten, dass sie dem Zweck dienen, 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 4.15: Aktivierung des Baumodus für Siedlungen

Listing 4.15 zeigt, wie dieser Mechanismus für den Baumodus von Siedlungen implementiert ist. Sobald die angesprochene Repräsentation auf dem Spielfeld angeklickt wird, wird der Baumodus aktiviert, sofern er deaktiviert ist bzw. deaktiviert, wenn er aktiviert ist. Beachtet werden sollte in diesem Zusammenhang zudem die Variable *roadFocus*. Diese enthält das äquivalente Skript für die Straßen und gewährleistet, dass nicht beide Modi gleichzeitig zur Verfügung stehen. Das bedeutet, falls der Baumodus für Siedlungen aktiviert wurde, wird der für Straßen entsprechend deaktiviert. Eine äquivalente Klasse zur Aktivierung des Baumodus für Straßen befindet sich im Anhang.

#### 4.2.5 Der Bau von Siedlungen und Straßen

Die Klasse *buildVillage* (siehe Listing 4.16) 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 die Methode *OnMouseDown()*, die als OnClick-Reaktion pro Bauplatz fungiert. Dieser OnClick-Mechanismus funktioniert nur, wenn die Bauplätze aktiviert sind, daher wurde im Abschnitt 4.2.3 ausgiebig beschrieben, wann dies der Fall ist.

Wenn ein aktiver Bauplatz angeklickt wird, wird zunächst mit der dafür vorgesehenen

Methode *BuildVillage(Vector3 position)*, die in der Klasse PlayerScript 4.8 zu finden ist und durch den aktiven Spieler aufgerufen wird, eine Instanz einer Siedlung erzeugt. Die Erzeugung einer neuen Instanz für den aktuellen Spieler ist in Kapitel 4.2.2 näher beschrieben. Nachdem die Instanz zurückgeliefert wurde, wird auf sein Skript *Village* zugegriffen. Dieses ist unter Listing 4.18 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. Mithilfe dieser beider Klassen können dem entstandenen Dorf angrenzenden Ressourcen vermittelt werden. Diese Zuweisung ist in den Zeilen 13-18 zu sehen. Die Prüfung auf *null*-Werte erfolgt zusätzlich, da noch keine Wasserfelder existieren und einige Bauplätze, die nicht an drei Felder grenzen, zum Teil *null*-Werte enthalten können.

Abschließend wird das entstandene Dorf der Liste der Dörfer des Spielers (s. Zeile 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 4.2.3 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 =
9             player.BuildVillage(this.transform.position);
10
11         Village villageScript = villageInstance.GetComponent<Village>();
12         PlaceScript placeScript = this.GetComponent<PlaceScript>();
13
14         if (placeScript.resourceTile1 != null)
15             villageScript.tiles.Add(placeScript.resourceTile1);
16         if (placeScript.resourceTile2 != null)
17             villageScript.tiles.Add(placeScript.resourceTile2);
18         if (placeScript.resourceTile3 != null)
19             villageScript.tiles.Add(placeScript.resourceTile3);
20
21         player.villages.Add(villageInstance);
22         gameManager.buildVillage = villageInstance;
23     }
24
25 }
```

---

Listing 4.16: Bau von Siedlungen

In Listing 4.17 ist der gleiche Ablauf für den Bau von Straßen zu sehen. Hier sind wieder 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* wird sie der Liste der Straßen des Spielers hinzugefügt, woraufhin seine längste Straße erneut ermittelt wird und der Gamemanager Zugriff auf die Instanz erhält. Abschließend wird noch das aufrufende GameObject zerstört, um den Bauplatz unter der Straße zu entfernen. Dieser Schritt entfällt hingegen beim Bau von Siedlungen, da hier stattdessen 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,
9             this.transform.rotation);
10
11         if (roadInstance != null)
12         {
13             player.roads.Add(roadInstance);
14             player.CalculateRoadLength();
15             gameManager.buildRoad = roadInstance;
16
17             Destroy(this.gameObject);
18         }
19     }

```

---

Listing 4.17: 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 4.18: Hilfsklassen Village und PlaceScript

### 4.3 Die Schnittstelle

Die Schnittstelle wurde bereits im Kapitel 3.2 thematisiert, jedoch soll nachfolgend genauer auf die konkrete Implementation dieser eingegangen werden. Zunächst wurde die Entscheidung getroffen, dass die KIs auf Java basieren sollen, da so während der Implementation eine einfache Nutzung eines *Case Based Reasoning-Systems* (CBR-Systems) möglich ist. Grundlegend wurde sich bei der Implementation an der Masterarbeit von Jannis Hillmann [Hil17, vgl.] orientiert. In seiner Arbeit wendet er ein ähnliches Konzept auf einen *Ego-Shooter* an. Im Rahmen dieses Projekts wurde vor allem auf Hillmanns Java-Klasse *CBRSystem.jar* zurückgegriffen, die entsprechend angepasst wurde.

Die Klasse *CBRSystem.jar* aktiviert einen Server, der vom Client (dem Spiel) angesteuert werden kann, um Anfragen zu stellen, die in Form von Anweisungen von der KI beantwortet werden. Die Kommunikation zwischen dem Server und dem Client findet hierbei wie geplant - mittels *JSON* statt. Die Anfragen, die das Spiel an den Server richtet, bestehen immer aus einer ganzen Situation. Das bedeutet, die KIs werden über die Schnittstelle jedes Mal über den aktuellen Spieler sowie über den Zustand des Spiels und des Spielers informiert.

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

---

```
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 4.19: Die relevanten Informationen für eine Situation

Listing 4.19 zeigt die Attribute einer Situation. Auffällig ist, dass der Name des Spielers als *String* gespeichert wird, die Karte und der Status aber jeweils durch eigene Klassen repräsentiert werden. Die Klasse *Status* bedarf an dieser Stelle keiner weiteren Erläuterung, da dort lediglich die Attribute des betroffenen Spielers zur Verarbeitung durch *JSON* gespeichert werden. Bei der Klasse *Map* ist dies ähnlich, allerdings muss hier ein Weg gefunden werden, die sichtbare Karte als abstraktes Objekt zu formulieren. Zu sehen sind die Attribute in Listing 4.20. 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 sind. 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 4.20: Die Attribute der Klasse Map

Die beiden Klassen zur Speicherung von Bauplätzen verfügen jeweils über ein Attribut, das der Prüfung der Aktivierung des Bauplatzes sowie einer potentiellen Veränderung dieses Zustands durch entsprechende Methoden dient. Die Klasse *Tile* speichert Informationen über die Sechsecke des Spielfeldes und benötigt hierfür Informationen über die Nummern der Sechsecke und die Art des Feldes. Um dies zu ermöglichen, wird das *Enum* genutzt.

Durch die angeführten Klassen und Methoden wird zwar das Speichern des sichtbaren Spielfeldes möglich, jedoch muss auch eine Verbindung zwischen den abstrakten Modellen und dem Spiel selbst aufgebaut werden, damit die benötigten Informationen gewonnen werden können. Um dies zu gewährleisten, wird bereits bei der Generierung des Spielfeldes eine erste Verbindung aufgebaut. Listing 4.21 zeigt die Erstellung eines abstrakten Feldes. Dies geschieht für jedes sichtbare Feld, dass kreiert wird. Es speichert Informationen über die Nummer des jeweiligen Feldes und die darin vorhandenen Ressourcen. Nach der Erstellung wird es der entsprechenden Liste in der Karte hinzugefügt.

---

```

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 4.21: Verknüpfung der Erstellung der Karte und den Feldern

Die Verlinkung der Bauplätze wird nach der Generierung des Spielfeldes im *GameManager* aufgebaut. Listing 4.22 zeigt die Methode *LateStart()*. Diese wird einmal zu Beginn des ersten Aufrufs der *Update()*-Methode aufgerufen. Hierdurch wird sichergestellt, dass die Start-Methode aller anderen Skripte bereits ausgeführt wurde und es zu keinen *Exceptions* kommt. Konkret muss auf die Erstellung der Karte im *MapGenerator* und auf die der physischen Bauplätze gewartet werden. Diese Methode übernimmt zunächst die Kontrolle über die Karte des *MapGenerators* und lädt anschließend aus jedem Bauplatz den enthaltenen abstrakten Bauplatz, um diesen in der passenden Liste der Karte zu speichern. Der abstrakte Bauplatz wird zur Laufzeit in jedem physischen Bauplatz in der Startmethode erstellt, sodass eine Zuordnung möglich wird. Nachdem dies geschehen ist, darf der erste Zug des ersten Spielers ausgeführt werden, da nun alle Vorbereitungen abgeschlossen wurden. Dazu wird die erzeugt und der erste Zug für von *Spieler 1* begonnen.

---

```

1 private void LateStart()
2 {
3     //Die Karte kann erst jetzt geholt werden, da sie vorher noch nicht
4     //fertig war.
5     map = mapGenerator.map;
6
7     Transform transform = villagePlaces.GetComponent<Transform>();
8     for (int i = villagePlaces.GetComponent<Transform>().childCount - 1; i >=
9         0; i--)
10    {
11        Transform child = transform.GetChild(i);
12        map.villageBuildPlaces.Add(child.gameObject.GetComponent<buildVillage>().villagePlace);
13    }
14
15    transform = roadPlaces.GetComponent<Transform>();
16    for (int i = roadPlaces.GetComponent<Transform>().childCount - 1; i >=
17        0; i--)
18    {
19        Transform child = transform.GetChild(i);
20        map.roadBuildPlaces.Add(child.gameObject.GetComponent<BuildRoad>().roadPlace);
21    }
22
23    //Nun kann der erste Zug von Spieler 1 ausgeführt werden
24    activePlayer.FirstTurn();
25    StartAIProcess();
26    MakeAiTurn(4f);

```

---

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

---

24 }

---

Listing 4.22: 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\CBRSys tem.jar";
6     //foo.StartInfo.FileName = Environment.CurrentDirectory +
7         @"\Assets\HalloWelt.jar";
8     foo.StartInfo.Arguments = "" + Constants.PORT;
9     foo.Start();
10    connection = new Connection();
11 }
```

---

Listing 4.23: GameManager: StartAIProcess

Die Methode *StartAIProcess()* dient dem Starten des Servers, zu dem sie die Verbindung aufbaut (siehe Listing 4.23). Die Methode *MakeAITurn* (Listing 4.24) wird anschließend aufgerufen, um die erste Anfrage an den Server geschickt werden kann. Die Methode startet fünf Co-Routinen, die jeweils um das angegebene *Delay* eine Anfrage an den Server schicken (siehe *ActivateAi()*, Listing 4.24). Als auf diese Anfrage 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 im gleichen Listing 4.24 abgebildet sind. Der Unterschied besteht darin, dass der zuletzt gezeigte *IEnumerator* zunächst gestartet wird und anschließend rekursiv erneut eine Anfrage verschickt, falls der empfangene Plan keine Anweisung zum Beenden des Zuges enthält.

```
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,
15         rollDiceBtn.interactable);
16     Plan plan = response.plan;
17     plan.StringToActions();
18     UnityEngine.Debug.Log("Plan " + plan.ToString());
19     activePlayer.FulfillPlan(plan);
20 }
21 private void MakeAiMainTurn()
22 {
23     StartCoroutine(ActivateAiMainTurn(4f));
24 }
```

---

```

26 IEnumerator ActivateAiMainTurn(float wait)
27 {
28     yield return new WaitForSeconds(wait);
29     Response response = SendToAI(endTurnBtn.interactable,
30         rollDiceBtn.interactable);
31     Plan plan = response.plan;
32     plan.StringToActions();
33     UnityEngine.Debug.Log("Plan " + plan.ToString());
34
35     bool wantsToEndTurn = false;
36     for (int i = 0; i < plan.actions.Count; i++) {
37         if (plan.actions[i].GetType() == typeof(EndTurn))
38         {
39             wantsToEndTurn = true;
40         }
41         activePlayer.FulfillPlan(plan);
42         yield return new WaitForSeconds(wait);
43         if (!wantsToEndTurn)
44         {
45             StartCoroutine(ActivateAiMainTurn(5f));
46         }
47 }

```

Listing 4.24: Ausführen einer Anfrage an den Server

Die Verarbeitung der Pläne erfolgt im *PlayerScript* eines Spielers selbst. Hierzu muss das Skript so erweitert werden wie in Listing 4.25 dargestellt. 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 nacheinander jede enthaltene Aktion verarbeitet. Ob eine Aktion durchgeführt werden kann, hängt davon ab, ob die nötigen Bedingungen erfüllt sind und ob die jeweilige Aktion existiert. Jede Aktion wird durch eine eigene Klasse repräsentiert, die von der abstrakten Klasse Aktion erbt. Bereits verfügbar sind das Aktivieren und Deaktivieren von Bauplätzen, das Bauen von Straßen und Siedlungen (auf zufälligen Bauplätzen), das Beenden von Zügen sowie das Würfeln. Welche dieser Aktionen ausgeführt wird, hängt jeweils davon ab, welcher Klassename 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
13             //aufgerufen wurde.
14         }
15         else if (plan.actions[i].GetType() == typeof(ActivateRoadPlaces))
16         {
17             ...//Äquivalent zu ActivateVillagePlaces
18         }
19         else if (plan.actions[i].GetType() == typeof(BuildVillage))
20         {
21             gm.Update();
22         }
23     }
24 }

```

```
19     {
20         //Zurzeit noch zufälliger Bauplatz
21         Transform transform = gm.villagePlaces.GetComponent<Transform>();
22         int rand = Random.Range(0,
23             gm.villagePlaces.GetComponent<Transform>().childCount - 1);
24         Transform child = transform.GetChild(rand);
25         child.gameObject.GetComponent<buildVillage>().Instantiate();
26     }
27     else if (plan.actions[i].GetType() ==
28         typeof(Assets.Scripts.CBR.Plan.BuildRoad))
29     {
30         ...
31         ... //Weitesgehend äquivalent zu BuildVillage
32     }
33     else if (plan.actions[i].GetType() == typeof(EndTurn))
34     {
35         if (gm.endTurnBtn.interactable)
36             gm.OnEndTurn();
37     }
38     else if (plan.actions[i].GetType() == typeof(RollDice))
39     {
40         if (gm.rollDiceBtn.interactable)
41             gm.OnRollDice();
42     }
43 }
```

---

Listing 4.25: PlayerScript: FulfillPlan

Der Plan wird auf Seiten des Servers zusammengestellt. Hierfür sind vor allem drei Klassen von Relevanz: *CBRSystem*, *PlayerManager* und *Player*. Die Klasse *CBRSystem* ist für die Verbindung zwischen Client und Server verantwortlich. Sie empfängt Anfragen des Spiels und sendet die Antworten der KI ab. Die Verarbeitung der Anfragen wird allerdings in den anderen beiden genannten Klassen vorgenommen. In der Klasse *PlayerManager* wird mithilfe von Informationen aus der Anfrage über das Ansteuern des richtigen Spielers entschieden. Außerdem sorgt diese Klasse dafür, dass die Informationen, die der KI zur Verfügung stehen, aktuell sind. Die Spieler entwickeln auf Grundlage dieser Informationen einen Plan, der in der Klasse *PlayerManager* zu einer Antwort in Form eines Response-Objektes geformt, an die aufrufende Klasse *CBRSystem* zurückgegeben und von dort wieder an das Spiel geschickt wird.

## 4.4 Erweiterung durch den Aufbau von Städten

Beim Einpflegen von Städten in das Spiel handelt es sich um eine 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. In dieser inbegriffen waren lediglich folgende Bestandteile: Straßen, Siedlungen, Spielzüge, die Funktion des Würfels und die Ermittlung des Gewinners. Durch die Einführung der Städte in das Spiel erhöht sich die Komplexität für die Spieler ? sowohl für computergesteuerte als auch für menschliche. Entsprechend erhöht sich der strategische Anspruch an die Spieler, denn für den Fall, dass sowohl Siedlungen als auch Straßen und Städte gebaut werden können, aber nicht alle, muss eine Auswahl getroffen werden.

Die Implementierung der Städte ist sehr ähnlich zu der der Siedlungen. Einige Aspekte für Siedlungen wurden analog zu denen von Städten umgesetzt, weshalb eine erneute Auflistung des Quellcodes an dieser Stelle keinen Mehrwert bietet.

