Inhalt

Abbildungsverzeichnis

Tabellenverzeichnis

Codebeispielsverzeichnis

Abkürzungsverzeichnis

AADL	Architecture Analysis & Design Language
ACP	Agile Certified Partitioner
ADL	Architecture Modeling Languages
ANTLR	ANother Tool for Language Recognition
API	Application Programm Interface
AST	Abstract Syntax Tree
Aufl.	Auflage
AWK	AWK (Unix)
BNF	Backus-Naur Form
DSL	Domain Specific Language
DSP	Domain Specific Prototype
EBNF	Extendet Backus-Naur Form
EMF	Eclipse Modeling Framework
FAQ	Frequently asked questions
GPL	General Purpose Language
GUI	graphical user interface
Hg.	Herausgeber
IDE	integrated development environment
IKVM	IKVM.NET
ISBN	International Standard Book Number
J.	Jahr
Kap.	Kapitel

LL	**L**eft to right, **L**eftmost derivation
M2C	Model to Code
M2M	Model to Model
MDSD	Model Driven Software Development
MOF	Meta Object Facility
MPS	Meta Programming System
MSDK	Modeling SDK
o.J.	Ohne Jahr
OMG	Object Management Group
PCD	Partial Class Description
PLE	Product Line Engineering
PM	Personenmonate
PMI	Project Management Institute
SDK	Software Development Kit
SED	stream editor (Unix)
SEQ	seq (Unix) [sequence]
TDD	Test Driven Development
UML	Unified Modeling Language
VS	Visual Studio
WPF	Windows Presentation Foundation
XML	Extensible Markup Language
XP	Extreme Programming
XSLT	Extensible Stylesheet Language Transformations

1 Einleitung

1.1 Motivation

In unserer Zeit existiert für beinahe jede Aufgabe eine Softwareimplementierung, sei es für die Verwaltung großer Datenmengen, für Bildbearbeitung, Textverarbeitung, Website-Darstellung oder Finanzbuchhaltung. Während des Entwicklungsprozesses eines konkreten Projektes wird häufig ein Experte für das jeweilig zu bearbeitende Fachgebiet, z.B. ein Informatiker, Steuerberater oder Designer, herangezogen. Dies bedeutet allerdings nicht, dass der Experte sein Fachwissen in der formalen Sprache des Projektes nach dessen Qualität-Standards sowie in der vorgegebenen Zeit auch implementieren kann. So wird gegebenenfalls, vor allem im Umfeld großer komplexer Enterprise Applikationen, auf Methoden des *Model Driven Software Development (MDSD)* wie der Bereitstellung einer *Domain Specific Language (DSL)* zurückgegriffen. Die vorliegende Arbeit eruiert deren praktische Grenzen sowie ihre Umsetzbarkeit im Umfeld kleinerer agiler Softwareprojekte.

1.2 Aufgabenstellung

Zuerst ist der aktuelle Stand von MDSD und insbesondere von DSL samt den zugehörigen Tools zu ermitteln. Der ermittelte Stand ist anschließend auf seine Brauchbarkeit im agilen Umfeld anhand einer Machbarkeitsstudie zu untersuchen.

Als Praxisbeispiel wird eine „*Incremental Game* Domain Specific Language for Prototyping“, auch als *Clicker Game* oder *Idle Game* bekannt, entwickelt. Ziel ist es, die Entwicklung und Umsetzung der Spielmechaniken vollständig auf den Gamedesigner zu verlagern, sodass dieser mithilfe der DSL ohne den Beistand eines Programmierers einen Spiel-Prototyp erstellen kann.

Die Umsetzung des Beispiels erfolgt mit den Tools:

- Eclipse DSL Tools
  Version: Neon Release (4.6.0)
  Build id: 20160613-1800
- Xtext 2.10.0

1.3 Abgrenzung

Die vorliegende Arbeit stellt keine Einstiegsliteratur, auch kein Nachschlagewerk für die Themen MDSD, Agile Softwareentwicklung oder DSL dar. Obwohl grundlegende Konzepte und Methoden behandelt werden, empfiehlt es sich, Fachliteratur wie (Thomas Stahl et al., 2007), (Voelter, 2013), (Tomas Bjorkholm und Jannika Bjorkholm, 2015) oder (Rubin, 2012) zurate zu ziehen.

Die wirtschaftlichen Vorzüge von DSLs werden anhand von Idealfällen betrachtet und setzen voraus, dass der Programmierer ausreichend Erfahrung bei der Planung und Umsetzung hat.

Auch können nicht alle sich auf den Markt befindlichen MDSD Technologien behandelt werden, weshalb der Hauptfokus auf Eclipse mit Xtext (Domain Specific Language) gelegt wird.

1.4 Kapitelübersicht

Die Bachelorarbeit beginnt mit einer Einführung in das Thema agile Softwareentwicklung und einigen agilen Vorgehensmodellen. Anschließend werden die grundlegenden Konzepte und Begriffe im Model Driven Software Development anhand des Buches „Modellgetriebene Softwareentwicklung“ (Thomas Stahl et al., 2007) erklärt. Das nächste Kapitel befasst sich mit den Grundlagen von Domain Specific Languages und einer Katalogisierung von Domain Specific Languages nach Markus Voelter.

Nach der Behandlung der wichtigsten Grundlagen für diese Arbeit beginnt ab Kapitel 5 „Aktuelle DSL Tools“ die Bearbeitung der Aufgabenstellung. Zu diesem Zweck werden zunächst mit den derzeit[1] aktuellen DSL Tools kleine Versuchsprojekte umgesetzt und die gewonnenen Erkenntnisse in Form eines Überblicks festgehalten. Die Ergebnisse über die einzelnen DSL Tools sind Entscheidungsgrundlage bei der Toolauswahl im nachfolgenden Kapitel „Pilotprojekt“.

Wie bereits der Titel des Kapitels vermuten lässt, wird darin ein Pilotprojekt mit MDSD unter dem Aspekt der agilen Softwareentwicklung geplant und durchgeführt. Zu diesem Zweck werden teilweise eigens dafür entwickelte Strukturen, Verfahren und Vorgehensweisen in der Theorie beschrieben und im Pilotprojekt praktisch umgesetzt.

[1] Mitte 2016

Die Analyse der gewonnenen Erkenntnisse und Ergebnisse des Pilotprojekts werden im Anschluss in einem eigenen Kapitel aufgearbeitet. Ziel dieses Kapitels ist es, die neu entwickelten Methoden genauer zu beschreiben, Theorien und mögliche Versuchsaufbauten aus den gewonnenen Erkenntnissen zu bilden sowie Hinweise und Zielvorgaben für weitere Forschungen in diese Richtung aufzuzählen.

Das letzte Kapitel besteht aus einer kurzen Reflexion darüber, über welche Erfahrungen ein Programmierer beim Arbeiten mit MDSD verfügen sollte. Den Abschluss bildet ein kurzes Schlusswort mit einem Ausblick in die mögliche Zukunft von MDSD.

2 Agile Softwareentwicklung

Dieses zweiteilige Kapitel soll einen kurzen Einblick in die agile Softwareentwicklung liefern. Dafür wird zuerst auf den historischen Entstehungsprozess und die heutige Grundlage aller modernen Agilen Vorgehensmodelle eingegangen. Anschließend werden ein paar der bekanntesten Modelle angesprochen und erläutert.

2.1 Geschichte und Grundlagen

In den späten 90er Jahren entwarf der Software Ingenieur Kent Beck das agile Vorgehensmodell *Extreme Programming (XP)* alternativ zum traditionellen Wasserfallmodell. Es sollte als das erste wirtschaftlich erfolgreiche Modell seiner Art bekannt werden (Beck und Andres, 2007, Kap. 17). Durch den Erfolg von XP und das Erscheinen des sogenannten Agilen Manifests (siehe Abbildung 1) erlangten in den darauffolgenden Jahren unterschiedlichste Vorgehensmodelle die Aufmerksamkeit der Öffentlichkeit.

Abbildung 1: Agiles Manifesto (Beck et al., 2001)

Alle agilen Vorgehensmodelle unterliegen den vier Grundwerten (siehe Abbildung 1) und den folgenden zwölf Prinzipien des Agilen Manifests (Beck et al., 2001):

Our highest priority is to satisfy the customer
through early and continuous delivery
of valuable software.

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Business people and developers must work together daily throughout the project.

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Working software is the primary measure of progress.

Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Continuous attention to technical excellence and good design enhances agility.

Simplicity--the art of maximizing the amount of work not done--is essential.

The best architectures, requirements, and designs emerge from self-organizing teams.

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

2.2 Vorgehensmodelle

Im Laufe der Zeit hat sich eine Vielzahl an agilen Methoden und zugehörigen Praktiken entwickelt. Beispielsweise legen Methoden wie XP ihren Fokus auf die Anwendung von agilen Praktiken wie Pair Programming oder Test Driven Development, während andere wie Scrum oder Kanban den vollständigen Workflow in einem Projekt beschreiben.

2.2.1 Scrum

Scrum ist das iterative, inkrementelle Entwickeln von Software auf der Basis von agilen Praktiken mit dem Ziel, den Kunden in jeder Iteration eine funktionierende Software zu liefern (Schwaber, o.J. [2002], S. 1).

Eine Iteration wird in Scrum als Sprint bezeichnet und hat eine Dauer von einem Kalendermonat (Schwaber und Sutherland, 2016, S. 8). In seiner grundlegenden Definition kennt es drei Rollen: Den **Product Owner**, der für die erfolgreiche Umsetzung des Produktes zuständig ist. Das **Entwicklerteam**, das für das Liefern des geplanten Inkrements verantwortlich ist, und den **Scrum-Master**. Seine Aufgaben sind die Kommunikationsförderung zwischen dem Product Owner und dem Entwicklerteam sowie die Beseitigung von dessen eventuellen Arbeitsbehinderungen.

2.2.2 Extreme Programming (XP)

XP oder auch Extreme Programming wird von seinem Erfinder Kent Beck als „[…] a style of software development focusing [sic] on excellent application of programming techniques, clear communication, and teamwork […]“ (Beck und Andres, 2007, Kap. 1) beschrieben. Um diese Werte in die Praxis umzusetzen und gleichzeitig auf die Wünsche des Kunden einzugehen, werden in XP 12 Praktiken, wie zum Beispiel Pair Programming oder Test Driven Development, angewandt.

In Gegensatz zu Scrum mit seinen Sprints hat XP mit seinen *Weekly Cycles* das Ziel, zusammen mit dem Kunden zu planen, welche User Stories in der kommenden Woche zu implementieren sind (Beck und Andres, 2007, Kap 7). XP kann auf die Zuweisung von festen Rollen verzichten; denn sein Ziel ist es, „[…] to have everyone contribute the best he has to offer […]“ (Beck und Andres, 2007, Kap. 10).

2.2.3 Kanban

Kanban ist eine der Inventar-Planung von Supermärkten nachempfundene Methode zur Produktionsprozessteuerung von Toyota (Toyota Motor Corporation, o.J.). Die Idee dahinter ist, Überschussproduktion und Zwischenlagerung von Autobauteilen zu minimieren. Dieser Gedanke ist die Grundlage der Kanban-Variante für die Softwareentwicklung. Ihr Erfinder David J. Anderson erkannte, dass zu viele einem Arbeitsschritt zugeordnete Aufgaben zur Verschlechterung der Produktqualität und zur Verzögerung bei der Lieferung führen (Tomas Bjorkholm und Jannika Bjorkholm, 2015, Kap. 1). Kanban verhindert dies, indem es die maximale Anzahl von Aufgaben pro Arbeitsschritt limitiert und auf diese Weise den Workflow optimiert.

Die Festlegung von Iterationsschritten ist in Kanban optional; der für einen Arbeitsschritt notwendigen Zeitaufwand wird aus früheren Erfahrungen mit dem Zeitmanagement ähnlicher Aufgaben errechnet (Tomas Bjorkholm und Jannika Bjorkholm, 2015, Kap. 4).

In seiner ursprünglichen Form kennt Kanban keine Rollen. Es wird jedoch empfohlen, bei einem Wechsel auf Kanban die Rollen aus dem bisherigen Vorgehensmodell zu übernehmen, um den Übergang auf Kanban zu erleichtern (Tomas Bjorkholm und Jannika Bjorkholm, 2015, Kap. 5). Einige Fachbücher[2] bieten auch auf Kanban zugeschnittene Rollenmodelle an.

Wegen der Unterschiede zu anderen agilen Methoden und teilweiser Verstöße gegen das Agile Manifest wird Kanban als Abkömmling von Lean meist als Alternative zu agilen Methoden angesehen. Es ist jedoch möglich, Scrum oder andere agile Methoden für einzelne Arbeitsschritte von Kanban anzuwenden oder sogar eine Hybridlösung wie Scrumban[3] zu verwenden.

[2] Siehe „Kanban in 30 days“ von Tomas Bjorkholm und Jannika Bjorkholm (2015)

[3] Siehe „The Scrumban [r]evolution“ von Reddy (2015)

3 Model Driven Software Development

Dieses Kapitel beschäftigt sich mit den Grundlagen von MDSD. Der erste Abschnitt erklärt, was unter dem Begriff Model Driven Software Development zu verstehen ist. Der zweite Abschnitt beschreibt die im deutschen Sprachraum gebräuchliche Terminologie von MDSD, wie sie 2007 von Thomas Stahl und Markus Völter in ihrem Buch „Modellgetriebene Softwareentwicklung“ (Thomas Stahl et al., 2007) definiert wird.

3.1 Definition

Model Driven Software Development beschreibt eine Ansammlung von Techniken, deren Ziel es ist, aus formalen Modellen automatisiert lauffähige Software zu erzeugen (Thomas Stahl et al., 2007, S. 11). Ein **formales Modell** ergibt sich aus einem Modell, z.B. Flowchart, User-Story oder UML-Diagramm, und beschreibt durch klare Regeln und Abgrenzungen vollständig einen bestimmten Aspekt einer Software. Anders als die meisten Modelle, dient dieses nicht nur der Dokumentation oder Spezifikation von manuellen Implementierungen, sondern hat zum Ziel, durch einen Generator oder Interpreter **lauffähige Software** zu erzeugen. Der Schritt vom Modell zur ausführbaren Software soll **automatisch** erfolgen. Das bedeutet, dass das Modell die Rolle des Quelltextes übernimmt und der daraus generierte Code ein temporäres Zwischenprodukt darstellt. Dieser Code darf anschließend nicht mehr manuell modifiziert werden, da es sich sonst um einen Wizard[4] handeln und das Modell seine Rolle als Quelltext verlieren würde. (Thomas Stahl et al., 2007, S. 11–13)

[4] Ein Wizard generiert nur einmal Code für eine anschließende manuelle Bearbeitung. Die Generierung von Code aus einem formalen Modell erfolgt bei jeder Änderung der konkreten Umsetzung des formalen Modells.

3.2 Begriffe und Basiskonzepte

Thomas Stahl und Markus Völter legen in dem Buch „Modellgetriebene Softwareentwicklung“ die Begriffe für MDSD und DSL im deutschen Sprachraum fest (Thomas Stahl et al., 2007, S. 28–34). Die folgenden Abschnitte greifen diese Begriffe (siehe Abbildung 2) auf und erklären sie in gekürzter Form.

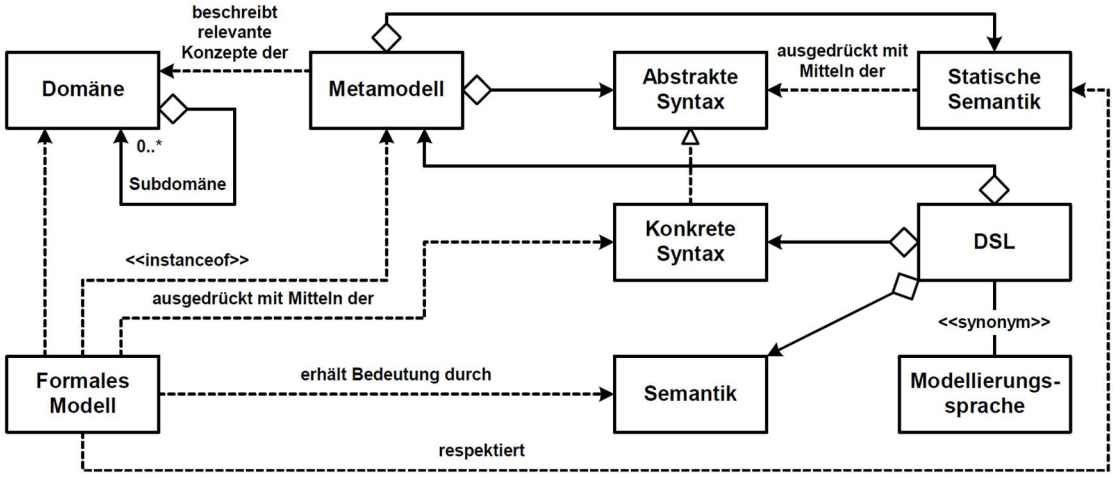

Abbildung 2: Begriffsbildung - Modellierung und DSLs (Thomas Stahl et al., 2007, S. 28)

3.2.1 Domäne

Eine Domäne ist im Kontext der MDSD eine sogenannte Problemdomäne. Diese beschreibt einen beschränkten Wissensbereich der zur Lösung einer Aufgabenstellung notwendig ist und nicht direkt mit Informatik an sich zu tun haben muss. Domänen können in mehr oder weniger hierarchischen Beziehungen zueinanderstehen, wobei hierarchisch untergeordnete Domänen als Subdomänen bezeichnet werden (Voelter, 2013, S. 60). Diese sind immer Teil der allgemeineren Eltern-Domäne und können wiederum in Subdomänen unterteilt werden.

In der nebenstehenden Grafik wird diese Domänenhierarchie dargestellt (siehe Abbildung 3). D wird als Abkürzung für Domäne benutzt, die tiefgestellten Zahlen stellen die hierarchische Position dar.

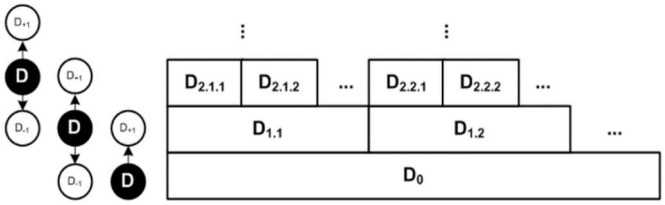

Abbildung 3: Domänen Hierarchie (Voelter, 2013)

D_0 steht für die Gesamtheit aller Domänen. Sie wird durch eine General Purpose Language implementiert. Eine Hierarchieebene darüber befinden sich die leicht eingeschränkten Domänen $D_{1.1}$, exemplarisch könnte dafür z.B. „Medizin“ eingesetzt werden. In der Ebene darüber wird die Domäne noch weiter eingeschränkt: Die für unser Fallbeispiel „Medizin“ passenden Subdomänen wären dann etwa „Human“- und „Veterinärmedizin“.

3.2.2 Metamodell

Ein Metamodell ist die formale Beschreibung einer Domäne und setzt voraus, dass das Verständnis des Programmierers dafür vorhanden und anwendbar ist. Es umfasst sowohl die abstrakte Syntax als auch die statische Semantik einer Sprache.

3.2.3 Abstrakte und konkrete Syntax

Die Metamodellelemente und deren Beziehungen untereinander werden als abstrakte Syntax bezeichnet. Die konkrete Syntax beschreibt indessen, wie die tatsächliche Darstellung des Quellcodes erfolgt.

Beispielsweise in C#: Die abstrakte Syntax von C# beschreibt, dass es Klassen gibt. Sie beschreibt auch, dass diese Klassen einen Inhalt haben, von anderen Klassen erben können, Interfaces implementieren können usw.

Die konkrete Syntax (siehe Codebeispiel 1) legt nun fest, dass die Klassen jeweils mit dem Schlüsselwort *class* definiert werden und anschließend der Name der Klasse steht. Nach dem Namen der Klasse wird für das Erben von einer Parent-Klasse deren Name mit vorangestelltem Doppelpunkt (":") eingetragen.

```
class A { }
class B : A { }
```

Codebeispiel 1: konkrete C# Syntax

Die Modelle werden in einer konkreten Syntax beschrieben, anschließend liest ein Parser oder Instantiator diese Beschreibungen ein, um die entsprechenden Instanzen der abstrakten Syntax-Klassen zu erzeugen. Die Darstellung der konkreten Syntax als Quellcode ist dabei unabhängig von der abstrakten Syntax und kann sowohl textuell (siehe Codebeispiel 1) als auch grafisch sein.

3.2.4 Statische Semantik

Die statische Semantik legt die Bedingungen (Constrains) fest, die für die Wohlgeformtheit eines Modells erfüllt werden müssen. Diese Bedingungen werden anhand der

abstrakten Syntax definiert und legen zum Beispiel fest, ob und wie Variablen deklariert werden müssen.[5]

3.2.5 Domänenspezifische Sprache

Eine domänenspezifische Sprache, im weiteren Verlauf der Arbeit als *Domain Specific Language (DSL)* bezeichnet, ist nichts anderes als eine Programmiersprache für eine spezielle Domäne. Sie besteht im Kern aus der abstrakten Syntax, aus Constrains und einer konkreten Syntax.[6]

3.2.6 Formale Modelle

Ein formales Modell ist ein Programm, das mithilfe einer DSL geschrieben ist, um das Verhalten eines daraus generierten Programms anhand der abstrakten Ebene zu beschreiben. Die Überführung in ausführbaren Code verläuft dabei automatisch und benötigt keine weitere manuelle Nachbearbeitung.

3.2.7 Metametamodell

Das Metamodell eines Metamodells wird als Metametamodell bezeichnet. Dabei sind die Elemente eines Metametamodells die Instanzen eines zugehörigen Metamodells. Sie definieren oft generische Dateiformate mit zugehörigen Parsern zum Modellaustausch sowie ein Framework für Metamodellklassen, mit deren Hilfe generische Parser instanziiert werden können. Beispiele dafür sind die Meta Object Facility (MOF) der OMG oder Ecore vom Eclipse Modeling Framework (EMF).

