Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Andre Goldflam

Eine domänenspezifische Sprache zur Implementierung komplexer Abläufe in Gesellschaftsspielen

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Andre Goldflam

Eine domänenspezifische Sprache zur Implementierung komplexer Abläufe in Gesellschaftsspielen

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Masterstudiengang Informatik
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Kai von Luck
Zweitgutachter : Gunter Klemke

Abgegeben am 4. Juli 2012

Andre Goldflam

Thema der Masterarbeit

Eine domänenspezifische Sprache zur Implementierung komplexer Abläufe in Gesellschaftsspielen

Stichworte

Domänenspezifische Sprachen, Computerspiele, Gesellschaftsspiele, Spielmechaniken, Spiel-Engine Architektur

Kurzzusammenfassung

Im Rahmen dieser Arbeit wurde eine Definition zur Beschreibung der Elemente von Spielmechaniken in Gesellschaftsspielen erarbeitet. Auf dieser Grundlage wurde die domänenspezifische Sprache „lGold“ entwickelt. Anhand von lGold können die Abläufe in Gesellschaftsspielen abgebildet werden. Anschließend wurde ein Design für die Entwicklungsumgebung und die Runtime vorgestellt und gezeigt, dass auf diese Weise in lGold definierte Spiele ausgeführt werden können.

Andre Goldflam

Title of the paper

A domain specific language to implement complex processes in board and card games

Keywords

Domain-specific language, digital games, board games, card games, game mechanics, game engine architecture

Abstract

This thesis presents a definition on how to describe game mechanics in board and card games. Based on that definition a domain specific language called “lGold” was developed, which allows for a precise description of board and card games. Finally, an IDE and a runtime design were introduced showing how games defined via lGold can be executed.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

1.1. Zielsetzung

Moderne Computerspiele stellen sehr hohe Anforderungen an ihre Architektur und die unterschiedlichen Komponenten. Sie können durchaus als eine treibende Kraft in der Forschung angesehen werden. Daher ist es nicht verwunderlich, dass der Bereich der Spiele innerhalb der Informatik des Öfteren als experimentelles Umfeld genutzt wird.

Da die Entwicklung eines Spieles ein sehr aufwändiger Prozess sein kann, sind Produktionskosten in Millionenhöhe keine Seltenheit. Auch in dem Markt der „casual Games“ hat sich mittlerweile eine hohe Qualität etabliert und große Publisher dazu bewegt, entsprechend zu investieren. Diese Rahmenbedingungen haben sich auch auf den Entwicklungsprozess der Spiele ausgewirkt. Über unterschiedliche Ansätze aus dem Bereich der agilen Software-Entwicklung, wie z.B. dem Rapid-Prototyping, welches eine hohe Anzahl von Entwicklungszyklen propagiert, wird versucht, möglichst früh auf Komplikationen in Konzept und technischer Umsetzung reagieren zu können. Im Bereich der unabhängigen Spieleentwicklung suggerieren Erfolgsgeschichten wie im Fall von Minecraft [1] ein dem amerikanischen Traum ähnliches Versprechen von einem möglichen Erfolg. In der Realität ist die Entwicklung eines Spieles jedoch mit einem hohen Grad an persönlichem und finanziellem Risiko verbunden. Die Investitionen fließen in ein einziges Produkt und der Erfolg ist von sehr vielen unterschiedlichen Faktoren abhängig. Es ist alles andere als vorausgesetzt, dass der sogenannte „Break Even“ erreicht wird und die Entwicklungskosten wieder einspielt werden können.

Um die Effizienz bei der Produktion zu steigern, werden domänenspezifische Sprachen entwickelt, welche bestimmte Aspekte einer Domäne gezielt abbilden. Die Konzeption und die Entwicklung so einer Sprache sind zum einen ungewohnt und erfordern zum anderen ein großes Maß an interdisziplinärem Wissen. Diese spezialisierten Sprachen und zugehörigen Werkzeuge existieren, sind aber für die verschiedenen Kategorien von Computerspielen unterschiedlich ausgereift. Die meisten solcher Werkzeuge beziehen sich auf den Bereich der 3D-Spiele, die Unreal Engine kann hier als ein Beispiel genannt werden[2].

[1] http://www.minecraft.net/

[2] http://www.unrealengine.com/

Seitens der Forschung wird versucht, den Begriff des Spieles und seiner Elemente möglichst genau zu definieren. Da Spiele in sehr unterschiedlicher Ausprägung existieren und diese wiederum in einzelne Kategorien mit jeweiligen Schwerpunkten zerfallen, ist eine Klassifikation entsprechend problematisch. Diese Schwierigkeiten spiegeln sich auch in dem Bestreben wider, sogenannte „Best Practices“ für die Entwicklung von Computerspielen zu definieren, wobei es dort darum geht, die konzeptionellen und technischen Herausforderungen bei der Umsetzung miteinander zu verknüpfen.

Die Umsetzungen von Gesellschaftsspielen stellt eine kleine Nische mit eigenen Anforderungen dar. Diese bestehen unter anderem darin, dass Gesellschaftsspiele in der Regel mit mehreren menschlichen Spielern gleichzeitig gespielt werden und einen Mehrspieler-Modus voraussetzen. Dieser ist allerdings technisch aufwändig zu implementieren und erhöht den Entwicklungsaufwand wiederum entsprechend. Wahrscheinlich ist es auf diesen Umstand zurückzuführen, dass sich die Anzahl der Adaptionen von Gesellschaftsspielen in Grenzen hält. Für diesen Bereich der Computerspiele sind unterstützende Werkzeuge, vergleichbar mit der Qualität anderer Game-Engines, nicht vorhanden.

Im Rahmen dieser Arbeit sollen also die Themenbereiche der Gesellschaftsspiele und domänenspezifischen Sprachen kombiniert werden und eine Sprache für die Beschreibung von Gesellschaftsspielen vorgestellt werden. Die Sprache soll die Beschreibung des Spieles in den Vordergrund stellen, um so für die Umsetzung eines Spieles auf Programmierung im traditionellen Sinne verzichten zu können. Um den hohen Entwicklungsaufwand zu rechtfertigen, sollen dabei möglichst viele unterschiedliche Konzepte von Gesellschaftsspielen durch diese Sprache abgebildet werden.

Als Ziel soll es Spieleentwicklern ermöglicht werden, ihren Produktionsaufwand stark zu reduzieren und ihre Anstrengungen auf die Spielmechanik konzentrieren zu können. Auf diese Weise soll die beschriebene Lücke im Bereich der unterstützenden Werkzeuge bestmöglich geschlossen werden.

1.2. Gliederung

In Kapitel 2 soll zunächst auf die Grundlagen von domänenspezifischen Sprachen eingegangen werden, wobei der Fokus auf die für das Verständnis dieser Arbeit notwendigen Thematiken beschränkt sein soll. Anschließend soll im Rahmen einer Diskussion von unterschiedlichen Ansätzen eine eigene für diese Arbeit als Grundlage anzusehende Definition von Elementen in Gesellschaftsspielen erarbeitet werden. Abschließend soll eine Klassifikation für Gesellschaftsspiele aufgezeigt werden, bei der die Spiele nach Spielmechaniken unterteilt werden sollen.

Innerhalb des Kapitels 3 soll zunächst ein Gesellschaftsspiel als Referenzbeispiel vorgestellt und anhand der vorher festgelegten Definition analysiert werden. Anschließend soll eine Vision vorgestellt und aus dieser ein Minimal-Entwurf abgeleitet werden. Im Rahmen dieses Entwurfes sollen die Anforderungen an die domänenspezifische Sprache und das Design festgelegt werden. Schließlich sollen zwei vergleichbare Arbeiten vorgestellt werden, Eberos GML2D und LudoCore. Die dort genutzten Ansätze sollen inhaltlich diskutiert und von dieser Arbeit abgegrenzt werden.

Das Kapitel 4 soll zunächst die grundlegenden Konzepte der Sprache beschreiben und anschließend jedes Element einzeln vorstellen und diskutieren.

Die Umsetzung der beschriebenen Sprache wird in Kapitel 5 vorgestellt. Dazu soll zunächst eine Architektur-Übersicht das Zusammenspiel der unterschiedlichen Komponenten aufzeigen. Danach werden diese Komponenten, die Entwicklungsoberfläche, der DSL-Generator, der Code-Generator, das Framework und die Test-Runtime beschrieben werden.

Abschließend soll in Kapitel 6 der Inhalt dieser Arbeit zusammengefasst und bewertet werden. Zuletzt sollen im Rahmen eines Ausblicks mögliche weitere Schritte skizziert werden.

2. Domänenspezifische Sprachen und die Elemente der Spielmechanik

2.1. Domain Specific Languages

2.1.1. Einführung

Im Rahmen der Entwicklung von Computersprachen wurde sich bisher auf die sogenannten „General-purpose languages“ (GPLs), die Universalsprachen, wie C++, Java und C# konzentriert. Der Hintergrund dieser Entwicklung ist der Antrieb gewesen, Programmiersprachen zu entwickeln, welche besser den Anforderungen der imperativen Programmierung genügen und in dem Sinne zu einer effizienteren Nutzung führen. Paradoxerweise existieren mittlerweile eine Vielzahl solcher Universalsprachen und es werden ständig neue entwickelt. Innerhalb des letzten Jahrzehnts hat sich dabei die Aufmerksamkeit der Forschung zugunsten der sogenannte „domain-specific-languages“ (DSLs) verschoben[1]. Dabei ist das Konzept von DSLs nicht neu. Ein bekannter Vertreter ist Beispielsweise die BNF [2] aus dem Jahr 1959.

Zunächst soll also der Begriff der DSLs anhand von den durch Fowler[3] verwendeten Merkmalen beschrieben werden:

Programmiersprache: Eine DSL ist eine von einem Menschen genutzte Sprache, um einem Computer Anweisungen zu erteilen.

Sprachliche Natur: Eine DSL ist eine Programmiersprache, bei der ein gewisser Sprachfluss vorhanden sein sollte und die Ausdrucksfähigkeit nicht nur auf einzelne Definitionen beschränkt ist, sondern das Verhalten eines Systems beschreibt.

Beschränkte Ausdruckskraft: Eine DSL sollte sich im Gegensatz zu einer Universalsprache auf eine minimale Menge von Sprach-Elementen beschränken, die für die Unterstützung der Domäne notwendig ist.

---

[1] Vgl. Taha (2008)
[2] Siehe Backus
[3] (Fowler, 2010, Vgl. S. 32)

Fokus auf Domäne: Eine beschränkte Sprache ist nur sinnvoll, wenn sie sich deutlich auf bestimmte Aspekte einer Domäne konzentriert.

Ein bekanntes Beispiel einer DSL stellt Microsoft Excel dar. Mithilfe der „Spreadsheets“ lassen sich zwar keine dreidimensionalen Animationen erstellen oder Echtzeitsysteme programmieren, aber sie sind exzellent, um die unterschiedlichsten Berechnungen im Rahmen der Tabellenkalkulation durchzuführen. Weiterhin ist nur ein geringes Vorwissen notwendig, um sie zu nutzen. Diese beiden Faktoren sind maßgeblich für die hohe Verbreitung von Excel.

Betrachtet man Excel in Bezug auf die Pflege und den Aufwand, den der Hersteller in dieses einzige Produkt fließen lässt, stellt sich die berechtigte Frage, wann sich die Entwicklung einer domänenspezifischen Sprache überhaupt lohnt. Indem eine DSL maßgeschneiderte Lösungswege für eine bestimmte Domäne erzeugen kann, weist sie im Gegensatz zu GPLs einen höhere Ausdrucksfähigkeit und einfachere Nutzbarkeit auf. Diese beiden Merkmale korrelieren dabei mit einer erhöhten Produktivität und niedrigeren Wartungskosten. Da zusätzlich das notwendige Programmier- und Domänenwissen in die Sprache eingeflossen ist, wird diese auch für eine große Gruppe von Personen ohne ausgereifte Programmierkenntnisse zugänglich. Die Abbildung 2.1 zeigt einige bekannte DSLs und teilt diese laut Mernik [4] in von Jones (1996) definierte „Language levels“ ein. Dieser „level“ steht dabei mit der in

DSL	Application Domain	Level	
BNF	Syntax specification	n.a.	
Excel	Spreadsheets	57	(version 5)
HTML	Hypertext web pages	22	(version 3.0)
LaTeX	Typesetting	n.a.	
Make	Software building	15	
MATLAB	Technical computing	n.a.	
SQL	Database queries	25	
VHDL	Hardware design	17	
Java	General-purpose	6	(comparison only)

Abbildung 2.1.: Auszug bekannter domänenspezifischer Sprachen

Abbildung 2.2 zu sehenden Einteilung von Produktivität in Beziehung[5].

Diese quantitativen Untersuchungen unterstützen die Aussage, dass die Entwicklung einer DSL mit den bereits genannten Vorteilen einhergehen.

Die Realisierung einer eigenen DSL stellt dabei sehr hohe Anforderungen an den Entwicklungsprozess. Um diesen Prozess zu erleichtern, werden unterschiedliche Entscheidungs- und Analyse-Muster vorgeschlagen, welche die Basis für unterschiedliche Design-Muster bieten [6].

[4] (Mernik u. a., 2005, Vgl. S. 317)
[5] Aus Mernik u. a. (2005) nach Jones (1996)
[6] (Mernik u. a., 2005, Vgl. S.317- 327)

Level	Productivity Average per Staff Month (FP)
1–3	5–10
4–8	10–20
9–15	16–23
16–23	15–30
24–55	30–50
> 55	40–100

Abbildung 2.2.: Language Level in Beziehung zur Produktivität, gemessen in Function Points (FP)

In Bezug auf den Aufwand argumentiert Fowler[7], dass eine DSL eine zusätzliche Ebene über den eigentlichen APIs darstellen und nicht unbedingt mit viel Arbeit verbunden sein muss. Weiterhin hebt Fowler hervor, dass die Implementierung von DSLs nicht verbreitet ist und daher eine entsprechende Lernphase mit einzuberechnen ist. Den größten Aufwand und die Herausforderung sieht er dabei in dem Prozess, das Konzept der domänenspezifischen Sprache zu entwerfen.

Die Hintergrundinformationen der nächsten Abschnitte sollen sich dabei auf die für diese Arbeit relevanten Themenbereiche beschränken. Im weiteren Verlauf sollen Parsing-Verfahren für die Entwicklung für externe domänenspezifische Sprachen im Vordergrund stehen und die für interne DSL relevanten Techniken vernachlässigt werden. Weiterhin soll die Verwendung von XML als Basis von domänenspezifischen Sprachen diskutiert werden. Abschließend sollen alternative Ausführungsmodelle vorgestellt und das Modell des Zustandsautomaten näher betrachtet werden.

2.1.2. Grundlagen

Eine domänenspezifische Sprache stellt grundsätzlich die Ausdrucksmöglichkeiten für die darunter liegende Funktionalität zur Verfügung. Wenn beispielsweise ein beschriebenes Zustandsdiagramm auch ausgeführt werden soll, muss ein entsprechender Zustandsautomat als Anwendung implementiert werden. Der Zustandsautomat stellt in diesem Falle die Funktionalität und auch das semantische Modell der DSL dar. Abbildung 2.3 verdeutlicht den Zusammenhang. In diesem Fall war das semantische Modell ein Objekt-Modell. Ein semantisches Modell kann auch andere Modelle abbilden, wie z.B. die Beschreibung einer Datenstruktur oder eine bestimmte Code-Bibliothek. Betrachtet man eine DSL aus der Sicht des semantischen Modells, kann die domänenspezifische Sprache für dessen Konfiguration genutzt werden.

[7](Fowler, 2010, Vgl. S. 39)

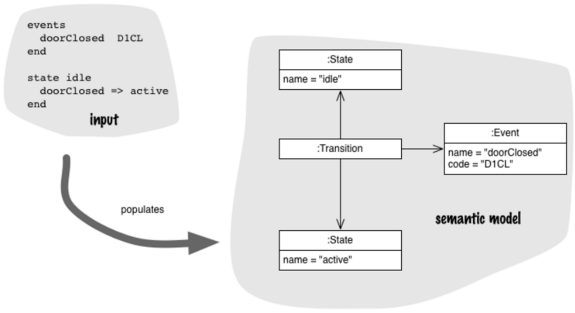


Abbildung 2.3.: Die DSL wird interpretiert und bestückt das semantische Modell, Quelle: (Fowler, 2010, S. 159)

Die konzeptionelle Trennung zwischen semantischem Modell und der DSL verhindert, dass die DSL-Syntax die Sichtweise auf die Domäne erschwert. Eine DSL kann über die Runtime direkt interpretiert werden. Die Runtime ist eine Anwendung, welche in der Lage ist, die DSL zu verarbeiten. Abbildung 2.4 zeigt, dass beim Interpretieren der Parser direkt in der Runtime angesiedelt ist und so die DSL zur Laufzeit verarbeitet werden kann.

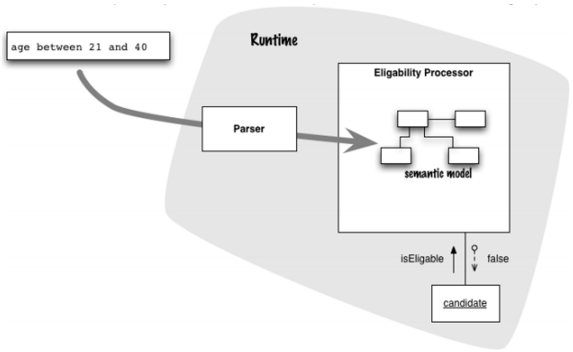


Abbildung 2.4.: Interpretieren einer DSL, Quelle: (Fowler, 2010, S. 22)

Eine Alternative zur Interpretierung ist die Generierung von Code. Der Code einer DSL sollte generiert werden, wenn die Ausführung und das Parsen der DSL an unterschiedlichen Orten ausgeführt werden soll. Weiterhin können mögliche Abhängigkeiten von Bibliotheken, welche nur für die Entwicklungs- oder die Produktionsumgebung relevant sind, bei der Generierung einbezogen werden. Ebenso relevant ist der Aufwand, den eine Verarbeitung zur Laufzeit mit sich bringt [8].

Abbildung 2.5 zeigt, dass bei der Generierung des Codes über den DSL-Processor Code-Fragemente generiert und anschließend innerhalb der Runtime genutzt werden.

[8] (Spinellis, 2001, Vgl. S. 97)

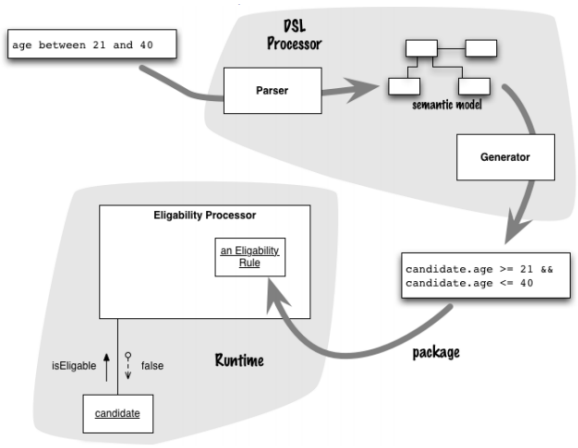


Abbildung 2.5.: Generierung über einen DSL-Processor, Quelle: (Fowler, 2010, S. 23)

Weiterhin lassen sich domänenspezifische Sprachen in drei Kategorien unterteilen [9]:

- Eine externe DSL ist eine eigene separate Sprache, welche nicht der Sprache der Runtime entspricht. Gewöhnlicher weise besitzt sie eine eigene Syntax, wobei ebenso auch die Syntax einer dritten Sprache verwendet werden kann (XML ist ein häufiges Beispiel). SQL ist ein Beispiel einer externen DSL.
- Eine interne DSL basiert auf einer Trägersprache, meist einer GPL. Ein Script in einer internen DSL stellt validen Code innerhalb der GPL dar, nutzt aber nur einen Teil der Sprach-Eigenschaften, um einen bestimmten Aspekt eines größeren Systems abzubilden. Das Resultat sollt einer eigenen Sprache entsprechen und leicht von der genutzten GPL abgrenzbar sein. Das klassische Beispiels für eine interne DSL ist Lisp.
- Eine Language Workbench ist eine spezialisierte Entwicklungsumgebung für die Entwicklung und Umsetzung von DSLs. Dabei wird die Workbench nicht nur für die Umsetzung der Syntax der DSL genutzt, sondern bietet gleichzeitig eine für die DSL angepasste Oberfläche für das Arbeiten mit der Sprache mit an.

2.1.3. Grammatik, Syntax und Semantik

Die Wörter einer Sprache bestehen aus einer endlichen Menge von Symbolen. Diese Menge wird das Alphabet genannt. Die einfachste Sprache des Alphabets besteht aus sämtlichen Kombinationen, die sich aus den Symbolen des Alphabets ergeben. Diese Kombinationen

[9] (Fowler, 2010, Vgl. S. 27)

lassen sich in einer Baumstruktur abbilden. Wird die Grammatik genutzt, um den Baum generieren zu lassen, spricht man von einem Erzeugungsbaum. Eine Sprache kann näher anhand von Grammatiken beschrieben werden. Der Bereich der formalen Grammatiken entstammt dem Feld der Mathematik wobei, Chomsky (1959) dabei mit seiner Forschung die Grundlage für die Bereiche der formalen Sprachen, Parser und eine großen Teil der Compiler-Konstruktion gelegt hat[10].

Die Grammatiken lassen dabei wiederum in kontextabhängige-, kontextfreie-, reguläre- und Finite-Choice- und weitere Grammatiken mit unterschiedlichen Eigenschaften einteilen[11]. An dieser Stelle soll genügen, dass die Grammatiken genutzt werden um die Syntax der Sprache zu definieren. Die Syntax ist gleichbedeutend mit der formalen Struktur, welche durch die Grammatik vorgegeben ist.

Die Semantik hingegen wird allein durch das semantische Modell gegeben. Beispielsweise könnte `5 + 3` entweder `8` oder `53` bedeuten.

Für das Parsen einer externen DSL werden oft Grammatiken eingesetzt, bei einer internen werden diese hingegen seltener. Dies ist darauf zurückzuführen, dass eine internen DSL über unterschiedliche Verfahren auf die Trägersprache abgebildet werden kann. Diese besitzt zwar eine Grammatik, aber die notwendigen Parser stehen bereits zur Verfügung[12].

2.1.4. Parsing

Einen String zu parsen bedeutet, dass der Erzeugungsbaum erstellt werden muss, welcher den zu parsenden String in einer gegebenen Grammatik erzeugen würde. Wenn der Baum nicht existiert, ist der String nicht Teil der Grammatik und in dem Sinne ist ein Fehler aufgetreten. Der Baum, der anhand der Syntax der Grammatik erzeugt wird, heißt Syntaxbaum. [13].

Über einen Pre-Processor kann dabei eine Vorverarbeitung der Eingabe erfolgen, welche dann von dem Parser interpretiert wird. Beispielweise können über Macros zum einen bestimmte Texte oder auch syntaktische Konstrukte in einer Sprache ersetzt werden. Bei `C` wird z.B. die Anweisung `sqrt(x)` durch den Pre-Processor durch `x*x` ersetzt. Folgend sollen die Verfahren für die syntaktische Analyse vorgestellt werden.

Laut Fowler[14] bieten externe DSL größere Freiheit, weil sie nicht durch die Syntax der Trägersprache eingeschränkt sind. Als Resultat dieser Freiheit muss die syntaktische Analyse der

[10](Chomsky, 1959, Vgl. Seite. 20 - 24)
[11](Chomsky, 1959, Vgl. Seite. 28 - 40)
[12](Fowler, 2010, Vgl. S. 50)
[13](Grune und Jacobs, 2008, Vgl. S. 62-63)
[14](Fowler, 2010, Vgl. S. 89)

DSL selbst vorgenommen werden, wobei hierfür unterschiedliche Strategien existieren. Über die Syntax kann ein DSL-Script analysiert werden und daraus ein sogenannter Syntaxbaum erzeugt werden. Abbildung 2.6 verdeutlicht diesen Zusammenhang.

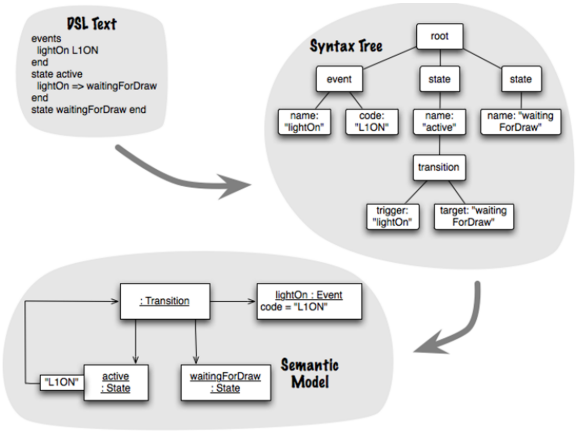


Abbildung 2.6.: Beispiel eines Syntaxbaumes Quelle: (Fowler, 2010, Vgl. S. 49)

Bei der „Delimiter-Directed Translation“ findet keine Grammatik Anwendung und die Eingabe wird einfach in Anweisungen eingeteilt, dabei kann ein Symbol das Ende einer Anweisung anzeigen oder nicht. Nachteilig ist, dass die Eingabe-Hierarchie nicht abgebildet wird. Diese „manuelle“ Verarbeitung ist nur für besonders einfache DSL nutzbar [15].

Bei der „Syntax-Directed Translation“ wird zunächst eine Grammatik definiert, über welche die Eingabe analysiert und der Syntaxbaum erstellt werden kann. Um eine Grammatik in einen Parser zu übersetzen, gibt es wiederum unterschiedliche Techniken, wie „Recursive Descent Parser“[16], „Parser Combinator“[17] oder auch „Parser Generator“ [18], welcher den meist genutzten Ansatz darstellt. Dabei wird in einer BNF ähnlichen DSL die syntaktische Struktur beschrieben und darauf basierend der Parser generiert. Dieser Schritt stellt eine große Arbeitserleichterung dar, da die generierten Parser auch das schwierige Thema der Fehlerbehandlung mit einschließen[19].

Dennoch soll die Behandlung von Fehlern folgend kurz vorgestellt werden. Ein großes Problem besteht bereits in der Lokalisierung des Fehlers, da er nicht unbedingt an der Stelle liegen muss, an welcher der Parser das Zeichen nicht identifizieren konnte. Weiterhin soll der Parser auch nicht nach dem ersten Fehler vollständig abbrechen, damit auch weitere Fehler innerhalb des Scripts identifiziert werden können. [20]

[15] Cunningham (2008)
[16] siehe (Parr, 2010, S. 40-41)
[17] siehe (Parr, 2010, S. 72)
[18] siehe (Parr, 2010, S. 42)
[19] (Parr, 2010, Vgl. S. 292)
[20] (Grune und Jacobs, 2008, Vgl. S. 227 - 228)

Die Fehlerbehandlung lässt sich in Ad-hoc-, regionale und lokale Verfahren einteilen. Als ein einfaches Beispiels soll ein lokales Verfahren dienen, der sogenannte „Panic Mode“. Bei allen lokalen Verfahren findet die sogenannte Fehlerwiederherstellungs-Technik „acceptable-set“ Anwendung. Wenn ein Parsing-Fehler auftritt, wird die Eingabe solange ignoriert, bis auf eines der Zeichen des Sets getroffen wird. Bei „Panic Mode“ besteht dieses Set aus jenen Symbolen, welche das Ende einer Anweisung signalisieren, in vielen Sprachen handelt es sich dabei das `;`. Auf diese Weise kann die Position des Fehlers eingeschränkt werden und gleichzeitig der Rest der Eingabe verarbeitet werden [21].

Als Resultat des Parsing-Prozesses sollen die Objekte des semantischen Modells erzeugt werden. Diese können in einigen Fällen allerdings nicht direkt erstellt werden, da diese sich erst nach und nach im Laufe des Parsens zusammenfügen. Dieses Problem kann anhand von „Construction-Builder“ gelöst werden. Diese entsprechen den semantischen Objekten, wobei sämtliche Eigenschaften les- und schreibbar sind und zum Ende das semantische Objekt erzeugen können. Weiterhin können die Hierarchie-Informationen des Parsers genutzt werden[22].

Um Zugriff auf Informationen außerhalb des aktuelles Zweiges des Syntaxbaumes zu erhalten, wird eine sogenannte Symboltabelle angelegt, welche grundsätzlich als Nachschlagewerk für definierte Elemente fungiert [23].

2.1.5. XML DSLs

XML kann ähnlich wie eine Trägersprache bei einer internen DSL angesehen werden. Der grundlegende XML-Syntax legt dabei fest, wie Elemente und deren Attribute sowie Kommentare formuliert werden können. Zusätzlich kann die Syntax über eine Schema-Definition näher spezifiziert werden. Das DOM [24] eines XML-Dokumentes kann mit dem Syntaxbaum verglichen werden.

Die Verarbeitung einer externen DSL basierend auf XML wird maßgeblich erleichtert. Um das DOM zu verarbeiten kann auf vorhanden SAX-Parser [25] aufgebaut werden, um die Objekte des semantischen Modells zu erzeugen. Dies gilt ebenso für die Definition der DSL-Scripte. Durch die ausgeprägte Syntaxdefinitionen durch die Schemata unterstützen mittlerweile viele Editoren eine ausgereifte Fehlerbehandlung.

[21] (Grune und Jacobs, 2008, Vgl. S. 240 - 241)
[22] (Fowler, 2010, Vgl. S. 50)
[23] (Parr, 2010, Vgl. S. 146-147)
[24] Das Document Object Model ist eine Spezifikation für den Zugriff auf XML-Dokumente
[25] Simple Api For XML

Als großer Nachteil muss an dieser Stelle die schlechtere Lesbarkeit angeführt werden. Dies ist auf die Art und Weise zurückzuführen, wie in XML Elemente und deren Verschachtelung formuliert werden.

2.1.6. Alternative Ausführungsmodelle

Allgemein kann zwischen imperativer und deklarativer Programmierung unterschieden werden. Die GPLs folgen dem imperativen Ansatz. Dieser zeichnet sich dadurch aus, dass die Abläufe im Code im Detail nachvollziehbar sind. Der große Zusammenhang hingegen ist schwer nachzuvollziehen, da dieser sich meist auf unterschiedliche Ebenen verteilt. Die deklarative Programmierung kann auf der gegensätzlichen Seite angeordnet werden. Sie beschreibt die Zusammenhänge und versteckt die detaillierten Abläufe. Allerdings muss der Entwickler für die Programmierung das Verarbeitungsmodell kennen, auf welchem der deklarative Code ausgeführt wird. Ein gutes Beispiel stellt Prolog dar [26]. DSLs zeigen besonders ihre Stärken, wenn Probleme schlecht über den imperativen Ansatz gelöst werden können[27].

Folgend sollen unterschiedliche Ausführungsmodelle vorgestellt werden [28]:

Entscheidungstabellen Die Ausführung wird in Form einer leicht nachvollziehbaren Entscheidungstabelle durchgeführt[29]. Abbildung 2.7 zeigt ein Beispiel und verdeutlicht, dass diese Sichtweise besonders verständlich für Nichtprogrammierer ist.

Produktionssysteme Ein Produktionssystem erlaubt das Definieren von Vorbedingungen und Auswirkungen. Aus der Menge der definierten Regeln ergibt sich die Verhaltensweise des Systems. Dabei kann diese in manchen Fällen unvorhergesehen und schwer nachvollziehbar sein. Produktionssysteme werden in automatisierten Planungs- und Experten-Systemen sowie in der KI eingesetzt, um das Verhalten von Agenten zu definieren und die nächste Aktion auszuwählen.

Die folgende Deklaration zeigt eine beispielhafte Definition eines Produktionssystems:

```
if
passenger.frequentFlier
then
passenger.priorityHandling = true;

if
mileage > 25000
then
passenger.frequentFlier = true;
```

[26] siehe Apt (1993)
[27] (Fowler, 2010, Vgl. S. 89 - 92)
[28] (Fowler, 2010, Vgl. S. 92-94)
[29] (Kirk, 1965, Vgl. S. 41)

Tabellenbezeichnung	R1	R2	R3	R4	R5	R6	R7	R8
Bedingungen								
Lieferfähig	j	j	j	j	n	n	n	n
Angaben vollständig	j	j	n	n	j	j	n	n
Bonität in Ordnung	j	n	j	n	j	n	j	n
Aktionen								
Lieferung mit Rechnung	x		x					
Lieferung als Nachnahme		x		x				
Angaben vervollständigen			x	x				
Mitteilen: nicht lieferbar					x	x	x	x

Abbildung 2.7.: Beispiel einer Entscheidungstabelle, Quelle: Wikipedia (a)

Dependency Network Erlaubt das Verknüpfen von Tasks, indem die jeweiligen Abhängigkeiten in Form von Vorbedingungen beschrieben werden können. Als bekannte Beispiel können Make [30] und Ant[31].

Die folgende Deklaration zeigt wie in Make Regeln definiert werden, um eine Menge von Kommandos auszuführen mit dem Ziel, eine Datei zu erstellen.

```
target:   dependencies ...
            commands
          ...
```

Zustandsautomaten Teilt das Verhalten eines Systems in Zustände ein. Durch Ereignisse kann das System in andere Zustände übergehen. Dieses Ausführungsmodell wird im Anschluss näher betrachtet.

Ausführungsmodell Zustandsautomat

Für das Ausführungsmodell eines Zustandsautomaten existieren unterschiedliche Variationen. Grundsätzlich kann ein Zustandsautomat aus beliebig vielen Zuständen bestehen. Für jeden Zustand können beliebig viele Zustandsübergänge definiert werden, wobei jede Zustandsänderung durch ein Ereignis ausgelöst wird und der Automat anschließend in den entsprechenden Ziel-Zustand wechselt. Dieser Ziel-Zustand kann, muss aber nicht ein andere

[30] http://www.gnu.org/software/make/
[31] http://ant.apache.org/

Zustand als der vorherige Zustand sein. Allgemein muss geklärt werden, wie ein Zustandsautomat mit Ereignissen umgeht, die nicht definiert sind. Diese können entweder einen Fehler erzeugen oder ignoriert werden. Weiterhin können Vorbedingungen für Zustandsübergänge definiert werden, sogenannte Wächterbedingungen. Abbildung 2.8 zeigt ein Beispiel eines Zustandsautomaten. Diese Vorbedingungen müssen dabei ausschlussfrei definiert sein, damit jeder Zustandsübergang eindeutig ist. Damit eine DSL mit dem Zustandsautomaten

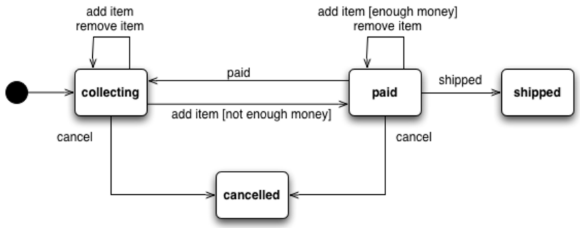


Abbildung 2.8.: Beispiel eines Zustandsautomaten

interagieren können, müssen Aktionen auf irgendeine Art und Weise angebunden werden können. Dabei ist es möglich, für diese Schnittstelle die Zustandsübergänge oder die Zustände zu wählen [32].

2.2. Elemente der Spielmechanik

2.2.1. Einleitung

Wenn man ein Spiel um den visuellen, technologischen und geschichtlichen Rahmen reduziert, verbleiben bestimmte Konzepte und Objekte, welche untereinander in Interaktionen und Beziehungen stehen. Trotz der umfassenden Versuche, eine möglichst allgemeingültige Ontologie zu definieren, hat sich bisher keine dominante Klassifizierung aus den unterschiedlichen Ansätzen herausgestellt[33]. Dies ist zum einen darauf zurückzuführen, dass das Thema der Spielmechanik aus sehr unterschiedlichen Richtungen betrachtet werden kann und zum anderen auf den Umstand, dass eine Masse an unterschiedlichen Taxonomien existiert, welche sich sprachlich zwar ähneln, sich aber wiederum in ihren formalen Definitionen unterscheiden.

Da das Ziel dieser Arbeit der Entwurf und das Design einer domänenspezifischen Sprache zur Beschreibung von Gesellschaftsspielen ist, soll in den folgenden Abschnitten eine

[32] (Fowler, 2010, Vgl. S. 527 - 530)

[33] Beispielsweise betrachten Salen und Zimmerman (2004) Spiele als System, für Cook (2006) ist das Feedback ein zentraler Bestandteil, Björk u. a. (2003) unterteilt Spiele in Muster und Järvinen (2003) definiert Umgebungen, Regeln und Komponenten

grundlegende Definition erarbeitet werden. Diese Definition soll dabei anhand der bestehenden Forschung diskutiert werden und den domänenspezifischen Rahmen für die in Kapitel 4 beschriebene Sprache bilden. Die Spielmechanik-Elemente sollen dabei ausschließlich in dem Zusammenhang der Gesellschaftsspiele betrachtet werden, um eine Vermischung mit anderen Spiele-Typen zu vermeiden.

Die für diese Arbeit definierten domänenspezifischen Begriffe sollen dabei auf **diese Weise** hervorgehoben werden, um sie von den anderen Begrifflichkeiten eindeutig abzugrenzen.

2.2.2. Spaces, Fields und Visibility

Jedes Spiel findet in einem Raum statt. Dieser Spielraum definiert eine beliebigen Anzahl von Orten und legt die Beziehungen der Orte untereinander fest. Spielräume zeichnen sich auf einer abstrakten Ebene durch folgende Eigenschaften aus [34]:

1. Sie sind diskret oder kontinuierlich.
2. Sie besitzen eine Anzahl von Dimensionen.
3. Sie besitzen Orte, welche miteinander in Verbindung stehen können.

Die Spielräume bei Gesellschaftsspielen, insbesondere den Brettspielen, finden in der Regel in diskreten Spielräumen statt. Diese Spielräume sollen im weiteren Verlauf als **Spaces** bezeichnet werden. Ein konkreter Bereich innerhalb eines **Spaces** soll wiederum als **Field** gelten.

Järvinen[35] fasst die Spielräume unter dem Begriff „Environments“ (zu deutsch Umgebungen) zusammen. Die anderen Elemente des Spieles befinden sich entweder in der Umgebung oder stehen mit ihr in Beziehung. Dabei wirken sich die räumlichen Beschränkungen, die sich aus der Definition des Spielraumes ergeben, stark auf die Kombinationsmöglichkeiten der anderen Spiel-Elemente aus. Weiterhin beschreibt er, dass Environments folgende Merkmale aufweisen können:

Ganzes/Teile-Relation: Diese Eigenschaft beschreibt, wie und ob sich eine Umgebung in Beziehung mit anderen Umgebungen befindet [36] .

Zustand: Eine Umgebung oder ein enthaltener Ort kann einen Zustand besitzen, beispielsweise ob ein Feld besetzt oder nicht besetzt ist.

Vektoren: Beschreiben, welche Bewegungen unter Einbeziehung der Richtung zwischen Orten einer Umgebung zulässig sind.

Die von Järvinen (2008) vorgestellten Eigenschaften einer Umgebung sollen auf die hier vorgestellten **Spaces** übertragen werden: Die **Fields** eines **Spaces** sollen mit den **Fields** anderer **Spaces** verknüpft und eigene Zustände besitzen können. Wiederum kann die Richtung der Verknüpfung von **Fields** festgelegt werden.

Weiterhin beschreibt (Järvinen, 2008, Vgl. S.68) zwei Typen von Umgebungen für Gesellschatsspiele: Spielbretter/ Spielfelder und Aufbau. Eine Umgebung bestehend aus Spielbrett und Spielfeld beschreibt entweder eine festgelegte Spielumgebung, in welcher die Elemente

[34] Vgl. (Schell, 2008, Seite 131)
[35] (Järvinen, 2008, Seite 66-68)
[36] (Schell, 2008, Seite 134) greift die Verschachtelung von Spaces ebenfalls auf.

der Spielmechanik miteinander interagieren können, oder stellt eine sich dynamisch verändernde Umgebung dar. In diesen Fällen visualisiert die Umgebung die Interaktionsmöglichkeiten und den aktuellen Spielzustand. Bei einer Aufbau-Umgebung sind keine Spielbretter oder sonstige visualisierenden Hilfsmittel für die Definition der Umgebung notwendig, dennoch müssen die Elemente des Spieles auf irgendeine Art und Weise arrangiert werden, um einen bestimmten Spielzustand auszudrücken. Beispielsweise ist bei BlackJack[37] vorgegeben, dass die Karten des Spielers auf einer Seite und die Karten des Hauses auf der anderen Seite des Tisches platziert werden. Aus einer abstrakten Sichtweise lässt sich argumentieren, dass auch bei „Aufbau"-Umgebungen dedizierte Spielräume existieren und daher ebenso eine Umgebung abbilden. Daher soll in der hier vorgestellten Definition von **Spaces** nicht zwischen diesen beiden Kategorien unterschieden werden.

Ein recht bekanntes Beispiel für ein null-dimensionales und diskretes Spielfeld stellt „Risiko" dar. Wie in Abbildung 2.9 zu sehen, besteht das Spielbrett aus einer losen Anordnung von Ländern, die miteinander verknüpft sind.

Abbildung 2.9.: Risiko-Spielbrett, Quelle: Hasbro

Obwohl das Monopoly-Spielbrett, zu sehen in Abbildung 2.10, auf den ersten Blick zweidimensional zu sein scheint, kann es auch als eine lineare Anordnung von Feldern angesehen werden. Aus dieser linearen Anordnung wird ein Kreis gebildet, indem das am Ende befindliche Feld, die Schlossallee, mit dem ersten Feld (Los) verknüpft wird.

Schach stellt eine Ausprägung des klassischen zwei-dimensionalen Spielbrettes dar.

Ein kontinuierliches Spielfeld findet sich bei den sogenannten „Tabletop"-Strategiespiele[38], bei welchen keine Felder existieren, sondern Distanzen anhand von Maßbändern gemessenen werden. Wie in Abbildung 2.12 zu sehen, ähneln diese Spiele visuell schon stark den Computer-Strategiespielen.

Bei Järvinen[39] wird argumentiert, dass die Spielumgebung neben der Definition der Rahmenbedingungen für ein Spiel dazu genutzt wird, um die Atmosphäre des Spieles zu kommuni-

[37] siehe Wikipedia (b)
[38] Vgl. Magerkurth u. a. (2004)
[39] (Järvinen, 2003, Vgl. S. 75)

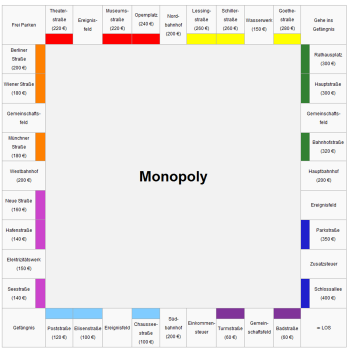


Abbildung 2.10.: Monopoly-Spielbrett, Quelle: Brothers

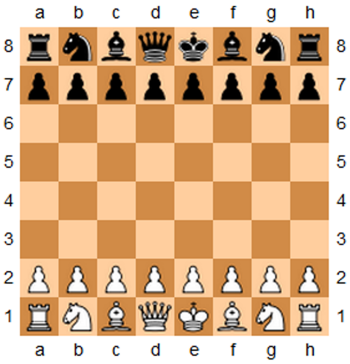


Abbildung 2.11.: Schach-Spielbrett, Quelle: Wikipedia (c)

zieren. Bei Brathwaite und Schreiber[40] wird diese Atmosphäre als das „Theme“ des Spieles bezeichnet und darauf hingewiesen, dass es für den Spielspaß einen beträchtlichen Unterschied macht, in was für einem visuellen und narrativen Rahmen ein Spiel eingebettet ist, unabhängig davon, ob es durch dasselbe Regelwerk abgebildet werden kann. Da sich diese Arbeit auf den abstrakten Kern von Spielmechaniken konzentriert, fließt die Atmosphäre nicht in die hier vorgestellte Definition der **Spaces** und anderer Elemente mit ein.

Salen und Zimmermann[41] betrachten Spiele unter anderem als ein System von Informationen, wobei grundlegend zwischen den durch Parlett (1990) geprägten Begriffen der perfekten und unvollständigen Information unterschieden wird. Eine perfekte Information ist dabei für alle Teilnehmer vollständig sichtbar (wie bei einem Schachspiel), eine unvollständigen Information nicht (Bei einem Kartenspiel sehen die anderen Spieler meist nur die Anzahl der Karten, die ein anderer Spieler in der Hand hält). Dies zeigt, dass die Sichtbarkeit der **Spaces** und deren **Fields** bei Gesellschaftsspielen eine Rolle spielen kann und entsprechend abge-

[40] (Brathwaite und Schreiber, 2009, Vgl. Seite 32)
[41] (Salen und Zimmerman, 2004, Seite 190-201)

Abbildung 2.12.: Tabletop-Spiel, Quelle: edinburghwargames.com

bildet werden muss. Im Rahmen der hier vorgestellten Definition soll die Sichtbarkeit durch das **Visibility**-Element beschrieben werden.

2.2.3. Objects

Neben **Spaces** lassen sich nach Schell[42] Objekte als ein Bestandteil der Spielemechanik identifizieren. Ein Objekt kann eine Spielfigur, irgendwelche Spielmarken, Karten oder sonstige visuelle Repräsentation innerhalb des Spieles darstellen.

Schell[43] argumentiert, dass sämtliche Elemente innerhalb von **Spaces** als Objekte dargestellt werden können und nennt als Beispiel die Straßen des Monopoly-Spielbrettes. Vergleicht man nach dieser Sichtweise jedoch die Spielfigur und das Straßen-Feld in Monopoly, wird deutlich, dass hier zwei unterschiedliche Verhaltensweisen miteinander vermischt werden. Das Spielfigur-Objekt befindet sich zwar in dem **Space**, bewegt sich aber auf Straßen-Objekten. An dieser Stelle soll nun festgelegt werden, dass sich ein **Object** dadurch auszeichnet, dass es sich jederzeit einem **Field** eines **Spaces** zuordnen lässt.

Järvinen [44] hingegen spricht an dieser Stelle von Komponenten. Dabei weist er darauf hin, dass es bei Komponenten ausschlaggebend ist, ob ein Spieler als Besitzer einer Komponente gilt oder nicht. Komponenten, die nicht in dem Besitz eines Spielers sind, sind im Besitz des Systems. Aus Sicht eines Spielers ergibt sich daraus die folgende Unterteilung: Eigene Komponenten, Komponenten anderer Spieler und Komponenten des Systems. Die Relation ist beispielsweise wichtig, wenn man sich eine Spielfigur in Monopoly als eine Komponente vorstellt. Sie darf entsprechend nur von ihrem Besitzer bewegt werden. Ebenso ist es vorstellbar, dass mehrere Spieler ein Team formen und eine Komponente viele Besitzer hat. Die von Järvinen (2008) vorgeschlagene Besitzer-Relation soll in die hier beschriebene Definitionen einfließen. Dabei sollen **Objects**, **Spaces** und **Fields** einem oder mehreren Spieler gehören können.

Die Abbildung 2.13 zeigt ein Spielbrett des Spiels Samurai. Es soll hervorheben, dass **Objects** nicht nur die unterschiedlichen schwarzen Figuren, sondern auch die unterschiedlich farbigen Marker repräsentieren können. Zudem sind alle **Objects** einem **Field** zugeordnet.

Abbildung 2.13.: Samurai-Spielbrett mit Objekten, Quelle: BoardGameGeek

[42] (Schell, 2008, Vgl. Seite 136-137)
[43] (Schell, 2008, Seite 136)
[44] (Järvinen, 2008, Vgl. Seite 62 - 66)

2.2.4. Attributes

Schell[45] definiert, dass die unterschiedlichen Eigenschaften eines Objekts anhand von Attributen definiert werden können. Über Attribute lassen sich Informationen unterschiedlicher Kategorien abbilden. Diese Eigenschaften soll im weiteren Verlauf als **Attributes** bezeichnet werden. Da Schell[46] , wie bereits erwähnt, den Begriff eines Objektes sehr weit fasst, soll an dieser Stelle verdeutlicht werden, dass **Attributes** nicht nur die Eigenschaften von **Objects**, sondern auch zur Beschreibung der Eigenschaften sämtlicher Elemente der Spielmechanik genutzt werden können.

Weiterhin teilt Schell[47] die Attribute in statische und dynamische Attribute ein. Diese Unterteilung ergibt während der Entwicklung des Spiele-Designs Sinn, da sie die im Laufe des Spieles veränderlichen von den unveränderlichen Attributen abgrenzt. Im Rahmen der hier vorgestellten Definition stellt sie allerdings eine Einschränkung dar und es wird bewusst nicht zwischen unveränderlichen und veränderlichen **Attributes** unterschieden. Wenn ein **Attribute** nicht verändert werden soll, sollte kein Element definiert werden, welches dies zulässt.

Bei Järvinen[48] sind die beschreibenden Informationen gleichbedeutend mit den Komponenten. An dieser Stelle soll hervorgehoben werden, dass die **Attributes** bewusst von den anderen Elementen der Spielmechanik abgegrenzt werden und diese nur beschreiben können.

[45](Schell, 2008, Seite 136)
[46](Schell, 2008, Seite 136)
[47](Schell, 2008, Seite 136)
[48](Järvinen, 2008, Vgl. Seite 63 - 64)

2.2.5. Actions, Operations und Implications

Als **Actions** werden die Möglichkeiten bezeichnet, die ein Spieler hat, um mit dem Spiel zu interagieren.

Schell[49] unterteilt dabei zwischen operativen und resultierenden Aktionen. Operative Aktionen stellen die grundlegenden Aktionen des Spielers dar, also im Beispiel von Schach das Bewegen der Spielfiguren nach den vorgegebenen Regeln. Die resultierenden Aktionen hingegen sind die Aktionen oder auch Strategien, welche sich aus den operativen Aktionen ergeben um ein bestimmtes Ziel zu erreichen. Bei Schach wäre das das Schützen einer eigenen Figur durch eine andere Figur. Die resultierenden Aktionen sind dabei nicht Teil der vorgegebenen Regeln, sondern ergeben sich aus der Kombination der im Rahmen der Regeln möglichen Aktionen. Dabei ist der Anteil von resultierenden zu operativen Aktionen zwar kein absolutes, aber dennoch ein adäquates Maß, um die verschiedenen Aktionsmöglichkeiten eines Spieles zu bewerten. Man könnte ein Spiel als elegant betrachten, welches mit wenigen operativen besonders viele resultierende Aktionen ermöglicht. Der Grund, dass sich viele Spiele ähneln, ist darauf zurückzuführen, dass sie die gleiche Menge von Aktionen nutzen. Innovative Spiele grenzen sich in dem Sinne ab, dass sie dem Spieler entweder neue operative oder resultierende Aktionen zur Verfügung stellen. Die Wahl der Aktionen beeinflusst die Spielmechanik grundlegend und sollte daher entsprechend wohlüberlegt sein.

Diese Einteilung ermöglicht eine Bewertung von Aktionen in Bezug auf ihren Einfluss innerhalb des Spieles, doch ist sie nur konzeptioneller Natur und für die Definition der hier vorgestellten **Action** unerheblich.

Viele Autoren betrachten daher die für ein Spiel prägnanten Aktionen als „Kern-Spielmechaniken“. Sicart (2008) vertieft dabei, wie Spielmechaniken in primäre und sekundäre Mechaniken eingeteilt werden können. Diese Einteilung richtet sich danach, ob die Mechaniken zum Erreichen des Spielzieles notwendig oder nur unterstützend sind.

Im Gegensatz zu diesen Definition soll die hier vorgestellte **Action** als eine einzige Aktion des Spielers gelten und nicht zur konzeptionellen Kategorisierung von Spielen dienen. Sie grenzt sich von den anderen Elementen ab, da sie die einzige Möglichkeit des Spielers darstellt, aus eigener Initiative zu interagieren.

Ein von Järvinen[50] geführter Vergleich stellt die Gegenüberstellung der Spielelemente mit dem Satzbau dar. Dabei werden Objekte als Nomen, Attribute als Adjektive und Aktionen als Verben bezeichnet. Eine konkrete Instanz eines Objektes, z.B. ein bestimmter Bauer in einem Schachspiel, wird entsprechend als Subjekt bezeichnet. Auf diese Weise wird versucht, die abstrakte Sicht auf die Spielmechanik durch die Klassifizierung in wohlbekannte und

[49] Vgl. 140-142(Schell, 2008, Seite 140)
[50] (Järvinen, 2008, Seite 30)

deutlich voneinander getrennte Begriffe zu vereinfachen. Diese zusätzliche sprachlichen Einteilung soll aber nicht in die hier aufgebaute Definition der Spielmechanik einfließen. Anstelle dessen sollen die Begrifflichkeiten wie Instanzen oder Typen bewusst genutzt werden.

Salen und Zimmermann[51] verwenden den Begriff der Interaktivität und betrachten detailliert die „Anatomie einer Entscheidung". Dabei unterscheiden sie zwischen Entscheidungen, die gefällt werden müssen und Entscheidungen, die gefällt werden können. Ein Spieler kann also dazu gezwungen werden, eine Aktion auszuführen, da die Regeln des Spieles dies vorschreiben. Die hier vorgestellte der Definition soll nur die atomare Ausführung der **Action** enthalten. Ob und wann diese **Action** ausgeführt werden darf, soll entsprechend durch die Regeln des Systems vorgegeben werden.

Nach Järvinen[52] werden die möglichen Aktionen der Spieler durch Regeln definiert. Dabei lassen sich unterschiedliche konzeptionelle Regeltypen unterscheiden; auf diese Typen wird erst im nächsten Abschnitt eingegangen. Ein weiterer in diesem Kontext genannter Begriff ist die Prozedur. Sie stellt die Durchführung der in den Regeln beschriebenen Auswirkungen dar. Dabei ist der Begriff der Prozedur laut Järvinen (2008) oft mit unterschiedlichen Bedeutungen belegt, wie z.B. Fullerton u. a. (2004). Bei Avedon und Sutton-Smith[53] wird eine Prozedur als eine Aktion angesehen, die durch einen Spieler oder durch das System durchgeführt werden kann. Järvinen (2008) selbst unterscheidet dabei, dass nur Spieler Aktionen ausführen können und das System über Prozeduren den Zustand des Spieles beeinflusst.

Ähnlich wie die von Järvinen (2008) vorgeschlagene Trennung soll die Ausführung von Aktionen und deren Auswirkungen voneinander abgegrenzt werden. Im Schach ist das Bewegen des Bauers aus Sicht des Spielers eine Aktion. Dennoch lässt sich die Durchführung einer bestimmten Bewegung und das eigentliche Verschieben eines Spielsteines ebenso voneinander trennen. Das Verschieben stellt in diesem Beispiel die Auswirkung der Bewegen-Aktion des Bauers da. Die Auswirkungen auf den Spielzustand sollen als **Operations** gelten. Dies bedeutet, dass die Veränderung eines **Attributes**, das Bewegen eines **Objects**, eine Veränderung des Besitzer eines Element ebenso **Operations** darstellen.

Da **Actions** nur durch einen Spieler durchgeführt werden können, und der Spielzustand nur über **Operations** verändert werden kann, muss das System ebenso **Operations** nutzen können, um den Spielzustand unabhängig von einer **Action** verändern zu können.

Vorübergehende Änderungen an dem Spielzustand sollen über **Implications** formuliert werden können. Sie sollen es ermöglichen, Auswirkungen zu beschreiben, die nur unter bestimmten Bedingungen in Kraft treten. Wenn diese Vorbedingungen nicht mehr erfüllt sind, wird auch der Effekt rückgängig gemacht. Auf diese Weise sollen Wechselwirkungen von

[51] (Salen und Zimmerman, 2004, Vgl. Seite 61-65)
[52] (Järvinen, 2008, Vgl. Seite 71)
[53] (Avedon und Sutton-Smith, 1971, Vgl. Seite 422)

Elementen untereinander beschrieben werden können. Dabei können sich **Implications** nur auf **Attributes** und die **Visibility** von Elementen auswirken.

2.2.6. Rules und Timer

Ein großer Unterschied zwischen traditionellen Spielen und Videospielen liegt in der Art und Weise, wie die Einhaltung der Spielregeln gewährleistet wird. Bei einem traditionellen Spiel wird das Einhalten der Regeln entweder durch die Spieler selbst oder durch einen „Unparteiischen“ überwacht. In einem Computerspiel wird diese Rolle von dem System übernommen. Dabei ist dies nicht nur als eine reine Komfort-Funktion seitens des Spielers zu betrachten, da hierdurch eine höhere Komplexität der Spielwelt ermöglicht wird. Der Spieler muss sich nicht an die Regeln erinnern, er muss sie teilweise noch nicht einmal kennen, da er nur Aktionen durchführen kann, die im Rahmen der Regeln erlaubt sind[54].

Weiterhin ist unbestritten, dass die Regeln einen Kernteil der Spielmechanik darstellen. Manche Autoren gehen sogar soweit zu sagen, dass ein Spiel nur aus seinen Regeln besteht:

> Every game is its rules, for they are what define it.“ Parlett (1999)

Formal unterscheidet Parlett dabei folgende Ebenen von Regeln und beschreibt deren Abhängigkeiten untereinander:

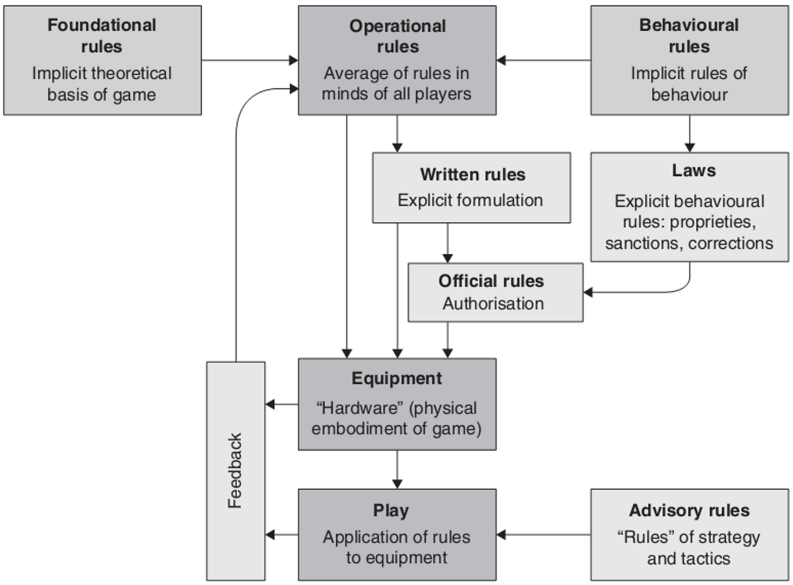


Abbildung 2.14.: Regeltypen aus (Schell, 2008, Seite 145) nach Parlett

Operational rules Dies sind die Regeln, welche die Spieler verstanden haben müssen, um das Spiel spielen zu können.