Städte können auf Siedlungen errichtet werden: Sie stellen gewissermaßen ein Upgrade für diese dar. Für Straßen und Siedlungen geeignete Bauplätze wurden manuell als *Game-Objects* auf der Spielfläche platziert. Diese konnten nach Aktivierung des entsprechenden Baumenüs durch den Spieler ausgewählt werden. Wurde ein Bauplatz angeklickt, so entstand dort eine Straße oder eine Siedlung. Für die KI wurde dies durch

entsprechende Anfragen über die Schnittstelle realisiert. Auf gleiche Weise soll dies für die Städte bewerkstelligt werden. Der große Unterschied hinsichtlich der Bauplätze besteht in dem Zeitpunkt ihrer Erstellung: Die Bauplätze von Straßen und Siedlungen sind bereits vor Beginn einer Partie im Spiel vorhanden und lediglich deaktiviert, bis sie zum Einsatz kommen. Dies ist für Städte nicht möglich. Ihre Bauplätze entstehen erst dann, wenn eine Siedlung gebaut wird, da sie an den gleichen Stellen wie die Siedlung errichtet werden. 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 ein Bauplatz für eine Siedlung, nur dass er diesmal für Städte gilt. Listing 4.26 zeigt, wie das Erzeugen eines Bauplatzes für Städte in den *Gamemanager* integriert wird. In Listing 4.11 ist in den Zeilen 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
2 //nahen Bauplaetze werden zerstoert
3 if (buildVillage != null)
4 {
5     GameObject cityPlaceInstance = Instantiate(cityPlace);
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
13        List<GameObject>(buildVillage.GetComponent<Village>().tiles);
14
15    cityPlaceInstances.Add(cityPlaceInstance);
16    map.cityBuildPlaces.Add(cityPlaceInstance.GetComponent<BuildCity>().cityPlace);
17
18    DestroyVillagePlacesInRadius();
19    buildVillage = null;
}

```

---

Listing 4.26: GameManager: Erstellung des Bauplatzes für Städte

Im nächsten Schritt gilt es, den Zugriff auf den erzeugten Bauplatz abzuspeichern, damit dieser auch ansteuerbar bleibt. Dies geschieht direkt in Form einer Liste im *GameManager*. Außerdem wird eine Liste der Bauplätze für Städte in der Klasse *Map* aktualisiert, sodass auch dort Zugriff besteht. Dieser Zugriff gilt im *GameManager* für das *GameObject*, während die *Map* nur auf die abstrakte Repräsentation des Bauplatzes zugreifen kann. Zu

sehen ist dieser Vorgang in Listing 4.26. Der Zugriff erfolgt äquivalent zu dem auf Straßen oder Siedlungen.

Neben dem Zugriff müssen auch die Aktivierung und der Bau von Städten ermöglicht werden. Hierzu hat der Städtebauplatz ein passendes OnClick-Event, welches an dem OnClick-Event für Siedlungen orientiert ist. Es lässt sich ebenfalls durch eine geeignete Anweisung der KI aufrufen, welche der Schnittstelle als Option 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 diese über, sodass nach dem Würfeln anhand dieser Informationen die Ressourcenverteilung für die Städte bestimmt werden kann. Es bleibt 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 wird beim Bau einer Stadt aus dem Spiel entfernt, was auch für den jeweiligen Bauplatz gilt. Dies geschieht in der *CityBuild*-Klasse, die äquivalent zur Version dieser Klasse für den Bau von Siedlungen aufgebaut ist, aber zudem die Löschung des Bauplatzes aus dem Spiel und der Löschung aus der Listen umsetzt (siehe Listing 4.27).

---

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

---

Listing 4.27: Erzeugen einer Stadt im OnClick-Event des Bauplatzes

Abgesehen von einem höheren Ressourcenertrag verfügen Städte über die gleichen Eigenschaften wie Siedlungen. Zwischen ihnen müssen sich jeweils mindestens zwei Straßen befinden, was durch den Aufbau auf ehemaligen Siedlungen garantiert wird. Außerdem können um sie herum Straßen errichtet werden. Es gibt für Städte allerdings kein weiteres Upgrade, sodass eine einmal errichtete Stadt also an dieser Position bestehen bleibt, bis das Spiel beendet ist. Des Weiteren erhalten die Spieler je zwei Siegpunkte für jede ihrer Städte.

Neben dem Spiel selbst musste auch die Schnittstelle an die Implementation der Städte angepasst werden. Wichtig war dabei, eine zusätzliche Aktion einzuführen, die das Aktivieren des Baumodus für Städte und deren Bau ermöglicht. Hierzu wurden für die Städte die gleichen Mechanismen wie für Straßen und Siedlungen implementiert. Entsprechend kann die KI eine Aktion auswählen, die die Bauplätze sichtbar macht und so einen Bauplatz

als Ziel auswählen. Auf diesem ausgewählten Platz wird anschließend eine Stadt errichtet. Gleichermaßen wurde auf Seiten des Spiels aufgesetzt, damit entsprechende Anfragen auch von diesem verstanden werden. Dazu wurde die Klasse *Plan* erweitert (siehe 4.28), sodass sie jetzt die genannten Aktionen erkennen und der entsprechenden Liste hinzufügen kann. Im *PlayerScript* erfolgt die Verarbeitung dieser Liste. Hier wird, sofern die Voraussetzungen dafür erfüllt sind, entweder der Baumodus aktiviert oder eine Stadt errichtet (siehe 4.30).

---

```

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]),
10                      int.Parse(splits[2])));
11 }
```

---

Listing 4.28: 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
5         wurde.
6
7 else if (plan.actions[i].GetType() ==
8     typeof(Assets.Scripts.CBR.Plan.BuildCity))
9 {
10    Assets.Scripts.CBR.Plan.BuildCity buildcity =
11        (Assets.Scripts.CBR.Plan.BuildCity) plan.actions[i];
12    if (gm.map.getCityPlaceByPosition(buildcity.row,
13        buildcity.column).gameObject.activeSelf)
14    {
15        gm.map.getCityPlaceByPosition(buildcity.row,
16            buildcity.column).gameObject.GetComponent<BuildCity>().Instantiate();
17    }
18 }
```

---

Listing 4.29: PlayerScript: Umsetzung der Aktionen durch die KI für Städte

Durch die erhöhte Komplexität sind beim Testen des Spielentwurfes neue Fehler aufgetreten: Beispielsweise kam es zu dem Problem, dass die verwendete KI für einige Spielsituationen keine Lösung wusste. Trat so ein Fall auf, wurde eine leere Antwort verschickt. Diese konnte das Spiel jedoch nicht verarbeiten, sodass eine *Exception* geworfen wurde. Diese wird nun abgefangen. Die KI erhält in einem solchen Fall drei Versuche, eine sinnvolle Antwort zu liefern. Ist sie dazu nicht in der Lage bzw. tut sie dies nicht, wechselt der Spieler und der Zug des aktuellen Spielers gilt als beendet. Zusätzlich dazu erfolgt das Abfangen der geworfenen *Exception* über einen *try-catch*-Block im *JsonParser*, der die Bearbeitung einer leeren Antwort verhindert. Der *JsonParser* liefert lediglich ein leeres Objekt T zurück, sollte der geschilderte Fall eintreten. Ein weiteres Problem stellte die Auswahl des Bauplatzes dar: Die KI war nicht wirklich in der Lage dazu, auszuwählen, wo eine Stadt,

eine Siedlung oder eine Straße errichtet werden sollte. Diese Schwierigkeit konnte dadurch gelöst werden, dass der Situationsbeschreibung Informationen zu den Positionen alle Bauplätze hinzugefügt wurden, sodass die KI basierend auf diesen Informationen eine Auswahl treffen kann. In dem String, der die durchzuführenden Aktionen enthält, sind Zeile und Spalte der Bauplätze durch einen Doppelpunkt getrennt angegeben, und werden für alle drei Arten von Gebäuden auf gleiche Weise verarbeitet. Für Städte ist dies in Listing 4.28 zu sehen; für Straßen und Siedlungen erfolgt die Verarbeitung analog hierzu.

---

```
1 else if (plan.actions[i].GetType() == typeof(ActivateCityPlaces))
2 {
3     gm.cityFocus.OnMouseDown();
4     gm.Update(); //sicher gehen, dass Update mindestens einmal aufgerufen
5     wurde.
6
7 else if (plan.actions[i].GetType() ==
8     typeof(Assets.Scripts.CBR.Plan.BuildCity))
9 {
10    Assets.Scripts.CBR.Plan.buildcity =
11        (Assets.Scripts.CBR.Plan.BuildCity) plan.actions[i];
12    if (gm.map.getCityPlaceByPosition(buildcity.row,
13        buildcity.column).gameObject.activeSelf)
14    {
15        gm.map.getCityPlaceByPosition(buildcity.row,
16            buildcity.column).gameObject.GetComponent<BuildCity>().Instantiate();
17    }
18 }
```

---

Listing 4.30: PlayerScript: Umsetzung der Aktionen durch die KI für Städte

## 4.5 Initiale KIs

Die initialen KIs wurden auf Seite der Schnittstelle implementiert und dementsprechend in Java umgesetzt. Das hat den Hintergrund, dass die KIs im späteren Verlauf fallbasiert vorgehen sollen und auf Basis bereits bekannter Fälle Lösungen für Situationen finden sollen.

Zunächst wurde der Aspekt der Fallbasiertheit jedoch vernachlässigt und seitens der Computerspieler mit festgelegten Reaktionen gearbeitet. Auf diese Weise wurden die Funktionalitäten geprüft und etwaige Fehler behoben. Dies erwies sich besonders während des Einfügens neuer Funktionalitäten in das Spiel als hilfreich, da so eine Testumgebung zur Verfügung stand. Zudem musste so nicht mehr jeder Test manuell durchgeführt werden, was Zeit einsparte.

Die festgelegten Züge wurden in “den ersten Zug”, “den zweiten Zug” und “die übrigen Züge” unterteilt, wobei der erste und der zweite Zug sich stark ähnelten und sich entsprechend nur die übrigen Züge unterschieden. Wie ein normaler Zug durchgeführt wurde, ist in Listing 4.31 zu sehen. Zu erkennen ist, dass jede auszuführende Aktion durch eine passend benannte Methode repräsentiert wird. Jede dieser Methoden fügt dem Plan, der zurück ans Spiel geschickt wird, eine geeignete Aktion in Form eines Strings hinzu. Dieser String kann anschließend auf Seiten des Spiels wie in Abschnitt 4.3 und 4.4 interpretiert werden.

---

```

1 if (allowedToRollDice) {
2     RollDice();
3     first = true;
4 } else if (HasResourcesForCity() || HasResourcesForVillage() ||
5             HasResourcesForRoad()) {
6     if (HasResourcesForCity() && first && map.cityPlacesExist()) {
7         triesCity++;
8         first = false;
9         ActivateCityPlaces();
10    } else if (HasResourcesForCity() && !first && map.cityPlacesExist() &&
11                triesCity <= 3) {
12        triesRoad = 0;
13        triesCity = 0;
14        triesVillage = 0;
15        first = true;
16        BuildCity(map.getRandomCityPlace());
17        ActivateCityPlaces();
18    } else if (HasResourcesForVillage() && (first || triesVillage == 0)) {
19        triesVillage++;
20        first = false;
21        ActivateVillagePlaces();
22    } else if (HasResourcesForVillage() && !first && map.villagePlacesActive()
23               && triesVillage <= 3) {
24        triesRoad = 0;
25        triesCity = 0;
26        triesVillage = 0;
27        first = true;
28        BuildVillage(map.getRandomVillagePlace());
29        ActivateVillagePlaces();
30    } else if (HasResourcesForRoad() && (first || triesRoad == 0)) {
31        triesRoad++;
32        first = false;
33        ActivateRoadPlaces();
34    } else if (HasResourcesForRoad() && !first && triesRoad <= 3) {
35        triesRoad = 0;
36        triesCity = 0;
37        triesVillage = 0;
38        first = true;
39        BuildRoad(map.getRandomRoadPlace());
40        ActivateRoadPlaces();
41    }
42 } else {
43     EndTurn();
44 }
```

---

Listing 4.31: Java-Klasse Player: Erstellen eines normalen Zuges zum Testen

Neben dem Hinzufügen der Aktionen zum Plan verdeutlicht Listing 4.31 ebenfalls, wie verschiedene Aktionen zum Testen priorisiert werden: Ist der Bau einer Stadt möglich, so wird sie immer gebaut. Kann keine Stadt, aber eine Siedlung gebaut werden, so wählt die KI diese Option. Falls auch dies nicht möglich ist, weil zu wenige Ressourcen gesammelt wurden oder kein passender Bauplatz verfügbar ist, so versucht die KI, eine Straße zu bauen. Ist auch der Bau einer Straße ist nicht möglich, wird der Zug beendet.

Es kann jedoch auch vorkommen, dass die KI keine Antwort auf eine vorliegende Situation findet. Dann schickt sie einen leeren Plan an das Spiel zurück, worauf so reagiert wird, dass nach drei ungültigen Antworten eines Spielers automatisch der nächste am Zug ist.

Soviel zu den bisher festgelegten Reaktionen der KIs. Darüber hinaus soll auch die Fallbasiertheit in die Entwicklung einbezogen werden, 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 zu erzeugen. Stattdessen ist nun das Vorhandensein einer Wissensbasis nötig, auf die zurückgegriffen werden kann, um mit ihrer Hilfe Pläne generieren zu können.

---

```
1 //Konkreter Fall fÃ¼r den ersten Zug, wenn ein kostenloses Dorf gebaut
2 //werden darf
3 Status firstTurnVillage = new Status();
4
5 //Wichtig
6 firstTurnVillage.villagePlacesActive = true;
7 firstTurnVillage.isFirstTurn = true;
8 firstTurnVillage.isSecondTurn = false;
9 firstTurnVillage.freeBuild = true;
10 firstTurnVillage.freeBuildRoad = false;
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.isAbleToEndTurn = false;
27 firstTurnVillage.allowedToRollDice = false;
28
29 plan = "BuildVillage;ActivateVillagePlaces";
30 status.put(firstTurnVillage, plan);
```

---

Listing 4.32: Java-Klasse DefaultCases: Beispiel

Für die Erstellung der Wissensbasis dient die Klasse *DefaultCases*, die in Java geschrieben wurde und grundlegende Fälle enthält, um der KI Intelligenz zu verleihen. Hierzu wird auf den Status eines Spielers zugegriffen, der alle wichtigen Attribute enthält. Es werden Fälle für jede grundlegende Situation erstellt. Ein Beispielfall ist hierfür in Listing 4.32 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, da 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 dann auszuführen, wenn das Spiel dies auch zulässt. Durch die Standardfälle wird die bereits festgelegte Folge an Reaktionen aus Listing 4.31 simuliert. Der Unterschied zu festgelegten Reaktionen besteht allerdings darin, dass nun ganz einfach neue Fälle hinzukommen können, die die Wissensbasis erweitern und so die Performanz der KI erhöhen.

Zum aktuellen Stand der Implementation würde das Hinzufügen speziellerer Fälle allerdings keinen Effekt erzielen, da zuvor auf die Wissensbasis zugegriffen werden muss, wo zu die tatsächlichen Situationen mit den Fällen abgeglichen werden müssen, sodass anschließend anhand der wichtigen Attributen der Plan ausgewählt werden kann. Bei der Implementation dieses Prozesses, die nachfolgend erläutert wird, wurde sich eng an der Vorgehensweise von Hillmann [Hil17, vgl.] orientiert.

In der Klasse *CBREngine* läuft das *Retrieval*. Es werden alle Attribute des Status der aktuellen Situation genutzt, um diesen als Input in die Fallbasis geben zu können und so einen möglichst ähnlich Fall zu finden. Ein gefundener Fall enthält immer einen Plan, der auf Validität überprüft und ggf. an das Spiel geschickt wird.

In diesem Schritt kann auch die Intelligenz einer KI beeinflusst werden. Zum Bau von Siedlungen kann eine präferierte Ressource angegeben werden. Dies wird in das *Retrieval* einbezogen. Als Ergebnis kann ein Plan entstehen, der die Suche nach einem Bauplatz mit Zugang zu einer bestimmten Ressource anweist.

Eine mögliche Strategie wäre es, besetzte Ressourcenfelder möglichst auszureißen und dadurch mit wenig Platz auf der Karte viele Siegpunkte zu erwirtschaften. Hierzu sollten möglichst wenig 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 erscheint es sinnvoll, im späteren Spielverlauf aufgrund der hohen Kosten für den Bau von Städten vermehrt Wert auf Getreide und Stein gelegt werden.

Eine konträre Strategie für *Spieler 2* zu finden, sollte für einen besonders interessanten Vergleich zwischen beiden Spielern sorgen. Daher soll *Spieler 2* sich darauf konzentrieren, möglichst viel Kontrolle über das Spielfeld zu erhalten. Hierzu muss *Spieler 2* viele Straßen und Siedlungen bauen, damit sie einen großen Anteil der potenziell verfügbaren Bauplätze einnehmen kann. Der Bau von Städten ist bei dieser Strategie hingegen eher zu vernachlässigen; es sei denn, die KI kann dadurch ihr Ressourceneinkommen signifikant steigern. Außerdem sollte *Spieler 2* sich vor allem auf den Gewinn der Ressourcen Lehm und Holz konzentrieren, da beide Rohstoffe sowohl für Straßen als auch für Siedlungen benötigt werden. Allerdings muss besonders zu Beginn des Spiels ebenfalls auf den Zugang zu Getreide und Schafen geachtet werden, da eine Expansion auf der Spielfläche, die den Gegner einschränkt, ohne diese gar nicht möglich wäre.

Nachdem *Spieler 2* dementsprechend hinsichtlich der ersten beiden Züge die gleichen Präferenzen hat wie *Spieler 1*, unterscheidet sich das Vorgehen der beiden Spieler hinsichtlich der gewöhnlichen Zügen. Wenn nicht gewürfelt werden darf, kommt die jeweilige Strategie des Spielers zum Tragen. Beim Bau von Siedlungen werden nicht allein Stein und Getreide priorisiert, sondern zudem in ausgeglichener Form Holz, Lehm, Schafe und Getreide.

Eine interessante Erweiterung für eine weitere Strategie wäre die Kombination beider Herangehensweisen: Statt einer einfachen Priorisierung könnte die Berücksichtigung der Überlegung einbezogen werden, welcher Schritt für die kommenden Runden den größten Ressourcenertrag verspricht. Die auf Städte konzentrierte KI würde Steine und Weizen

priorisieren, die auf Expansion gerichtete KI eher Lehm und Holz. Eine KI, die keine klare Präferenz an Gebäuden hat, könnte möglicherweise mehr Siegpunkte erlangen, da ihre Wahl an die Situation angepasst würde.

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

Zur Umsetzung zweier verschiedener Strategien muss eine getrennte Berechnung der Präferenzen erfolgen. Diese wird direkt in der Java-Klasse Status vorgenommen. Diese wurde um zwei Methoden mit den Namen *CalculatePreferencePlayer1* und *CalculatePreferencePlayer2* erweitert. In Listing 4.33 ist allerdings nur die zweite dieser beiden Methoden abgebildet, die zweite Methode ist im Anhang zu finden. Allerdings reicht zum Verständnis die Betrachtung einer der beiden Methoden, da beide ähnlich vorgehen: Auf Basis der vorhandenen Mengen an Ressourcen wird eine Präferenz berechnet. Dabei wird zunächst zwischen dem ersten Zug, dem zweiten Zug und den übrigen Zügen unterschieden, da für beide Spieler gilt: Sie bauen im ersten Zug die kostenlose Siedlung an einem Ressourcenfeld, welches Holz liefert, während sie im zweiten Zug an einem Feld bauen, welches Lehm liefert. Dies ist deshalb entscheidend, weil zum aktuell implementierten Stand des Spiels kein späteres Tauschen möglich ist. Entsprechend sollten die KIs versuchen, schnellstmöglich Zugang zu den für sie wichtigen Ressourcen zu erhalten. Da Holz und Lehm sowohl für Siedlungen als auch für Straßen benötigt werden, fokussieren sich die Spieler zunächst auf diese.

Anschließend an die ersten beiden Züge wird jedoch langsam die Strategie des jeweiligen Spielers deutlich. Die gezeigte Methode setzt immer dann, wenn der Bau einer Siedlung möglich ist, diejenige Ressource als Präferenz, die am wenigsten zur Verfügung steht. Sollten allerdings zwei Ressourcen gleich häufig verfügbar sein, so wird die wichtigere für den Bau von Straßen und Siedlungen (meist Holz oder Lehm) gewählt. Wenn die Entscheidung zwischen Getreide und Schafen getroffen werden muss, dann ist das Getreide vorzuziehen. Stein wird nie präferiert, da dieser zur Expansion nicht entscheidend ist und weder zum Bau von Siedlungen noch zum Bau von Straßen gebraucht wird. *Spieler 1* präferiert ? mit Ausnahme der ersten beiden Züge - nie Holz oder Lehm.