3.2.8 Transformation

Eine Transformation bezeichnet die Umwandlung von einem formalen Modell in etwas Anderes. Dabei kann es sich um eine Model2Model Transformation (M2M, *Model to Model*) oder Model2Plattform/Model2Code Transformation (M2C, *Model to Code*) handeln (siehe Abbildung 4).

---

[5] siehe „CSharp Language Specification“ Kapitel: „3.3 Declarations“ von Microsoft (2013)

[6] Dazu mehr in Kapitel 4: Domain Specific Language

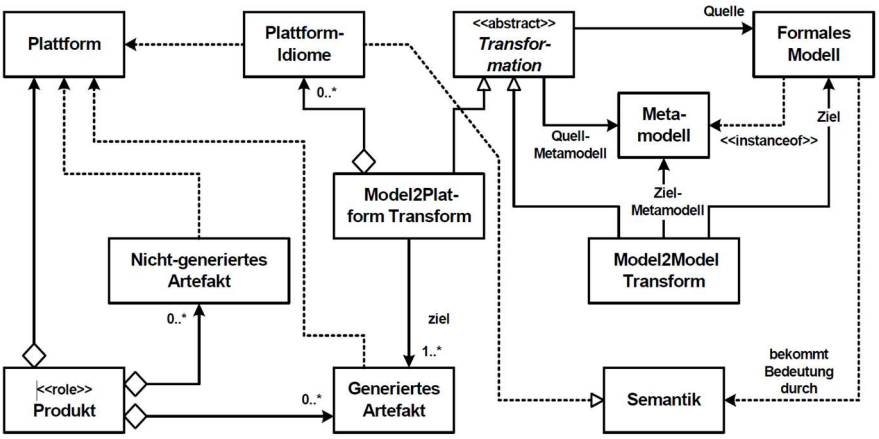

Abbildung 4: Begriffsbildung: Transformation (Thomas Stahl et al., 2007, S. 33)

M2M kann ein oder mehrere Modelle in ein Ziel-Modell transformieren, ist also Plattform-unabhängig, wobei es keinen Unterschied macht, ob das Metamodell des Ziels dasselbe ist wie das der Quell-Modelle, solange eine Transformation möglich ist.

M2C generiert Code, der für eine bestimmte Plattform benötigt wird; der Vorgang wird deshalb als Plattform-abhängig bezeichnet. Prinzipiell ist es möglich, für ein Modell endlos viele Model2Code-Transformationen zu definieren, solange es sich bei der Ziel-Sprache um eine General Purpose Language (GPL) wie Java, C# oder C++ handelt.

Nicht-generierte Artefakte werden im weiteren Verlauf der Arbeit als *Programm-Core* oder einfach *Core* bezeichnet.

3.2.9 Plattform

Eine Plattform hat die Aufgabe, die Umsetzung der Domäne zu „stützen“, das heißt, die Transformation von formalen Modellen möglichst zu vereinfachen. Die meisten Plattformen stellen Hilfsbibliotheken oder Ähnliches zur Verfügung, die als Fundament zur Codegenerierung dienen. Des Weiteren liefert eine Plattform Idiome, die vom Generator genutzt werden können.

3.2.10 Produkt

Das Generieren von Artefakten ist kein Selbstzweck; denn sie bilden zusammen mit der Plattform und dem handgeschriebenen Code das Produkt. Dieses ist entweder ein komplettes System oder nur eine Komponente.

4 Domain Specific Language

Die Entwicklung und Anwendung von Domain Specific Languages ist eine der ältesten und bekanntesten Praktiken aus dem Model Driven Software Development (s.u.). Deshalb wird im Pilotprojekt der Bachelorarbeit (siehe Kapitel 6 „Pilotprojekt“) als Vertreter für MDSD eine Domain Specific Language entwickelt.

Zu diesem Zweck gibt dieses Kapitel zu Beginn eine Definition mit der Abgrenzung von General Purpose Languages von Domain Specific Languages. Anschließend werden Domain Specific Languages anhand von Anwendungsfällen kategorisiert. Der letzte Teil dieses Kapitels befasst sich mit den Grundlagen, die zur Planung und Erstellung einer Programmiersprache, speziell einer Domain Specific Language, notwendig sind.

4.1 Definition und Abgrenzung

Früher waren Domain Specific Languages (DSL) in der Unix-Welt als Minilanguages bekannt (Raymon, 2003, Kap. 8). Minilanguages wurden als beschränkte, höhere Sprachen für eine bestimmte Domäne konzipiert mit dem Ziel, die Anzahl der Codezeilen zu reduzieren und dadurch die Übersichtlichkeit des Codes zu erhöhen. Weitere Beweggründe dafür waren, teuren Speicherplatz[7] zu sparen oder das Schreiben und anschließende Debuggen von Programmen zu vereinfachen.

Heutzutage werden aber sowohl DSLs, Minilanguages eingeschlossen, als auch General Purpose Languages (GPL) einfach als Programmiersprachen bezeichnet. Das liegt daran, dass die Grenzen zwischen DSL und GPL immer mehr verschwimmen. Die Charakteristika für eine „Ideale“ DSL oder GPL (siehe Abbildung 5) sind in jeder Sprache unterschiedlich stark ausgeprägt und ändern sich, wenn die Programmiersprache wächst. Beispielsweise war Perl als Ersatz für AWK und Unix Shell[8] gedacht (Richardson, 1999) [Weitere Quellen: (Wall, 1988) oder (Fish, o.J., Kap. 1.2)] und somit auch eine Minilanguage (DSL). Über die Jahre hinweg wurde Perl immer mehr erweitert und modifiziert bis irgendwann die Grenze zwischen DSL und GPL überschritten wurde.

[7] Beispielsweise kostete in den 80er Jahren eine Interne Festplatte mit 10MB ca. 895USD siehe Creative Microsales (1984)

[8] Beides Minilanguages für die Unix Kommandozeile.

	GPLs	DSLs
Domain	large and complex	smaller and well-defined
Language size	large	small
Turing completeness	always	often not
User-defined abstractions	sophisticated	limited
Execution	via intermediate GPL	native
Lifespan	years to decades	months to years (driven by context)
Designed by	guru or committee	a few engineers and domain experts
User community	large, anonymous and widespread	small, accessible and local
Evolution	slow, often standardized	fast-paced
Deprecation/incompatible changes	almost impossible	feasible

Abbildung 5: GPL versus DSL Charakteristika (Voelter, 2013, S. 31)

Abgrenzung DSL und GPL

Eine konkrete Unterscheidung zwischen DSL und GPL ist trotzdem möglich, wenn man die seinerzeitige Definition der Vorgänger-DSLs (Minilanguages) wie folgt aufgreift[9]:

Eine DSL ist eine Sprache für ein formales Modell, das zugunsten der besseren Nutzbarkeit und Produktivität auf eine spezifische Domäne beschränkt wurde. Die formalen Modelle von GPLs wie Java oder C# beschränken sich indessen nicht auf eine Domäne, damit sie (allerdings auf Kosten der leichten Nutzbarkeit und schnellen Produktivität) generell in jeder Domäne nutzbar sind.

Implementierungsmethoden

Es gibt zwei Möglichkeiten, eine DSL zu implementieren. Die erste Variante wird als *externe DSL* bezeichnet. Sie besitzt eine völlig eigene Syntax und wird meistens mithilfe von Interpretern und Generatoren in eine andere Programmiersprache transformiert.

Die andere Variante heißt *Interne DSL*. Diese wird in die zugrundeliegende dynamische GPL eingebettet und implementiert eine eigene formale Syntax für den von ihr abgedeckten Aspekt. Dabei ist der Übergang zwischen einer internen DSL und einem Framework meist fließend (Voelter, 2013, S. 50). Diese Hybriden aus DSL und Framework wurden im Jahr 2005 als FluentInterfaces, alias FluentAPI, (Fowler, 2005) durch Martin Fowler bekannt[10]. Die Idee dahinter ist, die Lesbarkeit von Code mithilfe einer natürlich anmutenden formalen Syntax zu verbessern. Dies geschieht meist mithilfe von Methoden wie „Method Chaining“. Ein Beispiel dafür ist die .net Bibliothek FluentAssertions[11]. Sie ermöglicht es,

[9] Vergleiche Raymon (2003, Kap. 8)

[10] Nicht zu verwechseln mit Method Chaining, Method Chaining ist nur eine Methode von FluentInterfaces. Fowler (2005)

[11] Siehe: http://www.fluentassertions.com/ (zuletzt besucht am 30.07.2016)

Tests für Objekte auf eine natürlich lesbare Art zu schreiben (siehe Codebeispiel 2) und verbessert so das Verständnis für das Ziel der Tests:

```
const string TEST = "Hello World!";
TEST.Should().StartWith("He").And.EndWith("d!").And.Contain("ll").And.HaveLength(12);
```

Codebeispiel 2: FluentAssertions Beispiel

4.2 DSL Kategorien

DSLs werden aus den unterschiedlichsten Gründen geschaffen, sei es Automatisierung, Prototyping, Cross-Funktionale Arbeitsweisen oder Qualitätssicherung. Markus Völter teilt in seinem Buch „DSL Engineering“ (Voelter, 2013, S. 47–50) diese DSLs anhand ihres Verwendungszwecks und ihrer Funktionsweise in unterschiedliche Kategorien ein. Diese Klassifikation ist wichtig, um die Funktion einer DSL in einer Domäne sicherzustellen. Beispielsweise darf durch das Klassifikationsverfahren eine DSL, die für das Beschreiben von Tests entwickelt wurde, nicht auf das Planen von Softwarearchitektur erweitert werden. Deshalb greift dieses Kapitel Völters Definitionen auf und erklärt ihre Klassifikation in einer zusammengefassten Form (Voelter, 2013, S. 47–50).

4.2.1 Utility DSL

Utility DSLs sind Werkzeuge für die Automatisierung von meist repetitiven, schwer dokumentierbaren oder komplexen Aufgaben in der plattformspezifischen Programmierung. Sie werden meist von kleineren Entwicklerteams entworfen, wobei die Entwicklung der Software nicht von der DSL abhängig ist. Ein Beispiel dafür ist die auf Xtext basierte DSL Jnario zum Beschreiben und automatisiertem Generieren von Unit Tests. Diese könnten auch regulär implementiert werden, das Beschreiben mit Jnario macht jedoch die Dokumentation der Tests überflüssig und vereinfacht die Lesbarkeit für Nicht-Programmierer.

4.2.2 Architecture DSL

Eines der größten Anwendungsgebiete für DSLs ist die Software Architektur. Dabei wird eine DSL genutzt, um die Architektur (Komponenten, Interfaces, Nachrichten, Abhängigkeiten, Prozesse, geteilte Ressourcen etc.) und Constrains (z.B. Timing, Ressourcen Verbrauch etc.) eines meist größeren Softwaresystems oder einer Plattform zu beschreiben. In Gegensatz zu *Architecture Modeling Languages (ADL)* wie *Unified Modeling Language (UML)* oder *Architecture Analysis & Design Language (AADL)* ist diese DSL speziell der Zielplattform oder Systemarchitektur angepasst, was ermöglicht, präzisere Analysen des Modells und besseren Code zu generieren. Die Besonderheit an der Architecture DSL Code Generierung besteht darin, dass meist Codeskelette generiert werden, die man anschließend mit Codeschnipseln füllt.

4.2.3 Full Technical DSL

Full Technical DSLs sind Architecture DSLs, die zusätzlich die gesamte Businesslogik eines Systems beschreiben und somit zur vollständigen Codegenerierung genutzt werden können. Sie sind meist modular aufgebaut, wobei jedes Sprachmodul dazu beiträgt, einen anderen Aspekt des darunterliegenden Systems zu beschreiben. Markus Völter bezeichnet sie als „Full Technical“, da diese DSLs für Entwickler, nicht für Domain Experten, ausgelegt sind.

4.2.4 Application Domain DSL

Sie beschreibt die Kern-Businesslogik eines Systems unabhängig von der technischen Implementation. Anders als die vorigen DSLs, sind Application Domain DSLs für die Nutzung von Experten ausgelegt, die nur wenig bis keine Erfahrungen im Programmieren haben. Die Anforderungen an eine Application Domain DSL sind für die Notation, den Ease of Use und Tool Support strenger als bei anderen DSLs. Neben diesen strengeren Anforderungen wird die Entwicklung zusätzlich dadurch erschwert, dass der Entwickler zuerst die Problemdomäne verstehen, dann strukturieren und nach der Umsetzung den Experten die Benutzung der DSL erklären muss.

4.2.5 Requirement Engineering DSL

Requirement Engineering DSLs sind stark mit den Application Domain DSLs verwandt, aber im Gegensatz zu diesen liegt der Fokus nicht auf der automatischen Codegenerierung, sondern auf der präzisen, prüfbaren und vollständigen Beschreibung von Requirements. Meist müssen diese DSLs in ein bestehendes System eingebunden oder anderweitig mit prosaisch anmutenden Texten verbunden werden, um sie mit den klassischen Methoden der Requirement Beschreibung zu kombinieren. Beispiele dafür sind pseudostrukturierte, an eine natürliche Sprache angenäherte DSLs, die für ihre Domain Entities formale Begriffe wie *should* oder *must* übernehmen.

4.2.6 Analyse DSL

Für die Beschreibung von Analysen, Tests oder Beweisführungen werden Analyse DSLs entwickelt. Sie nutzen die formelle Beschreibung einer Sachlage, um daraus Verifikationen für die Sicherheit, das Scheduling oder den Ressourcenverbrauch von Software abzuleiten. Die Generierung von Code ist dabei häufig mit inbegriffen, liegt aber nicht im Fokus bei der Entwicklung einer Analyse DSL. Vielmehr ist es das Ziel, ein System als Ganzes, mitsamt seinen physischen Aspekten wie Mechanik, Elektronik und Fluiddynamik, zu betrachten, was besonders bei komplexen, technischen Systemen oder System Engineering eine entscheidende Rolle spielt.

4.2.7 Product Line Engineering DSL

Product Line Engineering in der Softwaretechnik lässt sich in einem Satz wie folgt definieren: “Systems and Software Product Line Engineering, abbreviated as Product Line Engineering (or PLE for short), is defined as the engineering of a portfolio of related products using a shared set of engineering assets and an efficient means of production.“ (BigLever Software, o.J. [2013]). Diese *engineering assets* müssen so designt werden, dass sie sich an vorher festgelegten Variations-Punkten an das Produkt, in dem sie eingesetzt werden, anpassen. Diese Anpassbarkeit an sich ändernde Umstände wird als Variabilität bezeichnet. Je nach Art dieser Variabilität kann eine DSL sehr gut für die Darstellung und Beschreibung der einzelnen Softwarevarianten geeignet sein. Diese PLE DSLs werden dabei häufig für die Konfiguration, nicht für die kreative Schaffung der Lösung eines Problems genutzt.

4.3 Basiskonzepte zur Sprachentwicklung

Zweck der Programmiersprachen ist es, Berechnungen zu beschreiben, die sowohl von Menschen als auch von Maschinen lesbar sind. Deshalb ist es bei der Planung und Umsetzung von DSLs zwingend notwendig, dass einige grundlegende Konzepte und Regeln eingehalten werden. Weitere Konzepte, die zur Umsetzung einer DSL nötig sind, werden direkt in das praktische Beispiel eingebunden und erklärt.

4.3.1 Anweisungen

Eine Anweisung ist eines der grundlegenden Konzepte von vielen Programmiersprachen. Sie sind zumeist fix formulierte Vorschriften, die es ermöglichen, in Programmen Aktionen zu formulieren. Je nach Domäne und Anwendungsfall ist es erforderlich, Anweisungen in der eigenen DSL zu implementieren. Die folgende Tabelle (siehe Tabelle 1) nennt die wichtigsten Anweisungskonzepte und verweist entweder auf das zugehörige Kapitel oder gibt ein kurzes Codebeispiel mit Pseudocode:

Anweisung	Codebeispiel
Zuweisung	A := A + 5

Deklaration	// Explizite Deklaration
	LET INTEGER A := 5
	
	// Implizite Deklaration
	LET A := 5

Block	Siehe Kapitel 4.3.3
Bedingte Anweisungen	Siehe Kapitel 4.3.3
Schleifen	Siehe Kapitel 4.3.3
Unterprogramm Aufruf	Siehe Kapitel 4.3.3
Return Anweisung	Siehe Kapitel 4.3.3

Tabelle 1: Anweisungen einer Programmiersprache (Pseudocode, BASIC-like)

4.3.2 Ausdrücke

Ausdrücke sind eine Kombination aus Werten (Operanden), Operatoren und Unterprogrammen (s.u.), die ein Ergebnis in Form eines Operanden zurückgeben (siehe Codebeispiel 3). Die wichtigsten Mittel zur Beschreibung von Ausdrücken sind mathematische Operatoren (+, -, *, /, =), logische Operatoren (>, <, >=, <=, ==, !=, ==, &, &&, |, ||, ~, !) und Klammern („()“, „[]“, „{}“).

```
arithmetischer Ausdruck:
Papier:       5x²+2*4x+9²
Informatik:   5*POW(x, 2) + 2 * 4 * x + POW(9, 2)

aussagelogischer Ausdruck:
Annahme:      A = falsch, B = falsch
Wunsch:       A NAND B
Papier:       ¬(A ∧ B)
Informatik:   !(A&B)
```

Codebeispiel 3: Beispiel für Ausdrücke

Für die erfolgreiche Interpretation und Umsetzung des Ausdrucks benötigt eine Programmiersprache Operatorrangfolgen (z.B. „Punkt-vor-Stich“) und Operatorassoziativität. Die

Operatorassoziativität beschreibt, ob ein Ausdruck links- oder rechtsassoziativ interpretiert wird. Beispielsweise werden in C# die meisten binären Operatoren nach der mathematischen Operatorrangfolge und linksassoziativ interpretiert (Microsoft, 2013, S. 134–135). Somit wird der Ausdruck *a + b + c * d* vom Computer als *(a+b)+(c*d)* verstanden und umgesetzt.

4.3.3 Unterprogramme, Block- und Kontrollstrukturen

Blockstruktur oder kurz **Block** ist das wichtigste Charakteristikum für eine moderne Programmiersprache. Die Möglichkeit, mithilfe von Blöcken Anweisungen für ein Programm zu strukturieren, verbessert nicht nur die Lesbarkeit vom Programm Code, sondern ermöglicht es auch, durch Features wie Block Scope Variablen auf mehreren Ebenen zu definieren. Ein einfaches Beispiel für das Fehlen dieses Charakteristikums ist der folgende Pseudocode (siehe Codebeispiel 4):

```
100   LET A := 1                  // Startwert
110   LET J := 50                 // Wann sollen X Stellen ausgelassen werden
120   LET X := 11                 // Wie viele Stellen sollen ausgelassen werden
130   LET E := 100                // Zielwert
140   PRINT A                     // A ausgeben
150   IF A = J THEN GOTO 190      // Springe zu Zeile 190
160   A := A + 1
170   IF A <= E THEN GOTO 140     // Springe zu Zeile 140
180   END
190   A := A + X
200   GOTO 140                    // Springe zu Zeile 140
```

Ausgabe: 1, 2, 3, …, 49, 50, 61, 62, …, 99, 100

Codebeispiel 4: Unstrukturiertes Pseudocode Programm (BASIC-Like)[12]

In einem unstrukturierten Programm müssen die Anweisungen nicht nur untereinandergeschrieben werden, sondern das „Ende“ und der „Anfang“ eines „Anweisungsblocks“ werden durch die Anweisung „GOTO“ mit der entsprechenden Zeilennummer umgesetzt. Eine solche Arbeitsweise (siehe Codebeispiel 4) ermöglicht nur eine sehr beschränkte Codekomplexität, damit der Code übersichtlich bleibt.

Abhilfe schafft die Definition von Blöcken innerhalb der Listen von Anweisungen unter Zuhilfenahme von sogenannten **Kontrollstrukturen**. Sie werden durch eine spezielle Anweisung mit den nachfolgenden Schlüsselwörtern (BEGIN…END), mit geschweiften Klammern ({…}) oder gemeinsamer Einrückung des Textes (Tabstopps/Leerzeichen) gekenn-

[12] Analog zu Kemeny und Kurtz (1968, S. 53)

zeichnet und schließen meist eine Ansammlung von Anweisungen ein, die nur unter speziellen Bedingungen ein- oder mehrmals ausgeführt werden. Die Bedingungen für Kontrollstrukturen sind im Regelfall aussagenlogische Ausdrücke mit den möglichen Werten „wahr“ oder „falsch“.

Bei der Kontrollstruktur für einmalige Ausführungen handelt es sich um sogenannte **bedingte Anweisungen**. Sie legen fest, ob ein und welcher von einem oder mehreren Programmabschnitten an einer bestimmten Stelle im Programm ausgeführt wird. Die Kontrollstruktur wird in einer konkreten Syntax meist durch eine Formulierung wie diese dargestellt: IF (Bedingung) THEN {Anweisungsblock} ENDIF.

Für die mehrmalige Ausführung des gleichen Anweisungsblocks werden sog. **Schleifen** benutzt. Eine Schleife besteht aus einer Schleifenbedingung und einem Anweisungsblock. Der Anweisungsblock wird so lange wiederholt, wie die Schleifenbedingung wahr ist. Eine übliche Umsetzung in eine konkrete Syntax wäre: WHILE (Bedingung) DO {Anweisungsblock} ENDWHILE.