[54] Vgl. (Salen und Zimmerman, 2004, Seite Seite 87 - 90)

Foundational rules Sie stellen die abstrakte Version der „operational rules“ dar. Eine operative Regel, z.B. „der Spieler würfelt einen sechsseitigen Würfel und erhält die entsprechende Menge Gold“, würde als „foundation rules“ folgendermaßen beschrieben werden: „Die Anzahl des Goldes des Spielers wird durch eine zufällige Zahl zwischen 1 und 6 erhöht“.

Foundational rules In Gesellschaftsspielen dienen z.B. das Brett, die Figuren, die Chips oder Counter und weitere Elemente dazu, den elementaren Spielzustand darzustellen.

Behavioral Rules So werden die Regeln genannt, die man allgemein unter dem Begriff „Fairness“ zusammenfassen kann. Ein Beispiel wäre hierfür, dass es sich von selbst versteht, dass man nicht stundenweise über einen Spielzug nachdenken kann oder den Gegner während seines Spielzuges nicht über die Maßen ablenkt. Diese Regeln verdeutlichen, dass bei einem Spiel zwischen den Teilnehmern eine Art von sozialer Übereinkunft über die Art und Weise existiert, wie das Spiel durchgeführt werden soll. Diese sozialen Regeln wirken sich auf die „operational rules“ aus.

Written Rules Dies sind die schriftlichen Regeln, welche den Spielern die „operational rules“ erklären sollen. Dabei ziehen die meisten Menschen es vor, die Regeln eines Spieles von einem Mitspieler erklärt zu bekommen. Dies ist darauf zurückzuführen, dass zum einen die Feinheiten eines Spieles in vielen Fällen nicht leicht zu beschreiben und zum anderen diese Beschreibungen oft nicht deutlich genug sind, ohne das Spiel selbst bereits zu kennen. Aktuelle Videospiele integrieren dabei die grundlegenden Lernphasen mit in das Spiel und binden somit den Spieler bereits von Anfang an gut ein.

Laws Die „laws“ finden nur Anwendung, wenn das Spiel in einem wettkampfähnlichen und somit auch deutlich ernsterem Umfeld gespielt wird. Sie dienen zum einen dazu, den Rahmen eines Wettkampfes festzulegen (Wie viele Runden, wie viele Gegner, etc.) und zum anderen, um möglicherweise das Spiel in bestimmten Punkten zu verändern damit das Turnier ausgeglichener und fairer ist. Dabei können z.B. bestimmte Elemente im Spiel verboten werden, da diese deutlich stärker als andere Elemente sind. Ein Turnier, in dem alle Spieler die gleiche Taktik anwenden, ist für alle Beteiligten öde.

Official Rules Wenn die Spieler ihr Spiel auf eine ernsthafte Art und Weise betreiben, können die „official rules“ aus den „laws“ und „written rules“ hervorgehen. Sie sind noch nicht Teil der schriftlichen Regeln, werden aber allgemein angewendet. In Schach muss beispielsweise der Spieler, der den König des anderen Spielers bedroht, diesen informieren, indem er „Schach“ sagt. Diese Regel wurde ursprünglich nur auf Turnieren angewendet, heutzutage spielt wohl die Mehrheit der Spieler auf diese Art. Wenn „official rules“ weitgehend akzeptiert und über einen längeren Zeitraum eingesetzt worden sind, werden sie in der Regel in die „written rules“ aufgenommen.

Advisory Rules Diese Regeln dienen nur als Hilfestellung und zählen im Sinne der Spielmechanik nicht zu den Regeln des Spieles. Sie sind eher als Strategie-Leitpfaden zu betrachten. In Bezug auf Schach wären „Vermeide Doppelbauern“ oder „Pferd am Rand bringt Kummer und Schand“ ein Beispiel für diese Regelklasse.

Feedback (House Rules) Das Feedback, welche die „operational rules“ beeinflussen und anpassen, kann man auch als Hausregeln auffassen, welche von den Spielern eingeführt werden, um die Schwächen eines Spieles auszugleichen und es nach ihren Vorstellungen interessanter und spielenswerter zu gestalten.

Diese Defintion von Regeln verdeutlicht wiederum eine andere mögliche Sichtweise auf die Elemente eines Spieles. Die „Operational Rules“ fassen sämtliche Interaktionen zwischen Spielern und Spiel zusammen, wobei die „Foundational Rules“ eine gewisse Abstraktion dieser Regeln darstellen. Das „Equipment“ ist gleichzusetzen mit den hier vorgestellten **Objects**, wobei an dieser Stelle die Steuerung des Spieles nur den Regeln zugeordnet wird und das „Equipment“ als reine visuelle Repräsentation dient.

Die hier vorgestellte Definition von **Rules** soll es ermöglichen, Bedingungen zu definieren, welche die Abfrage von bestimmte Aspekten des Spielzustandes betreffen. In diesem Sinne ähneln sie den „Foundational Rules“. Allerdings enthalten sie im Gegensatz zu der Definition von Parlett keine Auswirkungen. Die Auswirkungen, welche bereits in Form der **Operations** beschrieben worden sind, sollen strikt von den Bedingungen getrennt sein.

Weiterhin soll eine **Rule** nicht eine Verkettung von Regeln beschreiben, sondern nur eine einzige atomare Regel. Über die Verkettung von mehreren **Rules** sollen komplexere Umstände beschrieben werden können.

Interessant sind an dieser Stelle auch die „Behavioral Rules“. Da Gesellschaftsspiele in der Regel in einem sozialen Kontext stattfinden, sollte das System hinter dem Spiel die Möglichkeiten bieten, ein Teil dieser „Behavioral Rules“ durchzusetzen. Dies betrifft unter anderem das oben beschriebene Beispiel, bei dem die Zeit eines Spielzuges begrenzt sein sollte. Für genau diesen Fall sollen sogenannte **Timer** existieren, welche festlegen können, dass eine **Action** oder eine **Operation** innerhalb einer bestimmten Zeitspanne ausgeführt werden muss.

Grundsätzlich folgt Järvinen[55] der Sichtweise, dass ein Spiel aus seinen Regeln besteht, versucht aber zusätzlich die Regeln in unterschiedliche Typen zu gliedern und enger mit den von ihm genannten Spiel-Elementen zu verknüpfen. Dabei wird argumentiert, dass die Regeln zwar die konzeptionelle Basis darstellen, diese sich aber entsprechend auf die anderen Elemente und deren Beschaffenheit auswirkt, da sie diese Regeln vergegenständlichen. Auf diese Weise legt Järvinen (2008) fest, dass eine Regel selbst kein eigenes Element darstellt, sondern die Elemente des Spieles maßgeblich beschreibt.

[55](Järvinen, 2008, Vgl. S. 69 - 72)

Die hier vorgestellten **Rules** sollen ebenso an die Spiel-Elemente gebunden sein, für welche sie Bedingungen beschreiben. Eine **Rule**, welche einen Spielzustand als Vorbedingung für das Ausführen einer bestimmte **Action** definiert, ist auch entsprechend dem **Action**-Element zugeordnet.

In Bezug auf die unterschiedlichen Regel-Typen sind jene Regeln von besonderer Wichtigkeit, welche die Siegbedingungen und somit das Ziel des Spieles definieren. Diese Regel-Typen stellen auch in der theoretischen Betrachtung von Spielen ein wichtiges Merkmal dar, da sie Spiele von Nicht-Spielen abgrenzen. Die Siegbedingungen charakterisieren unterschiedliche Spiele-Typen und stellen eine der Grundlagen der Taxonomie von Spielen dar. Bereits Schank und Abelson[56] stellen eine Kategorisierung zur Verfügung, wobei sie diese allerdings an der menschlichen Psyche ausrichten und Kategorien wie „Satisfaction goals“, „Enjoyemt goals“ und „Crisis goals“ definieren. Bjork und Holopainen[57] arbeiten das Thema der Siegbedingungen im Spielekontext grundlegend auf. Auf diese Weise können sie unterschiedliche Ziele wie z.B. „Gain Ownership“, „Overcome“, „Guard“ oder „Gain Information“ [58] von einander abgrenzen. Juul[59] argumentiert, dass sich diese unterschiedlichen Kategorien ebenso auf den Spieler auswirken. Juul (2007) zeigt, dass während des Spielens die Siegbedingungen mit bekannten Konzepten verglichen werden und so der Einstieg in ein unbekanntes Spiel maßgeblich erleichtert wird.

Die hier beschriebene Definition der **Rules** reduziert Regeln auf die Funktion von Vorbedingungen. Erst gemeinsam mit den unterschiedlichen Elementen wie **Actions** und **Operations** formen, das hier vorgestellte Fundament, bei dem eine Regel grundsätzliche Prozesse von Spielen beschreiben.

[56](Schank und Abelson, 1977, Vgl. S. 112-119)
[57](Bjork und Holopainen, 2004, Vgl. 277-338)
[58]Siehe (Bjork und Holopainen, 2004, S. 278-280, 297 und 302)
[59](Juul, 2007, Vgl. S. 511)

2.2.7. Phases, PhaseStates und ActionStates

Der Ablauf von Spielen lässt sich nach Juul (2005) grundsätzlich in Form von Zustandsautomaten abbilden:

> A game is a machine that can be in different states, it responds differently to the same input at different times, it contains input and output functions and definitions of what state and what input will lead to what following state (Juul, 2005, Seite 88)

Nach Perron und Wolf[60] wird das Spiel in zeitliche Referenzpunkte unterteilt, welche ein bestimmtes Ereignis innerhalb des Spieles repräsentiert. Dabei stellen das Spiel, seine Spieler und alle relevante Informationen eine bestimmte Konstellation dar. Insbesondere lässt sich das Spiel in unterschiedliche Phasen einteilen.

Weiterhin beschreibt Schell[61] sogenannte Modi in Spielen, welche sich aus voneinander unterscheidenden Spielabschnitten ergeben. Oft wird dabei zwischen einem Hauptmodus und mehreren Submodi unterteilt.

Salen und Zimmermann[62] heben hervor, dass ein Vergleich der unterschiedlichen Spielzustände zu einem Zustandsautomaten in rundenbasierten Spielen besonders naheliegend ist.

Basierend auf der Grundlage der hier genannten Definitionen soll ein Spiel in **Phases** unterteilt werden können. Jede **Phase** besteht dabei aus unterschiedlichen Zuständen, den **PhaseStates**. Innerhalb dieser **PhaseStates** sollen die erforderlichen Änderungen am Spielzustand in Form von **Operations** durchgeführt werden.

Da die **Actions** die einzigen durch den Spieler initiierten Interaktionsmöglichkeiten darstellen, soll es für sie ebenso möglich sein, Abläufe definieren zu können. Diese sollen anhand von **ActionStates** analog zu den **PhaseStates** beschrieben werden können.

[60] (Perron und Wolf, 2008, Vgl. S. 88)
[61] (Schell, 2008, Seite 147)
[62] (Salen und Zimmerman, 2004, Vgl. S. 218)

2.2.8. Actors und Selectors

Järvinen[63] beschreibt, dass der Spieler im Rahmen der Spielmechanik-Theorie eine bedeutende Position erhält, da erst durch seine Entscheidungen das System des Spiels Bedeutung gewinnt. Das Verhalten des Spielers kann im Rahmen des „Game Designs“ in bestimmte Richtungen gelenkt werden. Grundsätzlich können dem Spieler dabei Elemente des Spieles gehören, er kann Eigenschaften besitzen und Rollen zugewiesen bekommen und in Beziehen zu anderen Spielern oder Elementen des Spieles stehen.

Das hier vorgestellte Element **Actor** repräsentiert den Spieler innerhalb der in dieser Arbeit vorgestellten Definition. Ein **Actor** kann **Spaces** besitzen, welche **Fields** enthalten in denen wiederum **Objects** vorhanden sind. Die Eigenschaften eines **Actors** werden über **Attributes** definiert. Weiterhin kann er **Actions** besitzen.

Sicart (2008) führt aus, dass die durch das Spiel gesteuerte Spieler, auch Agenten genannt, möglicherweise noch andere Möglichkeiten haben, mit der Spielwelt zu interagieren, als ein menschlicher Spieler. Im Gegensatz zu dieser Definition eines **Actors** soll es dabei unerheblich sein, ob es sich um einen menschlichen Spieler oder um einen durch das System gesteuerten Agenten handelt.

Dem Thema der Spielerfahrung widmen sich viele Arbeiten, unter anderem Järvinen [64], Wei[65], Rouse sowie Ogden[66]. Da es sich im Rahmen dieser Arbeit um eine Implementierungssprache für Gesellschaftsspiele handelt, sollen diese Themenbereiche nicht näher betrachtet werden.

Anhand eines weiteren Elementes, einem **Selector**, sollen Verknüpfungen innerhalb der Spielmechanik hergestellt werden können. Bei diesen Verknüpfungen soll auch die Beziehung der Elemente untereinander, wie z.B. die Tatsache, dass ein **Actor** der Besitzer eines **Spaces** sein kann, genutzt werden können.

[63] (Järvinen, 2008, Vgl. S. 83 - 84)
[64] (Järvinen, 2008, S. 150 - 220)
[65] Wei u. a. (2010)
[66] (Rouse und Ogden, 2005, S. 310)

2.2.9. Skill

Der Begriff „Skill“ bezieht sich auf die Fähigkeiten des Spielers, welche von dem Spiel besonders gefordert werden. Stimmen die Fähigkeiten des Spielers und die Schwierigkeit des Spieles überein, fühlt sich der Spieler heraus- aber nicht überfordert. Dabei setzen die meisten Spiele mehrere Fähigkeiten voraus. Grob lassen sich diese Fähigkeiten nach Schell[67] in die folgende Kategorien abgrenzen:

Physische Fähigkeiten Stärke, Ausdauer, Koordination und Schnelligkeit gehören zu den physischen Fähigkeiten und sind entsprechend in sportlichen Spielen von Wichtigkeit. Einen Spiele-Controller mit einer bestimmten Präzision und Reaktionsfähigkeit bedienen zu können gehört ebenfalls in diesen Bereich.

Mentale Fähigkeiten Zu dieser Kategorie zählen Gedächtnis, Beobachtungsgabe und das Lösen von Rätseln.

Soziale Fähigkeiten Das Hineinversetzen in einen Gegner, um entweder seine Handlung vorherzusehen oder ihn gezielt in die Irre zu führen, sowie die Koordination mit Mitspielern fallen in diesen Bereich.

In Gesellschaftsspielen können alle drei vorgestellten Kategorien eine Rolle spielen, wobei bei den physischen Fähigkeiten hauptsächlich die Schnelligkeit relevant ist. Mentale und soziale Fähigkeiten stellen den Schwerpunkt von Gesellschaftsspielen dar. Die Herausforderungen ergeben sich dabei aus dem Zusammenspiel der unterschiedlichen Komponenten und es soll kein Element geben, welches sich direkt auf die notwendigen Fähigkeiten auswirkt.

Salen und Zimmermann[68] beschäftigen sich weiterhin mit den psychologischen Mechanismen, welche einen Spieler besonders in ein Spiel eintauchen lassen. Dabei beziehen sie sich auf die von Csikszentmihalyi (2008) vorgestellten Theorien, welche nicht auf den Bereich der Spiele beschränkt sind. Dort wird ein sogenannter „Flow“ beschrieben, den Menschen in sehr unterschiedlichen Situation, wie z.B. während des Joggens, der Arbeit an einer Produktionslinie oder auch während einer Operation empfinden können. Das Erreichen dieses Zustandes durch ein Spiel wird als besonders erstrebenswert angesehen, da es zum einen als konzentriert und zum anderen als glücklicher Zustand beschrieben wird, bei dem Herausforderung und Belohnung in einem wünschenswerten Gleichgewicht zueinander stehen.

[67] (Schell, 2008, Seite 150)
[68] (Salen und Zimmerman, 2004, Vgl. S. 336 - 339)

2.2.10. Chance

Der Zufall ist ein wichtiger Bestandteil des Spielspaßes, da so unvorhergesehen Situationen entstehen können und das Spiel entsprechend interessanter wird. Bei Gesellschaftsspielen werden meist Würfel oder Karten eingesetzt, um das Zufallselement auszudrücken. Entweder kann der Zufall dabei bereits selbst einer der Kernbestandteile des Spieles, wie z.B. bei Poker, sein oder dazu dienen, das Spiel unvorhergesehen und in diesem Sinne auch abwechslungsreich zu gestalten. Bei Risiko müssen die Spiele ihre Strategien entsprechend ihrem Würfelglück anpassen. Im Rahmen der unterschiedlichen Wahrscheinlichkeiten und wie diese das sogenannte Balancing eines Spiel beeinflussen, existiert dabei eine Vielzahl an Literatur, welche sich wiederum an die Spiele-Designer richtet [69].

Ein weiterer psychologischer Aspekt des Zufalls liegt darin, dass ein Mensch sich tendenziell eher für die sichere, wenn auch nicht optimale Möglichkeit entscheidet. Schell[70] beschreibt ein Beispiel, bei dem Personen über ihre Entscheidung in einem Spiel mit einem Geldgewinn gefragt worden sind. Dabei konnten sich die Teilnehmer zwischen zwei Varianten entscheiden. Bei Spiel A gewinnen sie mit 66% 2400$, mit 33% 2500$ und mit 1% 0$. Bei Spiel B gewinnen sie mit 100% 2400$. Der Erwartungswert von Spiel A (2409$)liegt dabei leicht höher als der Erwartungswert von Spiel B (2400$), dennoch würden 82% der Teilnehmer Spiel B vorziehen. Dies wird darin begründet, dass Menschen versuchen negative Erfahrung wie das Bedauern einer Entscheidung zu vermeiden, und bereit sind für diese Sicherheit gewissen Abstriche in Kauf zu nehmen.

Im Rahmen der hier vorgestellten Definition soll es möglich sein, über das **Chance**-Element Wahrscheinlichkeitsbereiche zu definieren und auf diese Weise Würfel oder sonstige zufällige Ziehungen von Markern abbilden zu können. **Chance** stellt dabei kein eigenständiges Element dar, sondern muss immer in Zusammenhang mit einer **Operation** genutzt werden.

[69] (Brathwaite und Schreiber, 2009, Vgl. S. 69 - 73)
[70] (Schell, 2008, Seite 165-166)

2.2.11. GameState

Mit den vorgestellten Definitionen kann nur die Spielmechanik von Spielen, nicht aber ein bestimmter Zustand während eines Spieles beschrieben werden. Um diesen beschreiben zu können, müssen konkrete Ausprägungen der definierten Elemente innerhalb einer Hülle existieren können, die den aktuellen Zustand eines Spieles repräsentiert. Diese Hülle soll als **GameState** bezeichnet werden. In dem Sinne beschreibt die Spielmechanik, wie sich die ausgeprägten Elemente innerhalb des **GameStates** verändern können und so den Gesamtzustand eines Spieles abbilden.

An dieser Stelle soll angemerkt werden, dass der Zustand von dem Ablauf des Spieles getrennt ist. Innerhalb des Zustands wird nur festgehalten, an welchem Zeitpunkt des Spielablaufes sich der Zustand in diesem Moment befindet.

2.2.12. Zusammenfassung

In den vorherigen Abschnitten wurden die unterschiedlichen Sichten auf die Elemente der Spielmechanik diskutiert und ein erster Ansatz für die Begrifflichkeiten der in dieser Arbeit entwickelten domänenspezifischen Sprache vorgestellt. Diese hier erarbeiteten Definitionen sollen im nächsten Kapitel für die Analyse des Referenzbeispieles angewendet und im Anschluss diskutiert werden.

Die bisher definierten Begriffe sollen an dieser Stelle noch einmal kurz zusammengefasst werden:

Spaces: Der **Space** repräsentiert einen Bereich, welcher durch **Fields** genauer definiert und aufgeteilt wird.

Fields: Ein **Field** stellt ein bestimmtes Feld innerhalb des **Spaces** dar.

Visibility: Die **Visibility** bestimmt die Sichtbarkeit eines Elementes.

Objects: Ein **Object** stellt ein Objekt innerhalb eines Spieles dar.

Attribute: Ein **Attribute** repräsentiert ein einfaches Merkmal.

Actions: Eine **Action** repräsentiert eine durch einen Spieler aktiv ausgeführte Aktion. Eine **Action** kann zur Abbildung von komplexen Abläufen zusätzlich in **ActionStates** eingeteilt werden.

Operations: Eine **Operation** dient dazu, die unterschiedlichen Elemente der Spielmechanik einmalig und dauerhaft manipulieren zu können.

Implication: Eine **Implication** kann sich auf den Spielzustand auswirken, so lange die Vorbedingungen erfüllt werden.

Rules: **Rules** werden miteinander verknüpft und bilden gemeinsam eine Vorbedingung. Über diese Vorbedingungen kann auf bestimmte Situation während des Spiels reagiert werden.

Phases: **Phases** stellen eine Unterteilung des Spielablaufes dar.

PhaseStates: Ein **PhaseState** beschreibt einen bestimmten Unterzustand innerhalb einer **Phase**.

ActionStates: Ein **ActionState** beschreibt einen bestimmten Unterzustand innerhalb einer **Action**.

Actors: Ein **Actor** repräsentiert einen Spieler.

Selectors: Über einen **Selector** können bestimmte Elemente ausgewählt werden.

Timer: Garantiert, dass eine **Action** oder **Operation** im Rahmen eines definierten Zeitabschnittes ausgeführt wird.

Chance: Eine **Chance** repräsentiert eine Wahrscheinlichkeit und kann als Zufallsgenerator verwendet werden.

GameState: Der **GameState** enthält sämtliche Ausprägungen der Elemente eines Spieles und repräsentiert den aktuellen Spielzustand.

2.3. Klassifikation der Spielmechaniken von Gesellschaftsspielen

In dem vorangegangen Abschnitt 2.2 wurden die elementaren Bestandteile der Spielmechaniken vorgestellt, aus denen ein Gesellschaftsspiel bestehen kann. Durch das Zusammenspiel der Elemente ergeben sich unterschiedliche Spielerfahrungen. Dabei können zwischen den unterschiedlichsten Spielen Überschneidungen existieren, beispielsweise können bei „Mensch ärgere dich nicht“ und Monopoly die Spielfiguren in Abhängigkeit von einem Würfelwurf bewegt werden. Das Bewegen von Spielfiguren anhand eines Zufalls ist in diesem Sinne bereits eine Klasse von Spielmechanik.

An dieser Stelle soll eine Klassifizierung der populären Webseite Boardgamegeek.com (BGG) für Gesellschaftsspiele dienen [71]. Järvinen[72] argumentiert, dass die dort vorgestellte Klassifikation für eine analytische Betrachtung nicht ausreichend ist, da die jeweiligen Spielmechaniken nicht konkreten Elementen der Spiele zugeordnet werden können. Zusätzlich schließen sich die Spielmechaniken untereinander nicht aus, da sie sich nur auf einen Aspekt eines Spieles beziehen. Im Rahmen einer analytischen Klassifikation ist diese Mehrdeutigkeit zwar problematisch, doch die dort vorgeschlagene Definition spiegelt die für diese Arbeit relevante Domäne deutlich besser wieder als die von Järvinen [73] vorgestellte Klassifikation.

Zunächst sollen jene Spielmechaniken vorgestellt werden, welche im Rahmen dieser Arbeit relevant sind. Eine vollständige Liste der insgesamt 47 Kategorien, findet sich auf BGG [74].

Area Control / Area Influence: Diese Mechanik spielt eine Rolle wenn die Kontrolle eines Bereiches einen wichtigen Bestandteil des Spieles darstellt.

Area Movement: Bezeichnet Spiele, bei denen die Felder beliebige Größen annehmen können und die Bewegungen dadurch definiert werden, dass die Felder aneinander angrenzen.

Auction/Bidding: Diese Mechanik bezeichnet Spiele, bei denen auf irgendeine Weise mit anderen Spielern im Rahmen einer Auktion oder Versteigerung konkurriert wird. Meist müssen die Spieler abschätzen, ob die Vorteile des Zuschlages die Kosten aufwiegen.

Card Drafting: Bezeichnet Spiele, bei denen die Teilnehmer Karten aus einem gemeinsamen Pool beziehen um kurzfristige Vorteile zu erhalten, oder bei der eine bestimmte

[71] http://boardgamegeek.com/browse/boardgamemechanic
[72] (Järvinen, 2008, Vgl. S. 251)
[73] (Järvinen, 2008, Vgl. 393-394)
[74] http://boardgamegeek.com/browse/boardgamemechanic Eine Beschreibung erhält man, nachdem auf eine der Kategorien geklickt wird

Kombination von Karten gesammelt werden muss, um das Ziel des Spieles zu erreichen. Spiele, bei denen Karten einfach nur von einem Stapel gezogen werden, gehören nicht in diese Kategorie!

Dice Rolling: In diesem Spielen werden Würfel als Zufallsgenerator eingesetzt.

Roll / Spin and Move: Bei diesen Spielen bewegen die Spieler Spielemente in Abhängigkeit eines Zufallselements.

Set Collection: Bei dieser Mechanik ist es essenziell, dass die Spieler einen Satz von Elementen sammeln.

Trading: Spiele, bei denen die Spieler untereinander tauschen können.

Variable Phase Order: Bezeichnet den Umstand, dass die Spielphasen in jeweils unterschiedlicher Reihenfolge ausgeführt werden können.

Variable Player Powers: Bei dieser Mechanik können die Spieler im Laufe des Spieles unterschiedliche Fähigkeiten.

„Monopoly“ lässt sich nach BGG in folgende Spielmechaniken einteilen:

Monopoly

Auction/Bidding: Wenn ein Spieler als eine Straße betritt und diese nicht kaufen möchte, können die anderen Spieler diese im Rahmen einer Auktion kaufen.

Dice Rolling: Würfel werden genutzt um die Anzahl der Felder festzulegen, die sich ein Spieler bewegen darf.

Roll / Spin and Move: Anhand eines Würfelwurfes muss ein Spieler seine Spielfigur bewegen.

Set Collection: Es ergeben sich wichtige Vorteile für die Spieler, wenn sie alle Straßen eines Typs gesammelt haben. Erst dann können sie Häuser und Hotels errichten.

Trading: Die Spieler können Straßen untereinander handeln.

„Risiko“ kann wiederum in diese Kategorien eingeteilt werden:

Area Control / Area Influence: Wenn ein Spieler ein Land mit einer Armee betritt, gilt dieses als erobert. Die eroberten Ländern sind ausschlaggebend für die Anzahl der Armeen, die ein Spieler erhält.

Area Movement: Die Armeen können von einem Land zu einem angrenzenden Land bewegt werden.

Dice Rolling: Der Kampf zwischen zwei Armeen wird mithilfe von Würfeln abgebildet.

Set Collection: Wenn die Spieler einen bestimmen Satz von Karten gesammelt haben, können sie durch diese zusätzliche Armeen anfordern und so einen wichtigen Vorteil gegenüber den Mitspielern erhalten.

„Risiko“ und „Monopoly“ lassen sich in dem Sinne vergleichen, dass sie beide Würfel als Zufallselement nutzen. Weiterhin stellt es bei beiden Spielen ein wichtigen Bestandteil dar, einen gewissen Satz von Elementen zu sammeln um einen Vorteil gegenüber den Mitspielern zu gewinnen.

2.4. Fazit

In diesem Kapitel wurden die Grundlagen domänenspezifischer Sprachen vorgestellt. Anschließend wurden die Elemente der Spielmechanik aus unterschiedlichen Blickwinkeln betrachtet, mit dem Ziel, eine eigene Definition für Gesellschaftsspiele zu erarbeiten. Diese Definition stellt eine erste Grundlage für die in dieser Arbeit zu entwickelnden Sprache dar. Abschließend wurde eine Klassifikation vorgestellt, welche es ermöglicht Gesellschaftsspiele anhand deren Spielmechaniken miteinander zu vergleichen oder voneinander abzugrenzen.

3. Analyse

3.1. Einleitung

In diesem Kapitel soll zunächst ein Referenzbeispiel vorgestellt und anhand der ausgearbeiteten Definition analysiert und diskutiert werden. Danach sollen Anforderungen an die zu entwickelnde Sprache sowie die dazugehörigen Werkzeuge und Systeme formuliert werden. Diese Anforderungen sollen die Grundlagen für die in den Kapiteln 4 und 5 ausgearbeiteten Inhalte darstellen. Folgend sollen zwei vergleichbare Arbeiten präsentiert und von dem hier vorgestellten Ansatz abgegrenzt werden.

3.2. Referenzbeispiel: Ohne Furcht und Adel

3.2.1. Hintergrund

Das Spiel „Ohne Furcht und Adel“ (OFuA) erschien im Jahr 2000, wurde von dem Autor Bruno Faidutti [1] entwickelt und wird in Deutschland durch den Hans im Glück-Verlag [2] vertrieben. Das Spiel gewann verschiedene Preisen und wurde im Jahr 2000 zum „Spiel des Jahres“ nominiert.

An dieser Stelle soll die Spielmechanik erläutert werden, das vollständige Regelwerk ist im Anhang in Abschnitt B.1 zu finden. Die Mechanik soll dabei im weiteren Verlauf zur Veranschaulichung von Beispielen in Bezug auf die domänenspezifische Sprache und das Design herangezogen werden.

[1] http://www.faidutti.com/
[2] http://www.hans-im-glueck.de/

3.2.2. Vorstellung

„Ohne Furcht und Adel“ ist ein Spiel für zwei bis sieben Spieler. Das Ziel des Spieles ist es, möglichst schnell eine Stadt zu errichten und dabei die meisten Siegpunkte zu erringen. Das Spielende tritt ein, wenn ein Spieler das achte Bauwerk errichtet hat. Als Spielmaterial dienen 65 unterschiedliche Bauwerk- sowie acht Charakterkarten und 30 Gold-Stücke. Zum Anfang eines Spieles erhalten die Spieler eine feste Anzahl an Gold sowie eine feste Anzahl von zufällig gezogenen Bauwerkkarten. In jeder Spielrunde werden zuerst die acht Charakterkarten in Abhängigkeit von der Spieleranzahl nach einem bestimmten Muster ausgewählt. Dabei ist es zuerst geheim, welcher Spieler sich welchen Charakter ausgesucht hat. Nachdem die Charakter-Auswahl abgeschlossen ist, werden die Charaktere in einer vorbestimmten Reihenfolge aufgerufen und der Spieler, der diesen Charakter ausgewählt hat, ist am Zug. Nun kann dieser Spieler zum einen die Fähigkeiten seines Charakters nutzen, Bauwerke errichte (indem er eine Bauwerkkarte von der Hand vor sich legt und die notwendige Höhe Gold bezahlt) und entweder Gold nehmen oder Bauwerkkarten ziehen. Anschließend wird der nächste Charakter aufgerufen. Sind alle Charaktere einmal benannt worden und keiner der Spieler hat bereits mehr als acht Bauwerk errichtet, werden alle Charakterkarten gemischt und die nächste Runde beginnt erneut mit der Charakterauswahl. Die Charaktere haben dabei entweder konstruktive oder destruktive Fähigkeiten, wobei die Angriffs-Fähigkeiten nur auf einen noch nicht ausgerufenen Charakter und nicht gezielt auf einen bestimmten Spieler angewendet werden können. Zusätzlich ergeben sich bei manchen Charakteren Synergieeffekte mit den ausliegenden Bauwerken und ein Sechstel der 65 Bauwerke besitzen Spezialeigenschaften die ihrem Besitzer einen Vorteil verschaffen. Die Herausforderung des Spieles liegt dabei in der richtigen Auswahl der Charaktere, da sich der Spieler in seine Gegner versetzen muss um ihre nächsten Schritte abzuschätzen um entweder die eigene Position zu verbessern, den Gegnern zu schaden oder sich im Gegenzug vor Schaden schützen. Abbildung 3.1 zeigt einige Elemente des Gesellschaftsspieles.

Der Ablauf des Spieles wird im Rahmen einer Beispielrunde im Anhang unter Abschnitt B.2 exemplarisch beschrieben. Da die für die Entwicklung der Sprache und für das Design vorgestellten Beispiele sich auf das Referenzbeispiel beziehen, sollte es bei Verständnisschwierigkeiten herangezogen werden. Folgend soll die Mechanik des Spieles und die einzelnen Elemente analysiert werden.

3.2.3. Spielmechanik

Einleitung

Zunächst soll das Referenzbeispiel auf Basis der in Abschnitt 2.3 vorgestellten Klassifizierung eingeordnet werden. Anschließend soll es anhand der in Abschnitt 2.2 ausgearbeiteten

Abbildung 3.1.: Ohne Furcht und Adel

Definition analysiert werden. Dabei ist es nicht das Ziel, OFuA vollständig zu analysieren, sondern beispielhaft zu zeigen, wie die Elemente einer konkreten Spielmechanik durch die vorgestellte Definition eingeordnet werden können.

Klassifizierung

Nach der durch in Abschnitt 2.3 beschriebenen Klassifizierung von Spielmechaniken durch BGG lässt sich OFuA folgenden Spielmechanik-Klassen [3] zuordnen:

Card Drafting: Grundlegend für ein siegreiches Spiel ist, dass ein bestimmter Mix aus den unterschiedlichen Typen von Bauwerken errichtet wird, um die Stärken der unterschiedlichen Charaktere optimal einsetzen zu können.

Variable Phase Order: Die Reihenfolge, in der ein Spieler seinen Zug durchführen kann, ist von dem Charakter abhängig, den er ausgewählt hat.

Variable Player Powers: Die Möglichkeiten, die ein Spieler hat, um auf das Spiel Einfluss zu nehmen, sind maßgeblich von dem Charakter abhängig, den er ausgewählt hat. Beispielsweise kann der Söldner eingesetzt werden, um die Gegner zu stören und der Händler bringt viel Gold ein, um besonders teure Bauwerke errichten zu können.

Spaces und Fields

Bei dem Referenzbeispiel lassen sich zwei **Space**-Typen von einander unterscheiden:

[3] Eine Beschreibung der hier aufgelisteten Klassen findet sich ebenfalls in Abschnitt 2.3

1. Der **Space**, der die allgemeinen Bereiche für das Spiel enthält und direkt dem **GameState** zugeordnet ist. Es existieren für jeden der folgenden Bereich je ein **Field**:
    - Ein **Field**, welche die zu ziehenden Bauwerk-Karten enthält. Die enthaltenen Bauwerke können von keinem Spieler gesehen werden.
    - Ein Bereich, welcher für die abgelegten Bauwerk-Karten gedacht ist. Die enthaltenen Bauwerke können von allen gesehen werden.
    - Der Bereich, in dem sich die Charaktere befinden, die ausgewählt werden können. Die Charaktere sind für keinen Spieler sichtbar.
    - Das **Field**, welches die verdeckt abgelegten Charaktere enthält. Sie können von keinem Spieler gesehen werden.
    - Ein **Field**, welches die offenen und für diese Spielrunde nicht auswählbaren Charaktere enthält. Diese Charaktere sind für alle sichtbar.
2. Der **Space**, den jeder Spieler besitzt und folgende **Fields** enthält:
    - Ein Bereich, der die Bauwerkkarten auf der Hand repräsentiert. Nur für den jeweiligen Spieler sichtbar.
    - Ein **Field**, welches die Bauwerke enthält, die durch den Spieler errichtet worden sind. Für alle Spieler sichtbar.
    - Das **Field**, welches die Bauwerke enthält, aus denen er auswählen kann, wenn er aus den gezogenen Bauwerk-Karten nicht alle behalten darf. Nur für den jeweiligen Spieler sichtbar.
    - Ein Bereich, in dem die für den Spieler auszuwählenden Charaktere enthalten sind. Nur für den jeweiligen Spieler sichtbar.
    - Der Bereich, der die ausgewählten und noch nicht gespielten Charaktere enthält. Nur für den jeweiligen Spieler sichtbar.
    - Das **Field**, dem die bereits durch den Spieler gespielten Charaktere zugehören. Für alle Spieler sichtbar.
    - Ein Bereich für den Charakter, dessen Spielphase in diesem Moment durchgeführt wird. Für alle Spieler sichtbar.

Es wurden die beiden für OFuA notwendigen **Spaces** vorgestellt. Für die unterschiedlichen **Fields** wurde neben der konzeptionellen Betrachtung definiert, für wen sie sichtbar sind. Diese Sichtbarkeit wir anhand des **Visibility**-Elementes beschrieben.

Objects

Bei OFuA lassen sich die Karten in zwei grundsätzliche Typen unterteilen: In Charaktere und Bauwerke. Wiederum existieren sieben unterschiedliche Charaktere und fünf unterschiedliche Bauwerk-Arten. Die Bauwerke unterscheiden sich durch ihre Farbe, Kosten und Siegpunkte. Diese Merkmale werden als **Attributes** abgebildet.

Jeden Charakter gibt es einmal innerhalb des Spieles, die unterschiedlichen Bauwerke können öfters vorhanden sein. Die meisten Charaktere und einige der lila Spezial-Bauwerke erlauben den Spielern, bestimmte **Actions** durchzuführen.

Attributes

Bei den **Attributes** können unterschiedliche Typen identifiziert werden. Die Kosten und die Siegpunkte der Bauwerk-**Objects** können als ein positiver Integer betrachtet werden. Die Farben der Bauwerke beschränken sich hingegen auf eine definierte Menge und können als Enum repräsentiert werden. Zudem gibt es ein **Attribute**, welches festlegt, ob ein Charakter durch den Meuchler ermordet wurde und entsprechend am Leben ist oder nicht. Dieses Merkmal besteht aus einem Booleschen Typ.

Actions und Operations

Die **Actions** verteilen sich zum einen auf die vorgestellten Charakter- und Bauwerk-**Objects** und zusätzlich auf die Spielphase. Innerhalb jeder Spielphase kann ein Spieler entweder Bauwerk-Karten ziehen oder Gold nehmen. Dies sind ebenso **Actions**, können aber eindeutig einer **Phase** und nicht einem speziellen Charakter zugeordnet werden.

Die Auswirkungen der **Actions** auf den Spielzustand sind sehr unterschiedlich. Wenn sich ein Spieler Gold nimmt, wird einfach sein Vorrat erhöht. Wenn ein Spieler ein Bauwerk baut, wird sein Gold-Vorrat entsprechend der Kosten verringert und von dem Handkarten-**Field** in das **Field** verschoben, welche die errichteten Bauwerke des Spieles repräsentiert. Diese beiden **Actions** lassen sich als eine Sequenz von **Operations** betrachten.

Zusätzlich können auch **Actions** definiert werden, die aus mehreren **ActionStates** bestehen scheinen und Entscheidungen in Abhängigkeit des Spielzustandes enthalten. Wenn sich ein Spieler in der Spielphase Karten nimmt und mehr Karten ziehen kann, als er behalten darf, muss er aus diesen Karten auswählen. Wenn er allerdings alle Karten nehmen darf, die er zieht, muss er keine auswählen. Dieser Ablauf wird durch zwei unterschiedliche **ActionStates** abgebildet.

Phases und PhaseStates

OFuA lässt sich zunächst einmal in die **Phase** der Charakterauswahl und die jeweiligen **Phases** unterteilen, welche die Spielphase eines Charakters darstellen. Zusätzlich existiert eine Vorbereitungs-**Phase** zu Beginn des Spieles, und nachdem die Siegbedingung erfüllt wurde, eine abschließende **Phase**, in der ein Gewinner bestimmt werden muss. Der Ablauf kann durch einen Zustandsautomaten, wie in Abbildung 3.2 zu sehen, dargestellt werden.

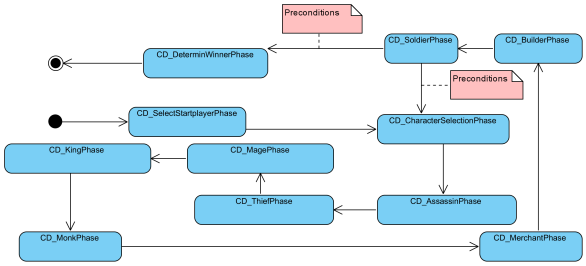


Abbildung 3.2.: Spielablauf

Die Spielphase eines Charakters lässt sich wiederum in die in Abbildung 3.3 abgebildeten **PhaseStates** unterteilen.

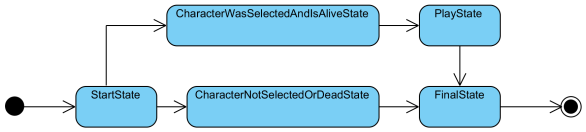


Abbildung 3.3.: PhaseStates einer Charakterphase

Diese Darstellung zeigt, wie gut sich die Abläufe anhand von Zustandsautomaten abbilden lassen.

Actors und Implications

Zur Beschreibung des Referenzbeispieles reicht ein einziger **Actor**-Typ aus. Die Spieler unterscheiden sich nur ja ihren unterschiedlichen Ausprägungen der **Attributes**. Wenn ein **Actor** Karten nimmt, kann er eine Anzahl von Karten ziehen und wiederum eine Anzahl von ihnen behalten. Beide Anzahlen können sich im Spiel verändern. Wiederum kann eine bestimmte Anzahl von Bauwerken innerhalb einer Spielphase errichtet werden und es muss zusätzlich festgehalten werden, wie viele sie schon errichtet haben. Zuletzt ist der aktuelle Gold-Vorrat und die Gesamtanzahl der Siegpunkte ein **Attribute** des **Actors**.

Zusätzlich besitzt jeder **Actor**, wie bereits auf Seite 53 beschrieben, einen eigenen **Space**. Der **Space** enthält dir dort beschriebenen **Fields**. Relevant ist die Beziehung, die zwischen dem **Actor** und dem **Space**, den darin enthaltenen **Fields** und den darin enthaltenen Bauwerk-**Objects**, besteht. Der **Actor** besitzt diese Elemente. Über diese Besitz-Relation wurde unter anderem bereits die Sichtbarkeit der **Fields** beschrieben.

Es lässt sich eine logische Beziehung zwischen den Siegpunkten eines Spielers und den Siegpunkten der errichteten Bauwerke feststellen. Die Summe der Siegpunkte der Bauwerke ergibt die aktuelle Siegpunkte-Zahl des Spielers. Diese Wechselwirkung wird durch eine **Implication** abgebildet. So lange das Bauwerk als errichtet gilt, in dem es sich in dem entsprechenden **Field** befindet, wirkt sich der Effekt aus. Ansonsten werden die Auswirkungen rückgängig gemacht. Dies wäre z.B. der Fall, wenn ein Bauwerk durch den Söldner zerstört wird.

An dieser Stelle soll ein Beispiel für das **Selector**-Element vorgestellt werden. Es wurde beschrieben, dass der Gold-Vorrat im Rahmen einer „Gold nehmen“-**Action** durch eine **Operation** erhöht werden soll. Damit die **Operation** durchgeführt werden kann, muss ein **Actor** und ein zu veränderndes **Attribute** ausgewählt werden. Dies kann anhand eines **Selectors** über die Beziehung zwischen dem **Actor**, der die **Action** ausführt, und dem das **Attribute** gehört, welches manipuliert werden soll, durchgeführt werden.

Rules und Timer

In OFuA lassen sich viele unterschiedliche Regel-Typen erkennen. An dieser Stelle soll nicht versucht werden, einzelne Regel zu identifizieren, sondern unterschiedliche Regel-Gruppen von einander abzugrenzen:

- Zunächst existieren bestimmte **Rules**, die vorgeben, wie der Spielablauf gewährleistet werden soll. Sie beziehen sich auf die **Phases** und die **PhaseStates**. Wie bereits beschrieben, kann eine **Action** in **ActionStates** unterteilt werden. Hier wird der Ablauf ebenso anhand von **Rules** geregelt.
- Weiterhin gibt es **Rules**, die bestimmen, wann eine **Action** ausgeführt werden darf.
- Für die **Implications** wird anhand von **Rules** definiert, wann sie sich auswirken.

Sämtliche **Rules** lassen sich diesen Elementen zuordnen.

Das **Timer**-Element findet in der Spielmechanik von OFuA keine Anwendung.

Chance

Bei Ohne Furcht und Adel existieren zwar Zufallsmomente, da zu Beginn des Spieles die Bauwerk-Karten und die Charakter-Karten ebenso nach jedem Durchlauf gemischt werden, aber es ist nicht notwendig, diese als **Chance**-Element abzubilden. Das **Chance**-Element dient nur dazu, um direkte Zufälle wie z.B. einen Würfelwurf, zu simulieren.

3.2.4. Zusammenfassung

Das Referenzbeispiel wurde zunächst vorgestellt, danach anhand der Spielmechanik klassifiziert und anschließend mithilfe der in Abschnitt 2.2 vorgestellten Definitionen analysiert und diskutiert. Das Spiel OFuA konnte dabei klar in die unterschiedlichen Elemente aufgetrennt werden. Auf diese Weise wurde das Verständnis für die Definitionen und ihre Anwendungsmöglichkeiten vertieft.

3.3. Anforderungen

3.3.1. Einleitung

In diesem Abschnitt sollen Anforderungen an die Entwicklung der Sprache und die unterstützenden Werkzeuge und deren Komponenten definiert werden. Dabei soll zunächst eine Vision vorgestellt werden, welche das Potenzial einer vollständig ausgereiften Lösung vermittelt. Anschließend soll ein Minimal-Entwurf die Anforderungen formulieren, die für die grundsätzliche Realisierung der Vision notwendig sind. Diese Anforderungen sollen in Kapitel 4 und Kapitel 5 aufgegriffen werden.

3.3.2. Vision

domänenspezifische Sprache

Die domänenspezifische Sprache soll möglichst viele Gesellschaftsspiele unterstützen. Ausgeschlossen sollen Spiele sein, welche entweder einen starken sozialen Schwerpunkt aufweisen (wie z.B. das Beschreiben, Malen oder Vorspielen von Begriffen), besonders auf physische Geschicklichkeit aufbauen (wie z.B. das Herausziehen von Blöcken aus einem Turm) oder bei denen ein unabhängiger Spiel-Leiter subjektiv das Spielgeschehen bewerten und

den Spielablauf weiter steuern muss, ohne dass dies von dem System übernommen werden kann.

Weiterhin soll die domänenspezifische Sprache folgende Eigenschaften aufweisen:

Konformität Die Elemente der domänenspezifischen Sprache müssen die Kernkonzepte der Domäne abbilden können.

Orthogonalität Jedes Element der Sprache sollte genau zur Beschreibung eines Konzeptes innerhalb der Domäne zugeordnet werden können.

Unterstützung Für das Arbeiten mit einer domänenspezifischen Sprache sollten in der Regel Werkzeuge angeboten werden, welche das Erstellen, Bearbeiten, Debugging und Kompilieren der Sprache in ein maschinenlesbares Format und der Ausführung auf eine Runtime unterstützen.

Erweiterbarkeit Die domänenspezifische Sprache und die dazugehörigen Werkzeuge sollten um zusätzliche Sprach-Elemente und Konzepte erweitert werden können.

Langlebigkeit Der Einsatz der domänenspezifische Sprache sollte sich für eine langen Zeitraum lohnen, um den Aufwand der Entwicklung der Sprache und der zusätzlichen Werkzeuge rechtfertigen zu können.

Entwicklungsumgebung

Das Ziel soll die Entwicklung eines Editors sein, welcher es ermöglicht, die Elemente der hier zu entwickelnden DSL grafisch zu verwalten. Dabei soll die Verwaltung sich nach jenen Methoden richten, wie sie auch bei der Entwicklungs von Computerspielen eingesetzt werden. Hierfür sollte eine Studie aktueller Entwicklungsoberflächen angefertigt werden, ein optimales UI-Konzept erarbeitet und umgesetzt werden.

Damit die Arbeit an einem Spiel nicht durch das Wechseln zwischen mehreren Programmen unnötigerweise gestört wird, sollen sämtliche Komponenten der Entwicklungsphase in die Entwicklungsoberfläche integriert sein. Für den Entwickler soll also die Trennung von graphischer Oberfläche, DSL-Processor und einer Test-Runtime transparent sein. Zusätzlich soll es eine Produktiv-Runtime geben, welche nicht in der Entwicklungsoberfläche integriert sein soll. Über die Entwicklungsoberfläche sollen sich also die definierten Spiele direkt testen und auch für eine produktive Umgebung generieren lassen.

Zuletzt soll die Entwicklungsoberfläche über über einen ausgereiften Hilfebereich verfügen, der aus einer Dokumentation, einer Referenz, einem sogenannten Kochbuch mit Code-Beispielen und Tutorials bestehen, welche den Einstieg in die Entwicklung mit der domänenspezifschen Sprache erleichtern sollen.

Test-Runtime

Die Test-Runtime soll nahtlos in die Entwicklungsumgebung integriert sein. Daher soll sie lokal auf dem Entwicklungsrechner ausgeführt werden können. Damit der aktuelle Entwicklungsstand sofort getestet werden kann, soll pro teilnehmendem Spieler ein rudimentäres graphisches Interface generiert werden. Da ein Spieler jeweils definierte Handlungsmöglichkeiten hat, soll für jeden Spieler ein eigenes Fenster im Editor erscheinen, welches seine Sicht auf das Spielgeschehen darstellt. In einem zusätzlichen Fenster soll der Gesamtspielzustand erscheinen.

Zusätzlich solle die Test-Runtime über einen Debugger angebunden werden können. Der Debugger soll es dabei ermöglichen, die Ausführung des Spielcodes auf die Ebene der domänenspezifischen Sprache zu reduzieren. Dabei sollen nicht wie in einem traditionellen Debugger die einzelnen ausgeführten Programmzeilen als „kleinste Einheit“ gelten, sondern die Schritte innerhalb der Spielmechanik. Es soll also möglich sein, die einzelnen **Operations** im Debugger kontrolliert auszuführen. Die Ansichten für das Debugging und die Analyse sollen sich an den Entwicklungs-Ansichten orientieren. Es soll zu jeder Zeit deutlich sein, welche **Rules** durch den aktuellen Spielzustand erfüllt sind und welche nicht. Zusätzlich soll es möglich sein, Breakpoints zu setzen und die Spielmechanik an dieser Stelle anzuhalten.

Die Test-Runtime soll über ein KI-Modul verfügen, welches es ermöglicht, für die definierten Spiele eine grundlegende KI zu generieren. Anschließend soll es möglich sein, diese KI weiter auszuarbeiten und zu optimieren.

Zuletzt soll es möglich sein, einen durch eine Runtime erstellten Dump eines Spieles zu laden. Dieser Dump soll Teil eines Fehlerberichtes sein, den ein Spieler nach einem Absturz der Runtime verschicken kann, um bei der Behebung des Problems zu helfen. Anhand des Dumps soll es möglich sein, die Veränderungen des Spielzustandes zeitlich verfolgen zu können, um die Quelle des Fehlers besser lokalisieren zu können.

Produktiv-Runtime

Die Produktiv-Runtime soll auf den Plattformen PC, Android, iOS und im Web lauffähig sein. Entweder soll die Runtime selbst plattformunabhängig sein oder über ein Tool für diese Ziel-Plattformen übersetzt werden können, ohne dass aufwändige Anpassungen durchgeführt werden müssen. Zusätzlich soll es möglich sein, dass die Spiele auch plattformübergreifend gespielt werden können.

Da es sich bei Gesellschaftsspielen in der Regel um Mehrspieler-Spiele handelt, sollen die erstellten Spiele automatisch einen Netzwerkmodus unterstützen. Bei der Produktiv-Runtime

soll es zwei unterschiedliche Ausfühungs-Modi geben: Einen lokalen und einen internetgestützen Modus.

Der lokale Modus soll ohne eine ständige Internetverbindung lauffähig sein und Spiele über das W-LAN sowie den Einzelspieler-Modus mit KI-Gegnern unterstützen. Bei einem Mehrspieler-Spiel über den lokalen Modus soll ein Gerät als Server fungieren und den Spielablauf steuern.

Zum anderen soll ein internetgestützter Modus existieren, wobei sich die Runtime zu einem Server im Internet verbindet, auf dem das eigentliche Spiel ausgeführt wird. Dabei soll es einen live-Modus für das direkte gemeinsame Spielen und einen unterwegs-Modus für ein zeitversetztes Spielen geben.

Die Kommunikation zwischen den Spielern ist in verschiedenen Spielen ein wichtiger Bestandteil des Spieles. Daher sollen Text- Audio- und Video-Chat optional in die Spiele eingebunden werden können.

Die Produktiv-Runtime umfasst unter anderem die Dienste Sicherheit (Authentifizierung, etc.), die Kommunikation (Empfangen, Versenden von Nachrichten, etc.), Persistenz (Speichern der Spielzustände, etc.), Vermeidung von Cheating aber auch dem eigentlichen Spiel vorgelagerte Dienste wie z.B. das Matchmaking für den Mehrspieler-Modus. Diese unterschiedlichen Aufgaben sollen, wie allgemein üblich, sinnvoll auf voneinander getrennte Komponenten verteilt werden, um eine flexible Architektur zu gewährleisten. Weiterhin ist es notwendig, eine individuelle GUI für jedes Spiel zu erstellen.

Framework

Das Framework soll als Grundlage einer Spiel-Instanz dienen, welche durch eine Runtime ausgeführt werden kann. Der durch den DSL-Processor generierten Code soll von der Spiel-Instanz getrennt sein. Die Spiel-Instanz soll den Code laden können und so die gewünschte Verhaltensweise annehmen.

Weiterhin sollen die bestehenden Sprachkonzepte flexibel umgesetzt werden, so dass mögliche Erweiterungen der Sprache leichter implementiert werden können. Daher soll bei dem Entwurf der Architektur gezielt auf entsprechende Software-Patterns zurückgegriffen werden.

Da die Runtimes bei einem Absturz einen Fehlerbericht erzeugen können sollen, müssen die Aktionen des Spieles entsprechend aufgezeichnet werden.

3.3.3. Minimal-Entwurf

Da das in der Vision beschriebenen Szenario den Rahmen einer Masterarbeit um ein vielfaches übersteigt, soll an dieser Stelle der Entwurf auf jene Aspekte beschränkt werden, welche notwendig sind um eine allgemeine Durchführbarkeit des hier vorgestellten Vorhabens belegen zu können. Auf diese Weise sollen Anforderungen [4] formuliert werden, welche in den folgenden Kapiteln erfüllt werden sollen.

Anforderungen: domänenspezifische Sprache

Die Umsetzbarkeit der unterschiedlichen Spiel-Typen lässt sich nur im Rahmen einer eigenen Studie bewerten. Die hier vorgestellte Sprache soll jedoch die beschrieben Eigenschaften aufweisen: Konformität [5], Orthogonalität[6], Unterstützung[7], Erweiterbarkeit[8] und Langlebigkeit[9].

Anforderungen: Entwicklungsumgebung

Für die Entwicklungsumgebung soll gezeigt werden, auf welche grundsätzliche Weise eine Ansicht umgesetzt werden kann[10]. Weiterhin soll die Generierung durch den DSL-Processor exemplarisch vorgestellt werden[11]. Weiterhin soll diskutiert werden, wie die Debugging-Funktionalität implementiert werden könnte [12].

Anforderungen: Test-Runtime

Es soll ein Design für die Architektur einer Test-Runtime skizziert werden [13]. Es soll gezeigt werden, wie die Ansichten mehrerer Spieler innerhalb der Runtime modelliert werden können [14]. Im Rahmen eines Prototyps soll die grundsätzliche Machbarkeit demonstriert werden [15].

[4] Diese können im Anhang unter Abschnitt A.1 nachgeschlagen werden.
[5] Anforderung DSL-1
[6] Anforderung DSL-2
[7] Anforderung DSL-3
[8] Anforderung DSL-4
[9] Anforderung DSL-5
[10] Anforderung IDE-1
[11] Anforderung IDE-2
[12] Anforderung IDE-3
[13] Anforderung TR-1
[14] Anforderung TR-2
[15] Anforderung TR-3

Anforderungen: Framework

Für das Framework soll ein detailliertes Design die Abbildung der domänenspezifischen Sprache belegen [16]. Für jedes in Kapitel 4.3 vorgestellte Sprachkonzept soll ein entsprechender Designvorschlag erarbeitet werden [17].

Der Code, der durch den DSL-Processor generiert wird, soll von der Implementierung des Frameworks getrennt sein [18]. Damit die Durchführung eines lokalen und eines Netzwerk-Spiel sich nicht auf das Design des Frameworks auswirkt, soll die Kommunikation zwischen Runtime und Framework entsprechend transparent gestaltet sein [19].

3.4. Vergleichbare Arbeiten

3.4.1. Einleitung

An dieser Stelle soll angemerkt werden, dass im Rahmen einer Recherche keine vergleichbaren Arbeiten gefunden werden konnten, die auf die hier vorgestellte Weise eine domänenspezifische Sprache für Gesellschaftsspiele entwickeln. Da die Entwicklung einer domänenspezifischen Sprache stark an die Anforderungen der Domäne ausgerichtet ist, lassen sich die folgenden Arbeiten nur in dem Sinne vergleichen, dass sie domänenspezifischen Sprachen für den Bereich der Computerspiele beschreiben.

3.4.2. Eberos GML2D

Die von Hernandez und Ortega entwickelte Eberos Game Modeling Language 2D (Eberos GML2D) stellt eine domänenspezifische Sprache für die Implementierung von zweidimensionalen Spielen dar [20]. Um den Vorteil der domänenspezifischen Sprache messbar zu machen, wurden zwei Spieler einmal manuell und einmal unter Verwendung von GML2D umgesetzt. Auf diese Weise konnte nachgewiesen werden, dass der Einsatz von GML2D erheblich die Entwicklungszeit reduziert. Abbildung 3.4 zeigt die GML2D-Definition für das Spiel Pong.