Die Berechnung der Präferenzen wurde in der obigen Erklärung erläutert. Eine solche findet auf ähnliche Weise auch für *Spieler 1* statt, der in normalen Zügen allerdings immer Getreide oder Stein präferiert, was ihn sehr anfällig für Fehler macht.

---

```
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 if (bricks < sheep || bricks == sheep) {  
15                    preference = "sheep";  
16                } else {  
17                    preference = "wheat";  
18                }  
19            } else {  
20                preference = "sheep";  
21            }  
22        } else {  
23            preference = "wheat";  
24        }  
25    } else if (isAbledToBuildRoad) {  
26        if (bricks < wood || bricks == wood) {  
27            if (bricks < sheep || bricks == sheep) {  
28                if (bricks < wheat || bricks == wheat) {  
29                    preference = "brick";  
30                } else if (bricks < sheep || bricks == sheep) {  
31                    preference = "sheep";  
32                } else {  
33                    preference = "wheat";  
34                }  
35            } else {  
36                preference = "sheep";  
37            }  
38        } else {  
39            preference = "wheat";  
40        }  
41    } else if (isAbledToBuildSettlement) {  
42        if (bricks < wood || bricks == wood) {  
43            if (bricks < sheep || bricks == sheep) {  
44                if (bricks < wheat || bricks == wheat) {  
45                    preference = "brick";  
46                } else if (bricks < sheep || bricks == sheep) {  
47                    preference = "sheep";  
48                } else {  
49                    preference = "wheat";  
50                }  
51            } else {  
52                preference = "sheep";  
53            }  
54        } else {  
55            preference = "wheat";  
56        }  
57    } else if (isAbledToBuildCity) {  
58        if (bricks < wood || bricks == wood) {  
59            if (bricks < sheep || bricks == sheep) {  
60                if (bricks < wheat || bricks == wheat) {  
61                    preference = "brick";  
62                } else if (bricks < sheep || bricks == sheep) {  
63                    preference = "sheep";  
64                } else {  
65                    preference = "wheat";  
66                }  
67            } else {  
68                preference = "sheep";  
69            }  
70        } else {  
71            preference = "wheat";  
72        }  
73    } else if (isAbledToBuildVillage) {  
74        if (bricks < wood || bricks == wood) {  
75            if (bricks < sheep || bricks == sheep) {  
76                if (bricks < wheat || bricks == wheat) {  
77                    preference = "brick";  
78                } else if (bricks < sheep || bricks == sheep) {  
79                    preference = "sheep";  
80                } else {  
81                    preference = "wheat";  
82                }  
83            } else {  
84                preference = "sheep";  
85            }  
86        } else {  
87            preference = "wheat";  
88        }  
89    } else if (isAbledToBuildRoad) {  
90        if (bricks < wood || bricks == wood) {  
91            if (bricks < sheep || bricks == sheep) {  
92                if (bricks < wheat || bricks == wheat) {  
93                    preference = "brick";  
94                } else if (bricks < sheep || bricks == sheep) {  
95                    preference = "sheep";  
96                } else {  
97                    preference = "wheat";  
98                }  
99            } else {  
100               preference = "sheep";  
101            }  
102        } else {  
103            preference = "wheat";  
104        }  
105    } else if (isAbledToBuildSettlement) {  
106        if (bricks < wood || bricks == wood) {  
107            if (bricks < sheep || bricks == sheep) {  
108                if (bricks < wheat || bricks == wheat) {  
109                    preference = "brick";  
110                } else if (bricks < sheep || bricks == sheep) {  
111                    preference = "sheep";  
112                } else {  
113                    preference = "wheat";  
114                }  
115            } else {  
116                preference = "sheep";  
117            }  
118        } else {  
119            preference = "wheat";  
120        }  
121    } else if (isAbledToBuildCity) {  
122        if (bricks < wood || bricks == wood) {  
123            if (bricks < sheep || bricks == sheep) {  
124                if (bricks < wheat || bricks == wheat) {  
125                    preference = "brick";  
126                } else if (bricks < sheep || bricks == sheep) {  
127                    preference = "sheep";  
128                } else {  
129                    preference = "wheat";  
130                }  
131            } else {  
132                preference = "sheep";  
133            }  
134        } else {  
135            preference = "wheat";  
136        }  
137    } else if (isAbledToBuildVillage) {  
138        if (bricks < wood || bricks == wood) {  
139            if (bricks < sheep || bricks == sheep) {  
140                if (bricks < wheat || bricks == wheat) {  
141                    preference = "brick";  
142                } else if (bricks < sheep || bricks == sheep) {  
143                    preference = "sheep";  
144                } else {  
145                    preference = "wheat";  
146                }  
147            } else {  
148                preference = "sheep";  
149            }  
150        } else {  
151            preference = "wheat";  
152        }  
153    } else if (isAbledToBuildRoad) {  
154        if (bricks < wood || bricks == wood) {  
155            if (bricks < sheep || bricks == sheep) {  
156                if (bricks < wheat || bricks == wheat) {  
157                    preference = "brick";  
158                } else if (bricks < sheep || bricks == sheep) {  
159                    preference = "sheep";  
160                } else {  
161                    preference = "wheat";  
162                }  
163            } else {  
164                preference = "sheep";  
165            }  
166        } else {  
167            preference = "wheat";  
168        }  
169    } else if (isAbledToBuildSettlement) {  
170        if (bricks < wood || bricks == wood) {  
171            if (bricks < sheep || bricks == sheep) {  
172                if (bricks < wheat || bricks == wheat) {  
173                    preference = "brick";  
174                } else if (bricks < sheep || bricks == sheep) {  
175                    preference = "sheep";  
176                } else {  
177                    preference = "wheat";  
178                }  
179            } else {  
180                preference = "sheep";  
181            }  
182        } else {  
183            preference = "wheat";  
184        }  
185    } else if (isAbledToBuildCity) {  
186        if (bricks < wood || bricks == wood) {  
187            if (bricks < sheep || bricks == sheep) {  
188                if (bricks < wheat || bricks == wheat) {  
189                    preference = "brick";  
190                } else if (bricks < sheep || bricks == sheep) {  
191                    preference = "sheep";  
192                } else {  
193                    preference = "wheat";  
194                }  
195            } else {  
196                preference = "sheep";  
197            }  
198        } else {  
199            preference = "wheat";  
200        }  
201    } else if (isAbledToBuildVillage) {  
202        if (bricks < wood || bricks == wood) {  
203            if (bricks < sheep || bricks == sheep) {  
204                if (bricks < wheat || bricks == wheat) {  
205                    preference = "brick";  
206                } else if (bricks < sheep || bricks == sheep) {  
207                    preference = "sheep";  
208                } else {  
209                    preference = "wheat";  
210                }  
211            } else {  
212                preference = "sheep";  
213            }  
214        } else {  
215            preference = "wheat";  
216        }  
217    } else if (isAbledToBuildRoad) {  
218        if (bricks < wood || bricks == wood) {  
219            if (bricks < sheep || bricks == sheep) {  
220                if (bricks < wheat || bricks == wheat) {  
221                    preference = "brick";  
222                } else if (bricks < sheep || bricks == sheep) {  
223                    preference = "sheep";  
224                } else {  
225                    preference = "wheat";  
226                }  
227            } else {  
228                preference = "sheep";  
229            }  
230        } else {  
231            preference = "wheat";  
232        }  
233    } else if (isAbledToBuildSettlement) {  
234        if (bricks < wood || bricks == wood) {  
235            if (bricks < sheep || bricks == sheep) {  
236                if (bricks < wheat || bricks == wheat) {  
237                    preference = "brick";  
238                } else if (bricks < sheep || bricks == sheep) {  
239                    preference = "sheep";  
240                } else {  
241                    preference = "wheat";  
242                }  
243            } else {  
244                preference = "sheep";  
245            }  
246        } else {  
247            preference = "wheat";  
248        }  
249    } else if (isAbledToBuildCity) {  
250        if (bricks < wood || bricks == wood) {  
251            if (bricks < sheep || bricks == sheep) {  
252                if (bricks < wheat || bricks == wheat) {  
253                    preference = "brick";  
254                } else if (bricks < sheep || bricks == sheep) {  
255                    preference = "sheep";  
256                } else {  
257                    preference = "wheat";  
258                }  
259            } else {  
260                preference = "sheep";  
261            }  
262        } else {  
263            preference = "wheat";  
264        }  
265    } else if (isAbledToBuildVillage) {  
266        if (bricks < wood || bricks == wood) {  
267            if (bricks < sheep || bricks == sheep) {  
268                if (bricks < wheat || bricks == wheat) {  
269                    preference = "brick";  
270                } else if (bricks < sheep || bricks == sheep) {  
271                    preference = "sheep";  
272                } else {  
273                    preference = "wheat";  
274                }  
275            } else {  
276                preference = "sheep";  
277            }  
278        } else {  
279            preference = "wheat";  
280        }  
281    } else if (isAbledToBuildRoad) {  
282        if (bricks < wood || bricks == wood) {  
283            if (bricks < sheep || bricks == sheep) {  
284                if (bricks < wheat || bricks == wheat) {  
285                    preference = "brick";  
286                } else if (bricks < sheep || bricks == sheep) {  
287                    preference = "sheep";  
288                } else {  
289                    preference = "wheat";  
290                }  
291            } else {  
292                preference = "sheep";  
293            }  
294        } else {  
295            preference = "wheat";  
296        }  
297    } else if (isAbledToBuildSettlement) {  
298        if (bricks < wood || bricks == wood) {  
299            if (bricks < sheep || bricks == sheep) {  
300                if (bricks < wheat || bricks == wheat) {  
301                    preference = "brick";  
302                } else if (bricks < sheep || bricks == sheep) {  
303                    preference = "sheep";  
304                } else {  
305                    preference = "wheat";  
306                }  
307            } else {  
308                preference = "sheep";  
309            }  
310        } else {  
311            preference = "wheat";  
312        }  
313    } else if (isAbledToBuildCity) {  
314        if (bricks < wood || bricks == wood) {  
315            if (bricks < sheep || bricks == sheep) {  
316                if (bricks < wheat || bricks == wheat) {  
317                    preference = "brick";  
318                } else if (bricks < sheep || bricks == sheep) {  
319                    preference = "sheep";  
320                } else {  
321                    preference = "wheat";  
322                }  
323            } else {  
324                preference = "sheep";  
325            }  
326        } else {  
327            preference = "wheat";  
328        }  
329    } else if (isAbledToBuildVillage) {  
330        if (bricks < wood || bricks == wood) {  
331            if (bricks < sheep || bricks == sheep) {  
332                if (bricks < wheat || bricks == wheat) {  
333                    preference = "brick";  
334                } else if (bricks < sheep || bricks == sheep) {  
335                    preference = "sheep";  
336                } else {  
337                    preference = "wheat";  
338                }  
339            } else {  
340                preference = "sheep";  
341            }  
342        } else {  
343            preference = "wheat";  
344        }  
345    } else if (isAbledToBuildRoad) {  
346        if (bricks < wood || bricks == wood) {  
347            if (bricks < sheep || bricks == sheep) {  
348                if (bricks < wheat || bricks == wheat) {  
349                    preference = "brick";  
350                } else if (bricks < sheep || bricks == sheep) {  
351                    preference = "sheep";  
352                } else {  
353                    preference = "wheat";  
354                }  
355            } else {  
356                preference = "sheep";  
357            }  
358        } else {  
359            preference = "wheat";  
360        }  
361    } else if (isAbledToBuildSettlement) {  
362        if (bricks < wood || bricks == wood) {  
363            if (bricks < sheep || bricks == sheep) {  
364                if (bricks < wheat || bricks == wheat) {  
365                    preference = "brick";  
366                } else if (bricks < sheep || bricks == sheep) {  
367                    preference = "sheep";  
368                } else {  
369                    preference = "wheat";  
370                }  
371            } else {  
372                preference = "sheep";  
373            }  
374        } else {  
375            preference = "wheat";  
376        }  
377    } else if (isAbledToBuildCity) {  
378        if (bricks < wood || bricks == wood) {  
379            if (bricks < sheep || bricks == sheep) {  
380                if (bricks < wheat || bricks == wheat) {  
381                    preference = "brick";  
382                } else if (bricks < sheep || bricks == sheep) {  
383                    preference = "sheep";  
384                } else {  
385                    preference = "wheat";  
386                }  
387            } else {  
388                preference = "sheep";  
389            }  
390        } else {  
391            preference = "wheat";  
392        }  
393    } else if (isAbledToBuildVillage) {  
394        if (bricks < wood || bricks == wood) {  
395            if (bricks < sheep || bricks == sheep) {  
396                if (bricks < wheat || bricks == wheat) {  
397                    preference = "brick";  
398                } else if (bricks < sheep || bricks == sheep) {  
399                    preference = "sheep";  
400                } else {  
401                    preference = "wheat";  
402                }  
403            } else {  
404                preference = "sheep";  
405            }  
406        } else {  
407            preference = "wheat";  
408        }  
409    } else if (isAbledToBuildRoad) {  
410        if (bricks < wood || bricks == wood) {  
411            if (bricks < sheep || bricks == sheep) {  
412                if (bricks < wheat || bricks == wheat) {  
413                    preference = "brick";  
414                } else if (bricks < sheep || bricks == sheep) {  
415                    preference = "sheep";  
416                } else {  
417                    preference = "wheat";  
418                }  
419            } else {  
420                preference = "sheep";  
421            }  
422        } else {  
423            preference = "wheat";  
424        }  
425    } else if (isAbledToBuildSettlement) {  
426        if (bricks < wood || bricks == wood) {  
427            if (bricks < sheep || bricks == sheep) {  
428                if (bricks < wheat || bricks == wheat) {  
429                    preference = "brick";  
430                } else if (bricks < sheep || bricks == sheep) {  
431                    preference = "sheep";  
432                } else {  
433                    preference = "wheat";  
434                }  
435            } else {  
436                preference = "sheep";  
437            }  
438        } else {  
439            preference = "wheat";  
440        }  
441    } else if (isAbledToBuildCity) {  
442        if (bricks < wood || bricks == wood) {  
443            if (bricks < sheep || bricks == sheep) {  
444                if (bricks < wheat || bricks == wheat) {  
445                    preference = "brick";  
446                } else if (bricks < sheep || bricks == sheep) {  
447                    preference = "sheep";  
448                } else {  
449                    preference = "wheat";  
450                }  
451            } else {  
452                preference = "sheep";  
453            }  
454        } else {  
455            preference = "wheat";  
456        }  
457    } else if (isAbledToBuildVillage) {  
458        if (bricks < wood || bricks == wood) {  
459            if (bricks < sheep || bricks == sheep) {  
460                if (bricks < wheat || bricks == wheat) {  
461                    preference = "brick";  
462                } else if (bricks < sheep || bricks == sheep) {  
463                    preference = "sheep";  
464                } else {  
465                    preference = "wheat";  
466                }  
467            } else {  
468                preference = "sheep";  
469            }  
470        } else {  
471            preference = "wheat";  
472        }  
473    } else if (isAbledToBuildRoad) {  
474        if (bricks < wood || bricks == wood) {  
475            if (bricks < sheep || bricks == sheep) {  
476                if (bricks < wheat || bricks == wheat) {  
477                    preference = "brick";  
478                } else if (bricks < sheep || bricks == sheep) {  
479                    preference = "sheep";  
480                } else {  
481                    preference = "wheat";  
482                }  
483            } else {  
484                preference = "sheep";  
485            }  
486        } else {  
487            preference = "wheat";  
488        }  
489    } else if (isAbledToBuildSettlement) {  
490        if (bricks < wood || bricks == wood) {  
491            if (bricks < sheep || bricks == sheep) {  
492                if (bricks < wheat || bricks == wheat) {  
493                    preference = "brick";  
494                } else if (bricks < sheep || bricks == sheep) {  
495                    preference = "sheep";  
496                } else {  
497                    preference = "wheat";  
498                }  
499            } else {  
500                preference = "sheep";  
501            }  
502        } else {  
503            preference = "wheat";  
504        }  
505    } else if (isAbledToBuildCity) {  
506        if (bricks < wood || bricks == wood) {  
507            if (bricks < sheep || bricks == sheep) {  
508                if (bricks < wheat || bricks == wheat) {  
509                    preference = "brick";  
510                } else if (bricks < sheep || bricks == sheep) {  
511                    preference = "sheep";  
512                } else {  
513                    preference = "wheat";  
514                }  
515            } else {  
516                preference = "sheep";  
517            }  
518        } else {  
519            preference = "wheat";  
520        }  
521    } else if (isAbledToBuildVillage) {  
522        if (bricks < wood || bricks == wood) {  
523            if (bricks < sheep || bricks == sheep) {  
524                if (bricks < wheat || bricks == wheat) {  
525                    preference = "brick";  
526                } else if (bricks < sheep || bricks == sheep) {  
527                    preference = "sheep";  
528                } else {  
529                    preference = "wheat";  
530                }  
531            } else {  
532                preference = "sheep";  
533            }  
534        } else {  
535            preference = "wheat";  
536        }  
537    } else if (isAbledToBuildRoad) {  