Im folgenden Beispiel (siehe Codebeispiel 5) wird deshalb das vorige Codebeispiel (siehe Codebeispiel 4) nochmals aufgenommen und mit modernen Kontrollstrukturen umgesetzt:

```
100  LET A := 1            // Startwert
110  LET J := 50           // Wann sollen X Stellen ausgelassen werden
120  LET X := 11           // Wie viele Stellen sollen ausgelassen werden
130  LET E := 100          // Zielwert
140  DO                    // Sprungziel der nächsten WHILE Bedingung ohne voriges DO
150      PRINT A           // A Ausgeben
160      IF A = J THEN
170          A := A + X
180      ELSE
190          A := A + 1
200      ENDIF
210  WHILE A <=  E         // Springe zum darüber liegenden DO ohne zugehörigen WHILE
220  END
```

Ausgabe: 1, 2, 3, …, 49, 50, 61, 62, …, 99, 100

Codebeispiel 5: Strukturiertes Pseudocode Programm (BASIC-Like)

Viele Programmiersprachen haben diese Strukturierung noch einen Schritt weiterentwickelt und begonnen, **Unterprogramme** zu definieren. Ein Unterprogramm, in einigen Programmiersprachen als Funktion, Methode oder Prozedur bekannt, ist ein isolierter Anweisungsblock mit der Möglichkeit, Variablen im lokalen Kontext zu deklarieren. Er wird so mithilfe einer Signatur, bestehend aus dem Funktionsnamen, den Funktionsparametern und manchmal dem Rückgabewert, dem restlichen Code zugänglich. Im Vergleich zum alten Beispiel (siehe Codebeispiel 5) wird der eigentliche Code für das Programm gekürzt, die Menge von Zeilen aber werden erhöht (siehe Codebeispiel 6).

```
100  LET A := 1            // Startwert
110  LET J := 50           // Wann sollen X Stellen ausgelassen werden
120  LET X := 11           // Wie viele Stellen sollen ausgelassen werden
130  LET E := 100          // Zielwert
140  DO                    // Sprungziel der nächsten WHILE Bedingung ohne voriges DO
150     PRINT A            // A Ausgeben
160     A := INKREMENT (A, J, X)
170  WHILE A <=  E         // Springe zum darüber liegenden DO ohne zugehörigen WHILE
180  END
190
200  DEF INT INKREMENT (INT AL, INT JL, INT XL)  // Signatur des Unterprogramms
210     IF AL = JL THEN
220        AL := AL + XL
230     ELSE
240        AL := AL + 1
250     ENDIF
260     RETURN AL          // Gebe den Wert von AL zurück
270  ENDDEF                // Ende der Definition des Unterprogramms
```

Ausgabe: 1, 2, 3, …, 49, 50, 61, 62, …, 99, 100

Codebeispiel 6: Strukturiertes Pseudocode Programm mit Unterprogramm (BASIC-Like)

4.3.4 Klassen

Viele der bekanntesten Programmiersprachen (C#, C++, Java, Python, JavaScript, TypeScript, Perl etc.) orientieren sich an dem Kernprogrammierparadigma der objektorientierten Programmierung (**OOP**), den **Klassen**. Diese sind Vorlagen für Objekte in einem Programm und beschreiben deren Attribute und Methoden. Zu ihren wichtigsten Eigenschaften zählt die Möglichkeit des Erbens, um hierarchische Strukturen untereinander aufzubauen und die Attribute und Methoden der Elternklasse zu übernehmen. Die Darstellung von Klassen erfolgt meist durch ein Schlüsselwort wie *class* mit anschließendem Namen der Klasse und darauffolgender Kontrollstruktur. Das Erben von einer anderen Klasse wird meist mit Schlüsselwörtern wie *extends* und nachfolgendem Elternnamen zwischen Name und Kontrollstruktur beschrieben (siehe Codebeispiel 7).

```
public class MyClass extends MyBaseClass
{
    private int Number;

    public void HelloWorld() {

    }
}
```

Codebeispiel 7: Klassen Definition

Die Umsetzung von Zugriffsmodifikatoren wie *public* oder *private* (siehe Codebeispiel 7) findet meist in objektorientierten Sprachen wie C++[13], Java[14], C#[15] oder Delphi[16] Verwendung. Für DSL sind diese aber optional, da beispielsweise ein Domänen-Experte ohne Programmierkenntnisse nicht auf die Datenkapselung oder andere Programmteile Zugriff haben sollte, die später bei der Generierung und anschließenden Nutzung Einfluss haben.

4.3.5 Backus-Naur Form

Die Syntax einer Programmiersprache kann mithilfe einer kontextfreien Grammatik in einer Metasyntax beschrieben werden. Eine der bekanntesten Metasyntaxen in der Informatik ist die **Backus-Naur Form (BNF)**. Die BNF kann als eine der ersten Domain Specific Languages bezeichnet werden; denn sie wurde konzipiert, um mit „[…] bequeme[n] und kurze[n] Ausdrücke[n] […] wirklich alle Verfahren der numerischen Rechentechnik ausdrücken zu können [...] [und dabei] […] möglichst wenig[e] Regeln der Satzlehre und Satzarten [zu] verwenden.“ (Backus, [1959], S. 125).

Die Festlegung einer kontextfreien Grammatik ist besonders bei der Verwendung von Parsern notwendig, da sie bestimmt, wie geparst werden muss (Fowler, 2011, Kap. 19.2).

BNFs unterscheiden zwischen terminalen und nicht-terminalen Symbolen. Terminale Symbole sind atomare Elemente und können nicht in weitere Symbole zerlegt werden. Nicht-terminale Symbole können so lange zerlegt werden, bis terminale Symbole erreicht werden. Im unteren Codebeispiel (siehe Codebeispiel 8) sind V, Z und R nicht-terminale Symbole und S ein terminales Symbol.

---

[13] Siehe Cpp Reference: http://en.cppreference.com/w/cpp/language/access (Letzter Besuch: 07.08.2016)

[14] Siehe Java Reference: https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html (Letzter Besuch: 07.08.2016)

[15] Siehe C# Reference: https://msdn.microsoft.com/en-us/library/wxh6fsc7.aspx (Letzter Besuch: 07.08.2016)

[16] Siehe Delphi Reference: http://docwiki.embarcadero.com/RADStudio/Seattle/en/Private,_Protected,_Public,_and_Published_Declarations (Letzter Besuch: 07.08.2016)

```
V  :=  Z   |'-' Z
Z  :=  R   | R ',' R
R  :=  S   | S R
S  :=  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

V = Eine Zahl mit Vorzeichen, Entry-Point
Z = Eine Zahl
R = Eine Reihe von Zahlensymbolen
S = Die Symbole für eine Zahl

Die Grammatik erlaubt die Beschreibung einer Zahl mit Vorzeichen und Nachkommastelle.

Nutzung:
Um eine Gleitkommazahl zu erstellen, wird mit dem Entry-Point der Sprache begonnen: **V**

Das Symbol wird anschließend in eines seiner beiden möglichen Untersymbole zerlegt: **Z**

Anschließend wird in mehreren Schritten das **Z** immer weiter zerlegt, bis nur noch Terminale Symbole '0' bis '9' übrig sind:

```
R     ,  R
S  R  ,  S  R
4  S  ,  9  S  R
4  3  ,  9  3  S
4  3  ,  9  3  1  = 43,931
```

Codebeispiel 8: Backus-Naur Form Beispiel[17]

4.3.6 Parse Tree und Abstract Syntax Tree

Sowohl der **Parse Tree** als auch der **Abstract Syntax Tree (AST)** sind objektorientierte Darstellungsformen für Source Code. Sie unterscheiden sich darin, dass der Parse Tree den gesamten Source Code mitsamt Leerzeichen und Zeilenumbrüchen darstellt, während der Abstract Syntax Tree von diesem nur die semantisch signifikanten Strukturen abbildet (siehe Anhang: Parser Tree und Abstract Syntax Tree).

Besonders der AST ist für die Programmierung von Generatoren und Parsern von DSLs wichtig; denn die reduzierte, aber vollständige Darstellung des Source Codes in Form von Objekten bietet einen stark vereinfachten und einheitlichen Zugang zur beschriebenen Geschäftslogik und dessen Interpretation.

[17] Analog zum Beispiel von Garshol (25. Juli 2014)

4.3.7 Sprachfragmente

Fragmentation (Voelter, 2013, S. 64) ist eher in DSLs als in GPLs anzutreffen. Sie bedeutet, dass eine Sprache aus mehreren Sprachfragmenten mit eigenem Sprachstamm bestehen kann, die trotz ihrer Unterschiede in Syntax- und Funktionalität zur gegenseitigen Vollständigkeit beitragen. Dies bietet sich besonders an, wenn eine Domäne sowohl strukturelle als auch formelbasierte Anteile hat und eine gemeinsame Darstellung für einen Domänenexperten verwirrend wäre.

Fragmentation darf aber nicht mit einer Abhängigkeit von anderen DSLs verwechselt werden. Ein bestehender Generator oder Interpreter für eine DSL sollte alle Sprachfragmente einer DSL übersetzen können, ohne auf eine fremde DSL-Implementation zurückgreifen zu müssen. Der Grund dafür ist, dass DSLs in Gegensatz zu GPLs eine sehr schnelle, häufig sprunghafte Evolution haben können. Sollte nun eine DSL von einer fremden DSL abhängen, muss diese auf jede Änderung der fremden DSL reagieren um Up-to-Date zu bleiben.

5 Aktuelle DSL Tools

Kapitel 5 bietet einen Überblick über die bekanntesten Tools des Jahres 2015[18]. Von jedem Tool werden die Funktionsweise, Besonderheiten und Transformationsvorgänge beschrieben. Anschließend werden die DSL Tools aus der Sicht einer kleinen Firma betrachtet. Das Tool soll dabei preislich günstig sein, aber für die Generierung von jeder Programmiersprache genutzt werden können. Um die einzelnen Tools, auch als Language Workbenches bezeichnet, zu evaluieren, wurde für jedes Tool ein kleines Testprojekt[19] umgesetzt und die Erkenntnisse darüber festgehalten.

5.1 JetBrains: Meta Programming System (MPS)

MPS von JetBrains ist eine kostenfreie Language Workbench. Sie wurde dafür konzipiert, vorhandene Sprachen zu erweitern oder DSLs zu definieren. Eine Besonderheit an DSLs, die mit MPS erstellt wurden, besteht darin, dass sie projektionsgetrieben sind. Das bedeutet, dass die Darstellung des Source Codes nur eine Projektion des AST ist. Obwohl der Nutzer der DSL die Änderungen anhand der konkreten Syntax sieht und vermeintlich damit ausführt, werden die Änderungen direkt im AST umgesetzt (siehe Abbildung 6). Dadurch ergibt sich nicht nur die Möglichkeit, textuelle oder grafische Syntaxen zu entwickeln, sondern auch hybride Modelle wie Tabellen oder grafische Formen mitten in den Text einzuführen. Ein weiterer Vorteil der Darstellung des Source-Codes durch AST ist, dass dadurch jedes Sprachelement eine AST-Node ist. Dies ermöglicht ein willkürliches Ineinanderschachteln von Auszeichnungs- und Programmiersprachen sowie deren Konzepten. So können beispielsweise SQL-Ausdrücke direkt in Java eingebunden und genutzt werden. Der Nachteil dieser Methode ist, dass der AST wegen seiner Komplexität

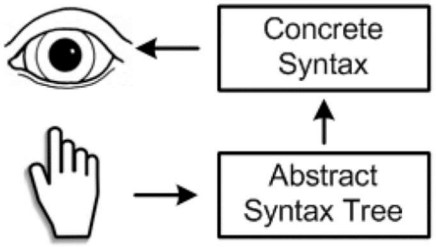

Abbildung 6: Konzept Projektion (Voelter, 2013, S. 68)

[18] ANTLR wird dabei ausgelassen; denn ANTLR ist keine Language Workbench, es ist ein Framework zur Parser Programmierung.

[19] Tutorials oder strukturierte Beispiele

nicht als Plaintext gespeichert wird und deshalb zu einer sehr starken Bindung an das Tool führt.

In MPS bauen DSLs auf anderen DSLs oder einer Base Language auf. Eine Base Language ist ein Mapping zwischen dem AST der DSL und dem plattformspezifischen Source Code. JetBrains stellt Base Languages für Java, XML und Ant. Weitere von der Community entwickelte Base Languages sind C (mbeddr), JavaScript (EcmaScript), R (MetaR) und Bash (NYoSh). Base Languages sind Teil des letzten Schrittes bei der Source Code Generierung von MPS: Dieser Schritt tritt nach mehreren M2M-Transformationen zwischen unterschiedlichen AST Modellen als Code-Generator auf (siehe Abbildung 7). Wegen der Unterschiede zwischen den M2M- und M2C-Transformationen wird die Arbeitsweise der MPS zweigeteilt, in den Generator-Teil mit den M2M-Transformationen und den Text Generator-Teil (kurz TextGen) für die M2C-Transformation.

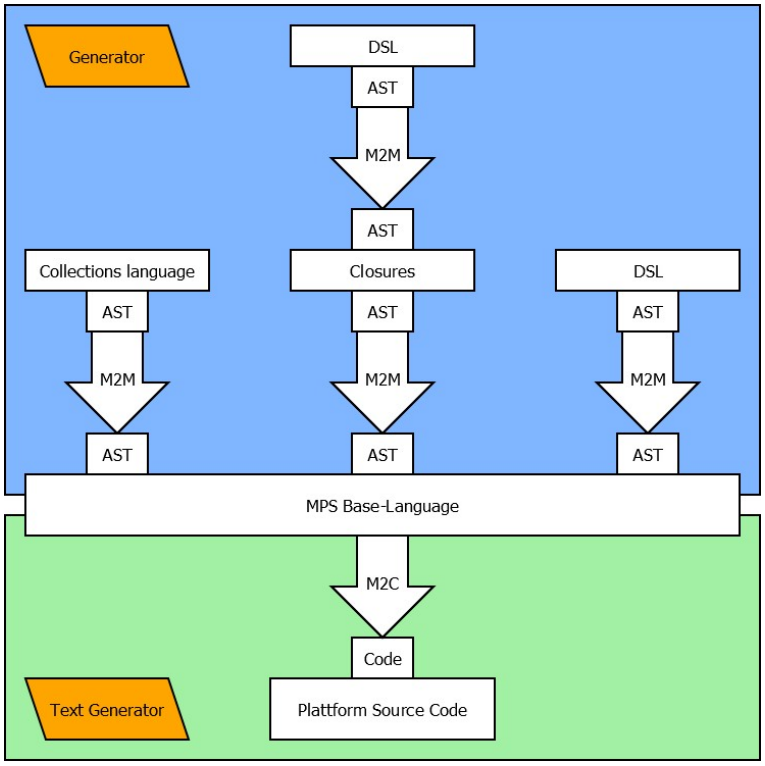

Abbildung 7: MPS Arbeitsweise[20]

Die Trennung hat den Vorteil, dass sich ein Programmierer für die Entwicklung einer DSL nicht zwingend mit dem TextGen und der darunterliegenden GPL befassen muss, führt aber gleichzeitig dazu, dass für jede Programmiersprache eine Base Language vorhanden sein oder erstellt werden muss. Beispielsweise fehlt in MPS die Base Language für

[20] Analog zu 0:07 in JetBrains (2015)

C#, was dazu führt, dass für C# Projekte parallel zur eigentlichen DSL eine Base Language entwickelt werden muss.

5.2 Visual Studio DSL Tools

Microsoft stellt für die Entwicklung von DSLs in Visual Studio (VS) das Modeling SDK (MSDK) bereit. Anders als bei MPS (s.o.) oder Xtext (s.u.) wird im MSDK zwischen zwei Versionen unterschieden, der *Authoring* und der *Deployment* Version. Die *Authoring* Version kann in jeder Visual Studio Version[21] installiert und benutzt werden. Sie ermöglicht es, vom MSDK erstellte DSLs zu nutzen, bietet aber keine Möglichkeit zu deren Manipulation oder der Entwicklung einer eigenen DSL. Indessen befähigt die *Deployment* Version einen Entwickler, diese beiden Features zu nutzen. Sie steht aber nur Besitzern von kostenpflichtigen[22] VS Versionen wie Professional oder Premium[23] zur Verfügung bzw. kann nur in eigens dafür entwickelten VS Shells bereitgestellt werden.

MSDK ist für die Entwicklung von visuellen DSLs ausgelegt. Es hat wie MPS einen projektionsgetriebenen Ansatz, trennt aber die Entwicklung der abstrakten und konkreten Syntax absolut von der Entwicklung des M2C-Transformators (Generator). Die abstrakte Syntax einer neuen DSL wird mit der Hilfe eines WYSIWYG[24]-Editors und dem zugehörigen *DSL Definition Diagram* strukturiert. Das DSL Definition Diagram baut auf DSL Diagram auf und hat deshalb eine starke Ähnlichkeit zu UML Diagrammen, weshalb es bei der Erstellung einer DSL gleichzeitig ihre abstrakte Syntax dokumentiert. Der Umfang der im DSL Definition Diagram erstellbaren Sprachelemente ist stark beschränkt und erlaubt nur Domain Klassen mit ihren Properties sowie Beziehungen zwischen den Klassen; terminale Symbole, Ausdrücke oder Anweisungen werden nicht angeboten. Aufgrund dieser Spezialisierungen fehlen dem MSDK essentielle Features für die Umsetzung einer textbasierten DSL, wie die Anbindung an den VS Texteditor mit Code-Validation, Quickfixes oder Scope-Testing. Die Folge davon ist, dass für die Umsetzung einer textbasierten DSL unter hohem Aufwand eine Lösung von Dritten wie ANTLR implementiert werden muss.

Die Projektion des AST erfolgt entweder mit DSL Diagram oder WPF. Die Darstellung der Sprachelemente im **DSL Diagram** kann im Editor der DSL mit *Shapes* (für Domain Klassen) oder *Connectors* (für Beziehungen) etc. beeinflusst werden. Die Einbindung von

---

[21] Siehe https://msdn.microsoft.com/en-us/library/bb126459.aspx (Zuletzt besucht am 17.08.2016)

[22] Deshalb wird die Visual Studio DSL Lösung grundsätzlich als Shareware betrachtet, da nicht die Nutzung von DSLs als Produkt, sondern die Vertriebsform der Language Workbench entscheidend ist.

[23] Siehe https://msdn.microsoft.com/en-us/library/bb126459.aspx (zuletzt besucht am 14.08.2016)

[24] **W**hat **Y**ou **S**ee **I**s **W**hat **Y**ou **G**et

ANTLR oder ähnlichen Lösungen ist bei dieser Darstellungsmethode besonders umständlich, da kein direkter Zugriff auf die Editorkomponenten vorliegt. **WPF** ist eine Alternative zu DSL Diagram und bietet mehr Möglichkeiten zur Individualisierung, einschließlich des Einbaus von editorbasierten Drittlösungen für textuelle DSLs, da die Oberfläche nicht vorgegeben ist und vollständig manuell programmiert werden muss.

Die Source-Code-Generierung wird im MSDK durch die manuelle Implementierung eines Generators verwirklicht. Der AST der DSL wird dabei direkt durch einen Text Generator verarbeitet, deshalb finden keine M2M-Transformationen vor dem Generator wie im MPS (s.o.) statt (siehe Abbildung 8). Die möglichen M2M- oder M2C-Transformationen[25] sind drei Technologien: XSLT, Domain-Specific API (Metametamodel) und T4 (Text Template Transformation Toolkit). Jede dieser Technologien kann für die Generierung von jeder beliebigen Auszeichnungs- und Programmiersprache genutzt werden.

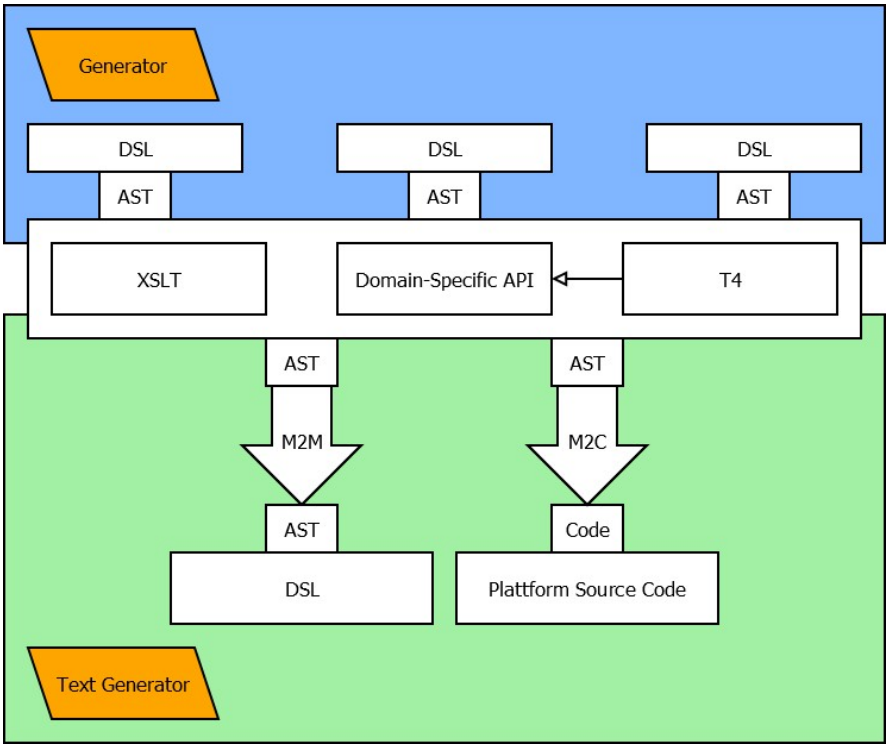

Abbildung 8: MSDK Arbeitsweise (Eigene Darstellung)

XSLT ist die schwächste Methode für die AST Transformation der DSL und eignet sich besser für M2M-Transformationen zwischen zwei Versionen des selben Modells als zwischen zwei Modellen unterschiedlicher DSLs oder M2C-Transformationen (Cook et al., 2007, Kap. 8). Die Verwendung von XSLT ist möglich, da der AST in einem XML-Format gespeichert wird und XSLT mit seinem identify-transform-Mechanismus ideal für die Aktualisierung des alten, ähnlichen ASTs zu einem neuen ist. Des Weiteren verliert es bei

[25] Die allgemeine Vorgehensweise der Generator Programmierung bezüglich komplexer Beziehungen etc. unterscheidet sich nicht wesentlich von der Xtext Herangehensweise (s.u.).

komplexeren oder einfachen Sachverhalten im Vergleich zu GPLs, sehr schnell an Übersichtlichkeit.

Die automatisch generierte **Domain-Specific API** kann zur direkten Manipulation des ASTs im Speicher und zur Programmierung eines Generators für M2M- oder M2C-Transformationen genutzt werden (Cook et al., 2007, Kap. 8). Die .Net Sprachen, die bei der Programmierung des Generators verwendet werden, können in ihrem vollen Umfang genutzt werden. Das erlaubt entweder eine direkte Umsetzung des Generators mit C# und String-Operationen oder eine indirekte mit beispielsweise CodeDOM. Diese erlaubt nämlich die Definition eines ASTs für eine Programmiersprache mit anschließender Transformation in eine der von CodeDOM unterstützen Sprachen. Aber "[…] that advantage is usually greatly outweighed by the fact that the code to create the abstract syntax tree is so far removed from the generated code that small modifications become relatively major pieces of work." (Cook et al., 2007, Kap. 8).

T4 *text templates* stellen eine weitere Nutzungsmöglichkeit der Domain-Specific API dar. Sie sind eine Kombination aus Textblöcken und Control Logik, durch welche Textfiles mit beliebigem Inhalt generiert werden können (Microsoft, o.J. [2015a]). Die Control Logik kann dabei mit C# 6.0 Features zwischen den statischen Textblöcken erstellt werden. Die Herangehensweise gleicht der Programmierung eines „normal" programmierten Generators (s.o.), hat aber den Vorteil, dass die T4 Files wegen der Ausfüllformular-ähnlichen Syntax zuerst in einer groben Form umgesetzt werden können (Cook et al., 2007, Kap. 8). Beispielsweise könnten alle Parameter von einem Sprachelement vorerst als *object* in der generierten Klasse angelegt und die richtigen Datentypen erst während der Verfeinerung des T4 *text templates* festgelegt werden.

5.3 Xtext

Xtext ist ein Open Source Framework von Eclipse für die Entwicklung von GPLs, Auszeichnungssprachen und DSLs. Wichtige Firmen bei der Weiterentwicklung und Verbesserung von Xtext sind die Eclipse Foundation, TypeFox und itemis AG. Im Gegensatz zu den DSL Tools MPS oder MSDK hat Xtext in der Java-Szene einen hohen Bekanntheitsgrad durch Xtend erreicht. Xtend ist eine statische, stark typisierte Programmiersprache, die nicht nur auf Java aufbaut, sondern auch zu Java-Code kompiliert wird, weshalb Xtend vollständig mit Java kompatibel bleibt und sogar parallel dazu genutzt werden kann.

Xtext ist eine parserbasierte Technologie (siehe Abbildung 9); sie erlaubt die Speicherung des Source Codes in normalen Textfiles ohne die Struktur des ASTs zu benötigen. Dies führt zu einer IDE-Unabhängigkeit seitens der DSL. Es bietet sich jedoch die Nutzung von Eclipse oder IntelliJ IDEA für eine Xtext-basierte DSL an; denn das Xtext Framework kann die Funktionen des Texteditors wie den Generator, die Code-Validation, das Scope Testing etc., mithilfe von Dependency Injections modifizieren und jeder Eigenheit der eigenen DSL präzise anpassen. Es ist auch möglich, textbasierte DSLs mit Frameworks für grafisches Editieren, wie Sirius oder Graphiti, zu kombinieren, um quasi eine grafische DSL zu erhalten; denn das Data Layer von Xtext ist wie bei Sirius und Graphiti das Eclipse Modeling Framework (EMF).

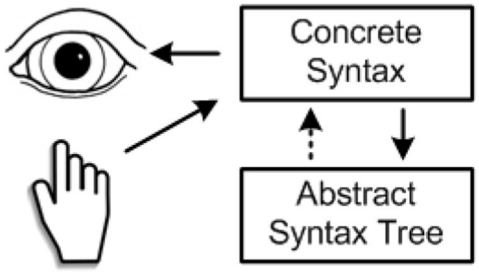

Abbildung 9: Parser Konzept (Voelter, 2013, S. 68)

Die Entwicklung der konkreten Syntax einer DSL erfolgt mit der *Grammar Language*, einer EBNF[26] ähnlichen DSL zur Beschreibung von textbasierten Sprachen. Die zugehörige Domain-Specific API wird automatisch als Java Code generiert. Dieser kann mit Java oder Xtend genutzt werden, um den Zugriff auf den AST offenzulegen, und damit die manuelle Programmierung von Linkern, Scope Testern, Code-Validators, Generatoren etc. (siehe Abbildung 10) zu erlauben.

[26] Extended Backus-Naur Form

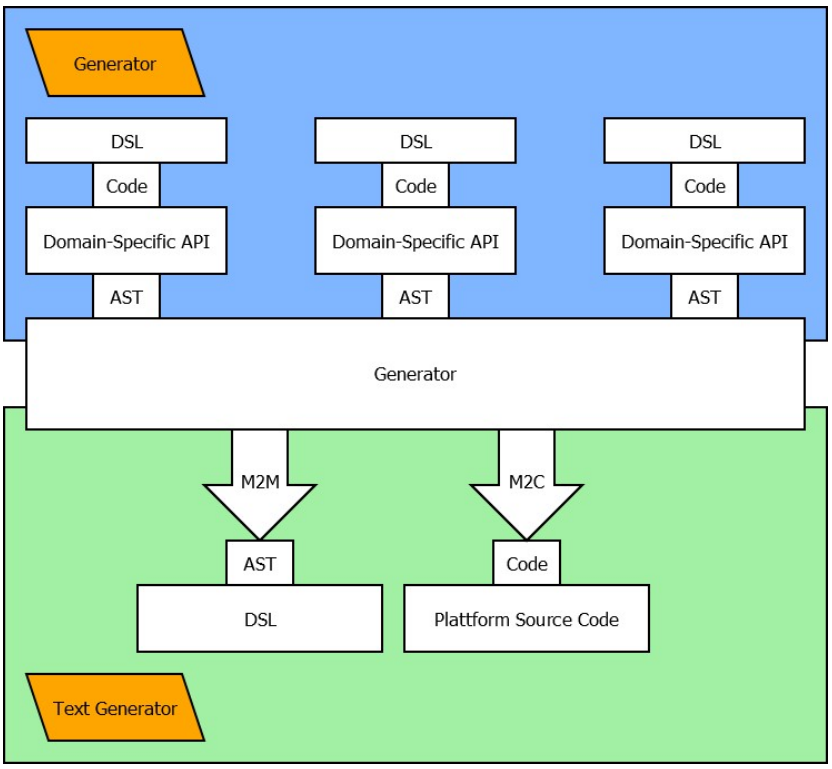

Abbildung 10: Xtext Arbeitsweise (Eigene Darstellung)

Auf die Details der Implementierung für Xtext wird hier verzichtet, da sie im Beispielprojekt von Kapitel 6 behandelt werden.

5.4 Vergleich und Fazit

MPS, MSDK und Xtext sind gut konzipierte, mächtige Tools für die Erstellung von DSLs, obschon keines von ihnen das „omnipotente DSL-Universaltool“ ist; denn alle drei haben ihre Stärken und Schwächen.

Visual Studio besticht mit **MSDK** durch seine einfache Handhabung, nämlich seine grafische DSL für die Syntaxentwicklung und das passgenaue Adaptieren der erstellten DSL mit eigener Oberfläche in jedem Visual Studio Editor. Durch die manuelle Implementierung des Generators steht es frei, ausnahmslos jede Sprache aus dem AST der DSL zu generieren. Als Design-Time Code Generator ist T4 eine Besonderheit unter allen genannten DSL Generator Technologien. Dennoch ist das MSDK wegen seiner Fixierung auf visuelle DSLs stark beschränkt, zumal die Beschreibung von Ausdrücken in der Praxis nicht schriftlich, sondern bildhaft erfolgen muss. Wegen der projektionsgetriebenen Darstellung sind mit MSDK entwickelte DSLs stark an Visual Studio gebunden. Ferner wird für das Entwickeln von DSLs pro Entwickler mindestens eine Visual Studio Professional Lizenz benötigt. Gewiss stellt der Preis dieser Visual Studio Version für finanzstarke Unternehmen keinen erwähnenswerten Nachteil bei der Wahl des DSL Tools dar, aber für Einmannbetriebe, Freiberufler, Privatpersonen oder weniger finanzstarke Unternehmen

kann der Preis von mehr als 600 € pro Lizenz[27] ein wichtiges Entscheidungskriterium sein. Abschließend ist zu MSDK zu sagen, dass es sich ausgezeichnet für Situationen eignet, in denen eine visuelle DSL ohne Geschäftslogik umgesetzt werden muss. Beispielsweise bei Architecture, Requirement oder PLE DSLs.

MPS ist ein kostenfreies Tool, das, anders als MSDK und Xtext, keinen manuell implementierten Generator für M2C Transformationen benötigt; denn dieser Schritt wird von Base Languages und deren Mapping auf eine Zielsprache, sei es eine GPL oder Auszeichnungssprache, übernommen. Das Mapping erübrigt neben der Generator Implementierung auch, dass der Programmierer die besagte Programmiersprache mitsamt ihren Eigenheiten kennen muss, solange eine entsprechende Base Language vorhanden ist. Dieser Vorteil geht selbstredend verloren, wenn es keine Base Language für die gewünschte Sprache gibt. In einem solchen Fall muss die Base Language plus deren Mapping parallel zur DSL entwickelt werden, was voraussetzt, dass der Entwickler über entsprechendes Wissen zur Domäne der DSL und deren Entwicklung braucht. Er muss auch die Syntax und die Konzepte der zu mappenden Zielsprache und die MPS Mapping-Sprache an sich kennen. Das Erlernen der Mapping-Sprache fällt bei den manuellen Generatorimplementierungen von MSDK und Xtext weg, da das Mapping auf GPLs automatisch erfolgt. Der projektionsgetriebene Ansatz von MPS führt, wie bei Visual Studio, zu einer sehr starken Bindung an den MPS Editor. MPS erlaubt aber im Gegensatz zu MSDK die Möglichkeit, eine hybride DSL mit textuellen und grafischen Elementen im selben Dokument zu schaffen. Zusammenfassend kann festgestellt werden, dass MPS ein solides DSL Tool mit der einzigartigen Möglichkeit ist, die konkrete Syntax losgelöst vom darunterliegenden AST zu behandeln. Unternehmen, deren Softwarelösungen mit Java umgesetzt werden und ein einfach zu lernendes und dennoch mächtiges DSL Tool mit idealer Darstellung brauchen, sollten darauf zurückgreifen. Wegen seines hybriden Charakters kann mit MPS ausnahmslos jede DSL Kategorie sehr gut umgesetzt werden.

Xtext ist ein bekanntes Open Source Framework für die Entwicklung von textbasierten Programmiersprachen aller Art in der Eclipse oder der IntelliJ IDEA IDE. Es hebt sich nicht nur wegen seiner Fixierung auf textbasierte Sprachen von den anderen ab, sondern verfügt auch über einen hohen Freiheitsgrad als bezeichnendes Charakteristikum. So ermöglicht es beispielsweise, dass durch den parserbasierten Ansatz die konkreten Implementierungen der DSL in überall editierbaren Textdateien gespeichert werden oder wegen der manuellen Implementierung und Anpassung aller IDE Features jede Eigenheit der DSL perfekt in die IDE eingepasst werden kann. Diese Freiheiten sind aber nur auf Kosten der Einfachheit von Xtext möglich. Anders als bei MPS oder MSDK, stehen weder visuelle Editoren noch einfache, noch automatische Implementationsmöglichkeiten von

[27] Neupreis für eine Standalone Version im Microsoft Store, ohne Updates 636,00 € inkl. MwSt. https://www.microsoftstore.com/store/msde/de_DE/pdp/Visual-Studio-Professional-2015-Deutsch/productID.323836900/?vid=323838000 (Stand 19.08.2016)

Funktionen wie Linking oder Scope Testing zu Verfügung. Schlussendlich kann Xtext als das DSL Tool mit der größten Omnipotenz, Unabhängigkeit und Lernhürde angesehen werden. Es kann wie MPS für jede DSL Kategorie genutzt werden. Hybride DSL-Darstellungen oder grafische DSLs können allerdings nur unter großem Aufwand implementiert werden.

Als **Fazit** lässt sich für ein kleineres Unternehmen die Aussage treffen, dass sich das MSDK wegen seiner starren Grenzen und hohen Anschaffungskosten als das ungeeignetste Tool darstellt. MPS ist zwar kostenlos und bietet mehr Freiheiten, zwingt aber den Nutzer, entweder eine Plattform mit unterstützter Base Language zu wählen oder diese selbst zu entwickeln. Xtext ist unter den dreien das flexibelste Tool; es bindet weder an eine IDE noch an eine Programmiersprache für die Code-Generierung. Selbst bei einem Plattformwechsel, beispielsweise von Java auf C#, kann in Xtext mit Leichtigkeit auf die Änderung reagiert werden. Deshalb ist Xtext für kleinere Unternehmen das Tool der Wahl, sofern nicht feststeht, dass Java die einzige Plattformsprache ist und bleiben wird. In diesem Fall sollte zu MPS gegriffen werden.

	MPS	**MSDK**	**Xtext**
DSL Sourcecode	AST	AST	Plaintext
Sprachsupport	3 (Nativ) + 4 (Community)	Keinen	Keinen
Code Generator	Base Language	XSLT/T4/ Domain-Specific API	Domain-Specific API
Domain-Specific API Sprache	Keine	.Net Sprache	Java
Kosten	Freeware	Shareware[28]	Freeware
DSL Darstellung	Text/Grafik/Hybrid	Grafik	Text
DSL-Darstellung Editor	Projektion	Projektion	Parser
IDE-Anpassung	Mittel	Niedrig	Hoch

[28] Deployment Version

Editor	MPS	Visual Studio	Eclipse
DSL Editorgebunden	Ja	Ja	Nein
Zielsprachen	Base Language	Alle	Alle
Versionierung	Ja	Ja	Ja

Tabelle 2: DSL Tool Tabellenvergleich (Eigene Arbeit)

6 Pilotprojekt

Folgende Kapitel behandeln das Vorgehen im Pilotprojekt „Incremental Game Prototyp DSL“. Die Ergebnisse des Pilotprojekts werden im darauffolgenden Kapitel dargestellt.

6.1 Fragestellungen und Hypothesen

Die zentrale Frage dieser Machbarkeitsstudie lautet zusammengefasst: „Ist die Nutzung von MDSD im kleineren agilen Umfeld möglich?“. Sie führt zu einer ersten Hypothese:

1. **Hypothese:** Die Nutzung von MDSD ist im kleineren agilen Umfeld möglich.

Die Analyse, ob einzelne Methoden des MDSD mit einer speziellen agilen Methode harmonieren, unterliegt starken Einschränkungen. Nicht zuletzt erschweren die Vielfalt von unterschiedlichen agilen Methoden und Herangehensweisen an ein Projekt die Analyse. Auch die Art des Projektes, die Arbeitssituation und die Fähigkeiten des Teams haben Einfluss darauf, ob die Nutzung von MDSD Techniken der Sache förderlich ist.

Für einen Selbstversuch mit einem Pilotprojekt werden gezielt einzelne agile Methoden und MDSD Praktiken ausgewählt, die gegebenenfalls auch von der kleinstmöglichen Gruppe im vollen Umfang umgesetzt werden können[29]. Als Organisationsstruktur für den agilen Part fiel die Wahl auf **Kanban** und die agile Methode **TDD** (Test Driven Development). TDD wird schon im Extreme Programming von K. Beck mit den Worten “In XP, when possible, tests are written in advance of implementation.“ (Beck und Andres, 2007, Kap. 13) beschrieben und findet auch in weiteren agilen Frameworks wie Scrum Verwendung. Als MDSD Praktik wurde die Entwicklung einer **DSL** gewählt, da die Praxis der Mini-Language-Entwicklung seit den 80ern[30] bis zum heutigen Zeitpunkt[31] durchgehend Verwendung findet.

Dies führt zu den folgenden weiteren Hypothesen:

2. **Hypothese:** Wenn in einem mit Kanban organisierten Softwareprojekt nach der Methode des TDDs entwickelt wird, dann vereinfacht (bei bestimmten Aufgaben)

[29] Das Pilotprojekt wird wegen des kleinen Zeitfensters ein Ein-Mann-Projekt.

[30] Siehe AWK (Scriptsprache – Linux)

[31] Stand: 10.09.2016

die Entwicklung einer DSL die Arbeit des Teams durch verbesserte Aufgabenverteilung und die Trennung der Domäne von der Plattform-Implementation bei der Entwicklung und Wartung.

3. **Hypothese:** Unternehmen mit kleineren agil arbeitenden Teams profitieren auf lange Sicht von der DSL-Entwicklung, wenn die Nutzung der DSLs eine gewisse Häufigkeit überschreitet oder eine hohe Flexibilität hinsichtlich der Kundenwünsche erforderlich ist.

6.2 Zielsetzung und Versuchsaufbau

Das Pilotprojekt wird im Rahmen eines Selbstversuches mit den ausgewählten agilen Techniken Kanban und TDD umgesetzt. Der Selbstversuch hat zum Ziel, eine Xtext basierte DSL zur Erstellung von Incremental Game Prototypen (Titel des Projekts: „Incremental Game Prototyp DSL“) zu entwickeln. Um einen besonderen Kundenwunsch zu simulieren, wird als Zielplattform Windows mit einem Visual Studio C# Projekt als Program Core gewählt. Die besondere Herausforderung dabei ist, damit umzugehen, dass C# als Sprache und Visual Studio Projekt-Strukturen bis jetzt noch nicht von Eclipse unterstützt werden[32]. Als Zeitlimit für die Umsetzung der „Incremental Game Prototyp DSL“ werden 180 Stunden oder 1,125 Personenmonate festgesetzt. **Ziel** ist, neben der Verifikation oder Falsifikation der Hypothesen, mit dem Gelingen des Pilotprojekts zu beweisen, dass die Erforschung und Anwendung von MDSD-Methoden in kleinen, agilen Gruppen möglich und lohnenswert ist. Im Folgenden werden die einzelnen zu verwendenden Methoden und Tools in Form einer Liste aufgezählt:

Tools:

- Eclipse DSL Tools
  Version: Neon Release (4.6.0)
  Build id: 20160613-1800
- Microsoft Visual Studio Enterprise 2015
  Version: 14.0.25424.00 Update 3
- Visual Studio Code
  Version: 1.5.2
- JetBrains ReSharper Ultimate 2016.1.1
- Dia
  Version: 0.97.2
  Ein Programm zum Zeichnen von Diagrammen

[32] Stand: Eclipse Neon (10.09.2016)

- Beyond Compare 4.1.8

Frameworks:

- Xtext 2.10.0
- .Net Framework 4.6.1
- xUnit 2.1.0
- Moq 4.5.21

Programmiersprachen:

- C# 6
- Java 1.8
- Xtend 2.9.0

Versionierung:

- Visual Studio Team Services

Agile Techniken:

- Kanban (Organisation)
- Test Driven Development (Methode)

Gruppengröße: 1

Zeitraum: 180h (1,125 Personenmonate)

6.3 Incremental Game

Da der Autor auf Erfahrungen im Game Development zurückgreifen kann, wurde als Beispieldomäne ein Incremental Game ausgewählt. Bevor die Umsetzung des Pilotprojektes beginnt, wird der ersten Phase der DSL Entwicklung (siehe Kapitel 6.5.1 Domain Research) vorgegriffen und anhand einer kurzen Definition beschrieben, was ein Incremental Game ausmacht:

> Ein Incremental Game ist ein Endlosspiel mit dem Ziel, durch eine simple Interaktion des Users, beispielsweise einen Klick, In-Game Währung zu erhalten. Die Währung kann für den Kauf oder die Weiterentwicklung von Objekten genutzt werden, um in kürzerer Zeit mehr von dieser Währung zu erzeugen, und zwar mit dem Bestreben, automatisiert und möglichst schnell große Mengen der Währung zu generieren, ohne selbst aktiv werden zu müssen. Durch weitere Features, wie das absichtliche Reseten des Spiels, um irgendwelche Boni zu erhalten, oder Qicktime-Events wird das Spielvergnügen zusätzlich gesteigert.

Die wichtigste und in jedem Incremental Game vorkommende Mechanik ist mindestens eine **In-Game Währung,** die zu Beginn durch eine **simple Interaktion des Spielers**, meist einen simplen Mausklick oder Ähnliches auf eine bestimmte Fläche, und später durch **Generator-Objekte** vermehrt werden kann. Die Generator-Objekte beschleunigen entweder das Erzeugungsintervall oder erhöhen die generierte Menge der Währung. Der Spieler kann Generator-Objekte meist mit der selbstverdienten In-Game Währung kaufen. **Entwicklungen** inklusive **Erfolge** haben entweder denselben Einfluss wie Generator-Objekte oder beeinflussen die Einkaufspreise.

Als Thema für das Pilotprojekt wurde vom Autor die Entwicklung eines Computerspiels aus zwei Gründen gewählt. Erstens hat er seit Studienbeginn häufig an interdisziplinären Projekten in Cross-funktionalen Teams, bestehend aus Programmierern, 2D/3D- Künstlern, Musikern und Gamedesignern, gearbeitet und dementsprechend viele Erfahrungen bei der Entwicklung von Entwicklertools für Gamedesigner und von Spielen gesammelt. Zweitens ist ein Incremental Game durch seine übersichtliche und weniger umfangreiche Spiel-Mechanik[33] ein ideales Pilotprojekt für einen Selbstversuch.

6.4 Hybridprojekt

Für die Entwicklung eines Hybridprojekts (in dem sowohl Eclipse als auch Visual Studio benutzt werden), waren zwei Gründe maßgeblich: Einmal die Simulation eines besonderen Kundenwunsches (siehe Kapitel 6.2 „Zielsetzung und Versuchsaufbau“) und zum anderen die Einsicht, dass die Entwicklung von Xtext DSLs für Visual Studio Projekte bislang stark vernachlässigt wird.
Es gibt zwar einen Eintrag auf einen Blog[34] von 2015, in dem berichtet wird, dass ein Team daran arbeitet, Xtext mit dem IKVM nach Visual Studio zu portieren, aber seitdem wurde dazu nichts weiter bekannt.
Ein zweites Projekt, nämlich die Einbindung von Xtext in Visual Studio Code oder Eclipse Che durch das Language Server Protocol[35] (Xtext-Team, 2016) befindet sich derzeit[36] noch in einer sehr frühen Phase. Ob damit überhaupt Xtext DSLs mit IDE Support für Visual Studio Projekte erstellt werden können, ist zudem bis dato ungewiss.

---

[33] Die Geschäftslogik eines Spiels.

[34] „[…] and we are in contact with a group that has ported Xtext to Visual Studio through IKVM.“ Efftinge (2015)

[35] Siehe Language Server Protocol: https://github.com/Microsoft/language-server-protocol (Zuletzt Besucht: 24.09.2016)

[36] Stand: Mitte 2016

Deshalb muss vor der Entwicklung der „Incremental Game Prototyp DSL“ ein Hybrid- „Eclipse zu Visual Studio“- Projekt ausgearbeitet werden. Dafür ist zum einen eine passende Ordnerstruktur notwendig, zum anderen müssen die in Eclipse erstellten Dateien im Visual Studio Projekt referenziert und geladen werden.

6.4.1 Ordnerstruktur

Der erste Schritt ist der Entwurf einer Ordnerstruktur mit Projektkonfigurationsdateien (siehe Abbildung 11 oder Anhang Hybrid Projekt Folder Tree).

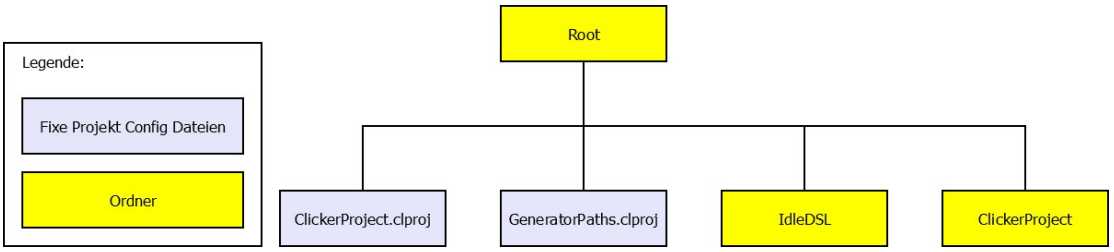

Abbildung 11: Ordnerstruktur (Ausschnitt) (Eigene Darstellung)

Das Rootverzeichnis des Projekts (siehe Abbildung 11) hat zwei Projektkonfigurationsdateien *ClickerProject.clproj* und *GeneratorPaths.clproj.* Beide Dateien beschreiben die wichtigsten relativen Pfade innerhalb des Projekts und liefern die wichtigsten Informationen für das Funktionieren des Hybridprojekts (siehe Anhang „Project Files Aufbau“). Neben den beiden Dateien liegen im Rootverzeichnis zwei Ordner. Der *IdleDSL* Ordner enthält Eclipse Workspaces bzw. Eclipse Projekte, welche die Dateien für das Visual Studio Projekt beinhalten, erstellen oder generieren. Der *ClickerProject* Ordner enthält indessen Visual Studio Solutions oder Visual Studio Projekte[37], in welche die oben genannten Dateien eingebunden werden sollen.

6.4.2 ProposalDaemon

Ein wichtiger Bestandteil des Hybridprojekts ist der *ProposalDaemon*[38]. Er bindet C#-Source-Files, die außerhalb eines bestimmten Visual Studio Projekts erstellt wurden, in eben dieses ein.

[37] Nicht zu verwechseln mit einer Visual Studio Solution; denn eine Visual Studio Solution ist eine Ansammlung von Visual Studio Projekten.

[38] Der *ProposalDaemon* wird als *Daemon* bezeichnet, da er keiner direkten Userkontrolle unterliegt, sondern nur gestartet werden kann, um anschließend selbstständig zu agieren.

6.4.2.1 Visual Studio Projekt Datei

Die Visual Studio Projektdatei, manchmal auch als cs-Projektdatei[39] bezeichnet, ist eigentlich eine MSBuild Project File. Microsoft definiert das MSBuild Project File Format wie folgt: “MSBuild uses an XML-based project file format that's straightforward and extensible. The MSBuild project file format lets developers describe the items that are to be built, and also how they are to be built for different operating systems and configurations. In addition, the project file format lets developers author reusable build rules that can be factored into separate files so that builds can be performed consistently across different projects in the product.“ (Microsoft, o.J. [2015b]). Eine wichtige Beobachtung bei den cs-Projektdateien ist, dass von einem Visual Studio geladene Projektdateien nicht nur editiert werden können, sondern die besagte Visual Studio Instanz Änderungen an den geladenen Projektdateien erkennt, um sie bei Bedarf neu zu laden. Dieses dem Observer-Pattern ähnelnde Verhalten wird vom *ProposalDaemon* genutzt, um selbst bei einem geöffneten Projekt Änderungen vorzunehmen, ohne Visual Studio neustarten zu müssen.

6.4.2.2 Programmablauf

Der *ProposalDaemon* ist ein Konsolen-Programm und kann nur gestartet werden, wenn als Kommandozeilenargument genau ein vollständiger Pfad zu einem Ordner übergeben wird. Dieser Ordner und dessen Unterordner werden nach sogenannten Proposal-Files durchsucht.

Eine Proposal-File mit der Dateiendung „.prop“ hat ein XML-basiertes Datei-Format zur Beschreibung von in ein MSBuild-Projekt einzubindenden Klassen. Eine Proposal File besteht aus einem Rootelement mit mindestens drei Kind Elementen (siehe Abbildung 12)[40].

1. *AbsolutePath*: Dem absoluten Pfad zum CSharp Ordner;
2. *BaseDir*: Dem letzten Glied des absoluten Pfads zum CSharp Ordner;
3. *GenericPaths*: Dem absoluten Pfad zum Ordner mit den Generischen Dateien.

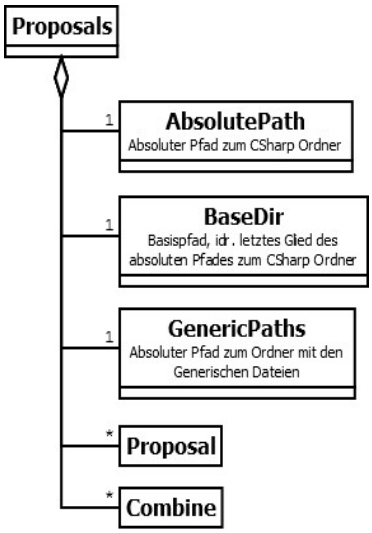

Abbildung 12: Aufbau Proposal File (Eigene Darstellung)

[39] Wegen seiner Dateiendung *.csproj*

[40] Die Struktur der XML basierten Datei wird anhand eines UML Diagramms dargestellt. Attribute der einzelnen Knoten werden wie Klassen-Attribute dargestellt. Ein leeres Attribute-Feld bedeutet, dass der Knoten keine Attribute hat. Sollte das Attribut-Feld fehlen wird der Knoten als eigenes Element in einer anderen UML dargestellt.

Ein Proposal- oder Combine-Element[41] beschreibt alle für eine SourceFile notwendigen Informationen zum Einbinden in eine cs-Projektfile mithilfe von markierten *ItemGroups* (siehe Codebeispiel 9):

- Der Klasse der einzubindenden Datei;
- dem relativen Pfad im Visual Studio Projekt;
- dem Markernamen der zugehörigen *ItemGroup*.