Bei GML2D handelt es sich um eine graphische Sprache. Sie soll unabhängig von der auszuführenden Spiele-Plattform und ebenso unabhängig von GameEngines sein. Die Elemente der Sprache lassen sich in graphische Elemente, Entitäten und Verhalten einordnen.

[16] Anforderung FW-12
[17] Anforderungen FW-3, FW-4, FW-5, FW-6, FW-7, FW-8, FW-9, FW-10, FW-11
[18] Anforderung FW-1
[19] Anforderung FW-2
[20] siehe Hernandez und Ortega (2010)

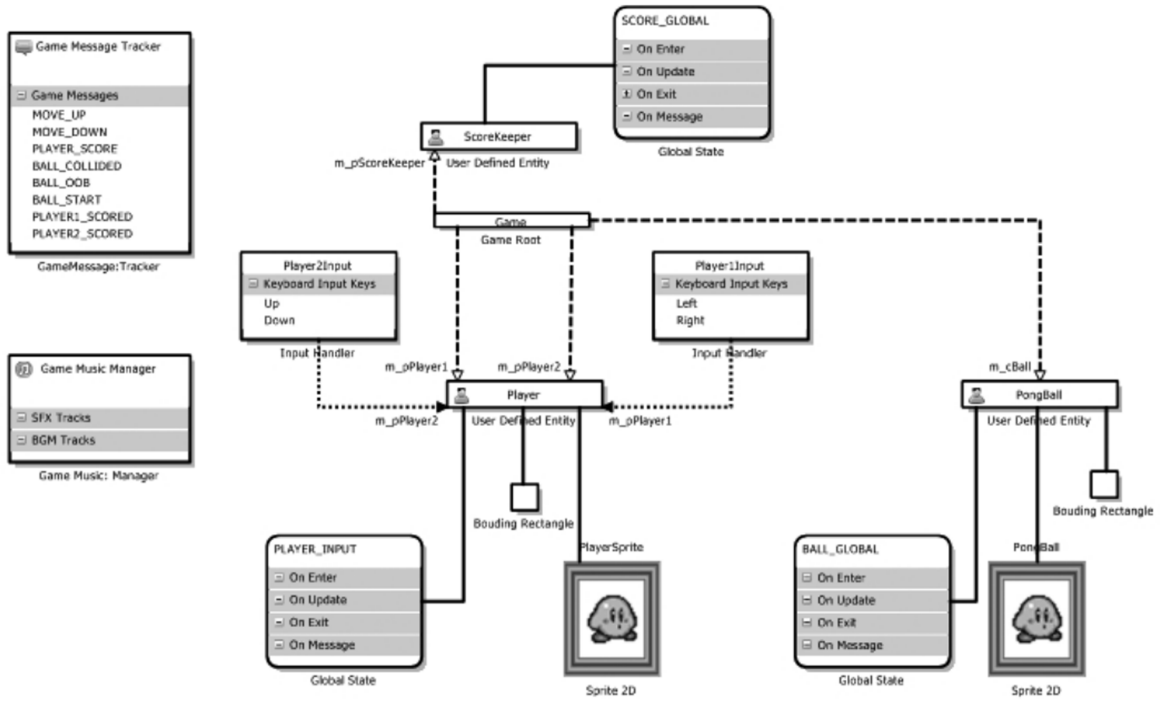


Abbildung 3.4.: GML2D

Sprachelemente

Ein **Sprite2D** stellt ein Bild dar und kann durch eine **SimpleAnimation** und **ComplexAnimation** animiert werden. Beide Animationen nutzen das **Frame2D**-Element.

Sogenannte **UserDefinedEntities** können graphische oder nicht graphische Entitäten innerhalb des Spieles repräsentieren. Graphische Entitäten können z.B. Spieler oder Gegner darstellen. Nicht graphische Entitäten können hingegen dazu verwendet werden, um Beschränkungen des Spielfeldes zu modellieren.

Über einen **InputHandler** können **UserDefinedEntities** entweder von einem Spieler oder durch das Spiel gesteuert werden. Ein **InputHandler** ermöglicht es, die Eingabe des Spielers zu verarbeiten. Die Eingabe von Tasten wird dabei in vier unterschiedliche Zustände (DOWN, RELEASED, WAS_PRESSED und IS_HOLDING) aufgeteilt, die separat behandelt werden können. Als Reaktion auf eine Eingabe wird eine **GameMessage** geschickt.

Jede Entität kann durch zwei Zustandsautomaten beschrieben werden, welche durch **GlobalStates** und **States** abgebildet werden. Das Verhalten des **State** kann von dem **GlobalState** abhängig sein. Der **GlobalState** hingegen verhält sich unabhängig von dem **State**. Die Zustandsübergänge werden unter der Verwendung einer **TransitionMessage** ausgelöst. Ein Übergang kann nur innerhalb eines Typs von Zustandsautomaten erfolgen.

Weiterhin lässt sich über **ActionScripts** das Verhalten für bestimmte Ereignisse anpassen. Es gibt vier unterschiedliche Arten von **ActionScripts**: **OnEnter**, **OnUpdate**, **OnExit** und **OnMessage**. Für jedes der Ereignisse kann ein Script geladen werden, welches die eigentliche Verarbeitung übernimmt. Auf diese Weise sollen sämtliche spielspezifischen Funktionalitäten, welche nicht durch in GML2D vorhanden sind, manuell implementiert werden. Dabei muss der Code der Ziel-Plattform entsprechen. Hernandez weist zwar darauf hin, dass durch dieses Design das Spiel-Modell an eine bestimmte Plattform oder GameEngine gebunden wird. Weiterhin wird argumentiert, dass die Funktionalität ansonsten nur auf die durch die Sprache vorgegebenen Konstrukte möglich wäre.

Über **BoundingRectangles** und **BoundingCircles** kann eine Kollisionserfassung definiert werden.

Zuletzt können unterschiedliche Controller modelliert werden. Dabei müssen alle Controller von der **GameRoot** abstammen. Der **GameMessage:Tracker** enthält sämtliche **GameMessages**-Typen und der **GameMusic:Manager** erlaubt die das Einbinden von Musik und Sound-Effekten.

Evaluation der Produktivität

Für die Evaluation wurde zum einen die Implementierung des Spieles Pong [21] zum einen mithilfe der GML2D und einmal manuell durchgeführt. Für ein weiteres Spiel „Space Katz“[22] wurde die durch Hernandez entwickelten Version mit GML2D ebenso mit einer manuell implementierten Variante verglichen.

Für die Umsetzung beider Spiele und für die **ActionScripts** wurde auf XNA/C# [23] aufgebaut. Als vergleichender Maßstab wurde zum einen die Entwicklungszeit und zum anderen die Anzahl der Code-Zeilen gewählt. Pong konnte anhand von GML2D 8,82% schneller und mit 29% weniger selbstgeschriebenen Code-Zeilen umgesetzt werden. Für Space Katz reduzierte sich die Entwicklungszeit unter der Verwendung von GML2D um 82,3% und die Anzahl der selbst geschriebenen Zeilen Code auf 86,4%.

Abgrenzung

Die graphische Repräsentation spielt für 2D-Spiele eine übergeordnete Rolle und legt wichtige Rahmenbedingungen für die Entwicklung fest. Die Entitäten werden durch Bilder und Ani-

[21] Gilt als das erste weltweit populäre Videospiel und wurde 1972 von Atari veröffentlicht. Es bildet die damalige Umsetzung eines Tischtennis-Spiels ab.

[22] Es handelt sich dabei um einen 2D-Sidescroller. Der Spieler muss ein Raumschiff durch Asteroiden steuern und gleichzeitig Feinde vernichten.

[23] siehe http://msdn.microsoft.com/en-us/centrum-xna.aspx

mationen dargestellt, und Thematiken wie die Kollisionserfassung stellen relevante Aspekte dar. Weiterhin handelt es sich bei GML2D im Gegensatz zu der hier vorgestellten DSL um eine graphische Sprache. Allerdings wurde in der Vision bereits angemerkt, dass eine graphische Entwicklungsoberfläche für Spieleentwickler bereitgestellt werden soll. Die hier vorgestellte DSL erlaubt nur die Definition der Spielmechanik; die Entwicklung einer GUI wird im Rahmen der Produktiv-Runtime abgedeckt. Bei GML2D werden die Spielmechanik und die GUI sowie die Eingabe enger miteinander verknüpft.

GML2D scheint eher den Entwicklungsprozess zu vereinfachen, indem die Elemente mit Verhalten verknüpft und mit Graphiken, Musik und Sound-Effekten angereichert werden können. Für die Spielmechanik existiert ein struktureller Rahmen. Das eigentliche Verhalten selbst kann über die **ActionScripts** realisiert werden. Diese müssen weiterhin manuell und für eine bestimmte Zielumgebung implementiert werden. An dieser Stelle kann argumentiert werden, dass diese Funktionalität spielspezifisch, und in der Regel eigens umgesetzt werden muss. Allerdings wird auf diese Weise die Definition des Spieles an eine bestimmte Plattform gekoppelt.

Die in dieser Arbeit vorgestellte Sprache ermöglicht hingegen eine vollständige Definition eines Gesellschaftsspieles, wobei es nicht notwendig ist, für spielspezifische Funktionalität eigenen Code zu implementieren. Sie ist zudem vollständig plattformunabhängig.

Interessant hingegen ist die Evaluation, welche anhand des Space Katz-Beispiel zeigt, dass die Entwicklung maßgeblich erleichtert wurde. Für die hier vorgestellte Sprache soll keine Evaluation im Rahmen dieser Arbeit durchgeführt werden.

3.4.3. LudoCore

LudoCore[24] soll Spiele-Designer während der Entwicklungsphase unterstützen und verknüpft die Themenbereiche der Spiele und der KI. Es ermöglicht, Anfragen an die Spielmechanik zu zu stellen und so bestimmte Aussagen formulieren zu können. Weiterhin existiert eine graphische Oberfläche, siehe Abbildung 3.5, welche die Ausführung der Scripte erlaubt.

Das Ziel von LudoCore ist es, die Entwicklung der abstrakten Spielmechanik gezielt zu unterstützen. Dabei spielt besonders die inkrementelle Ausarbeitung des zentralen Regelwerkes eine wichtige Rolle. Dieser iterative Ansatz steht dabei im Kontrast zu dem Ziel der formalen Verifikation. Dies ist darauf zurückzuführen, dass sich die zu überprüfenden Eigenschaften ständig verändern. In diesem Sinne soll LudoCore gerade dabei helfen, die für ein Spiel

[24] siehe Smith u. a. (2010)

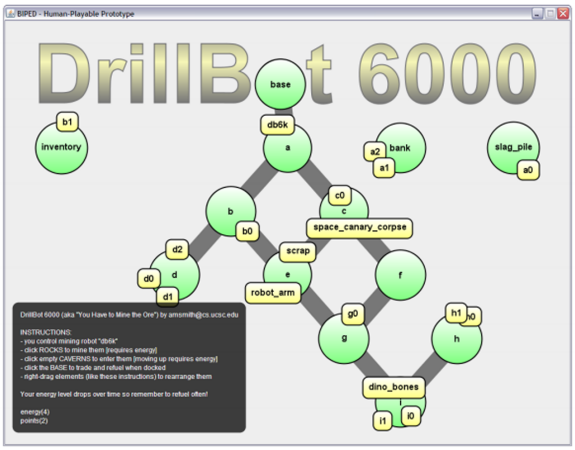


Abbildung 3.5.: graphische Oberfläche eines in LudoCore definierten Spieles

gewünschten Eigenschaften der Spielmechanik zu finden. Als Bindeglied zwischen der Definition der Spielmechanik und der Logik basiert LudoCore auf der logischen Sprache Event Calculus.

Event Calculus

Der diskrete Event Calculus (EC) basiert auf Fluents (Prädikate, welche sich im Laufe der Zeit verändern können) und Events, welche zu einem bestimmten ganzzahligen Zeitpunkt ausgelöst werden und die Wahrheitswerte der Fluents verändern können. Es existieren folgende Prädikate:

happens(E,T): Definiert, dass ein Event E zu einem Zeitpunkt T ausgelöst wird

holds_at(F,T): Beschreibt, dass ein Fluent F zu dem Zeitpunkt T wahr ist

initiates(E, F, T): Löst ein Event E aus, wenn ein Fluent F wahr wird

terminates(E, F, T): Löst ein Event E aus, wenn ein Fluent F falsch wird

Um das logische Modell der EC umzusetzen, wurde ein Verfahren namens „answer set programming“ (ASP) eingesetzt. Die Kombination von EC und ASP soll sich dabei besonders für die Beschreibung von KI in Spielen eignen, da ASP eine schnelle logische Auswertung ermöglicht und über EC eine solide Repräsentation der Spielwelt definiert werden kann.

Logische GameEngine

Basierend auf dem EC lassen sich in LudoCore Zustände, Events und Auswirkungen beschreiben. Dabei existieren unterschiedliche Definitionsebenen. Mit dem Prädikat `game_state` lassen sich globale Zustände und mit dem `game_event` das Eintreten eines Events formulieren. Auf diese Weise kann über `game_event(heals(A)) <- agent(A)` formuliert werden, dass ein Agent die Heil-Aktion durchführt.

Weiterhin existieren das `possible`- und das `conflicts`-Prädikat, mit dem bestimmte Events entweder erlaubt oder verboten werden. Über die Definition von `possible(heals(A)) <- home(A)` könnte ausgedrückt werden, dass sich der Agent an seinem Startpunkt heilen darf. Zusätzlich können bestimmte Ereignisse entweder durch einen Spieler (`player_event`)oder durch die Umgebung(`nature_event`) erzeugt werden. Dabei kann weiterhin zwischen `player_asserts` und `player_forbids` unterschieden werden. Diese Unterteilung existiert ebenso für die Umgebung. Der initiale Zustand des Spieles kann über das `initially`-Prädikat formuliert werden. Abschließend soll ein Beispiel zeigen, wie Objekte innerhalb der Spielwelt durch einen Spieler aufgenommen werden können.

```
player asserts(pickup(O)) <- kind(O, gold).
player forbids(pickup(O1)) <-
kind(O1,K ), holds(carrying(O2)), kind(O2,K ).
```

Diese Statements beschreiben, dass ein Spieler immer Gold aufnimmt. Er darf aber niemals zwei Objekte desselben Typs tragen, auch keine Gold-Objekte.

Auf die hier vorgestellte Weise lassen sich komplexe Spielmechaniken definieren und analysieren. Basierend auf einem in LudoCore definierten Spiel lassen sich über Abfragen Traces der Spielwelt ermitteln. Auf diese Weise kann geprüft werden, ob ein Spiel gewonnen werden kann. Dieser Umstand kann über weitere Abfragen näher analysiert werden. Beispielsweise könnte gefragt werden, ob das Spiel gewonnen werden kann, wobei die Gesundheit nicht unter 5 sinken soll.

Über diese Traces kann die abstrakte Spielmechanik näher verfeinert und unerwünschte Nebeneffekte können vermieden werden. Die Veränderungen im Rahmen einer inkrementellen Entwicklung können schnell ausprobiert werden. LudoCore kann also zum einen für den abstrakten Prototyp einer Spielmechanik und zum anderen für das anschließende Balancing gut eingesetzt werden.

Abgrenzung

Zunächst unterscheidet sich LudoCore von der hier vorgestellten Sprache in dem Sinne, dass LudoCore während der Entwicklungs- und Prototyping-Phase eingesetzt wird. Die zu entwickelnde DSL zielt eher auf die Umsetzungsphase eines gehärteten Konzeptes ab. Entsprechend ist LudoCore auf eine agile Veränderung der Spielmechanik ausgelegt, wobei die hier vorgestellte DSL im Vergleich eher schwerfällig ist. Nachdem die Veränderungen über den Editor durchgeführt worden sind, muss der Code generiert und durch die Runtime ausgeführt werden.

Letztendlich ist LudoCore eine logische GameEngine, die es erlaubt, den Spielzustand über Abfragen zu analysieren. Auf diese Weise lassen sich nicht eingeplante Nebeneffekte innerhalb der Spielmechanik lokalisieren. Die hier vorgestellte Sprache bietet in Bezug auf eine Analyse keine Unterstützung. Bei einer fehlerhaften Umsetzung ist es durchaus möglich, dass ein Spiel nicht gewonnen werden kann.

3.5. Fazit

In diesem Kapitel wurde das Referenzbeispiel vorgestellt und anhand der aus Kapitel 2.2 erarbeiteten Definition analysiert. Anschließend wurde eine Vision der Sprache und der dazugehörigen Komponenten aufgezeigt. Im Rahmen eines Minimal-Entwurfes wurden Anforderungen für die Umsetzung der Sprache und das Design formuliert.

4. Entwurf der Domain Specific Language

4.1. Einleitung

Die domänenspezifische Sprache soll die Umsetzung von komplexen Abläufen in Gesellschaftsspielen unterstützen und erleichtern. Ihr Name soll „lGold“ lauten. Die Sprache selbst soll flexibel sein, um möglichst viele unterschiedliche Spielmechaniken unterstützen zu können und die Entwicklung maßgeblich zu erleichtern. Allgemein gilt die Balance zwischen Flexibilität, Komplexität, Nutzbar- sowie Umsetzbarkeit einzuhalten.

In dem folgenden Abschnitt sollen die grundlegenden Entscheidungen, welche relevant für die Entwicklung der Sprache waren, vorgestellt werden. Danach sollen die allgemeinen Sprachkonzepte von lGold geschildert werden. Anschließend sollen in Abschnitt 4.4 die einzelnen Sprachelemente vorgestellt und diskutiert werden. Für ausgewählte Elemente soll dabei ein in lGold definiertes Beispiel den Einsatz erläutern. Die Beispiele folgen dabei mit zwei Ausnahmen dem in der Analyse beschriebenen Referenzbeispiel „Ohne Furcht und Adel“. Sämtliche Beispiele finden sich im Anhang unter Abschnitt C.1.

4.2. Grundlagen

In diesem Abschnitt werden die grundlegenden Entscheidungen im Rahmen der Entwicklung der Sprache beschrieben. Da die Sprache eine vollständig eigene Syntax besitzen soll, um die unterschiedlichen Konstrukte und Konzepte adequat abzubilden, ist sie eine externe DSL. Die Syntax baut auf der Basis-XML-Syntax und einem selbst definieren XML-Schema auf. Das Hintergrundwissen über XML wird dabei vorausgesetzt [1].

Wird XML als Grundlage für eine DSL herangezogen, steht die verminderte Lesbarkeit in der Kritik. Da eine vollständige Unterstützung der Sprache ebenso einen ausgereiften Editor einschließt, wird davon ausgegangen, dass nur erfahrene Softwareentwickler Spiele direkt in

[1] Bei Bedarf stellt Ray (2004) eine gute Lektüre dar.

der Sprache und ohne Editor formulieren. Dabei können wiederum XML-Editoren verwendet werden, die eine syntaktische Fehlerbehandlung und Code-Completion unterstützen [2]. Die Nutzung von XML erleichtert die Entwicklung des DSL-Processors maßgeblich, da der Syntaxbaum über das DOM bereitgestellt wird und ein SAX-Parser genutzt werden kann, um auf dem Baum zu arbeiten. Auf dieser Grundlagen soll ein Parser entwickelt werden, welcher den zu generierenden Code erzeugt.

Die Sprache wird nicht imperativ, sondern deklarativ programmiert. Die Verarbeitung wird nach dem Modell des Zustandsautomaten durchgeführt. Schon in Abschnitt 2.2.7 wurde beschrieben, dass sich der Ablauf von Spielen in Zustandsautomaten abbilden lässt. Abbildung 4.1 zeigt den Gesamtablauf eines Spieles, der aus unterschiedlichen **Phases** besteht.

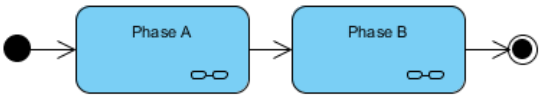


Abbildung 4.1.: Ausführung eines Spieles

Als Interaktions-Schnittstelle wurden die Zustände gewählt. Dies bedeutet, dass der Spielzustand verändert wird und daraufhin der Zustandsautomat in den Folgezustand übergeht. Abbildung 4.2 zeigt den Ablauf innerhalb einer **Phase**. Zustandsübergänge können entweder über **Actions** ausgelöst oder durch die Spiellogik erzwungen werden, wenn innerhalb eines Zustands keine **Action** ausgeführt werden kann.

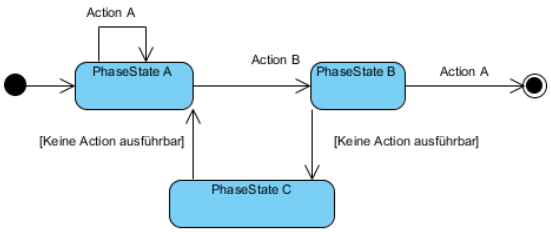


Abbildung 4.2.: Ablauf innerhalb einer Phase

Diese Darstellungen sollen nur einen groben Eindruck vermitteln, auf welche Weise ein Spiel innerhalb der Runtime ausgeführt wird. Über die Sprache wird nur das semantische Modell beschrieben, aus dem wiederum Code generiert wird. Dieser Code bestimmt, wie sich der Zustandsautomat verhalten soll. In diesem Sinne kann davon gesprochen werden, dass die Sprache die Konfiguration für die Runtime erzeugt.

[2] Wie z.B. Notepad++ (http://notepad-plus-plus.org/) mit dem zugehörigen Standard-Plugin XML-Tools

Die Entscheidung für die Generierung von Code anstelle einer Interpretierung zur Laufzeit ist auf mehrere Umstände zurückzuführen. Zunächst soll der DSL-Processor von der Runtime getrennt bleiben, da das Verhalten des Spiel-Zustandsautomaten zur Laufzeit nicht angepasst werden können soll. Weiterhin soll eine enge Kopplung zwischen den ansonsten unabhängigen Komponenten vermieden werden. Ebenso spielt die Performance bei der Ausführung der Runtime eine entscheidende Rolle, wobei davon ausgegangen wird, dass der vorgenerierte Code schneller ausgeführt werden kann als bei einer Interpretation zur Laufzeit.

4.3. Sprachkonzepte

4.3.1. Skizzierung der Sprachstruktur

Die unterschiedlichen Prozesse und Informationen, die ein Spiel beschreiben, sollen mit lGold abgebildet werden können. Dabei wird dem Entwickler durch lGold auch die konzeptionelle Arbeit erleichtert, da durch die Struktur der Sprache bereits bestimmte Lösungswege vorgegeben sind.

Anhand von lGold ist es möglich, eine Spielmechanik mit den vorgestellten Begrifflichkeiten aus Abschnitt 2.2 zu beschreiben. Die Elemente der Sprache sind alle als atomar und grundlegend anzusehen. Jedes Element ist die abstrakte Repräsentation eines bestimmten Konzeptes. Zum einen ermöglicht die abstrakte Definition eine sehr große Flexibilität, zum anderen muss die Funktionalität vollständig beschrieben werden. Dies bedeutet, dass lGold an sich keine automatischen Hilfs-Konstrukte enthält, welche bei der Entwicklung die Definitions-Arbeit erleichtern. Der Entwickler muss den gesamten Spielfluss definieren und kann nicht auf Module zurückgreifen, welche beispielsweise vorschreiben, dass die Spieler reihum ihren Zug durchführen können. Er muss hingegen definieren, dass ein aktiver Spieler vorhanden ist und beschreiben, wann der aktive Spieler wechselt. Dies ist darauf zurückzuführen, dass eine sehr große Domäne mit lGold abgedeckt werden soll. Über zusätzliche auf lGold basierende Automatisierungen könnte so ein Modul allerdings im Nachhinein erstellt werden, da es das vorliegende abstrakte Fundament von lGold nur näher definiert. Im Rahmen der Bedienungsfreundlichkeit spielt die graphische Entwicklungsoberfläche eine große Rolle.

Eine Spieldefinition lässt sich in folgende Basiselemente unterteilen:**GameState**, **Actors**, **Attributes**, **Spaces**, **Fields**, **Rules**, **Actions**, **Implications**, **Objects**, **Phases**, **PhaseStates**, **ActionStates**, **Operations**, **Timer** und **Chance**. Diese Unterteilung entspricht den ausgearbeiteten Definitionen, soll aber dennoch an dieser Stelle noch einmal vorgestellt werden:

GameState: Der **GameState** enthält sämtliche Ausprägungen der Elemente eines Spieles und repräsentiert den aktuellen Spielzustand.

Attribute: Ein **Attribute** repräsentiert ein einfaches Merkmal.

Objects: Ein **Object** stellt ein Objekt innerhalb des Spieles dar.

Spaces: Der **Space** repräsentiert einen Bereich, welcher durch **Fields** genauer definiert und aufgeteilt werden.

Fields: Ein **Field** stellt ein bestimmtes Feld innerhalb des **Spaces** dar.

Actors: Ein **Actor** repräsentiert einen Spieler.

Phases: Eine **Phase** stellt einen bestimmten Ablauf des Spieles dar.

PhaseStates: Ein **PhaseState** beschreibt einen bestimmten Unterzustand innerhalb einer **Phase**.

Rules: **Rules** werden miteinander verknüpft und bilden gemeinsam eine Vorbedingung. Über diese Vorbedingung kann auf bestimmte Situation während des Spiels reagiert werden.

Actions: Eine **Action** repräsentiert eine durch einen Spieler aktiv ausgeführte Aktion. Eine **Action** kann zur Abbildung von komplexen Abläufen zusätzlich in **ActionStates** eingeteilt werden.

ActionStates: Ein **ActionState** beschreibt einen bestimmten Unterzustand innerhalb einer **Action**.

Implication: Eine **Implication** kann sich auf den Spielzustand auswirken, so lange die Vorbedingungen erfüllt werden.

Operations: Eine **Operation** dient dazu, die unterschiedlichen Elemente der Spielmechanik einmalig und dauerhaft manipulieren zu können.

Selectors: Über einen **Selector** können bestimmte Elemente ausgewählt werden.

Visibility: Die **Visibility** bestimmt die Sichtbarkeit eines Elementes.

Timer: Garantiert, dass eine **Action** oder **Operation** im Rahmen eines definierten Zeitabschnittes ausgeführt wird.

Chance: Eine **Chance** repräsentiert eine Wahrscheinlichkeit und kann kann als Zufallsgenerator verwendet werden.

Die Elemente lassen sich in drei unterschiedliche Zuständigkeitsbereiche einteilen: Inhalt, Ablauf und Manipulation. Die Inhaltselemente enthalten Informationen welche den aktuellen Spielzustand repräsentieren. Hierzu gehören **GameState**, **Object**, **Space**, **Field**, **Actor**, **Phase**,**Selector** ,**Visibility** und **Chance**. Zu der Klasse der Ablauf-Elemente gehören **Phase**, **PhaseState**, **ActionState**, **Rule** und **Timer**. Die Ablauf-Elemente dienen der Steuerung des Spieflusses. Als Manipulationselemente gelten **Action**, **Implication** und **Operation**. Sie ermöglichen es, Veränderungen an dem Spielzustand vorzunehmen.

Die Herausforderung bei der Entwicklung von lGold bestand darin, zum einen eine flexible Struktur zu schaffen, mit der sich eine Vielzahl grundsätzlich unterschiedlicher Spielkonzepte realisieren lässt, und zum anderen, die Sprache trotz der Flexibilität kompakt und ausdrucksstark zu halten. Neben der Domäne der Computerspielentwicklung werden in lGold auch Begrifflichkeiten und Konzepte der objektorientierten Programmierung verwendet.

In lGold wird dabei zwischen vier unterschiedlichen Arten von Elementdefinitionen unterschieden: Typen, Instanzen, Links und Templates. Eine Typ-Definition ist gleichzusetzen mit einer Klassendefinition in der Objektorientierung. Innerhalb des Typs wird festgelegt, aus welchen Elementen dieser Typ aufgebaut ist. Eine Instanz stellt entsprechend eine Ausprägung eines Typs dar. Wenn eine Instanz definiert wird, bedeutet dies, dass ein konkretes Element des Typs neu erzeugt wird. Eine Link wiederum ist nur ein Platzhalter und kann auf eine Instanz zeigen oder aber auch leer sein. Ein Link kann mit einer Variable verglichen werden, welche nur auf vorher erzeugte Instanzen zeigen darf. Es existiert weiterhin eine LinkList, welche eine Liste von Links auf unterschiedliche Elemente beschreibt. Diese Liste können Elemente hinzugefügt oder entfernt werden. Templates sind Elemente, die nur aus vordefinierten Bestandteilen zusammengestellt werden können. Diese Bestandteile legen das eigentliche Verhalten des Template-Elementes fest.

Jede Typ-Definition besitzt eine eindeutige Typ-Id, jede Instanz-Definition eine eindeutige Instanz-Id und jede Link- bzw. LinkList-Definition eine eindeutige Link-Id. Templates besitzen, mit einer Ausnahme, keine Id, da sie nicht von anderen Elementen referenziert werden müssen.

4.3.2. Element-Typen, Vererbung und Instanzen

Die Elemente **Actor**, **Space**, **Field**, **Object**, **Phase** und **PhaseState** unterstützen zusätzlich das Konzept der Vererbung. Dabei ist es möglich, ähnliche Elemente in einem abstrakten Typ zusammenzufassen. In dieser abstrakten Typ-Definition können Instanzen beschrieben werden, ohne dass diese vollständig definiert werden müssen. Beispielsweise könnte ein abstrakter Element-Typ „Bauwerk“ eine **Attribute**-Instanz ”Kosten“ enthalten und der von „Bauwerk“ erbende „Jagdschloss“-Typ würde für die definierte Kosten-Instanz nur den Wert „3“ angeben. Ein konkreter Element-Typ muss dabei sämtliche nicht definierte Werte von

Attributs-Instanzen definieren, ansonsten ist die Definition nicht zulässig. Zusätzlich werden alle in dem abstrakten Typ definierten Elemente ebenso vererbt.

Neben der Vererbung der enthaltenen Elemente zählt das Element zusätzlich zu dem geerbten Typ. Bei einem **Object** kann entsprechend direkt geprüft werden, ob es sich um ein Bauwerk handelt. Es muss nicht kontrolliert werden, ob es sich um ein Jagdschloss, ein Kirche, einen Marktplatz oder einen anderen konkreten Bauwerk-Typ handelt. Ein Jagdschloss zählt entsprechend zu mehreren Typen, dem Bauwerk- und dem Jagschloss-Typ. Auf diese Art und Weise lassen sich gemeinsame Strukturen von Elementen schnell und elegant abbilden. Jeder Typ, unabhängig davon ob er abstrakt oder konkret ist, muss eine eindeutige Typ-Id aufweisen. Über diese Typ-Id kann zum einen bestimmt werden, von welchem abstrakten Typ geerbt werden und zum anderen, von welchem Element-Typ eine Instanz erzeugt werden soll.

Eine Instanz kann dabei nur von einem konkreten und nicht von einem abstrakten Element-Typ erzeugt werden. Alle Instanzen besitzen eine eindeutige Instanz-Id. Über diese sind sie identifizier- und von anderen Instanzen referenzierbar. Beispielsweise könnte das Kosten-Attribut eines Bauwerkes über die Instanz-Id „instance.BuildingCosts“ von einer Regel angesprochen werden, welche die Kosten mit dem aktuellen Goldvorrat des Spielers vergleicht.

Es wurde bereits beschrieben, dass ein Element durch Vererbung zu unterschiedlichen Typen zugehörig ist. Die Instanz- bzw. die Link-Ids sind ebenfalls nur innerhalb der Element-Definition eindeutig. Anhand des vorgestellten Beispiels des Kosten-**Attributes** lässt sich das Problem leicht aufzeigen: Ein Bauwerk-**Object** besitzt die **Attribute**-Instanz mit der Id „instance.buildingCosts“. Innerhalb dieses **Object** kann das **Attribute** über diese Id eindeutig identifiziert werden. In OFuA werden aber unterschiedliche Instanzen von Bauwerke benötigt, wobei die Kosten des Bauwerkes selbst weiterhin über die Id „instance.buildingCosts“ referenzierbar sind. Soll nun auf die Kosten eines bestimmten Bauwerkes referenziert werden, muss dieses ausgewählt werden, da die Instanz-Id durch die Erzeugung von Instanzen nur innerhalb einer Typ-Definition eindeutig ist. Dies gilt ebenso für die Link- und LinkList-Ids.

4.3.3. Definitionsbereiche

Eine Instanz muss dabei immer in dem Instanziierungsbereich des Elementes definiert werden, für welches diese Instanz erstellt werden soll. Der **GameState** besitzt beispielsweise einen Instanziierungsbereich für **Attribute**, **Spaces**, **Actors**, **Implications** und **Actions**. Links und LinkLists werden wiederum in einem separaten Link-Bereich definiert. Die Deklaration der Element-Typen, abstrakt oder konkret, erfolgt in dem zugehörigen Typdefinitions-Bereich des Sprachelementes.

Rules und **Operations** werden in einem sogenannten Template-Definitionsbereich erstellt. Sie verfügen weder über Typ und Id und können auch nicht über Links referenziert werden [3]. Sie setzen sich aus vorgegebenen Elementen zusammen.

4.3.4. Abläufe der Spielmechanik

In diesem Abschnitt soll ein grundlegendes Verständnis für den Spielablauf von mit IGold definierten Spielen vermittelt werden. Der Ablauf eines Spieles lässt sich dabei in unterschiedliche Phasen und Zustände einteilen. Eine **Phase** repräsentiert einen Abschnitt des Ablaufs, der sich von anderen Abschnitten entweder in seinen Eigenschaften, seinen Aktionsmöglichkeiten oder in der Art der Zustandsabfolge abgrenzen lässt. Abbildung 4.3 verdeutlicht den Ablauf der Phasen des Referenzbeispieles.

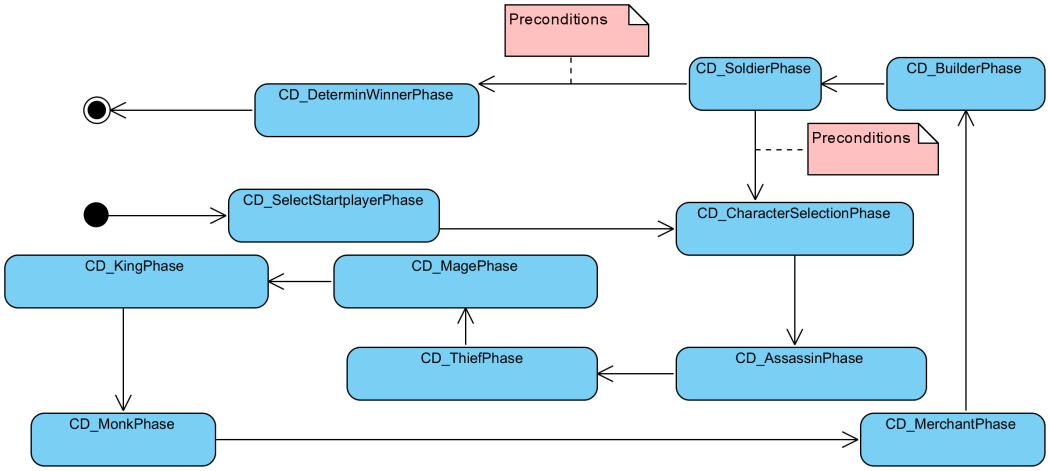


Abbildung 4.3.: OFuA Phasen

Die einzelnen Zustände der **Phase** werden **PhaseStates** genannt. Abbildung 4.4 zeigt den grundsätzlichen Aufbau der **PhaseStates** einer der Charakterphasen.

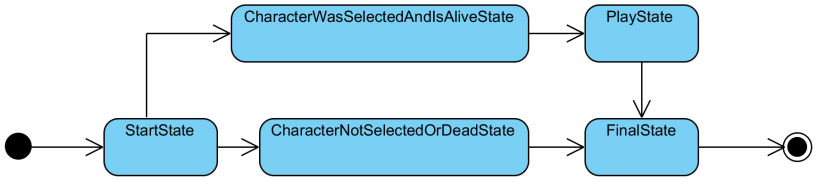


Abbildung 4.4.: PhaseStates einer Charakterphase

[3] Wiederum mit einer Ausnahme

Zusätzlich existiert ein weiterer Typ von Zuständen, die **ActionStates**. Eine **Action** ist ein aktiv durch einen Spieler ausgelöster Vorgang. Liegt diesem Vorgang eine entsprechende Komplexität zu Grunde, kann diese durch **ActionStates** abgebildet werden. An dieser Stelle soll nicht näher auf die Feinheiten von **Actions** eingegangen, sondern nur hervorgehoben werden, dass ein weiterer Typ von Zuständen existiert. Der Aufbau der in Abbildung 4.4 vorgestellten **PhaseStates** ist so gewählt, dass während des „PlayState“ die unterschiedlichen **Actions** ausgeführt werden sollen. Wenn nun ein Spieler Karten ziehen möchte, muss er in der Regel aus zwei Karten auswählen. Allerdings kann er durch Spezial-Bauwerke möglicherweise so viele Karten nehmen, dass er einfach alle Karten nimmt und die Auswahl überflüssig wird. Abbildung 4.5 zeigt, wie die entsprechenden **ActionStates** aussehen.

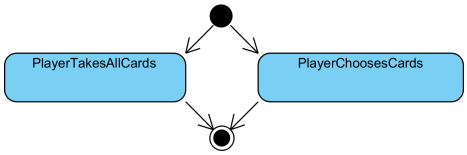


Abbildung 4.5.: ActionStates der „Karten-Ziehen“-Action

Innerhalb des Spielverlaufes ist genau eine **Phase** und ein **PhaseState** aktiv, es sei denn es findet gerade ein entsprechender Zustandsübergang statt. Während eines **PhaseStates** können mehrere **Actions** gleichzeitig ausgeführt werden und entsprechend mehrere **ActionStates** aktiv sein. Es kann nur in einen anderen **PhaseState** übergegangen werden, wenn keine **Actions** mehr ausgeführt werden. Eine **Action** kann zwar aus mehrere unterschiedlichen Zustandsabfolgen bestehen, wird aber dennoch als ein in sich geschlossener Prozess behandelt.

Der Übergang von einem Zustand zu einem anderen wird anhand Vorbedingungen bestimmt. Für einen erfolgreichen Übergang müssen sämtliche **Rules** einer Vorbedingung erfüllt werden. Diese Regeln beziehen sich auf Informationen, die innerhalb des Spieles verfügbar sind, z.B. prüft eine **Rule**, ob derzeit eine bestimme **Phase** oder ein bestimmter **PhaseState** aktiv ist. Nachdem ein Zustand abgeschlossen ist, wird anhand der Vorbedingungen der nächste Zustandsübergang bestimmt. Dabei muss bei der Auswahl der **Rules** gewährleistet werden, dass immer exakt ein Folgezustand die Vorbedingungen erfüllt.

Die auf diese Art erzeugte Abfolge lässt sich als ein deterministischer Zustandsautomat verstehen. Da der Spielverlauf nur von Zustand zu Zustand schaltet, kann dieser als diskret angesehen werden. Der Zustandsautomat arbeitet die Zustände so automatisch lange ab, bis einem der Spieler eine Interaktionsmöglichkeit geboten wird. Diese Interaktionsmöglichkeit erfolgt in Form der **Actions** oder **InteractiveOperations**. Sobald der Spieler entweder keine Interaktionsmöglichkeiten mehr besitzt oder den **PhaseState** durch eine bestimmte **Action** als beendet erklärt, wird in den nächsten **PhaseState** geschaltet.

Unabhängig von den unterschiedlichen Zustandsübergängen spielt der sich daraus ergebene Lebenszyklus einer **Phase** eine wichtige Rolle, da zeitgleich nicht mehrere **Phases** vorhanden sein können. Eine **Phase** kann bestimmte Informationen und Eigenschaften enthalten, die Einfluss auf den Spielfluss haben können. Wenn nun ein Zustandsübergang von einer zu einer anderen **Phase** durchgeführt wird, gehen die Informationen der alten **Phase** verloren. Wenn Informationen phasenübergreifend verfügbar sein sollen, müssen sie im **GameState**-Element definiert werden. Die Art und Weise, wie eine konkrete Spielmechanik umgesetzt wird, hängt also zum einen von den definierten spielspezifischen Abläufen und zum anderen von den in IGold vorgegebenen Lebenszyklen der unterschiedlichen Elemente ab.

4.3.5. Kontext und Selektoren

Der Kontext eines Elementes beschreibt, auf welche anderen Elemente es zugreifen kann. Vergleicht man den in dem vorherigen Abschnitt beschrieben Übergang einer Phase eines Phasenzustandes und eines Aktionszustandes, wird deutlich, dass das Prinzip des Übergangs dasselbe ist. Bei den dreien Varianten werden Vorbedingungen definiert, welche erfüllt werden müssen, um einen Zustandsübergang zu gewährleisten. Der Unterschied zwischen den drei Arten von Zustandsübergängen besteht in den Informationen, welche für die Definition der Regeln genutzt werden können.

Beispielsweise können sich die Regeln des Phasenzustandsübergangs nur auf die aktuelle **Phase** und sämtliche globalen Informationen beziehen. Der **ActionState** hingegen kann nur auftreten, während ein Spieler eine **Action** durchführt. Daher können sich die Vorbedingungen des **ActionStates** sich nicht nur auf die **Phase** und die globalen Informationen, sondern auch zusätzlich auf die **Action** und den auslösenden **Actor** beziehen. Ein **ActionState** besitzt im Vergleich zu einem **PhaseState** einen erweiterten Kontext. Der Kontext beschreibt entsprechend nicht nur allein die Möglichkeit eines Zugriffs, sondern auch die möglichen Zugriffspfade.

Abbildung 4.6 soll die Verschachtelung des Kontextes verdeutlichen. Es zeigt den Kontext einer **Action**, welche zu einem **Object** gehört. Umschließt ein Kontext einen anderen, enthält er zusätzlich zu seinen eigenen Zugriffsmöglichkeiten auch die des umschlossenen Kontextes. Folgend sollen die Ebenen der Kontextes skizziert werden:

GameState-Kontext: Bietet Zugriff auf die Instanzen und Links des **GameStates**.

Phase-Kontext: Bietet Zugriff auf die Instanzen und Links der aktuellen **Phase** und auf den **PhaseState**. Enthält den **GameState**-Kontext.

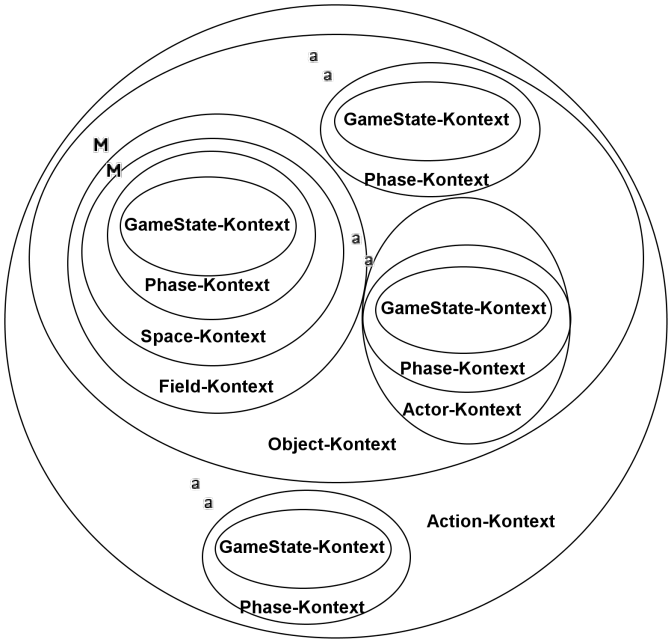


Abbildung 4.6.: Auswahl unterschiedlicher Kontext-Typen

Actor-Kontext: Bietet Zugriff auf die Instanzen und Links des **Actors** und enthält den **Phase**-Kontext. Zusätzlich kann auf das Element zugegriffen werden, für welche die Instanz oder der Link des **Actors** erstellt wurde.

Action-Kontext: Bietet Zugriff auf die Instanzen und Links der **Action** und enthält den **Phase**-Kontext. Zusätzlich kann auf das Element zugegriffen werden, für welche die Instanz oder der Link der **Action** erstellt wurde.

Space-Kontext: Bietet Zugriff auf die Instanzen und Links des **Spaces** und enthält den **Phase**-Kontext. Zusätzlich kann auf das Element zugegriffen werden, für welche die Instanz oder der Link des **Spaces** erstellt wurde.

Field-Kontext: Bietet Zugriff auf die Instanzen und Links des **Fields** und enthält den **Phase**-Kontext. Zusätzlich kann auf das Element zugegriffen werden, für welche die Instanz oder der Link des **Fields** erstellt wurde.

Object-Kontext: Bietet Zugriff auf die Instanzen und Links des **Objects** und enthält den **Phase**-Kontext. Eine **Object**-Instanz kann nur innerhalb eines **Fields** erzeugt werden. Daher enthält es auch den entsprechenden **Field**-Kontext. Weiterhin kann ein **Object** einen **Actor** als Besitzer haben. Falls dies der Fall ist, kann ebenso auf den entsprechenden **Actor**-Kontext zugegriffen werden. Zusätzlich kann auf das Element zugegriffen werden, für welche der Link des **Objects** erstellt wurde.

Der **GameState**-Kontext bietet theoretisch Zugriff auf sämtliche Elemente innerhalb der Spielmechanik. Das Navigieren durch den Kontext erfolgt mit sogenannten Selektoren. Dabei unterstützen die Kontexte jeweils unterschiedliche Selektoren. Würde man über den **GameState**-Kontext den Besitzer eine **Objects** ausfindig machen wollen, müssten zuerst alle **Actors** des **GameStates**, bei diesen alle **Spaces** und dort entsprechend in sämtlichen **Fields** nach den **Objects** gesucht werden, um den Besitzer zu finden. Diese Art der Suche erfolgt über einen Regel-Selektor. Bei dieser Variante des Selektors können unter der Verwendung von **Rules** beliebige Elemente gesucht werden. Diese Navigation von „oben“ durch die unterschiedlichen Ebenen gestaltet sich allerdings zum einen sehr aufwändig und zum anderen sehr unflexibel. Der **Object**-Kontext unterstützt daher einen Selektor, mit dem der Besitzer des **Objects** gefunden werden kann, ohne eine komplizierte Suche auszuführen.

Es gibt grundsätzlich zwei unterschiedliche Typen von Selektoren: **SingleSelectors** und **MultipleSelectors**. Ein **SingleSelector** wählt genau ein Element aus. Falls aus irgendeinem Grund kein Element durch diesen **SingleSelector** ausgewählt werden kann, stellt dies einen fehlerhaften Zustand dar. Ein **MultipleSelector** hingegen wählt kein oder mehrere Elemente aus.

Das Konzept des Kontextes und der Selektoren ist ein zentraler Mechanismus in IGold und wird in jedem Bereich eingesetzt, in welchem eine gewisse Flexibilität in der Notation erforderlich ist.

4.3.6. Regeln

Die **Rules** werden an unterschiedlichen Stellen der Spielmechanik eingesetzt, um Entscheidungen treffen zu können. Dieser Mechanismus stellt einen zentralen Bestandteil von IGold dar, entsprechend wichtig ist eine flexible und dennoch präzise Umsetzung des Konzeptes. Ein wichtiger Faktor ist dabei der Kontext des Elementes, für welches die Regel definiert wird. Die **Rules**, welche die Übergänge zwischen **Phases** beschreiben, können nicht auf dieselben Informationen zugreifen wie jene, welche die Übergänge von **PhaseStates** beschreiben.

Um einen hohen Grad an Flexibilität zu erreichen, werden die **Rules** in Form von Templates beschrieben. Beispielsweise gibt es eine **Rule**, die prüft, ob die aktuell aktive **Phase** von einem bestimmten Typ ist. Auf diese Art kann bei OFuA geprüft werden, ob derzeit die Phase des Meuchlers, des Diebes oder die eines anderen Charakters aktiv ist, indem die gleiche Regel mit einer unterschiedlichen Typ-Id verwendet wird. Diese **Rules** werden miteinander verkettet und es müssen jeweils alle **Rules** zutreffen, damit die Vorbedingung positiv ausfällt. Ebenso ist es möglich, das Ergebnis einer **Rule** zu negieren.

Vorbedingungen werden genutzt, um folgende Situationen zu prüfen:

- Ob ein Übergang von einer **Phase**, einem **PhaseState** oder einem **ActionState** zulässig ist.
- Ob eine **Action** durch einen **Actor** ausgeführt werden kann.
- Zum Selektieren von **Objects**, **Spaces**, **Actors** und **Fields**.
- Ob ein **Implication** aktiv ist.

4.3.7. Beeinflussung des Spielzustandes

Bisher wurde die Art und Weise beschrieben, wie der Spielfluss eines Spieles unter der Verwendung der **Phases** und anderen Elemente modelliert werden kann. An dieser Stelle soll nun auf die Möglichkeiten eingegangen werden, welche lGold bietet, um den Spielzustand während der Ausführung zu beeinflussen. Dafür stehen zwei grundsätzliche Mechanismen zur Verfügung: **Operations** und **Implications**.

Eine **Operation** stellt eine dauerhafte und einmalige Veränderung dar. Sie ist vergleichbar mit einem Methodenaufruf bei der Programmierung. Dabei existieren zwei unterschiedliche Typen von **Operations**: **InternalOperations** und **InteractiveOperations**. Bei einer **InteractiveOperation** ist mindestens ein Spieler in dem Sinne an der Ausführung beteiligt, dass er eine Auswahl trifft, welche entsprechend verarbeitet wird. Eine **InternalOperation** hingegen wird innerhalb der Spielmechanik eingesetzt, um die automatischen Veränderungen des Zustandes durchzuführen.

Allgemein erlauben **Operations** folgende Änderungen an dem Spielzustand:

- Veränderung der Werte von **Attributes**, indem sie erhöht, verringert oder auf einen vorgegebenen Wert gesetzt werden.
- Verschieben von **Objects** von einem **Field** zu einem anderen **Field**.
- Setzen einer Element-Referenz für einen **Link**.
- Hinzufügen, Entfernen einer Element-Referenz zu einer **LinkList** oder das vollständige Leeren der Liste.

Für jeden Typ von Veränderung existiert ein entsprechender **Operation**-Template-Typ. Es existiert also eine **Operation**, mit welcher ein **Object** von seinem aktuellen **Field** in ein anderes **Field** verschoben oder der Wert einer **Attribute**-Instanz gesetzt wird. Dabei spielt der Kontext, in dem die **Operation** ausgeführt wird, eine wichtige Rolle. Die **Operation**, welche ein **Object** verschiebt, benötigt zwei Informationen: Welches **Object** in welches **Field** verschoben werden soll. Diese beiden Parameter können über die Selektoren des Kontextes bestimmt werden, in dem die **Operation** ausgeführt wird.

Über eine **Implication** ist es möglich, eine vorübergehende Veränderung bei Werten von **Attribute**-Instanzen zu bewirken oder die Sichtbarkeit eines Elementes zu beeinflussen. Es können mehrere Vorbedingungen definiert werden, welche jeweils wiederum aus einer Menge von **Rules** bestehen. Trifft eine dieser Vorbedingungen zu, wird die **Implication** aktiviert. Sollte keine der Vorbedingungen mehr zutreffen, werden die Auswirkungen rückgängig gemacht. Auf diese Art lassen sich logische Zusammenhänge beschreiben. Bei OFuA kann auf diese Art die aktuelle Höhe der Siegpunkte eines Spielers verändert werden. Für jedes errichtete Bauwerk erhöhen sich seine Siegpunkte entsprechend. Wird das Bauwerk nun beispielsweise durch den Söldner zerstört und landet auf dem Ablagestapel, sind die Vorbedingungen nicht mehr erfüllt und dem Spieler werden automatisch eine entsprechende Anzahl an Siegpunkten abgezogen.

In Bezug auf die Interaktionsmöglichkeit des Spielers sind die **Action**- und die **InteractiveOperation**-Elemente relevant. Eine **Action** ist eine durch einen Spieler ausgelöste Aktion. Sie kann immer ausgeführt werden, wenn ihre Vorbedingungen erfüllt sind. Die **Actions** lassen sich in **SimpleAction**- und **ComplexAction**-Elemente unterteilen. Eine **ComplexAction** kann ähnlich wie eine **Phase** aus mehreren Zuständen, den **ActionStates**, bestehen. Allerdings müssen sämtliche **ComplexActions** abgeschlossen sein, bevor ein Wechsel der **Phase**- und **PhaseStates** möglich ist. Bei einer **ComplexAction** kann es möglich sein, dass der Spieler die Ausführung der **Action** abbrechen kann, ohne dass die **Action** als ausgeführt gilt.

Über eine **InteractiveOperation** wird ein Spieler durch das System aufgefordert, eine Entscheidung zu treffen. Dabei kann es sich in den meisten Fällen um die Auswahl eines Elementes handeln, z.B. bei der Charakterauswahl, bei der ein **Object** aus einem **Field** gewählt werden muss, welches die noch verfügbaren Charaktere enthält.

4.3.8. Sichtbarkeit

Die Sichtbarkeit kann für sämtliche Elemente der Spielmechanik definiert werden, welche für einen Spieler relevant sein können. Die Sichtbarkeit kann für **Actors**, **Spaces**, **Fields**, **Objects**, **Attributes**, **Implications** und **Actions** bestimmt werden. Nach Standardverhalten sind sämtliche Elemente der Spielmechanik zunächst sichtbar und müssen durch den Spieleentwickler eingeschränkt werden. Um die Sichtbarkeit festzulegen, werden wiederum die Selektoren des jeweiligen Kontextes des Elementes genutzt.

Die Sichtbarkeitsinformationen werden dazu genutzt, um dem jeweiligen Spieler eine automatisch erstellte Momentaufnahme des Spielzustandes zusenden zu können. So erhält jeder Spieler auch wirklich nur jene Informationen, die er während dieses Zeitpunkts einsehen darf.

Um den Aufwand der Definition bei der Sichtbarkeit in Grenzen zu halten, wird diese hierarchisch strukturiert. Abbildung 4.7 soll diese Hierarchie veranschaulichen. Wenn ein **Space** nur für seinen Besitzer sichtbar ist, sind auch alle **Fields** und deren **Objects** innerhalb des **Spaces** nur für den Besitzer sichtbar. Bei untergeordneten Elementen ist es nur möglich, die Sichtbarkeit weiter einzuschränken. Es kann kein **Object** geben, welches für alle Spieler sichtbar ist, sich aber in einem **Space** befindet, welcher wiederum von keinem der Spieler gesehen werden kann.

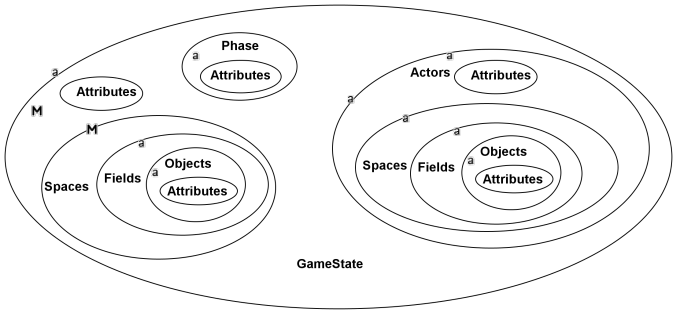


Abbildung 4.7.: Hierarchie der Sichtbarkeitsdefinition

So ist es beispielsweise möglich, sämtliche Objekte innerhalb eines **Spaces** für alle Spieler sichtbar zu machen. Ein einzelnes **Object** könnte eine Ausnahme enthalten und so nur für den Besitzer des **Objects** sichtbar sein. Zusätzlich zu der Definition der Sichtbarkeit innerhalb des Typs kann diese auch für eine einzelne Instanz angegeben werden. Dabei handelt es sich um die präziseste Art und Weise, die Sichtbarkeit zu definieren, da sie nur für eine bestimmte Instanz eines Typs gilt.

4.3.9. Timer

Wenn ein Teilnehmer durch das Spiel aufgefordert wird, eine Entscheidung zu fällen, achten in der Regel die Mitspieler darauf, dass dieser seine Wahl in angemessener Zeit trifft. Handelt es sich bei den Beteiligten um anonyme Spieler aus dem Internet, kann es vorkommen, dass einer der Spieler seine Züge stark verzögert und den anderen Teilnehmern den Spaß verdirbt. Unabhängig von dieser „Kontrollfunktion“ existieren Spielmechaniken, welche die Zeitbegrenzung als integralen Bestandteil ihres Spielprinzips nutzen, wie z.B das Spiel Tabu[4].

In lGold können Zeitbegrenzungen durch das **Timer**-Element modelliert werden. Dabei kann zwischen drei Situationen unterschieden werden, bei denen der Timer angewendet werden

[4] siehe http://de.wikipedia.org/wiki/Tabu_(Spiel)

kann: Bei **Phases**, **Actions** und **InteractiveOperations**. Innerhalb einer **Phase** kann es wünschenswert sein, dass in einer bestimmten Zeit eine bestimmte Sequenz von **Actions** ausgeführt wurde. Bei OFuA ist die Spielphase so aufgebaut, dass ein Spieler immer entweder Gold oder Karten genommen haben muss, bevor er seinen Zug beenden kann. Wenn ein Teilnehmer nun nicht reagiert, kann ein **Timer** so modelliert werden, dass dieser nach einigen Minuten automatisch die notwendigen **Actions** ausführt. Wenn wie bei diesem Beispiel nur entweder eine **Action** oder eine andere **Action** ausgeführt werden kann, wird diese zufällig ausgewählt.

Ein **Timer**, welcher sich auf eine einzige **Action** bezieht, stellt sicher, dass die Action innerhalb eines Zeitraumes abgeschlossen wird. Wird dieser Zeitraum überschritten, werden die **Operations** in der **Action** automatisch abgearbeitet. Falls auf diese Weise eine **InteractiveOperation** ausgeführt wird, bei der ein Spieler eine Entscheidung treffen muss, wird diese ebenso zufällig durchgeführt. Analog verhält sich ein **Timer**, welcher nur für eine **InteractiveOperation** definiert ist, wobei sich dieser Zeitraum nur auf die Auswahl einer einzigen **InteractiveOperation** bezieht.