538        if (bricks < wood || bricks == wood) {  
539            if (bricks < sheep || bricks == sheep) {  
540                if (bricks < wheat || bricks == wheat) {  
541                    preference = "brick";  
542                } else if (bricks < sheep || bricks == sheep) {  
543                    preference = "sheep";  
544                } else {  
545                    preference = "wheat";  
546                }  
547            } else {  
548                preference = "sheep";  
549            }  
550        } else {  
551            preference = "wheat";  
552        }  
553    } else if (isAbledToBuildSettlement) {  
554        if (bricks < wood || bricks == wood) {  
555            if (bricks < sheep || bricks == sheep) {  
556                if (bricks < wheat || bricks == wheat) {  
557                    preference = "brick";  
558                } else if (bricks < sheep || bricks == sheep) {  
559                    preference = "sheep";  
560                } else {  
561                    preference = "wheat";  
562                }  
563            } else {  
564                preference = "sheep";  
565            }  
566        } else {  
567            preference = "wheat";  
568        }  
569    } else if (isAbledToBuildCity) {  
570        if (bricks < wood || bricks == wood) {  
571            if (bricks < sheep || bricks == sheep) {  
572                if (bricks < wheat || bricks == wheat) {  
573                    preference = "brick";  
574                } else if (bricks < sheep || bricks == sheep) {  
575                    preference = "sheep";  
576                } else {  
577                    preference = "wheat";  
578                }  
579            } else {  
580                preference = "sheep";  
581            }  
582        } else {  
583            preference = "wheat";  
584        }  
585    } else if (isAbledToBuildVillage) {  
586        if (bricks < wood || bricks == wood) {  
587            if (bricks < sheep || bricks == sheep) {  
588                if (bricks < wheat || bricks == wheat) {  
589                    preference = "brick";  
590                } else if (bricks < sheep || bricks == sheep) {  
591                    preference = "sheep";  
592                } else {  
593                    preference = "wheat";  
594                }  
595            } else {  
596                preference = "sheep";  
597            }  
598        } else {  
599            preference = "wheat";  
600        }  
601    } else if (isAbledToBuildRoad) {  
602        if (bricks < wood || bricks == wood) {  
603            if (bricks < sheep || bricks == sheep) {  
604                if (bricks < wheat || bricks == wheat) {  
605                    preference = "brick";  
606                } else if (bricks < sheep || bricks == sheep) {  
607                    preference = "sheep";  
608                } else {  
609                    preference = "wheat";  
610                }  
611            } else {  
612                preference = "sheep";  
613            }  
614        } else {  
615            preference = "wheat";  
616        }  
617    } else if (isAbledToBuildSettlement) {  
618        if (bricks < wood || bricks == wood) {  
619            if (bricks < sheep || bricks == sheep) {  
620                if (bricks < wheat || bricks == wheat) {  
621                    preference = "brick";  
622                } else if (bricks < sheep || bricks == sheep) {  
623                    preference = "sheep";  
624                } else {  
625                    preference = "wheat";  
626                }  
627            } else {  
628                preference = "sheep";  
629            }  
630        } else {  
631            preference = "wheat";  
632        }  
633    } else if (isAbledToBuildCity) {  
634        if (bricks < wood || bricks == wood) {  
635            if (bricks < sheep || bricks == sheep) {  
636                if (bricks < wheat || bricks == wheat) {  
637                    preference = "brick";  
638                } else if (bricks < sheep || bricks == sheep) {  
639                    preference = "sheep";  
640                } else {  
641                    preference = "wheat";  
642                }  
643            } else {  
644                preference = "sheep";  
645            }  
646        } else {  
647            preference = "wheat";  
648        }  
649    } else if (isAbledToBuildVillage) {  
650        if (bricks < wood || bricks == wood) {  
651            if (bricks < sheep || bricks == sheep) {  
652                if (bricks < wheat || bricks == wheat) {  
653                    preference = "brick";  
654                } else if (bricks < sheep || bricks == sheep) {  
655                    preference = "sheep";  
656                } else {  
657                    preference = "wheat";  
658                }  
659            } else {  
660                preference = "sheep";  
661            }  
662        } else {  
663            preference = "wheat";  
664        }  
665    } else if (isAbledToBuildRoad) {  
666        if (bricks < wood || bricks == wood) {  
667            if (bricks < sheep || bricks == sheep) {  
668                if (bricks < wheat || bricks == wheat) {  
669                    preference = "brick";  
670                } else if (bricks < sheep || bricks == sheep) {  
671                    preference = "sheep";  
672                } else {  
673                    preference = "wheat";  
674                }  
675            } else {  
676                preference = "sheep";  
677            }  
678        } else {  
679            preference = "wheat";  
680        }  
681    } else if (isAbledToBuildSettlement) {  
682        if (bricks < wood || bricks == wood) {  
683            if (bricks < sheep || bricks == sheep) {  
684                if (bricks < wheat || bricks == wheat) {  
685                    preference = "brick";  
686                } else if (bricks < sheep || bricks == sheep) {  
687                    preference = "sheep";  
688                } else {  
689                    preference = "wheat";  
690                }  
691            } else {  
692                preference = "sheep";  
693            }  
694        } else {  
695            preference = "wheat";  
696        }  
697    } else if (isAbledToBuildCity) {  
698        if (bricks < wood || bricks == wood) {  
699            if (bricks < sheep || bricks == sheep) {  
700                if (bricks < wheat || bricks == wheat) {  
701                    preference = "brick";  
702                } else if (bricks < sheep || bricks == sheep) {  
703                    preference = "sheep";  
704                } else {  
705                    preference = "wheat";  
706                }  
707            } else {  
708                preference = "sheep";  
709            }  
710        } else {  
711            preference = "wheat";  
712        }  
713    } else if (isAbledToBuildVillage) {  
714        if (bricks < wood || bricks == wood) {  
715            if (bricks < sheep || bricks == sheep) {  
716                if (bricks < wheat || bricks == wheat) {  
717                    preference = "brick";  
718                } else if (bricks < sheep || bricks == sheep) {  
719                    preference = "sheep";  
720                } else {  
721                    preference = "wheat";  
722                }  
723            } else {  
724                preference = "sheep";  
725            }  
726        } else {  
727            preference = "wheat";  
728        }  
729    } else if (isAbledToBuildRoad) {  
730        if (bricks < wood || bricks == wood) {  
731            if (bricks <
```

```

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 4.33: Java-Klasse Status: CalculatePreferencePlayer2

## 4.6 Bug-Fixes

Während der Entwicklung der Software traten mehrere Bugs auf, die es zu lösen galt. Einige dieser Bugs wurden bereits in anderen Abschnitten dieses Kapitels kurz angesprochen. Nachfolgend sollen diese allerdings nochmals aufgelistet und ihr Auftreten sowie ihre Lösung detaillierter besprochen werden.

Der erste Bug zeigte sich beim Generieren des Spielfelds, womit u.a. erreicht werden soll, dass einem Feld des Typus Wüste keine Zahl zugeordnet wird. Dies wird erreicht, solange die Wüste nicht direkt in der Mitte des Spielfeldes liegt, davor der Generierung zunächst Prefabs auf jedem einzelnen Feld platziert werden, die später befüllt und im Falle der Wüste gelöscht werden. Liegt die Wüste allerdings in der Mitte des Spielfeldes, so wird der Prefab nicht befüllt, aber auch nicht gelöscht. Dieser Bug ist deshalb nicht gravierend, weil er das Spielgeschehen an sich nicht beeinflusst. Da er nicht gelöst wurde, wird er erneut im Ausblick (siehe Abschnitt 6.2) aufgegriffen.

Zudem existierte der Bug, dass die erste Siedlung eines Spielers nicht durch einen Bauplatz für Städte ergänzt wurde. Dieser Bug trat allerdings nur bei KI-Spielern auf. Der Grund für das Auftreten dieses Bugs schien an der Geschwindigkeit der Abfrage zu liegen. Offensichtlich wurde nach der Ausführung des bestehenden Plans so schnell eine neue Anfrage gesendet, dass der Bauplatz noch nicht erstellt wurde. Dieses Problem konnte gelöst

werden, indem der Abschnitt des *Gamemanagers* - welcher die umliegenden Siedlungsbauplätze löscht und den Städtebauplatz erzeugt - nach dem Bau einer Siedlung in die Klasse *buildVillage* verlegt wurde.

Ein weiterer Bug entstand durch das Zusammenspiel von Spiel und KI über die Schnittstelle: Hierbei zeigte sich, dass die KI in einigen Fällen eine leere Antwort lieferte. Da das Weiterspielen in diesen Fällen unmöglich war, wurde der Bug auf zweifache Weise gelöst. Zunächst fängt eine entsprechende Exception den Fehler auf Seiten des Spiels ab und erlaubt der KI noch zwei weitere Versuche. Des Weiteren wird durch die Schnittstelle nun in jedem Fall ein Plan zur Antwort hinzugefügt. Dieser Plan besteht im Zweifelsfall daraus, einen Zug zu beenden oder zu Würfeln ? abhängig davon, welcher Zustand gerade herrscht.

Außerdem kam es vor, dass die KI teilweise unfaähig zu entscheiden war, welcher Fall am ehesten auf die jeweilige Situation passte. Sie fand in einigen Fällen keine Lösung und lieferte nahezu identische Ähnlichkeitswerte. Dieses Problem konnte durch eine Erweiterung der Fallbasis um zusätzliche Fälle sowie eine Anpassung der bestehenden Fälle gelöst werden. Zudem schien die Gewichtung einzelner Symptome mehr Nachteile als Vorteile mit sich zu bringen, weshalb dieser Schritt aus dem *Retrieval* entfernt wurde.

Ein letzter Bug, der hier erwähnt werden soll, trat auf, wenn menschliche Spieler versuchten, während der ersten beiden Züge zusätzlich zu den kostenlosen Objekten weitere Straßen, Siedlungen oder Städte zu bauen. Diese Aktion muss ihnen verboten werden, da es auch KI-Spielern untersagt ist, und der Grundsatz der Fairness gelten soll. Der Plan einer KI wird ein erstes Mal beim Erstellen und ein zweites Mal vor seiner Ausführung überprüft. Außerdem wird die Ausführung eines Plan über die gleichen Methoden wie bei einem menschlichen Spieler gesteuert. Das bedeutet, wenn die KI maximal die gleichen Züge wie ein Mensch durchführen kann, so beschränken die zwei Überprüfungen die KI zusätzlich, sofern diese nicht genauso oder weniger eng wie die für den menschlichen Spieler gefasst sind. Somit musste die Einhaltung der Regeln in den Bedingungen zur Ausführung eines Zuges angepasst werden: Es musste dafür gesorgt werden, dass einige Optionen nur, andere gerade nicht in den ersten beiden Zügen erlaubt sind. Dies wurde hauptsächlich innerhalb der Update-Methode realisiert, die für solche Überprüfungen zur Verfügung steht.

## 4.7 Anpassungen

Nachdem im Kapitel 3.2 die Planung der Implementierung beschrieben wurde, dient dieses Kapitel der Erläuterung der Umsetzung. Hierbei ergaben sich teilweise große Diskrepanzen zwischen der Planung und der Realisierung, weshalb dieser Abschnitt auf diese Unterschiede eingehen und erläutern soll, weshalb sich teilweise gegen die ursprüngliche Konzeption entschieden wurde.

### 4.7.1 Anpassungen am Spiel

Im Vergleich zum Entwurf wurden in der Implementation des Spiels nur wenige Änderungen vorgenommen. So erfolgte beispielsweise der Aufbau des Spielfelds wie geplant. Dieser berücksichtigt alle Aspekte, die im Entwurf geschildert wurden, und dessen Implementation wird im Abschnitt 4.2 detailliert erläutert. Des Weiteren sind die ersten beiden Züge und das übrige Spiel nach Maßgabe der Konzeption gestaltet. Damit das Spiel gewonnen werden kann, wurden ? wie geplant - 10 Siegpunkte als Standardeinstellung festgelegt.

Es wurden lediglich kleinere Details vernachlässigt, was eng mit den Kann- bzw. Soll-Anforderungen verknüpft ist. So ist der Handel kein Teil des gewöhnlichen Spielzuges

und Errungenschaften wie *die größte Rittermacht* können nicht zum Sieg beitragen, da keine Ereigniskarten zur Verfügung stehen. Des Weiteren wurde auch der Räuber nicht im Spiel umgesetzt. Davon abgesehen entspricht das Spiel dem Entwurf. Lediglich auf einen zusätzlichen Aspekt, der in der Implementation bereits beschrieben wurde, soll an dieser Stelle noch hingewiesen werden. So wird im Regelwerk des Spiels *Siedler von Catan* (s. Anhang) beschrieben, dass die in den ersten beiden Zügen kostenlos gebaute Straßen nur an derjenigen Siedlung angebaut werden können, die im jeweiligen Zug errichtet wurde. Dieser Aspekt wurde nicht in die Implementierung einbezogen.

#### 4.7.2 Anpassungen an der Schnittstelle

Die Schnittstelle bzw. die Kommunikation zwischen Spiel und KI ist der Teil der Implementation, der die größte Diskrepanz zum Entwurf aufweist. Es wurde zunächst geplant, dass unterschiedliche Nachrichten über die Schnittstelle hin und her geschickt werden. Hierbei wurde häufig die KI als der Initiator einer Nachricht genannt und geplant, dass das Spiel antworten und der KI die nötigen Informationen auf Anfrage bereitstellen würde. Dieses Konzept wurde nicht angewandt, sondern durch ein einfacheres ersetzt, welches die Anforderungen im gleichen Umfang erfüllt.

In der aktuellen Umsetzung agiert die Schnittstelle auf Seiten der KI als Anfragen verarbeitender Server. Dieser Server hat einen Client: das Spiel. Dementsprechend sendet das Spiel seine Anfragen an die Schnittstelle und nicht die Schnittstelle an das Spiel. Eine Anfrage besteht dabei aus der Situation, die alle nötigen Informationen enthält. Serverseitig kann daraufhin eine Antwort generiert werden. Diese Antwort ist so aufgebaut wie die Anfrage, jedoch enthält sie zusätzlich einen Plan, welcher vom Spiel empfangen und ggf. verarbeitet wird.

Dieses Vorgehen bedarf weniger Komplexität in der Kommunikation. Es muss lediglich eine Art von Nachricht versendet bzw. empfangen und verarbeitet werden. Das Nachrichtenformat ist dabei für alle Zustände des Spiels identisch. Auch die Zeitpunkte, zu denen Kommunikation erfolgt, konnten reduziert werden. Ein Spieler erhält nur dann Informationen, wenn er am Zug ist und kann daraufhin einen Plan ausarbeiten. Wenn dieser Plan ausgeführt wurde und der Zug des jeweiligen Spielers noch nicht beendet ist, wird eine erneute Anfrage gestellt, die wieder vom Spieler mit einem Plan beantwortet wird. Dieses Frage-Antwort-Prozedere wird so lange fortgesetzt, bis der Spieler seinen Zug beendet oder das Beenden des Zuges durch das Spiel selbst eingeleitet wird. Anschließend an die Zugbeendigung wird eine neue Kommunikation mit dem nächsten Spieler eingeleitet. Welcher Spieler jeweils Informationen erhält, ist in der Situationsbeschreibung enthalten und wird serverseitig ausgewertet. Abhängig vom Spieler können so Nuancen hinsichtlich der Strategie festgelegt werden.

Das beschriebene Verfahren erfüllt die gestellten Anforderungen auch weiterhin. Ein Aspekt, den es besonders zu bedenken gilt, ist der der Fairness. Dieser behält auch durch das eingesetzte Verfahren Bestand, da mit der Situationsbeschreibung zwar das Spielgeschehen übermittelt wird, dabei den KIs aber keine Informationen geliefert werden, auf die ein menschlicher Spieler nicht auch Zugriff hätte.

## 5 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 auftreten. Abschließend wird ein durchgeführter Test beleuchtet, der einige weitere Probleme aufzeigt, die ebenfalls betrachtet werden sollen.

### 5.1 Muss-Anforderungen

Die erste lautet, dass das Spielfeld nach den Regeln des Originals aufgebaut sein muss. Diese Anforderung ist erfüllt worden, da die sechseckigen Felder zufällig verteilt, den Regeln entsprechend angeordnet und die darauf liegenden Zahlen in einer vorgegebenen Reihe auf diesen Feldern verteilt werden. 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 3.2 zum Spiel fordern das Erlauben vom Bau von Straßen, Siedlungen und Städten. All diese Objekte stehen dem Bau zur Verfügung, wie die Abbildung 5.1 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.

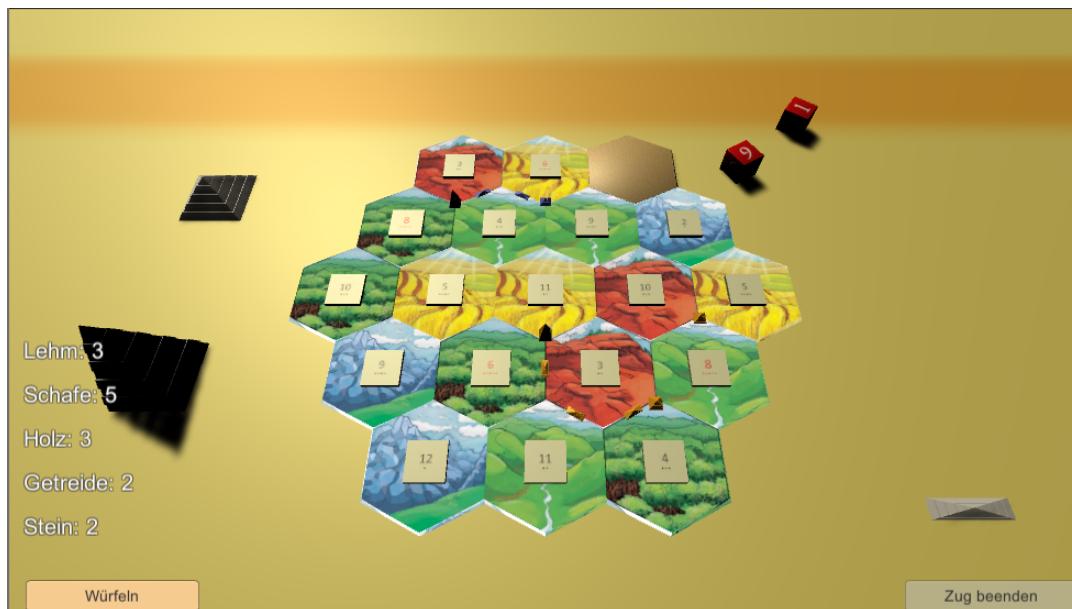


Abbildung 5.1: Bau von Städten, Siedlungen und Straßen (eigene Darstellung)

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 5.2 zu sehen.



Abbildung 5.2: Ausgabe des Gewinners auf dem Bildschirm (eigene Darstellung)

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 die 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 kann auftreten, wenn bereits alle Siedlungen zu Städten aufgewertet wurden und der Spieler dennoch den entsprechenden Modus aktiviert. In solch einem Fall, wird 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 keine Voraussetzung zum Spielen darstellen. Eine dieser Anforderungen 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 löst jedoch nur einen Teil des Problems. Abgesehen von den ersten beiden Zügen darf der Spieler eine Siedlung nur an eine bereits bestehende gebaute Straße gebaut werden. Hierzu macht sich das Spiel erneut Entfernung 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.

## 5.2 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 oder mit der Bank 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 ausgewählt, 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 initiale KIs bereitgestellt. In Abschnitt 4.5 wird darauf näher eingegangen. Daher wird 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 initialen 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 denen in Abschnitt 4.5 beschriebenen Testreaktionen der KI.

### 5.3 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. Hieran 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 sie am Zug ist und die Situation so deutlich 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 von 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 4.4 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.

Zu den Soll- und Kann-Anforderungen an die Schnittstelle bleibt festzuhalten, dass diese im gleichen Umfang umgesetzt wurden, wie sie den umgesetzten Anforderungen an das Spiel dienen.

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 und das angesteuerte Programm im Quellcode des Spiels vermerkt wird.

## 5.4 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 Zeiter sparnis durch die Konzeption entsteht.

Auch die Implementierung von Testreaktionen hätte besser konzipiert werden können und hängt eng 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 Fällen, 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.

Das Vorherbestimmen der Antworten der fertigen KI erwies sich als weiteres Problem. Hierbei wurden zwar Standardfälle in die Fallbasis eingefügt, die für ein bestimmtes Verhalten der KIs sorgen sollten. Die KIs verhielten sich allerdings nicht immer wie erwartet. Sodass schließlich einige Einschränkungen in das Retrieval eingeführt und eine Erweiterung der Fallbasis vorgenommen werden musste.

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 wird die Situation an sich verändert. Hierfür musste pro Aktivierungsaktion eine Methode zur Verfügung gestellt werden, die einbezieht, ob es überhaupt aktivierbare Plätze gibt. Diese Information wird in Form eines weiteren Symptoms in den Fällen gespeichert. Konkret handelt es sich dabei um drei Symptome, die mit Wahrheitswerten zu füllen sind, pro Bauplatzart einer.

Am simpelsten ist die Überprüfung von Städtebauplätzen. Hierfür prüft die entsprechende Methode lediglich, 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. Hierfür muss das Spiel diese Information direkt mit in die Situation einfügen. Hierzu wurde eine Methode gestaltet, 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 äquivalent einsetzbar. Zu beachten ist, dass hierdurch das Spiel eine 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. Der Grundsatz der Fairness könnte hierdurch zwar als verletzt betrachtet werden jedoch nur theoretisch. In der Praxis ist das Erkennen der gleichen Handlung als unsinnig so grundlegend, dass die KI dadurch nicht bevorteilt wird.

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 3.2 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.