```
<ItemGroup>
  <!--MarkerComment:[KlassenName]-->
  <Compile Include="[Pfad zur Source File]">
   <Link>[Darstellung in der Projekthierarchie]</Link>
   <AutoGen>[Optional: true oder false]</AutoGen>
  </Compile>
</ItemGroup>
```

Codebeispiel 9: Eine markierte ItemGroup

Die Markierung (siehe Codebeispiel 9) erfolgt durch einen speziell formatierten Kommentar, da das Einbringen von eigenen Tags in eine cs-Projektfile nicht vorgesehen ist.

Nach dem erfolgreichen Programm-Start werden zuerst alle Combine-Elemente dazu genutzt, *Partial Class Descriptions* (PCD) in C#-Source-Files und zugehörige Proposal-Files zu transformieren. Die Proposal-Elemente der Proposal-Files werden anschließend direkt in Compile-Elemente umgewandelt und dem ItemGroup-Element mit den passenden Markerkommentar zugeordnet. Dabei wird auch darauf geachtet, dass Zwillinge unter den neuen Einträgen vermieden werden. Danach werden die Kind-Elemente aller markierten ItemGroup-Elemente auf tote Referenzen überprüft, indem die Existenz der beschriebenen SourceFile sichergestellt wird. Als letzter Schritt wird die überarbeitete cs-Projektfile gespeichert.

6.4.2.3 Event wait handles

Da der Daemon in Eclipse mithilfe von Powershell oder der Kommandozeile gestartet wird, kann Eclipse nicht sicherstellen, dass der *ProposalDaemon*-Prozess nur einmal zur selben Zeit ausgeführt wird. Deshalb werden die einzelnen Prozesse über globale, prozessübergreifende event wait handles synchronisiert. Diese erlauben es, mehrere *ProposalDaemon* Prozesse, nach ihrer Startzeit sortiert, hintereinander durchzuführen.

[41] Siehe Anhang „Proposal Aufbau“

6.5 Domain Specific Prototype Methode (DSL Entwicklung)

Für die inkrementelle Entwicklung von Xtext-DSLs mit TDD hat sich während des Selbstversuchs das folgende Vorgehen mit vier Phasen als erfolgreich herausgestellt. Bevor die einzelnen Schritte anhand der „Incremental Game Prototyp DSL“-Entwicklung beschrieben werden, wird der Ablauf der Methode mit einem Aktivitätsdiagramm grafisch dargestellt (siehe Abbildung 13).

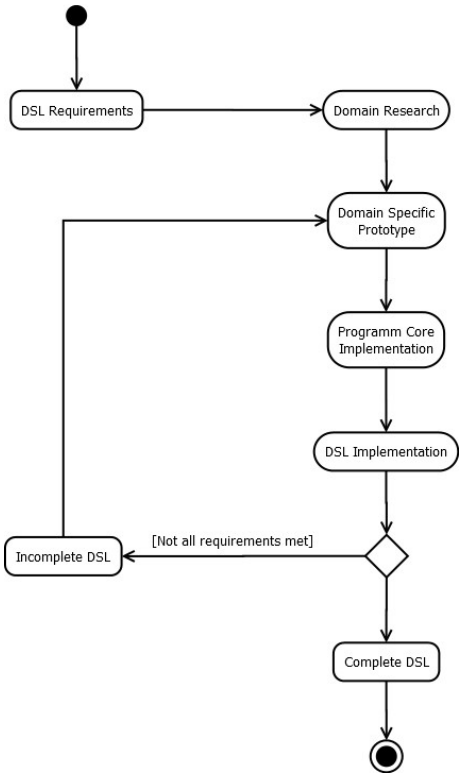

Abbildung 13: Aktivitätsdiagram DSL Entwicklung (Eigene Darstellung)

Die „DSL Requirements“ in der Abbildung können aus einem Lastenheft, Pflichtenheft, Product Backlog etc. stammen und sollen dabei möglichst grob gehalten werden; denn die vollständige Beschreibung der Requirements für die DSL werden in der „Domain Specific Prototype“-Phase im *Domain Specific Prototype* (DSP) dokumentiert.

Beispielsweise genügt es für die „Incremental Game Prototyp DSL“, die „DSL Requirements“ in sechs kurzen Punkten festzuhalten:

- Der DSL User will Währungen erstellen können;
- der DSL User will Geldwechsel beschreiben können;
- der DSL User will Geld-Generatoren erstellen können;
- der DSL User will Entwicklungen für die generierte Geldmenge beschreiben können;
- der DSL User will die Spieler-Interaktion festlegen können;
- der DSL User will beschränkten Einfluss auf die Darstellung in der GUI ausüben können.

6.5.1 Domain Research

Der erste Schritt zur Entwicklung einer Domain Specific Language ist das Bestimmen der Ansprüche, denen die DSL gerecht werden muss. Zu diesem Zweck entwickelt der Autor in Anlehnung an die in der Biologiewissenschaft verwendeten Bestimmungsschlüssel einen eigenen Bestimmungsschlüssel für DSL Kategorien (siehe Abbildung 14).

Exkurs: Bestimmungsschlüssel

In der Biologie werden zum Bestimmen von bestimmten Pflanzen oder Tierarten (Bestimm-)Bücher wie „Exkursionsflora von Deutschland“ (Jäger, 2011) oder „Brohmer – Fauna von Deutschland“ (Schaefer,

Ansorge und Brohmer, 2010) herangezogen. Diese Bücher werden auch als Bestimmungsschlüssel bezeichnet.

Ein Bestimmungsschlüssel ist ein System aus Fragen zu bestimmten Merkmalen eines zu untersuchenden Lebewesens. Dabei führt die Antwort auf eine Frage entweder zu einer weiteren Frage oder klassifiziert das untersuchte Lebewesen. Ein Bestimmungsschlüssel ist quasi ein großer nichtlinearer multivarianter Entscheidungsbaum in Buchform zur Klassifizierung.

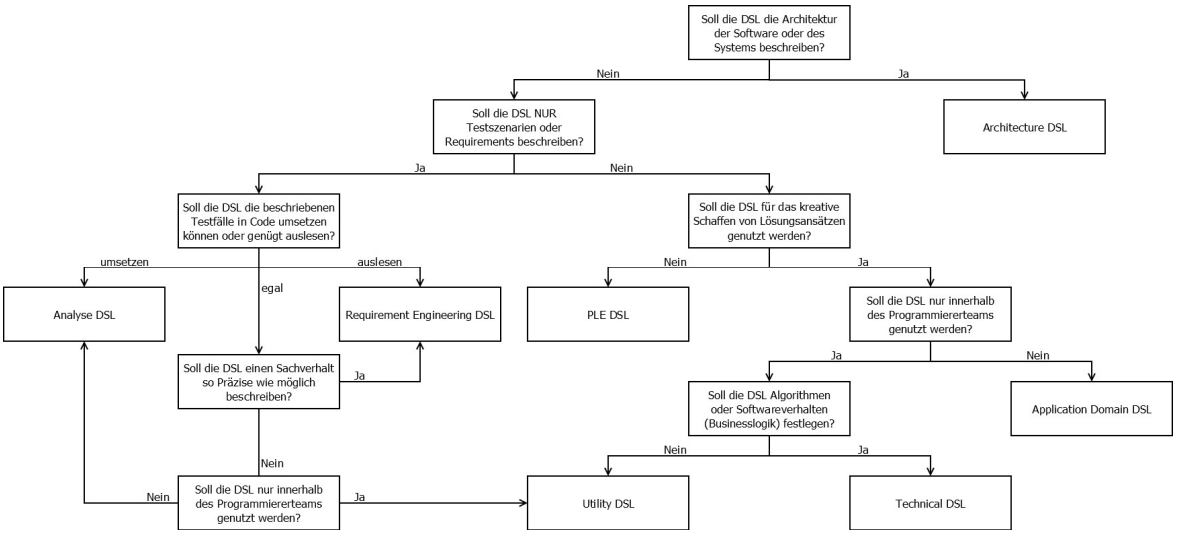

Abbildung 14: Bestimmungsschlüssel für DSL Kategorien (Eigene Darstellung)[42]

Wird der Bestimmungsschlüssel für DSL Kategorien (siehe Abbildung 14) auf die „Incremental Game Prototyp DSL“ angewandt, führt er zu folgendem Ergebnis:

Frage: Soll die DSL die Architektur der Software oder des Systems beschreiben?
Antwort: Nein, die Domänen Experten haben kein Interesse an der Softwarearchitektur.

Frage: Soll die DSL NUR Testszenarien oder Requirements beschreiben?
Antwort: Nein, die DSL soll zur Erstellung eines Prototyps genutzt werden.

Frage: Soll die DSL für das kreative Schaffen von Lösungsansätzen genutzt werden?
Antwort: Ja, die Experten wollen ihr eigenes Fachwissen anwenden, um Problemstellungen beim Prototypbau zu lösen.

[42] Auf der beiliegenden CD als Vektorgrafik enthalten

Frage: Soll die DSL nur innerhalb des Programmierer Teams genutzt werden?
Antwort: Nein, die Experten müssen nicht zwingend Programmierer sein.

➔ Application Domain DSL

Das Wissen, dass eine Application Domain DSL entwickelt wird, liefert dem Entwickler-Team Informationen über die **Rahmenbedingungen** der geplanten DSL bezüglich ihrer Syntax und Mächtigkeit (siehe Kapitel 4.2 „DSL Kategorien“):

- Die Syntax der DSL muss leicht verständlich und übersichtlich sein.
- Die DSL darf nicht zur Generierung von Programmteilen genutzt werden, die für die Sicherheit oder Stabilität der Software notwendig sind.
- Die DSL muss so restriktiv wie nötig/möglich sein, um Fehlern so weit wie möglich vorzubeugen.
- Grundlegende Incremental Game Mechaniken müssen im Program Core manuell programmiert oder zumindest ohne Einfluss des DSL User generiert werden.
- Die Programmierer benötigen ein gutes Domänenwissen.

Bevor nun die Entwicklung eines *Domain Specific Prototype* beginnt, informieren sich die Programmierer an dieser Stelle des Projekts über die Domäne (hier: Incremental Game), um ein grundlegendes Verständnis dafür zu entwickeln. Das Sammeln von Informationen ist besonders bei DSLs wichtig, die den Programmierer zwingen, mit bisher unbekannten Domänen zu arbeiten.

6.5.2 Domain Specific Prototype Development

Nach der Domain Research entwickeln die Programmierer zusammen mit den Experten einen *Domain Specific Prototype* (DSP). Ein *Domain Specific Prototype* ist ein möglichst vollständiges, aber nicht funktionales Syntaxbeispiel und nicht die Xtext Grammatik der DSL oder eines DSL-Fragments. Der Prototyp beschreibt dabei syntaktisch alle Objekte, Beziehungen und Anweisungen, wie es auch die fertige DSL beschreiben soll. Anschließend werden der DSP oder die DSPs zu *Blueprints* erweitert. *Blueprint*: eine Ansammlung von Templates für zu generierende Klassen, die aus einem bestimmten Stück der DSPs generiert werden (s.u.).

Um das Vorgehen dieses Schrittes zu verdeutlichen, wird er anhand eines Ausschnittes vom DSP des Pilotprojekts „Incremental Game Prototyp DSL“ demonstriert (siehe Abbildung 15).