4.3.10. Spielfelder

Die Spielfelder nehmen bei vielen Gesellschaftsspielen eine wichtige Rolle ein. Sie stellen einen großen Teil der Visualisierung des Spielzustandes dar und grenzen die unterschiedliche Bereiche des Spieles zum einen konzeptionell und zum anderen in Abhängigkeit ihrer Positionen voneinander ab. In lGold kann ein Spielfeld mithilfe von **Spaces** abgebildet werden. Ein **Space** kann dabei eine Menge von **Fields** enthalten und ein **Object** innerhalb des Spieles muss zu jeder Zeit genau einem **Field** zugeordnet sein.

Die **Fields** können mithilfe eines **Layouts** im **Space** angeordnet werden. Bei den **Layouts** wird zwischen vier unterschiedlichen Typen unterschieden: Dem **Custom-**, **One-**, **Two-** und **ThreeDimensionalLayout**. Der Typ des **Layouts** legt fest, auf welche Art ein **Field** an ein anderes **Field** angrenzt. Die Art, wie die **Fields** aneinandergrenzen bestimmt, aus welchen Richtungen ein Objekt sich von einem zum einem anderen **Field** bewegen kann. Dabei wird zwischen ein- und beidseitiger Bewegungsrichtung unterschieden. Ebenso kann festgelegt werden, dass ein **Field** an ein **Field** eines anderen **Spaces** angrenzt. Auf diese Art lassen sich sämtliche Felder des Spielfeldes, falls dies notwendig sein sollte, miteinander verknüpfen.

Ein **Object** befindet sich in einem **Field** und kann jeweils nur in angrenzende **Fields** bewegt werden. Wenn ein **Object** über eine längere Strecke bewegt werden soll, muss es zwischen Start- und Endpunkt eine erlaubte Verbindung durch die **Fields** geben. Dabei zählt jedes **Field** als ein Bewegungsschritt. Auf diese Weise ergibt sich ein Pfad, welchem das **Object** bis zum Ziel folgt. Ob ein Pfad zwischen zwei **Fields** existiert und wie lang dieser ist, kann

über IGold abgefragt werden. Das Bewegen eines **Objects** erfolgt anhand einer speziellen **Operation**.

Diese Art der Spielfeld-Modellierung ermöglicht zwar auf der einen Seite viele Gestaltungsfreiheiten, kann aber auf der anderen Seite mit entsprechendem Aufwand verbunden sein. Um den initialen Aufwand zu verringern und es einem Entwickler zu ermöglichen, möglichst schnell eine funktionsfähige Spielumgebung zu erstellen, bevor jedes Detail ausgearbeitet wird, existieren verschiedene Standardverhalten und Einstellungsmöglichkeiten.

Das vorgegebene Standardverhalten der **Layouts** und deren **Fields** richtet sich nach dem Typ des **Layouts**. Bei einem **CustomLayout** müssen sämtliche Verknüpfungen der **Fields** manuell definiert werden. Bei einem **OneDimensionalLayout** grenzen die **Fields** entsprechend ihrer Definitionsfolge aneinander an. Bei einem **TwoDimensionalLayout** werden sie nach Zeilen und Spalten und bei einem **ThreeDimensionalLayout** zusätzlich noch nach Ebene jeweils horizontal, vertikal und diagonal miteinander verknüpft.

Zusätzlich kann eingestellt werden, ob die **Fields** der gegenüberliegenden Seiten aneinander angrenzen, bei einem **OneDimensionalLayout** handelt es sich dabei also um das erste und letzte **Field**. Auf diese Art würde beispielsweise ein Monopoly-Spielbrett realisiert werden können. Weiterhin kann bei **Two-** und **ThreeDimensionalLayouts** definiert werden, ob ein **Field** an die horizontalen, vertikalen und diagonalen Nachbarn der zwei- und dreidimensionalen Achse angrenzt.

Sobald für ein **Field** ein abweichendes Verhalten definiert wird, wird das jeweilige Standardverhalten außer Kraft gesetzt. So ist es möglich, zunächst ein zweidimensionales Spielfeld zu modellieren und anschließend über das **customLayout** Abweichungen für einzelne **Fields** zu definieren.

Obwohl OFuA ein Kartenspiel ist und ohne ein zusätzliches Spielbrett auskommt, besitzen die unterschiedlichen „imaginären“ Bereiche, welchen man beim Spielen automatisch die Karten zuordnet, einen bestimmten Sinn. Jeder Spieler hat einen Bereich der jeweils eine der folgenden Funktionen erfüllt: Ein Bereich für seine Handkarten, seine errichteten Bauwerke, für die Bauwerke, zwischen denen er entscheiden muss, für den ausgewählten aber noch verdeckten Charakter, den bereits gespielten und für alle sichtbaren Charakter und einen Bereich für den derzeit aktiven Charakter. Zusätzlich gibt es auch globale Bereiche, wie z.B. den Nachziehstapel mit den Bauwerken. An diesem Beispiel ist gut zu erkennen, dass das Spielfeld und die Sichtbarkeit eng miteinander verknüpft sind. Meist wird die Sichtbarkeit nur für die **Spaces** und **Fields** des Spieles definiert, wobei die untergeordneten Elemente die Sichtbarkeit erben.

4.3.11. Zufallsgenerator

In den meisten Gesellschaftsspielen werden Würfel verwendet um den Zufallsfaktor zu bestimmen. In lGold können Zufälle durch das **Chance**-Element abgebildet werden. Die Wahrscheinlichkeiten einer **Chance** können entweder gleich- oder ungleich verteilt sein. Bei der Definition einer ungleich verteilten **Chance** muss für jeden Wert definiert werden, wie oft diese eintreten kann. Auf diese Art ergeben sich die unterschiedlichen Wahrscheinlichkeiten der unterschiedlichen Werte. Ein **Chance** kann nicht nur aus einer Menge von Werten wie Zahlen, sondern auch aus einer Anzahl von anderen Elementen bestehen. Wenn mehrere zufällige Ziehungen hintereinander durchgeführt werden sollen, kann bestimmt werden, ob die Wahrscheinlichkeiten dabei gleich verteilt bleiben oder das gezogene Element aus der Menge der Möglichkeiten entfernt werden soll.

Eine **Chance** kann an jeder Stelle eingesetzt werden, an der über einen **Selector** ein instanziierbares Element ausgewählt wird oder ein primitiver Typ wie ein Integer genutzt werden kann.

4.3.12. Erweiterbarkeit

Grundsätzlich lassen sich die möglichen Erweiterungen in zwei unterschiedliche Kategorien unterteilen: In eine grundlegende Erweiterung der Basis-Elemente und in eine aufbauende Erweiterung, bei der die bestehenden Konzepte genutzt werden, aber eine deutliche Vereinfachung der lGold-Definition das Ziel ist.

Bei einer grundlegenden Erweiterung müssen folgende Themen bedacht werden:

- Was für Definitionsmöglichkeiten existieren für das neue Element? Typ-, Instanz-, Link-, LinkList- oder Template-Definition?
- Wie wirkt sich das neue Element auf den Spielfluss aus?
- Mit welchen Elementen steht das neue Element in Beziehung? Wie wirkt sich dies auf den Kontext und die Selektoren aus?
- Wie wirkt sich das neue Element auf die Sichtbarkeit aus?
- Muss das neue Element bei der Definition des Spielfeldes bedacht werden?

Eine aufbauende Erweiterung kann eigene Elemente einführen, welche die vorhandenen Konzepte nutzen. Beispielsweise könnte für die Definition der Zustandsübergänge ein eigenes Element erschaffen werden, welches definiert, in welcher Reihenfolge die **Phases** durchgeführt werden. Dabei könnten die Vorbedingungen für die jeweiligen Zustandsübergänge erzeugt werden. Für eine **Phase** selbst müssten dann nur noch jene **Rules** definiert

werden, welche zusätzlich erfüllt werden müssen. Anhand der aufbauenden Erweiterung kann die abstrakte Basis entsprechend konkretisiert und für bestimmte Teilbereiche der Domäne der Gesellschaftsspiele entsprechend optimiert werden.

4.4. Sprachelemente

4.4.1. Einleitung

In den folgenden Abschnitten sollen die einzelnen Sprachelemente und deren Definition in XML beschrieben werden. Die Syntax leitet sich dabei von einem XML-Schema ab, welches sich ebenfalls auf der der Arbeit beiliegenden CD enthält. Sämtliche in diesem Abschnitt enthaltenen Graphiken zeigen die Typen aus der Schema-Definition. Die XML-Tags werden dabei `auf diese Art` hervorgehoben, wobei die domänenspezifisches Elemente wie auch bisher **auf diese Art** verdeutlicht werden sollen.

Zuerst werden wiederkehrende Konzepte wie die Typ-Definition, die Vererbung, das Definieren von Instanzen, Links und LinkLists, Selektoren und Sichtbarkeit eingegangen. Anschließend werden die unterschiedlichen Sprachelemente der Reihe nach beschrieben. Ein Grundwissen der Definitionsmöglichkeiten von XML wird an dieser Stelle vorausgesetzt.

4.4.2. Game

Das `game`-Tag ist das Wurzelelement der XML-Definition und umschließt sämtliche IGold-Elemente. Abbildung 4.8 zeigt die durch das XML-Schema vorgegebene Struktur. Dabei enthält das `game` neben der eigentlichen `gameDefinition`, welche sämtliche spielspezifischen Definitionen enthält, auch ein `plugins`-Element, bei welchem wiederum sämtlich zu ladende Plugins registriert werden können.

Das `gameDefinition`-Element enthält die spielspezifischen Elemente der Spielmechanik. Diese unterteilt sich in die Typ-Definitionsbereiche der **Attributes**, **Objects**, **Spaces**, **Fields**, **Actors**, **Actions**, **Implications**, **Phases** und **Chances**. Für **Rules** und **Operations** können keine Typen definiert werden.

Das `gameState`-Element ist ein reiner Instanziierungsbereich, da für den globalen Spielzustand keine Typdefinition notwendig ist und direkt Instanzen der anderen Element instanziiert werden können.

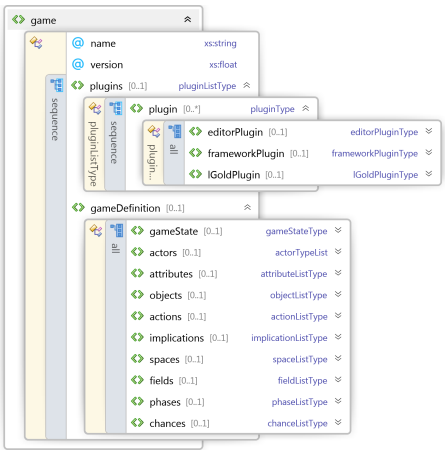


Abbildung 4.8.: game

Ein in XML definiertes Beispiel findet sich im Anhang in Abschnitt C.1.1.

4.4.3. Typen, Instanzen und Links

Typen

Die Abbildung 4.9 zeigt die XML-Vererbungsstruktur der unterschiedlichen Typen von Sprachelementen. Für **Attributes**, **Actions**, **ActionStates**, **Chances** und **Implications** können nur Typen definiert werden. Sie unterstützen keine Vererbung, und ihre XML-Typen erweitern den `elementType`. Hingegen können für **Actors**, **Objects**, **Spaces**, **Fields**, **Phases**, **PhaseStates** konkrete und abstrakte Typen definiert werden, von denen entsprechend geerbt werden kann. Ihre XML-Typen erweitern den `extendableElementType`.

Die **Operations** werden in zwei XML-Gruppen unterteilt, in `interactiveOperationGroup` und `internalOperationGroup`. Jede Gruppe besteht aus einer Menge von **Operation**-Typen. Wenn diese Gruppe referenziert wird, kann nur einer der enthaltenen **Operations** genutzt werden.

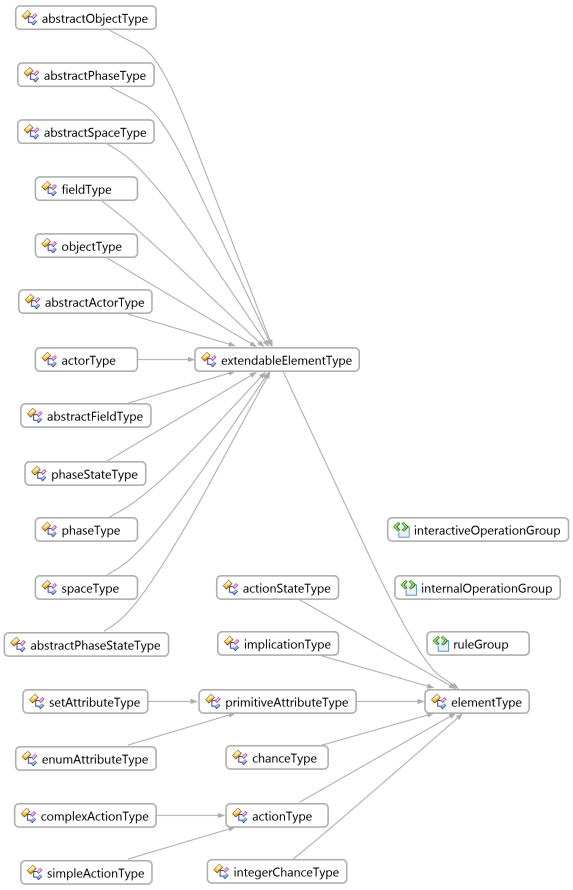


Abbildung 4.9.: Typen und Vererbung

Der in Abbildung 4.9 dargestellte `elementType` besteht aus den Attributen `name` und `typeId`. Das Attribut `name` dient nur zur besseren Lesbarkeit, wichtiger ist die `typeId`, welche eindeutig innerhalb der Spieldefinition sein muss.

Der `extendableElementType`, zu sehen in Abbildung 4.10, erweitert den `elementType` und enthält zusätzlich das `extends`-Attribut. Hier wird die `typeId` des Typs eingetragen, von dem geerbt werden soll. Ein konkreter Typ kann nur von einem abstrakten Typ erben. Ein abstrakter kann zusätzlich von einem anderen abstrakten Typ erben.

Abbildung 4.10.: extendableElementType

Instanzen

Eine Instanz kann für alle Elemente, mit Ausnahme von **Rules**,**Operations** und **Chances**, erstellt werden. Abbildung 4.11 zeigt die unterschiedlichen XML-Typen für Instanzdefinitionen und ihre Vererbung innerhalb des Schemas.

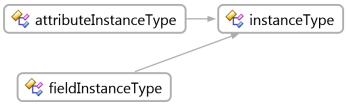


Abbildung 4.11.: Vererbungsstruktur der Instanz-Typen

Für alle instanziierbaren Elemente, bis auf **Attributes** und **Fields**, wird der `instanceType` für die Definition eine Instanz verwendet. Dieser wird in Abbildung 4.12 dargestellt. Bei einer Instanz eines **Attributes** oder eines **Fields** können noch zusätzliche Informationen angegeben werden. Diese Fälle werden bei der Vorstellung des jeweiligen Elementes erläutert.

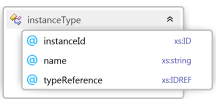


Abbildung 4.12.: instanceType

Er definiert jene Attribute, die für eine allgemeine Instanzdefinition notwendig sind. Die `instanceId` muss innerhalb der Definition eindeutig sein, die `typeReference` verweist auf eine `typeId`, der `name` dient nur der besseren Lesbarkeit.

Links

Abbildung 4.13 zeigt die Element-Typen für die Definition von **Links** und **LinkLists**. Sie können für **Actors**, **Spaces**, **Fields**, **Objects**, **Attributes**, **Implications** und **Actions** erstellt werden.

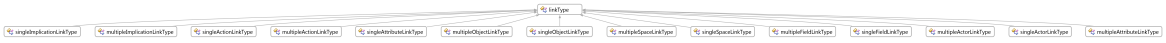


Abbildung 4.13.: Struktur der Link-Typen

Die Definition eines **Links** soll beispielsweise für einen **Actor** anhand des `single-` und eine **LinkList** mithilfe des `multipleActorLinkType` vorgestellt werden. Abbildung 4.14 zeigt die Struktur des `singleActorLinkType`. Er besteht aus einer eindeutigen `linkId` und einer `typeReference`. Über einen `singleActorSelector` kann ein **Actor** gewählt werden, welcher initial beim Starten des Spieles ausgewählt werden soll. Der `multipleActorLinkType` unterscheidet sich nur in dem Sinne von dem `singleActorSelector`, dass er einen `multipleActorSelector` anstelle eines `singleActorSelector` besitzt. Auf die Definition von **Selectors** wird im Folgenden Abschnitt eingegangen.

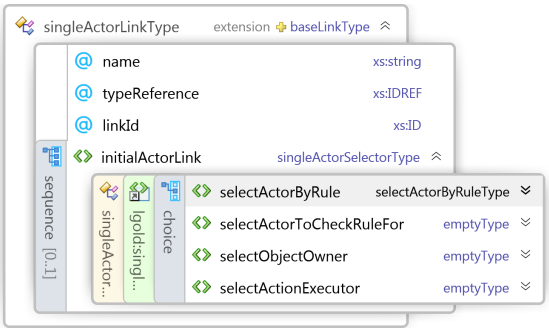


Abbildung 4.14.: singleActorLinkType

4.4.4. Selectors

Die **Selectors** ermöglichen es **Actions**, **Actors**, **Attributes**, **Implications**, **Spaces**, **Fields** und **Objects** auf unterschiedlichen Wegen auszuwählen. Da sie an unterschiedlichen Stellen Anwendung finden, sollen sie zu Beginn vorgestellt werden.

Für jedes Element das selektiert werden kann, existiert eine Menge von unterschiedlichen **Selectors**. Abbildung 4.15 zeigt die unterschiedlichen `SingleSelector`-Gruppen für die jeweiligen Elemente. Eine Gruppe besteht dabei aus einer Menge von **Selectors**. Für jedes oben genannte Element existiert wiederum eine entsprechende `MultiSelector`-Gruppe.

Abbildung 4.15.: singleSelector-Gruppen

Die Struktur eines **Selectors** soll anhand des `SingleActorSelectorGroup`-Typ beschrieben werden. Abbildung 4.16 zeigt die möglichen **Selectors** für einen **Actor**.

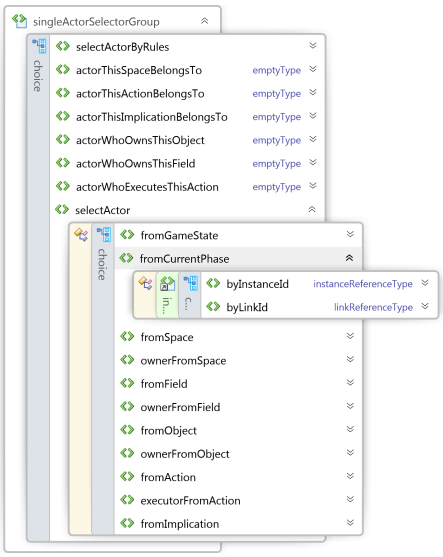


Abbildung 4.16.: singleActorSelectorGroup

Die wichtigsten `SingleSelector`-Typen sollen folgend kurz erläutert werden:

singleActorByRules: Wählt einen **Actor** anhand von **Rules** aus

actorThisSpaceBelongsTo: Wählt den **Actor** aus, zu dem der **Space** gehört. Kann nur verwendet werden, wenn die **Space**-Instanz in einem **Actor**-Typ erstellt wurde.

actorThisActionBelongsTo: Analog zu `actorThisSpaceBelongsTo`.

actorThisImplicationBelongsTo: Entspricht `actorThisSpaceBelongsTo` nur für eine **Implication**.

actorWhoOwnsThisObject: Wählt den **Actor** aus, welche das **Object** besitzt. Kann nur im Rahmen des **Object**-Kontexts genutzt werden.

actorWhoOwnsThisField: Analog zu `actorWhoOwnsThisObject`.

actorWhoExecutesThisAction: Wählt den **Actor** aus, der diese **Action** ausführt.

selectActor: Selektiert einen **Actor** anhand eines anderen **Selectors**. Auf diese Weise lassen sich **Actors** von sämtlichen Elementen beziehen. Die Abbildung 4.16 zeigt, wie ein **Actor** über eine Instanz- oder Link-Id von der aktuellen **Phase** bezogen werden kann.

Abbildung 4.17 zeigt die Struktur der **MultipleSelectors** für **Actors**. Sie ermöglichen es alle, keine und mehrere **Actors** anhand von **SingleSelectors** auszuwählen.

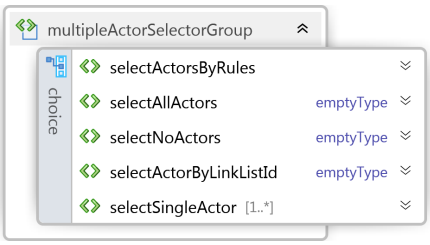


Abbildung 4.17.: multipleActorSelectorGroup

4.4.5. Visibility

Die Sichtbarkeit kann für die Elemente **Actor**, **Attribute**, **Object**, **Space**, **Field**, **Action** und **Implication** definiert werden. Jedes dieser Elemente unterstützt ein `visibility`-Tag des Typs `elementVisibilityType`. Abbildung 4.18 zeigt, dass die Definition nur aus einem `multipleActorSelectorType` besteht. Über diesen kann die Sichtbarkeit für alle, keinen oder mehrere **Actors** festgelegt werden.

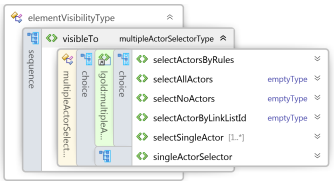


Abbildung 4.18.: elementVisibility

Für einen **Field** kann zusätzlich über die `objectCountVisibility` bestimmt werden, ob die Anzahl der Elemente sichtbar ist. Sie besteht ebenfalls aus einem `multipleActorSelectorType`.

4.4.6. GameState

Das `gameState`-Element repräsentiert den Container für den Spielzustand. Abbildung 4.19 verdeutlicht den durch das Schema vorgegebenen Aufbau. Innerhalb von `gameState` lassen sich Instanzen von **Actors**, **Attributes**, **Spaces**, **Implications** und **Actions** erstellen. Zusätzlich lassen sich **Links** und **LinkLists** von **Actors**, **Attributes**, **Spaces**, **Objects**, **Fields**, **Actions** und **Implications** definieren.

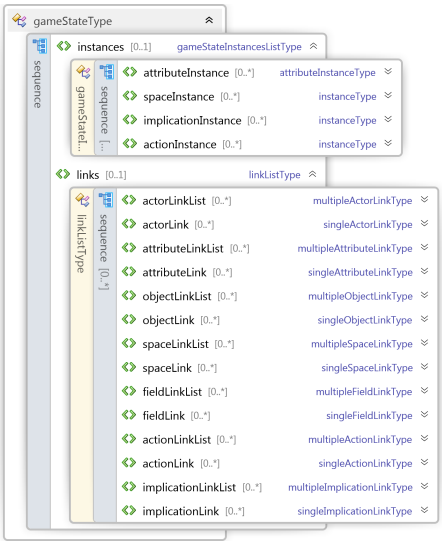


Abbildung 4.19.: gameState

Ein Beispiel für eine `gameState`-Definition findet sich im Anhang in Abschnitt C.1.2.

4.4.7. Actors

Die konkreten und abstrakten Typen von **Actors** können innerhalb der `actorTypeList` definiert werden, welche Teil der `gameDefinition` ist.

Abbildung 4.20.: actorTypeList

Ein **Actor** beschreibt einen Spieler. Ein **Actor** kann Instanzen von **Attribute**, **Spaces**, **Implications** und **Actions** sowie Links auf **Actors**, **Objects**, **Spaces**, **Fields**, **Actions**, **Implications** und **Attributes** enthalten. Die Abbildung 4.21 zeigt die Struktur einer **Actor**-Definition.

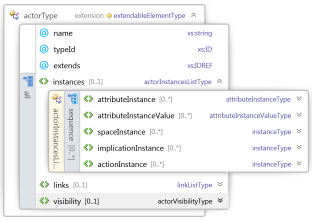


Abbildung 4.21.: actorType

Das **Actor**-Element unterstützt die Vererbung. Das `abstractActor`- unterscheidet sich nur in dem Sinne von dem `actor`-Element, dass es im Instanziierungsbereich `instances` zusätzlich die Verwendung des `abstractAttributeInstance`-Tags erlaubt. Die Art und Weise, wie abstrakte **Attributes** und das spätere Setzen ihrer Variablen definiert werden können, wird in Abschnitt 4.4.9 beschrieben.

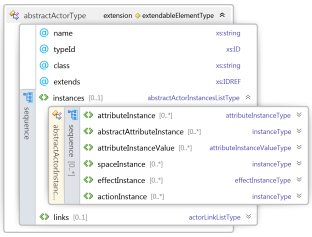


Abbildung 4.22.: abstractActorType

Von **Actors** können keine Instanzen, sondern nur **Links** und **LinkLists** erstellt werden, da für jeden teilnehmenden Spieler automatisch eine Instanz erzeugt wird. Es ist nur möglich, über **Links** oder **LinkLists** Relationen zu **Actors** herzustellen. Über die Spielmechanik kann keine zusätzliche Instanz eines **Actors** erzeugt werden. Ein Beispiel für eine `actor`-Definition findet sich im Anhang in Abschnitt C.1.3.

4.4.8. Attributes

Das Sprachelement **Attribute** beschreibt eine Eigenschaft innerhalb der Spielmechanik. Die Abbildung 4.23 zeigt die Struktur des `attributeListType`. Ein **Attribute** selbst unterstützt weder Instanzen noch Links auf andere Elemente der Spielmechanik. Es existiert für jeden Typ von **Attribute** ein entsprechendes XML-Tag.

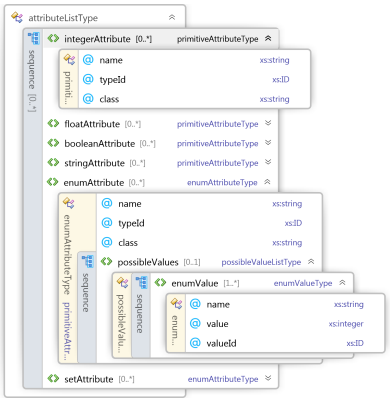


Abbildung 4.23.: attributeListType

Für das `enumAttribute` und das `setAttribute` muss weiterhin definiert werden, welche möglichen Werte eine Instanz des **Attributes** annehmen kann. Diese werden mithilfe des `possibleValues`-Elementes definiert. Für jeden möglichen Wert muss ein entsprechendes `enumValue`-Element erzeugt werden.

Ein **Attribute** kann in dem entsprechenden Instanziierungsbereich eines anderen Elementes instanziiert werden. Die Instanziierung wird durch das `attributeInstance`-Element durchgeführt. Die Struktur des entsprechenden Element-Typs ist in 4.24 zu sehen. Dabei muss jeweils der Startwert für die Instanz angegeben werden. Handelt es sich bei dem instanziierten Element um einen **Enum-** oder **Set-Attribute**, kann über das `enumValueReference`-Attribut des `enumValue`-Tags die `valueId` des `enumValue`-Elementes der gewünschte Enum-Wert referenziert werden. Zusätzlich können bei der Instanz eines **Set-Attributes** im Gegensatz zu einem **Enum-Attribute** mehrere unterschiedliche Enum-Werte gleichzeitig, jedoch kein Wert mehrmalig, ausgewählt werden.

Ein **Attribute** kann zwar nicht wie die anderen Elemente seine Struktur vererben, doch es ist möglich, eine abstrakte Instanz eines **Attributes** zu erzeugen. Sie kann anhand des `abstractAttributeInstance`-Elements definiert werden. Das Element ist vom Typ `instanceType`, welcher bereits in Abbildung 4.12 in Abschnitt 4.4.3 vorgestellt wurde. So wird eine abstrakte Instanz eines **Attributes** erzeugt, ohne dass ein Initialwert angegeben werden muss.

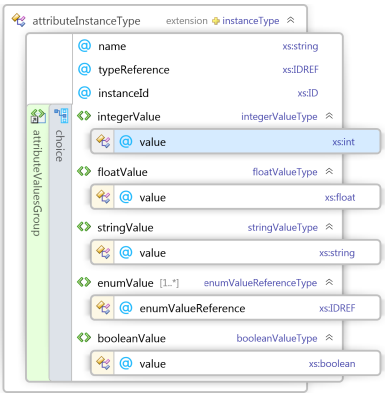


Abbildung 4.24.: attributeInstanceType

Wenn nun ein konkreter Element-Typ den abstrakten Typ erweitert, muss in seiner Definition für jedes `abstractAttributeInstance`-Element ein Wert definiert sein, insofern diese Instanz nicht bereits in der abstrakten Definition einen Wert zugewiesen bekommen hat. Die Wertzuweisung für eine `abstractAttributeInstance` erfolgt über das `attributeInstanceValue`-Element. Abbildung 4.25 zeigt die Struktur des zugehörigen Typs an.

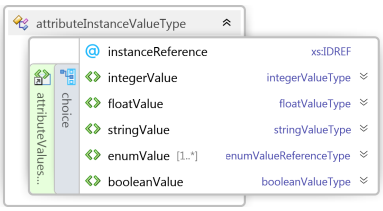


Abbildung 4.25.: attributeInstanceValueType

Im Anhang unter Abschnitt C.1.4 findet sich ein Beispiel für die Verwendung einer **Attribute**-Definition.

4.4.9. Objects

Objekte können in IGold anhand des `object`-Elementes definiert werden. Abbildung 4.26 zeigt den Typdefinitionsbereich im `gameDefinition`-Element.

Abbildung 4.26.: objectListType

Die Abbildung 4.27 stellt die Struktur eines **Object**-Typs in XML dar. Ein **Object** kann Instanzen von **Attributes**, **Implications** und **Actions** sowie die üblichen **Links** und **LinkLists** enthalten.

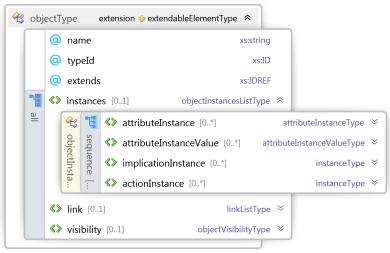


Abbildung 4.27.: objectType

Die Definition eines abstrakten **Object**-Typs unterscheidet sich von der eines konkreten Typs darin, dass bei dem abstrakten Typ in den `instances` auch das `abstractAttributeInstance`-Element genutzt werden kann.

Ein **Object** kann von einem **Field** zu einem anderen **Field** verschoben werden, wenn sämtliche zwischen diesen beiden liegende **Fields** miteinander verknüpft sind. Ein **Object** kann einem **Actor** gehören. Das Bewegen sowie die Zuordnung kann anhand einer bestimmten **Operation** durchgeführt werden. Nach der Instanziierung eines **Objects** gehört es zunächst dem **Actor**, welchem das **Field** gehört, in dem es instanziiert wurde.

Da bereits im Rahmen des **Attribute**-Beispieles im Anhang unter Abschnitt C.1.4 die Vererbung eines Objektes gezeigt wurde, soll an dieser Stelle auf ein weiteres Beispiel verzichtet werden.

Im Anhang unter Abschnitt C.1.5 wird der abstrakte Bauwerk-Typ und der davon erbende Kirche-Typ beispielhaft vorgestellt.

4.4.10. Spaces

Ein **Space** repräsentiert einen Bereich innerhalb des Spieles, welcher eine Menge von **Fields** enthält. Die Abbildung 4.28 zeigt die Struktur des Typdefinitionsbereiches innerhalb des `gameDefinition`-Elements.

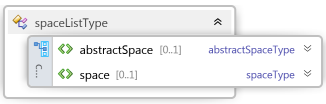


Abbildung 4.28.: spaceListType

Die Struktur eines **Spaces** wird in Abbildung 4.29 dargestellt. Er unterstützt Instanzen von **Attributes**, **Implications**, **Actions** und **Fields**. Zusätzlich kann die Definition wie bei den bisher vorgestellten Sprachelementen die jeweiligen **Links** und **LinkLists** enthalten. In Abschnitt 4.3.10 wurde bereits auf die unterschiedlichen Layout-Typen eines **Spaces** eingegangen, mit welchen Spielfelder erstellt werden können. Diese Typen können innerhalb des `fieldLayout` genutzt werden. Dabei kann immer das `customLayout` und eines der `oneDimensionalLayout`, `twoDimensionalLayout` oder `threeDimensionalLayout` genutzt werden um die Verknüpfungen zwischen **Fields** zu beschreiben.

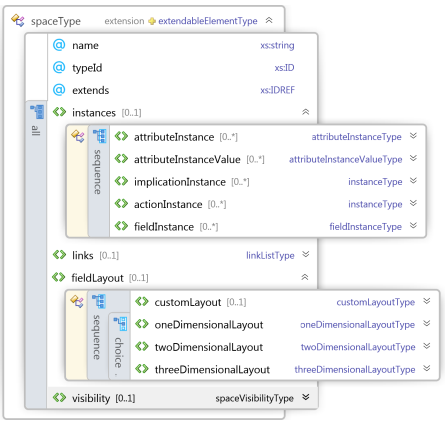


Abbildung 4.29.: spaceType

Die Abbildung 4.30 verdeutlicht die Struktur des `customLayout`. Sie ermöglicht es, eine Verbindungen zwischen zwei **Fields** zuzulassen oder zu beschränken. Dies wird geregelt, indem die Verbindung entweder im `allow`- oder im `restrict`-Bereich des Layouts definiert wird. Eine Verknüpfung in eine bzw. beide Richtungen kann über das `uniDirectionalConnection`- bzw. `biDirectionalConnection`-Element erzeugt werden. Dabei kann als Ziel entweder ein oder mehrere **Fields** über einen **Field-**

Selector ausgewählt werden. Im Anhang unter Abschnitt C.1.6 findet sich als Beispiel die Definition des Spielbereiches von OFuA.

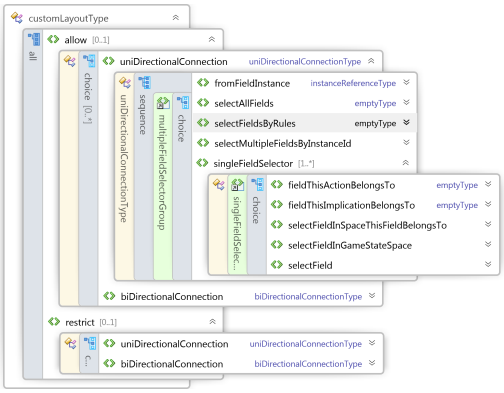


Abbildung 4.30.: customLayoutType

Das `oneDimensionalLayout` (siehe Abbildung 4.31) ermöglicht es, `Fields` auf einem Strahl anzuordnen und automatisch miteinander verknüpfen zu lassen. Anhand des `connectionType`-Attributs kann geregelt werden, ob die **Fields** ein- oder beidseitig verknüpft werden sollen. Das `connectFirstAndLastField`-Attribut legt fest, ob das erste und letzte **Field** miteinander verknüpft werden und so einen Ring bilden sollen. Die **Fields** werden dabei über die Instanz-Id referenziert. Ein Beispiel im Anhang unter Abschnitt C.1.7 verdeutlicht die Definition anhand des Monopoly-Spielbrettes.

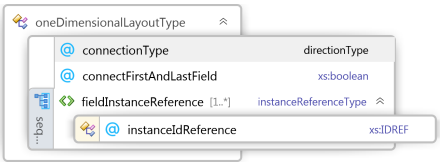


Abbildung 4.31.: oneDimensionalLayoutType

Unter Verwendung des `twoDimensionalLayout` können `Fields` in Reihen und Spalten angeordnet und miteinander verknüpfen werden. Die Abbildung 4.32 zeigt die XML-Typdefinition. Anhand des `howToConnectAdjacentFields`-Attributs wird geregelt, ob die **Fields** vertikal, horizontal, diagonal, durch Kombination aus diesen oder überhaupt miteinander verknüpft werden sollen. Das `connectFieldsOnSides`-Attribut legt fest, ob die **Fields** an den Rändern miteinander verknüpft werden. Auf diese Art entsteht eine zweidimensionale Matrix von **Fields**. Als Beispiel findet sich im Anhang in Abschnitt C.1.8 die Definition eines Schachbrettes.

Wie in Abbildung 4.33 zu sehen, erweitert das `threeDimensionalLayout` die Funktionalität des `twoDimensionalLayout` in dem Sinne, dass zusätzlich Ebenen defi-

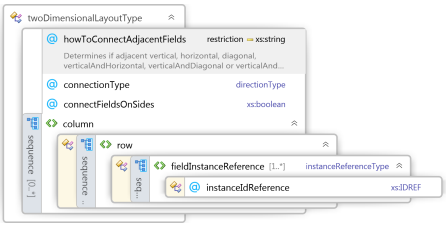


Abbildung 4.32.: twoDimensionalLayoutType

niert werden können. Es unterstützt neben dem `howToConnectAdjacentFields`- das `howToConnectLayers`-Attribut. Mit diesem Attribut wird mit den gleichen Möglichkeiten des `howToConnectAdjacentFields` gesteuert, wie ein **Field** mit den **Fields** auf den Ebenen über und unter dem **Field** verknüpft wird.

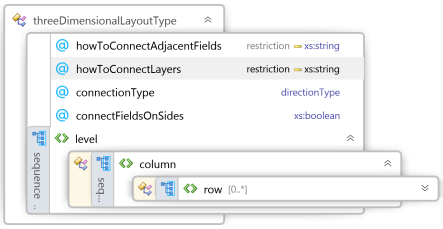


Abbildung 4.33.: threeDimensionalLayoutType

Da auf diese Art an unterschiedlichen Stellen die Verknüpfung oder Trennung von **Fields** festgelegt werden kann, soll folgend festgelegt werden, welche Definition zuerst ausgeführt wird:

1. Die Verknüpfungen aus `oneDimensionalLayout`, `twoDimensionalLayout` oder `threeDimensionalLayout`. An dieser Stelle soll noch einmal angemerkt werden, dass nur eins dieser Layouts verwendet werden kann.
2. Die Verknüpfungen aus dem `allow`-Bereich des `customLayouts`.
3. Die Trennungen aus dem `restrict`-Bereich des `customLayouts`.

Der Sinn dieser Abfolge liegt darin, dass über das ein-, zwei- und dreidimensionale Layout die **Fields** grundlegend miteinander verknüpft werden können. Über das `customLayout` können Abweichungen von diesem Standardmuster definiert werden.

Ein abstrakter **Space**-Typ unterscheidet sich nur in dem Sinne von einer konkreten **Space**-Typdefinition, dass er zusätzlich im Instanziierungsbereich das `abstractAttributeInstance`-Element unterstützt.

Zusätzlich kann ein **Space** von einem **Actor** besessen werden. Wenn der **Space** in dem Typ-Definitionsbereich eines **Actors** instanziiert wurde, gehört er dem **Actors**, ansonsten hat er keinen Besitzer. Der Besitzer kann über eine **Operation** gewechselt werden.

4.4.11. Fields

Ein **Field** stellt einen bestimmten Bereich innerhalb eines **Spaces** dar. Abbildung 4.34 zeigt den Typdefinitionsbereich innerhalb von `gameDefinition`.

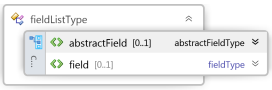


Abbildung 4.34.: fieldListType

Abbildung 4.35 zeigt die Struktur des `fieldType`. Die Typdefinition eines **Fields** unterstützt die Definition von Instanzen von **Attributes**, **Objects**, **Implications** und **Actions** sowie die üblichen **Links** und **LinkLists**. Für ein **Field** kann anhand des `allowedObjectTypes`-Elements angegeben werden, welche **Object**-Typen innerhalb des **Fields** zugelassen sind. Falls an dieser Stelle keine Angabe gemacht wird, kann das **Field** sämtliche Typen von **Objects** enthalten. **Field** unterstützt neben der `elementVisibility` zusätzlich die `objectCountVisibility`. Die `objectCountVisibility` bestimmt, für wen die aktuelle Anzahl der **Objects** des **Fields** sichtbar ist.

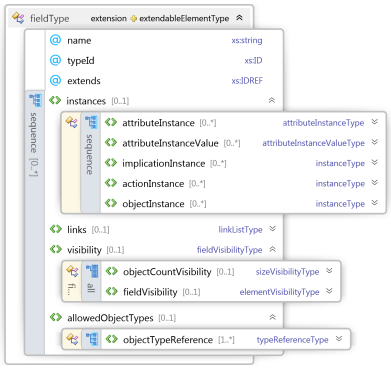


Abbildung 4.35.: fieldType

Abbildung 4.36 zeigt den XML-Typ, welcher für die Instanziierung eines **Fields** genutzt wird. Bei der Instanziierung eines **Fields** kann über das `initialObjectInstances`-Element eine Liste von `objectInstance`-Elemente angeben werden, welche zu Beginn des Spieles instanziiert werden sollen. Diese Definitionsweise stellt eine Abweichung mit der sonstigen Definitionsweise von Instanzen dar und soll vermeiden, dass unnötig viele **Field**-Typen definiert werden müssen, nur weil sie unterschiedliche Instanzen von **Objects** enthalten. Wird `initialObjectInstances` nicht angegeben, ist das **Field** leer.

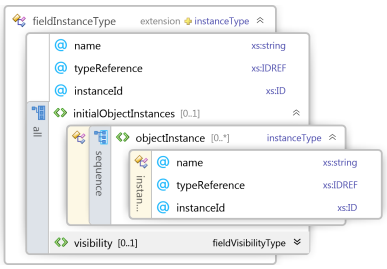


Abbildung 4.36.: fieldInstanceType

4.4.12. Phases

Eine **Phase** repräsentiert einen größeren Zeitabschnitt innerhalb eines Spieles. Abbildung 4.37 zeigt den Typdefinitionsbereich innerhalb von `gameDefinition`.

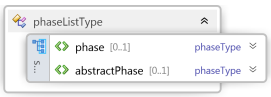


Abbildung 4.37.: phaseListType

Die Abbildung 4.38 zeigt die Struktur einer **Phase**-Typ-Definition. Es ist möglich, Instanzen von **Attributes**, **Implications** und **Actions** sowie **Links** und **LinkLists** zu definieren.

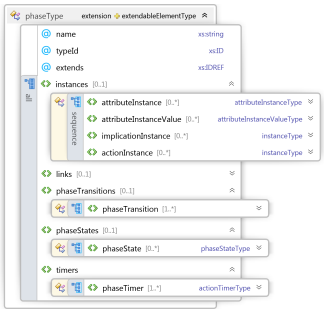


Abbildung 4.38.: phaseType

Innerhalb von `phaseTransitions` können einzelne `phaseTransition`-Elemente unter Angabe von **Rules** definiert werden. Eine **Phase** besteht aus einer Menge von **PhaseStates**, welche in `phaseStates` beschrieben werden können. Innerhalb der Definition eines konkreten **Phase**-Typs müssen sämtliche abstrakte **Attributes**, **PhaseTransitions** und **PhaseStates** entsprechend mit konkreten Typen erweitert werden.

Die abstrakte Typdefinition einer **Phase** wird durch den `abstractPhaseType` abgebildet und ist in Abbildung 4.39 zu sehen.

Diese unterstützt zusätzlich im Instanziierungsbereich das `abstractAttributeInstance`-Element, in der Liste der `phaseTransitions` das `abstractPhaseTransition`- und innerhalb von den `phaseStates` zusätzlich das `abstractPhaseState`-Element. Erbt ein `phase`- von einem `abstractPhase`-Element, werden neben den Instanzen, den Links und der Sichtbarkeit sämtliche **PhaseTransitions** und **PhaseStates** geerbt.

Für eine **Phase** kann an keiner Stelle in IGold eine Instanz definiert werden. Dies ist darauf zurückzuführen, dass eine **Phase** automatisch nach dem Starten des Spieles oder nach dem Ablauf einer **Phase** in Abhängigkeit von den definierten **PhaseTransitions** durch die Runtime ausgewählt wird.

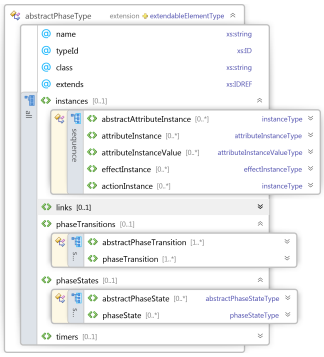


Abbildung 4.39.: abstractPhaseType

Für eine **Phase** können mehrere **Timer** in Form des `actionTimerTypes` definiert werden. Abbildung 4.40 zeigt die Struktur des Typs. Über das `duration`-Element kann angegeben werden, nach wie vielen Sekunden die Anweisungen des `onTimeout` geprüft werden sollen. Dort kann bestimmt werden, welche **Actions** nach dem Ablauf des **Timers** ausgeführt worden sein müssen. Die **Actions** können über die `singleActionSelectorGroup` ausgewählt werden. Innerhalb des `eitherAction`-Elementes kann festgelegt werden, dass nur eine der dort ausgewählten **Actions** ausgewählt werden musste. Wenn die in `onTimeout` angegebenen **Actions** nicht ausgeführt worden sind, wird dies automatisch für alle **Actors** durchgeführt, welche dies zurzeit können.

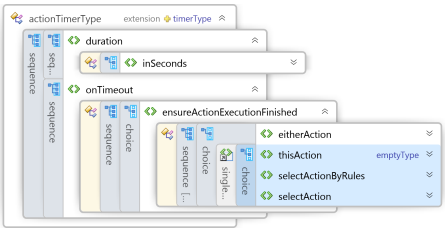


Abbildung 4.40.: actionTimerType

Das Beispiel im Anhang in Abschnitt C.1.9 zeigt, wie die Charakterphasen in OFuA unter der Verwendung von abstrakten und konkreten **Phase**-Definitionen modelliert werden können.

4.4.13. PhaseStates

Die **PhaseStates** werden im Gegensatz zu den anderen Elementen nicht direkt in einem eigenen Typdefinitionsbereich von `gameDefinition`, sondern immer direkt in der zugehörigen **Phase** definiert. Auf diese Art ist der **PhaseState** direkt mit der **Phase** verknüpft und es muss keine separate Instanz für ihn definiert werden. Diese Abweichung der Trennung von Instanz- und Typdefinitionsbereich ist auf den Umstand zurückzuführen, dass **PhaseStates** in der Regel nur für eine bestimmte **Phase** beschrieben werden.

Abbildung 4.41 zeigt die Struktur eines **PhaseStates**.

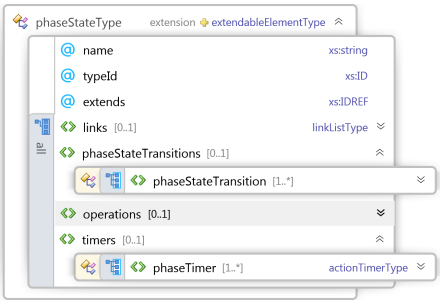


Abbildung 4.41.: phaseStateType

Ein **PhaseState** unterstützt keine Instanzen, allerdings **Links** und **LinkLists** der gewohnten Elemente. Dies ist darauf zurückzuführen, dass der Lebenszyklus eines **PhaseStates** sehr kurz ist. Über die **PhaseStateTransitions** können anhand von **Rules** die Vorbedingungen für einen Zustandsübergang festgelegt werden. Innerhalb des `operations`-Elementes werden die **Operations** definiert, welche bei einer Ausführung des **PhaseState** von oben nach unten sequenziell durchgeführt werden.

Ein konkreter **PhaseState**- kann einen abstrakten **PhaseState**-Typ erweitern. Abbildung 4.42 zeigt die Struktur des abstrakten Typ-Elements.

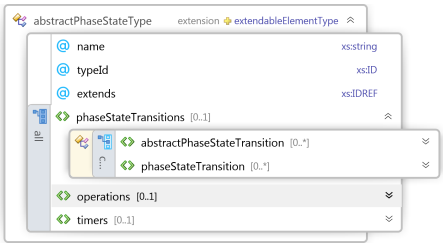


Abbildung 4.42.: abstractPhaseStateType

Innerhalb des `phaseStateTransitions`-Elements können zusätzlich `abstractPhaseStateTransition` deklariert werden. Ein konkreter **PhaseState**-Typ muss für jede definierte `abstractPhaseStateTransition` ein erbendes `phaseStateTransition`-Element aufweisen. Bei der Prüfung der enthaltenen **Rules** werden zuerst die Regeln des abstrakten und anschließend die Regeln der konkreten **PhaseStateTransition** geprüft. Bei der Ausführung der **Operations** werden ebenso zuerst die **Operations** des abstrakten und anschließend die **Operations** des konkreten Typs ausgeführt.

Ebenso wie für eine **Phase** kann ein **Timer** für einen **PhaseState** definiert werden, der nach Ablauf die ausgewählten **Actions** ausführt.

Das Beispiel für einen **PhaseState** findet sich im Anhang unter Abschnitt C.1.10 und zeigt einen **PhaseState** der in Abschnitt C.1.9 vorgestellten Charakterphase von OFuA.

4.4.14. Operations

Für eine **Operation** muss kein Typ oder eine Instanz definiert werden, bevor sie genutzt werden kann. Sie stellt die Ausführung einer atomaren, einmaligen und dauerhaften Veränderung innerhalb des Spielzustandes dar. Ein **Operation**-Typ stellt dabei ein Template für die unterschiedlichen Parameter dar, welche notwendig sind, um eine bestimmte Veränderung durchzuführen.

Abbildung 4.43 zeigt die unterschiedlichen **InternalOperations**, welche genutzt werden können. Für jedes Element der Spielmechanik existiert ein eigenes Set von **InternalOperations**. Durch diese kann der Entwickler direkt auf den Spielzustand einwirken.

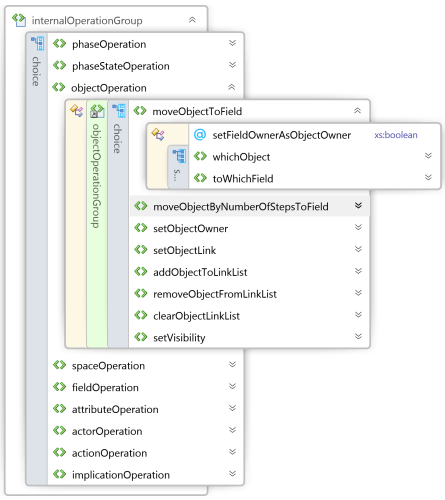


Abbildung 4.43.: internalOperationGroup

Die `moveObjectToField`-**Operation** besteht beispielsweise aus drei Parametern: Dem **Object**, welches bewegt werden soll, dem **Field**, in welches das **Object** verschoben werden soll und der Option, den Besitzer des neuen **Fields** auch als Besitzer des **Objects** zu setzen. Die Auswahl der beiden Elemente erfolgt über die jeweiligen **Selectors**, wobei der Kontext, in dem die **Operation** ausgeführt wird, sich auf die Auswahlmöglichkeiten auswirkt. Im Anhang findet sich in Abschnitt C.1.11 ein in IGold definiertes Beispiel.

Die Abbildung 4.44 zeigt die unterschiedlichen **InteractiveOperations**. Eine **Interactive-Operation** zeichnet sich dadurch aus, dass ein Spieler eine Auswahl treffen muss. Über das `objectSelection`-Element kann beispielsweise definiert werden, welche **Actors** aus wie vielen **Objects** auswählen dürfen und wie anschließend mit den ausgewählten und den anderen **Objects** umgegangen werden soll. Die Anzahl der **Objects** kann durch `numberOfElements` entweder in Abhängigkeit von einem **Attribute**, einen festen Wert oder durch eine **Chance** ermittelt werden. Nachdem ein **Actor** die **Objects** ausgewählt hat,

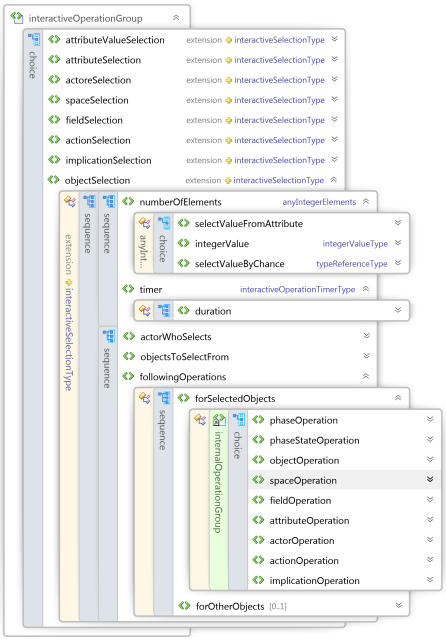


Abbildung 4.44.: interactiveOperationGroup

werden diese einzeln und nacheinander anhand der `followingOperations` mithilfe von **InternalOperations** verarbeitet. Das aktuell zu verarbeitende **Object** stellt den Kontext der **InternalOperation** dar.

Für eine **InteractiveOperation** kann ein **Timer** definiert werden. Dort wird nur über das `duration`-Element festgelegt, nach wie vielen Sekunden die **InteractiveOperation** automatisch und zufällig durchgeführt wird.

4.4.15. Rules

Die **Rules** kommen an Stellen der Spielmechanik zum Einsatz, bei denen ein gewisser Spielzustand vorausgesetzt werden muss. Dabei werden mehrere **Rules** definiert und in einem `precondition`-Element zusammengefasst. Sie stellen eine Vorbedingung dar, bei der sämtliche **Rules** erfüllt werden müssen. Abbildung 4.45 zeigt die unterschiedlichen Regel-Typen. Es werden dabei nur ein paar ausgewählte Typen vorgestellt.

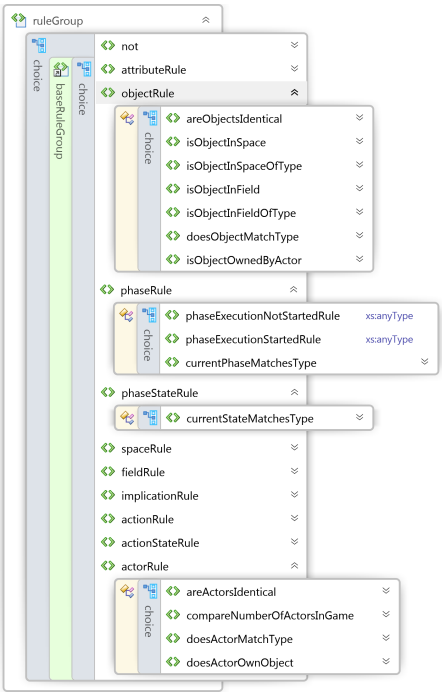


Abbildung 4.45.: ruleGroup

Für **Rules** ist es ebensowenig wie für **Operations** möglich, Typen und Instanzen zu definierten, da sie aus fest vorgegebenen Templates aufgebaut sind. Um zu prüfen ob ein **Object** sich in einem bestimmten **Field** befindet, z.B. in einem **Field** eines **Actors**, muss zunächst das **Object** und anschließend das zu prüfende **Field** selektiert werden. Die möglichen Selektoren sind dabei von dem Kontext des Elementes abhängig. Eine **Rule** kann in dem `not`-Element geschachtelt werden, um so das Ergebnis der Prüfung umzukehren.

Die **Rules** werden dabei immer innerhalb der `precondition` in der Definitionsreihenfolge von oben nach unten abgearbeitet. Dabei sollten oben allgemeinere Regeln und weiter unten präzisere Regeln abgefragt werden. Denn wenn das Spiel noch nicht gestartet wurde, muss auch nicht überprüft werden, ob ein Spieler noch ein Bauwerk errichten darf oder nicht.

Im Anhang unter Abschnitt C.1.12 finden sich verschiedene Beispiele für die Definition von **Rules**.

4.4.16. Actions

Eine **Action** stellt eine von einem Spieler ausgelöste Aktion dar. Abbildung 4.46 zeigt den Typdefinitionsbereich für **Actions** in der `gameDefinition`.

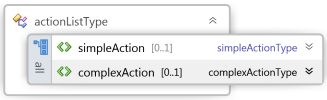


Abbildung 4.46.: actionListType

Die **Actions** lassen sich in **SimpleActions** und **ComplexActions** einteilen. Abbildung 4.47 zeigt die Struktur einer **SimpleAction**. Das `executableBy`-Element erlaubt es, über einen **Selector**, die **Actors** auszuwählen, welche die **Action** ausführen dürfen. So ist es beispielsweise möglich, für eine **Action**, die innerhalb eines **Objects** instanziiert wurde, festzulegen, dass die **Action** nur durch den Besitzer des **Objects** ausgeführt werden kann. Dieser **Actor** muss eine der innerhalb des `executableIfAnyApplies`-Elementes definierten `precondition` erfüllen um die **Action** durchführen zu können. Wenn der **Actor** die **Action** ausführt, werden die in `operations` definierten **Operations** von oben nach unten sequenziell abgearbeitet und die **Action** beendet. Ein Beispiel für eine `simpleAction` findet sich im Anhang unter Abschnitt C.1.13.

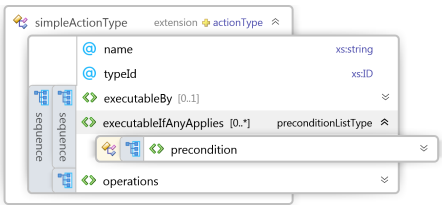


Abbildung 4.47.: simpleActionType

Abbildung 4.48 zeigt den Aufbau einer **ComplexAction**. Sie unterscheidet sich von einer **SimpleAction** in dem Sinne, dass sie aus mehreren Unterzuständen, in IGold **ActionStates** genannt, besteht. Jeder **ActionState** besitzt wie ein **PhaseState** eine Menge von **ActionStateTransitions**, um die Zustandsübergänge der einzelnen Zustände zu beschreiben. Eine **ActionStateTransition** besteht aus einer Menge von **Rules**. Jeder **ActionState** enthält ebenso wie die **SimpleAction** ein `operations`-Element, dessen **Opertions** abgearbeitet werden und die Durchführung des **ActionState** darstellen.