## 5.5 Tests

Zur genaueren Evaluierung der Software wurden 20 Testspiele durchgeführt, bei denen jeweils beide Spieler KI-Spieler waren. Die angewendeten Strategien sind unter Abschnitt 4.5 nachzulesen. Genau genommen, versucht Spieler eins Ressourcen zu gewinnen, um Städte zu bauen und Spieler 2 konzentriert sich auf nötige Ressourcen für Siedlungen und Straßen.

Nach 20 Spielen hatte Spieler 1 sechs Siege zu verzeichnen, Spieler 2 sieben und bei weiteren sieben Spielen kam es zu keiner finalen Entscheidung, weil die Spieler entweder nicht alle Ressourcen erlangten oder sie sich gegenseitig so eingebaut haben, dass keiner mehr den Sieg erlangen konnte. Eine Visualisierung der Ergebnisse ist in Abb. 5.3 zu sehen.

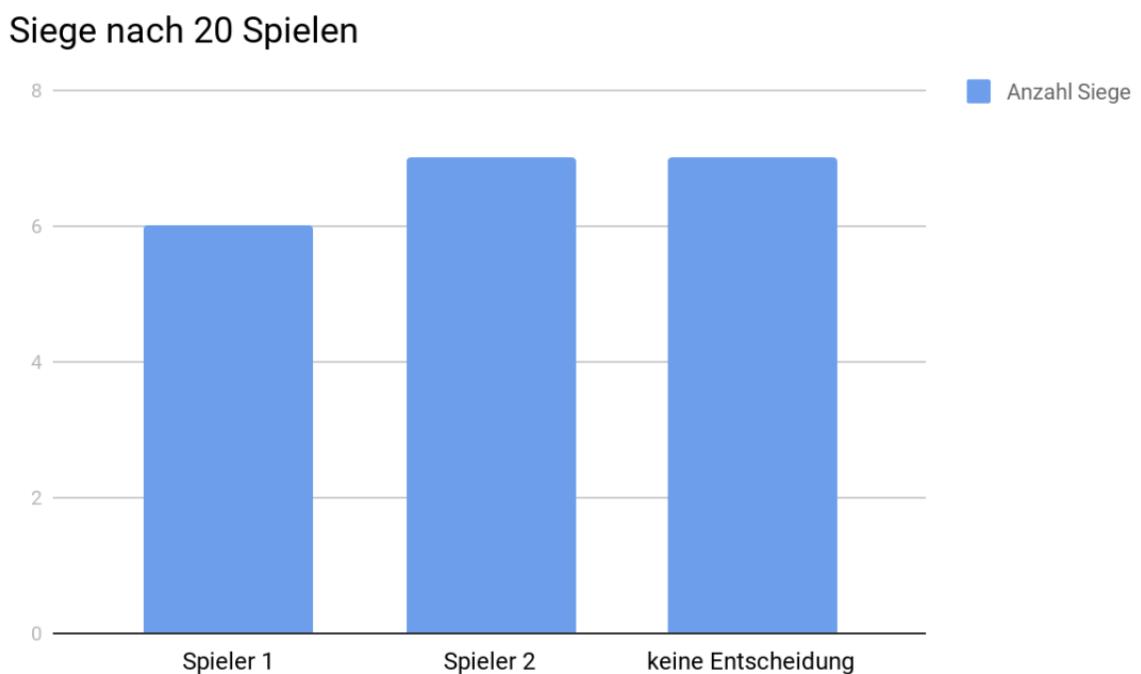


Abbildung 5.3: Säulendiagramm zu den Testergebnissen (eigene Darstellung)

Mithilfe dieser Ergebnisse lassen sich Probleme identifizieren, die noch nicht behoben wurden und einen Grundstein für spätere Verbesserungen legen.

### Die Strategien sorgten für keinen signifikanten Unterschied im Ergebnis:

Da Spieler 1 sechs mal gewann und Spieler 2 sieben der 20 Spiele, scheint die Strategie keinen oder zumindest keinen großen Einfluss auf das Ergebnis zu haben. Möglicherweise sind beide Strategien ungefähr gleich gut oder das Präferieren eines Rohstoffs wirkt sich nicht allzu stark aus.

### Spieler erlangen nicht immer alle nötigen Ressourcen:

Von den gespielten 20 Spielen konnten ganze sieben nicht zu Ende gespielt werden, weil keiner mehr eine Chance auf den Sieg hatte. Bei sechs dieser Spiele war der Grund, dass zu Beginn des Spiels keiner der Spieler Zugang zu allen nötigen Ressourcen erlangte. Ohne die Möglichkeit zum Handeln muss jeder Spieler seine ersten beiden Siedlungen so platzieren, dass er mindestens Zugang zu Lehm, Holz, Schafen und Getreide hat. Diese Ressourcen

sind nötig, um neue Siedlungen bauen zu können und ohne diese ist kein Sieg möglich.

**Der Wüstenbug trat auf:**

In ca. einem von 19 Fällen wird die Mitte des Spielfelds als Wüste gewählt, wenn dies geschieht, wird ein leeres Prefab für Nummern dort platziert. Dies sollte nicht vorkommen, beeinflusst das Spielgeschehen an sich jedoch nicht.

**Spieler können sich gegenseitig einbauen:**

Der übrige Fall, der für ein Spiel ohne Sieger sorgte, ist durch gegenseitiges Einbauen zu erklären. Es kann vorkommen, dass einer oder beide Spieler vermehrt Straßen bauen, weil andere Ressourcen Ihnen gerade nicht zur Verfügung stehen. Dadurch wird in einigen Spielen, dass gesamte Spielfeld durch Straßen belegt und es bleibt kein freier Platz. Die aktuellen KIs Wissen nicht, wie auf eine solche Situation zu reagieren ist und stellen das Handeln ein. Dies führt dazu, dass ein Spiel, welches theoretisch noch beendet werden könnte, aufgrund der Intelligenz der KI keinen Sieger findet.

Abschließend bleibt zu den identifizierten Problemen zu sagen, dass innerhalb der 20 Spiele keine Situation auftrat, auf die die Spieler anders als erwartet reagierten. Es kam demnach nicht zu Fehlern, die einen Spielabbruch zur Folge gehabt hätten. Hierdurch kann nun zumindest vermutet werden, dass die Software sicher läuft und für eine Einhaltung der geltenden Spielregeln sorgt. Außerdem haben auch alle Funktionalitäten wie erwartet funktioniert. Die beschriebenen Probleme hängen hauptsächlich mit der Intelligenz der KI zusammen und nicht mit dem Spiel an sich.

## 6 Fazit & Ausblick

Ziel dieser Arbeit war es ein rundenbasiertes Strategiespiel für Künstliche Intelligenz spielbar umzusetzen und die Konzeption, Implementation und Evaluation zu dokumentieren. Konkret wurde dabei das bekannte Gemeinschaftsspiel “Siedler von Catan” als Ziel der Implementation gewählt.

Die Arbeit an sich war in mehrere Kapitel unterteilt. So wurde zunächst in Kapitel 1 das Thema eingeleitet und beschrieben, wovon die Arbeit handeln wird. Das zweite Kapitel befasste sich anschließend mit den zum Verständnis der übrigen Arbeit nötigen Grundlagen. Danach wurde sich in Kapitel 3 das erste Mal mit der entstehenden Software beschäftigt. Hier wurde im Detail die Konzeption erläutert, die sich wiederum in die Problemstellung und den konkreten Entwurf aufteilen lässt. Nachdem die Konzeption abgeschlossen wurde, folgt die Implementation des beschriebenen in Kapitel 5. Dieses beschreibt von den ersten Schritten bis zur fertigen Software alle wichtigen Aspekte der Umsetzung. Eine Unterteilung erfolgt innerhalb des Kapitels in Abschnitte über das Spiel selbst, die Schnittstelle und die Implementierung von initialen KIs. Außerdem erfolgt noch eine Betrachtung der Anpassungen, die im Vergleich zum Entwurf vorgenommen wurden. Mit Kapitel 6 fügt sich die Evaluation an. Diese befasst sich zunächst einmal mit den Muss-Anforderungen und bewertet, ob alle erfüllt wurden. Im weiteren Verlauf des Kapitels wird auf die Kann- bzw. Soll-Anforderungen eingegangen. Auch wird geprüft, welche dieser Anforderungen umgesetzt wurden und wie sich die Umsetzung weiterer Anforderungen auswirken würde. Nachdem die Anforderungen an das Spiel überprüft wurden, wird auf die an die Schnittstelle übergegangen. Neben der Überprüfung von Anforderungen werden in der Evaluation auch Tests betrachtet und Schlüsse gezogen, wie die KIs abgeschnitten haben. Abgeschlossen wird das Kapitel durch eine Betrachtung von Problemen, die während der Entwicklung aufgetreten sind. Hierbei wird zwischen behobenen und andauernden unterschieden.

### 6.1 Fazit

Die angefertigte Software und diese schriftliche Ausarbeitung beschreiben nicht die komplette Umsetzung des Spiels “Siedler von Catan” mit allen Details. Stattdessen wurden aus den Spielregeln die wichtigsten Aspekte extrahiert und als Anforderungen formuliert. Diese dienen als Grundlagen, um den Erfolg des Projektes bewerten zu können. Wie im oberen Absatz bereits erwähnt, wurde die Erfüllung dieser Anforderungen in der Evaluation überprüft. Hierbei wurde deutlich, dass alle Muss-Anforderungen umgesetzt wurden und auch ein großer Teil der Kann- bzw. Soll-Anforderungen. Dadurch ergibt sich insgesamt, dass der Umfang des Spiels im Vergleich zur Brettspielvariante etwas reduziert wurde, aber dennoch ein funktionierendes, in sich stimmiges Produkt geschaffen wurde. Der Umfang ist insofern beschränkt, indem der Handel und der Einsatz des Räubers sowie Ereigniskarten nicht zur Verfügung stehen. Stattdessen wurde sich auf initiale KIs fokussiert. Hierdurch konnte letztendlich erreicht werden, dass die Software bereits funktioniert und die KIs gegeneinander spielen können.

Insgesamt ergibt sich auf diese Weise ein fertiges Produkt und das Ergebnis dieser Arbeit lässt sich als Erfolg bezeichnen. An vielen Stellen im Verlauf des Projektes hätte es zu Verzögerungen kommen können, die die Implementierung des umgesetzten hätten gefährden können. Dies war immer dann der Fall, wenn es zu unvorhergesehenen Problemen kam, wie solche, die in der Evaluation (Kapitel 5) beschrieben werden. Der erfolgreiche Abschluss war jedoch zu keinem Zeitpunkt in Gefahr. Das hängt zum einen mit der Konzeption im Vorhinein und zum anderen mit der zeitlichen Planung zusammen. Es wurde bereits früh mit der Umsetzung begonnen, sodass die schriftliche Arbeit und die Weiterentwicklung

der Software größtenteils zeitgleich erfolgte. Besonders wichtig war hierbei, dass ich mich bereits mit Unity und C# auseinander setzte, als ich die Konzeptionierung noch nicht fertig hatte. Dadurch konnte ich mein gewonnenes Wissen bereits in die Konzeption mit einfließen lassen und diese genauer gestalten. Auch konnte ich die Entwicklung der Software vor der schriftlichen Ausarbeitung abschließen, wodurch weitere zeitliche Probleme verhindert werden konnten.

Abgesehen von der zeitlichen Dimension und der Erstellung eines funktionierenden Produktes war das Projekt auch auf anderen Dimensionen ein Erfolg. Als Bachelorarbeit diente es als Abschluss des Studiums und im besten Fall fließt möglichst viel Wissen, das bis dort im Studium gewonnen wurde, in die Arbeit ein. Dies war hier der Fall. Im Verlauf des Studiums konnte ich mir Kenntnisse über Wissensbasierte Systeme und Softwareentwicklung aneignen. Auch die Erfahrung aus vorherigen Softwareprojekten konnte ich für mich nutzen, um die zeitlichen Ausmaße besser einzuschätzen. Neben diesen Fähigkeiten war es sicherlich auch von Vorteil, dass bereits eine Projektarbeit abgelegt werden musste, die auch zumindest zum Teil aus der Implementierung eines Algorithmus bestand. Dieses gewonnene Wissen trug zum erfolgreichen Abschluss der Entwicklung der Software und der schriftlichen Ausarbeitung bei.

Selbstverständlich musste für die erfolgreiche Entwicklung der Software und dem Abschluss des Projektes sich auch neues Wissen angeeignet werden. Hierbei sind allem voran der Umgang mit Unity und die Programmiersprache C# zu nennen. Zwar kannte ich mich bereits in ähnlichen Sprachen wie Java und C++ aus, aber mit C# hatte ich davor nur wenig Kontakt gehabt, geschweige denn ganze Projekte darin umgesetzt. Es half mir daher sehr, dass ich mich schon vor Projektbeginn mit C# auseinandersetze und kleinere Projekte realisierte. So konnte ich letztendlich schneller das eigentliche Projekt umsetzen. Diese Lernzeit brauchte ich aber weniger zum Lernen der Programmiersprache an sich, sondern für den Umgang mit Unity. Mit Unity oder einer anderen Game-Engine hatte ich bis dato keinen Kontakt gehabt und musste mich daher völlig neu darin einfinden. Ich kombinierte meinen Lernaufwand für C# und Unity, indem ich direkt in Unity begann kleinere C#-Projekte umzusetzen. Diese halfen mir ein allgemeines Verständnis über Scenes, Prefabs, Scripts und weitere Aspekte von Unity zu erlangen. Im Endeffekt stellte Unity die deutlich größere Hürde als C# dar, weshalb ich darauf auch mehr Zeit aufwendete. Nichtsdestotrotz stellten beide kein so großes Hindernis dar, dass sie eine erfolgreiche Umsetzung des Projektes hätten verhindern können.

Die Umsetzung des Spiels gestaltete sich dann auch nicht als allzu aufwendig. Es gab zwar einige Aspekte, die mit einem hohen Aufwand verbunden waren, aber im Allgemeinen waren Schnittstelle und initiale KI deutlich schwieriger umzusetzen. Hierbei erwies sich die Möglichkeit auf ein ähnliches, bereits abgeschlossenes Projekt zuzugreifen als großer Vorteil. Ich konnte mir die Arbeit von Jannis Hillmann ansehen und auch Aspekte seiner Programmierung ähnlich wiederverwenden. Er setzte ein ähnliches Projekt aber mit Grundlage eines Ego-Shooters um. Ich konnte durch das Nachvollziehen seiner Arbeit und der Übertragung dieses Wissens auf meinen Anwendungsfall enorm viel Zeit sparen. Wenn dieses Projekt nicht zur Verfügung gestanden hätte, dann wäre wahrscheinlich die Anfertigung von initialen KIs aus Zeitgründen kürzer und damit weniger umfangreich ausgefallen.

Abschließend lässt sich, wie eingangs erwähnt, das gesamte Projekt als Erfolg bezeichnen. Nicht nur wurden alle Muss-Anforderungen umgesetzt, es fanden sogar zusätzliche Kann- und Muss-Anforderungen Einzug in die Implementation. Abgesehen davon konnte ich mein bereits im Studium erworbene Wissen einsetzen und mit neuem kombinieren, was auch den praktischen Nutzen von theoretisch gelernten Inhalten darlegt. In Bezug auf

den zeitlichen Rahmen konnten durch die Planung und die Einteilung Probleme vermieden werden.

## 6.2 Ausblick

Wie der Name dieses Abschnitts schon vermuten lässt, sollen hier sowohl für die entwickelte Software als auch für die schriftliche Ausarbeitung Aspekte beschrieben werden, an denen weitere Arbeiten ansetzen können. Als große Punkte sind hier das Lösen von bestehenden Problemen, der Handel zwischen Spielern, der Dieb und das Verwenden von Ereigniskarten zu nennen. Durch die Erweiterung des Spiels um diese Funktionen, würde das Spielen an sich eine komplexere Strategie erfordern bzw. eine bessere Performanz aufweisen. In

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.

### 6.2.1 Verbesserungen

Allgemein stellen alle geschilderten Probleme aus Abschnitt 5.5 Tests einen möglichen Ansatz für Verbesserungen dar.

So ist hier die Lösung des Bugs zu nennen, der auftritt, wenn die Wüste in der Mitte des Spielfelds erstellt wird. Dieser Bug hat keinen Einfluss auf das Spiel an sich, weshalb eine Lösung in kommenden Softwareerweiterungen möglich, aber nicht notwendig ist. Jedenfalls muss zu seiner Lösung lediglich ein Weg gefunden werden, das Prefab auf dem mittleren Sechseck zu löschen. Falls sich dies als zu aufwendig erweisen sollte, kann alternativ auch verhindert werden, dass die Wüste die Mitte belegt.

Des Weiteren müsste ein Vorgehen für den Fall entwickelt werden, dass ein oder beide Spieler keinen Zug mehr machen können. Beispielsweise wenn eine KI, weil sie von der anderen eingebaut wurde oder weil alle Bauplätze bereits belegt sind, nichts mehr tun kann. In solch einem Fall könnte sie zwar ihren Zug beenden, aber an sich würde die andere KI ab diesem Zeitpunkt alleine spielen. Sie könnte nun solange weitermachen, bis sie genug Siegpunkte gesammelt hat. Andere Lösungen wären jedoch auch, der KI zu diesem Zeitpunkt den Sieg einfach zuzuschreiben, da für die andere keine Chance mehr auf den Sieg besteht. Außerdem wäre es möglich, das Spiel dort abzubrechen und den mit mehr Siegpunkten als Sieger zu deklarieren oder das Spiel einfach neu zu starten. Die letzten beiden Optionen sind besonders relevant, wenn beide KIs handlungsunfähig werden.

Ebenfalls gibt es in der aktuellen Form keinen Sieger, sofern zu Beginn des Spiels keiner der Spieler alle nötigen Ressourcen erlangt. Um dies zu lösen, gibt es gleich mehrere Möglichkeiten.

Zurzeit kann ein Spieler nur eine Präferenz wählen. Beide beginnen damit, Lehm und Holz zu präferieren und es bleibt zu hoffen, dass Getreide und Schafe zufällig mit dabei sind. Das ist natürlich kein gutes Vorgehen und sollte in Zukunft angepasst werden. Die Lösung für ein solches Problem wäre entweder, mindestens den Tausch 4 zu 1 mit der Bank zu ermöglichen oder die Fälle um eine Liste an Präferenzen zu erweitern. Hierbei könnte die KI nicht nur eine Präferenz angeben sondern eine Liste, die bis zu jeder Ressourcen enthalten kann. Diese Liste müsste nach Priorität sortiert sein. Jetzt könnte die Auswahl

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

---

des Bauplatzes für Siedlungen nach Priorität erfolgen und im besten Fall der Platz gewählt werden, der die ersten drei Prioritäten abdeckt.

Die Erweiterung der Fälle um Listen würden die Fallbasis allerdings deutlich aufblähen, da jede mögliche Kombination enthalten sein müsste. Als Alternative bietet es sich hier, die Priorisierung gar nicht fallbasiert durchzuführen. Die Prioritäten könnten pro Spieler über eigene Methoden gesetzt werden. Falls dann der Bau einer Siedlung als Plan gewählt wird, kann ein Bauplatz gewählt werden, der möglichst gut passt.

Die zweite Möglichkeit scheint auf den ersten Blick aufwendiger umzusetzen, doch würde sie tatsächlich nicht allzu große Änderungen voraus setzen. Lediglich die Anpassung der Fallbasis und das Finden eines guten Algorithmus, der die Prioritäten abhängig von der Strategie bestimmt, könnten zeitlich kostspielig werden.

Die Implementierung des Handels ist funktionell wenig aufwendig zu implementieren, doch müsste auch hier die Fallbasis wahrscheinlich stark erweitert werden, denn es bleibt zu bestimmen, ob der Tausch einer Ressourcen gegen eine andere gerade sinnvoll ist. Das sollte im besten Fall davon abhängig gemacht werden, ob der Tausch die Situation der KI verbessern würde. Beispielsweise könnte sie prüfen, ob sie danach zum Bau eines Objektes in der Lage ist.

Das letzte übrige Problem ist, dass die verschiedenen Strategien nicht zu signifikant unterschiedlichen Siegraten führen.

Jedoch würde die Umsetzung der obigen Lösungen nicht zu einem größeren Unterschied in der Performanz beider KIs führen. Zwar könnten dann besser Präferenzen genutzt werden, aber meistens darf hat die KI nur einen aktiven Bauplatz zur Verfügung (Abgesehen von den ersten beiden Zügen). Sie baut dann dort und die Präferenz hilft ihr dabei kaum. Die Lösung dieses Problems müsste berücksichtigen, in welche Richtung eine KI Straßen errichten müsste, um schließlich dort eine Siedlungen errichten zu können. Auch dies könnte über Präferenzen realisiert werden, wäre aber aufwendiger, da auch Straßenbauplätze in größerer Entfernung betrachtet würden.

Allgemein scheint die aktuelle KI noch nicht intelligent genug, um eine Konkurrenz für einen menschlichen Spieler darzustellen. Die genannten Änderungen könnten sie allerdings signifikant verbessern. Dadurch würde sie wahrscheinlich nicht das Niveau eines Menschen erreichen, aber sie könnte zumindest in einigen Duellen gewinnen. Hier bleibt noch viel Potenzial, dass es gilt, in Zukunft auszuschöpfen.

### 6.2.2 Erweiterungen

Die Erweiterungen unterscheiden sich in sofern von den Verbesserungen, indem es sich bei Ihnen nicht um Problembehebungen handelt, sondern neue Elemente Einzug in die Software erhalten bzw. Bestehendes signifikant verändert wird. Hierbei ist sich im Sinne der übrigen Kann- und Soll-Anforderungen auf den Handel zwischen Spielern, den Dieb und das Verwenden von Ereigniskarten zu berufen. 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 gilt abzuwägen, 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 auch schon bei den Verbesserungen betrachtet wurde und je nach Umgang mit diesem Zustand könnte das Einbauen von Spielern sogar zur Strategieoption werden. Eine KI würde 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.

Die Verbesserung der KI stellt an sich selbstverständlich auch einen Weg zur Aufwertung der Software dar, der in Zukunft verfolgt werden kann. In den folgenden Abschnitten wird daher auf interessante Strategien eingegangen, die so oder in Verbindung mit Erweiterungen umgesetzt werden können.

### **Strategie 1: Sammeln eines Rohstoffs**

Wenn der Tausch von Rohstoffen mit der Bank oder anderen Spielern implementiert würde, ließe sich das Sammeln eines einzelnen Rohstoffs als valide Strategie einsetzen. 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 muss sich mit wenigen Siedlungen bzw. Städte weniger weit verteilt aufgestellt 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 durchaus 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, Ereigniskarten zu sammeln bis die passende Karte gezogen wird und diese solange aufzubewahren, bis erwartete Nutzen groß genug ist.

### 6.2.3 Weitere Möglichkeiten

In Bezug auf die KIs wäre es sicherlich auch interessant sich in der Theorie mit diesem Thema zu beschäftigen. Es könnte als Folge auf diese Arbeit, eine Ausarbeitung erstellt werden, die sich im Detail mit KIs für rundenbasierte Strategiespiele befasst. Sie könnte einen Blick in die Literatur werfen und die besten Modelle identifizieren, die dann auch praktisch umgesetzt werden können. Hierbei könnte auch über den Tellerrand der Wissenbasierten Systeme geblickt und neue Ansätze könnten in Betracht gezogen werden.

Abschließend bleibt zu sagen, dass es tatsächlich noch viele Optionen gibt, um die Software aufzuwerten und entschieden werden muss, welche Richtung die richtige ist. Der beste Weg zu sein scheint, zunächst einmal Bugs und bestehende Probleme zu lösen. Danach die KIs zu verbessern und schließlich das Spiel zu erweitern, damit daraufhin wieder komplexere KIs geschult werden können. Die Erkenntnisse der Literaturarbeit könnten dann zusätzlich bei Bedarf eingesetzt werden. Dies wäre wahrscheinlich dann am sinnvollsten, wenn die KIs bereits eine hohe Performanz erreicht haben.

## Literaturverzeichnis

- [AHSB14] AGOSTINELLI, Forest ; HOFFMAN, Matthew ; SADOWSKI, Peter ; BALDI, Pierre: Learning Activation Functions to Improve Deep Neural Networks. (2014)
- [Alt19] ALTHOFF, Prof. Dr. Klaus-Dieter: *Vorlesung Wissensbasierte Systeme*. November 2019
- [BB17] BARAHA, Satyakam ; BISWAL, Pradyut K.: Implementation of activation functions for ELM based classifiers. In: *2017 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, IEEE, mar 2017
- [But16] <https://github.com/buttsj/unity-settlers-of-catan>
- [Dav99] DAVIS, Ian L.: Strategies for Strategy Game AI / ActivisionI, Inc. 1999. – Forschungsbericht
- [Hil17] HILLMANN, Jannis: *Konzeption und Entwicklung eines Prototypen für ein lernfähiges Multi-Agenten-System mittels des fallbasierten Schließens im Szenario einer First-Person-Perspektive*, Universität Hildesheim, Diplomarbeit, 2017
- [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
- [NIGM18] NWANKPA, Chigozie ; IJOMAH, Winifred ; GACHAGAN, Anthony ; MARSHALL, Stephen: Activation Functions: Comparison of trends in Practice and Research for Deep Learning. (2018)
- [PBGP96] PUPPE, Frank ; BAMBERGER, Stefan ; GAPPA, Ute ; POECK, Karsten: *Wissensbasierte Diagnose- und Informationssysteme*. Springer Berlin Heidelberg, 1996. <http://dx.doi.org/10.1007/978-3-642-61471-2>. <http://dx.doi.org/10.1007/978-3-642-61471-2>
- [RZL17] RAMACHANDRAN, Prajit ; ZOPH, Barret ; LE, Quoc V.: Searching for Activation Functions. (2017)
- [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
- [Zup94] ZUPAN, J.: Introduction to Artificial Neural Network (ANN) Methods: What They Are and How to Use Them\*. , 1994

## A Anhang

### A.1 Ergänzungen zum abgebildeten Quellcode

---