```
Currency CurrencyA {
    Name => "Geld1";
    Amount => 0;
    static MainCurrency => this;
    BelowZero => false;
}

Currency CurrencyB {
    Name => "Geld2";
    Amount => 1;
    BelowZero => false;
}

Exchange CurrencyAToB {
    ExchangeCurrency1 => CurrencyA;
    ExchangeCurrency2 => CurrencyB;
    Calculation (valueToExchange){
        ExchangeCurrency1.Amount = ExchangeCurrency1.Amount - valueToExchange;
        ExchangeCurrency2.Amount = valueToExchange * 2;
    }
}


Exchange CurrencyBToA {
    ExchangeCurrency1 => CurrencyB;
    ExchangeCurrency2 => CurrencyA;
    Calculation (valueToExchange){
        ExchangeCurrency1.Amount = ExchangeCurrency1.Amount - valueToExchange;
        ExchangeCurrency2.Amount = valueToExchange * 2;
    }
}
```

Abbildung 15: DSP – Currency (Eigene Darstellung)

Wie im DSP (siehe Abbildung 15) erkennbar ist, wurde neben der „Currency“-Klasse die „Exchange“-Klasse für den Geldwechsel eingeführt; denn ein Geldwechsel betrifft nur die „Currency“-Klassen untereinander. Die grün und rot markierten Stellen in dem DSP greifen einer *Usability Evaluation* vor. Grün steht für keine oder nur geringe Makel, Rot bedeutet, dass das Sprachfeature in diesen Kontext verwirrend, zu komplex oder fehleranfällig ist. Der DSP wird nun so lange einer *Usability Evaluation* unterzogen und im Anschluss daran überarbeitet, bis ein akzeptabler DSP mit möglichst wenigen roten Stellen vorliegt. Bei diesem Vorgehen ist es durchaus möglich, dass sich für den DSP nicht die „beste“ Lösung, falls es eine solche überhaupt gibt, finden oder durchsetzen lässt; denn die Syntax, die Features und die Funktionalität eines DSP hängen sehr stark von den Erfahrungen und Präferenzen der einzelnen Teammitglieder ab.

Für die Currency DSP (siehe Abbildung 15) führt die *Usability Evaluation* zu folgenden Erkenntnissen:

Erstens ist die Einführung des Konzepts für statische Variablen in Klassen wie „Currency“ stark fehleranfällig. Ein DSL User, der das Konzept statischer Variablen nicht kennt oder falsch verstanden hat, wird bei der fehlerhaften Verwendung einer statischen Variablen mit einem für ihn unvorhersehbaren Verhalten konfrontiert werden. Angenommen, der DSL User versteht nicht, dass die Variable *MainCurrency* keinen direkten Bezug zu einer Instanz der „Currency“-Klasse haben muss, um einen Wert zu enthalten, und weist deshalb die Variable *MainCurrency* mehrmals in unterschiedlichen *Currency*-Klassen zu. Die statische *MainCurrency* Variable wird dann vom Generator mehrmals überschrieben oder zugewiesen und hat im Incremental Game Prototyp des DSL Users einen falschen Wert.

Als Konsequenz muss der DSL User entweder einen Kollegen fragen, der sich mit der DSL auskennt, oder vergeudet Zeit in dem Versuch, das Problem alleine zu lösen.

Um diese Konsequenz zu vermeiden, können drei Ansätze verfolgt werden.

1. Es wird eine weitere Klasse eingeführt, die als Container für ansonsten statische Variablen dient.
2. Es wird sichergestellt, dass die statische Variable nur einmal in einer „Currency"-Klasse zugewiesen wird.
3. Es wird in der „Currency"-Klasse eine lokale boolsche Variable „*IsMain*" definiert und damit die Klasse mit „*IsMain* = *true*" zur Hauptwährung. Um sicherzustellen, dass der User *IsMain* nur einmal als *true* setzt, wird eine Constrain im Xtext Validator[43] festgelegt.

Für die „Incremental Game Prototyp DSL" fiel die Wahl auf die *IsMain*-Methode[44], da eines der Ziele eine möglichst übersichtliche Sprache ist und die Dezentralisierung der Informationen zu einer „Currency"-Klasse kontraproduktiv wäre (siehe Abbildung 15).

Zweitens ist es beim objektorientierten Programmieren naheliegend, die Wechselkurse einer Währung in Form eines eigenen Objektes darzustellen, um eine lockere Assoziation zwischen der „Currency"-Klasse und der „Exchange"-Klasse zu bilden. Für einen DSL User ohne Erfahrung mit Objektorientierung stellt die zusätzliche Klasse aber einen unnötigen Overhead und eine Dezentralisierung der Informationen über eine Currency dar. Schließlich will der DSL User der „Incremental Game Prototyp DSL" nicht wissen, dass der Währungswechsel von A nach B durch ein eigenes Objekt ermöglicht wird. Er will nur wissen, mit welcher Formel der Wechselkurs beschrieben wird. Deshalb wird die „Exchange"-Klasse in der DSL durch eine Funktion in der betroffenen „Currency"-Klasse ersetzt (siehe Abbildung 16).

Zusätzlich wurde das Zugreifen auf Objekt Variablen weitgehend durch einen kontextbezogenen, impliziten Variablenzugriff abgelöst. Des Weiteren wurden die Semikolons weggekürzt, der Variable-Setzen-Operator wurde von einem Pfeil („=>") zu einem einfachen Gleichzeichen („=") umgeändert und die Attribute und neuen „Exchange"-Funktionen mit Regionen in der „Currency"-Klasse voneinander getrennt (siehe Abbildung 16).

---

[43] Eine solche Constrain direkt in der Xtext Grammatik zu beschreiben, wäre sehr aufwändig, wenn nicht sogar unmöglich. Deshalb wird ein Constrain besser mit einem IDE Feature wie der Validierung umgesetzt als mit der Xtext Grammatik.

[44] Sie unterscheidet sich im Wesentlichen nicht vom Konzept der statischen Klassenvariablen, doch die Verwendung einer lokalen Variablen ist einfacher als die einer statischen Variablen.

```
Currency Currency1 {
   Attributes{
     Name = "Geld1"
     Acronym = "$$1"
     BelowZero = false
     StartValue = 0
     IsMain = true
   }
   Exchange{
      from Currency2 (Amount in: moneyInFrom2){
         Currency2 = Currency2 - moneyInFrom2
         this = this + moneyInFrom2 / 3
      }
   }
}