Möglicherweise soll eine **ComplexAction** durch den auszuführenden **Actor** abgebrochen werden können, ohne dass die **Action** als durchgeführt gilt. Dieses Verhalten kann anhand des `isActionStillAbortable`-Attributs gesteuert werden. Wenn ein einziger

ActionState verarbeitet wurde, bei dem das `isActionStillAbortable` auf `false` gesetzt ist, muss die **Action** vollständig durchgeführt werden. Ein Abbruch ist bei OFuA beispielsweise notwendig, wenn ein Spieler den zu meuchelnden Charakter auswählen soll. Die Auswahl wird in Form einer **InteractiveOperation** umgesetzt. Da der **ActionState** abbrechbar ist, kann der Spieler nun entweder eine Auswahl treffen oder die gesamte **Action** abbrechen. Wäre die **Action** nicht abbrechbar, müsste er eine Auswahl treffen. Ein in IGold definiertes Beispiel findet sich im Anhang unter Abschnitt C.1.14.

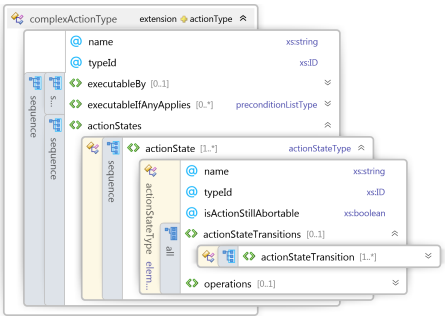


Abbildung 4.48.: complexActionType

Für jede **Action** wird gespeichert, wie oft und innerhalb welcher Spielabschnitte sie ausgeführt wurde. Diese Information kann über eine `actionRule` abgefragt werden und ermöglicht es beispielsweise, die Ausführung einer **Actions** auf ein Mal pro **Phase** oder pro Spiel zu beschränken.

4.4.17. Implications

Eine **Implication** beschreibt eine vorübergehende Veränderung des Zustandes eines **Attributes** oder der Sichtbarkeit eines Elementes innerhalb des Spieles. Dabei hält diese Veränderung nur so lange an, wie die Vorbedingungen der **Implications** erfüllt sind. Die Bedingungen werden dabei nach jeder durchgeführten **Operation** und nach einem Zustandsübergang einer **Phase**, **PhaseState** oder **ActionState** erneut geprüft. Abbildung 4.49 zeigt, dass nur ein **Implication**-Typ innerhalb des `gameDefinition` deklariert werden kann.

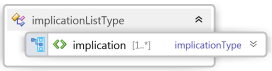


Abbildung 4.49.: implicationListType

Die Struktur einer **Implication** wird in Abbildung 4.50 gezeigt. Über das `activeIfAnyApplies-`

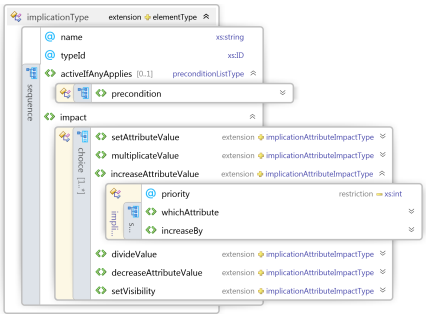


Abbildung 4.50.: implicationType

können Vorbedingungen in Form von **Rules** im Rahmen des `precondition`-Elementes definiert werden. Wenn eine der `preconditions` erfüllt ist, ist die **Implication** aktiv und die in `impact` definierten Veränderungen wirken sich aus. Um die Auswirkung zu beschreiben, existiert für jeden Typ ein eigenes Element, wie z.B. das `increaseAttributeValue`-Element, welches hier exemplarisch beschrieben werden soll. Für jedes Element kann über das `priority`-Attribut festgelegt werden, in welcher Reihenfolge die unterschiedlichen Auswirkungen abgearbeitet werden sollen. Die Priorität kann dabei zwischen 0 und 100 liegen, wobei Auswirkungen mit 0 zuerst und 100 zuletzt abgearbeitet werden. Das `whichAttribute`-Element ermöglich die Auswahl eines **Attributes** über einen **Selector** oder eine **Chance**. Das ausgewählte **Attribute** wird dann entsprechend des `increaseBy` erhöht. Dabei kann es sich um den Wert eines selektierten **Attributes**, um einen Wert einer **Chance** oder aber auch um einen fest definierten Wert handeln.

Im Anhang in Abschnitt C.1.15 wird ein in lGold definiertes Beispiel beschrieben.

4.4.18. Chance

Abbildung 4.51 zeigt die unterschiedlichen **Chance**-Typen, die innerhalb des `chances`-Element der `gameDefinition` beschrieben werden können. Für jeden primitiven Typ und für jeden instanziierbaren Element-Typ kann ein Ergebnisraum definiert werden. Die unterschiedlichen Typen können nicht gemischt werden.

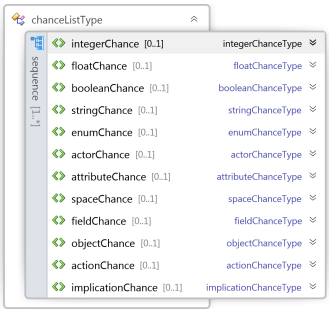


Abbildung 4.51.: chanceListType

Abbildung 4.52 zeigt den Aufbau einer `integerChance`. Für jeden möglichen Wert kann ein `chance`-Element definiert werden. Über das `amount`-Attribut wird festgelegt, wie oft der angegebene Wert in der Menge aller möglicher Werte vorhanden ist. Wenn sämtliche `chances` als `amount` „1“ angegeben haben, ist die Wahrscheinlichkeit gleich verteilt. Der Wert selbst wird über das `value`-Attribut festgelegt.

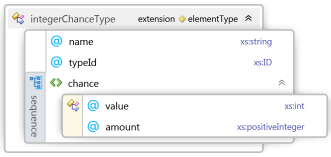


Abbildung 4.52.: integerChanceType

Die Abbildung 4.53 zeigt beispielsweise den Aufbau des `actorChance`-Elements. Der Wert wird über einen `singleActorSelector` ausgewählt.

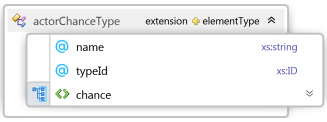


Abbildung 4.53.: actorChanceType

Eine **Chance** kann an jeder Stelle eingesetzt werden, an der auch ein **Selector** oder ein fester Wert genutzt werden kann. Um eine **Chance** zu definieren, existieren, wie in Abbildung 4.54 dargestellt, für jeden Attributs-Typ ein entsprechender Element-Typ.

Abbildung 4.54.: anyPrimitiveTypes

Abbildung 4.55 zeigt exemplarisch den Aufbau des `anyIntegerValue`-Typs. Dort kann entweder über einen `SingleAttributeSelector` ein Wert aus einem **Attribute** bezogen, über das `integerValue`-Element ein fest definierter oder über das `selectValueByChance` ein zufälliger Integer-Wert bezogen werden.

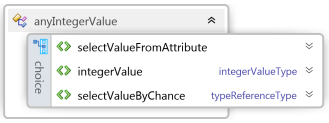


Abbildung 4.55.: anyIntegerValue

Der `anyIntegerValue`-Typ wird beispielsweise bei sämtlichen **InteractiveOperations** genutzt, um die Anzahl der auszuwählenden Elemente zu bestimmen. Die Abbildung des Elementes wurde bereits in Abschnitt 4.4.14 vorgestellt. Im Anhang in Abschnitt C.1.16 findet sich ein Beispiel, welches zeigt, wie **Chance** genutzt werden kann, um einen Würfelwurf zur Bewegung eines **Objects** zu definieren.

4.5. Fazit

4.5.1. Bewertung

Konformität

Die für lGold ausgewählte Zieldomäne ist sehr breit aufgestellt. In Abschnitt 2.2 wurde eine erste Definition zur Abbildung dieser Domäne erarbeitet und im Rahmen der Analyse erweitert. Zusätzlich wurde die Konformität der Sprache in Bezug auf die Domäne bereits am Referenzbeispiel in einem gewissen Maße erwiesen. Da dies kein Beweis für die Abbildbarkeit der Domäne darstellt, sollen an dieser Stelle wichtige Konzepte und Eigenschaften der Sprache in Bezug auf eine umfassende Abdeckung von Gesellschaftsspielen diskutiert werden.

Die Sprache lGold bedient sich neben der Domäne der Gesellschaftsspiele auch Begrifflichkeiten aus der Programmierung. Zu diesen gehören die Vererbung sowie die Trennung zwischen Typen, Instanzen und Links. Dies sind zentrale grundlegende Konzepte und können bei der Entwicklung eines Computerspieles vorausgesetzt werden.

Die Flexibilität, welche anhand der vorgestellten Konzepte des Kontextes und der Selektoren ermöglicht wird, stellt eine Grundlage für die Abbildbarkeit einer möglichst großen Domäne dar. Weiterhin spielen die Template-Elemente in diesem Zusammenhang eine wichtige Rolle, da sie in Verbindung mit den Selektoren ein sehr ausdrucksstarke Definition erlauben. Zudem wird über den Kontext, die Selektoren und die Templates ein hoher Grad an Wiederverwendbarkeit der einzelnen Sprachelemente sichergestellt.

Die unterschiedlichen Sprachelemente und ihre Funktion sind klar definiert und lassen sich deutlich voneinander abgrenzen. Die Begrifflichkeiten orientieren sich dabei an der Domäne. Im Folgenden sollen die besonders relevante Aspekte hervorgehoben werden.

Durch die Verwendung der **Layouts** können die unterschiedlichsten Anforderungen an ein Spielfeld modelliert werden. Für geometrische Spielfelder existieren das **One**-, **Two**- oder **ThreeDimensionalLayout**. Sämtliche Abweichungen können anhand des **CustomLayouts** definiert werden.

Anhand des **Visibility**-Elementes kann für jedes Sprachelement von lGold bestimmt werden, welche Informationen für welchen Spieler sichtbar sind. Auf diese Weise lassen sich Spiele abbilden, die auf perfekten und unvollständigen Informationen basieren [5].

[5] Siehe Abschnitt 2.2.2

Über das **Timer**-Element lassen sich zum einen für die Durchsetzung der „Behavioral Rules“ [6] und zum anderen für die Realisierung bestimmter Spielkonzepte nutzen. Bei manchen Spielen müssen die Teilnehmer beispielsweise eine Aufgabe unter Zeitdruck lösen.

Die **Implications** ermöglichen es, die Eigenschaften sowie die Sichtbarkeit von Elementen in Abhängigkeit von einem Spielzustand zu beeinflussen. Die Auswirkungen werden dabei automatisch wirksam, wenn der gewünschte Zustand eintritt oder entsprechend unwirksam, wenn er nicht mehr zutreffen sollte. Auf diese Weise lassen sich komfortabel Abhängigkeiten und Beziehungen von Elementen abbilden.

Die vorgestellten Argumente weisen darauf hin, dass sich die Konzepte der Domäne der Gesellschaftsspiele in lGold gut abbilden lassen [7]. Eine belastbare Aussage kann nur im Rahmen einer separaten Fallstudie erbracht werden.

Orthogonalität

In Abschnitt 4.3.1 wurden die einzelnen Elemente vorgestellt. Sie lassen sich klar und eindeutig voneinander abgrenzen. Dies lässt sich auch an der klaren Unterteilung in die unterschiedlichen Kategorien von Inhalts-, Ablauf- und Manipulationselemente belegen. Das Zusammenspiel von Kontext, Selektoren und Template-Elementen ermöglicht eine gezielte Definition von unterschiedlichen Spielmechaniken

Weiterhin lässt sich der allgemeine Spielablauf auf die beiden in Abschnitt 4.2 vorgestellten Abbildungen 4.1 und 4.2 reduzieren.

Da die unterschiedlichen Elemente sich in ihrer Funktionalität und die Anwendung der Sprachkonzepte ebenfalls gut voneinander abgrenzen lassen, wird der Sprache lGold an dieser Stelle ein hoher Grad an Orthogonalität beigemessen werden [8].

Unterstützung

Die in lGold vorgestellten Beispiele der einzelnen Sprachelemente zeigen bereits, dass die Verwaltung der Sprache in Form eines XML-Dokumentes schnell unhandlich und unübersichtlich wird. Daher stellt die Unterstützung von lGold in Form von Werkzeugen ein wichtiges Kriterium für dessen allgemeine Einsatzfähigkeit dar. Die Werkzeuge werden im Folgenden Kapitel beschrieben. Die anschließende Bewertung findet findet in Abschnitt 5.6.2 statt.

[6] Siehe Abschnitt 2.2.6
[7] Anforderung DSL-1 gilt als erfüllt
[8] Anforderung DSL-2 gilt als erfüllt

Erweiterbarkeit

In Abschnitt 4.3.12 wurde aufgezeigt, welche Aspekte bei einer grundlegenden Erweiterung bedacht werden müssen. Da die zentralen Konzepte von Kontext, Selektoren und Templates sehr flexibel ausgelegt sind, werden auch grundlegende Erweiterungen gut in die bestehende Sprachdefinition integriert werden können. Aufbauende Erweiterungen, welche nicht grundlegend die Sprache ergänzen, sondern mit den vorhandenen lGold-Elementen abgebildet werden können, sollten sich ebenfalls einfach in die Sprache eingliedern lassen [9].

Langlebigkeit

Die potenzielle Langlebigkeit von lGold wird am besten durch die gute Abbildbarkeit einer so großen Domäne gezeigt. Falls eine entsprechende Unterstützung und Erweiterbarkeit garantiert werden können, lässt sich lGold für die Entwicklung von sehr unterschiedlichen Spielkonzepten einsetzen [10].

Sonstiges

Derzeit ist es in lGold nicht möglich, während eines laufenden Spieles Instanzen von Elementen zu erzeugen. Es wird derzeit angenommen, dass ein begrenzter Vorrat an Elementen existiert und diese, wenn auch für die Spieler nicht sichtbar, bereits an irgendeinem Ort existieren. Dies spiegelt die in den meisten Brett- und Gesellschaftsspielen Begrenzung von Spiel-Elementen wider. Weiterhin kann derzeit das ursprüngliche definierte **Layout** eines **Spaces** nicht während des Spieles verändert werden.

Da die Sprache gut erweiterbar zu sein scheint, stellen zum einen die hier vorgestellten und zum anderen die möglicherweise in dieser Arbeit nicht bedachten Konzepte von Gesellschaftsspielen keine grundlegenden Probleme für die Abbildbarkeit der Domäne dar.

4.5.2. Zusammenfassung

In diesem Kapitel wurden zu Beginn die grundlegenden Entscheidungen für die Entwicklung der domänenspezifischen Sprache lGold vorgestellt und diskutiert. Anschließend wurden die Sprachelemente, welche auf den ausgearbeiteten Definitionen der vorherigen Kapiteln basieren, skizziert und klassifiziert. Danach wurden die jeweiligen Konzepte der Sprache beschrieben. Im anschließenden Abschnitt wurden die jeweiligen Sprachelemente einzeln

[9] Anforderung DSL-4 gilt als erfüllt
[10] Anforderung DSL-5 gilt im Sinne der domänenspezifischen Sprache als erfüllt

betrachtet, diskutiert und in vielen Fällen anhand von in lGold definierten Beispielen verdeutlicht. Abschließend wurde die Sprache in Bezug auf die in Abschnitt 3.3.3 definierten Anforderungen bewertet. Das nun folgende Kapitel soll zeigen, auf welche Weise die in lGold definierten Spiele erzeugt werden können.

5. Design

5.1. Einleitung

Zu Beginn dieses Kapitels soll eine Übersicht der Architektur beschrieben und im Anschluss die Art und Weise vorgestellt werden, wie die Umsetzbarkeit der unterschiedlichen Komponenten in Form von Feasibility-Studies demonstriert wird. Anschließend soll das Design der Entwicklungsoberfläche, der Runtime für die in der Sprache lGold definierten Spiele vorgestellt werden. Der Fokus des Designs liegt dabei auf dem grundlegenden Framework, welches als Basis der Runtime anzusehen ist und für die Umsetzung der in der Sprache definierten Konzepte verantwortlich ist.

5.2. Architektur-Übersicht

Zu Anfang dieses Kapitels soll zunächst die grundlegende Architektur der unterschiedlichen Komponenten vorgestellt werden. Abbildung 5.1 zeigt, wie diese miteinander verknüpft sind. Folgende Komponenten sollen kurz vorgestellt werden:

graphische Oberfläche: Ist Teil der Entwicklungsumgebung. Anhand der graphische Oberfläche kann ein Entwickler Spiele in lGold erzeugen, bearbeiten und debuggen.

DSL-Processor: Der DSL-Processor ist ein Bestandteil der Entwicklungsumgebung und generiert die ElementTemplates, welche in der Runtime benötigt werden.

ElementTemplates: Sie werden von dem DSL-Processor generiert und durch die GameInstance genutzt.

GameInstance: Die GameInstance stellt eine Instanz des Frameworks dar und verhält sich wie ein Zustandsautomat. Das Verhalten wird dabei durch die instanziierten ElementTemplates bestimmt.

Framework: Das Framework stellt die Grundlage für die GameInstance dar und ist für die Umsetzung der in lGold beschriebenen Konzepte verantwortlich.

RuntimeCore: Repräsentiert den Kern einer Runtime, welcher die GameInstance nutzt, um mit dem Zustandsautomaten des Spiels kommunizieren zu können. Sie ist verantwortlich für die spielunabhängigen Dienste wie z.B. Persistenz, Kommunikation, Sicherheit, etc..

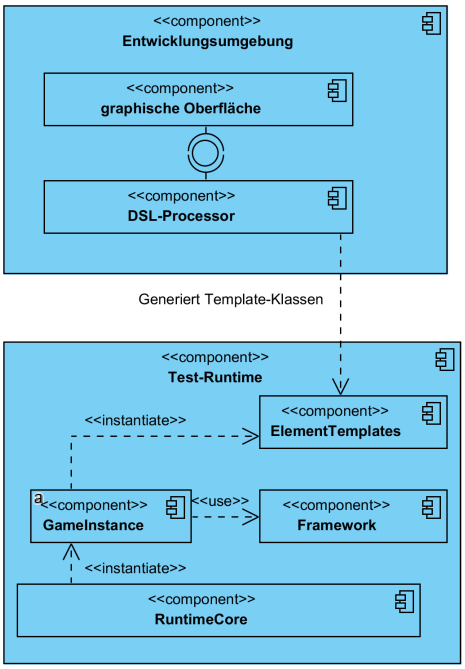


Abbildung 5.1.: Komponenten

5.2.1. Feasibility-Studies

Für die Entwicklungsumgebung soll anhand eines Feasibility-Tests die grundlegende Umsetzbarkeit der graphischen Oberfläche demonstriert werden. Dabei soll gezeigt werden, auf welche Weise eine Ansicht implementiert werden könnte. Für den DSL-Processor soll veranschaulicht werden, wie aus den Ansichten der Code für die Runtime generiert werden kann.

Für die Umsetzung der Test-Runtime und des Frameworks wurde ein Prototyp angefertigt, welcher die grundlegende Funktionsweise einiger Konzepte des Frameworks demonstriert. Die während dieser Implementierung gewonnen Erfahrungen flossen wiederum in das Design des Frameworks und der Test-Runtime mit ein und trugen maßgeblich zu dem hier vorgestellten Stand des Designs bei.

Die Feasibility-Study und der Prototyp findet sich auf der beiliegenden CD.

5.3. Entwicklungsoberfläche

5.3.1. Einleitung

Das Ziel von IGold war eine vollständige Beschreibung von Gesellschaftsspielen zu ermöglichen. Aus dieser Vollständigkeit ist eine gewisse Komplexität erwachsen, die es für einen Spieleentwickler sehr schwierig macht, Spiele direkt in IGold zu definieren. Daher stellt die Entwicklungsoberfläche und die Ausarbeitung der unterschiedlichen Ansichten eine große Herausforderung dar. Um dieser Herausforderung gerecht zu werden, bedarf es dabei einer eigenen vertiefenden Studie und der Erstellung eines UI-Konzeptes, welche nicht im Rahmen dieser Arbeit behandelt werden sollen.

Innerhalb dieses Abschnittes soll jedoch gezeigt werden, mit welchen Mitteln diese Entwicklungsoberfläche realisiert werden könnte, ohne dass eine vollständige Neuentwicklung notwendig ist. Mit den bereits kurz vorgestellten Language Workbenches [1] ist es möglich, eigene Diagrammtypen in Form von Meta-Modellen zu entwickeln. Weiterhin bieten die meisten Language Workbenches bereits eine Hilfestellung zur Generierung von Code.

Komplexere Zusammenhänge werden während der Spieleentwicklung oft anhand von UML-Diagrammen verdeutlicht, daher liegt es nahe, dass die graphische Aufbereitung sich an den bekannten Formen der UML-Diagramme orientieren. Der Schwerpunkt der Typen-Ansichten liegt zum einen in der Vererbung der Typen untereinander und zum anderen in der Definition von Instanzen, Links und LinkLists. Hier wäre eine Ansicht denkbar, welche sich an ein UML-Klassendiagramm anlehnt.

Die Ablauf-Ansichten hingegen könnten sich an einem UML-Zustandsdiagramm orientieren. Im bisherigen Verlauf der Arbeit wurden diese bereits an verschiedenen Stellen eingesetzt, um Abläufe zu visualisieren. Die Abbildung 5.2 zeigt beispielhaft, wie der Ablauf der Phasen von OFuA definiert werden könnte. Von der zusätzlichen Definition der beiden Preconditions abgesehen, ergibt sich aus dieser Ansicht bereits der vollständige Spielablauf.

In dem nun folgenden Feasibility-Test soll gezeigt werden, auf welche Weise eigene Meta-Modelle erstellt werden können.

[1] siehe Abschnitt 2.1.2

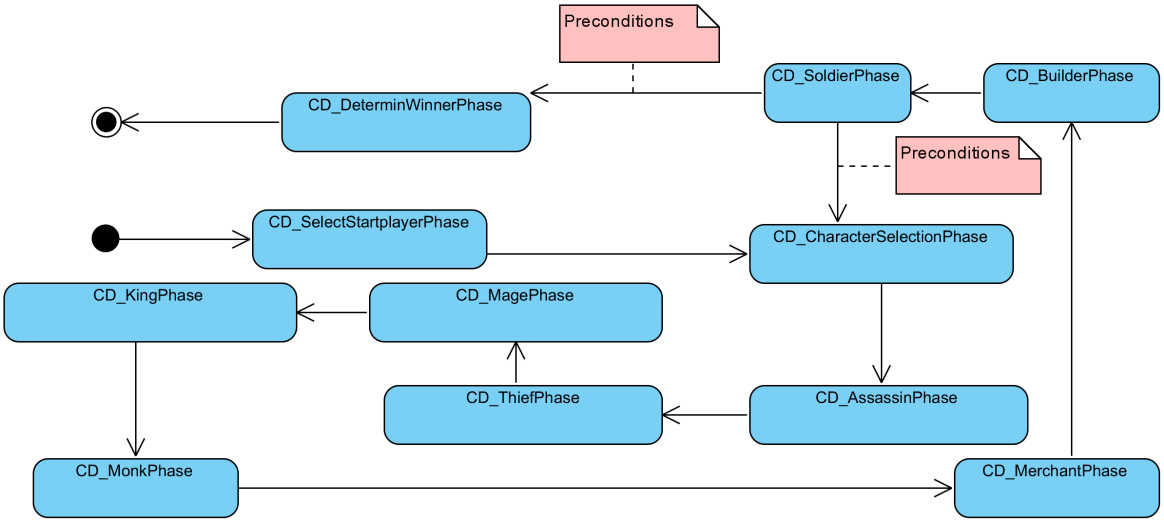


Abbildung 5.2.: Spielablauf

5.3.2. Feasibility-Test

Diagrammansicht

Für die Umsetzung des Editors eignet sich, wie bereits erwähnt, die Nutzung der sogenannten Language Workbenches. Dabei existieren zum einen eigenständige Lösungen wie z.B. MetaEdit [2], aber auch in Entwicklungsumgebungen integrierte Tools. Der Vorteil bei der Nutzung einer integrierten Lösung besteht darin, dass sich auch andere Funktionalitäten und Erweiterungen der IDE nutzen lassen. Dies ist insbesondere für die Realisierung der Debug-Funktionalität interessant, da das Debugging ein essenzieller Bestandteil einer Entwicklungsumgebung darstellt.

Die in diesem Feasibility-Test vorgestellte Ansicht wurde mithilfe der Visual Studio Visualization and Modeling SDK (VMSDK) für Visual Studio 2010 erstellt [3]. Das SDK erlaubt es, eine eigene grafische domänenspezifische Sprache zu erstellen, indem die grafischen Elemente, ihre Merkmale und Verknüpfungen untereinander, modelliert werden können. Für komplexere Zusammenhänge ist es ebenso möglich, durch definierte Schnittstellen über individuellen Code das Verhalten zu erweitern und anzupassen. Für Visual Studio ist es ebenso möglich, eine eigene Debugging-Engine zu erstellen, indem auf die Funktionalitäten der IDE aufgebaut wird [4][5].

[2] http://www.metacase.com/MetaEdit.html
[3] Erhältlich unter http://archive.msdn.microsoft.com/vsvmsdk
[4] Nähere Informationen finden sich unter http://msdn.microsoft.com/en-us/library/bb161718.aspx
[5] Anforderung IDE-3 gilt als erfüllt.

Innerhalb des VMSDK wird zwischen Domänenklassen und der graphischen Repräsentation unterschieden. Abbildung 5.3 zeigt die Modellierungsansicht des VMSDK. Im Rahmen

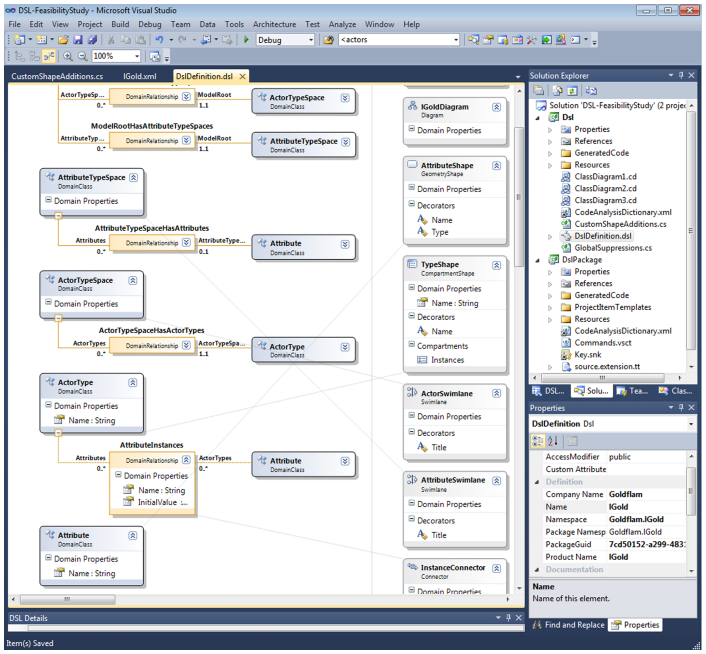


Abbildung 5.3.: Editor

dieses Feasibility-Tests wurden die notwendigen Elemente modelliert, um eine Diagrammansicht zu erzeugen, welche genutzt werden kann um **Actor**- und **Attribute**-Typen zu definieren. Zusätzlich kann ein **Actor**- über einen Connector mit einem **Attribute**-Typ verknüpft werden. Dies erzeugt eine Instanz des entsprechenden **Attribute**-Typs, die innerhalb der Actor-Form erscheint und für welche nun der Startwert eingegeben werden kann. Abbildung 5.4 zeigt das Diagramm, welches mithilfe des VMSDK erzeugt wurde.

Das hier erzeugte Diagramm kann entweder deployed oder in andere .NET Projekte integriert werden. Das VMSDK dient also lediglich dazu, die unterschiedlichen Diagrammansichten zu realisieren, und der eigentliche Editor kann mit den in .NET üblichen Werkzeugen erstellt werden[6].

Generierung von IGold

An dieser Stelle soll gezeigt werden, dass über das VMSDK alternativ auch IGold erzeugt werden kann. Für diese Überführung ermöglicht es das VMSDK, für die unterschiedlichen

[6] Erfüllt Anforderung IDE-1

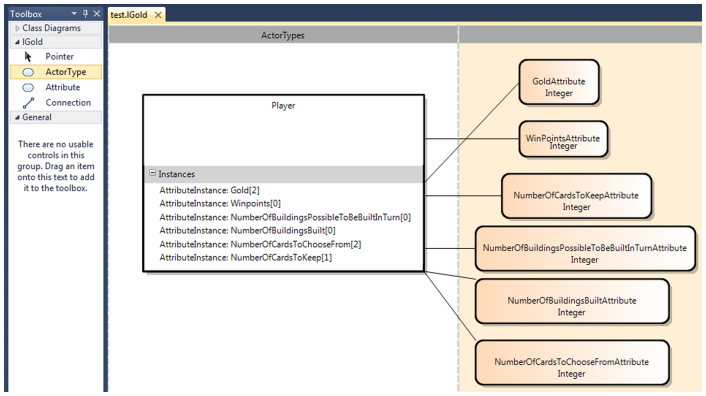


Abbildung 5.4.: Resultierende Diagrammansicht

Ansichten einen Generator zu erstellen. Dies erfolgt in Form von sogenannten T4 Text Templates. Es erlaubt, die statische Struktur des gewünschten Ausgabeformates zu definieren und die dynamischen Stellen mithilfe eines Skript-Blockes festzulegen [7].

Ein T4 Text Template für die in dem vorherigen Abschnitt vorgestellte Ansicht lässt sich folgendermaßen definieren:

```
<#@ template debug="true" inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransformation" #>
<#@ output extension=".xml" #>
<#@ lGold processor="lGoldDirectiveProcessor" requires="fileName='test.lGold';validation='open|load|save|menu'" #>
<actors>
<#
 foreach (ActorTypeSpace typeSpace in this.ModelRoot.ActorTypeSpaces)
 {
  foreach(ActorType actorType in typeSpace.ActorTypes){#>
   <actor name="<#= actorType.Name #>" typeId="type.<#= actorType.Name #>">
    <instances>
     <# foreach(Goldflam.lGold.Attribute attribute in actorType.Attributes){
      foreach(AttributeInstances attributeInstance in AttributeInstances.GetLinks(actorType,attribute)){#>
       <attributeInstance name="<#= attributeInstance.Name #>" typeReference="type.<#= attribute.Name #>"
                          instanceId="instance.<#= attributeInstance.Name #>">
        <integerValue value="<#= attributeInstance.InitialValue #>"/>
       </attributeInstance><#
      }
     }#>
    </instances>
   </actor>
  <#
  }
 }
#>
</actors>
```

Wenn dieses T4-Template auf das Diagramm in Abbildung 5.4 angewendet wird, wird aus diesem der folgende XML-Code generiert.

```
<actors>
 <actor name="Player" typeId="type.Player">
  <instances>
   <attributeInstance name="Gold" typeReference="type.GoldAttribute" instanceId="instance.Gold">
    <integerValue value="2"/>
   </attributeInstance>

   <attributeInstance name="Winpoints" typeReference="type.WinPointsAttribute"
```

[7] Nähere Informationen finden sich unter http://msdn.microsoft.com/en-us/library/bb126478

```
    instanceId="instance.Winpoints">
        <integerValue value="0"/>
   </attributeInstance>

   <attributeInstance name="NumberOfBuildingsPossibleToBeBuiltInTurn"
     typeReference="type.NumberOfCardsToKeepAttribute"
     instanceId="instance.NumberOfBuildingsPossibleToBeBuiltInTurn">
        <integerValue value="0"/>
   </attributeInstance>

   <attributeInstance name="NumberOfBuildingsBuilt"
      typeReference="type.NumberOfBuildingsPossibleToBeBuiltInTurnAttribute"
      instanceId="instance.NumberOfBuildingsBuilt">
        <integerValue value="0"/>
   </attributeInstance>

   <attributeInstance name="NumberOfCardsToChooseFrom"
       typeReference="type.NumberOfBuildingsBuiltAttribute"
       instanceId="instance.NumberOfCardsToChooseFrom">
        <integerValue value="2"/>
   </attributeInstance>

   <attributeInstance name="NumberOfCardsToKeep" typeReference="type.NumberOfCardsToChooseFromAttribute"
       instanceId="instance.NumberOfCardsToKeep">
        <integerValue value="1"/>
   </attributeInstance>
  </instances>
 </actor>
</actors>
```

Code-Generierung

Die Code-Generierung basiert auf den gleichen Mechanismen wie die Generierung von XML. Es werden T4 Text Templates genutzt, um die gewünschten Klassen zu erstellen. Das folgende Beispiel [8] zeigt, wie eine C#-Klasse generiert werden könnte.

```
<#@ template debug="false" hostspecific="false" language="C#" #>
<#@ output extension=".cs" #>
<# var properties = new string [] {"P1", "P2", "P3"}; #>
class MyGeneratedClass {
<#
  foreach (string propertyName in properties)
  { #>
  private int <#= propertyName #> = 0;
<# } #>
}
```

Da die Anwendung von T4 Text Templates bereits anhand der Generierung der IGold-XML demonstriert wurde, wird das Verfahren als allgemein durchführbar betrachtet und nicht noch einmal für die Generierung von C# Dateien im Rahmen eines eigenen Feasibility-Tests vorgeführt[9].

In IGold wird die Sprachdefinition auf sogenannte ElementTemplates abgebildet. Auf ihre Klassenstruktur und ihr Zusammenspiel wird in Abschnitt 5.5.4 ab Seite 138 eingegangen.

[8] Quelle: http://msdn.microsoft.com/en-us/library/dd820620#Y1600

[9] Anforderung IDE-2 kann als erfüllt angesehen werden. Die Generierung wurde bereits in Form von IGold-XML demonstriert.

5.4. Test-Runtime

5.4.1. Einleitung

In diesem Abschnitt wird die Test-Runtime vorgestellt. Sie soll eine schnelle Ausführung der durch die Entwicklungsumgebung definierten Spiele ermöglichen, ohne vorher eine eigene GUI realisieren zu müssen. Die Spielelogik soll dabei in der GameInstance gekapselt werden und von der Runtime entkoppelt sein. Auf diese Weise soll es möglich sein, die GameInstance in eine produktive Runtime ohne Probleme übernehmen zu können.

5.4.2. Übersicht

Die Runtime ist, wie Abbildung 5.5 zeigt, nach dem MVC Modell aufgebaut.

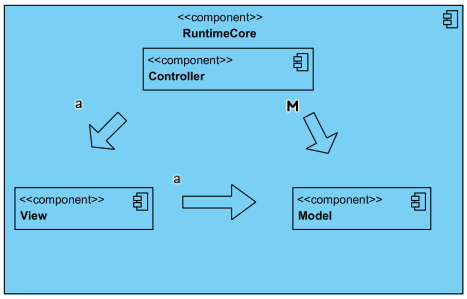


Abbildung 5.5.: Test-Runtime

Sie stellt eine primitive Visualisierungsmöglichkeit dar, bei der mehrere Spieler gleichzeitig mit dem Spiel interagieren können. Die einzelnen Systemkomponenten sollen im Folgenden skizziert werden [10].

Model

Die Runtime sollte für jeden Spieler eine eigene Repräsentation besitzen. Abbildung 5.6 zeigt, dass das `Model` für jeden Spieler ein eigenes `PlayerModel` besitzt [11]. Dieses besteht aus einem `GameState` und einer `CurrentInteraction`. Beide Elemente verweisen auf einen Container. Der `Controller` der Runtime erhält entweder den vollständigen Zustand des Spieles in Form eines `GameStateContainers`

[10] Anforderung TR-1 gilt als erfüllt.
[11] Relevant für Anforderung TR-2

oder sämtliche Änderungen seit der letzten Kommunikation im Rahmen einer Liste von `ElementChangeContainer`. Diese beiden Container können durch das `PlayerModel` verarbeitet werden. Für jedes sichtbare Element der Spielmechanik existiert ein eigener Container-Typ [12]

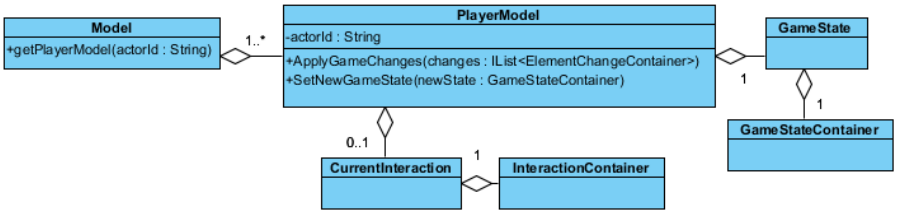


Abbildung 5.6.: Model

View

Innerhalb der `View` gibt es ebenfalls für jeden Spieler ein `PlayerView`-Objekt. Dieses unterteilt sich in zwei unterschiedliche Fenster. Die `GameStateTextView` zeigt den aktuellen Spielzustand des Spielers an. Über die `InteractionConsoleView` wird die Interaktion zwischen Spieler und Spiellogik abgewickelt. Für die Test-Runtime soll die `GameStateTextView` als Textfeld und die `InteractionConsoleView` als ein Konsolenfenster realisiert werden. Die beiden `View`-Objekte überwachen nach dem Observer-Muster die `CurrentInteraction` und den `GameState` und aktualisieren ihre Anzeigen entsprechend.

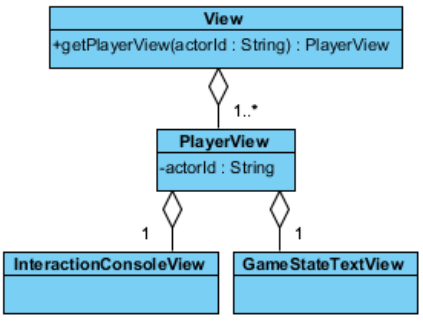


Abbildung 5.7.: View

[12] Dies wird näher in Abschnitt 5.5.4 ab Seite 149 im Rahmen des Frameworks beschrieben

Controller

Abbildung 5.8 zeigt die Komponenten, aus denen der `Controller` besteht. Die wichtigs-

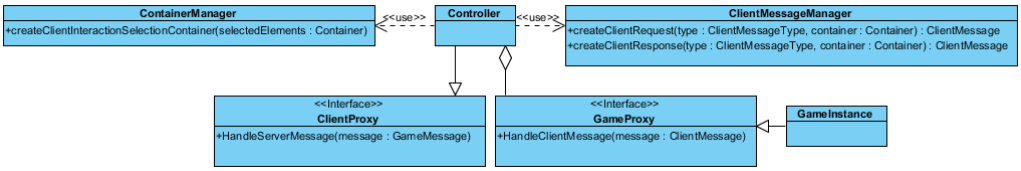


Abbildung 5.8.: Controller

te Komponente stellt die `GameInstance` dar, eine konkrete Instanz eines Spieles. Die Runtime kann nur anhand der `GameProxy`-Schnittstelle mit der `NetworkView` des Spiel kommunizieren[13][14]. Der `Controller` selbst implementiert das `ClientProxy`-Interface und ist somit für die Anfragen und Antworten der `GameInstance` verantwortlich. Über den `ContainerManager` und den `ClientMessageManager` kann der `Controller` `ClientMessages` erzeugen, über die er mit der `GameInstance` kommunizieren kann. Die in der empfangenen `GameMessage` enthaltenen `Container` werden anhand der enthaltenen `actorId` an das entsprechende `PlayerModel` weitergegeben.

Feasibility-Study

Der Prototyp erlaubt die Ausführung einer einfachen Version von OFuA in Form einer Konsolenanwendung. Zu Beginn des Spieles kann festgelegt werden, wie viele Spieler an dem Spiel teilnehmen wollen. Die Charakterauswahl, welche sich in Abhängigkeit der Spieleranzahl immer unterschiedlich verhält, ist vollständig implementiert. Nachdem jeder Spieler die notwendige Anzahl von Charakteren ausgewählt hat, beginnt die Spielphase der Charaktere. Während einer Spielphase kann ein Spieler allerdings nur Geld oder Bauwerkkarten nehmen und Bauwerke errichten sowie seinen Zug wieder beenden. Die speziellen Fähigkeiten der Charaktere und die der besonderen Bauwerke wurden nicht implementiert [15].

[13] Relevant für die Anforderung FW-2.
[14] Auf diese Kommunikation wird in Abschnitt 5.5.7 ab Seite 173 eingegangen
[15] Anforderung TR-3 gilt als erfüllt.

5.5. Framework

5.5.1. Einleitung

Das Framework soll sämtliche Kernfunktionalitäten für die in IGold definierten Spiele zur Verfügung stellen. Dabei soll zwischen dem durch den DSL-Processor generierten Code und der auszuführenden Komponente sauber getrennt werden. Das Ausführungsmodell soll nach einem Zustandsautomaten modelliert werden. Die für den Entwurf von Software-Architekturen gängigen Muster werden dabei vorausgesetzt [16].

5.5.2. Übersicht

Das Framework besteht aus drei verschiedenen Systemschichten: Der `NetworkView`, dem `CoreModel` und dem `CoreController`. Die `ElementTemplates` werden durch den DSL-Processor generiert und dem `CoreController` verwendet. Abbildung 5.9 zeigt die unterschiedlichen Komponenten und ihr Zusammenspiel.

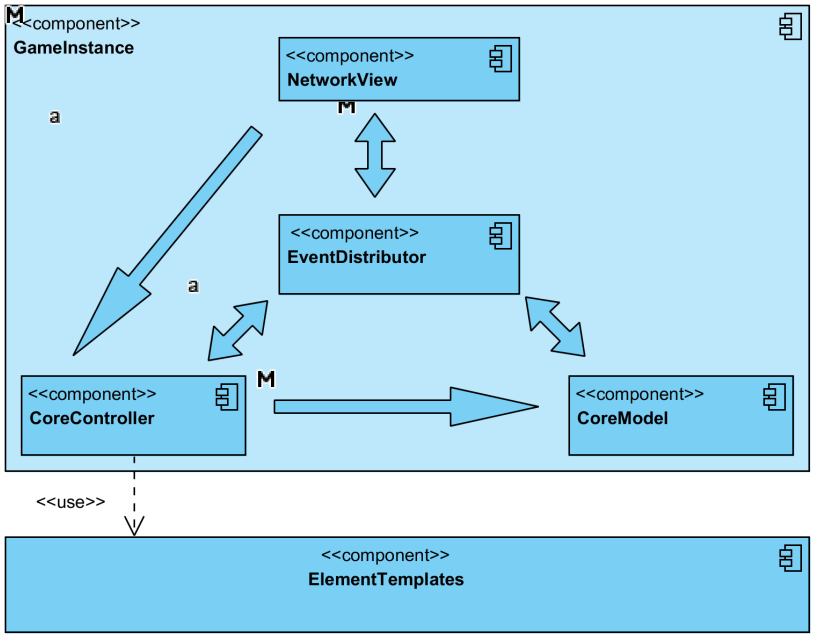


Abbildung 5.9.: Aufbau des Frameworks

[16] Bei Bedarf finden sie sich unter anderem in Gamma (1995) und Fowler (2012)

Die NetworkView hat Zugriff auf den CoreContoller, der wiederum Zugriff auf das CoreModel besitzt. Der EventDistributor wird als zusätzlicher Vermittler eingesetzt, um die Kommunikation flexibler zu gestalten und das Eintreten von Ereignissen und deren Ausführung voneinander zu entkoppeln. Änderungen an dem Model werden immer indirekt über den Controller durchgeführt. Dies ist unter anderem darauf zurückzuführen, dass der Controller durch die NetworkView erhaltenen Anfragen verarbeiten muss, bevor eine Änderung an dem Model erlaubt ist. Die NetworkView hat über den Controller Zugriff auf das Model, aber nur, um Objekte zu beziehen und diese in primitive Data-Transfer-Objects (DTOs) zu wandeln und nicht, um diese zu verändern.

GameInstance

Die GameInstance ist die Instanz eines Spieles. Sie initialisiert die Kernkomponenten und startet den Spielfluss.

NetworkView

Sie ist für die Kommunikation mit der Runtime zuständig. Für das Versenden einer Anfrage werden die diese in DTO verpackt. Beim Empfangen wird aus der Anfrage für jedes DTO das jeweilige Pendant-Objekt über den CoreController bezogen und diese dann als Antwort an den Controller zurückgegeben. So ist das Ver- und Entpacken der DTOs transparent für die anderen Komponenten. Die Anfragen werden dabei durch die Runtime gestellt und empfangen, welche die GameInstance ausführt. Erst die Runtime realisiert die eigentliche Kommunikationsschicht, welche entweder für das Verschicken der Anfragen über das Netzwerk oder für die Weitergabe an die GUI verantwortlich ist.

CoreModel

Das CoreModel enthält sämtliche Objekte, die den Spielzustand darstellen und welche gespeichert werden sollen. Zusätzlich wird für jeden Spieler abgespeichert, welche Veränderungen an den Objekten des CoreModels durchgeführt worden sind, seitdem sie das letzte Mal erfolgreich eine Anfrage gestellt haben. Der Umstand, dass für jeden Spieler die Veränderungen abgespeichert werden müssen, ist darauf zurückzuführen, dass jeder Spieler eine andere Sicht auf den Spielzustand haben kann. Die Runtime ist an dieser Stelle ebenso für die Persistenz verantwortlich.

CoreController

Der `CoreController` ist für die Steuerung des Spielflusses verantwortlich, wobei hier hauptsächlich drei Komponenten zum Einsatz kommen: Der `PhaseManager`, der `PhaseStateManager` und der `ActionStateManager`. Die anderen Manager-Komponenten sind dediziert für einen abgegrenzten Aufgabenbereich verantwortlich. Jeder Manager ist als ein Singleton realisiert.

EventDistributor

Der `EventDistributor` ist nach einer MessageQueue modelliert, die nach dem FIFO-Prinzip abgearbeitet wird. Der `EventDistributor` läuft in einem eigenen Prozess. So ermöglicht er eine asynchrone Kommunikation und entkoppelt Sender und Empfänger, wodurch das Framework flexibler gestaltet werden kann. Wie in Abbildung 5.9 bereits gezeigt wurde, wird der `EventDistributor` nicht exklusiv für die Kommunikation zwischen den Komponenten eingesetzt, sondern als zusätzlicher Kanal verwendet.

5.5.3. CoreModel

Übersicht

Das `CoreModel` repräsentiert den aktuellen Spielzustand und die Zustandsübergänge. Das Klassendiagramm 5.10 zeigt eine vereinfachte Übersicht des Aufbaus. Die Klassen oberhalb des `CoreModels` zeigen die internen Objekte, welche für die Steuerung des Spielflusses benötigt werden. Die Abbildung zeigt, dass jeweils nur eine `Phase` und ein `PhaseState`, aber gleichzeitig mehrere `Actions` durchgeführt werden können. Da `Operations` den Spielzustand manipulieren, dürfen sie nur sequenziell und nicht parallel ausgeführt werden.

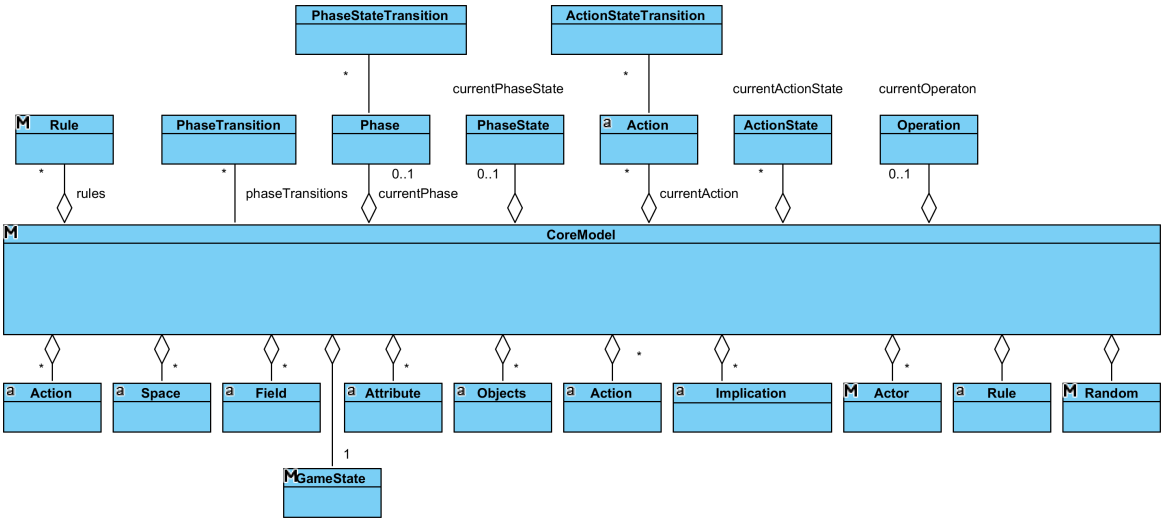


Abbildung 5.10.: Übersicht des CoreModels

Unterhalb des `CoreModels` im Diagramm findet sich für die unterschiedlichen nicht flüchtigen Objekt-Typen[17] jeweils eine Liste. Auf die Objekte dieser Listen kann über einen Hash zugegriffen werden. Diese Listen dienen als zentraler Zugriffspunkt auf sämtlich vorhanden Spiel-Elemente. Das `GameState`-Objekt spiegelt die Relationen der Objekte untereinander wider. Jedes Objekt ist also mindestens einmal in der Liste vorhanden und findet sich ebenfalls in der Hierarchie des `GameStates` wieder.

Die Komponente `ObjectManager` des `CoreController` nutzt die Listen und den `GameState` und ermöglicht es unter anderem, Anfragen nach Objekten zu definieren, welche sich relativ zu einem anderen Objekt innerhalb des `GameState` befinden.

[17] Diese Objekte sind also unabhängig von dem aktuellen Spielzustand immer vorhanden, im Gegensatz zu einer bestimmten Phase, welche nur zu einem bestimmten Zeitpunkt existiert und nach Durchführung wieder zerstört wird

Bis auf `PhaseTransition`, `PhaseStateTransition`, `ActionStateTransition`, `Rules` und `Operations`, welche zustandslose Objekte der Spiellogik sind, müssen alle anderen Objekte des `CoreModels` über die Persistenzschicht der Runtime gespeichert werden.

5.5.4. Sprachkonzepte

Einleitung

An dieser Stelle soll nicht auf das Design jedes einzelnen IGold Sprach-Elementes und dessen Klassenhierarchie, sondern auf die Architektur und Designentscheidungen zur Umsetzung der in Abschnitt 5.5.4 beschriebenen Sprachkonzepte eingegangen werden.

Element-Typen, Vererbung und Instanzen

Für die unterschiedlichen Typen der IGold-Elemente muss bestimmt werden, wie aus einer IGold-Definition der entsprechende Code generiert wird. Für jeden Element-Typ in IGold existiert eine abstrakte `ElementTemplate`-Klasse. Ein vom Spieleentwickler definiertes Element wird in Form einer konkreten Unterklasse der zugehörigen abstrakten Klasse generiert. Sämtliche durch den DSL-Processor generierten Unterklassen bilden die in Abbildung 5.9 vorgestellte ElementTemplates-Komponente. Diese Template-Klassen stellen die Essenz des Spieles dar. In einem gewissen Sinne konfigurieren sie den Zustandsautomaten der Runtime und repräsentieren so das vollständige Spiel.

Abbildung 5.11 verdeutlicht den beschriebenen Aufbau und zeigt, die vom DSL-Processor erzeugten Klassen `BuildingTemplate` und `ChurchTemplate`. Sämtliche in IGold definierten Typen werden auf diese Art abgebildet [18].

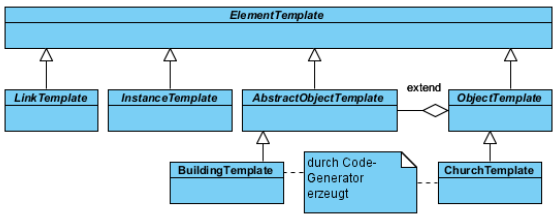


Abbildung 5.11.: Element-Typen

Abbildung 5.12 zeigt, wie aus den generierten Template-Klassen Factories, welche von der abstrakten `ElementFactory` erben, Instanzen der entsprechenden System-Klassen erzeugen. Der `ElementCreator`, selbst eine Factory, wählt in Abhängigkeit des Element-Typs die richtige Factory für eine Template-Klasse aus. Die einzelnen Factories haben dabei ebenso Zugriff auf den `ElementCreator`, um die Objekte zu erzeugen, welche sie enthalten. Sämtliche hier vorgestellten Factories sind als Singletons realisiert. Der `ElementCreator` ist Teil des `ObjectManager`.

[18] Anforderung FW-1 gilt als erfüllt.

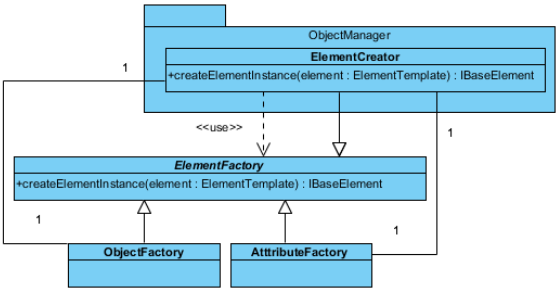


Abbildung 5.12.: Element-Factory

Abbildung 5.13 zeigt exemplarisch, wie jene System-Klassen, welche die IGold-Elemente repräsentieren, modelliert sind. Die durch die Sprachdefinition vorgegebenen strukturellen und funktionalen Vorgaben werden in Form von Schnittstellen definiert. Das Interface `IBaseElement` stellt den kleinsten gemeinsamen Nenner dar, da jedes Element innerhalb der Definition ein Eltern- und mehrere Kindelemente haben kann. Die Klassen, welche instanziierbare Elemente wie **Objects** repräsentieren, implementieren das `IInstantiableElement`-, eine Operation hingegen das `IExecutableElement`- oder eine Regel das `IRuleElement`-Interface. Auf diese Weise werden die generierten und spielspezifischen Unterklassen aus Abbildung 5.11 abstrahiert und auf die jeweiligen internen Klassen abgebildet.

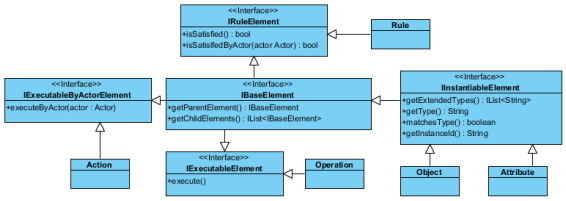


Abbildung 5.13.: System-Objekte

Der hier vorgestellte Aufbau trennt die Definitionsgrundlage der Template-Klasse und die Instanziierung voneinander. Auf diese Weise lassen sich mögliche Erweiterungen der Sprache flexibel umsetzen. Die durch die Sprache notwendigen Rahmenbedingungen werden in Form von Schnittstellen realisiert, um die Systemklassen und ihren strukturellen Aufbau flexibel zu halten.

Die in Abschnitt 4.3.2 vorgestellten Sprachkonzepte lassen sich auf die hier beschriebene Weise abbilden[19].

[19] Anforderung FW-3 kann in Zusammenhang mit dem ElementCreator als erfüllt angesehen werden.

Abläufe der Spielmechanik

An dieser Stelle sollen die Zusammenhänge der unterschiedlichen Klassen skizziert werden: Der Spielablauf selbst wird durch die `PhaseManager`, `PhaseStateManager` und `Action` gesteuert und im weiteren Verlauf vorgestellt. Abbildung 5.14 zeigt die Klassen, welche für den Spielfluss auf der Ebene der **Phases** beteiligt sind. Nachdem eine **Phase** abgeschlossen ist, werden sämtliche `PhaseTransition` geprüft. Eine `PhaseTransition` ist nur erfüllt, wenn sämtliche `Rules` erfüllt sind. Über die erfüllte `PhaseTransition` kann der `PhaseManager` das zu der Phase zugehörige `PhaseTemplate` beziehen und anhand des `ElementCreators` und der `PhaseElementFactory` das auszuführende Phasen-Objekt erstellen. Bei diesem Prozess darf immer nur genau eine `PhaseTransition` erfüllt werden, ansonsten gilt dies als fehlerhafter Zustand und nach Erzeugung eines schweren Fehlers wird das Spiel terminiert.

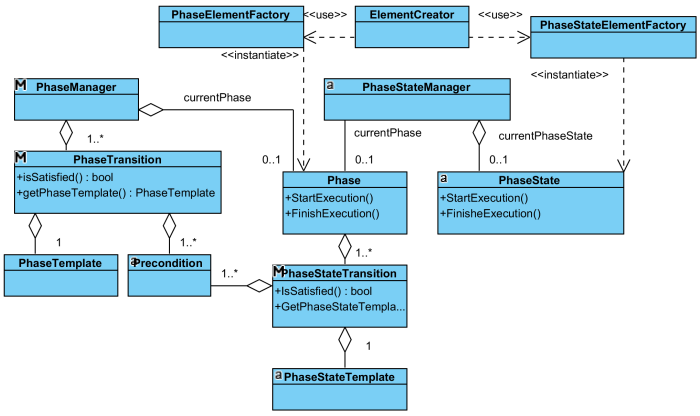


Abbildung 5.14.: Phasen-Klassen

Die Modellierung von **PhaseStates** und **ActionStates** verhält sich analog zu dem beschriebenen Aufbau der **Phases**.

Der Ablauf der unterschiedlichen Prozesse wird immer in Pre-, Main- und Post-Abschnitten abgearbeitet. Der Pre-Abschnitt signalisiert, dass als nächstes der Main-Abschnitt abgearbeitet wird, der Post-Abschnitt zeigt an, dass der Main-Abschnitt vollständig durchgeführt wurde. Abbildung 5.15 zeigt anhand eines Zustandsdiagramms, wie sich die Durchführung eines Spieles abbilden lässt.

Die verschiedenen Controller können dabei auf die unterschiedlichen Zustände reagieren und bilden so den Spielfluss ab[20].

[20] Anforderung FW-4 kann in Zusammenhang mit dem PhaseManager, PhaseStateManager und ActionManager als erfüllt angesehen werden.