```
1 public class Tile : MonoBehaviour
2 {
3     public int number;
4 }
```

---

Listing A.1: Hilfsklasse: Tile

---

```
1 /**
2  * Die Buttons muesssen zuerst deaktiviert werden, sonst kann dies von den
3  * KIs ausgenutzt werden
4 */
5 private void Awake()
6 {
7     endTurnBtn.interactable = false;
8     rollDiceBtn.interactable = false;
9 }
10
11 /**
12 * Start wird nach Awake aufgerufen und setzt die Namen der Spieler, sowie
13 * einige Variablen und bestimmt wer den ersten Zug hat.
14 */
15 void Start()
16 {
17     player1.isAI = PlayerPrefs.GetInt("player1Ai") == 1 ? true : false;
18     player2.isAI = PlayerPrefs.GetInt("player2Ai") == 1 ? true : false;
19     player1.name = "Player1";
20     player1.SetGameManager(this);
21     player2.name = "Player2";
22     player2.SetGameManager(this);
23     UpdateActivePlayer(player1);
24
25     if (isSpeedUp) {
26         enableEndTurnWait = 0.1f;
27         addResourcesWait = 0.05f;
28         MakeAiTurnWait = 0.4f;
29         MakeAiMainTurnWait = 0.5f;
30     } else
31     {
32         enableEndTurnWait = 4f;
33         addResourcesWait = 3f;
34         MakeAiTurnWait = 4f;
35         MakeAiMainTurnWait = 5f;
36     }
37     lateStart = true;
38 }
```

---

Listing A.2: Gamemanger: Start- und Awake-Methode

---

```
1 /*
2 * Sammeln von Ressourcen fuer eine bestimmte Siedlung
3 */
```

---

```

4 public void CollectResourcesForVillage(GameObject village)
5 {
6     foreach (GameObject tile in village.GetComponent<Village>().tiles)
7     {
8         if (tile.GetComponentInChildren<Renderer>().sharedMaterial.name ==
9             "wheat")
10        {
11            wheat++;
12        }
13        else if (tile.GetComponentInChildren<Renderer>().sharedMaterial.name
14            == "sheep")
15        {
16            sheep++;
17        }
18        else if (tile.GetComponentInChildren<Renderer>().sharedMaterial.name
19            == "rock")
20        {
21            stone++;
22        }
23        else if (tile.GetComponentInChildren<Renderer>().sharedMaterial.name
24            == "brick")
25        {
26            brick++;
27        }
28    }
29    UpdateResources();
30 }

```

---

Listing A.3: PlayerScript-Methode zum Sammeln von Ressourcen für eine bestimmte Siedlung

---