Currency Currency2 {
   Attributes{
      Name = "Geld2"
      Acronym = "$$2"
      BelowZero = false
      StartValue = 0
      IsMain = false
   }
   Exchange{
      from Currency1 (Amount in: moneyInFrom1){
         Currency1 = Currency1 - moneyInFrom1
         this = this + moneyInFrom1 * 3
      }
   }
}
```

Abbildung 16: Finaler Domain Specific Prototype (Eigene Darstellung)

Der neu entstandene finale *Domain Specific Prototyp* (siehe Abbildung 16) wird nach seiner Fertigstellung als Unit-Test für den Parser formuliert und mit der Xtext Grammatik beschrieben. Dabei ist besonders darauf zu achten, dass Ausdrücke anhand einer natürlichen Assoziativität interpretiert werden. Vorzugsweise wird dabei linksassoziativ interpretiert, da Xtext einen mächtigen *LL(*) Algorithmus*[45] implementiert und die meisten Programmiersprachen wie Java und C# grundsätzlich auch linksassoziativ orientiert sind.[46] Nachdem die Unit-Tests für den Parser erfolgreich waren, werden anschließend die Constrains der Grammatik als Unit-Tests für den Validator, Scoper und Linker beschrieben und implementiert. Die Umsetzung der M2C- oder M2M- Transformation erfolgt an dieser Stelle noch nicht, da vorher der Program Core auf der Zielplattform entwickelt werden muss.

[45] Für weitere Informationen zum *LL(*) Algorithmus* von Xtext siehe https://eclipse.org/Xtext/documentation/301_grammarlanguage.html (Zuletzt besucht am 12.09.2016)

[46] Beispielsweise wird der Ausdruck 6/9/5 in Java und C# als (6/9)/5 interpretiert, nicht als 6/(9/5).

6.5.3 Program Core Implementation

Nach der Erstellung *des Domain Specific Prototype* und seiner Umsetzung, wird der Prototyp dazu genutzt, *Blueprints* für die Klassen des *Program Core* zu erstellen. Dafür müssen aus dem Domain Specific Prototype sinnvolle Klassenbeschreibungen extrahiert und anschließend mit UML festgehalten werden; denn *Blueprints* sind die Grundlage zum Entwickeln der Architektur des *Program Cores*. Des Weiteren beschreibt ein *Blueprint*, wie bei der M2C-Transformation DSL-Klassen zu Klassen des *Program Cores* transformiert werden.

Beispielsweise lässt sich anhand des finalen Currency DSP (siehe Abbildung 16) der unten gezeigte *Blueprint* erstellen (siehe Abbildung 17).

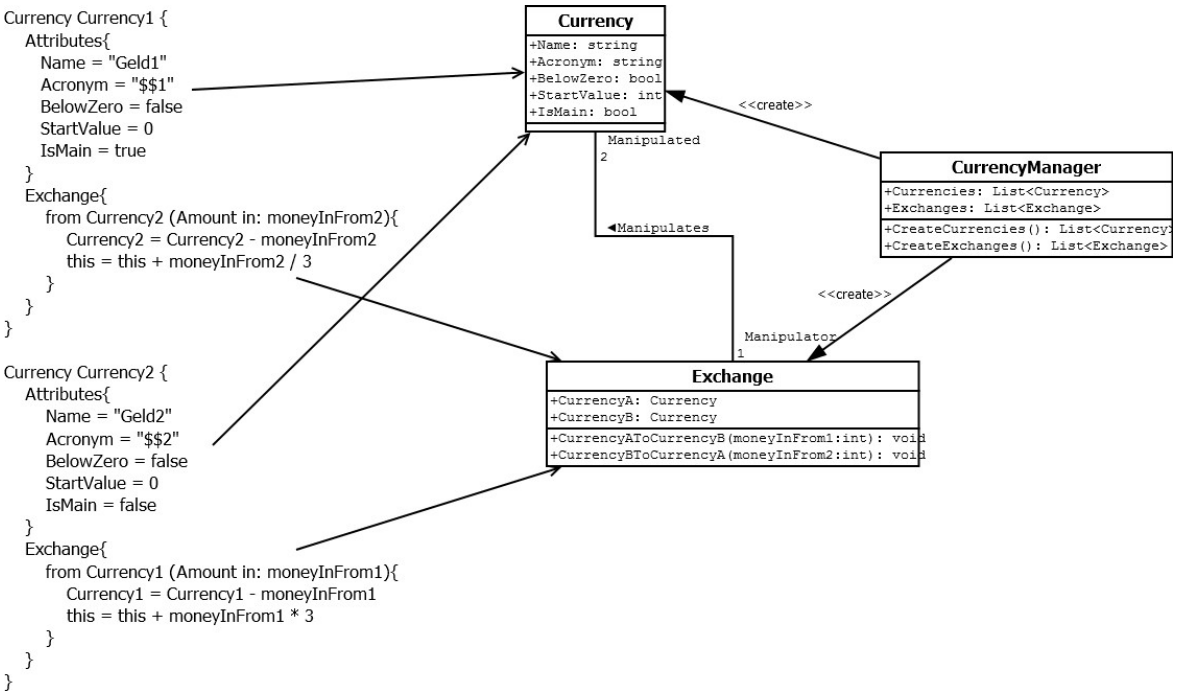

Abbildung 17: Currency Blueprint (Eigene Darstellung)

Anschließend benutzt ein Programmierer den/die *Blueprints* zum Erweitern oder Erstellen der Architektur des *Program Cores*. Aus dem Currency *Blueprint* (Siehe Abbildung 17) kann der Architekt folgende Informationen für die Architektur des Currency Features herauslesen:

- Es muss eine Containerklasse Currency geben, die mindestens fünf Attribute (Name, Acronym, BelowZero, StartValue, IsMain) hat.
- Eine Exchange Klasse referenziert immer auf zwei spezielle Currency Instanzen (CurrencyA und CurrencyB).
- Die Exchange Klasse hat zwei Funktionen mit jeweils dem Argument „amount“ der gewechselten Geldmenge.
- Die zwei Funktionen der Exchange Klasse stehen jeweils für eine Richtung des Währungswechsels zwischen CurrencyA und CurrencyB.

- Die Instanzen von Currency und Exchange werden von einem CurrencyManager Objekt verwaltet.
- Die Currency Instanzen müssen vor den Exchange Instanzen initialisiert werden.

Die Informationen führen so zu einer dem Mediator Pattern ähnelnden Architektur. Dabei wird der Klassenname „Exchange“ zu „CurrencyTransform“ geändert (siehe Abbildung 18 oder Vektorgrafik „ImcrementalGameDSL_UML.svg“ auf der beiliegenden CD).

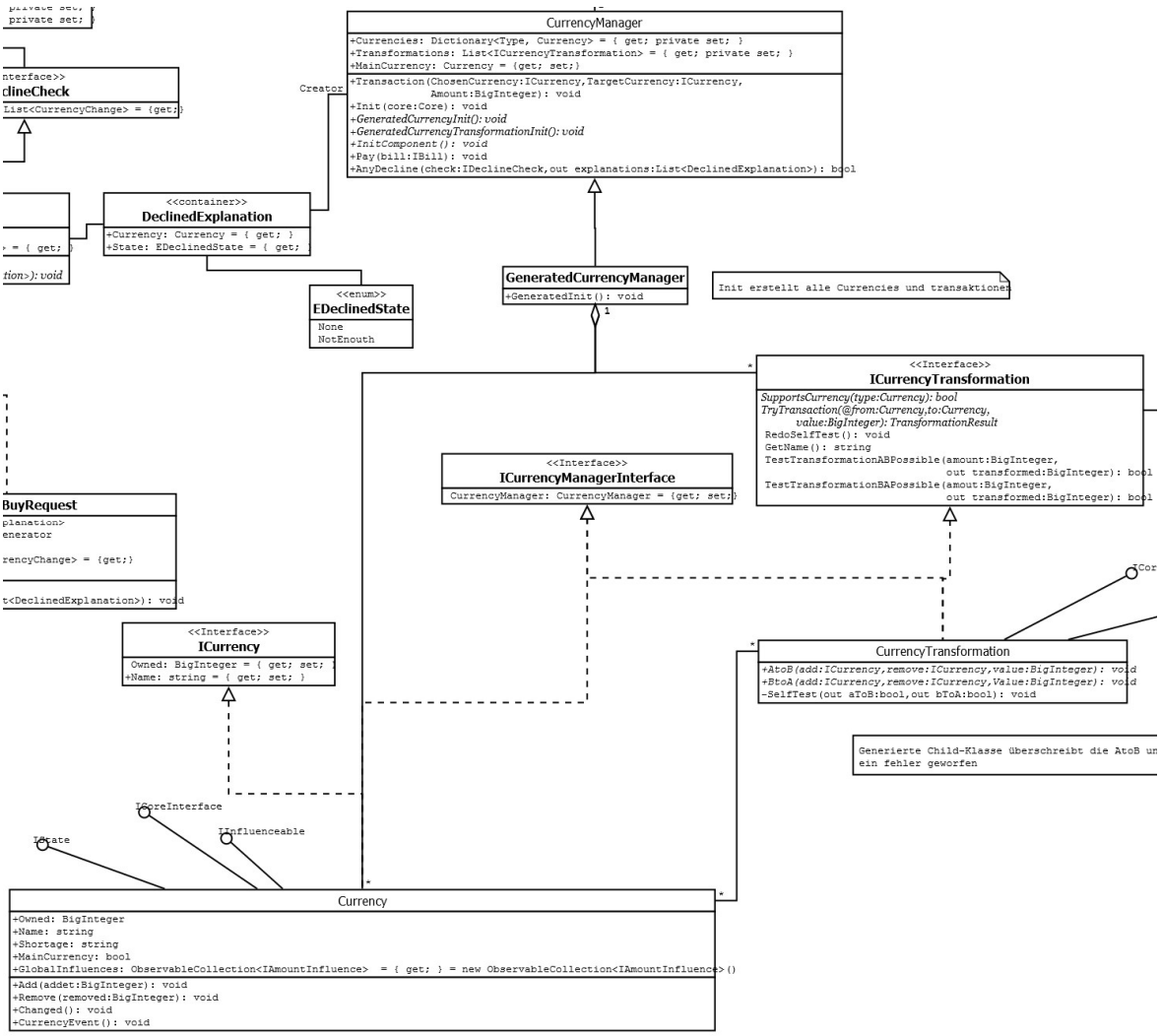

Abbildung 18: Core Programm UML (Ausschnitt Currency) (Eigene Darstellung)

Der Ausschnitt des „Incremental Game Prototyp DSL Core Programm“ UMLs zeigt, dass nicht von der DSL beschriebene Geschäftslogik direkt im *Program Core* mit abstrakten Klassen implementiert wird. Die Kinder der abstrakten Klassen werden für die Initialisierung von zugehörigen Klassen (*CurrencyManager*), zur Einbringung der mit der DSL beschriebenen Geschäftslogik (*CurrencyTransformation* – AtoB und BtoA) und zum Setzen der beschriebenen Werte bei der Initialisierung genutzt (*Currency* – Owned, Name etc.).

6.5.4 DSL Implementation

Wenn die Implementation des *Program Core* abgeschlossen ist, wird der Generator vom Entity Modelling Framework (EMF) mit den neuen notwendigen M2C- und M2M- Transformationen ausgestattet. Als Generator stellt das Xtext Projekt eine Kind-Klasse des *AbstractGenerators* mit den Funktionen *beforeGenerate(Resource, IFileSystemAccess2, IGeneratorContext), doGenerate(Resource, IFileSystemAccess2, IGeneratorContext) und afterGenerate(Resource, IFileSystemAccess2, IGeneratorContext)* bereit.

Resource übergibt eine Implementation der „Ecore Resource“ von Xtext, die Xtext-Resource[47]. Eine *Resource-Instanz* wird durch eine Resource Factory erstellt und liefert alle direkten und indirekten Inhalte einer übergebenen Source File[48] mit deren Fehlerdiagnosen und anderen Problemen. *IFileSystemAccess2* ist ein Sammelinterface für alle *IFileSystemAccess bezogenen* Interfaces. *IFileSystemAccess* ist dafür ausgelegt, mit dem Dateisystem zu interagieren, und wird dafür benötigt, die von der DSL generierten Source Files zu speichern. *IGeneratorContext* ermöglicht den Zugriff auf den *CancelIndikator*, dieser gibt an, ob das Build vom DSL User abgebrochen wurde.

Für die „Incremental Game Prototyp DSL“ wird der Generator mit einer filterbasierten Lösung realisiert. Dafür werden alle direkten Inhaltsklassen der *Resource* im *GenerationCollector* (siehe Abbildung 19) nach ihrer Klasse sortiert und gesammelt. Anschließend werden die dadurch gefüllten Listen den zugehörigen *Abstract2CSharp* Kind-Klassen übergeben. Diese *Abstract2CSharp* Klassen generieren aus jeder Inhaltsklasse eine Source File[49] . Sie melden sowohl die generierten Source Files (als *GeneratedObjectDetails*) als auch die Inhaltsklassen im *VisualStudioBinder* für das Proposal-File an. Alle vorher beim *VisualStudioBinder* angemeldeten Klassen werden im vorletzten Schritt dazu genutzt, die *Proposal File* zu generieren. Der letzte Schritt besteht darin, dass mithilfe des Programm Commanders der *ProposalDaemon* mit Powershell gestartet wird.

---

[47] Die XtextResource wird sowohl von Xtext Features wie den Parser, Linker und Serializer als auch im EMF Parser und Generator genutzt (Siehe https://eclipse.org/Xtext/documentation/308_emf_integration.html zuletzt Besucht am 10.09.2016).

[48] In Eclipse wird beim Speichern einer DSL Source File der Inhalt einer Resource Factory übergeben.

[49] Entweder eine C# Source File oder eine Partial Class Description.

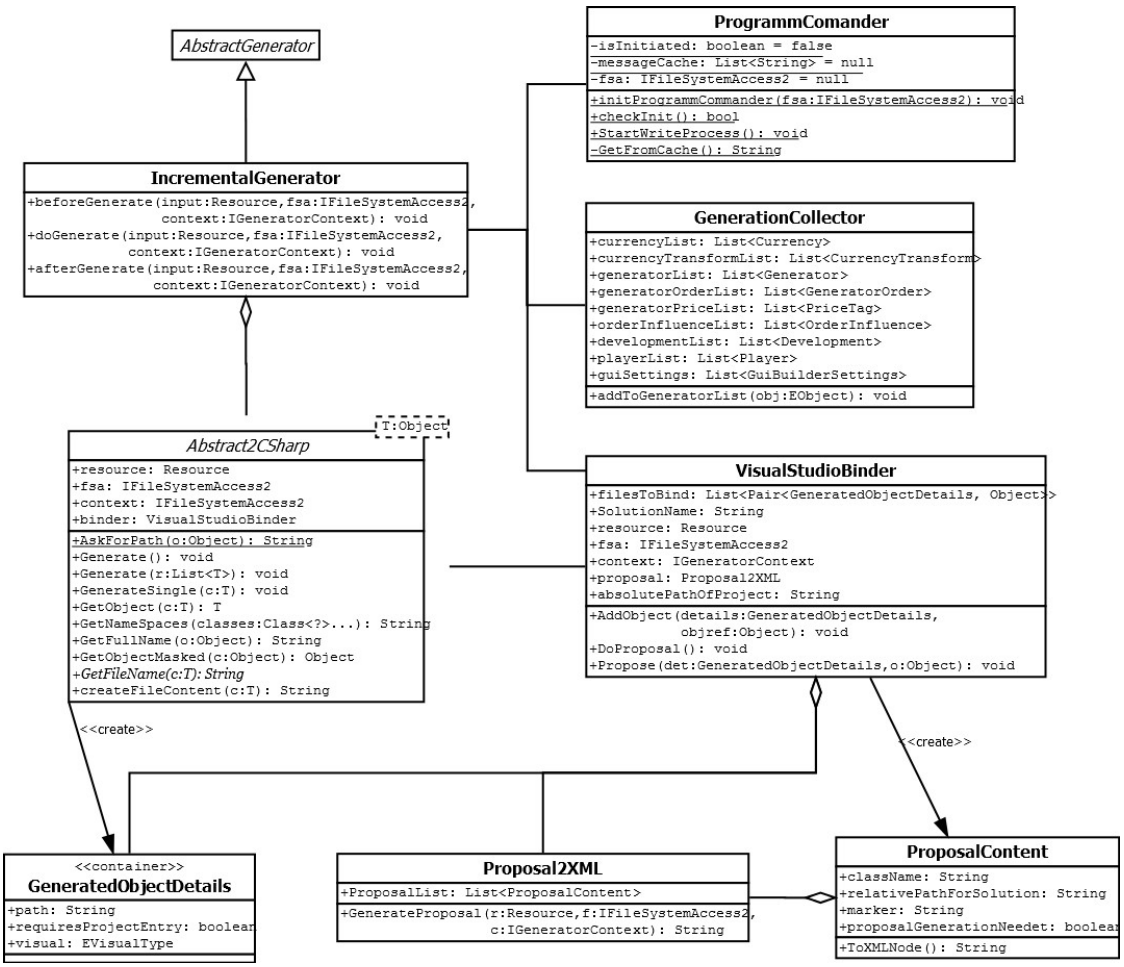

Abbildung 19: UML Generator (Xtend) (Eigene Darstellung)

7 Ergebnisanalyse des Pilotprojekts

Dieses Kapitel wertet das Pilotprojekt „Incremental Game Prototyp DSL“ im Stil einer Machbarkeitsanalyse aus. Dabei handelt es sich aber nicht um eine wirtschaftliche Machbarkeitsanalyse, sondern um eine Analyse der Erkenntnisse und neuen Methoden, die aus dem Pilotprojekt gewonnen werden konnten. Zu Beginn des Kapitels wird dafür nochmals die Aufgabenbeschreibung aus Kapitel 6.2 „Zielsetzung und Versuchsaufbau“ in einer kurzen Zusammenfassung wiederholt. Anschließend werden die Ergebnisse und gewonnenen Erkenntnisse des Pilotprojekts analysiert und erläutert. Abschließend wird im Fazit die Quintessenz der Machbarkeitsstudie für die zukünftige Forschung präsentiert.

7.1 Aufgabenbeschreibung (kurze Wiederholung)

Für das Pilotprojekt soll mit Xtext eine „Incremental Game Prototyp DSL“ entwickelt werden. Um einen besonderen Kundenwunsch zu simulieren, soll die Zielplattform Windows mit einem Visual Studio C# Projekt als *Program Core* fungieren. Die Herausforderung dabei ist, dass C# als Sprache und Visual Studio Projekt-Strukturen bis jetzt noch nicht von Eclipse unterstützt werden[50]. Als Zeitlimit für die Umsetzung der „Incremental Game Prototyp DSL“ werden 180 Stunden oder 1,2 Personenmonate festgesetzt. **Ziel** ist es, durch das Gelingen des Pilotprojekts zu beweisen, dass die Erforschung und Anwendung von MDSD-Methoden in kleinen agilen Gruppen möglich und lohnenswert ist.[51]

7.2 Ergebnisse

Das Pilotprojekt „Incremental Game Prototyp DSL“ wurde innerhalb der vorgegebenen Zeit von 1,125 Personenmonaten geplant und umgesetzt. Während der Entwicklung fielen keine Kosten für Tools oder Programme an. Bei einer kommerziellen Umsetzung des Projekts „Incremental Game Prototyp DSL“ wären jedoch im Vergleich dazu Kosten für ReSharper, BeyondCompare und Visual Studio Lizenzen angefallen. Zusätzlich hat die Entwicklung der „Incremental Game Prototyp DSL“ unterschiedliche Ergebnisse und Theorien hervorgebracht, die im Folgenden betrachtet werden.

---

[50] Stand: Eclipse Neon (10.09.2016)

[51] Für eine vollständige Aufgabenbeschreibung siehe Kapitel 6.2

Die Hypothese „*Die Nutzung von MDSD ist im kleineren agilen Umfeld möglich.*“ konnte durch das Gelingen des Pilotprojekts „Incremental Game Prototyp DSL“ als zutreffend bestätigt werden.

7.2.1 Bestimmungsschlüssel

Der Bestimmungsschlüssel für DSL Kategorien in „Domain Research“ (siehe Kapitel 6.5.1) ist ein im Rahmen des Pilotprojekts entwickelter großer nichtlinearer multivarianter Entscheidungsbaum.

Die Entscheidung für einen Bestimmungsschlüssel ist naheliegend, da die einzelnen DSL Kategorien anhand von eindeutigen Merkmalen oder einer Kombination aus vorhandenen und nicht vorhandenen Merkmalen ausgemacht werden können. Beispielsweise ist eine DSL für die Softwarearchitektur immer eine Architecture DSL, alle anderen DSLs haben entweder keinen Einfluss auf die Architektur (vgl. Analyse DSL) oder haben nicht das konkrete Ziel, die Architektur des Programms zu beeinflussen (vgl. Technical DSL).

7.2.2 Domain Specific Prototype

Die größte Herausforderung bei der Entwicklung der „Incremental Game Prototyp DSL“ mit Test Driven Development war das Testen des Parse-Vorgangs von einer Source File zu einem Model. Vor der Entwicklung der *Domain Specific Prototype* Methode wurde die Syntax direkt in den Unit Tests entwickelt (siehe Abbildung 21). Das hatte zur Folge, dass jede Modifikation oder Änderung der DSL Syntax zuerst mit einer Änderung der zugehörigen Unit Tests beginnen musste. Besonders bei der Entwicklung einer neuen DSL Syntax, wie der „Incremental Game Prototyp DSL“, kommt es zu Beginn sehr häufig zu notwendigen Syntaxänderungen mit Folgeänderungen in der restlichen Syntax. Um diese häufigen, meist zeitaufwändigen und notwendigen Syntaxänderungen vor der Implementation von Unit Tests oder der Xtext Grammatik zu entdecken und möglichst billig auszuführen, wird vor der Implementation der Unit Tests ein *Domain Specific Prototype* (DSP) entwickelt und so lange modifiziert, bis er eine *Usability Evaluation* des Teams besteht (siehe Abbildung 20).

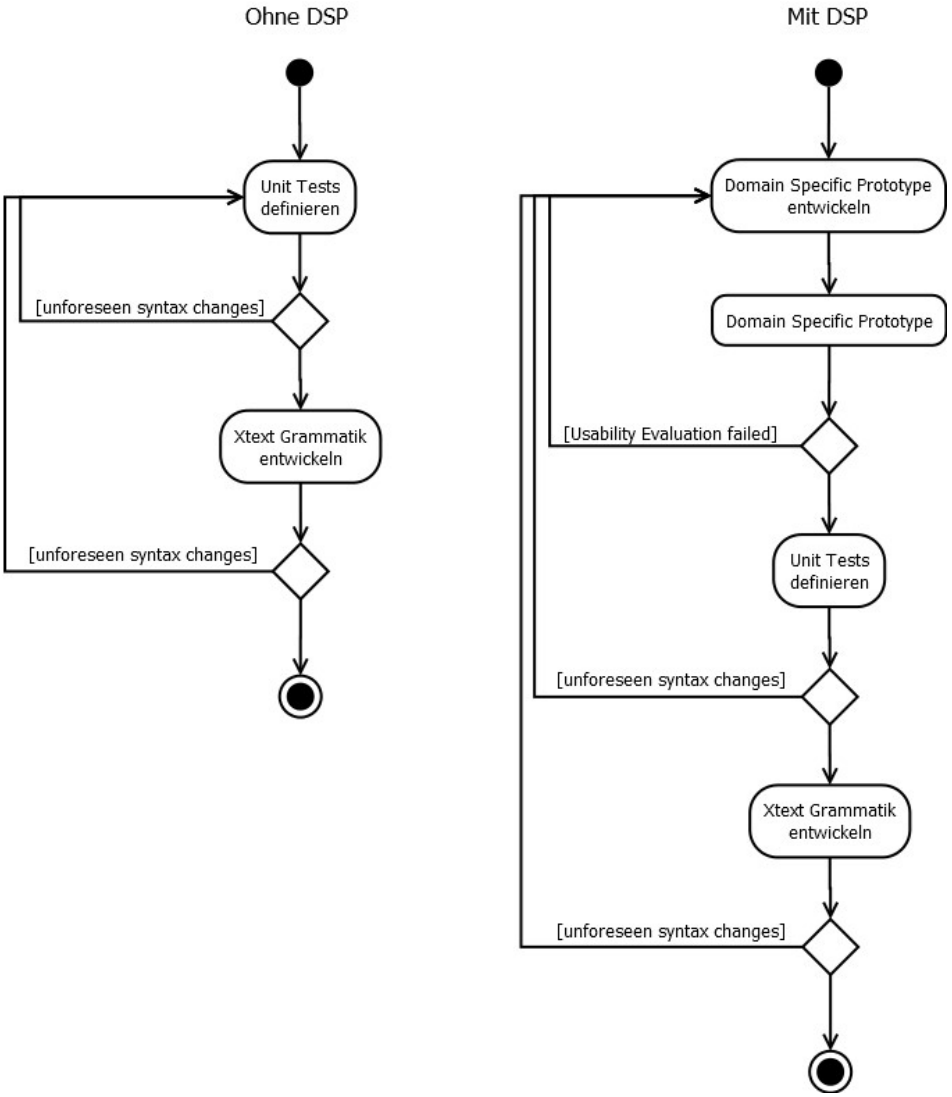

Abbildung 20: DSL Entwicklung mit TDD Aktivitätsdiagramm (Eigene Darstellung)

Die Entwicklung eines *Domain Specific Prototype* hat nicht nur Einfluss auf die Entwicklung der Xtext Grammatik. Sein Potential, zu einem sogenannten *Blueprint* weiterentwickelt zu werden (siehe Kapitel 6.5.3), hat auch Einfluss darauf, wie die Entwicklung des *Program Core* und des Xtext Generators vonstattengeht; denn durch das Extrahieren von sinnvollen Klassenbeschreibungen beinhaltet der *Blueprint* sowohl, was die zu generierenden Klassen als Variablen setzen müssen, als auch, wie bei der M2C-Transformation DSL-Klassen zu Klassen des *Program Cores* transformiert werden müssen. Dies hat zur Folge, dass der *Domain Specific Prototype* zu einem neuen Vorgehensmodell für die Entwicklung von DSLs heranwächst. Wie Abbildung 21 zeigt, haben die DSPs und *Blueprints* Einfluss auf jeden Teil der entwickelten DSL.

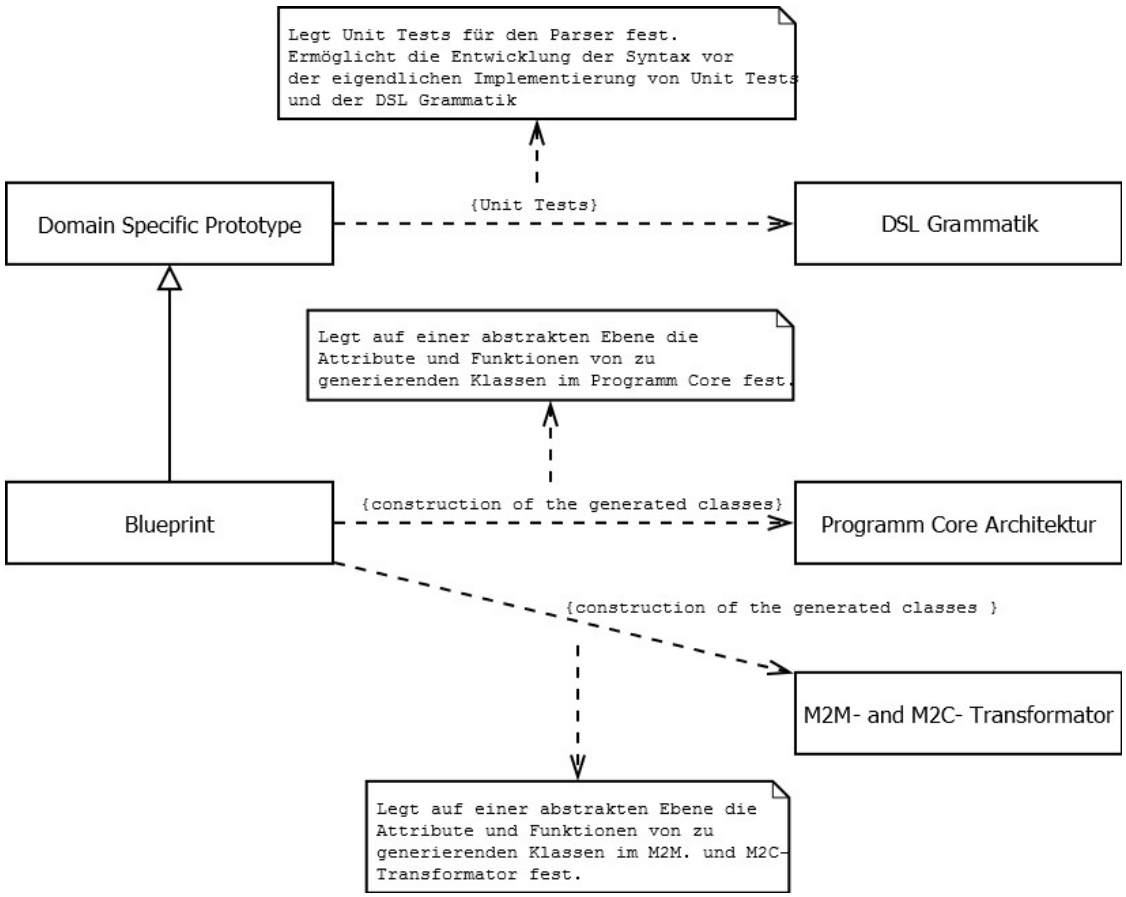

Abbildung 21: Einfluss des DSP auf ein DSL Projekt (Eigene Darstellung)

7.2.3 MDSD und Agile Methoden

Wegen der Vielzahl unterschiedlicher agiler Methoden wurde bei der Entwicklung der „Incremental Game Prototyp DSL“ der Fokus auf Kanban und Test Driven Development beschränkt. Kanban wurde wegen seiner Anpassungsfähigkeit hinsichtlich der Teamgröße und der Projektstruktur gewählt. TDD spielt in vielen agilen Philosophien und agilen Frameworks eine zentrale Rolle oder gilt als *best practise.*

Das „reine“ Test Driven Development war am Anfang des Projekts ohne die Verwendung der *Domain Specific Prototype* Methode schwer mit der Entwicklung einer DSL vereinbar. Die DSL Syntax und ihre Xtext Grammatik wurde besonders zu Beginn des Pilotprojekts häufig modifiziert oder verworfen. Als Folge wurden auch die zugehörigen Unit Tests häufig modifiziert oder verworfen, um im Anschluss daran die Xtext Grammatik anzupassen (siehe Abbildung 20). Die Entwicklung des Xtext Generator war bis zum Ende des Projektes nicht mit TDD vereinbar, da der Generator an sich eine zu hohe Komplexität für das Schreiben von aussagekräftigen Unit Tests aufweist. Stattdessen wurde beim Programmieren des Xtext Generators nach Trial-and-Error verfahren.

TDD hat sich zusammen mit der *Domain Specific Prototype* Methode während des Projektes bewährt. Besonders bei der Umsetzung des Program Core (siehe Kapitel 6.5.3)

und der Entwicklung einer Xtext Grammatik (siehe Kapitel 6.5.2) aus dem *Domain Specific Prototype* zeigten sich die Stärken von TDD, die unter anderem in einer niedrigen Fehlerrate bestanden.

Auch bei der Arbeit einer einzelnen Person stellt Kanban eine wesentliche Erleichterung hinsichtlich der Projektorganisation dar. Für die *Domain Specific Prototype* Methode ist Kanban insofern eine wichtige Grundlage, als das Konzept von Kanban für ihren Aufbau in vier strukturierte Phasen als Vorlage dient. (siehe Anhang „Kanban Board“)

Trotz einiger Herausforderungen beim Entwickeln einer DSL mit TDD und Kanban im Pilotprojekt „Incremental Game Prototyp DSL“ ist das Pilotprojekt aus der Sicht eines agilen Softwareentwicklers ein **großer Erfolg**. Durch das Arbeiten daran entstand nämlich nicht nur eine neue agile Methode zur DSL-Entwicklung, es wurden auch zwei Einsichten zur Erforschung agiler Methoden und MDSD sowie ein Vorschlag zum Umfang zukünftiger Studien gewonnen, die hier kurz erläutert werden:

1. Es ist wegen der großen Vielfalt nicht möglich eine allgemeingültige Aussage über das Zusammenspiel von agilen Methoden und MDSD Methoden zu treffen. Vielmehr müssen für jede Kombination separate Versuche gemacht werden.
2. Die Inkompatibilität einer bestimmten agilen Methode mit MDSD bedeutet nicht zwingend, dass sie nicht für die Entwicklung von MDSD angepasst oder weiterentwickelt werden kann.
3. Für weitere Studien zum Thema agile Methoden und MDSD ist es ratsam, die Versuche in einem größeren Maßstab anzulegen, sowohl was die Dauer des Versuchsprojekts als auch die Anzahl der Teammitglieder betrifft. So könnte beispielsweise ein Projekt mit sieben Teilnehmern und einer Projektdauer von vier bis sechs Personenmonaten durchgeführt werden.

Die Hypothese „*Wenn in einem mit Kanban organisierten Softwareprojekt nach der Methode des TDDs entwickelt wird, dann vereinfacht (bei bestimmten Aufgaben) die Entwicklung einer DSL die Arbeit des Teams durch verbesserte Aufgabenverteilung und die Trennung der Domäne von der Plattform-Implementation bei der Entwicklung und Wartung.*“ (siehe Kapitel 6.1) kann durch das Pilotprojekt in ihrer Aussage insofern teilweise bestätigt werden, als das Entwickeln der „Incremental Game Prototyp DSL“ zwar zeigt, dass eine DSL einen großen Teil des Domänenwissens aus dem *Program Core* herausziehen kann, dass jedoch die Entwickler sich mit der Domäne auseinandersetzen müssen, um den unveränderlichen Teil der Domäne und ihrer Konzepte in den *Program Core* und die DSL einfließen zu lassen. Andererseits hat es die gleichzeitige Anwendung von Kanban und TDD ermöglicht, die *Domain Specific Prototype* Methode zu entwickeln. Diese erlaubt eine Unterteilung der DSL Entwicklung in vier Phasen und hat deshalb eine verbesserte Aufgabenverteilung zur Folge. Zudem ist eine deutliche Vereinfachung der Wartung festzustellen; denn Änderungen am Prototyp können nun entweder direkt im *Program Core* oder in den separaten DSL Source Files vorgenommen werden. Damit fällt die Arbeitslast

bei der Änderung der Software entweder auf einen Entwickler des *Program Core* oder einen DSL User.

7.2.4 Wirtschaft

Durch das Arbeiten mit der fertigen „Incremental Game Prototyp DSL“ und deren Core konnte beobachtet werden, dass der zeitliche Aufwand und die Fehlerrate beim Erstellen eines neuen *Incremental Game Prototyp* um ein Vielfaches geringer ist, wenn anstatt einer manuellen Implementierung die DSL herangezogen wird.

In der unten angefügten Tabelle (siehe Tabelle 3) werden die Zeiten für die Entwicklung der gleichen Prototypen mit jeweils der manuellen Implementierung (C#) und der Incremental DSL gezeigt, einmal von einem Entwickler der mit beiden Sprachen und dem Projekt sehr gut vertraut ist, markiert durch (P)rofessional, und ein zweites Mal von einem Laien, markiert durch (L)aie, der bisher nur grundlegende Kenntnisse über die Sprache und das Projekt aufweist. Zusätzlich wurde der Laie vom Experten durch Fachliche Beratung und Führung unterstützt.

	C# (HH:MM, gerundet)	Incremental DSL (HH:MM, gerundet)	Gesamtzeit (HH:MM, gerundet)
Felix Engl (P)	01:11	00:25	01:25
Moritz Engl (L)	03:10	00:48	03:48
	04:21	01:13	05:34

Tabelle 3: Zeiten für den Prototyp (siehe CD für DSL Source Code)