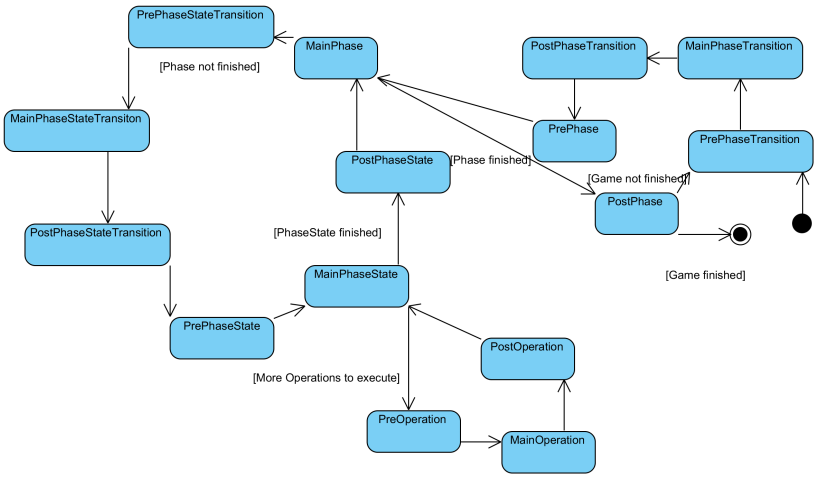


Abbildung 5.15.: Spielablauf in Pre-, Main- und Post-Abschnitten

Kontext und Selektoren

Der Kontext repräsentiert jene Informationen, welche in einer bestimmten Situation zur Verfügung stehen. Grundsätzlich ist es entweder möglich, auf die Objekte des Spieles von „oben“, also über das `GameState`-Objekt, oder durch eines der anderen Objekte von „unten“ aus zuzugreifen.

Abbildung 5.16 zeigt, auf welche Art die Klassen der Sprach-Elemente die Funktionalität des Kontextes implementieren. Über die einfache Hierarchie, welche anhand der bereits in Abbildung 5.11 vorgestellten `IBaseElement`-Schnittstelle zur Verfügung gestellt wird, kann von jedem Element aus ein bestimmter Kontexttyp gesucht werden. Wird kein Kontext dieses Typs gefunden, wird ein Fehler ausgelöst. Die Sprachstruktur garantiert, dass innerhalb der Sprachhierarchie, welche durch `IBaseElement` abgebildet wird, niemals zwei Kontextarten desselben Typs existieren, so lange die Kontextsuche in Richtung der Parent-Objekte durchgeführt wird. Der Umstand, dass dieselben Objekte den Kontext darstellen, welche auch die Sprachelemente repräsentieren, ist sehr wichtig. Würde der Kontext und seine Struktur anhand eigener Klassen von den eigentlichen Objekten entkoppelt werden, müssten Veränderungen eines Objektes auch im Kontext immer mitgepflegt werden. Dies wäre beispielsweise der Fall, wenn ein **Object** von einem **Field** in ein anderes **Field** verschoben wird.

Der eigentliche Zugriff auf ein gewünschtes Objekt erfolgt über einen Selektor, welcher über den Kontext nur die Selektion durchführt. Abbildung 5.17 zeigt eine Auswahl unterschiedlicher Selektor-Klassen. Die unterschiedlichen Selektoren werden von der `SelectorFactory`, welche Teil des `ObjectManager` ist, erzeugt. Ihre Parameter, wie z.B. die Instanz-Id eines zu selektierenden Attributes, wird der generierten Template-Klasse

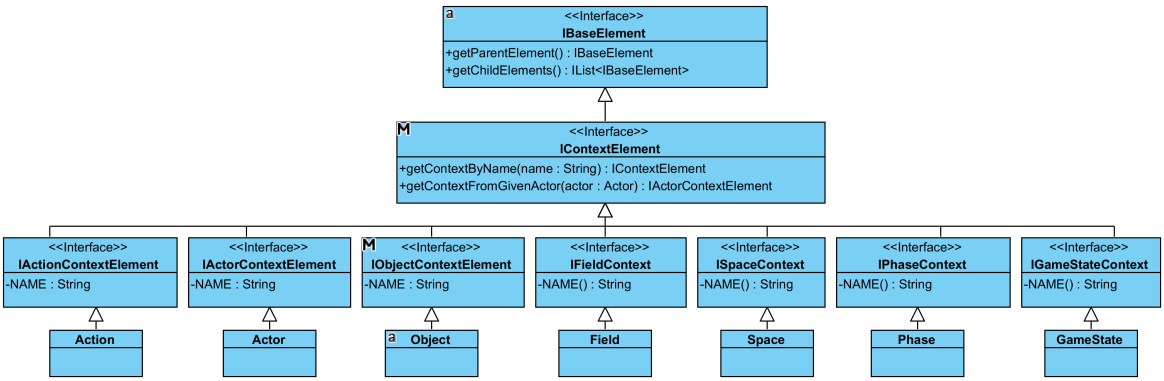


Abbildung 5.16.: Kontext-Klassen

entnommen. Die Ergebnisse des Selektors können sich dabei in Abhängigkeit des Kontextes verändern.

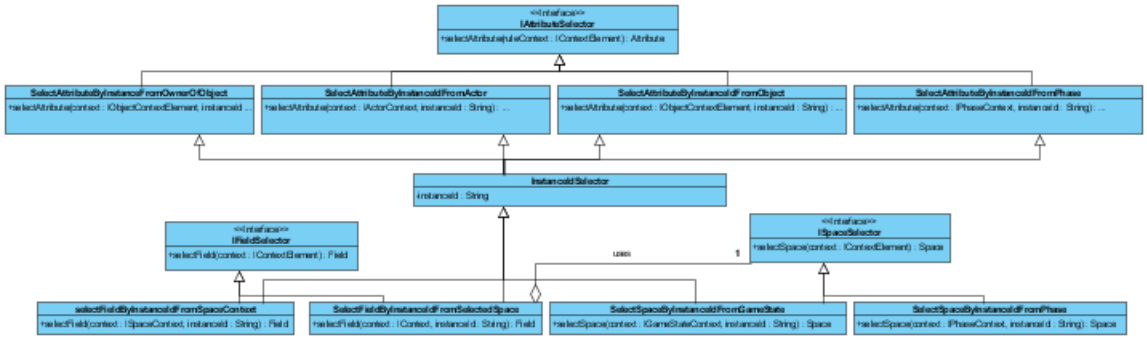


Abbildung 5.17.: Selektoren-Klassen

Die Selektoren können miteinander verknüpft werden, wie z.B. `SelectFieldByInstanceIdFromSel` welche ein Field-Objekt von einem Space-Objekt selektiert, welcher wiederum von einem `SpaceSelector` ausgewählt wird. Auf diese Weise kann ein Field beispielsweise über `SelectSpaceByInstanceIdFromGameState` von einem **Space** aus dem **GameState** oder über `SelectSpaceByInstanceIdFromPhase` von einem **Space** aus der aktuellen **Phase** bezogen werden.

Das hier vorgestellte Design bildet die in Abschnitt 4.3.5 vorgestellten Sprachkonzepte ab [21].

[21] Anforderung FW-5 kann in Zusammenhang mit dem ObjectManager als erfüllt angesehen werden.

Regeln

Die Vorbedingungen, welche in IGold durch das Element **Rule** repräsentiert werden, müssen eine flexible Struktur aufweisen. Die unterschiedlichen Einsatzgebiete zeigen, in was für unterschiedlichen Kontexten die Vorbedingungen eingesetzt werden. Zusätzlich soll die Vorbedingung nur validieren, wenn sämtliche Regeln erfüllt sind. Die Abbildung 5.18 zeigt, wie eine Vorbedingung unter Verwendung des Decorator-Patterns modelliert werden kann. Die `Precondition` gibt auf die Methoden `isSatisfied` und `isSatisfiedByActor` nur `true` zurück, wenn vorher alle `RuleDecorator` ebenso erfüllt sind. Diese können wie, im Decorator-Pattern üblich, über die Instanz des „Eltern“-Objektes miteinander verkettet werden. Die Komplexität der Regeln und die Verkettung selbst wird auf diese Art transparent gestaltet. Die Methode `isSatisfiedByActor` wird für sämtliche Prüfungen eingesetzt, bei der die Vorbedingungen für einen bestimmten **Actor** validiert werden sollen. Zum Beispiel, ob ein Spieler eine **Action** ausführen kann. Die Methode `isSatisfied` wird eingesetzt, wenn sich die Prüfung auf den allgemeinen Spielzustand und nicht auf einen bestimmten **Actor** bezieht.

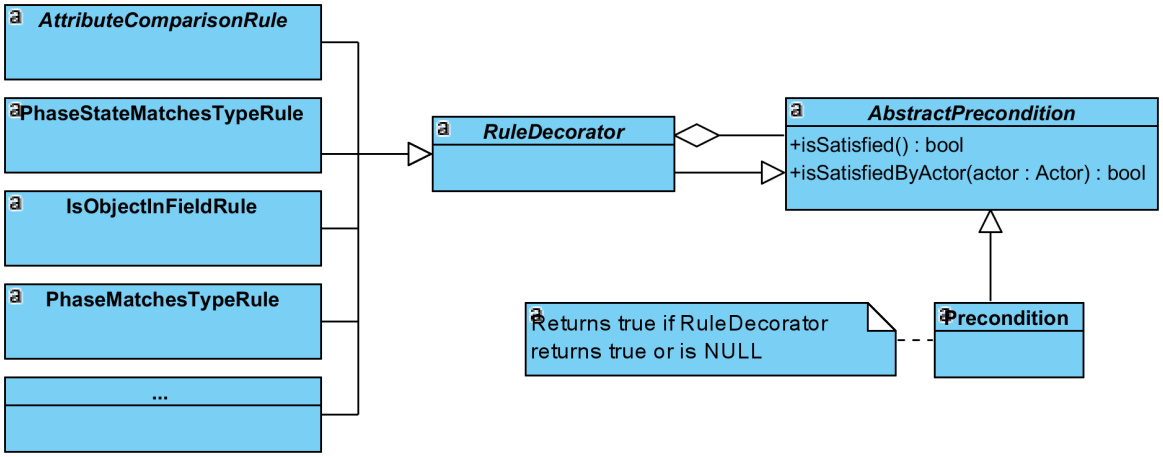


Abbildung 5.18.: Preconditions

Das Decorator-Pattern ermöglicht das „Mischen“ der unterschiedlichen Regeltypen. Noch nicht beschrieben wurde die Art und Weise, wie die unterschiedliche Funktionalität der einzelnen Regeln modelliert werden kann. Da ein hoher Grad an Wiederverwendbarkeit und Flexibilität gewährleistet werden soll, wird das Strategy-Pattern verwendet, um die Anforderungen zu realisieren. Die Abbildung 5.19 zeigt, wie sämtliche **Rules** definiert werden können, bei denen zwei **Attributes** miteinander verglichen werden sollen.

Die `AttributeComparisonRule` besteht aus vier unterschiedlichen Typen:

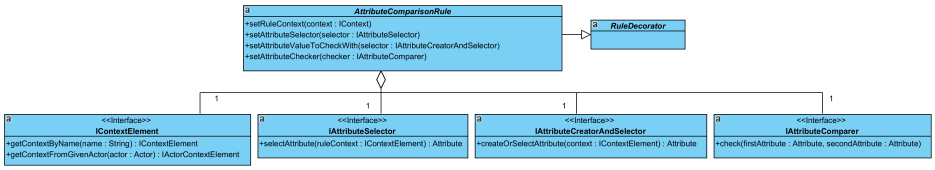


Abbildung 5.19.: AttributeCheckDecorator

IContextElement: Der Kontext der Regel ist ein Element der Spielmechanik, auf welches sich die Prüfung bezieht.

iAttributeSelector: Selektiert ein **Attribute**, welches mit einem anderen **Attribute** verglichen werden soll.

IAttributeCreatorAndSelector: Selektiert das „andere“ **Attribute**.

IAttributeComparer: Bestimmt, wie die beiden **Attributes** verglichen werden sollen.

Für jeden Typ existiert eine konkrete Klasse, welche ein bestimmtes Verhalten implementiert. Die Abbildung 5.20 zeigt, welche konkreten Klassen bei einer Regel eingesetzt werden würden, wenn geprüft werden soll, ob der Besitzer eines Bauwerkes genügend Gold besitzt, um es errichten zu können. Zur Prüfung dieser Regel sollen zwei **Attributes** verglichen werden, nämlich das Gold-Attribut des Besitzers des Objektes und das Kosten-Attribut des Objektes, auf welche die Regel angewendet wird. Dabei soll das Gold-Attribut größer gleich des Kosten-Attributs sein.

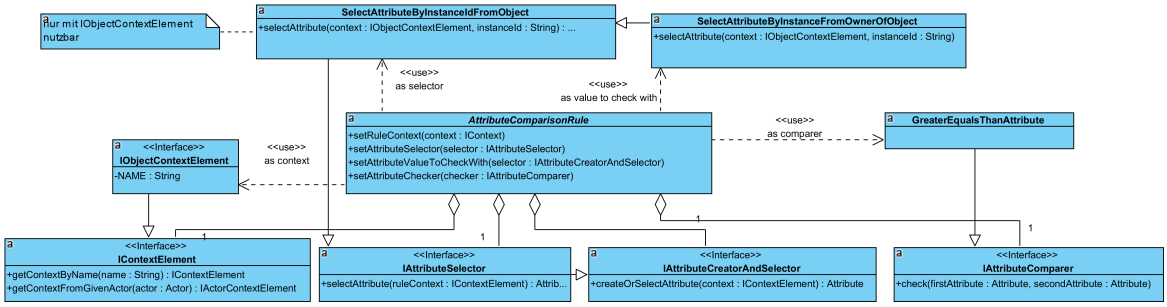


Abbildung 5.20.: Beispieler einer konkreten Regel

Für das `IContextElement`-, `IAttributeSelector`-, `IAttributeCreatorAndSelector`- und `IAttributeComparer`-Interface gibt es unterschiedliche Implementierungen, aus denen die `AttributeComparisonRule` zusammengesetzt wird. Im oberen Teil von Abbildung 5.20 sind diese konkreten Klassen zu sehen. Die Erzeugung der Regel-Instanzen wird mithilfe einer Factory realisiert, welche die

Komplexität der Kombinationsmöglichkeiten versteckt und eine enge Kopplung zwischen Erzeuger und Strategieimplementierung verhindert. Diese Factory gehört zu dem bereits vorgestellten `ElementCreator`, welcher eine Teil des `ObjectManager` ist [22].

Beeinflussung des Spielzustandes

Neben der Steuerung des Spielflusses kann der Zustand des Spieles durch **Operations** und durch **Implications** beeinflusst werden. Die **Operations** werden in **InternalOperation** und **InteractiveOperation** eingeteilt. Eine **InteractiveOperation** benötigt im Gegensatz zu einer **InternalOperation** einen vorgelagerten Zeitabschnitt, bei dem ein Spieler eine Entscheidung trifft, welche die anschließende Ausführung beeinflusst [23].

Operations

Die Abbildung 5.21 zeigt den exemplarischen Aufbau einer **InternalOperation**. Wie bereits bei den Vorbedingungen wird hier das Strategy-Muster genutzt, um die unterschiedlichen Abschnitte der Ausführung zu modularisieren. An dieser Stelle ist es gut sichtbar, inwiefern sich der Einsatz dieses Musters lohnt, da die Kontext- und Selektor-Klassen, welche auch bei den Regeln zum Einsatz kommen, hier ebenso wiederverwendet werden können. Die eigentliche Ausführung der Funktionalität wird durch die Utility-Klasse `CoreOperationUtil` durchgeführt. Die Abbildung zeigt dabei zwei unterschiedliche Operations: Die erste mischt sämtliche **Objects** eines **Spaces** und die zweite verschiebt eine bestimmte Anzahl von **Objects** von einem zu einem anderen **Space**.

Eine **InteractiveOperation** stellt einen mehrstufigen Prozess dar. Ein oder mehrere Spieler müssen eine Anfrage beantworten und anschließend wird die Antwort von der Operation verarbeitet. Wenn sämtliche Spieler geantwortet haben, wird die Operation abgeschlossen. Der eigentliche Ablauf einer `InteractiveOperation` wird in Abschnitt 5.5.6 anhand des `PhaseStateManagers` vorgestellt. Die Abbildung 5.22 zeigt die beispielhafte Klassen-Struktur, bei der ein Spieler aus einer Menge von **Objects** auswählen muss. Anschließend werden die ausgewählten und die nicht-ausgewählten **Objects** in einen **Space** verschoben.

Über den `IMultipleActorSelector` können die **Actors** selektiert werden, welche die Auswahl treffen sollen. Die abstrakte `InteractiveObjectOperation` besteht aus zwei `IObjectHandler`-Objekten, welche festlegen, wie die ausgewählten und nicht ausgewählten **Objects** verarbeitet werden sollen. Die `interactiveObjectSelection` stellt die konkrete Implementierung der `InteractiveObjectOperation`

[22] Anforderung FW-6 kann in Zusammenhang mit dem ObjectManager als erfüllt angesehen werden.

[23] Folgend kann die Anforderung FW-7 in Zusammenhang mit dem OperationManager, CoreOperationUtil, PhaseStateManager, ActionManager und ImplicationManager als erfüllt angesehen werden.

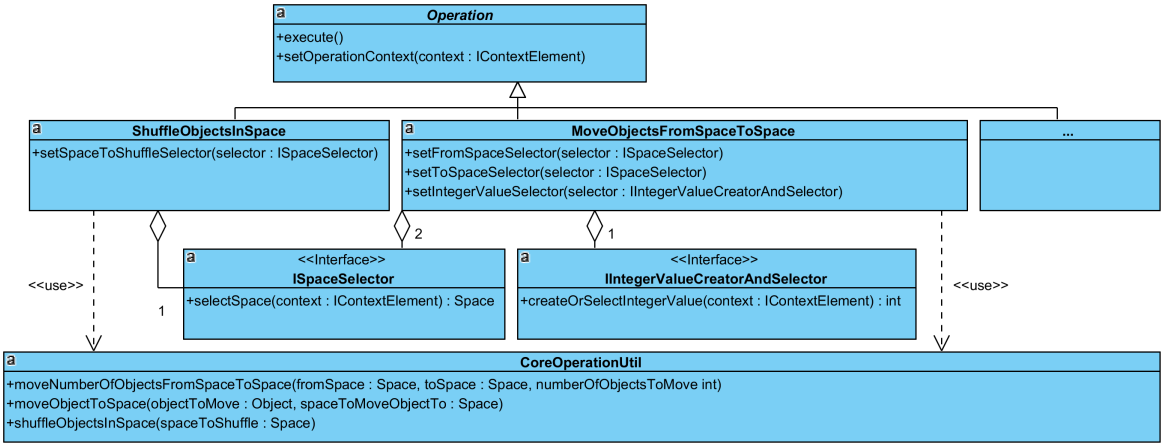


Abbildung 5.21.: Klassendiagramm einer Operation

dar und ermöglicht es über den `IFieldSelector`, eine bestimmte durch `IIntegerValueCreatorAndSelector` gegebene Anzahl von **Objects** zur Auswahl herauszusuchen.

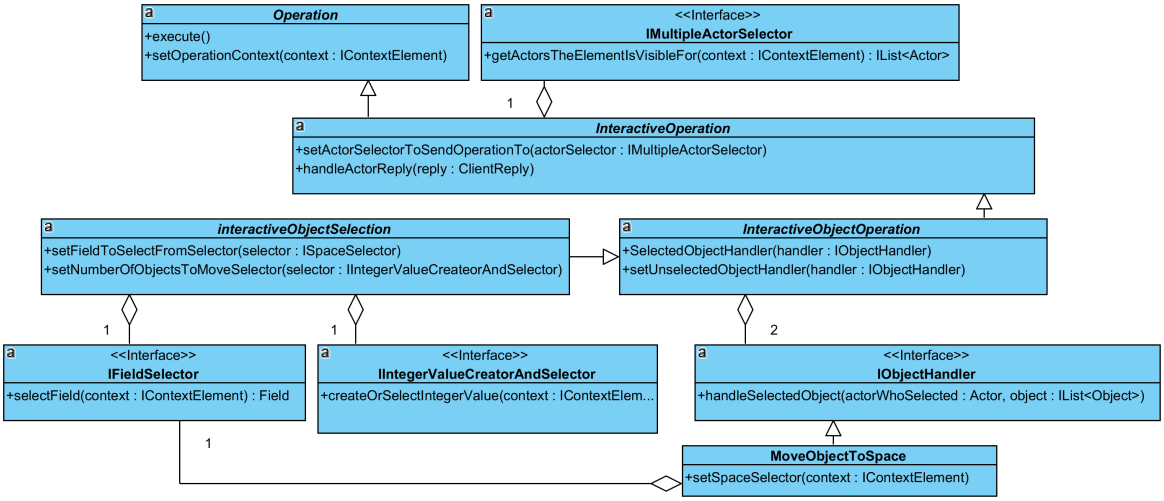


Abbildung 5.22.: Klassendiagramm einer interaktiven Operation

Im Gegensatz zu den `Rules` wird eine `Operation` erst erzeugt, bevor sie ausgeführt wird. Nach der Ausführung wird sie entsprechend vernichtet.

Actions

Die Klassenstruktur der beiden **Action**-Typen `SimpleAction` und `ComplexAction` wird in Abbildung 5.23 dargestellt. Für eine `Action` muss über einen `IMultipleActorSelector`

bestimmt werden, welche Spieler überhaupt für die Ausführung dieser `Action` in Frage kommen. Falls diese Spieler eine der `Preconditions` erfüllen, können sie die **Action** ausführen. Für jede `Action` wird dabei gespeichert, wie oft sie innerhalb des gesamten Spiels, innerhalb dieser **Phase** und innerhalb dieses **PhaseStates** insgesamt und pro **Actor** ausgeführt wurde. Diese Zahlen können beispielsweise zur Erstellung einer **Rule** genutzt werden, welche sicherstellt, dass eine **Action** nur einmal pro **Phase** ausgeführt werden darf.

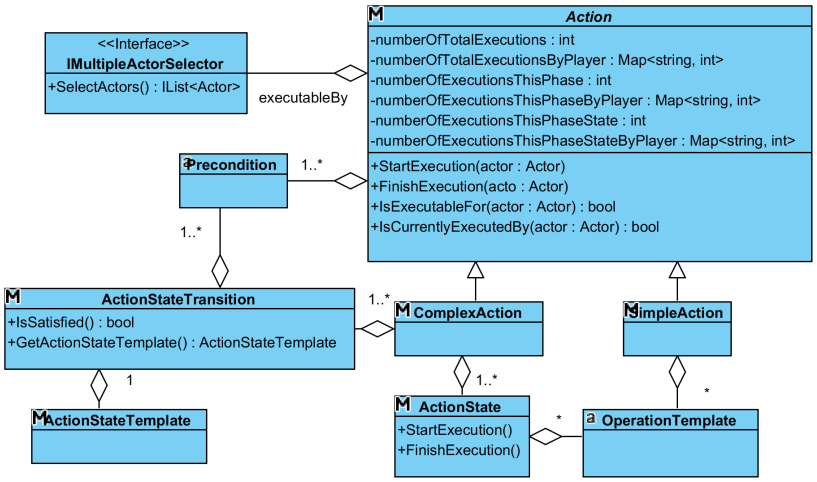


Abbildung 5.23.: Actions

Eine `SimpleAction` besteht nur aus einer Menge von `OperationTemplates`, welche zur Durchführung einer `Operation` benötigt werden. Die `ComplexAction` besitzt zusätzlich `ActionStateTransitions`, wobei jede ein `ActionStateTemplate` enthält. Anhand dieses Templates wird, wie bereits bei der Vorstellung des Spielflusses ab Seite 140 beschrieben, der `ActionState` durch den `ElementCreator` erzeugt und seine `Operations` ausgeführt.

Implications

Abbildung 5.24 zeigt die Klassen-Struktur der `Implication`. Sie besteht aus einer Anzahl von `Preconditions`, welche festlegen, ob der `Impact` sich auswirken sollte. Der `Impact` besteht aus einer Menge von `AttributeImpacts` und `VisibilityImpacts`.

Zunächst soll an dieser Stelle die Struktur eines `IntegerAttributes`, in Abbildung 5.25 zu sehen, vorgestellt werden. Das `AbstractIntegerAttribute` ist nach dem Decorator-Muster modelliert und erlaubt so, den Wert des eigentlichen `IntegerAttribute` dynamisch durch einen `IntegerAttributeDecorator` zu verändern.

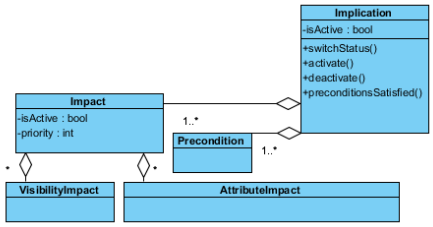


Abbildung 5.24.: Implication

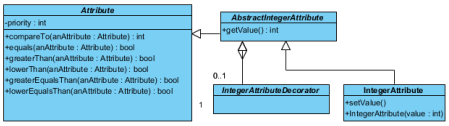


Abbildung 5.25.: IntegerAttribute

Dieser bietet die in Abbildung 5.26 dargestellten Möglichkeiten. Der Wert kann erhöht, verringert, gesetzt, geteilt und multipliziert werden.

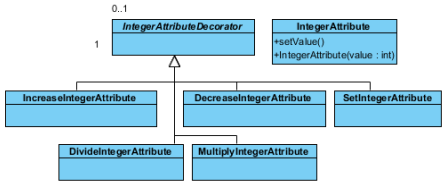


Abbildung 5.26.: IntegerAttributeDecorator

Die Struktur des `AttributeImpacts` wird in Abbildung 5.27 dargestellt. Ein `AttributeImpact` besteht demnach aus einem `IAttributeSelector` und einem `IntegerAttributeDecorator` und ist nach dem Strategy-Muster modelliert. Auf diese Weise kann die `AttributeImpact`-Klasse unterschiedliche Auswirkungen haben.

Die Reihenfolge spielt dabei eine wichtige Rolle, wie z.B. bei einem `AttributeImpact` mit einem `MultiplyIntegerAttribute` als `IntegerAttributeDecorator`. Die Reihenfolge wird durch die geerbte `priority` der `Attribute`-Klasse festgelegt. Wenn ein `IntegerAttributeDecorator` ein `AbstractIntegerAttribute` dekoriert, wird dies nicht wie gewöhnlich im Decorator-Muster von „außen“ hinzugefügt, sondern vor dem Decorator, welcher eine niedrigere `priority` als der neue Decorator besitzt. Auf diese Weise sind die `IntegerAttributeDecorator` mit hoher Priorität in

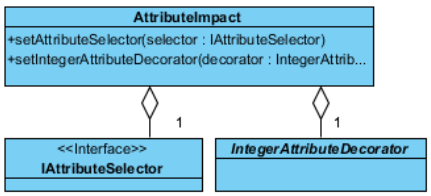


Abbildung 5.27.: AttributeImpact

der Hierarchie weiter oben. Das eigentliche `IntegerAttribute` besitzt immer die Priorität 0 und ist somit immer das letzte Element.

Sichtbarkeit

Für die meisten Elemente der Spielmechanik muss die Sichtbarkeit definiert werden. Bei der Sichtbarkeit spielt die Hierarchie der Elemente eine entscheidende Rolle, denn ein Element gilt nur als sichtbar für einen Spieler, wenn sämtliche übergeordneten Elemente ebenfalls für ihn sichtbar sind. Die Sichtbarkeit dient dazu, für jeden Spieler automatisch einen Spielzustand erzeugen zu können, welcher nur für ihn sichtbare Spielelemente enthält.

Es gibt dabei Spiel-Elemente, welche immer sichtbar sind. Dazu gehören die Klassen `Phase`,`PhaseState` und `ActionState`. Für die Klassen `Attribute`, `GameObject`, `Field`, `Space`, `Actor`, `Action` und `Effect` ist die Sichtbarkeit optional. Dabei ist zu beachten, dass manche Elemente, z.B. ein Attribut, unterschiedlichen Elementen zugeordnet werden können, im Falle des Attributs unter anderem dem `GameState`, einem `Actor`, einer `Phase` und anderen Elementen. Zunächst soll die Sichtbarkeit modelliert und anschließend die Verschachtelung durch die Objekt-Hierarchie beschrieben werden.

Das Verhalten der Sichtbarkeit soll einer Menge von Klassen hinzugefügt werden. Um eine flexible Vererbungsstruktur beizubehalten, wird die Funktionalität durch Verwendung des Delegate-Musters in eine eigene Klasse ausgelagert und der eigentliche Methodenaufruf an diese Klasse delegiert. Abbildung 5.28 zeigt, wie verschiedene Klassen das Interface implementieren und jeweils eine Instanz desselben besitzen. Da die `IVisibleElement`- die `IInstantiableElement`-Schnittstelle erweitert, kann für die Sichtbarkeit auf der Hierarchie der Spiel-Objekte aufgebaut werden. Ob ein Spiel-Objekt dabei die `isElementVisibleForActor(actor : Actor)`-Methode erfüllt, hängt entsprechend davon ab, ob sämtliche Eltern-Objekte ebenso sichtbar sind.

Die Sichtbarkeit selbst wird wiederum, wie bereits bei den Regeln geschehen, durch die Verwendung des Strategy-Musters modelliert. Abbildung 5.29 veranschau-

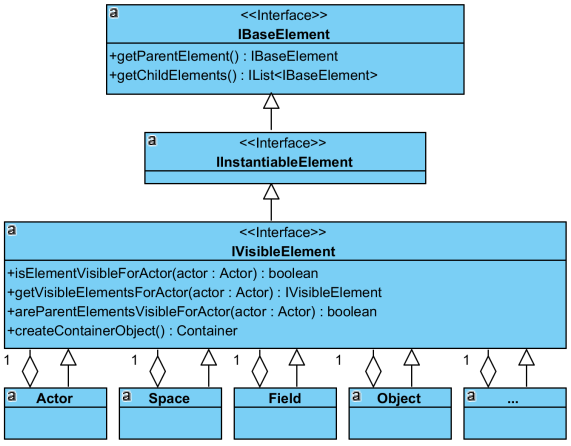


Abbildung 5.28.: Sichtbarkeit als Delegate

licht die Klassenstruktur. Für die Klassen, die immer als sichtbar gelten, wird das `AlwaysVisibleElement` erstellt. Für die anderen Elemente der Spiel-Mechanik kann durch einen `IMultipleActorSelector` bestimmt werden, welche Spieler dieses Element sehen können.

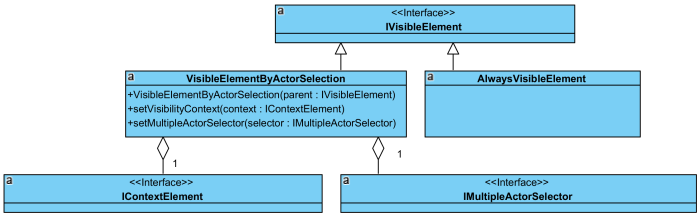


Abbildung 5.29.: Sichtbarkeitprüfung anhand des Stategy-Musters

Die für einen Spieler sichtbaren Elemente müssen in eine Datenrepräsentation überführt werden, bevor sie an den Spieler geschickt werden können. Dies erfolgt über die `createContainerObject()`-Methode des `IVisibleElement`-Interfaces. Die Abbildung 5.30 zeigt, wie die Container über den `ContainerManager` erzeugt werden.

Die Container selbst stellen, wie in Abbildung 5.31 zu sehen, eine Repräsentation dar, welche nur aus primitiven Datentypen besteht und das Ziel verfolgt, die Größe des zu verschickenden Spielzustandes möglichst zu minimieren. Ihre Modellierung entspricht einem Data Transfer Object (DTO).

Die in Abschnitt 4.3.8 vorgestellten Sprachkonzepte lassen sich auf die hier beschriebene Weise abbilden[24].

[24] Anforderung FW-8 kann in Zusammenhang mit dem VisibilityManager und ContainerManager als erfüllt angesehen werden.

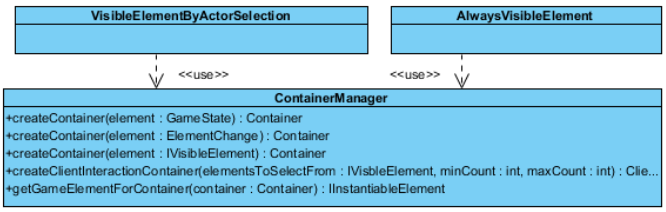


Abbildung 5.30.: Container-Factory

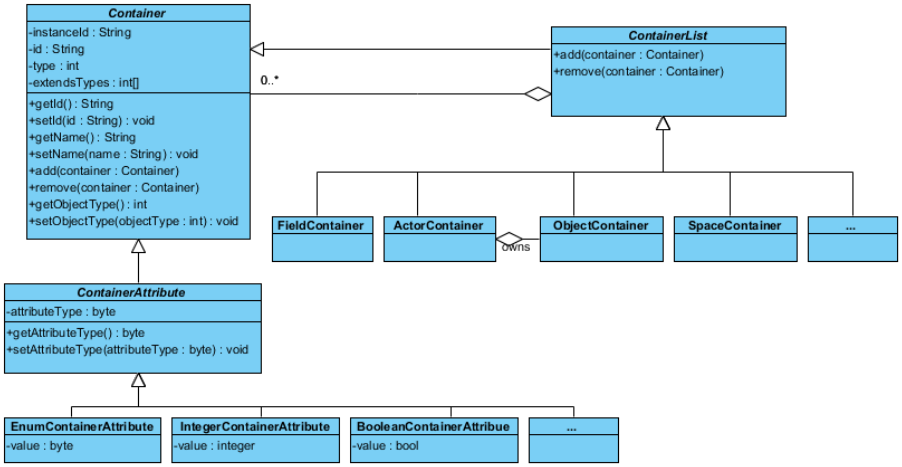


Abbildung 5.31.: Container

Timer

Der **Timer** soll garantieren, dass nach einer definierten Zeit bestimmte Vorbedingungen erfüllt sind, indem die Ausführung von **actions** und **operations** erzwungen wird. Der `TimerManager` ist für die Verwaltung der `TimedTasks` zuständig. Ein `TimedTask` besteht dabei aus zwei Bestandteilen: Einer Komponente, welche das Verhalten bestimmt, nachdem der `TimedTask` abgelaufen ist, und einer Komponente, welche in einer bestimmten Situation die Ausführung des `TimedTask` abbricht. Abbildung 5.32 zeigt den Aufbau samt dem `IAbortHandler` und dem `ITimeoutHandler`.

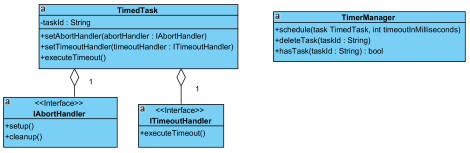


Abbildung 5.32.: Grundlegender Timer-Aufbau

Der `IAbortHandler`, zu sehen in Abbildung 5.33, dient dazu, den `TimedTask` nach dem Eintreten bestimmter Ereignisse abzubrechen. Der `IAbortHandler` registriert sich dabei für bestimmte Events des `EventDispatchers`. Der hier vorgestellte `AbortAfterEventsReceivedHandler` bricht die Ausführung des `TimedTask` ab, nachdem die geforderten Events eingetreten sind. Da sich der Handler nur für unterschiedliche Event-Typen registrieren kann (wie z.B. ActionExecutionFinishedEvent), muss geprüft werden, ob dieses Event auch die gewünschten Vorbedingungen erfüllt. Erfüllt ein Event die Vorbedingungen, wird es verarbeitet, ansonsten wird es ignoriert. Die Vorbedingungen sind dabei, wie die `Rules`, nach dem Decorator- und dem Strategy-Pattern modelliert.

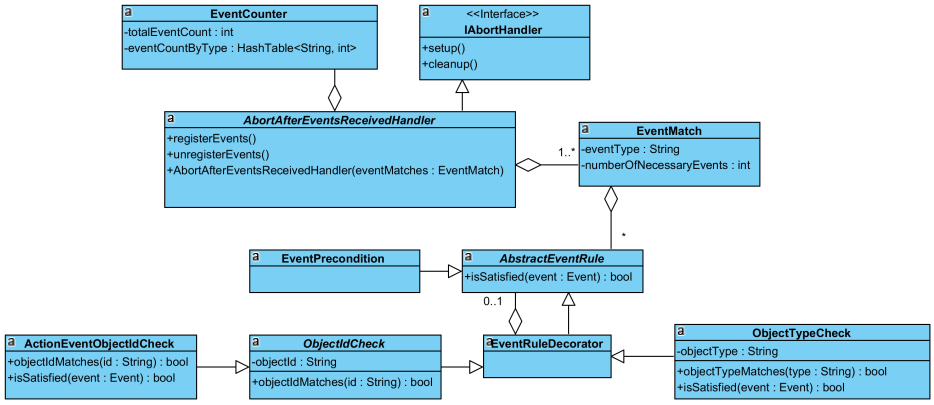


Abbildung 5.33.: AbortHandler

Der `TimeoutHandler`, zu sehen in Abbildung 5.34, unterstützt derzeit zwei unterschiedliche Implementierungen: `ExecuteOperationOnTimeout` und `EnsureActionExecutionFinishedOnTimeout`. Der `ExecuteOperationOnTimeout` führt einfach sämtliche **Operations** aus, welche er enthält. Der `EnsureActionExecutionFinishedOnTimeout` hingegen führt nur jene **Actions** stellvertretend für die **Actors** aus, welche in `ActionToExecute` festgelegt sind. Dabei kann eine **Action** auch mehrfach und für mehrere **Actors** ausgeführt werden [25].

Spielfelder

Spielfelder werden, wie bereits beschrieben, anhand von **Layouts** gebildet. Ein **Space** kann dabei ein spezifisches **Layout** und zusätzlich ein **CustomLayout** besitzen. Abbildung 5.35 zeigt, wie die unterschiedlichen Layout-Typen anhand des Strategy-Musters angewendet werden können. Die Layout-Typen `OneDimensionalLayout`, `TwoDimensionalLayout`, `ThreeDimensionalLayout` und das `CustomLayout` sollen im Folgenden näher beschrieben werden.

[25] Die Anforderung FW-9 kann in Zusammenhang mit dem TimerManager als erfüllt angesehen werden

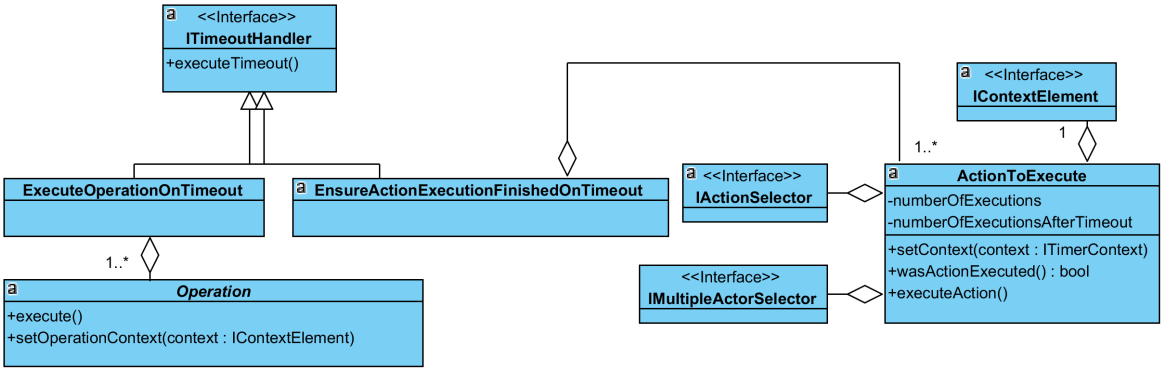


Abbildung 5.34.: TimeoutHandler

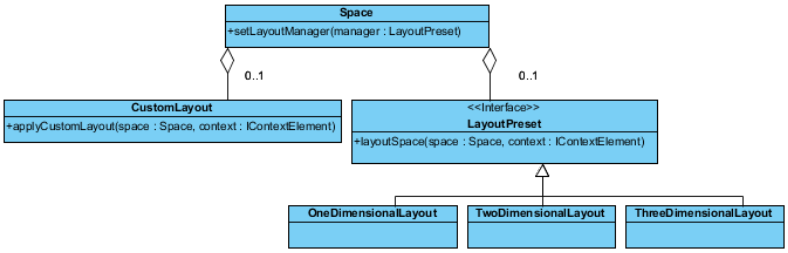


Abbildung 5.35.: Layouts

Der `OneDimensionalLayout`, siehe Abbildung 5.36, besteht aus einer Liste von `selectFieldByInstanceIdFromSpaceContext`-Selektoren. Die Reihenfolge der Liste stellt gleichzeitig die Reihenfolge dar, in der die in dem **Space** enthaltenen **Fields** angeordnet werden.

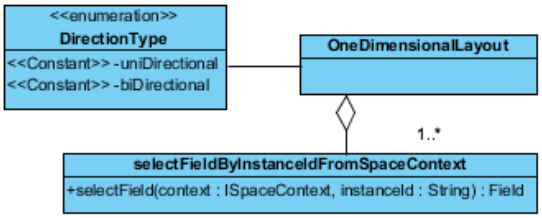


Abbildung 5.36.: OneDimensionalLayout

Über die `selectFieldByInstanceIdFromSpaceContext` ist es den hier vorgestellten `LayoutPresets` nur möglich, **Fields** miteinander zu verknüpfen, die sie selbst enthalten. Sämtliche Verknüpfungen von **Fields**, welche anderen **Spaces** zugeordnet sind, müssen über das `CustomLayout` definiert werden. Anhand

des `selectFieldByInstanceIdFromSpaceContext` erzeugt der jeweilige `LayoutPreset`, in diesem Fall das `OneDimensionalLayout`, für die selektierten `Field`-Objekte eine `FieldConnection`, welche in Abbildung 5.37 dargestellt ist. Eine `FieldConnection` besteht aus zwei `IFieldSelector` welche zusammen mit der `Direction` bestimmen, wie zwei **Fields** miteinander verknüpft sind. Das `LayoutPreset` nutzt dabei die `FieldFactory` des `ObjectManager`-Objekts.

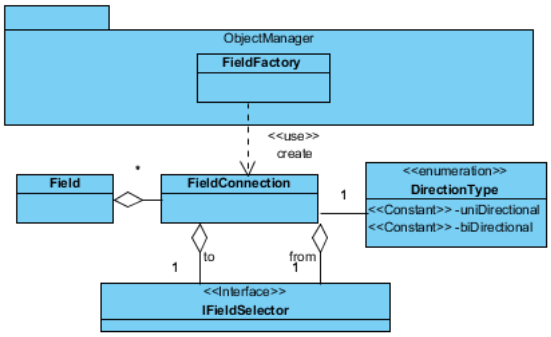


Abbildung 5.37.: FieldConnection

Abbildung 5.38 zeigt den `TwoDimensionalLayout`. Es besteht aus geschachtelten `Column`- und `Row`-Objekten, welche wiederum eine Liste von `selectFieldByInstanceIdFromSpaceContext` enthalten.

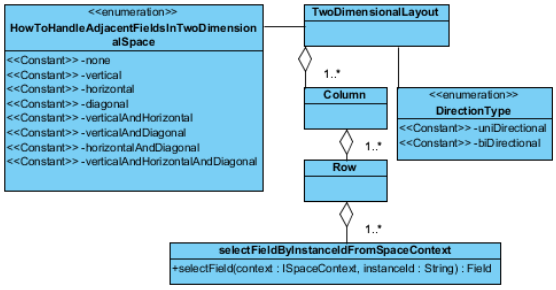


Abbildung 5.38.: TwoDimensionalLayout

Das `ThreeDimensionalLayout` fügt den `Column`-Objekten des `TwoDimensionalLayout` noch ein `Layer`-Objekt hinzu, welches die vertikale Achse repräsentiert. Das `HowToHandleAdjacentFieldsInTwoDimensionalSpace` wird eingesetzt, um jeweils auf horizontaler und vertikaler Achse bestimmen zu können, welche **Fields** miteinander verknüpft werden sollen. Abbildung 5.39 zeigt die Struktur des Layouts.

Das `CustomLayout`, zu sehen in Abbildung 5.40, kann zusätzlich zu einem `LayoutPreset` genutzt werden. Es besteht aus zwei unterschiedlichen Listen

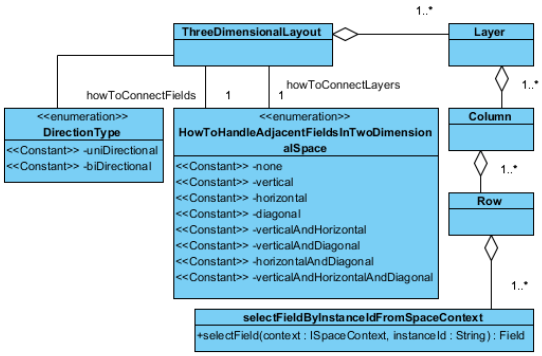


Abbildung 5.39.: ThreeDimensionalLayout

von `FieldConnection`: `Allow` und `Restrict`. Da an dieser Stellte direkt eine `FieldConnection` angegeben werden kann, welche aus zwei beliebigen `IFieldSelector` besteht, können über das `CustomLayout` sämtliche im Spiel befindlichen **Fields** miteinander verknüpft werden [26].

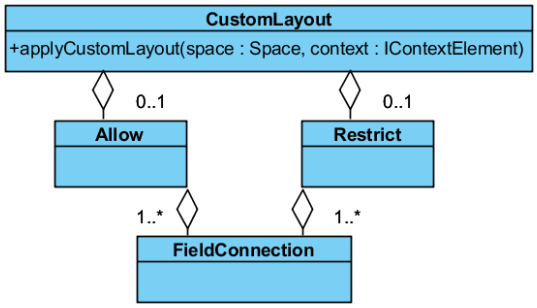


Abbildung 5.40.: CustomLayout

Das Anordnen der **Fields** eines `Spaces` erfolgt dabei in folgender Reihenfolge:

1. Falls `LayoutPreset` vorhanden, werden die **Fields** entsprechend miteinander verknüpft.
2. Falls `CustomLayout` vorhanden, werden die **Fields** entsprechend der `Allow`-Liste verknüpft, falls sie es nicht bereits sind.
3. Falls `CustomLayout` vorhanden, werden die **Fields** den Verknüpfungen entsprechend der `Restrict`-Liste entfernt, falls sie vorhanden sind.

[26] Die Anforderung FW-10 können in Zusammenhang mit dem ObjectManager als erfüllt angesehen werden.

Zufallsgenerator

Abbildung 5.41 zeigt, wie unterschiedliche Wahrscheinlichkeiten definiert werden können. Ein `SampleSpace` besteht aus unterschiedlichen `Chance`-Objekten. Für jede `Chance` kann über das `numberOfChances` festgelegt werden, wie hoch die Wahrscheinlichkeit ist. Die Wahrscheinlichkeiten aller `Chance`-Objekte ergibt die `totalNumberOfChances` des `SampleSpace`. Entsprechend ergibt sich aus der `totalNumberOfChances` geteilt durch die `numberOfChances` die Wahrscheinlichkeit, dass die jeweilige `Chance` ausgewählt wird. Das Element, welches den Wert der `Chance` darstellt, wird im Falle von „primitiven“ Variablen in Form von Attributes über den `IAttributeCreatorAndSelector` oder in Form von anderen Spielelementen, wie z.B. bei einem **Object** über den entsprechenden Selektor miteinander verknüpft. Der `IAttributeCreatorAndSelector` erlaubt es entweder, ein eigenes `Attribute`-Objekt zu erzeugen, welches nur als Behälter des angegebenen Wertes dient, oder anhand eines **Selektors** auf ein vorhandenes **Attribute** zu verweisen [27].

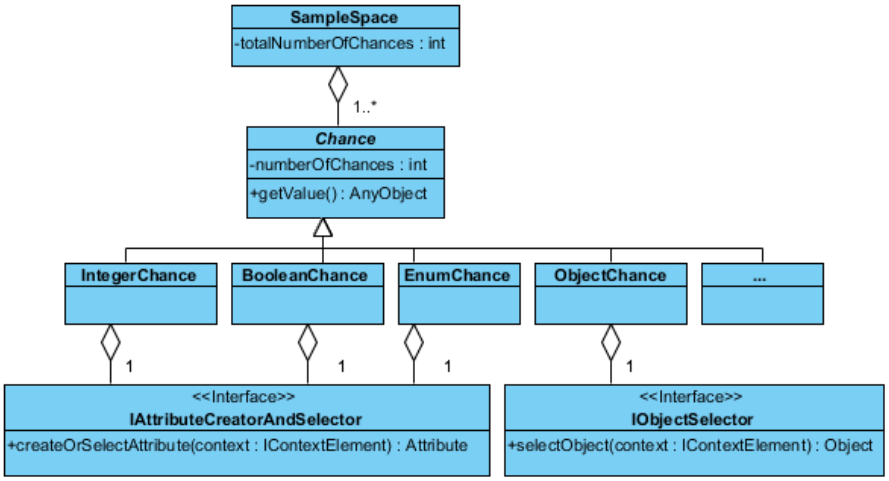


Abbildung 5.41.: SampleSpace

[27] Die Anforderung FW-11 kann in Zusammenhang mit dem RandomManager als erfüllt angesehen werden.

5.5.5. EventDistributor

Der `EventDistributor` erfüllt zwei Funktionen. Zunächst fungiert er als eine MessageQueue und verteilt die Events nach dem Publish-Subscribe-Muster. Die Events selbst werden nach dem FIFO-Verfahren abgearbeitet. Als Publisher und Subscriber treten in den meisten Fällen die unterschiedlichen Komponenten des `Controllers` auf. Die zweite Funktion liegt in der Teilung des `EventDistributors` in zwei unterschiedliche Komponenten: In den `Receiver` und den `Sender`, welche beide auf einem eigenen Thread und somit parallel ausgeführt werden können. Abbildung 5.42 zeigt den Aufbau de `EventDistributors`. Über den `Receiver` werden neue Events registiert. Der `Sender` arbeitet die `Events` nacheinander ab.

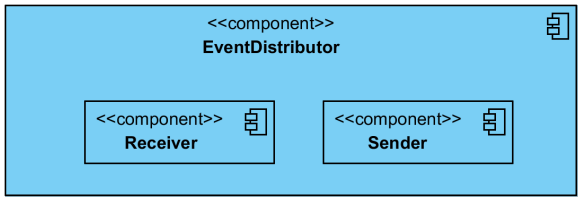


Abbildung 5.42.: EventDistributor

5.5.6. CoreController

Übersicht

Der `CoreController` besteht aus den in Abbildung 5.43 aufgeführten Komponenten. Jede Komponente ist für die Abwicklung und Steuerung eines dedizierten Aufgabenbereiches zuständig. An dieser Stelle sollen die einzelnen Komponenten kurz vorgestellt werden, um einen Überblick der Funktionen des `CoreControllers` zu vermitteln und in den folgenden Abschnitten näher beschrieben werden.

ObjectMaintenance: Ist für die Erstellung, Suche und Aufbereitung von Spiel-Elementen zuständig. Wird wiederum unterteilt in:

- **ElementCreator:** Ist für die Erstellung von Spiel-Objekten aus IGold-Defintion zuständig.
- **ObjectManager:** Ist für das Hinzufügen, Löschen und Finden von Spiel-Objekten des `CoreModels` zuständig.
- **ChangeManager:** Zeichnet Veränderungen an Spiel-Objekten auf.
- **ContainerManager:** Wandelt Spiel-Objekte in Container-Objekte und umgekehrt um.

PhaseManager: Regel im Spielfluss den Übergang von **Phases**.

PhaseStateManager: Steuert innerhalb einer **Phase** die Übergänge der **PhaseStates**.

ActionManager: Kontrolliert den Ablauf der **Actions** während eines **PhaseStates**.

OperationManager: Ist für die Abwicklung von **InternalOperations** und **InteractiveOperations** zuständig.

ImplicationManager: Kontrolliert die **Implications** und aktiviert bzw. deaktiviert diese je nach Spielzustand.

TimerManager: Steuert die Ausführung von **Timers**.

VisibilityManager: Verwaltet und aktualisiert die Sichtbarkeit der Spiel-Objekte.

RandomManager: Dient als Zufallsgenerator.

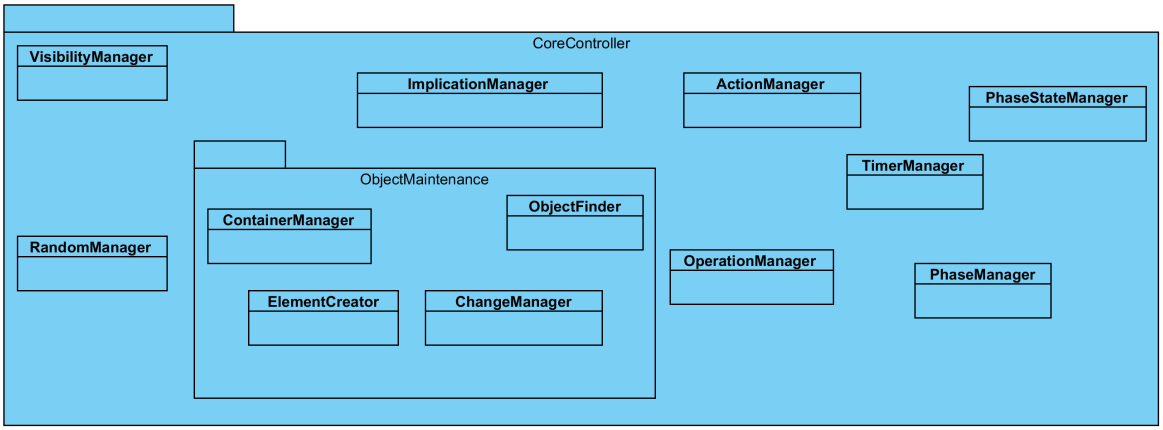


Abbildung 5.43.: CoreController

ObjectMaintenance: ElementCreator

Der `ElementCreator` ist für die Erzeugung von Objekt-Instanzen aus IGold-Definitionen verantwortlich. Dafür werden die in Abschnitt 5.5.4 vorgestellten und durch den Generator erstellten `ElementTemplates` von den entsprechenden Factories genutzt. Abbildung 5.44 zeigt sämtlich vorhandene Factories, welche bei der Erzeugung der Objektinstanzen genutzt werden. Anzumerken ist, dass die `Phase`-, `PhaseState`- und `ActionState`-Factory jeweils auch die zugehörigen `Transition`-Instanzen erstellt. Da eine `Transition` nur aus einer Menge von Regeln besteht, ist an dieser Stelle eine separate Factory nicht notwendig.

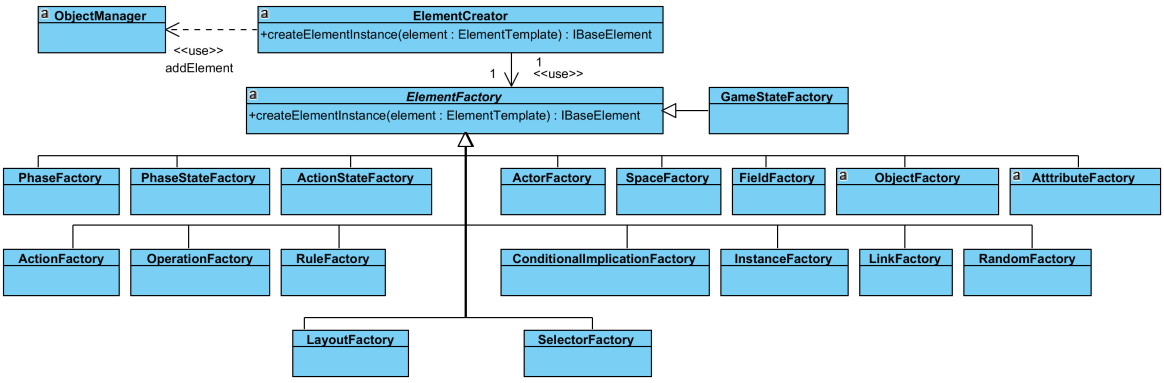


Abbildung 5.44.: ElementCreator

Die Erzeugung der permanent im Spiel vorhandenen Objekte wird dabei durch die `GameStateFactory` eingeleitet, welche das Wurzel-Element sämtlicher Instanzen von Spiel-Objekten darstellt. Der **GameState** wird durch das zugehörige `ElementTemplate`

generiert. Nachdem ein Element und sämtliche seiner Unter-Elemente erzeugt worden ist, fügt der `ElementCreator` die Instanz anhand des `ObjectManager` dem `CoreModel` hinzu. Da die GameState-Instanz sämtliche andere dauerhafte Spiel-Objekte einschließt, werden auf diese Art sämtliche Instanzen erzeugt, welche die gesamte Spielzeit über existieren. Instanzen von Spiel-Objekten, welche nur zwischenzeitlich eingesetzt werden, werden Lazy erstellt. Hierzu gehören **Phases**, **PhaseStates**, **ActionStates** und **Operations**. Beim Erzeugen einer Instanz wird diese nicht nur dem umschließenden Objekt zugeordnet, sondern auch gleichzeitig einer globalen Liste hinzugefügt. Für jeden Element-Typ existiert dabei eine globale Liste innerhalb des `CoreModels`, wie in Abbildung 5.10 auf Seite 136 dargestellt ist. Diese globalen Objekt-Listen werden von dem `ObjectManager` genutzt. Der Kontext eines Spiel-Objektes kann nicht gesetzt werden, da dieser erst bei der Verwendung ermittelt wird.

ObjectMaintenance: ObjectManager

Über den `ObjectManager` können Objekt-Instanzen dem `CoreModel` hinzugefügt oder gelöscht werden. Zusätzlich ermöglicht er den Zugriff auf sämtliche Spiel-Objekte. Abbildung 5.45 zeigt den Aufbau des `ObjectManager`. Zum einen nutzt er die Instanz-Listen des `CoreModels` und zum anderen pflegt er eigene, nach einem bestimmten Kriterium sortierte Liste von Instanz-Objekten.