```

1 public String calculatePreferencePlayer1() {
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 (wheat == 2 && stone == 2) {
11            preference = "rock";
12        } else if (stone == 3 && wheat == 1) {
13            preference = "wheat";
14        }
15    } else {
16        preference = "";
17    }
18    this.preference = preference;
19    return preference;
20 }

```

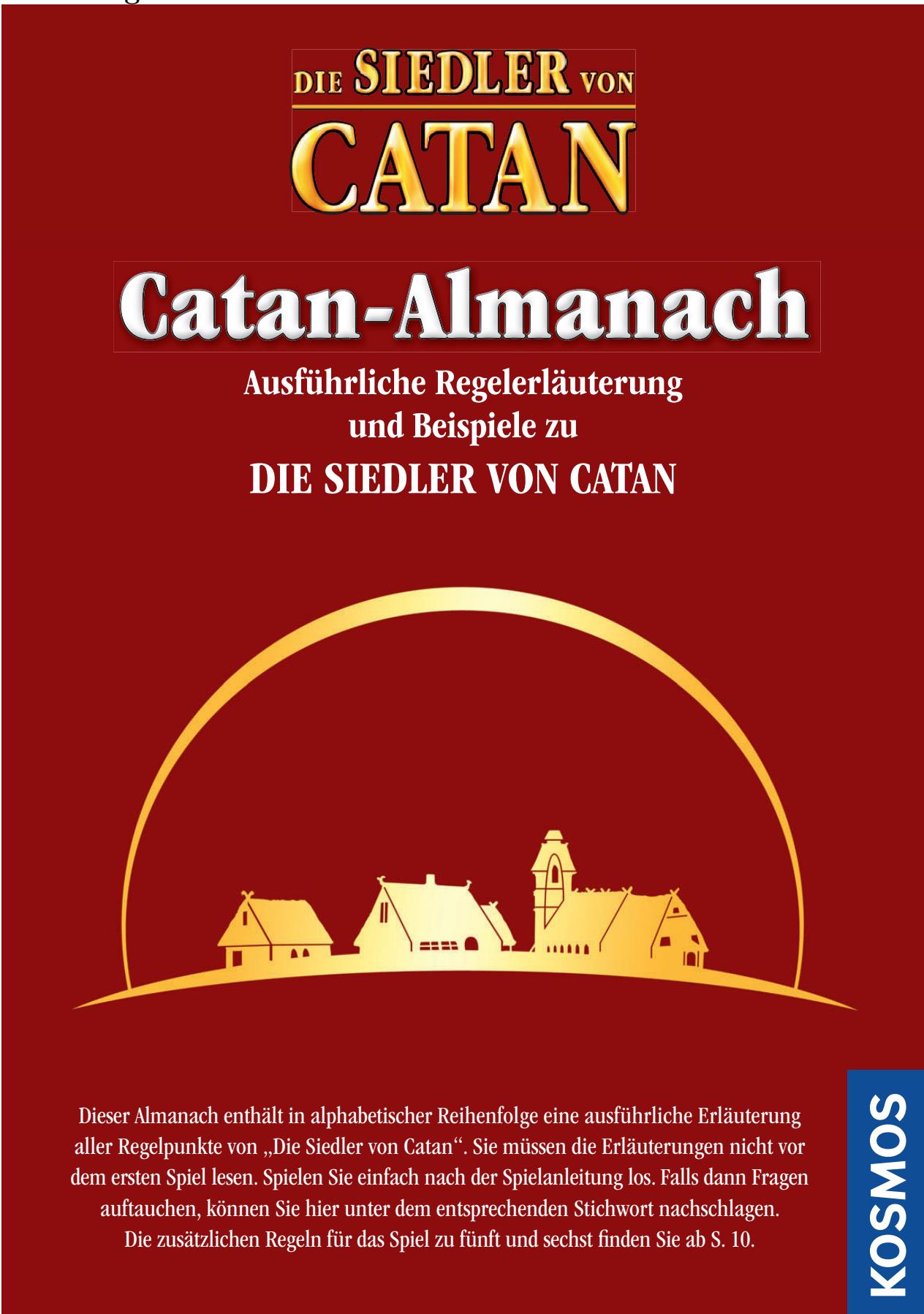
---

Listing A.4: Methode des Status zum Berechnen der Präferenz von Spieler 1

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

---

## A.2 Regelwerk: Siedler von Catan



Dieser Almanach enthält in alphabetischer Reihenfolge eine ausführliche Erläuterung aller Regelpunkte von „Die Siedler von Catan“. Sie müssen die Erläuterungen nicht vor dem ersten Spiel lesen. Spielen Sie einfach nach der Spielanleitung los. Falls dann Fragen auftauchen, können Sie hier unter dem entsprechenden Stichwort nachschlagen.  
Die zusätzlichen Regeln für das Spiel zu fünf und sechst finden Sie ab S. 10.

KOSMOS

## Spielmaterial für 3 – 4 Spieler

### 19 Sechseckfelder mit Landschaften

Wald (4)  
 Weideland (4)  
 Ackerland (4)  
 Hügelland (3)  
 Gebirge (3)  
 Wüste (1)  
 9 Meerfelder ohne Hafen  
 9 Meerfelder mit Hafen



### 120 Rohstoffkarten (je 24)

Holz	= Baumstämme	= aus Wald
Wolle	= Schaf	= aus Weideland
Getreide	= Ährenbündel	= aus Ackerland
Lehm	= Bausteine	= aus Hügelland
Erz	= Erzgestein	= aus Gebirge



### 25 Entwicklungskarten

Ritter (14)  
Fortschritt (6)  
Siegpunkte (5)



### 4 Karten „Baukosten“



### 2 Sonderkarten



Längste Handelsstraße

Größte Rittermacht

### 2 Kartenhalter

### 96 Spielfiguren (in vier Farben)

16 Städte  
20 Siedlungen  
60 Straßen



### 1 Räuber-Figur (grau)



### 18 Zahlenchips



### 2 Würfel

## Ausführliche Regelerläuterung und Beispiele zu „Die Siedler von Catan“

Aufbau, Variabler  
Bauen  
Binnenhandel  
Entwicklungskarten  
Fortschrittskarten  
Gründungsphase  
Hafenstandort  
Handel  
Handeln und Bauen – Trennung aufgehoben  
Kreuzung  
Küste  
Längste Handelsstraße  
Räuber  
Ritter  
Seehandel  
Sieben gewürfelt – Räuber wird aktiv  
Siedlung  
Siegpunkte  
Siegpunktkarten  
Spielernde  
Stadt  
Straße  
Taktik  
Wege  
Wüste  
Zahlenchips

### Die Siedler von Catan sind mehr als nur ein Spiel!

Möchten Sie mehr über die Welt der Siedler von Catan erfahren? Noch mehr Spieltiefe und Spielspaß erleben mit den Erweiterungen „Seefahrer“, „Händler & Barbaren“ und „Städte & Ritter“?

Lassen Sie sich auf [www.catan.de](http://www.catan.de) überraschen, was Catan noch für Sie bereit hält!

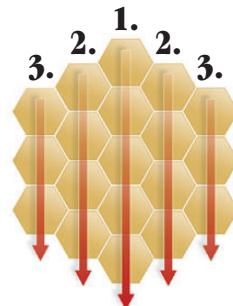


**A**

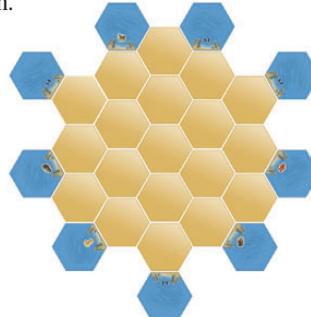
### Aufbau, Variabler

- Trennen Sie die Landfelder von den Meerfeldern.
- Mischen Sie die verdeckten Landfelder. Vom verdeckten Stapel nehmen Sie nacheinander das oberste Sechseck und legen es wie nachfolgend beschrieben auf den Tisch.

1. Legen Sie 5 Landteile untereinander in die Tischmitte.
2. Schließen Sie rechts und links je 1 Reihe zu je 4 Landfeldern an.
3. Daran schließen sich rechts und links je 3 Landfelder an.

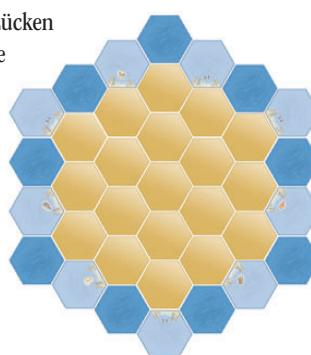


4. Plazieren Sie jetzt rundherum die Meerfelder mit den Hafenstandorten, lassen Sie aber dazwischen immer ein Feld frei. Es spielt keine Rolle, in welcher Reihenfolge die Hafenstandorte liegen.

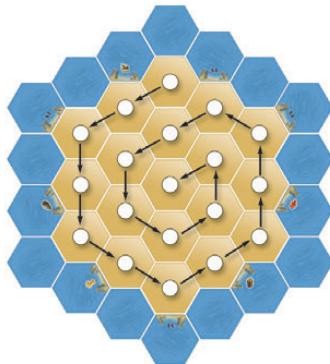


**Wichtig:** Die Sechsecke müssen so an ein Landfeld angelegt werden, daß beide Häfen an Land grenzen.

5. Füllen Sie dann die Lücken mit Meerfeldern ohne Hafenstandorte.



6.



Nach dem „Bauen“ ist der Zug des Spielers beendet, sein linker Nachbar setzt das Spiel fort. Regelvariante siehe unter „Handeln und Bauen – Trennung aufgehoben“.

### Binnenhandel (Handel mit Mitspielern)

Der Spieler, der an der Reihe ist, darf (nach dem Auswürfeln der Rohstofferträge) mit seinen Mitspielern Rohstoffkarten tauschen. Die Tauschbedingungen – wie viele Karten wofür – sind dem Verhandlungsgeschick der Spieler überlassen. Das Verschenken von Karten ist nicht erlaubt (Tausch von 0 gegen 1 oder mehr Karten).

**Wichtig:** Es darf immer nur mit dem Spieler getauscht werden, der an der Reihe ist. Die anderen Spieler dürfen untereinander nicht tauschen.

#### Beispiel:

Hans ist an der Reihe. Er benötigt zum Bau einer Straße 1 Lehm. Er selbst besitzt 2 Holz und 3 Erz. Hans fragt laut: „Wer gibt mir 1 Lehm, ich biete 1 Erz.“ Werner antwortet: „Wenn Du mir 3 Erz gibst, erhältst Du 1 Lehm.“ Gabi ruft dazwischen: „Du bekommst 1 Lehm, wenn Du mir 1 Holz und 1 Erz gibst.“ Hans entscheidet sich für Gabis Angebot und tauscht 1 Holz und 1 Erz gegen 1 Lehm.

**Wichtig:** Werner hätte nicht mit Gabi tauschen dürfen, da Hans am Zug war.

### E

### Entwicklungskarten

Es gibt drei verschiedene Arten von Entwicklungskarten: Ritter (→), Fortschritt (→) und Siegpunkte (→). Wer eine Entwicklungskarte kauft, nimmt die oberste Karte vom verdeckten Stapel auf die Hand. Jeder Spieler hält seine Entwicklungskarten bis zu ihrem Einsatz geheim.

Ein Spieler, der an der Reihe ist, darf in seinem Zug immer nur 1 Karte ausspielen: Entweder 1 Ritter- oder 1 Fortschrittskarte. Der Zeitpunkt des Ausspielens ist beliebig und auch **vor dem Würfeln** möglich; es darf aber keine Karte ausgespielt werden, die erst in diesem Zug gekauft wurde. **Ausnahme:** Kauft ein Spieler eine Karte und es ist eine Siegpunktkarte (→), mit der er seinen 10. Siegpunkt erreicht, so darf er diese Karte sofort aufdecken und hat das Spiel gewonnen.

Siegpunktkarten (eine oder mehrere) werden erst am Ende des Spiels aufgedeckt, wenn ein Spieler mit ihnen 10 Siegpunkte erreicht und damit das Spiel beendet.

**Hinweis:** Wird ein Spieler beraubt (siehe „7“ gewürfelt – Räuber wird aktiv), so dürfen nur Rohstoffkarten aus seiner Hand geraubt werden. Entwicklungskarten sollte er vorher weglegen oder anderweitig aufbewahren.

### B

### Bauen

Nachdem der Spieler, der an der Reihe ist, die Rohstofferträge ausgewürfelt und Handel betrieben hat, darf er bauen. Er muss dazu bestimmte Kombinationen von Rohstoffkarten (siehe Karte Baukosten) abgeben – auf die Vorratsstapel zurücklegen. Der Spieler kann beliebig viele Bauwerke errichten und Entwicklungskarten kaufen, so lange er Rohstoffe zum „bezahlen“ hat und so lange der Vorrat an Entwicklungskarten und Bauwerken reicht. Siehe auch Siedlung (→), Stadt (→), Straße (→), Entwicklungskarten (→).

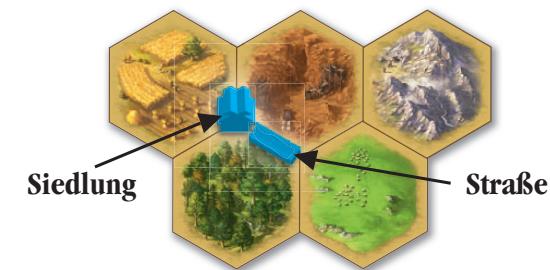
Jeder Spieler verfügt über 15 Straßen, 5 Siedlungen und 4 Städte. Errichtet ein Spieler eine Stadt, darf er die dadurch frei gewordene Siedlung wieder verbauen. Straßen und Städte dagegen, sind sie einmal errichtet, bleiben bis zum Ende des Spiels auf ihren Plätzen.

## F

### Fortschrittskarten

Fortschrittskarten sind eine Gruppe der Entwicklungskarten. In seinem Zug darf ein Spieler nur eine Entwicklungskarte ausspielen. Es gibt je zweimal:

- Straßenbau: Wer diese Karte ausspielt, darf ohne Rohstoff-Kosten 2 neue Straßen auf den Spielplan legen. Hierbei müssen die üblichen Regeln für den Bau von Straßen beachtet werden.
- Erfindung: Wer diese Karte ausspielt, darf sich 2 beliebige Rohstoffkarten von den Vorratsstapeln nehmen. Hat der Spieler die Bauphase noch vor sich, darf er diese Rohstoff-karte(n) zum Bauen verwenden.
- Monopol: Wer diese Karte ausspielt, wählt einen Rohstoff aus. Alle Mitspieler müssen ihm alle Rohstoffkarten geben, die sie von dieser Sorte auf der Hand halten. Wer keine solche Rohstoffkarte besitzt, muss auch nichts abgeben.



### 2. Runde

Haben alle Spieler ihre erste Siedlung errichtet, so startet der Spieler, der zuletzt an der Reihe war, die zweite Runde: Er darf jetzt zuerst seine zweite Siedlung gründen und seine zweite Straße anschließen. **Achtung:** Nach ihm folgen die anderen Spieler entgegen dem Uhrzeigersinn, der Startspieler ist also als Letzter mit seiner zweiten Siedlung dran.

Die zweite Siedlung kann völlig unabhängig von der ersten auf eine beliebige Kreuzung gesetzt werden, wobei auch dort die Abstandsregel beachtet werden muss. Die zweite Straße muss an die zweite Siedlung anschließen, aber wiederum in beliebiger Richtung.

Jeder Spieler erhält sofort nach der Gründung seiner zweiten Siedlung seine ersten Rohstofferträge: Für jedes Landfeld, das an diese zweite Siedlung angrenzt, nimmt er sich eine entsprechende Rohstoffkarte vom Vorrat.

Der Startspieler (der als Letzter seine zweite Siedlung gegründet hat) beginnt das Spiel: Er würfelt mit beiden Würfeln die Rohstofferträge des ersten Zuges aus. Hilfreiche Tipps über das Vorgehen bei der Gründung finden Sie unter „Taktik“.

## G

### Gründungsphase

Die „Gründungsphase“ beginnt, nachdem das variable Spielfeld (→) aufgebaut ist.

- Jeder Spieler wählt eine Farbe und erhält die entsprechenden Spielfiguren: 5 Siedlungen, 4 Städte, 15 Straßen. Dazu eine Übersichtskarte Baukosten.
- Die Rohstoffkarten werden in fünf Stapel sortiert und offen in die beiden Kartenhalter gelegt. Die Kartenhalter werden neben dem Spielfeld bereit gestellt.
- Die Entwicklungskarten (→) werden gemischt und als verdeckter Stapel in das freie Fach des Kartenhalters gelegt.
- Die beiden Sonderkarten und die Würfel werden griffbereit neben das Spielfeld gelegt.
- Der Räuber wird auf die Wüste gesetzt.

Die Gründungsphase erstreckt sich über zwei Runden, in der jeder Spieler 2 Straßen und 2 Siedlungen baut:

#### 1. Runde

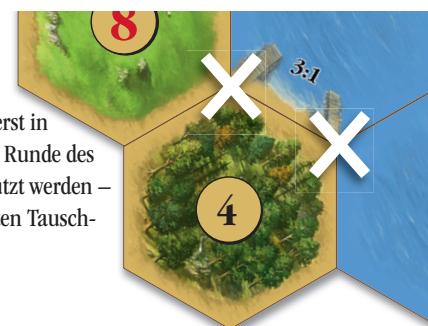
Reihum würfeln alle Spieler mit beiden Würfeln; wer die höchste Augenzahl erreicht, beginnt. Der Spieler setzt eine seiner Siedlungen auf eine freie Kreuzung (→) seiner Wahl. An diese Siedlung schließt er in beliebiger Richtung eine seiner Straßen an. Danach folgen die anderen Spieler im Uhrzeigersinn: Jeder setzt 1 Siedlung und 1 Straße.

**Achtung:** Beim Setzen aller weiteren Siedlungen muss die Abstandsregel beachtet werden!

## H

### Hafenstandort

Häfen haben den Vorteil, dass man Rohstoffe günstiger tauschen kann. Um in den Besitz eines Hafens zu kommen, muss ein Spieler eine Siedlung an der Küste (→) bauen – auf einer der beiden Kreuzungen (→), die zu einem Hafen gehören. Siehe auch „Seehandel“ (→).



## Handel

Nachdem der Spieler, der an der Reihe ist, die Rohstofferträge für diesen Zug ausgewürfelt hat, darf er Handel treiben. Er darf mit seinen Mitspielern Rohstoffkarten tauschen (Binnenhandel (→)), aber auch ohne Mitspieler tauschen (Seehandel (→)), indem er eigene Rohstoffkarten gegen Karten aus den Vorratsstapeln umtauscht. Der Spieler, der an der Reihe ist, darf so lange und so oft tauschen, wie es seine Rohstoffkarten (auf der Hand) zulassen.

## Handeln und Bauen – Trennung aufgehoben

Die Trennung von Handels- und Bauphase wurde eingeführt, um Neulingen durch klare Strukturen einen schnellen Einstieg ins Spiel zu ermöglichen. Erfahrenen Spielern empfehlen wir jedoch, die Trennung zwischen Handels- und Bauphase aufzuheben. Somit kann man nach dem Auswürfeln der Rohstoffe in beliebiger Reihenfolge handeln und bauen. Natürlich kann man zum Beispiel auch handeln, bauen, dann weiterhandeln und erneut bauen – so lange das die eigenen Handkarten zulassen.

Wenn die Trennung aufgehoben ist, darf ein Spieler, der eine Siedlung an einem Hafenstandort gebaut hat, den Hafen noch im gleichen Zug zum Tauschen nutzen.

## L

### Längste Handelsstraße

- Eine Handelsstraße kann für die Zählung des längsten Straßenzugs unterbrochen werden, wenn ein anderer Spieler eine Siedlung auf einer freien Kreuzung der Handelsstraße errichtet!



*Beispiel: Orange hatte die „Längste Handelsstraße“ mit 7 Straßen. Rot baute die mit einem schwarzen Kreis markierte Siedlung, unterbrach damit die Handelsstraße von Orange und besitzt nun selbst die „Längste Handelsstraße“ und die 2 Siegpunkte.*

**Bitte beachten:** Eigene Siedlungen/Städte unterbrechen die Handelsstraße nicht!

- Haben, nach der Unterbrechung einer Handelsstraße, mehrere Spieler gleich lange Handelsstraßen, so wird die Sonderkarte weggelegt. Sie kommt erst wieder ins Spiel, wenn ein Spieler allein die „Längste Handelsstraße“ besitzt. Die Sonderkarte wird auch weggelegt, wenn kein Spieler mehr eine „Längste Handelsstraße“ besitzt.

## R

### Räuber

Der Räuber steht zu Beginn des Spieles auf der Wüste (→). Er wird nur bewegt, wenn eine „7“ gewürfelt (→) wird oder wenn ein Spieler eine Ritterkarte (→) aufdeckt. Wurde der Räuber auf ein Landfeld versetzt, so verhindert er die Rohstoffproduktion dieses Feldes, so lange er darauf steht. Alle Spieler, die an diesem Landfeld Siedlungen und/oder Städte besitzen, erhalten von diesem Feld keine Rohstoffe, so lange der Räuber auf diesem Feld steht.

## K

### Kreuzung

Kreuzungen sind die Schnittpunkte, an denen sich drei Felder treffen. Nur auf Kreuzungen dürfen Siedlungen gegründet werden.



### Küste

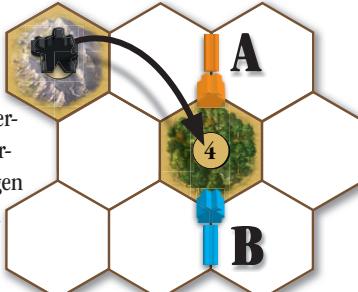
Grenzt ein Landfeld an ein Meerfeld, so spricht man von einer „Küste“. Entlang einer Küste kann eine Straße gebaut werden. Auf Kreuzungen, die an Meerfelder grenzen, können Siedlungen gegründet und zu Städten ausgebaut werden. Der Nachteil: An Küstenfeldern erhält man nur von einem oder von zwei Landschaftsfeldern Rohstofferträge. Der Vorteil: An Küsten gibt es Hafenstandorte, die es über den Seehandel erlauben, Rohstoffe günstiger zu tauschen. Allerdings: Siedlungen auf Küstenkreuzungen ohne Hafen bringen keine Tauschvorteile ein.

*Beispiel (Abb. siehe unten bei „Ritter“): Hans ist an der Reihe und hat eine „7“ gewürfelt. Er muss nun den Räuber versetzen. Der Räuber stand auf einem Gebirgsfeld. Hans setzt ihn auf den Zahlenchip „4“ eines Waldfeldes. Hans darf sich von einem der beiden Spieler, denen die Siedlungen „A“ und „B“ gehören, eine Rohstoffkarte aus dessen verdeckter Hand ziehen. Außerdem gilt: Sollte in den nächsten Runden eine „4“ gewürfelt werden, erhalten die*

*Besitzer der Siedlungen „A“ und „B“ keine Holz-Rohstoffkarte. Dies gilt so lange, bis der Räuber durch eine weitere „7“ oder eine Ritterkarte versetzt wird.*

## Ritter

Deckt ein Spieler während seines Zuges eine Entwicklungskarte „Ritter“ auf (das kann auch vor dem Würfeln sein), muss er sofort den Räuber ( $\rightarrow$ ) versetzen.

- Der Spieler, der die Ritterkarte gespielt hat, muss den Räuber nehmen und ihn auf ein beliebiges anderes Landfeld versetzen.
  - Dann darf er demjenigen Spieler 1 Rohstoffkarte rauben (und behalten), der eine Siedlung oder Stadt an diesem Landfeld stehen hat. Haben dort zwei oder mehr Spieler Gebäude stehen, darf er sich aussuchen, wen er berauben will.
  - Der Spieler, der beraubt wird, bekommt die Karte aus der verdeckten Hand gezogen.
  - Wer zuerst 3 Ritterkarten offen vor sich liegen hat, erhält die Sonderkarte „Größte Rittermacht“, die 2 Siegpunkte wert ist.
  - Sobald ein anderer Spieler eine Ritterkarte mehr aufdeckt, so erhält er diese Sonderkarte vom derzeitigen Besitzer; die beiden Siegpunkte wechseln mit.
- 

**Beispiel:** Hans ist an der Reihe und deckt eine Ritterkarte auf. Er versetzt den Räuber von dem Gebirgsfeld auf das Waldfeld mit der „4“. Hans darf nun entweder von Spieler A oder B eine Rohstoffkarte aus dessen verdeckten Handkarten ziehen.

**Wichtig:** Wenn eine Ritterkarte ausgespielt wird, wird nicht geprüft ob ein Spieler mehr als 7 Karten auf der Hand hält. Karten muss man nur abgeben, wenn eine 7 gewürfelt wird und man mehr als 7 Karten auf der Hand hält.

## S

## Seehandel

Ist ein Spieler an der Reihe, so kann er in der Handelsphase auch ohne Mitspieler Rohstoffkarten tauschen: Der Seehandel ermöglicht ihm das.

- Ohne Hafen:** Bei der einfachsten (und ungünstigsten) Tauschvariante 4:1 legt der Spieler 4 gleiche Rohstoffkarten

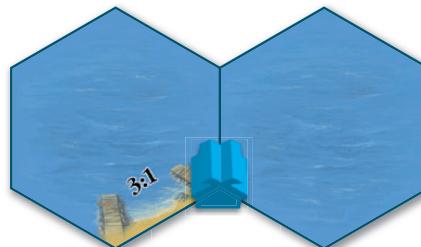
zurück auf die Vorratsstapel und nimmt sich dafür die gewünschte Karte (eine) vom entsprechenden Stapel.

Der Spieler benötigt für den Tausch 4:1 keinen Hafen ( $\rightarrow$ ) (Siedlung an einem Hafenstandort).

**Beispiel:** Hans legt 4 Karten Erz auf den Vorratsstapel zurück und nimmt sich 1 Holz-Karte. Sinnvoller wäre es natürlich, zuerst einen günstigeren Tausch mit den Mitspielern zu versuchen (Binnenhandel).

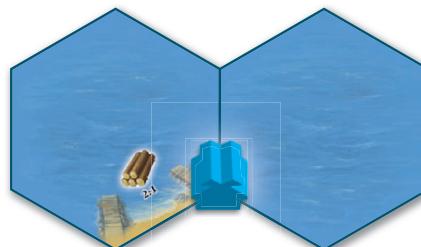
- Mit Hafen:** Bessere Tauschmöglichkeiten hat ein Spieler, wenn er seine Siedlung oder Stadt an einem Hafenstandort ( $\rightarrow$ ) errichtet hat. Es gibt zwei verschiedene Arten von Hafenstandorten:

**1. Einfacher Hafen (3:1):** Der Spieler, der an der Reihe ist, darf während seiner Handelsphase 3 gleiche Rohstoffkarten ablegen und sich dafür 1 andere beliebige Rohstoffkarte nehmen.



**Beispiel:** Spieler Blau legt 3 Holz-Karten auf den Vorratsstapel Holz und nimmt sich 1 Karte Erz.

**2. Spezialhafen (2:1):** Für jede Rohstoffart gibt es einen Spezialhafen. Die günstige Tauschmöglichkeit 2:1 gilt immer nur für den Rohstoff, der auf dem Spezial-Hafenfeld abgebildet ist. Bitte beachten: Ein Spezialhafen berechtigt nicht zum Tausch der anderen Rohstoffarten im Verhältnis 3:1!



**Beispiel:** Spieler Blau hat eine Siedlung (oder Stadt) an dem Spezialhafen für Holz errichtet. Der Spieler darf beim Tausch 2 Karten Holz auf den Vorratsstapel zurücklegen und sich dafür 1 beliebige, andere Rohstoffkarte nehmen. Er kann auch 4 Karten Holz gegen 2 andere Karten tauschen usw.

**Wichtig:** Nur der Spieler, der an der Reihe ist, darf den Seehandel durchführen!

## Sieben gewürfelt – Räuber wird aktiv

Würfelt ein Spieler in seiner Phase Rohstofferträge eine „7“, so erhält kein Spieler Erträge. Im Gegenteil:

- Alle Spieler zählen die Rohstoffkarten in ihrem Besitz. Wer mehr als 7 Rohstoffkarten hat (also 8, 9 und mehr), muss die Hälfte davon auswählen und ablegen – zurück auf die Vorratsstapel. Bei einer ungeraden Anzahl an Karten wird immer zum Vorteil des betroffenen Spielers abgerundet: wer z. B. 9 Rohstoffkarten besitzt, muss 4 davon ablegen.
- Beispiel:** Hans würfelt eine „7“. Er hat nur 6 Rohstoffkarten auf der Hand. Benni hat 8 Karten und Wolfgang 11. Benni muss 4 Karten ablegen und Wolfgang 5 (jeweils zu ihrem Vorteil abgerundet).
- Dann nimmt der Spieler, der an der Reihe ist, den Räuber (→) und versetzt ihn auf ein beliebiges anderes Landfeld. Damit sind die Rohstoffeinnahmen dieses Feldes blockiert. Außerdem darf der Spieler, der den Räuber versetzt hat, dem Spieler 1 Rohstoffkarte aus der verdeckten Hand rauben, der eine Siedlung oder Stadt an diesem Landfeld besitzt; gibt es dort 2 oder mehr Spieler mit Gebäuden, darf er sich einen davon aussuchen und berauben. Siehe auch Ritter (→).

Danach setzt der Spieler seinen Zug mit seiner Handelsphase fort.

## Siedlung

Eine Siedlung zählt 1 Siegpunkt und ihr Besitzer kann für alle angrenzenden Landfelder Rohstofferträge erhalten.

**Wichtig:** Beim Bau einer Siedlung muss die Abstandsregel beachtet werden – auf keiner der drei angrenzenden Kreuzungen darf bereits eine Siedlung (egal welchen Spielers) stehen. Hat ein Spieler seine 5 Siedlungen verbaut, so muss er zuerst eine seiner Siedlungen zu einer Stadt ausbauen. Er nimmt seine Siedlung zurück und ersetzt sie durch eine Stadt. Der Spieler kann jetzt eine neue Siedlung gründen.

## Siegpunkte

Wer zuerst 10 Siegpunkte erreicht und an der Reihe ist, gewinnt das Spiel. Siegpunkte erhält ein Spieler für:

Eine Siedlung	1 Siegpunkt
Eine Stadt	2 Siegpunkte
Längste Handelsstraße	2 Siegpunkte
Größte Rittermacht	2 Siegpunkte
Entwicklungs-karte Siegpunkt:	1 Siegpunkt

Jeder Spieler beginnt mit 2 Siedlungen, hat also schon von Anfang an 2 Siegpunkte. Es gilt also, noch 8 Siegpunkte dazu zu gewinnen.

## Siegpunktkarten

Siegpunktkarten gehören zu den Entwicklungskarten (→), sie können also „gekauft“ werden. Diese Entwicklungskarten stellen wichtige kulturelle Errungenschaften dar, die sich in bestimmten Gebäuden widerspiegeln. Jede Siegpunktkarte zählt 1 Siegpunkt. Hat ein Spieler eine Siegpunktkarte gekauft, bewahrt er sie verdeckt auf. Wer an der Reihe ist und zusammen mit seinen Siegpunktkarten 10 Punkte erreicht hat, deckt die Karte auf (auch mehrere) und hat damit gewonnen.

**Tipp:** Siegpunktkarten sollten so aufbewahrt werden, dass die Mitspieler keine Schlüsse daraus ziehen können. Wer vor sich immer 1 oder 2 verdeckte Karten liegen hat und diese nicht einsetzt, nährt den Verdacht, dass es Siegpunkte sein könnten.

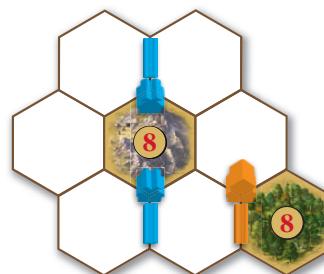
## Spielende

Ist ein Spieler an der Reihe und hat er 10 Siegpunkte erreicht (oder erreicht er sie in diesem Zug), so beendet er sofort das Spiel und gewinnt.

**Beispiel:** Ein Spieler hat 2 Siedlungen (2 SP), die Sonderkarte Längste Handelsstraße (2 SP), 2 Städte (4 SP) und 2 Siegpunktkarten (2 SP). Er deckt seine beiden Siegpunktkarten auf und besitzt damit die erforderlichen 10 Punkte zum Sieg.

## Stadt

Nur eine bestehende Siedlung kann zu einer Stadt ausgebaut werden. Jede Stadt zählt 2 Siegpunkte und ihr Besitzer erhält für jedes angrenzende Landfeld doppelte Rohstofferträge (2 Rohstoffkarten), falls die Zahl des Feldes gewürfelt wird. Frei gewordene Siedlungen (durch Städtebau) können erneut zur Gründung von Siedlungen eingesetzt werden.



**Beispiel:** Es wurde die „8“ gewürfelt. Spieler Blau erhält 3 Karten Erz: für die Siedlung 1 Erz und für die Stadt 2 Erz. Spieler Orange erhält für seine Stadt 2 Holz.

**Tipp:** Ohne den Ausbau von Siedlungen zu Städten (2 Siegpunkte jeweils) ist das Spiel kaum zu gewinnen. Da jeder Spieler nur 5 Siedlungen zur Verfügung hat, kann er alleine mit den Siedlungen nur 5 Siegpunkte erreichen.

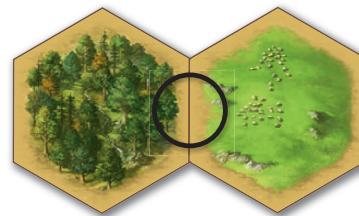
## Straße

Straßen sind Verbindungen zwischen eigenen Siedlungen oder Städten. Straßen werden auf Wegen (→) errichtet. Auf jedem Weg (auch an der Küste entlang) darf immer nur eine Straße gebaut werden. Eine Straße wird entweder an eine Kreuzung angelegt auf der eine eigene Siedlung oder Stadt steht oder sie wird an einer unbesetzten (freien) Kreuzung angelegt, an die eine eigene Straße grenzt. Ohne den Bau neuer Straßen können auch keine neuen Siedlungen errichtet werden. Straßen alleine verhelfen nur in einem Fall zu Siegpunkten – wenn man die Sonderkarte „Längste Handelstraße“ (→) besitzt.

## W

### Wege

Als Wege bezeichnet man die Kanten, an denen zwei Felder aneinander stoßen. Wege verlaufen also auf der Grenze zwischen Landfeldern bzw. zwischen Land- und Meerfeldern. Auf jedem Weg kann nur eine Straße (→) gebaut werden. Wege münden immer in eine Kreuzung (→) – das ist der Punkt, an dem drei Felder zusammenstoßen.



## T

### Taktik

Da „Die Siedler von Catan“ auf einem variablen Spielplan gespielt wird, sind die taktischen Überlegungen bei jedem Spiel anders. Dennoch gibt es ein paar allgemeine Punkte, die jeder Spieler beachten sollte. Hier die wichtigsten Punkte:

1. Lehm und Holz sind zu Beginn des Spieles die wichtigsten Rohstoffe. Beide benötigt man für Straßen und Siedlungen. Man sollte am Anfang seine Siedlungen zumindest an ein gutes Lehm- oder Holzfeld grenzen lassen.
2. Der Wert der Hafenstandorte sollte nicht unterschätzt werden. Wer z. B. Siedlungen oder Städte an guten Getreidefeldern besitzt, sollte im Lauf des Spieles eine Siedlung am Hafenstandort „Getreide“ gründen.
3. Man sollte bei der Gründung der ersten beiden Siedlungen darauf achten, genügend Hinterland zu besitzen, damit man sich weiter ausbreiten kann. Die Gründung beider Siedlungen in der Inselmitte ist gefährlich. Schnell sind die Wege von Mitspielern blockiert.
4. Wer viel handelt, hat bessere Chancen. Ruhig auch mal dem Spieler, der an der Reihe ist, von sich aus ein Angebot machen!

## Z

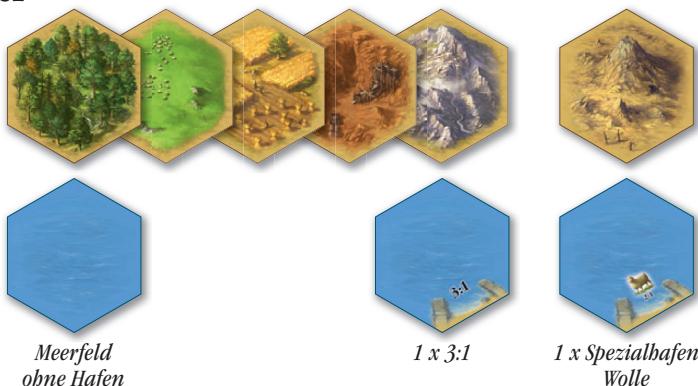
### Zahlenchips

Die Größe der Zahlen auf diesen Chips zeigt an, mit welcher Wahrscheinlichkeit das zugehörige Landfeld Rohstoffe abwirft. Je größer die Zahlen abgebildet sind, umso größer ist die Wahrscheinlichkeit, dass sie gewürfelt werden. So ist ein Feld mit einer „6“ oder „8“ wesentlich ertragreicher als ein Feld mit einer „2“ oder „12“.

## Spielmaterial für 5 – 6 Spieler

### 15 Sechseckfelder mit Landschaften

Wald (2)  
 Weideland (2)  
 Ackerland (2)  
 Hügelland (2)  
 Gebirge (2)  
 Wüste (1)  
 2 Meerfelder ohne Hafen  
 2 Meerfelder mit Hafen



Meerfeld  
ohne Hafen

1 x 3:1

1 x Spezialhafen  
Wolle

### 9 Entwicklungskarten

Ritter (6)  
Fortschritt (3)



### 2 Karten „Baukosten“



### 48 Spielfiguren (in zwei Farben)

8 Städte  
10 Siedlungen  
30 Straßen



### 10 Zahlenchips



Alle 28 Zahlenchips, die für das Spiel zu fünf oder zu sechs benötigt werden:

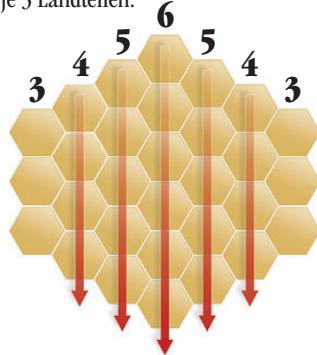


## Regeländerungen für das Spiel mit 5 und 6 Spielern

Für das Spiel zu fünft und zu sechst benötigen Sie alle Sechseckfelder und alle Zahlenchips sowie die neun, im Spiel zu dritt und viert aussortierten, Entwicklungskarten (6 Ritter, je 1 x Fortschritt: Straßenbau, Fortschritt: Monopol, Fortschritt: Erfindung).

### Variabler Aufbau des Spielfeldes

- Die Insel wird nun aus insgesamt 30 Landteilen zusammen gesetzt.
- Mischen Sie alle verdeckten Landteile und bilden Sie daraus einen verdeckten Stapel. Nehmen Sie dann immer das oberste verdeckte Landfeld und legen es offen nach folgendem Schema aus: 6 Landteile nebeneinander in die Mitte; schließen Sie darüber und darunter jeweils 1 Reihe zu je 5 Landteilen an. Daran schließen oben und unten zwei Reihen zu je 4 Landteilen an. Den Abschluss bilden zwei Reihen zu je 3 Landteilen.



- Platzieren Sie jetzt rundherum die Meerfelder, wie im Spiel zu dritt und viert.
- Legen Sie die Zahlenchips neben dem Spielplan bereit. Sortieren Sie die Zahlenchips in der auf S. 10 unter der Auflistung des Spielmaterials angegebenen Reihenfolge. Legen Sie jetzt die Chips in dieser Reihenfolge auf den Spielplan. Beginnen Sie auf einem beliebigen Eckfeld mit der 2 und setzen Sie die nächsten Chips entgegen dem Uhrzeigersinn, spiralförmig zum Zentrum hin, ab. (Siehe Beispiel auf S. 3.) Die beiden Wüsten bleiben ohne Chips.
- Natürlich können Sie die Zahlenchips auch frei verteilen. Achten Sie bei einer freien Verteilung aber darauf, dass keine roten Zahlen nebeneinander liegen.
- Der Räuber startet beliebig auf einer der beiden Wüsten.

### Impressum

© 1995, 2010 Kosmos Verlag  
Autor: Klaus Teuber, [www.klausteuber.de](http://www.klausteuber.de)  
Lizenz: Catan GmbH, [www.catan.com](http://www.catan.com)  
Grafik: Michaela Schelk/Fine Tuning  
Illustration: Michael Menzel  
Redaktion: Reiner Müller, Sebastian Rapp

Regelstand April 2010

### Der Spielablauf

Gespielt wird nach den gleichen Regeln wie im Spiel für 3 – 4 Spieler – mit einer Ausnahme!

Wie gewohnt erledigt der Spieler, der an der Reihe ist (Spieler A), zunächst seine drei Aktionen:

- 1) Er würfelt die Rohstofferträge aus.
- 2) Er kann handeln.
- 3) Er kann bauen.

Jetzt die Ausnahme:

Hat Spieler A seinen Zug beendet, beginnt eine „außerordentliche Bauphase“: Alle anderen Spieler dürfen nun reihum ebenfalls bauen und Entwicklungskarten kaufen.

*In der Praxis sieht das so aus:*

Spieler A hat gebaut. Bevor er den Würfel an seinen linken Nachbarn weitergibt, der ja als nächster an der Reihe wäre, fragt er seine Mitspieler, ob noch jemand bauen möchte. Meldet sich ein Spieler, darf dieser sofort bauen und/oder Entwicklungskarten kaufen. Melden sich mehrere Spieler, ist die Sitzreihenfolge entscheidend: Wer Spieler A im Uhrzeigersinn am nächsten sitzt, beginnt, die anderen Bauwilligen folgen ebenfalls im Uhrzeigersinn. Danach ist wie gewohnt der nächste Spieler an der Reihe.

**Wichtig:** Während der „außerordentlichen Bauphase“ dürfen Spieler nur Siedlungen, Straßen, Städte bauen und/oder Entwicklungskarten kaufen. Sie dürfen keine Karten ausspielen oder tauschen, jede Form von Handel (also Binnen- und Seehandel) ist in dieser Phase verboten.

**Tipp:** Durch die Einführung der „außerordentlichen Bauphase“ haben Sie die Möglichkeit, das Spiel zu beeinflussen, auch wenn Sie nicht an der Reihe sind. Deshalb sollten Sie, so lange Mitspieler in ihrer Handelsphase sind, mit ihnen handeln, was das Zeug hält. Da Sie nach jeder Runde eines Spielers bauen können, wenn Sie die entsprechenden Rohstoffkarten besitzen, können Sie anderen Spielern vorkommen und natürlich auch dem stets drohenden Räuber ein Schnippchen schlagen.

## 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.