Nach diesem Aufbau kann der Zugriff auf ein Objekt auf unterschiedliche Arten erfolgen:

1. Über die interne Id des Objektes (Nicht zu verwechseln mit der Instanz-Id!).
2. Über die Instanz-Id des Objektes.
3. Über die Typ-Id des Objektes.
4. Über die Link-Id des Objektes.
5. Über die LinkList-Id der Objekte.
6. Über ein gegebenes Set aus `Rules`, welches entweder eine Menge von Objekten oder genau ein Objekt als Rückgabewert erwartet.

Da bei allen Abfragen, bis auf die Suche über die interne Id, potenziell mehrere Instanzen gefunden werden können, existieren die Methoden in zwei Versionen: Einmal mit einer Liste und ein andermal mit nur einem Element als Rückgabewert. Wenn nur ein Element zurückgegeben werden soll, aber mehrere Elemente gefunden werden, erzeugt dies einen Fehler. Es liegt in der Verantwortung des Spiele-Entwickler, die richtigen Selektoren auszuwählen.

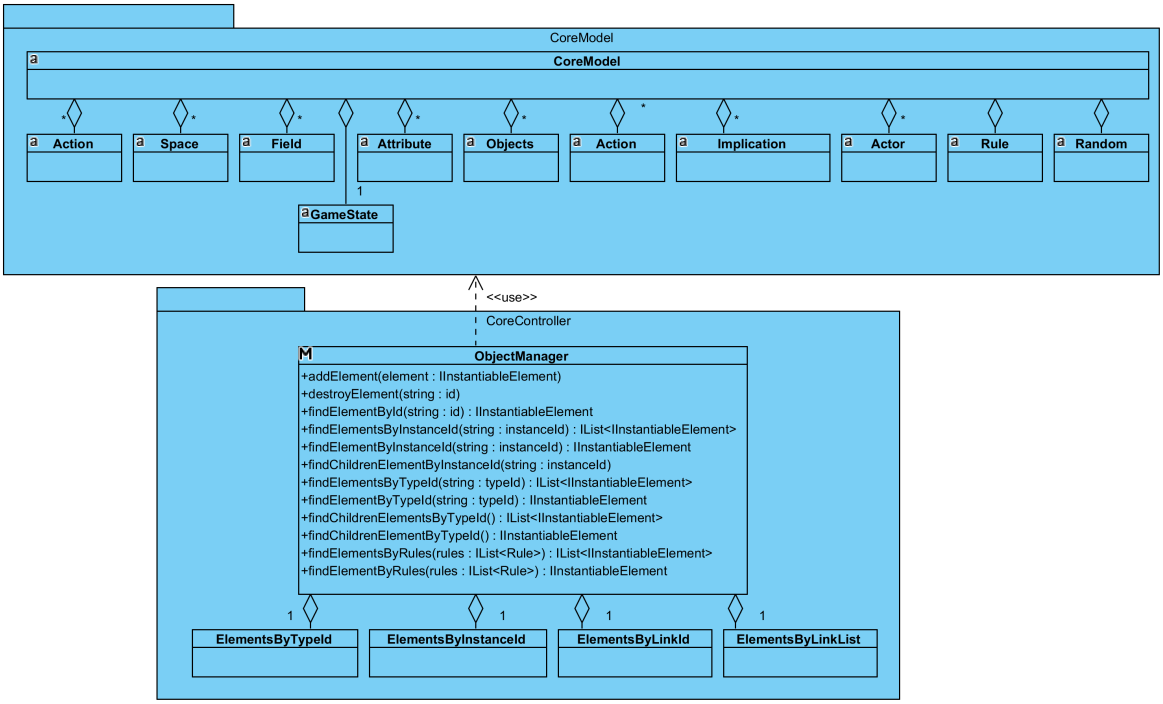


Abbildung 5.45.: ObjectFinder

Zusätzlich kann bei einigen Varianten ein Eltern-Element angegeben werden, in welchem gesucht werden soll. Auf diese Art ist die Suche nach einer Instanz-, Link- und LinkList-Id immer eindeutig.

ObjectMaintenance: ChangeManager

Abbildung 5.46 zeigt, dass der `ChangeManager` die Veränderungen an Objekt-Instanzen aufzeichnet und für jeden teilnehmenden Spieler zwei Listen von `ChangeVisibleToActor` mit jenen Änderungen führt, die für ihn sichtbar sind. In der ersten Liste sind sämtliche `ElementChange`-Objekte vorhanden, die noch nicht an den Spieler geschickt worden sind. In der zweiten Liste sind sämtliche `ElementChanges` enthalten, die bereits an den Spieler geschickt worden sind, für die jedoch noch keine Bestätigung eingegangen ist, dass der Spieler sie erhalten hat. Der `ChangeManager` empfängt das durch den `EventDistributor` gesendete Event, dass ein Spiel-Objekt geändert wurde. Anschließend wird diese Änderung gespeichert und über den `VisibilityManager` für jeden Actor geprüft, ob die geänderte Instanz für ihn sichtbar ist oder nicht. Wenn die `NetworkView` dem Spieler ein Update des Spielzustandes schickt, bezieht sie dieses durch den `ChangeManager` und der `ChangeManager` ver-

schiebt die `ElementChanges` in die andere Liste. Eine nähere Beschreibung dieses Vorgangs findet sich in Abschnitt 5.5.7 auf Seite 173.

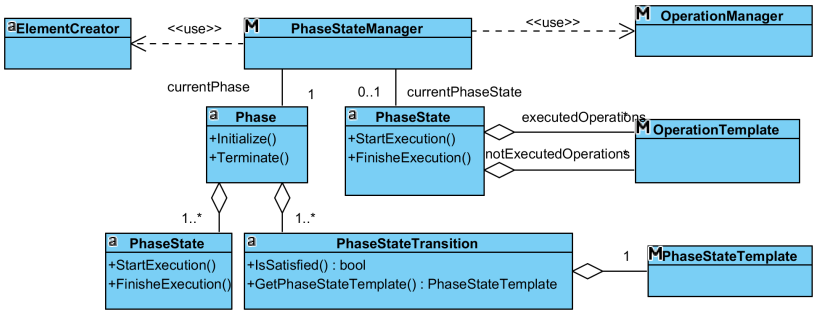


Abbildung 5.46.: ChangeManager

Die Struktur eines `ElementChangedEvents` ist in Abbildung 5.47 zu sehen und zeigt zugleich die `ElementChange`-Klasse.

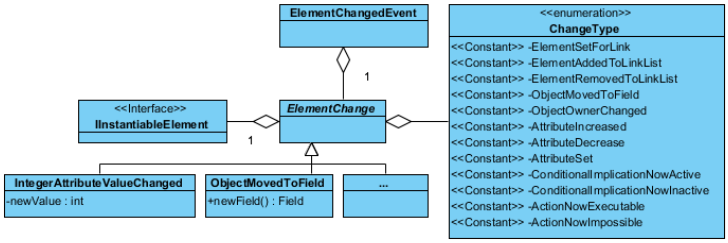


Abbildung 5.47.: ElementChangedEvent

ObjectMaintenance: ContainerManager

Der `ContainerManager` erzeugt die sogenannten Container-Objekte. Das Konzept der Sichtbarkeit wurde bereits in Abschnitt 5.5.4 ab Seite 149 erläutert. Ein Container-Objekt repräsentiert den aktuellen Zustand eines Spiel-Objektes, es verfügt selbst über keine eigene Funktionalität. Abbildung 5.48 zeigt, mit welchen anderen Komponenten der `ContainerManager` zusammenarbeitet. Die `NetworkView` kann sich durch das `IVisibleElement`-Interface einen Container des Objektes erstellen lassen. Anhand eines `Containers` kann sie wiederum über den `ContainerManager` die Instanz des Spiel-Objektes beziehen, für das der `Container` erzeugt wurde. Der `ContainerManager` nutzt für diesen Vorgang den `ObjectManager`.

Der `ContainerManager` dient dabei als Adapter zwischen der `NetworkView` und dem internem System. Wenn einem Spieler der Spielzustand gesendet werden soll, erhält er

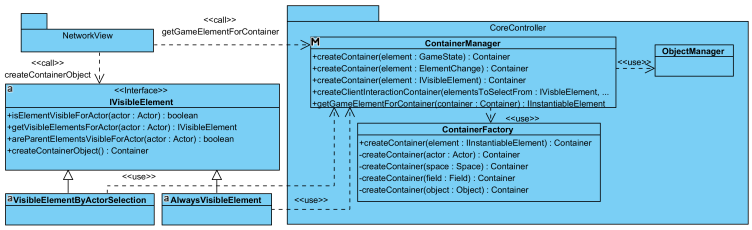


Abbildung 5.48.: ContainerManager

sämtliche für ihn sichtbare Container. Wenn ein Spieler wiederum auf eine Anfrage antwortet, indem er z.B. eine Aktion auswählt, die er ausführen möchte, bezieht die `NetworkView` das eigentliche `Action`-Objekt durch den `ContainerManager` und leitet dies entsprechend weiter.

PhaseManager

Der `PhaseManager` ist für die Zustandsübergänge zwischen den **Phases** verantwortlich. Die Struktur der für diesen Prozess relevanten Klassen wurde bereits in Abschnitt 5.5.4 ab Seite 140 beschrieben. Abbildung 5.49 zeigt, dass der `PhaseManager` sämtliche `PhaseTransitions` und jeweils eine oder keine `Phase` enthalten kann. Wenn eine `Phase` erzeugt werden soll, prüft der `PhaseManager` alle `PhaseTransitions`. Es muss genau eine `PhaseTransition` den aktuellen Spielzustand erfüllen, ansonsten gilt dies als fehlerhafter Zustand und das Spiel wird abgebrochen. Von dieser `PhaseTransition` kann anhand des enthaltenen `PhaseTemplates` über den `ElementCreator` die neue `Phase` erstellt und anschließend durchgeführt werden.

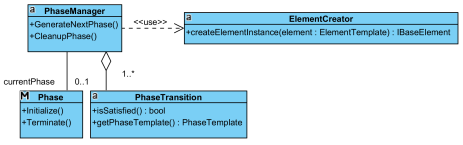


Abbildung 5.49.: PhaseManager

Die Funktionsweise des `PhaseManager` soll nun anhand eines konkreten Ablaufes, siehe Abbildung 5.50, vorgestellt werden.

Das Sequenzdiagramm zeigt den Zustand, nachdem die `GameInstance` den Start-Befehl gegeben hat und dieser über den `EventDistributor` an den `PhaseManager` weitergegeben wird. Dieser leitet nun das Auswählen der nächsten `PhaseTransition` ein, indem der `PrePhaseTransitionEvent` geschickt wird. Nachdem das

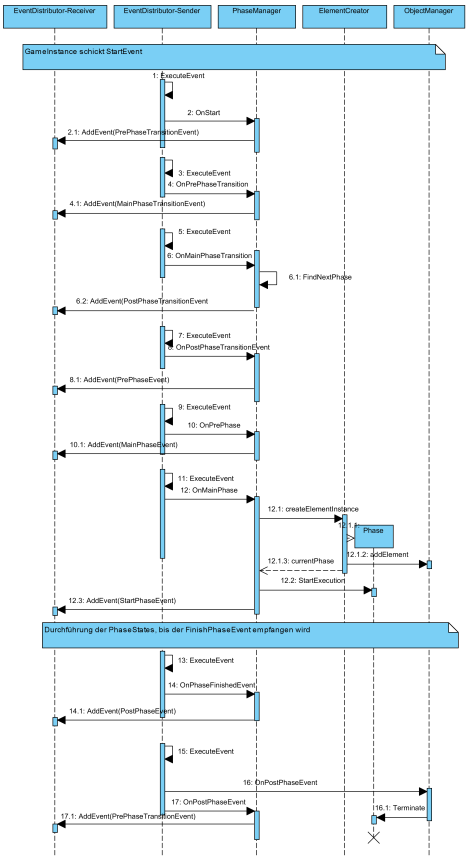


Abbildung 5.50.: Ablauf einer Phase

`MainPhaseTransitionEvent` empfangen wird, wird die `PhaseTransition` ausgewählt, welche den aktuellen Spielzustand erfüllt. Das enthaltene `PhaseTemplate` wird nach Erhalt des `MainPhaseEvent` dazu genutzt, um die gewünschte `Phase`-Instanz zu erstellen und zu starten. Anschließend wird das `StartPhaseEvent` gesendet. Dies veranlasst den `PhaseStateManager` mit der Durchführung der `PhaseStates`, die zu der `Phase` gehören, zu beginnen. Nachdem der `PhaseManager` das `PhaseFinishedEvent` erhält, welches durch den `PhaseStateManager` geschickt wird, wird das `PostPhaseEvent` geschickt. Als Reaktion auf dieses Event erzeugt der `PhaseManager` ein `PrePhaseEvent` und leitet somit einen Phasenübergang ein. Der `ObjectManager` zerstört gleichzeitig die nicht mehr notwendige `Phase`-Instanz.

PhaseStateManager

Die Abbildung 5.51 zeigt die Struktur des `PhaseStateManagers`.

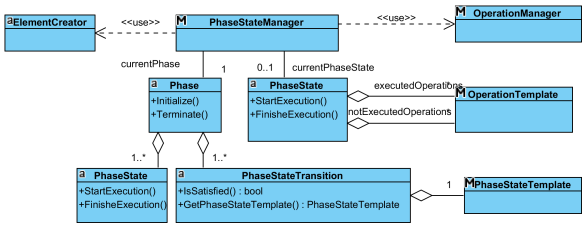


Abbildung 5.51.: PhaseStateManager

Er nutzt die in der `Phase` enthaltenen `PhaseStateTransitions`, um eine `PhaseState` zu erzeugen. Dieser `PhaseState` besteht aus einer Liste von abzuarbeitenden `OperationTemplates`, welche durch den `OperationManager` durchgeführt werden können.

Nach Erhalt des `MainPhaseEvent` ermittelt der `PhaseStateManager` den nächsten `PhaseState`, indem er über `PrePhaseStateTransition`-, `MainPhaseStateTransition`- und `PostPhaseStateTransition`-Schritte die `PhaseTransition` auswählt, welche den aktuellen Spielzustand erfüllt. Nach Erhalt des `MainPhaseStateTransitionEvent` wird die zu dem `PhaseTemplate` zugehörige `Phase` erzeugt. Nach Erhalt des `PostPhaseStateTransitionEvents` sendet der `PhaseStateManager` das `PrePhaseStateEvent`. Nach Empfang dieses Events wird die eigentliche Ausführung des `PhaseState` durchgeführt.

Das Sequenzdiagramm 5.52 stellt den Ablauf ab diesem Zeitpunkt von `PrePhaseStateEvent` über `MainPhaseStateEvent` bis hin zum `PostPhaseStateEvent` dar. Nachdem das `MainPhaseStateEvent` empfangen wird, wird zunächst geprüft, ob der `PhaseState` eine `Operation` zur Ausführung besitzt, und da dies der Fall ist, wird das `PreOperationEvent` ausgelöst. Anschließend wird die `Operation` ausgeführt und der `PhaseStateManager` wartet auf das `PostImplicationReevaluationEvent`. Nachdem er dieses erhalten hat, prüft er, ob eine weitere `Operation` durchgeführt werden muss, und führt diese durch. Nachdem der `PhaseStateManager` ein weiteres Mal das `PostImplicationReevaluationEvent` erhält, muss keine `Operation` mehr abgearbeitet werden und er sendet das `PostPhaseStateEvent`.

ActionManager

Die Struktur des `ActionManager` wird in Abbildung 5.53 gezeigt. Er setzt voraus, dass eine `Phase` und ein `PhaseState` vorhanden sind. Pro Spieler kann der `ActionManager` eine `Action` ausführen. Handelt es sich um eine `ComplexAction`, werden die Zustandsübergänge ebenso wie bei dem `PhaseManager` und dem `PhaseStateManager` über

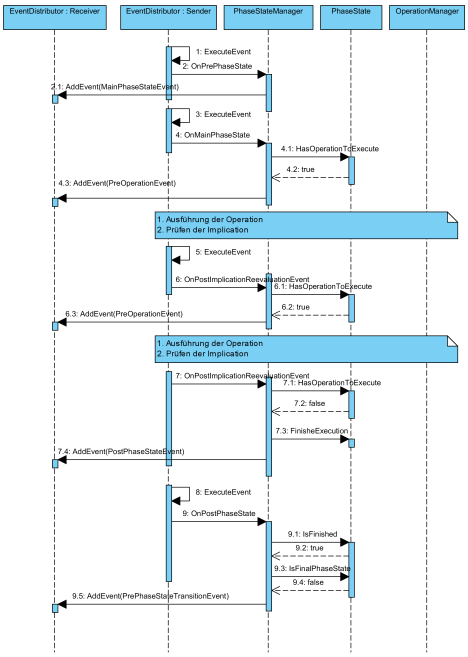


Abbildung 5.52.: Ablauf PhaseStates während des MainPhaseState-Abschnittes

`ActionStateTransitions` modelliert, welche ein `ActionTemplate` für die spätere Instanziierung durch den den `ElementCreator` enthalten. Ein `ActionState` sowie die `SimpleAction` bestehen aus einer Liste von `OperationTemplates`, welche durch den `OperationManager` durchgeführt werden können.

Der `ActionManager` ermöglicht die gleichzeitige Ausführung von ein oder mehr `Actions`, wobei ein **Actor** nur eine **Action** zurzeit ausführen darf. Gleichzeitig bedeutet in diesem Kontext, dass, während die `ComplexAction` eines **Actors** noch nicht vollständig durchlaufen wurde, ein anderer **Actor** seine eigene `ComplexAction` weiter durchführen oder eine neue **Action** ausführen kann, falls er derzeit keine laufende **Action** besitzt. Die Ausführung sämtlicher `Operations` eines `ActionStates` oder einer `SimpleAction` ist jedoch atomar und muss beendet werden, bevor andere `Operations` durchgeführt werden können.

Die Ausführung einer `Action` wird ebenso in Pre-, Main- und Post-Abschnitte unterteilt. Bei einer `ComplexAction` wird jeder `ActionState` ebenfalls in diese drei Abschnitte eingeteilt. Die Abbildung 5.54 soll anhand eines Sequenzdiagramms die Ausführung einer `SimpleAction` beschreiben. Dabei wurde im Vorfeld einem **Actor** eine Menge an unterschiedlichen **Actions** gesendet, aus denen er auswählen kann. Der **Actor** sendet die ausgewählte **Action** an die `NetworkView`, welche wiederum einen `PlayerRequestsActionExecutionEvent` erzeugt. Das Diagramm zeigt, wie der `ActionManager` diese Event erhält und über den `ObjectManager` die zugehörige

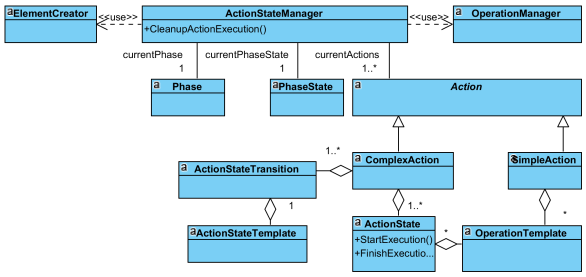


Abbildung 5.53.: ActionManager

`SimpleAction` bezieht. Nachdem geprüft wurde, ob die `SimpleAction` zu diesem Zeitpunkt auch durchführbar ist, wird das `PreActionEvent` gesendet. Anschließend wird das `MainActionEvent` ausgelöst, wieder empfangen und die `SimpleAction` gestartet. Bei einer `SimpleAction` besteht die Ausführung in der Abarbeitung der unterschiedlichen `OperationTemplates`. Die Durchführung einer `Operation` anhand des Templates wurde im folgenden Abschnitt anhand des Sequenzdiagramms 5.56 erläutert und soll im Detail aufgezeigt werden. Nachdem die Durchführung der `Operation` beendet ist und dies durch das `PostImplicationReevaluationEvent` signalisiert wurde, wird die nächste `Operation` durchgeführt. Falls es keine auszuführenden `Operations` mehr gibt, wird die `SimpleAction` als vollständig durchgeführt erklärt und das `PostActionEvent` gesendet.

Die Ausführung einer `ComplexAction` folgt dem hier vorgestellten Ablauf der `SimpleAction`. Dabei werden nach Empfangen des `MainActionEvent` allerdings die `ActionStateTransitions` geprüft und nach dem bereits bekannten Muster durch die Folge von `PreActionStateTransitions`, `MainActionStateTransitions`, `PostActionStateTransitions` durchgeführt. Die Verarbeitung erfolgt analog zu der einer `PhaseStateTransition` oder einer `PhaseTransition`.

Es muss genau eine `ActionStateTransition` durch den Spielzustand erfüllt werden. Anschließend instanziiert der `ElementCreator` über das enthaltene `ActionStateTemplate` den eigentlichen `ActionState`. Nachdem sämtliche `Operations` abgearbeitet worden sind, wird der `ActionState` terminiert und über die `ActionStateTransitions` der nächste Zustand ausgewählt, bis die `ComplexAction` durch das Erreichen eines finalen Zustands beendet wird.

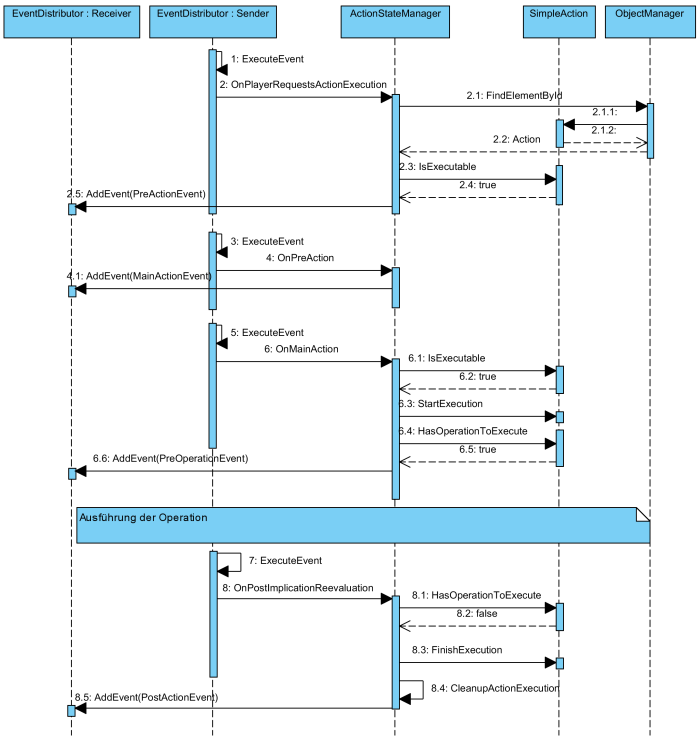


Abbildung 5.54.: Ablauf einer SimpleAction

OperationManager

Der `OperationManager` ist für die Erzeugung, Initialisierung und Ausführung von `Operations` und `InteractiveOperations` zuständig. Das Sequenzdiagramm in Abbildung 5.55 zeigt, wie dieser Prozess für eine `InternalOperation` abläuft.

Über das `PreOperationEvent` wird dem `OperationManager` signalisiert, dass er eine `Operation` ausführen soll. Der `OperationManager` bezieht über die Id des `OperationTemplate` die dazugehörige Instanz über den `ObjektManager`. Anschließend wird über den `ElementCreator` aus diesem Template eine Instanz der `Operation` erstellt und diese durch den `ObjektManager` registriert. Danach wird die `InternalOperation` ausgeführt und danach durch den `OperationExecutionFinishedEvent` signalisiert, dass sie durchgeführt wurde. Der `ObjectManager` reagiert auf dieses Event und zerstört die `InternalOperation` wieder.

Abbildung 5.56 zeigt die Unterschiede des Ablaufes einer `InteractiveOperation`.

Sie wird ebenso erzeugt und zerstört wie die `InternalOperation`. Nach der Erzeugung wird die Durchführung der `InteractiveOperation` gestar-

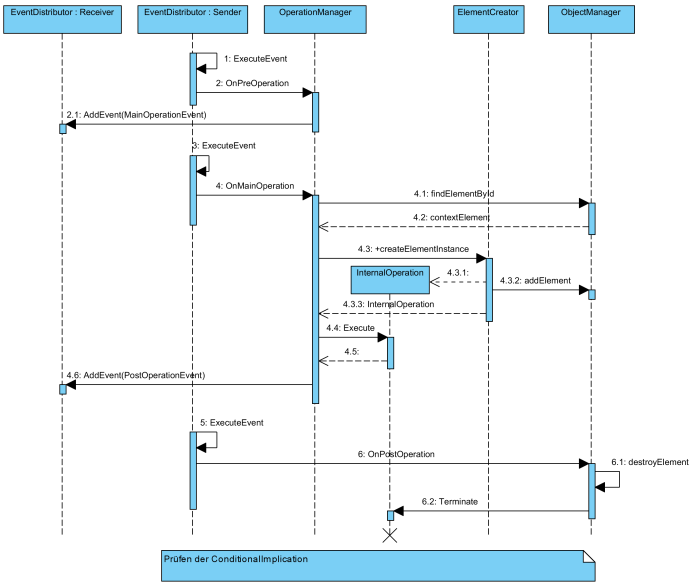


Abbildung 5.55.: Ablauf einer InternalOperation

tet. Daraufhin werden über `SendServerRequestEvent` die **Actors** aufgefordert, eine Entscheidung zu treffen. Die Antwort erreicht den `OperationManager` in Form eines `PlayerResponseReceivedEvent`. Diese Antwort wird an die `InteractiveOperation` weitergeleitet, welche die Antwort verarbeitet. Nach der Verarbeitung der Antwort der `InteractiveOperation` wird geprüft, ob die Durchführung abgeschlossen ist. Falls dies der Fall ist, wird über das `OperationExecutionFinishedEvent` die fertige Ausführung signalisiert und daraufhin die `InteractiveOperation` zerstört.

ImplicationManager

Der `ImplicationManager` stellt sicher, dass sämtliche im Spiel befindlichen `Implications` nach jeder Veränderung des Spielzustandes, also nach jeder durchgeführten `Operation`, aktualisiert werden. Abbildung 5.57 zeigt den Ablauf dieses Prozesses.

Nach Erhalt des `PostOperationEvents` werden über `PreImplicationReevaluation`, `MainImplicationReevaluation` und `PostImplicationReevaluation` alle `Implications` geprüft und bei Bedarf aktiviert, bzw. deaktiviert. Das `PostImplicationReevaluation` veranlasst den `PhaseStateManager` oder den `ActionManager`, die Arbeit wieder aufzunehmen.

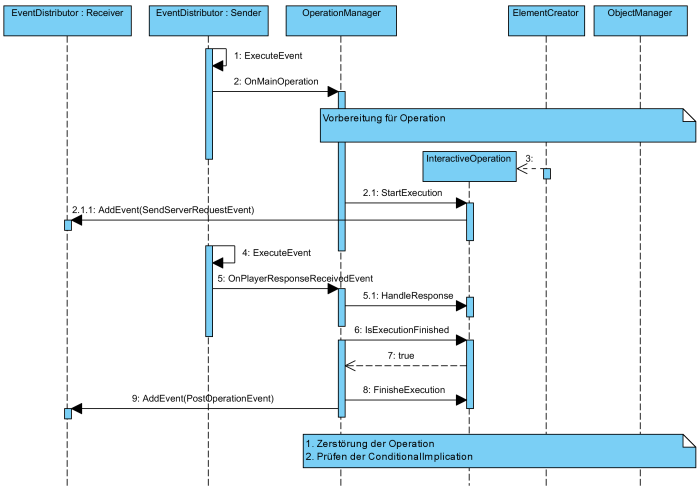


Abbildung 5.56.: Ablauf einer InteractiveOperation

TimerManager

Der `TimerManager`, siehe Abbildung 5.58, sorgt dafür, dass ein `Timer` nach einer definierten Zeitspanne durchgeführt wird.

Das Sequenzdiagramm in Abbildung 5.59 zeigt den Lebenszyklus eines `Timers`. Der `TimerManager` besitzt einen eigenen Thread, damit das Auslösen des `Timers` unabhängig von anderen Ereignissen ist. Nachdem ein `Timer` ausgelöst wurde, wird dieser innerhalb eines Pre-, Main- und Post-Abschnittes abgearbeitet. Die Events hierzu werden über den `EventDistributor` gesendet. Das `PreTimeoutEvent` signalisiert den `PhaseManager`, `PhaseStateManager` und `ActionManager`, dass die Ausführung eines `Timers` bevorsteht. Nachdem der `Timer` ausgeführt wurde, wird er nach Erhalt des `PostTimeoutEvent` zerstört.

VisibilityManager

Der `VisibilityManager` verwaltet die Sichtbarkeit der Spiel-Elemente. Er wird von dem `ChangeManager` genutzt, um bei einer Veränderung eines Objektes zu prüfen, für welche **Actors** diese Änderung sichtbar ist. Zusätzlich kann der `VisibilityManager` den Spielzustand aus der Sicht eines **Actors** liefern, welcher alle für ihn sichtbaren Objekte enthält. Dieser Spielzustand wird anhand des `ContainerManager` in Container umgewandelt und über die `NetworkView` verschickt.

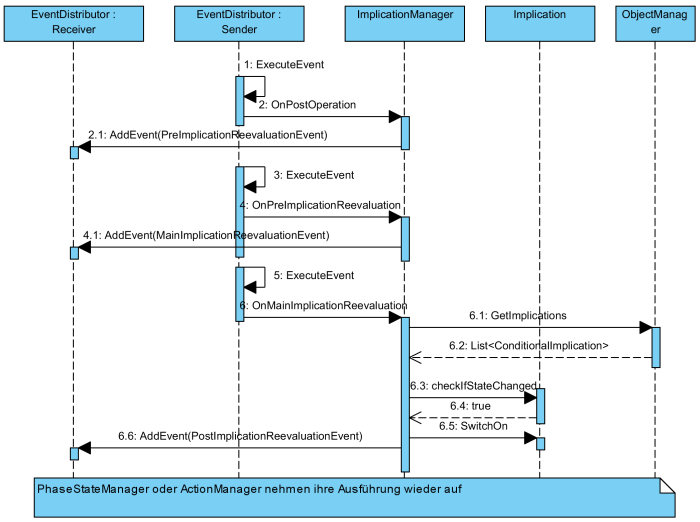


Abbildung 5.57.: ImplicationManager

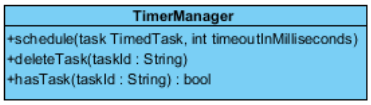


Abbildung 5.58.: TimerManager

RandomGenerator

Der `RandomGenerator` erlaubt das zufällige Auswählen von Elementen. Hierzu werden die in Abschnitt 5.5.4 ab Seite 156 vorgestellten `SampleSpaces` eingesetzt. Abbildung 5.60 zeigt die Klasse des `RandomGenerator`. Er ermöglicht es, ein zufälliges Element auf zwei unterschiedliche Art auszuwählen: Mit oder ohne Zurücklegen. Bei der zufälligen Auswahl von nur einem Element wirkt sich dies nicht auf das Ergebnis aus, daher stehen nur die drei in der Abbildung dargestellten Methoden zur Verfügung.

Der `RandomGenerator` kann innerhalb von `Operations` eingesetzt werden, um das Zufallselement innerhalb von Spielen abzubilden.

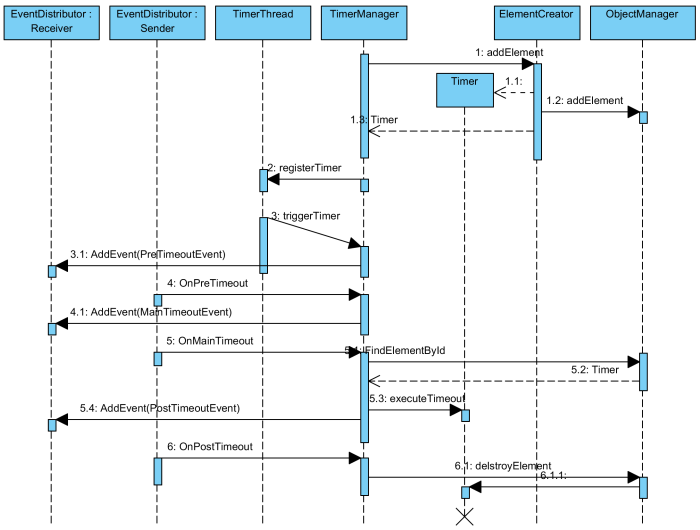


Abbildung 5.59.: Ablauf eines Timeouts

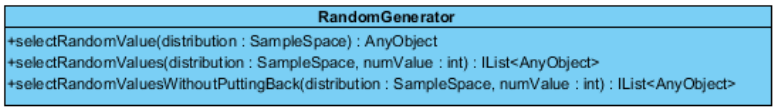


Abbildung 5.60.: RandomManager

5.5.7. NetworkView

Die NetworkView stellt die Schnittstelle zwischen der internen Spiele-Logik und der Runtime dar. Abbildung 5.61 zeigt die Komponenten, welche mit der NetworkView zusammenarbeiten. Die NetworkView implementiert die GameProxy-Schnittstelle und kann

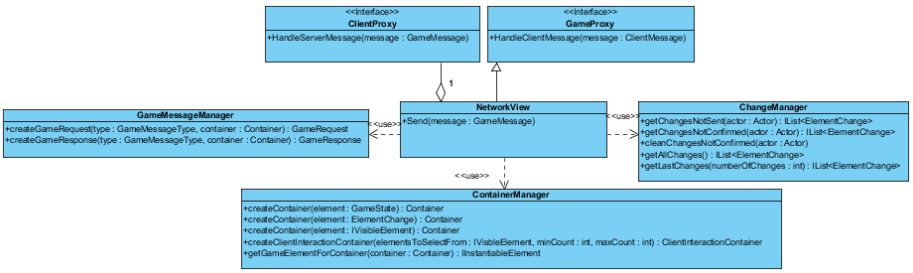


Abbildung 5.61.: NetworkView

ClientMessages verarbeiten, welche von der Runtime geliefert werden. Zusätzlich besitzt die NetworkView ein Objekt, welches die ClientProxy-Schnittstelle implementiert. Dieses Objekt ist eine Komponente der Runtime, welches für die GameMessages zuständig ist. Diese Konstruktion gestaltet die eigentliche Kommunikation zwischen Spiel-Mechanik und Laufzeitumgebung transparent [28].

Die NetworkView ist dafür verantwortlich, die Anfragen und Antworten seitens der Spiellogik in GameMessages zu verpacken. Abbildung 5.62 zeigt die Struktur dieser Nachrichten. Jede GameMessage besitzt eine eigene messageId, eine gameInstanceId und

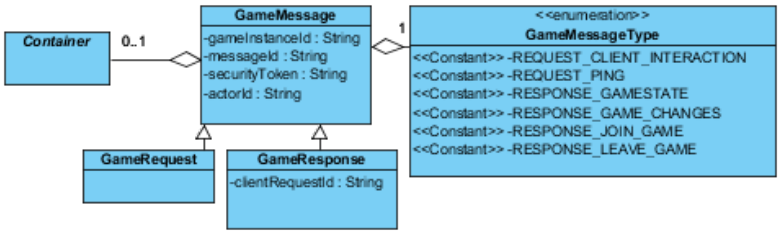


Abbildung 5.62.: GameMessage

eine actorId. Das securityToken stellt einen Hash über den restlichen Inhalt dar und dient zur Konsistenzprüfung der Nachricht. Von dem GameMessageType kann abgeleitet werden, welcher konkrete Container in der Nachricht vorhanden sind. Im Falle

[28] Anforderung FW-2 gilt als erfüllt.

des REQUEST_CLIENT_INTERACTION-Typs erhält der Client eine Menge von Elementen, aus denen er auswählen muss, und sämtliche Änderungen des Spielzustandes, welche seit der letzten Kommunikation durchgeführt worden sind. Diese Objekte werden durch den ChangeManager zur Verfügung gestellt und durch den ContainerManager in Container umgewandelt. Die Auswahlliste wird durch die Spiellogik vorgegeben. Dies kann entweder darauf zurückzuführen sein, dass am Ende eines PhaseStates ein Client auszuführende Actions besitzt oder eine InteractiveOperation die Interaktion verlangt.

Die Struktur der ClientMessage spiegelt die der GameMessage, siehe Abbildung 5.63. Über die ClientMessageTypes REQUEST_JOIN_GAME kann einem Spiel beigetre-

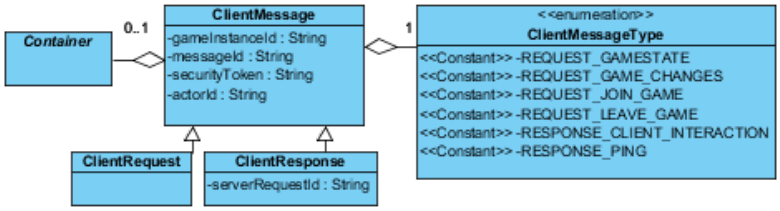


Abbildung 5.63.: ClientMessage

ten werden und über REQUEST_LEAVE_GAME entsprechend das Spiel wieder verlassen werden.

5.6. Fazit

5.6.1. Zusammenfassung

In diesem Kapitel wurde die Architektur jener Komponenten vorgestellt, die notwendig sind, um einem Spieleentwickler die Nutzung der Sprache lGold zu ermöglichen und die implementierten Spiele testen zu können. Dabei wurde zunächst das Design der Entwicklungsumgebung vorgestellt und anhand von Feasibility-Studien gezeigt, dass die Nutzung der präsentierten Language Workbench die notwendigen Anforderungen erfüllt. Sie ermöglicht die Implementierung von Ansichten zur Bearbeitung der lGold-Elemente, lässt sich in Bezug auf die Debugging-Funktionalität entsprechend erweitern und bietet Unterstützung für den Bereich der Code-Generierung. Dabei kommt kein eigener DSL-Processor zum Einsatz, da die bereits vorhandenen Tools eine vollständig eigene Implementierung unnötig machen.

Anschließend wurde die Test-Runtime vorgestellt, welche es ermöglicht, ein Spiel mit mehreren Spielern in einer einzigen Anwendung testen zu können. Bei dem Design der Runtime und der GameInstance, welche den Zustandsautomaten und die eigentliche Spiel-Logik darstellt, wurde eine enge Kopplung weitgehend vermieden. Auf diese Weise ist es möglich, die in der Test-Runtime notwendige GameInstance in eine andere Runtime ohne zusätzlichen Aufwand zu übernehmen.

Danach wurde das Framework vorgestellt, welches das Fundament der GameInstance darstellt, die in der Runtime instanziiert wird. Das Framework ermöglicht die Verarbeitung wie ein Zustandsautomat, dessen Verhalten durch den von der Entwicklungsumgebung generierten Code gesteuert wird. Detailliert Ausgeführt wurde das Design und das Zusammenspiel der unterschiedlichen Komponenten, welche die in Kapitel 4 vorgestellten Konzepte und Elemente implementieren. An dieser Stelle soll betont werden, dass für die für die Sprache besonders kritischen Aspekte wie Kontext, Selektoren und die flexibel ausgelegten Template-Elemente **Rules** und **Operation** ein nachvollziehbares und den Anforderung entsprechendes und flexibles Design ausgearbeitet wurde. Weiterhin wurde der Ablauf eines Spieles auf den unterschiedlichen Ebenen des Zustandsautomaten und den beteiligten Komponenten dargestellt.

Die Funktionsweise der unterschiedlichen Architektur-Konzepte des Frameworks wurde in einem funktionsfähigem Prototypen des Referenzbeispieles erforscht, und die daraus gewonnenen Erfahrungen sind in das derzeitige Design eingeflossen.

5.6.2. Bewertung

Das in diesem Kapitel vorgestellte Design stellt eine plausible Grundlage für eine vollständige Umsetzung der Sprache lGold dar. Die in dem Framework eingesetzte Architektur resultiert aus den Erfahrungen der Implementierung eines Prototypen. Dies deutet auf einen belastbaren und praktisch umsetzbaren Entwurf hin.

Die Flexibilität des Frameworks, welche sich in einer erhöhten Komplexität niederschlägt, könnte sich möglicherweise auf die Performance des Spieles auswirken. Dies lässt sich am einfachsten an den Template-Elemente wie einer **Rule** nachvollziehen. Die unterschiedlichen Parameter entsprechen jeweils einem eigenen Objekt. Auf diese Weise lässt sich zwar die Funktionalität dieser Parameter-Objekte immer wiederverwenden, allerdings wird eine Vielzahl von Objekten benötigt, um ein einziges Regel-Objekt zu erstellen. Da Gesellschaftsspiele in Bezug auf die Anforderungen hinsichtlich ihrer Reaktionsgeschwindigkeit genügsam sind, stellt dies jedoch kein fundamentales Problem dar.

6. Fazit

6.1. Zusammenfassung

Im Rahmen dieser Arbeit wurde die domänenspezifische Sprache lGold entwickelt. Die Sprach-Elemente wurden auf der Grundlage der in der Forschung vorhandenen Klassifikations- und Definitionsansätzen festgelegt und speziell auf den Bereich der Gesellschaftsspiele ausgerichtet. Anhand eines Referenzbeispieles wurde die Anwendung dieser grundlegenden Definition demonstriert. Danach wurde zunächst eine Vision der Sprache und sämtlicher unterstützenden Werkzeuge beschrieben und anschließend die notwendigen Aspekte in einem Minimal-Entwurf zusammengefasst. Basierend auf diesem Entwurf wurden die Anforderungen an die Struktur von lGold und an die zusammenhängende Systeme definiert.

Diese Anforderungen flossen entsprechend bei der Entwicklung der Sprachkonzepte und der einzelnen Elemente von lGold mit ein. Es wurde ein deklarativer Ansatz gewählt, bei dem die Definition in einer XML-Syntax beschrieben wird. Die Syntax wird dabei über ein XSD-Schema geprüft. Bei dem Ausführungsmodell handelt es sich um einen Zustandsautomaten. Der aus der Definition generierte Code bestimmt das Verhalten des Automaten.

Anschließend wurde eine Architektur, bestehend aus einer Entwicklungsumgebung, einem DSL- sowie einem Code-Generator und einem Framework, welches wiederum die Grundlage der Test-Runtime bildet, vorgestellt. Die Entwicklungsoberfläche wurde mithilfe einer Language Workbench erstellt, wobei anhand einer Feasibility-Study gezeigt wurde, wie eine grafische Repräsentation der Sprache aussehen könnte und wie diese in lGold-Syntax übersetzt wird. Im Rahmen der Architektur des Frameworks wurde zum einen die Funktionsweise des semantischen Modells erläutert. Es wurde in Form von Template-Klassen entworfen, welche die Verhaltensweise des Spiel-Zustandsautomaten steuern. Das Zusammenspiel der unterschiedlichen Elemente wurde detailliert vorgestellt und die Entscheidungen beim Entwurf des Designs entsprechend argumentiert. Eine Umsetzung der hier vorgestellten Sprache scheint somit durchführbar [1].

[1] Erfüllt somit Anforderung DSL-3.

6.2. Ausblick

In Bezug auf die nächsten Schritte wäre eine praktische Überprüfung der Sprache in Bezug auf ihren Domänen-Tauglichkeit sinnvoll. Der Entwurf könnte mit Domänen-Experten diskutiert und möglicherweise verfeinert werden. Eine grundsätzliche andere Ausrichtung scheint unwahrscheinlich, dennoch wäre es vorstellbar, dass weitere Konzepte dennoch in die Sprache einfließen sollten.

Nachdem die Erfahrungen aus der Zusammenarbeit mit den Experten in den Sprachentwurf mit eingeflossen ist kann mit der Umsetzung begonnen werden. Die Entwicklungsumgebung stellt dabei für die Überprüfung der Funktionsfähigkeit keine Voraussetzung dar und sollte in einem separaten Projekt ausgearbeitet werden. Diese sollte sinnvollerweise folgendermaßen durchgeführt werden:

1. Prototyp-Framework
2. Prototyp-Code-Generator
3. Prototyp-Test-Runtime

Der Code-Generator und das Erzeugen der Template-Klassen stellt zwar die Grundlage für die Konfiguration des Spieles dar, sollte aber erst umgesetzt werden nachdem das Framework einen ausführbaren Zustand erreicht hat. Bis dahin können die Templates noch manuell erstellt werden. Die Test-Runtime stellt die Umgebung und sollte entsprechend eingebunden werden, nachdem das Framework einen lauffähigen Zustand erreicht. Anschließend ergeben sich unterschiedliche Ansatzpunkte für folgende Produkte.

Um die Vorteile der domänenspezifischen Sprache messbar zu machen, könnte versucht werden der die Produktivität zu vergleichen. Ein direkter Vergleich, bei dem einmal ein Spiel „per Hand“ und ein anderes Mal mithilfe der Sprache realisiert wird erscheint einen nicht gerechtfertigten Aufwand zu bedeuten. Als Alternative könnte über eine Kooperation mit einem Spieleentwickler ein bereits fertig entwickeltes Spiel als Referenz dienen. Von diesem Spiel könnte die Benutzeroberfläche und sämtliche Graphiken genutzt werden und die Spiele-Logik entsprechend in lGold implementiert werden. Auf diese Weise werden Erfahrungen im Rahmen der Spieleentwicklung mit der DSL gesammelt und die beiden Herangehensweise können miteinander verglichen werden. Als Resultat lässt sich eine Aussage über die Produktivitätssteigerung durch das Nutzen der Sprache formulieren.

Weitere interessante Fragestellungen könnten lauten:

- Kann die hier vorgestellte Sprache auch für andere Bereiche, wie z.B. für den Bildungsbereich, adaptiert werden?

- Wie kann die hier vorgestellte Sprache für aktuelle Forschungsschwerpunkte wie Serious Games und Gamification eingesetzt werden?
- Wie wirkt sich die Weiterentwicklung einer bestehenden domänenspezifischen Sprache aus, wie können Komplikationen minimiert werden?

Literaturverzeichnis

[Apt 1993] APT, Krzysztof R.: Declarative Programming in Prolog. In: *Logic Programming - Proceedings of the 1993 International Symposium*, The MIT Press, 1993, S. 12–35

[Avedon und Sutton-Smith 1971] AVEDON, E.M. ; SUTTON-SMITH, B.: *The study of games*. J. Wiley, 1971. – URL http://books.google.de/books?id=eOeBAAAAMAAJ. – ISBN 9780471038399

[Backus] BACKUS, John W.: *The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference (1959)*

[Bjork und Holopainen 2004] BJORK, Staffan ; HOLOPAINEN, Jussi: *Patterns in Game Design (Game Development Series)*. Rockland, MA, USA : Charles River Media, Inc., 2004. – ISBN 1584503548

[Björk u. a. 2003] BJÖRK, Staffan ; LUNDGREN, Sus ; HOLOPAINEN, Jussi: Game Design Patterns. In: *Design* 54 (2003), Nr. 3, S. 180–193. – URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.10.4097&rep=rep1&type=pdf

[BoardGameGeek] BOARDGAMEGEEK: *Samurai-Bild*. – URL http://boardgamegeek.com/image/909798/samurai?size=large

[Brathwaite und Schreiber 2009] BRATHWAITE, B. ; SCHREIBER, I.: *Challenges for Game Designers*. Course Technology, 2009. – URL http://books.google.de/books?id=wzAJLwAACAAJ. – ISBN 9781584505808

[Brothers] BROTHERS, Parker: *Wikipedia: Monopoly*. – URL http://de.wikipedia.org/wiki/Monopoly

[Chomsky 1959] CHOMSKY, Noam: *On certain formal properties of grammars*. 1959

[Cook 2006] COOK, D: What are game mechanics? In: *Garden* (2006), S. 10–12. – URL http://lostgarden.com/2006/10/what-are-game-mechanics.html

[Csikszentmihalyi 2008] CSIKSZENTMIHALYI, M.: *Flow: The Psychology of Optimal Experience*. HarperCollins, 2008 (P. S. Series). – URL http://books.google.de/books?id=epmhVuaaoK0C. – ISBN 9780061339202

[Cunningham 2008] CUNNINGHAM, H. C.: A little language for surveys: constructing an internal DSL in Ruby. In: *Proceedings of the 46th Annual Southeast Regional Conference on XX*. New York, NY, USA : ACM, 2008 (ACM-SE 46), S. 282–287. – URL http://doi.acm.org/10.1145/1593105.1593181. – ISBN 978-1-60558-105-7

[edinburghwargames.com] EDINBURGHWARGAMES.COM: *Tabletop-Spielfeld*. – URL http://www.edinburghwargames.com/J23%20Images/DSCF3573.jpg

[Fowler 2010] FOWLER, M.: *Domain-Specific Languages*. Addison-Wesley, 2010 (The Addison-Wesley Signature Series). – URL http://books.google.de/books?id=ri1muolw_YwC. – ISBN 9780321712943

[Fowler 2012] FOWLER, M.J.: *Patterns of Enterprise Application Architecture*. Pearson Education, 2012 (Addison-Wesley Signature Series). – URL http://books.google.de/books?id=vqTfNFDzzdIC. – ISBN 9780133065213

[Fullerton u. a. 2004] FULLERTON, T. ; SWAIN, C. ; HOFFMAN, S.: *Game Design Workshop: Designing, Prototyping, and Playtesting Games*. CMP, 2004 (Com27 Series). – URL http://books.google.de/books?id=61LbUE2K3zoC. – ISBN 9781578202225

[Gamma 1995] GAMMA, E.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995 (Addison-Wesley Professional Computing Series). – URL http://books.google.de/books?id=6oHuKQe3TjQC. – ISBN 9780201633610

[Grune und Jacobs 2008] GRUNE, D. ; JACOBS, C.J.H.: *Parsing Techniques*. Springer, 2008 (Monographs in Computer Science). – URL http://books.google.de/books?id=05xA_d5dSwAC. – ISBN 9780387689548

[Hasbro] HASBRO: *Wikipedia: Risiko*. – URL http://de.wikipedia.org/wiki/Risiko_(Spiel)

[Hernandez und Ortega 2010] HERNANDEZ, Frank E. ; ORTEGA, Francisco R.: Eberos GML2D: a graphical domain-specific language for modeling 2D video games. In: *Proceedings of the 10th Workshop on Domain-Specific Modeling*. New York, NY, USA : ACM, 2010 (DSM '10), S. 4:1–4:1. – URL http://doi.acm.org/10.1145/2060329.2060342. – ISBN 978-1-4503-0549-5

[Jones 1996] JONES, C.: *SPR Programming Languages Table Release 8.2*. 1996. – URL http://www.theadvisors.com/langcomparison.htm

[Järvinen 2003] JÄRVINEN, Aki: Making and breaking games: a typology of rules. In: MARINKA, Copier (Hrsg.) ; JOOST, Raessens (Hrsg.): *Level Up Conference Proceedings:*

Proceedings of the 2003 Digital Games Research Association Conference. Utrecht : University of Utrecht, November 2003, S. 68–79. – URL http://www.digra.org/dl/display_html?chid=05163.56503.pdf

[Järvinen 2008] JÄRVINEN, Aki: *Games without Frontiers: Theories and Methods for Game Studies and Design*, University of Tampere, Dissertation, 2008. – 416 S. – URL http://acta.uta.fi/pdf/978-951-44-7252-7.pdf

[Juul 2005] JUUL, Jesper: *Half-Real: Video Games between Real Rules and Fictional Worlds*. The MIT Press, 2005. – 243 S. – URL http://www.amazon.com/dp/0262101106

[Juul 2007] JUUL, Jesper: A Certain Level of Abstraction. In: AKIRA, Baba (Hrsg.): *Situated Play: Proceedings of the 2007 Digital Games Research Association Conference*. Tokyo : The University of Tokyo, September 2007, S. 510–515. – URL http://www.digra.org/dl/display_html?chid=07312.29390.pdf

[Kirk 1965] KIRK, H. W.: Use of decision tables in computer programming. In: *Commun. ACM* 8 (1965), Januar, Nr. 1, S. 41–43. – URL http://doi.acm.org/10.1145/363707.363725. – ISSN 0001-0782

[Magerkurth u. a. 2004] MAGERKURTH, Carsten ; MEMISOGLU, Maral ; ENGELKE, Timo ; STREITZ, Norbert: Towards the next generation of tabletop gaming experiences, 2004, S. 73–80

[Mernik u. a. 2005] MERNIK, Marjan ; HEERING, Jan ; SLOANE, Anthony M.: When and how to develop domain-specific languages. In: *ACM Comput. Surv.* 37 (2005), Dezember, Nr. 4, S. 316–344. – URL http://doi.acm.org/10.1145/1118890.1118892. – ISSN 0360-0300

[Parlett] PARLETT, David: *RULES OK or Hoyle on troubled waters*. – URL http://www.davidparlett.co.uk/gamester/rulesOK.html

[Parlett 1990] PARLETT, D.S.: *The Oxford guide to card games*. Oxford University Press, 1990. – URL http://books.google.com/books?id=gtPfAAAAMAAJ. – ISBN 9780192141651

[Parlett 1999] PARLETT, D.S.: *The Oxford history of board games*. Oxford University Press, 1999. – URL http://books.google.de/books?id=rH6DAAAAMAAJ. – ISBN 9780192129987

[Parr 2010] PARR, T.: *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 2010 (Pragmatic Bookshelf Series). – URL http://books.google.de/books?id=C7xuPgAACAAJ. – ISBN 9781934356456

[Perron und Wolf 2008] PERRON, B. ; WOLF, M.J.P.: *The Video Game Theory Reader 2*. Routledge, 2008. – URL http://books.google.de/books?id=ONgfB1k52joC. – ISBN 9780415962827

[Ray 2004] RAY, E.T.: *Einführung in XML*. O'Reilly, 2004. – URL http://books.google.de/books?id=CBL6iyhtwBIC. – ISBN 9783897213708

[Rouse und Ogden 2005] ROUSE, R. ; OGDEN, S.: *Game design: theory & practice*. Wordware Pub., 2005 (Wordware game developer's library). – URL http://books.google.de/books?id=hXwhAQAAIAAJ. – ISBN 9781556229121

[Salen und Zimmerman 2004] SALEN, K. ; ZIMMERMAN, E.: *Rules of play - Game design fundamentals*. The MIT Press, 2004

[Schank und Abelson 1977] SCHANK, R.C. ; ABELSON, R.P.: *Scripts, plans, goals, and understanding: an inquiry into human knowledge structures*. L. Erlbaum Associates, 1977 (Artificial intelligence series). – URL http://books.google.de/books?id=YZ99AAAAMAAJ. – ISBN 9780470990339

[Schell 2008] SCHELL, J.: *The art of game design: a book of lenses*. Morgan Kaufmann, 2008. – URL http://books.google.co.ke/books?id=RV1EcAAACAAJ

[Sicart 2008] SICART, Miguel: Defining Game Mechanics. In: *Game Studies* 8 (2008), Nr. 2, S. 1–18. – URL http://gamestudies.org/0802/articles/sicart

[Smith u. a. 2010] SMITH, Adam M. ; NELSON, Mark J. ; MATEAS, Michael: L UDOCORE : A Logical Game Engine for Modeling Videogames. In: *Elements* (2010), S. 91–98. – URL http://users.soe.ucsc.edu/~amsmith/papers/ieeecig10_ludocore.pdf

[Spinellis 2001] SPINELLIS, Diomidis: *Notable design patterns for domain-specific languages*. 2001

[Taha 2008] TAHA, Walid: *Domain-Specific Languages*. 2008

[Wei u. a. 2010] WEI, Huaxin ; BIZZOCCHI, Jim ; CALVERT, Tom: Time and space in digital game storytelling. In: *Int. J. Comput. Games Technol.* 2010 (2010), January, S. 8:1–8:23. – URL http://dx.doi.org/10.1155/2010/897217. – ISSN 1687-7047

[Wikipedia a] WIKIPEDIA: *Entscheidungstabelle*. – URL http://de.wikipedia.org/wiki/Entscheidungstabelle

[Wikipedia b] WIKIPEDIA: *Wikipedia: Blackjack*. – URL http://de.wikipedia.org/wiki/Black_Jack

[Wikipedia c] WIKIPEDIA: *Wikipedia: Schach*. – URL http://de.wikipedia.org/wiki/Schach

A. Analyse

A.1. Anforderungen

A.1.1. Anforderung an die domänenspezifische Sprache

DSL-1 Konformität

DSL-2 Orthogonalität

DSL-3 Unterstützung

DSL-4 Erweiterbarkeit

DSL-5 Langlebigkeit

A.1.2. Anforderung an die Entwicklungsumgebung

IDE-1 Ansichten für Entwicklungsumgebung

IDE-2 Generierung von Code

IDE-3 Diskussion Debugging-Funktionalität

A.1.3. Anforderung an die Test-Runtime

TR-1 Architektur der Test-Runtime skizzieren

TR-2 Es soll gezeigt werden, dass die Ansichten mehrerer Spieler modelliert werden können

TR-3 Umsetzung anhand eines Prototypen, der eine ausführbare Version des Referenzbeispieles darstellt

A.1.4. Anforderung an das Framework

FW-1 Trennung zwischen generierten Code aus DSL-Processor und Framework

FW-2 Die Kommunikation der Runtime soll für das Framework transparent sein

FW-3 Element-Typen, Vererbung und Instanzen

FW-4 Abläufe der Spielmechanik

FW-5 Kontext und Selektoren

FW-6 Regeln

FW-7 Beeinflussung des Spielzustandes

FW-8 Sichtbarkeit

FW-9 Timer

FW-10 Spielfelder

FW-11 Zufallsgenerator

FW-12 Abbildung der domänenspezifischen Sprache gewährleistet

B. Ohne Furcht und Adel

B.1. Offizielles Regelwerk

Ohne Furcht und Adel

von Bruno Faidutti, für 2 – 7 Spieler ab 10 Jahren

65 Bauwerkkarten
20 grüne, 12 gelbe, je 11 rote, blaue und lila Karten

8 Charakterkarten

1 Kronenkarte mit Standfuß

7 Übersichtskarten

30 Goldstücke

Der Nebel legt sich über die vom Morgentau bedeckten Felder, die Sonne zeigt bereits am Horizont ihr Antlitz und erhellt die fruchtbaren Ebenen. Durch die Nebelschwaden kann man sie sehen, die tapferen Recken, die sich aufmachen, das Land zu befrieden und es in voller Schönheit erblühen zu lassen. Keiner ahnt die wahren Gründe und Intrigen, die hinter all dem stehen. Wie Marionetten werden die Edelleute des Landes dazu missbraucht, den Interessen der zwielichtigen Helden zu dienen ...

3. Buch des Stadtschreibers Junan, 13. Hornung 1478

Jeder Spieler versucht, mit der Auslage von acht möglichst wertvollen Bauwerkkarten seine eigene Stadt zu errichten. Hierzu schlüpfen die Spieler immer wieder in die Rolle anderer Charaktere, um sich deren Vorteile zu Nutze zu machen.

Vorbereitung

Die Kronenkarte wird auf den Standfuß gesteckt. Der älteste Spieler wird in der ersten Runde zum König ernannt und stellt als Zeichen seiner Würde die Kronenkarte vor sich auf. Zwei Spieler mischen in getrennten Stapeln die Bauwerk- und Charakterkarten. Die beiden Stapel werden verdeckt in die Mitte des Tisches gelegt, daneben kommen die 30 Goldstücke.

Dann nimmt sich jeder Spieler:
- 4 Bauwerkkarten vom verdeckten Stapel
- 2 Goldstücke aus der Tischmitte
- 1 Übersichtskarte

Die Karten

1) Die Bauwerkkarten
Jede Bauwerkkarte ist einer bestimmten Farbe zugeordnet:
Blau - Vorteile für den Prediger
Grün - Vorteile für den Händler
Gelb - Vorteile für den König
Rot - Vorteile für den Söldner
Violett - keine Vorteile für einen bestimmten Charakter. Die violetten Bauwerkkarten sind durchschnittlich teurer und wertvoller als die Bauwerkkarten der vier anderen Farben. Der aufgedruckte Text verleiht dem Spieler einen besonderen Vorteil, der eine solche Karte ausliegen hat. Dieser Vorteil kann jede Runde genutzt werden, wenn der Spieler an der Reihe ist.

Kosten und Punkte einer jeweiligen Bauwerkkarte sind identisch und an der Anzahl der aufgedruckten Goldstücke erkennbar.

2) Die Charakterkarten
Jede der acht Charakterkarten hat besondere Eigenschaften, die auf der nächsten Seite erklärt werden.

Ablauf einer Runde (für 4 - 7 Spieler)

(Die Änderungen bei 2 und 3 Spielern finden Sie am Ende dieser Anleitung). Der König ist Startspieler. Er mischt die Charakterkarten und legt je nach Spielerzahl 0-2 Karten offen in die Tischmitte. Befindet sich die Charakterkarte „König" unter den offenen Karten, wird sie wieder untergemischt und an ihrer Stelle eine andere Karte aufgedeckt.

Spielerzahl	Offene Karten
4	2
5	1
6-7	0

Dann legt er noch die nächste Karte verdeckt daneben.

Die restlichen Karten schaut sich der König an und wählt eine aus, die er verdeckt vor sich ablegt. Die übrigen Karten reicht er verdeckt an den linken Nachbarn weiter, der nun ebenfalls eine Karte auswählt, vor sich ablegt und die Karten weiterreicht. Für den letzten Spieler bleiben 2 Karten. Eine behält er und legt die andere verdeckt in die Tischmitte.

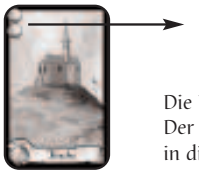
(Sonderfall: Bei 7 Spielern erhält der Letzte nur 1 Karte vom vorhergehenden Spieler. Er nimmt nun die von König zu Beginn verdeckt abgelegte Karte hinzu. Aus diesen beiden wählt er eine aus und legt die andere verdeckt in die Tischmitte.)

Anschließend ruft der König nacheinander die Charaktere in der **auf der Übersichtskarte angegebenen Reihenfolge** auf. Hat ein Spieler die aufgerufene Charakterkarte vor sich liegen, muss er sich zu erkennen geben, die Karte aufdecken und seinen Zug ausführen. Die Karte wird danach auf dem Charakterkartenstapel abgelegt. Meldet sich kein Spieler zu Wort - weil sich der Charakter unter den abgelegten Karten befindet oder weil der Charakter gemeuchelt wurde -, ruft der König den nächsten Charakter auf.
Sind alle acht Charakterkarten abgelegt, beginnt eine neue Runde.

Ablauf eines Charakterzuges

Hat sich ein Charakter zu erkennen gegeben, beginnt sein Spielzug:
- Er nimmt sich 2 Goldstücke **oder**
- Er nimmt sich 2 Bauwerkkarten vom verdeckten Stapel auf die Hand und wählt nun eine Handkarte aus, die er auf den Ablagestapel legt.
- Anschließend darf er 1 Bauwerkkarte aus der Hand vor sich auf den Tisch legen und seine Stadt weiterbauen. Er zahlt die auf der Karte angegebene Anzahl Goldstücke in die Tischmitte.

Beispiel:



Die Kirche wurde ausgelegt.
Der Spieler muss 2 Goldstücke in die Kasse zahlen.

- Jederzeit während seines Zuges **darf** er seine Charaktereigenschaft einsetzen (Ausnahme: König). Dies bedeutet, dass man sich für einen Zeitpunkt entscheiden muss, an dem man seine Eigenschaft einsetzt. (siehe auch Beispiel 1 und 2). So darf etwa der Händler nicht zu Beginn **und** am Ende seines Zuges Gold für ausliegende grüne Bauwerkkarten kassieren.

Die 8 Charaktereigenschaften

1) **Meuchler**

Er benennt einen Charakter, den er in dieser Runde meucheln will. Wird der ausgewählte Charakter anschließend aufgerufen, gibt dieser sich **nicht** zu erkennen und darf in dieser Runde auch keine Aktion durchführen. Hat der Meuchler beispielsweise den Händler ausgewählt, setzt der Händler diese Runde aus.

2) **Dieb**

Er benennt einen Charakter, den er in dieser Runde bestehlen will. Er darf nicht den Meuchler oder den gemeuchelten Charakter wählen.
Wird der ausgewählte Charakter anschießend aufgerufen, gibt er sich zu erkennen und muss dem Dieb sofort sein gesamtes Gold übergeben. Hat der Dieb beispielsweise den Prediger ausgewählt, muss der Prediger dem Dieb seinen gesamten Vorrat an Goldstücken übergeben. Gibt sich der Charakter nicht zu erkennen, weil die entsprechende Karte verdeckt abgelegt wurde, hat der Dieb Pech gehabt und geht leer aus.

3) **Magier**

Er darf entweder
- seine Handkarten mit den Handkarten eines Mitspielers tauschen (dies können auch unterschiedlich viele Karten sein), **oder**
- eine beliebige Anzahl Handkarten auf den offenen Ablagestapel legen und die gleiche Anzahl Bauwerkkarten nachziehen.

4) **König**

Er erhält die Kronenkarte und übernimmt die Aufgaben des Königs. Seine Aufgaben sind: Er ist Startspieler in der nächsten Runde und ruft ab sofort die weiteren Charaktere auf. Der König erhält 1 Goldstück für jede gelbe Bauwerkkarte, die er ausliegen hat.
Gibt sich kein neuer König zu erkennen (weil seine Karte verdeckt abgelegt oder sein Charakter gemeuchelt wurde), bleibt der bisherige König im Amt und übernimmt auch weiter dessen Aufgaben.
Sonderfälle:
a) Wird der König gemeuchelt, bleibt der amtierende König bis zum Ende der Runde im Amt. Zu Beginn der nächsten Runde wechselt die Kronenkarte zum in der Runde zuvor gemeuchelten König.
b) Ist die Königskarte abgelegt worden, bleibt der amtierende König auch in der nächsten Runde im Amt.

5) **Prediger**

Seine Bauwerkkarten können vom Söldner nicht entfernt werden. Der Prediger erhält 1 Goldstück für jede blaue Bauwerkkarte, die er ausliegen hat.

6) **Händler**

Er erhält 1 Goldstück. Der Händler erhält 1 Goldstück für jede grüne Bauwerkkarte, die er ausliegen hat.

7) **Baumeister**

Er erhält 2 Bauwerkkarten. Der Baumeister darf in seinem Zug bis zu 3 Bauwerkkarten auslegen. So kann der Baumeister beim Spielende mit seinem letzten Zug auch mehr als 8 Karten ausliegen haben.

8) **Söldner**

Er darf eine Bauwerkkarte eines Mitspielers entfernen. Karten, die nur 1 Goldstück wert sind, darf er kostenlos entfernen. Für andere Bauwerkkarten zahlt er 1 Goldstück weniger als sie den Besitzer gekostet hat. Das Gold kommt in den allgemeinen Vorrat. Hat er eine Bauwerkkarte entfernt, legt er sie auf den Ablagestapel. Der Söldner darf auch Bauwerke bei gemeuchelten Charakteren zerstören. Er darf aber keine Stadt angreifen, die bereits aus 8 oder mehr Bauwerkkarten besteht. Der Söldner erhält 1 Goldstück für jede rote Bauwerkkarte, die er ausliegen hat.

Drei Beispiele verdeutlichen den Einsatz der Charaktereigenschaften

- Der Händler hat zu Beginn seines Zuges kein Gold im Vorrat und nur noch die grüne Karte „Kontor" mit dem Wert 3 auf der Hand. In seiner Stadt liegen bereits 2 grüne Karten und 1 rote Karte. Er entscheidet sich dafür, eine Karte zu nehmen und kassiert im Rahmen seiner Charaktereigenschaft 1 Goldstück. Da er für jede grüne Bauwerkkarte in seiner Stadt ein weiteres Goldstück erhält, hat er nun bereits 3 in seinem Vorrat, mit denen er die Karte „Kontor" auslegen kann. Damit endet sein Zug.
- Hätte unser Händler statt eine neue Karte zu nehmen zu Beginn seines Zuges 2 Goldstücke kassiert, hätte er gemeinsam mit dem Goldstück für seine Charaktereigenschaft 3 Goldstücke zur Verfügung. Damit legt er die Karte „Kontor" aus. Und da der Händler für jede grüne Bauwerkkarte in seiner Stadt ein Goldstück kassiert, kann er am Ende seines Zuges gleich 3 Goldstücke einstreichen.
- Der Magier hat zu Beginn seines Zuges keine Karte mehr auf der Hand und entscheidet sich trotzdem für die Einnahme der beiden Goldstücke. Jetzt sucht er sich einen Mitspieler aus, der noch möglichst viele Bauwerkkarten auf der Hand hält und tauscht mit ihm. Der Mitspieler wird sich ärgern, da er nun selbst keine Handkarte mehr besitzt. Anschließend legt der Magier eine Bauwerkkarte aus, die er gerade mit dem Mitspieler getauscht hat.

Spielende & Sieger

Das Spiel endet, sobald ein Spieler die achte Bauwerkkarte auslegt. Die Runde wird noch zu Ende gespielt. Jeder Spieler erhält nun folgende Punkte:
- Alle Punkte aller Bauwerkkarten in seiner Stadt.
- 3 Punkte, wenn der Spieler Bauwerkkarten in allen 5 unterschiedlichen Farben ausliegen hat.
- 4 Punkte für denjenigen Spieler, der zuerst 8 Bauwerkkarten ausliegen hat.
- 2 Punkte für diejenigen Spieler, die anschließend ebenfalls 8 Bauwerkkarten auslegen.
- Das Gold bringt keine Punkte.

Gewinner ist derjenige Spieler, der die meisten Punkte erzielen kann. Bei Gleichstand gewinnt der Spieler, der die meisten Punkte allein mit seinen Bauwerkkarten erzielen kann.

Änderungen für 2 und 3 Spieler

2 Spieler: Der ältere Spieler erhält die Kronenkarte. Er mischt aus den Charakterkarten 1 Karte heraus, die er verdeckt ablegt. Aus den übrigen Karten sucht er sich eine Karte aus und reicht den Stapel an den Mitspieler weiter. Dieser wählt eine Karte aus, die er behält und 1 weitere, die er auf den Ablagestapel legt. Der Startspieler sucht nun ebenfalls 2 Karten aus, die er behält bzw. ablegt. Zuletzt wählt wieder der Gegenspieler, so dass nun jeder Spieler 2 Charakterkarten besitzt und 4 Karten auf dem Ablagestapel liegen.

3 Spieler: Der älteste Spieler mischt aus den Charakterkarten 1 heraus, die er verdeckt ablegt, und behält eine weitere. Nun wandert der Stapel reihum weiter. Jeder Spieler sucht sich 1 Karte aus und gibt den Stapel an den linken Nachbarn weiter. Der Spieler, der zuletzt 2 Karten bekommt, wählt eine Karte aus und legt die andere auf den Ablagestapel, so dass nun jeder Spieler 2 Charakterkaretn besitzt.

Spielablauf: Das Spiel läuft wie in der großen Runde: Der König ruft die Charaktere in der vorgegebenen Reihenfolge auf. Jeder Spieler ist nun allerdings zweimal pro Runde am Zug und kann dies taktisch nutzen, indem er etwa die Einnahmen seines ersten Charakters spart, um mit dem nächsten ein teures Bauwerk zu errichten.

Für unzählige Testrunden, Anregungen und Vorschläge bedankt sich der Autor bei Nadine Bernard, Pascal Bernard, Maud Bissonnet, Scarlett Bocchi, Frank Branham, David Calvo & Muriel Sitruk, Brent & Maryann Carter, Fabienne Cazalis, Pitt Crandlemire, Isabelle Duvaux, Thierry Fau, Nathalie Grégoire, Philippe Keyaerts, David Kuznik, Serge Laget, Myriam Lemaire, Pierre Lemoigne, Tristan Lhomme, Hervé Marly, Bernard Mendiburu, Hélène Michaux, Steffan O'Sullivan, Philippe des Pallieres, Jean-Marc Pauty, Pierre Rosenthal, Fred Savart, Mik Svellov, Irène Villa, Alan Moon und den Teilnehmern seines 10. Gathering of friends.

Illustration: Julien Delval, Florence Magnin, Jean-Louis Mourier
Grafik: Cyrille Daujean

Autor und Verlag bedanken sich bei der Firma Adlung-Spiele (Remseck) und dem Autor Marcel-André Casasola Merkle für die freundliche Genehmigung, das Verteilungselement aus dem Spiel VERRÄTER (© 1998) verwenden zu dürfen.

Haben Sie Anregungen, Fragen oder Kritik?
Schreiben Sie an unsere E-Mail-Adresse: glueck@cube-net.de oder
per Post: Hans im Glück Verlag
Birnauerstr. 15
80809 München
Fax: 089/302336

Varianten und Hinweise zum Spiel, zum Autor und über unser weiteres Programm finden Sie im Internet auf unserer Homepage: http://www.hans-im-glueck.de

B.2. Beispielrunde

Eine Beispielrunde soll den Ablauf und die Kernmerkmale der Spielmechanik verdeutlichen. Zuerst sollen die in diesem Beispiel verwendeten Karten vorgestellt werden. Tabelle B.1 enthält die unterschiedlichen Charaktere, die in der Reihenfolge aufgelistet sind, nach der sie in einer Spielrunde aufgerufen werden. Die Reihenfolge wird in der Tabelle in der Zeile „#“ festgehalten

Das Spiel selbst enthält sechs grüne, drei gelbe, vier blaue, vier rote unterschiedliche Bauwerktypen sowie zusätzlich 11 lila Bauwerke mit Spezialfähigkeiten. Die Spezialbauwerke, mit einer Ausnahme, sind nur einmalig vorhanden, von den grünen Bauwerken existieren insgesamt 20, von den gelben 12, von rot und blau jeweils 11. Insgesamt sind 65 Bauwerkkarten im Spiel vorhanden. Der Einfachheit halber werden hier nur die in der Beispielrunde namentlich genutzten Bauwerke vorgestellt.

Vier Spieler (Anna, Ben, Carla, und Dennis) befinden sich in einem bereits fortgeschrittenen Spiel von OFUA, es beginnt eine neue Runde und die Charakterkarten werden neu gemischt. Die Sitzreihenfolge lautet: Anna, Ben, Carla und Dennis. Anna ist Startspieler, da sie zuletzt den König ausgewählt hatte. Zu Beginn der neuen Spielrunde werden zunächst die Charaktere ausgewählt.

1. Es werden zwei zufällige Charakterkarten offen auf den Tisch gelegt. Es handelt sich um den Dieb und den Prediger.
2. Anna schaut sich Anna die nächste Charakterkarte an und legt sie verdeckt auf den Tisch. Es handelt sich um den Händler.
3. Anna wählt für sich selbst den Meuchler aus und gibt die restlichen 4 Karten weiter.
4. Ben erhält also jetzt die verbleibenden Charakterkarten (Magier, König, Baumeister, Söldner). Da Ben die offenen Charakterkarten kennt (Dieb und Prediger), muss Anna entweder den Meuchler oder den Händler gewählt haben. Ben selbst nimmt sich den Baumeister und gibt die restlichen 3 Karten an Carla.
5. Carla wählt nun aus Magier, König und Söldner den Magier aus und reicht die verbleibenden 2 Karten an Dennis.
6. Dennis nimmt sich den Söldner und legt den König verdeckt auf den Tisch.

Die Durchführung der Charakterauswahl ist dabei von der Anzahl der Spieler abhängig und variiert in der Anzahl der zu Beginn offenen gelegten Karten sowie in der Anzahl der zu ziehenden Karten pro Spieler. So wird gewährleistet, dass das Spiel auch mit unterschiedlich vielen Spielern interessant bleibt und einen eigenen Schwerpunkt entwickelt. Ein Spiel mit zwei Spielern unterscheidet sich gravierend von einem Spiel mit fünf Spielern.

#	Charakter	Sonderfähigkeiten
1	Meuchler	**Aktion Meucheln**: bestimmt einen Charakter, der gemeuchelt werden soll. Wenn dieser Charakter aufgerufen wird, gibt sich der Spieler nicht zu erkennen und der Zug dieses Charakters verfällt. Sollte es sich nach Abschließen des Söldner-Zuges zeigen, dass der gemeuchelte Charakter der König ist und ein Spieler diesen ausgewählt hatte, wird dieser Spieler dennoch der neue König und somit Startspieler.
2	Dieb	**Aktion Stehlen**: Bestimmt einen Charakter, der bestohlen werden soll, dabei darf weder der Meuchler noch der gemeuchelte Charakter bestohlen werden. Wenn dieser Charakter ausgerufen und von einem Spieler ausgewählt wurde, erhält er das Gold, das er zu Beginn des Zuges seines Charakterzuges besitzt, ansonsten erhält der Dieb nichts.
3	Magier	Es kann nur „Karten tauschen“ oder „Karten neu ziehen“ ausgeführt werden: **Aktion Karten tauschen**: Der Spieler kann alle seine Handkarten gegen alle Handkarten eines Mitspielers tauschen. **Aktion Karten neu ziehen**: Der Spieler legt alle seine Handkarten ab und zieht die entsprechende Anzahl von Bauwerkkarten nach.
4	König	Wenn der König ausgerufen wird, wird der Spieler, der den König ausgewählt hat, zum nächsten Startspieler. Wurde der König von keinem Spieler ausgewählt, bleibt der aktuelle Startspieler. **Aktion Gold nehmen**: Nimmt sich dieser Spieler Gold, erhält er zusätzlich für jedes gelbe Bauwerk, das er besitzt, 1 weiteres Gold.
5	Prediger	Die Bauwerke des Predigers können in dieser Runde nicht zerstört werden. **Aktion Gold nehmen**: Nimmt sich dieser Spieler Gold, erhält er zusätzlich für jedes blaue Bauwerk, das er besitzt, 1 weiteres Gold.
6	Händler	Der Spieler erhält sofort 1 Gold. **Aktion Gold nehmen**: Nimmt sich dieser Spieler Gold, erhält er zusätzlich für jedes grüne Bauwerk, das er besitzt, 1 weiteres Gold.
7	Baumeister	Der Spieler zieht sofort 2 Bauwerkkarten. Der Spieler kann diese Runde bis zu 3 Bauwerke errichten.
8	Söldner	Der Spieler kann eine Bauwerkkarte eines Mitspielers zerstören, wenn er die Kosten des Bauwerkes um 1 Gold reduziert bezahlt. Bauwerke mit Kosten von 1 Gold können umsonst zerstört werden. Das Bauwerk wird dann auf den Ablagestapel gelegt. Er kann nur Bauwerke von Spielern zerstören, die weniger als 8 Bauwerke besitzen. **Aktion Gold nehmen**: Nimmt sich dieser Spieler Gold, erhält er zusätzlich für jedes rote Bauwerk, das er besitzt, 1 weiteres Gold.

Tabelle B.1.: Charakterkarten

Name	Farbe	Kosten	Sonderfähigkeiten
Kirche	blau	2	
Wirtshaus	grün	1	
Kontor	grün	3	
Wachturm	lila	3	Kann nicht durch den Söldner zerstört werden
Friedhof	lila	5	Zerstört der Söldner ein Gebäude, kann der Besitzer 1 Gold zahlen und sich das zerstörte Bauwerk auf die Hand nehmen. Dies ist nur erlaubt, wenn er selbst nicht der Söldner ist.
Schmiede	lila	5	Der Spieler darf während seines Zuges für 3 Gold 2 Bauwerke ziehen.

Tabelle B.2.: Bauwerkkarten

Da die Charakterauswahl abgeschlossen ist, wird nun die Hauptphase durchgeführt:

1. Meuchler Da Anna den Meuchler besitzt, darf sie als erste ihren Zug durchführen. Sie entscheidet sich, den König zu meucheln. Nun kann Anna sich entweder zwei Bauwerkkarten ziehen, eine davon auf die Hand nehmen und die andere ablegen, oder zwei Gold nehmen. Anna entscheidet sich dazu, zwei Gold zu nehmen und hat jetzt insgesamt fünf Gold. Anna hat noch den Friedhof auf der Hand, den sie jetzt bauen möchte. Dazu bezahlt sie fünf Gold und legt den Friedhof offen vor sich hin und hat somit kein Gold mehr übrig. Anschließend beendet sie den Zug.

2. Dieb Nun wird der Dieb ausgerufen, da aber kein Spieler den Dieb wählen konnte (er war eine der beiden Karten die am Anfang offen auf den Tisch gelegt wurden), geht die Phase zum nächsten Charakter, dem Magier, über.

3. Magier Carla hat den Magier ausgewählt und darf nun ihren Zug beginnen. Carla hat noch ein Bauwerk auf der Hand, aber nicht genügend Geld. Daher nimmt sie sich zwei Gold und besitzt nun insgesamt drei Gold. Jetzt nutzt sie die Fähigkeit des Magiers und tauscht ihre eine Karte mit den Karten von Dennis, der vier Bauwerke auf der Hand hat. Bei den Bauwerken handelt es sich um drei Wirtshäuser, die ein Gold und ein Kontor, welches drei Gold zum Errichten kostet. Da Carla nur ein Bauwerk pro Runde errichten kann, bezahlt Carla drei Goldstücke, um das Kontor zu bauen, da dieses auch entsprechend mehr Siegpunkte einbringt.

4. König Der König wurde von dem Meuchler ermordet. Der nächste Charakter wird aufgerufen.

5. Prediger Der Prediger wurde ebenso wie der Dieb am Anfang der Charakterauswahl offen auf den Tisch gelegt und konnte nicht ausgewählt werden.

6. Händler Der Händler wurde nach den beiden offenen Charakterkarten verdeckt auf den Tisch gelegt, nur Anna weiß sicher, dass der Händler nicht ausgewählt wurde.

7. Baumeister Da Ben den Baumeister gewählt hat, ist er nun am Zug. Da der Baumeister das Errichten von bis zu drei Bauwerken erlaubt, wollte Ben ursprünglich die drei Wirtshäuser errichten, die auf der Hand hatte, bevor Carla mit der Fähigkeit des Magier die Bauwerke mit ihm getauscht hat. Jetzt hat Ben nur noch ein einziges Bauwerk mit Kosten von sechs Gold auf der Hand, welches er mit seinen drei Gold nicht bauen kann. Zusätzlich erlaubt der Baumeister das Ziehen von zwei Bauwerkkarten und Ben hat wieder drei Bauwerke auf der Hand. Nun nimmt sich Ben zwei Gold und hat nun insgesamt fünf Gold. Damit baut er den Wachturm für drei und die Kirche für zwei Gold und hat damit seinen gesamten Goldvorrat aufgebraucht.

8. Söldner Nun kann Dennis seinen Zug machen, da er den Söldner ausgewählt hatte. Dennis hat bereits vier rote Bauwerke und eine Schmiede errichtet. Allerdings hat er weder Bauwerke auf der Hand noch Gold. Daher entscheidet er sich dazu, Gold zu nehmen. Dafür erhält er die üblichen zwei und die für seine vier roten Bauwerke zusätzlichen vier, insgesamt also sechs Gold. Durch das Nutzen der Spezialfunktion der Schmiede zieht er zwei Bauwerke und muss dafür drei Gold bezahlen. Mit einem Gold baut er ein Wirtshaus und hat nun noch zwei Gold übrig. Für zwei Gold kann er mit der Fähigkeit des Söldners Bauwerke zerstören, die bis zu drei Gold kosten. Dies trifft auf das Kontor von Carla und den Wachturm von Ben zu. Da die Spezialfähigkeit des Wachturmes die Zerstörung durch den Söldner verhindert, bezahlt Dennis zwei Gold, um das Kontor von Carla zu zerstören. Normalerweise würde das zerstörte Bauwerk auf den Ablagestapel gelegt werden, da Anna aber den Friedhof besitzt, kann sie sich entscheiden, ob sie das zerstörte Kontor für ein Gold auf die Hand nehmen möchte. Da Anna allerdings kein Gold mehr hat, kann sie diese Fähigkeit nicht einsetzen.

Abschluss Nachdem der Söldner-Zug beendet wurde, wird überprüft, ob ein Spieler acht oder mehr Gebäude besitzt. Wenn ja, ist das Spiel zu Ende und der Sieger ergibt sich aus der Anzahl der Siegpunkte. Zusätzliche Siegpunkte am Ende erhält der Spieler der als erstes das achte Bauwerk errichtet hat, sowie die anderen Spieler für bestimmte Kombinationen von Bauwerken. Wenn noch kein Spieler acht Bauwerke errichtet hat, wird die nächste Runde mit der Charakterauswahl begonnen, wobei derjenige der Startspieler ist, der als letztes den König ausgewählt hatte.

C. lGold

C.1. Beispiel der Sprachelemente

C.1.1. Game-Beispiel

Das Beispiel zeigt die grundlegende Struktur für eine Spieldefinition in lGold.

```
<game name="Ohne Furch und Adel" version="1.0"
      xmlns="http://www.goldflam.de/lgold/v1.0"
      xmlns:lgold="http://www.goldflam.de/lgold/v1.0">
  <gameDefinition>
   <gameState>
   <!-- Instanziierungsbereich für Space-Instanzen,
         Actor-Instanzen und Actor-Instanz-Listen  -->
   </gameState>
   <actors>
   <!-- Typdefinitionsbereich für Actors-->
   </actors>
   <attributes>
   <!-- Typdefinitionsbereich für Attributes-->
   </attributes>
   <objects>
     <!-- Typdefinitionsbereich für Objects-->
   </objects>
   <actions>
     <!-- Typdefinitionsbereich für Actions-->
   </actions>
   <rules>
     <!-- Typdefinitionsbereich für Rules-->
   </rules>
   <effects>
     <!-- Typdefinitionsbereich für Effects-->
   </effects>
   <spaces>
     <!-- Typdefinitionsbereich für Spaces-->
   </spaces>
   <fields>
     <!-- Typdefinitionsbereich für Fields-->
   </fields>
   <phases>
   <!-- Typdefinitionsbereich für Phases-->
   </phases>
   <operations>
     <!-- Typdefinitionsbereich für Operations-->
   </operations>
```

```
   </gameDefinition>
</game>
```

C.1.2. Gamestate-Beispiel

Bei OFuA existieren unterschiedliche globale Elemente, unter anderem zählt das Spielfeld, die Reihenfolge der Spieler und welcher der Spieler derzeit König ist, dazu.

```
<gameState>
  <instances>
    <spaceInstance name="gameSpace" typeReference="type.GameSpace"
      instanceId="instance.gameSpace"/>
      <!-- Sonstige Instances -->
  </instances>

<links>
    <actorLinkList name="playerOrder"  linkId="link.playerOrder"
      typeReference="type.playerActor">
        <initialActorLinks>
          <selectAllActors/>
        </initialActorLinks>
    </actorLinkList

   <actorLink name="currentKing" linkId="link.currentKing"
         typeReference="type.playerActor" />
   <!-- Sonstige Links -->
  </links>
</gameState>
```

C.1.3. Actor-Beispiel

Bei OFuA wird der Spieler-Typ folgendermaßen definiert:

```
<actors>
  <actor name="playerActor" typeId="type.playerActor">
    <instances>

      <attributeInstance name="currentGold" typeReference="type.GoldAttribute"
       instanceId="instance.currentGold">
        <integerValue value="2"/>
      </attributeInstance>
      <!-- sonstige AttributeInstances -->
      <spaceInstance name="playerSpace" typeReference="type.playerSpace"
       instanceId="instance.playerSpace"/>
    </instances>
  </actor>
</actors>
```

Für den hier definierten `playerActor` wurde mehrere Instanzen von **Attributes** und eines **Spaces** erzeugt. Die `currentGold`-Instanz zeigt an, wie viel Gold der Spieler aktuell besitzt. Die `playerSpace`-Instanz stellt wiederum den Space dar, welche sämtliche **Fields** des Spielers enthält.

C.1.4. Attribute-Beispiel

Durch den beschriebenen Aufbau ist es beispielsweise möglich, einen abstrakten Objekt-Typ „Bauwerk“ zu definieren, der als Eigenschaft ein „Kosten“-Attribut besitzt, jedoch keinen konkreten Wert für diesen Typ definiert. Ein konkreter Objekt-Typ „Kirche“ kann so das „Bauwerk“ erweitern und für die Kosten einen Wert festlegen. Durch diese Struktur können andere Elemente flexibler beschrieben werden.

An dieser Stelle soll die Definition des Gold-**Attribute** als Beispiel dienen. Das Gold-**Attribute** wird anschließend in Form von Kosten für den **Object**-Typ `Building` durch die Definition einer `abstractAttributeInstance` verknüpft und in dem konkreten „Kirchen“-**Object** über das `attributeInstanceValue` mit einem Initialwert versehen.

Zuerst erfolgt die Definition des Attributes selbst:

```
<attributes>
  <integerAttribute name="Gold" typeId="type.Gold"/>
</attributes>
```

Für das Gold-**Attribute** wird das `integerAttribute` verwendet, es kann nur ganze Zahlen erfassen.

```
<abstractObject name="Building" typeId="type.Building">
  <instances>

    <abstractAttributeInstance name="buildingCosts"
       typeReference="type.Gold"
       instanceId="instance.buildingCosts"/>
     <!-- sonstige Instanzen-->
  </instances>
</abstractObject>
```

In dem abstrakten **Object** `Building` wurde anhand des `attributeInstance`-Element mit dem `typeReference`-Attribut unter der Angabe der von dem Gold-Attribute definierten `typeId` eine abstrakte Instanz des **Attributes** erzeugt.

Anschließend soll in dem konkreten Objekt-Typ `Church` der Initialwert für die Kosten-Instanz gesetzt werden:

```
<object name="Church" typeId="type.Church" extends="type.Building">
  <instances>
    <attributeInstanceValue instanceReference="instance.buildingCosts">
      <integerValue value="2"/>
    </attributeInstanceValue>
    <!-- sonstige Instanzen -->
  </instances>
</object>
```

Über das `attributeInstanceValue`-Element wird unter der Angabe der `instanceReference` der entsprechende Wert gesetzt.

C.1.5. Object-Beispiel

Das folgende IGold Script zeigt die Definition des abstrakten **Object**-Typs `Building`.

```
<abstractObject name="Building" typeId="type.Building">
<instances>
  <abstractAttributeInstance name="buildingWinPoints"     typeReference="type.WinPoints"
   instanceId="instance.buildingWinPoints"/>

  <abstractAttributeInstance name="buildingCosts"         typeReference="type.Gold"
   instanceId="instance.buildingCosts"/>

  <abstractAttributeInstance name="buildingType"          typeReference="type.BuildingType"
   instanceId="instance.buildingBuildingType"/>

  <attributeInstance name="buildingCardType"              typeReference="type.CardType"
   instanceId="instance.buildingCardType">
    <enumValue enumValueReference="cardType.Building"/>
  </attributeInstance>

  <actionInstance name="ConstructBuilding" instanceId="instance.ConstructBuilding"
   typeReference="type.ConstructBuildingAction"/>

  <effectInstance instanceId="instance.IncreaseNumberOfBuildingsBuiltEffect"
   typeReference="type.IncreaseNumberOfBuildingsBuiltEffect"/>

  <effectInstance instanceId="instance.IncreaseWinPointsEffect"
   typeReference="type.IncreaseWinPointsEffect"/>
</instances>
</abstractObject>
```

Anschließend wird ein konkreter **Object**-Typ den den abstrakten `Building`-Typ erweitern und für die `abstractAttributeInstance`-Elemente Werte festlegen.

```
<object name="Church" typeId="type.Church" extends="type.Building">
  <instances>
    <attributeInstanceValue instanceReference="instance.buildingBuildingType">
      <enumValue enumValueReference="buildingType.Blue"/>
    </attributeInstanceValue>
```

```
    <attributeInstanceValue instanceReference="instance.buildingCosts">
      <integerValue value="2"/>
    </attributeInstanceValue>

    <attributeInstanceValue instanceReference="instance.buildingWinPoints">
      <integerValue value="2"/>
    </attributeInstanceValue>
  </instances>
</object>
```

C.1.6. CustomLayout-Beispiel

Bei OFuA wird kein Spielbrett mit einer bestimmten Struktur benötigt. Der globale **Space** und der eines jeden Spielers dienen einem konzeptionellen Zweck. Sie enthalten für jeden Bereich ein **Field**. Auf diese Art hat jeder Spieler beispielsweise ein **Field** für die Handkarten oder die errichteten Bauwerke. Im globalen **Space** gibt es wiederum ein **Field** für den Nachzieh- oder Ablagestapel. An dieser Stelle soll nun gezeigt werden, wie anhand des `customLayouts` die **Fields** des globalen **Spaces** verknüpft werden können.

```
<space name="GameSpace" typeId="type.GameSpace">
  <instances>
    <fieldInstance name="buildingDrawField" instanceId="instance.buildingDrawField"
        typeReference="type.buildingField"/>
     <!-- Sonstige Instanzen -->
  </instances>

<fieldLayout>
<customLayout>
 <allow>

   <uniDirectionalConnection>
     <fromFieldInstance instanceIdReference="instance.buildingDrawField"/>
     <selectMultipleFieldsByInstanceId>
       <fromSpaces>
         <selectSpacesByInstanceId>
           <spacesFromActors>
             <selectAllActors/>
             <spaceInstanceId instanceIdReference="instance.playerSpace"/>
           </spacesFromActors>
         </selectSpacesByInstanceId>
       </fromSpaces>
       <fieldInstanceId instanceIdReference="instance.playerBuildingHandCardField"/>
     </selectMultipleFieldsByInstanceId>
   </uniDirectionalConnection>

   <!-- Sonstige Verknüpfungen -->
 </allow>
</customLayout>
</fieldLayout>
</space>
```

Dabei wurde das das `buildingDrawField` (der Nachziehstapel) mit den `playerBuildingHandCardField`-**Fields** (Handkartenbereich) sämtlicher Spieler verknüpft.

C.1.7. OneDimensionalLayout-Beispiel

Bei dem `oneDimensionalLayout` werden die Felder entsprechend der Reihenfolge eingeordnet, in welcher diese definiert worden sind. Die Felder des Monopoly-Spielbrettes würden folgendermaßen definiert werden:

```
<space name="MonopolyBoard" typeId="space.MonopolyBoard">

 <instances>
   <fieldInstance name="Go" instanceId="fieldInstance.Go" typeReference="field.Go"/>
   <fieldInstance name="MediterraneanAvenue" instanceId="fieldInstance.MediterraneanAvenue"
    typeReference="field.MediterraneanAvenue"/>
   <!-- Sonstige Straßen-->
   <fieldInstance name="Boardwalk" instanceId="fieldInstance.Boardwalk" typeReference="field.Boardwalk"/>
 </instances>

 <fieldLayout>
   <oneDimensionalLayout connectionType="bidirectional" connectFirstAndLastField="true">
     <fieldInstanceReference instanceIdReference="fieldInstance.Go"/>
     <fieldInstanceReference instanceIdReference="field.MediterraneanAvenue"/>
     <fieldInstanceReference instanceIdReference="fieldInstance.Boardwalk"/>
   </oneDimensionalLayout>
 </fieldLayout>

</space>
```

Durch das Setzen des `connectFirstAndLastField`-Attribut wird das `Boardwalk`-mit dem `Go`-**Field** verknüpft. Da sich die Spielfigureren vor- und rückwärts bewegen müssen, wurde als `connectionType` „bidirectional“ gewählt.

C.1.8. TwoDimensionalLayout-Beispiel

Als Beispiel für einen zweidimensionales Layout soll ein Schachbrett in IGold definiert werden:

```
<space name="ChessBoard" typeId="space.ChessBoard">
<instances>
 <fieldInstance name="A1" typeReference="field.BlackField" instanceId="instance.A1"/>
 <!-- weitere A-Fields -->
 <fieldInstance name="A8" typeReference="field.WhiteField" instanceId="instance.A8"/>
 <fieldInstance name="B1" typeReference="field.BlackField" instanceId="instance.B1"/>
 <!-- weitere B-Fields -->
```

```
 <fieldInstance name="B8" typeReference="field.WhiteField" instanceId="instance.B8"/>
</instances>

<fieldLayout>
 <twoDimensionalLayout howToConnectAdjacentFields="verticalAndHorizontalAndDiagonal"
  connectionType="bidirectional" connectFieldsOnSides="false">
   <column>

     <row>
       <fieldInstanceReference instanceIdReference="instance.A1"/>
       <!-- weitere A-Referenzen -->
       <fieldInstanceReference instanceIdReference="instance.A8"/>
     </row>

     <row>
       <fieldInstanceReference instanceIdReference="instance.B1"/>
       <!-- weitere A-Referenzen -->
       <fieldInstanceReference instanceIdReference="instance.B8"/>
     </row>
   </column>
 </twoDimensionalLayout>
</fieldLayout>
</space>
```

Dabei werden zuerst die Zeilen und anschließend pro Zeile die Spalte und für jedes Feld eine das Feld durch eine `fieldInstance` beschrieben.

C.1.9. Phase-Beispiel

Innerhalb des folgenden Beispiels soll gezeigt werden, dass durch den Einsatz der Vererbung eine elegante Definition der unterschiedlichen Spielphasen für die acht Charaktere möglich ist. Zunächst soll eine abstrakte `CharacterPhase` definiert werden:

```
<abstractPhase name="Characterphase" typeId="type.CharacterPhase">
  <!-- The actions for the CharacterPhase-->
  <instances>
    <actionInstance instanceId="instance.DrawCardsAction" typeReference="type.DrawCardsAction"/>
    <actionInstance instanceId="instance.TakeGoldAction" typeReference="type.TakeGoldAction"/>
    <actionInstance instanceId="instance.FinishTurnAction" typeReference="type.FinishTurnAction"/>
  </instances>

  <links>
    <objectLink name="characterOfPhase" linkId="link.characterOfPhase" typeReference="type.Character"/>
  </links>

  <phaseStates>

    <abstractPhaseState name="StartState" typeId="type.CharacterPhase.StartState"/>
    <phaseState name="CharacterNotSelectedOrDeadState"/>
    <phaseState name="FinalState" typeId="phase.CharacterPhase.FinalState"/>
    <phaseState name="CharacterWasSelectedAndIsAliveState"
                typeId="phase.CharacterPhase.CharacterWasSelectedAndIsAliveState"/>
    <phaseState name="PlayState" typeId="type.CharacterPhase.PlayState"/>
```

```
  </phaseStates>

</abstractPhase>
```

Innerhalb der `Characterphase` wurden nur die unterschiedlichen **Instances**, **Links** und die Definition der **PhaseStates** aufgelistet, ohne näher in die Tiefe zu gehen. Die **PhaseTransitions** wurden an dieser Stelle nicht aufgelistet. Die abstrakte Phasendefinition enthält zum einen jene **Actions**, welche in jeder der Charakter-Spielphasen möglich sind: Karten ziehen (`DrawCardsAction`), Gold nehmen (`TakeGoldAction`) und die Runde beenden (`FinishTurnAction`). Zum anderen wurde ein Link auf ein Charakter-`Object` deklariert. Die unterschiedlichen Spielphasen unterscheiden sich von ihrem Ablauf nur in dem Sinne, dass ein anderer Spieler mit einem anderen Charakter an der Reihe ist.

Der Ablauf einer Charakterphase gliedert sich in die folgenden Zustände:

1. `StartState` ist der Startzustand.
2. `CharacterNotSelectedOrDeadState`, falls der derzeitige Charakter ermordet oder von keinem Spieler ausgewählt wurde. Ansonsten weiter bei `CharacterWasSelectedAndIsAliveState`
3. `CharacterWasSelectedAndIsAliveState`, der Charakter wurde von einem Spieler ausgewählt.
4. `PlayState`, der Spieler mit dem derzeit aktiven Charakter kann seine Zug durchführen
5. `FinalState`

Anschließend kann nun die Spielphase des Meuchlers beschrieben werden, indem sie von der `CharacterPhase` erbt:

```
<phase name="AssasinPhase" typeId="type.AssasinPhase" extends="type.CharacterPhase">

 <phaseStates>
   <phaseState name="AssasinStartState" typeId="type.AssasinPhase.AssasinStartState"
               extends="type.CharacterPhase.StartState"/>
 </phaseStates>

 <instances>
  <actionInstance name="AssasinateCharacter" typeReference="phaseAction.AssasinPhase.AssasinateCharacter"
                  instanceId="instance.AssasinateCharacter"/>
 </instances>

</phase>
```

In der konkreten AssasinPhase wird nun innerhalb des AssasinStartState der characterOfPhase-Link mit dem Meuchler verknüpft. Auf diese Weise wird geprüft, ob der Meuchler von einem Spieler ausgewählt wurde, und erlaubt einem Spieler die Durchführung seines Spielzuges. Zusätzlich wird die **Action** AssasinateCharacter definiert, welche die besondere Eigenschaft des Meuchlers darstellt.

Bei der Definition einer anderen konkreten Charakterphase muss also innerhalb des StartState nur auf einen anderen Charakter verwiesen sowie die zusätzlichen Actions definiert werden.

C.1.10. PhaseState-Beispiel

Im Rahmen dieses Beispiels soll gezeigt werden, wie der abstrakte StartState aus dem Beispiel in Abschnitt C.1.9 definiert worden ist.

```
<abstractPhase name="Characterphase" typeId="type.CharacterPhase">

<links>
    <objectLink name="characterOfPhase" linkId="link.characterOfPhase" typeReference="type.Character"/>
    <!-- Sonstiges -->
  </links>

<phaseStates>

    <abstractPhaseState name="StartState" typeId="type.CharacterPhase.StartState">
      <phaseStateTransitions>
        <phaseStateTransition>
          <phaseRule>
            <phaseExecutionNotStartedRule/>
          </phaseRule>
        </phaseStateTransition>
      </phaseStateTransitions>

      <operations>
        <phaseOperation>
          <setCurrentPhaseExecutionStarted/>
        </phaseOperation>
      </operations>

    </abstractPhaseState>
      <!-- Sonstiges -->
   </phaseStates>
</abstractPhase>
```

Der StartState definiert über die phaseExecutionNotStartedRule-Regel, dass er ausgeführt werden soll, wenn die **Phase** noch nicht als gestartet gilt. Im Rahmen der operations setzt der StartState die **Phase** entsprechend auf „gestartet“.

Anschließend wird der abstrakte StartState durch einen konkreten AssasinStartState der von der CharacterPhase erbenden AssasinPhase konketisiert.

```
<phase name="AssasinPhase" typeId="type.AssasinPhase" extends="type.CharacterPhase">

  <phaseStates>
    <phaseState name="AssasinStartState" typeId="type.AssasinPhase.AssasinStartState"
     extends="type.CharacterPhase.StartState">

      <operations>
        <objectOperation>
          <setObjectLink>

            <objectToLinkTo>
              <selectSingleObject>
                <byType>
                  <objectTypeReferenc typeReference="type.Assasin"/>
                </byType>
              </selectSingleObject>
            </objectToLinkTo>

            <linkToSet>

            </linkToSet>

          </setObjectLink>
        </objectOperation>
      </operations>

    </phaseState>
  </phaseStates>
</phase>
```

Die hier definierte **Operation** wir nach der in `StartState` festgelegten **Operation** ausgelöst. Sie bewirkt, dass der `characterOfPhase`-Link auf das **Object** gesetzt wird, welches den Typ `type.Assasin` besitzt.

C.1.11. InternalOperation-Beispiel

Das folgende Beispiel zeigt, wie die Aktion definiert werden kann, bei der sich ein Spieler Karten vom Nachziehstapel nimmt. Es handelt sich dabei nur um einen Teilausschnitt.

```
<complexAction name="DrawCards" typeId="type.DrawCardsAction">

<fieldOperation>
  <moveObjectsFromFieldToField setFieldOwnerAsObjectOwner="true">
    <fromField>
      <selectField>
        <fromSpace>
          <selectSpace>
            <fromGameState/>
            <byInstanceId instanceIdReference="instance.gameSpace"/>
          </selectSpace>
        </fromSpace>
        <byInstanceId instanceIdReference="instance.buildingDrawField"/>
      </selectField>
    </fromField>
```

```
    <toField>
      <selectField>
        <fromActor>
          <actorWhoExecutesThisAction/>
        </fromActor>
        <byInstanceId instanceIdReference="instance.playerBuildingHandCardField"/>
      </selectField>
    </toField>

    <objectsToMove>
      <numberOfObjects>
        <selectValueFromAttribute>
          <selectAttributeFromActor>
            <fromWhichActor>
              <actorWhoExecutesThisAction/>
            </fromWhichActor>
            <attributeToSelect>
              <byInstanceId instanceIdReference="instance.numberOfCardsToKeep"/>
            </attributeToSelect>
          </selectAttributeFromActor>
        </selectValueFromAttribute>
      </numberOfObjects>
      <fromTop/>
    </objectsToMove>

  </moveObjectsFromFieldToField>

</fieldOperation>
<complexAction>
```

Dabei werden die `Buildings` des `buildingDrawField`-Feldes auf das `playerBuildingHandCardField` des **Actors** verschoben, welcher die **Action** `DrawCards` ausführt. Die Anzahl der Karten, die er nehmen kann, ist abhängig von seine `numberOfCardsToKeep`-**Attribute**. Über das `setFieldOwnerAsObjectOwner` definiert in `moveObjectsFromFieldToField` wird der Besitzer des `playerBuildingHandCardField` auch zum Besitzer der neuen `Buildings`.

C.1.12. Rule-Beispiel

Als Beispiel sollen einige **Rules** der Vorbedingung dienen, welche festlegt, ob ein Bauwerk errichtet werden darf. An dieser Stelle soll gezeigt werden, wie die unterschiedlichen Typen von **Rules** verwendet werden und wie auf den Kontext der **Action** zugegriffen werden kann.

Als erstes sollte die aktuelle **Phase** eine Charakter-Spielphase sein, da ansonsten keine Bauwerke errichtet werden dürfen.

```
<phaseRule>
  <currentPhaseMatchesType>
```

```
    <phaseTypeReference typeReference="type.CharacterPhase"/>
  </currentPhaseMatchesType>
</phaseRule>
```

Dabei wird der Typ der aktuellen **Phase** geprüft. Die `CharacterPhase` ist der abstrakte Typ, welcher durch die konkreten **Phases** der einzelnen Charaktere erweitert wird. Nun soll garantiert werden, dass der aktuelle **PhaseState** auch der Zustand ist, während der ein Spieler seine **Actions** normalerweise durchführen kann.

```
<phaseStateRule>
  <currentStateMatchesType>
    <phaseStateTypeReference typeReference="type.CharacterPhase.PlayState"/>
  </currentStateMatchesType>
</phaseStateRule>
```

Diese **Rule** wurde ebenfalls mithilfe des Typs definiert. Nun ist sichergestellt, dass sich das Spiel innerhalb des Zustands befindet, während dessen ein Bauwerk errichtet werden kann. Jetzt soll sichergestellt sein, dass das Bauwerk-**Object** auch einem Spieler gehört.

```
<objectRule>
  <isObjectOwnedByActor>
    <whichObject>
      <objectThisActionBelongsTo/>
    </whichObject>
  </isObjectOwnedByActor>
</objectRule>
```

An dieser Stelle wird geprüft, ob das **Object**, auf welches sich die **Action** bezieht, von einem **Actor** besessen wird. Über den Kontext der **Action** kann über das `objectThisActionBelongsTo`-Element auf das entsprechende **Object** zugegriffen werden.

In der `CharacterPhase` existiert ein Link auf die Instanz des **Objects** des derzeit aktiven Charakters. Wenn ein Charakter von einem Spieler ausgewählt wurde, gehört das Charakter-**Object** auch einem **Actor**. Nun soll geprüft werden, ob derjenige Besitzer des Charakter-**Objects** auch der Besitzer des Bauwerk-**Objects** und somit der Spieler mit dem aktiven Charakter ist.

```
<actorRule>
  <areActorsIdentical>
    <thisActor>
      <actorWhoExecutesThisAction/>
    </thisActor>

    <thatActor>
      <selectActor>
        <ownerFromObject>
```

```
          <selectObject>
            <fromCurrentPhase/>
            <byLinkId linkIdReference="link.characterOfPhase"/>
          </selectObject>
        </ownerFromObject>
      </selectActor>
    </thatActor>

  </areActorsIdentical>
</actorRule>
```

Um diese Bedingung zu formulieren, ist bereits eine der komplexeren **Rules** notwendig. Der **Action**-Kontext erlaubt es, dass in `thisActor` der `actorWhoExecutesThisAction`, also der Spieler, für den diese **Action** geprüft werden soll, ausgewählt werden kann. Dieser wird mit `thatActor` verglichen, welcher der Besitzer des aktuellen Charakters-**Objects** der aktuellen Charakter-**Phase** ist.

C.1.13. SimpleAction-Beispiel

Das Beispiel zeigt die `SimpleAction`, welche das Errichten eines Bauwerkes darstellt. Über das `executableBy` wird definiert, welcher Spieler diese **Action** überhaupt ausführen könnte. In `executableIfAnyApplies` kann über unterschiedliche `precondition` festgelegt werden, wann die **Action** ausgeführt werden kann.

```
<simpleAction name="ConstructBuilding" typeId="type.ConstructBuildingAction">

  <executableBy>
    <selectActor>
      <ownerFromObject>
        <objectThisActionBelongsTo/>
      </ownerFromObject>
    </selectActor>
  </executableBy>

  <executableIfAnyApplies>
    <precondition>

      <phaseRule>
        <currentPhaseMatchesType>
          <phaseTypeReference typeReference="type.CharacterPhase"/>
        </currentPhaseMatchesType>
      </phaseRule>
      <!-- Sonstige Rules -->

    </precondition>
  </executableIfAnyApplies>

  <operations>

    <!-- Reduce Gold-->
    <attributeOperation>
```

```
      <decreaseAttributeValue>

        <whichAttribute>
          <selectAttributeFromActor>
            <fromWhichActor>
              <actorWhoExecutesThisAction/>
            </fromWhichActor>
            <attributeToSelect>
              <byInstanceId instanceIdReference="instance.currentGold"/>
            </attributeToSelect>
          </selectAttributeFromActor>
        </whichAttribute>

        <decreaseBy>
          <selectValueFromAttribute>
            <selectAttributeFromObject>
              <fromWhichObject>
                <objectThisActionBelongsTo/>
              </fromWhichObject>
              <attributeToSelect>
                <byInstanceId instanceIdReference="instance.buildingCosts"/>
              </attributeToSelect>
            </selectAttributeFromObject>
          </selectValueFromAttribute>
        </decreaseBy>

      </decreaseAttributeValue>
    </attributeOperation>

    <!-- Move The Building to "constructed"-Field -->
    <objectOperation>

      <moveObjectToField>
       <!-- Sonstiges-->
      </moveObjectToField>
    </objectOperation>

  </operations>
</simpleAction>
```

Des Errichtens des Bauwerkes hat zwei Auswirkungen, die in Form von **Operations** definiert werden. Zuerst wird dem Spieler, der das Bauwerk errichtet, Gold entsprechend der Kosten des Bauwerkes abgezogen. Anschließend wird das Bauwerk-**Object** in das entsprechende **Field** des Spielers verschoben.

C.1.14. ComplexAction-Beispiel

Eine `complexAction` zeichnet sich durch einen eigenen Ablauf aus. Im Folgenden soll die **Action** `DrawCards` beispielhaft vorgestellt werden.

```
<complexAction name="DrawCards" typeId="type.DrawCardsAction">

  <executableBy>
   <!-- Sonstiges -->
  </executableBy>

  <executableIfAnyApplies>
    <precondition>
     <!-- Sonstiges -->
    </precondition>
  </executableIfAnyApplies>

  <actionStates>

    <actionState name="PlayerTakesAllCards" typeId="type.DrawCards.PlayerTakesAllCardsState">
      <actionStateTransitions>
        <!-- Sonstiges -->

      </actionStateTransitions>

      <operations>

        <fieldOperation>

          <moveObjectsFromFieldToField setFieldOwnerAsObjectOwner="true">
          <!-- Sonstiges -->
          </moveObjectsFromFieldToField>

        </fieldOperation>

      </operations>

    </actionState>

    <actionState name="PlayerChoosesCards" typeId="type.DrawCards.PlayerChoosesCards"
     isActionStillAbortable="false">

      <actionStateTransitions>
        <!-- Sonstiges -->
      </actionStateTransitions>

      <operations>
        <objectSelection>

          <numberOfElements>
           <!-- Sonstiges -->
          </numberOfElements>

          <actorWhoSelects>
            <actorWhoExecutesThisAction/>
          </actorWhoSelects>

          <objectsToSelectFrom>
           <!-- Sonstiges -->
          </objectsToSelectFrom>

          <followingOperations>
            <forSelectedObjects>
              <objectOperation>
                <moveObjectToField setFieldOwnerAsObjectOwner="true">
```

```
                <whichObject>
                  <objectSelectedByInteractiveOpeartion/>
                </whichObject>

                <toWhichField>
                  <selectField>
                    <fromActor>
                      <actorWhoExecutesThisAction/>
                    </fromActor>
                    <byInstanceId instanceIdReference="instance.playerBuildingHandCardField"/>
                  </selectField>
                </toWhichField>

              </moveObjectToField>
            </objectOperation>
          </forSelectedObjects>

          <forOtherObjects>
           <!-- Sonstiges -->
          </forOtherObjects>
        </followingOperations>
      </objectSelection>
    </operations>
  </actionState>

</actionStates>

</complexAction>
```

`DrawCards` besteht aus zwei verschiedenen **ActionStates**. Der **ActionState** `PlayerTakesAllCards` wird ausgeführt, wenn der Spieler alle gezogenen Bauwerk-**Objects** behalten dürfte. Dann werden automatische alle gezogenen Bauwerk-**Objects** in das `playerBuildingHandCardField` des Spielers verschoben.

Wenn er nicht alle Bauwerk-**Objects** behalten darf, wird hingegen der **ActionState** `PlayerChoosesCards` ausgeführt. Dort muss er anhand einer **InteractiveOperation** die **Objects** auswählen, welche anschließend in seinen `playerBuildingHandCardField` verschoben werden. Die nicht ausgewählten Bauwerke landen auf dem Ablagestapel.

C.1.15. Implication-Beispiel

Über die `IncreaseNumberOfBuildingsBuiltImplication` wird für jedes errichtete Bauwerk das **Attribute** `currentNumberOfBuildingsBuilt` des Spielers erhöht. Auf diese Weise kann an anderer Stelle innerhalb der Spielmechanik leicht abgefragt werden, wie viele Bauwerke ein Spieler zu einem Zeitpunkt errichtet hat. Dies ist unter anderem für die Siegbedingung wichtig.

```
<implication name="IncreaseNumberOfBuildingsBuiltImplication"
typeId="type.IncreaseNumberOfBuildingsBuiltImplication">

 <activeIfAnyApplies>
   <precondition>
     <objectRule>
       <isObjectOwnedByActor>
        <!-- Sonstiges -->
       </isObjectOwnedByActor>
     </objectRule>

     <objectRule>

       <isObjectInField>
        <!-- Sonstiges -->
       </isObjectInField>

     </objectRule>
   </precondition>
 </activeIfAnyApplies>

 <impact>

   <increaseAttributeValue priority="100">
     <whichAttribute>
       <selectAttributeFromActor>
         <fromWhichActor>
           <selectActor>
             <ownerFromObject>
               <objectThisImplicationBelongsTo/>
             </ownerFromObject>
           </selectActor>
         </fromWhichActor>
         <attributeToSelect>
           <byInstanceId instanceIdReference="instance.currentNumberOfBuildingsBuilt"/>
         </attributeToSelect>
       </selectAttributeFromActor>
     </whichAttribute>

     <increaseBy>
       <integerValue value="1"/>
     </increaseBy>

   </increaseAttributeValue>
 </impact>

</implication>
```

Die Priorität der Auswirkung liegt dabei bei 100 und wird entsprechend als letztes ausgeführt. Das `currentNumberOfBuildingsBuilt` dabei um 1 erhöht.

C.1.16. Chance-Beispiel

Bei OFuA wird kein **Chance**-Element genutzt. Das folgende Beispiel entspricht einen sechsseitigen Würfel. Da für die `chances` kein `amount` angegeben ist, ist dieser automatisch 1. Somit ist die Wahrscheinlichkeit gleichverteilt.

```
<integerChance name="W6" typeId="type.w6">
  <chance value="1"/>
  <chance value="2"/>
  <chance value="3"/>
  <chance value="4"/>
  <chance value="5"/>
  <chance value="6"/>
</integerChance>
```

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 4. Juli 2012	
Ort, Datum	Unterschrift