Die Tabelle zeigt, dass die Nutzung der DSL sowohl bei einem Programmierer als auch einen Laien zu kürzeren Entwicklungszeiten tendiert. Daraus ergibt sich unter Einbeziehung des Pareto-Prinzips die folgende Theorie:

Die Entwicklung einer DSL kostet im ersten Projektdurchlauf mehr Zeit als das direkte Entwickeln des Programms. Unter Berücksichtigung des Pareto-Prinzips lässt sich ein idealisierter zeitlicher Verlauf für die Entwicklung mehrerer Softwareprojekte erstellen (vergleiche Abbildung 22 und 23). Aus den Abbildungen 22 und 23 sind jeweils nur die theoretischen Verläufe der Projektentwicklungen ersichtlich; die Gewichtung der unterschiedlichen Anteile eines Projektes sind jeweils frei gewählt.

Aus den beiden Abbildungen 22 und 23 lässt sich zusätzlich folgern, dass die Dauer der Amortisierung einer DSL Entwicklung mit dem Anteil der Geschäftslogik an der gesamten Software steigt.

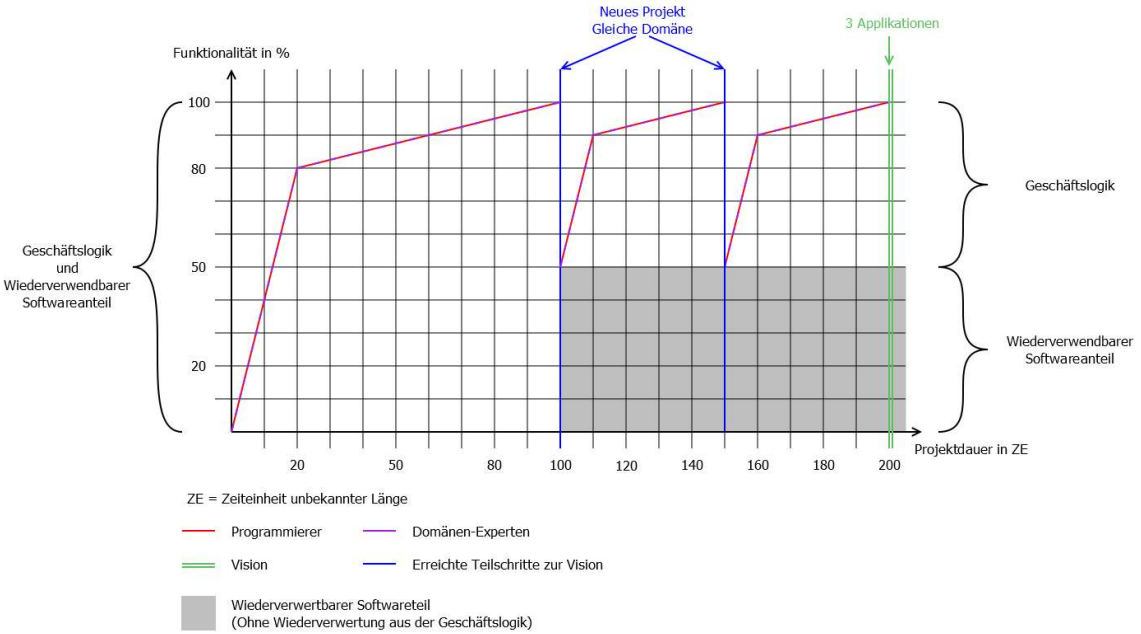

Abbildung 22: Entwicklung ohne DSL (Eigene Darstellung)

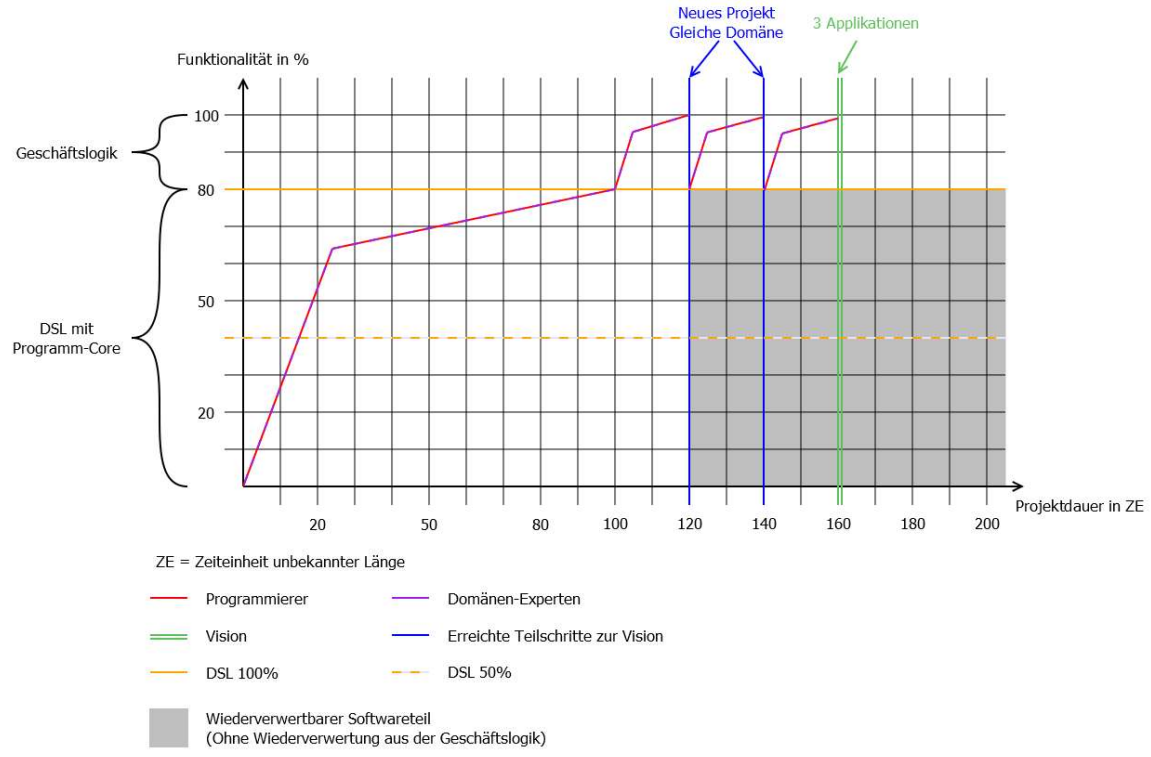

Abbildung 23: Entwicklung mit DSL (Eigene Darstellung)

Um diese Theorie zu prüfen, muss ein neuer und erweiterter Versuchsaufbau genutzt werden:

Es werden vier gleiche Entwickler-Teams, bestehend aus mindestens einem Domain Experten, einem Informatiker mit Fokus auf Softwarearchitektur und einem Informatiker als Plattformexperten für die gewählte Codeplattform gebildet. Der Wissensstand und die Fähigkeiten der Spezialisten der einzelnen Teams sollten ungefähr auf demselben Niveau und die Zusammensetzung der einzelnen Gruppen sollte identisch sein.

Alle vier Gruppen bekommen die Aufgabe, in einem Zeitraum von 6 Personenmonaten á 20 Personentagen dieselbe Software zu planen und umzusetzen. Die Vorgehensmodelle und Methoden der einzelnen Gruppen werden wie folgt festgelegt:

- *Gruppe 1:* Wasserfallmodel ohne MDSD
- *Gruppe 2:* Wasserfallmodel mit MDSD (DSL)
- *Gruppe 3:* Agile Methode ohne MDSD
- *Gruppe 4:* Agile Methode mit MDSD (DSL)

Gruppe 2 und 4 bekommen den Auftrag, so zu planen, dass ihre DSL der Beschreibung der Geschäftslogik dient. Des Weiteren müssen die Gruppen 3 und 4 dieselben agilen Methoden und Organisationsstrukturen anwenden.

Nach Ablauf von vier PM wird der Kundenwunsch bekanntgegeben, dass das Programm in mehreren Ausführungen für fünf unterschiedliche Fälle der Domäne programmiert werden soll. Zusätzlich wird der zeitliche Rahmen von 6 PM aufgehoben und das Ende des Projektes als *Open End* deklariert. Dabei sind zwei Punkte zu beachten:

Erstens werden die Gruppen in den ersten vier PM über die geplante Änderung in Unkenntnis gelassen.
Zweitens dürfen die fünf unterschiedlichen Fälle der Domäne jeweils nur auf eine Variante der ursprünglichen Geschäftslogik abzielen.

Der Theorie folgend, sollten nun die Gruppen, die mit der DSL arbeiten, schneller die Kundenwünsche bzw. die Projekte realisieren als ihre Gegenspieler. Unabhängig davon ergibt sich auch eine Steigerung der benötigten Arbeitszeit von Gruppe 4 zu Gruppe 3 bzw. von Gruppe 2 zu Gruppe 1.

Die Hypothese „*Unternehmen mit kleineren agil arbeitenden Teams profitieren auf lange Sicht von der DSL-Entwicklung, wenn die Nutzung der DSLs eine gewisse Häufigkeit überschreitet oder eine hohe Flexibilität hinsichtlich der Kundenwünsche erforderlich ist.*“ konnte mit den Werten aus Tabelle 3 weder verifiziert noch falsifiziert werden, da durch den begrenzten Umfang einer Bachelorarbeit nur eine kleine Testgruppe zur Verfügung stand. Die Ergebnisse der Versuchsdurchführung legen jedoch eine Korrektheit der Hypo-

these nahe. Um die Hypothese zu verifizieren oder zu falsifizieren, wären mehrere Durchläufe des oben bereits beschriebenen Experimentes notwendig gewesen. Auch dies ließ sich in dem beschränkten Rahmen der Bachelorarbeit nicht durchführen.

7.3 Ausblick

Die vorliegende Machbarkeitsstudie beweist unter anderem, dass eine gemeinsame Anwendung modellgetriebener und agiler Softwareentwicklung sowohl möglich als auch für bestimmte Projekte vorteilhaft ist. Die Kombination beider Entwicklungsverfahren ist ein zukunftsträchtiges System. Besonders hinsichtlich der Vereinbarkeit unterschiedlicher MDSD Methoden mit einzelnen agilen Vorgehensweisen hat die Literaturrecherche ein deutliches Defizit an geplanten, aussagekräftigen und sorgfältig ausgeführten Forschungen ergeben. Zwar wird in Büchern wie „DSL Engineering“ (Voelter, 2013), „Modellgetriebene Softwareentwicklung“ (Thomas Stahl et al., 2007) und „Emerging Innovations in Agile Software Development“ (Ghani et al., 2016) eine theoretische oder praktische Anwendungsmöglichkeit beschrieben, aber diese sind meist allgemeiner Natur oder oberflächlich gehalten und befassen sich nicht mit den explizit verwendeten MDSD Methoden und agilen Vorgehensmodellen. Folglich schrecken Unternehmen, unabhängig von ihrer Größe, und Programmierer vor der Anwendung von MDSD zurück. Damit jedoch droht das vielversprechende Gebiet der modellgetriebenen Softwareentwicklung in die Versenkung der „Sonderfälle“ zu verschwinden. Es liegt deshalb an den Hochschulen und Universitäten, die ersten Schritte in das bisher noch wenig erforschte Gebiet zu wagen, um neue Wege zu einem Miteinander von modellgetriebener und agiler Softwareentwicklung in der Projekttheorie und Projektpraxis zu erforschen.

8 Reflexion und Schlusswort

Das letzte Kapitel der Bachelorarbeit reflektiert Erkenntnisse, die über die Frage „Ist die Nutzung von MDSD im kleineren agilen Umfeld möglich?“ hinausgehen. Kern der Reflexion ist ein kritischer Blick auf den Programmierer, der plant, mit Modell Driven Software Development zu arbeiten. Abschließend wird im Schlusswort ein Ausblick für MDSD in der heutigen Zeit gewagt.

8.1 Reflexion

Welchen Kenntnisstand sollte ein Programmierer für das Arbeiten mit MDSD mitbringen?

Ein Programmierer für MDSD ist Softwarearchitekt und Programmierer in einem. Er verwirft die Idee, dass Softwarearchitektur und Programmcode unterschiedliche Bereiche sind; denn er erkennt, dass Änderungen in der Softwarearchitektur unmittelbar den Programmcode beeinflussen und Einschränkungen des Programmcodes unmittelbar die Softwarearchitektur bestimmen.

Besonders bei der Entwicklung einer DSL fällt auf, dass in beiden Bereichen gute Kenntnisse und Erfahrung erforderlich sind; denn für das Entwickeln einer Sprach-Syntax und Grammatik braucht der Entwickler **zum einen** Erfahrung mit dem Schreiben von Code in unterschiedlichen Programmiersprachen. Schließlich kann man von einem Sprachanfänger im Englischen (z.B. Sprachniveau A2) auch nicht erwarten, dass er einen Text auf dem Niveau von Shakespeare versteht oder sogar schreibt. **Zum anderen** muss der Entwickler einer Sprach-Syntax und Grammatik über gutes Verständnis für abstrakte Softwarearchitektur verfügen, um sowohl die Zusammenhänge der DSL mit dem Metamodell und dem Metametamodell zu bemerken als auch die Auswirkungen der DSL auf die Umsetzung des Program Cores zu begreifen.

8.2 Schlusswort

Die Bachelorarbeit beweist anhand eines Beispiels, dass die Entwicklung mit MDSD in einem agilen Umfeld möglich ist. Es ist jedoch auch durchaus möglich, dass Methoden des MDSD die agile Softwareentwicklung in völlig neue, bisher unbekannte Bahnen lenken kann. Schon ein Pilotprojekt im kleinsten Maßstab kann zu neuen Erkenntnissen und Verfahren führen, die sowohl vielversprechend als auch originell sind.

9 Literatur

BACKUS, J.W., [1959]. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference [online] [Zugriff am: 7. August 2016]. Verfügbar unter: http://www.softwarepreservation.org/projects/ALGOL/paper/Backus-ICIP-1959.pdf

BECK, K. und C. ANDRES, 2007. *Extreme programming explained* [online]. *Embrace change* [Zugriff am: 21. Juli 2016]. Verfügbar unter: https://www.safaribooksonline.com/library/view/extreme-programming-explained/0321278658/

BECK, K., J. GRENNING, R.C. MARTIN, M. BEEDLE, J. HIGHSMITH, S. MELLOR, A. VAN BENNEKUM, A. HUNT, K. SCHWABER, A. COCKBURN, R. JEFFRIES, J. SUTHERLAND, W. CUNNINGHAM, J. KERN, D. THOMAS, M. FOWLER und B. MARICK, 2001. *Manifesto for Agile Software Development* [online]. 11 Juli 2016, 12:00 [Zugriff am: 21. Juli 2016]. Verfügbar unter: http://agilemanifesto.org/

BIGLEVER SOFTWARE, o.J. [2013]. *Product Line Engineering* [online]. 29 Mai 2013, 12:00 [Zugriff am: 1. August 2016]. Verfügbar unter: http://www.productlineengineering.com/overview/what-is-ple.html

COOK, S., A.C. WILLS, S. KENT und G. JONES, 2007. *Domain-specific development with Visual Studio DSL tools* [Online]. Upper Saddle River, N.J.: Addison-Wesley. Microsoft .NET development series. ISBN 9780321398208. Verfügbar unter: http://proquest.safaribooksonline.com/9780321398208

CREATIVE MICROSALES, 1984. Be Creative. …and save valuable dollars enhancing your IBM PC or compartible! [Online]. In: CW COMMUNICATIONS INC., Hg. *Info world. The Newsweekly for Microcomputers.* Vol. 6. Framingham: CW Communications Inc., S. 64. ISBN 0199-6649 [Zugriff am: 30. Juli 2016]. Verfügbar unter: https://books.google.de/books?id=xS4EAAAAMBAJ&lpg=PA53&ots=GXBwk5Gvqe&dq=Model%20M-10%20computer&pg=PP1

EFFTINGE, S., 2015. *The Future of Xtext* [online]. 1 September 2016, 12:00 [Zugriff am: 22. September 2016]. Verfügbar unter: http://blog.efftinge.de/2015/05/the-future-of-xtext.html

FISH, S., o.J. *Perl for Perl Newbies* [online]. *A brief history of Perl.* 4 April 2016, 12:00 [Zugriff am: 29. Juli 2016]. Verfügbar unter: http://www.shlomifish.org/lecture/Perl/Newbies/lecture1/intro/history.html

FOWLER, M., 2005. *FluentInterface* [online]. 23 Juni 2008, 12:00 [Zugriff am: 30. Juli 2016]. Verfügbar unter: http://martinfowler.com/bliki/FluentInterface.html

FOWLER, M., 2011. *Domain-specific languages* [Online]. Upper Saddle River, NJ: Addison-Wesley. The Addison-Wesley signature series. ISBN 9780132107549. Verfügbar unter: https://www.safaribooksonline.com/library/view/domain-specific-languages/9780132107549/

GARSHOL, L.M., 25. Juli 2014. *BNF and EBNF: What are they and how do they work?* [online]. 25 Juli 2014, 12:00 [Zugriff am: 8. August 2016]. Verfügbar unter: http://www.garshol.priv.no/download/text/bnf.html

GHANI, I., D.N.A. JAWAWI, S. DORAIRAJ und A. SIDKY, 2016. *Emerging innovations in agile software development* [Online]. Hershey, PA: Information Science Reference, an imprint of IGI Global. Advances in systems analysis, software engineering and high performance computing (ASASEHPC) book series. ISBN 9781466698581 [Zugriff am: 17. September 2016]. Verfügbar unter: https://www.safaribooksonline.com/library/view/emerging-innovations-in/9781466698581/

JÄGER, E.J., 2011. *Exkursionsflora von Deutschland. Gefäßpflanzen: Grundband.* 20., neu bearb. und erw. Aufl. Heidelberg: Spektrum Akademischer Verlag. ISBN 9783827416063.

JETBRAINS, 2015. *Introduction to JetBrains MPS* [online]. *Generator* [Zugriff am: 20. September 2016]. Verfügbar unter: https://www.youtube.com/watch?v=9qlRztenuIc&index=8&list=PLQ176FUIyIUY9rAcAH6MNOxJqGfau0Jb1

KEMENY, J.G. und T.E. KURTZ, 1968. *BASIC* [online]. *Fourth Edition* [Zugriff am: 25. September 2016]. Verfügbar unter: http://www.bitsavers.org/pdf/dartmouth/BASIC_4th_Edition_Jan68.pdf

MICROSOFT, 2013. *C# Language Specification 5.0* [online] [Zugriff am: 27. Juli 2016]. Verfügbar unter: https://www.microsoft.com/en-us/download/details.aspx?id=7029

MICROSOFT, o.J. [2015a]. *Code Generation and T4 Text Templates* [online] [Zugriff am: 15. August 2016]. Verfügbar unter: https://msdn.microsoft.com/en-us/library/bb126445.aspx

MICROSOFT, o.J. [2015b]. *MSBuild* [online] [Zugriff am: 7. September 2016]. Verfügbar unter: https://msdn.microsoft.com/en-us/library/dd393574.aspx

RAYMON, E.S., 2003. *The Art of Unix Programming* [online]. 3 August 2008, 12:00 [Zugriff am: 30. Juli 2016]. Verfügbar unter: http://www.faqs.org/docs/artu/

REDDY, A., 2015. *The Scrumban [r]evolution* [online]. *Getting the most out of Agile, Scrum, and lean Kanban* [Zugriff am: 26. Juli 2016]. Verfügbar unter: https://www.safaribooksonline.com/library/view/the-scrumban-revolution/9780134077543/

RICHARDSON, M., 1999. Larry Wall, the Guru of Perl [online]. Discover a bit about Perl's creator and what's happening with Perl. *Linux Journal*, (61) [Zugriff am: 31. Juli 2016]. Verfügbar unter: http://www.linuxjournal.com/article/3394

RUBIN, K.S., 2012. *Essential Scrum* [online]. *A practical guide to the most popular agile process* [Zugriff am: 22. Juli 2013]. Verfügbar unter: https://www.safaribooksonline.com/library/view/essential-scrum-a/9780321700407/

SCHAEFER, M., H. ANSORGE und P. BROHMER, 2010. *Brohmer - Fauna von Deutschland. Ein Bestimmungsbuch unserer heimischen Tierwelt.* 23., durchges. Aufl. Wiebelsheim: Quelle & Meyer. Quelle & Meyer Bestimmungsbücher. ISBN 9783494014722.

SCHWABER, K. und J. SUTHERLAND, 2016. *The Scrum Guide* [online]. *The Definitive Guide to Scrum: The Rules of the Game* [Zugriff am: 25. Juli 2016]. Verfügbar unter: http://www.scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-US.pdf

SCHWABER, K., o.J. [2002]. *Scrum* [online]. *The art of Possible* [Zugriff am: 22. Juli 2016]. Verfügbar unter: http://static1.1.sqspcdn.com/static/f/447037/6486080/1270926852487/Brochure.pdf?token=W6%2FVzzYwc2YqySTLa06pLf7%2BK6o%3D

THOMAS STAHL, MARKUS VÖLTER, SVEN EFFTINGE und ARNO HAASE, 2007. *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management* [Online]. 2. aktualisierte und erweiterte Auflage. Heidelberg: dpunkt.verlag. ISBN 978-3-89864-881-3 [Zugriff am: 20. Juli 2016]. Verfügbar unter: https://www.dpunkt.de/buecher/3680/-modellgetriebene-softwareentwicklung-10826.html

TOMAS BJORKHOLM und JANNIKA BJORKHOLM, 2015. *Kanban in 30 days* [online] [Zugriff am: 22. Juli 2016]. Verfügbar unter: https://www.safaribooksonline.com/library/view/kanban-in-30/9781783000906/

TOYOTA MOTOR CORPORATION, o.J. *Just-in-Time* [online]. *Philosophy of complete elimination of waste* [Zugriff am: 26. Juli 2016]. Verfügbar unter: http://www.toyota-global.com/company/vision_philosophy/toyota_production_system/just-in-time.html

VOELTER, M., 2013. *DSL Engineering* [online]. *Designing, Implementing and Using Domain-Specific Languages* [Zugriff am: 7. Juli 2016]. Verfügbar unter: http://dslbook.org/

WALL, L., 1988. *v13i001* [online]. *Perl, a "replacement" for awk and sed.* Archive-name: perl/part01. 2 Januar 1988, 12:00 [Zugriff am: 31. Juli 2016]. Verfügbar unter: https://groups.google.com/forum/

XTEXT-TEAM, 2016. *Integrating Xtext Language Server Support in Visual Studio Code* [online] [Zugriff am: 22. September 2016]. Verfügbar unter: https://blogs.itemis.com/en/integrating-xtext-language-support-in-visual-studio-code

Anhang

Glossar

Begriff		Bedeutung
Domain Research		Erste Phase der vom Autor entwickelten Methode zur Entwicklung von Domain Specific Languages. Sie dient zur Bestimmung der notwendigen DSL Kategorie und zum Aneignen von Domänen-Wissen im Programmierer-Team.
Domain Specific Prototype		*Domain Specific Prototype* ist die Bezeichnung des Kernelements der vom Autor beschriebenen Methode zur Entwicklung von Domain Specific Languages. Es ist ein möglichst vollständiges, aber nicht funktionales Syntaxbeispiel, das den Informationsgehalt, den Aufbau und die Syntax der geplanten DSL aufzeigt.
Blueprint		Der *Blueprint* ist ein für die Softwarearchitektur und Implementation erweiterter *Domain Specific Prototype*; er liefert auf einer sehr abstrakten Ebene Informationen über die Klassen, die mit der DSL generiert oder beeinflusst werden müssen.
Useability Evaluation		*Useability Evaluation* ist ein Überbegriff für unterschiedliche Analyseverfahren zur Ermittlung der Benutzerfreundlichkeit eines Produktes für eine bestimmte Zielgruppe. Im Kontext der Bachelorarbeit bezieht sich die *Useability Evaluation* auf die Eigenschaften von Programmiersprachen, wie zum Beispiel ihre Übersichtlichkeit, Modularität oder Einsteigerfreundlichkeit.
LL(*) Algorithmus		Bezeichnet einen ***L**eft to right, **L**eftmost derivation* Algorithmus. *Leftmost derivation*, zu Deutsch *Linksableitung*, bedeutet, dass immer das am weitesten linksstehende nicht-terminale Symbol durch die Anwendung einer Produktionsregel ersetzt wird. *Left to right* bedeutet, dass die Zeichenketten von links nach rechts eingelesen werden.
Proposal File		Das *Proposal File* ist ein XML basiertes Dateiformat zur Beschreibung von neu anzulegenden Referenzen in einem Visual Studio Projektfile.
Program Core		Der *Program Core* ist eine Ansammlung von manuell programmierten, nicht generierten Artefakten für eine DSL (siehe Kapitel 3.2.8 „Transformation“).
Incremental Game Prototype		*Incremental Game Prototype* ist der Name des Pilotprojekts.
Partial Class Description		Die *Partial Class Description* ist ein XML basiertes Dateiformat zum Beschreiben von C# Klassen.
IKVM.NET		Eine freie Java Implementation für das Mono- und .Net-Framework. Der Name IKVM ist dabei ein Wortspiel zu JVM, bei dem das „J“ mit den beiden nächsten Buchstaben ersetzt wird.

Parser Tree und Abstract Syntax Tree

Beispielgrammatik[52] für eine Domain Specific Language:

```
grammar Expr; […]
prog:   stat+ ;

stat:   expr NEWLINE {System.out.println($expr.value);}
    |   ID '=' expr NEWLINE
        {memory.put($ID.text, new Integer($expr.value));}
    |   NEWLINE
    ;

expr returns [int value]
    :   e=multExpr {$value = $e.value;}
        (   '+' e=multExpr {$value += $e.value;}
        |   '-' e=multExpr {$value -= $e.value;}
        )*
    ;

multExpr returns [int value]
    :   e=atom {$value = $e.value;} ('*' e=atom {$value *= $e.value;})*
    ;

atom returns [int value]
    :   INT {$value = Integer.parseInt($INT.text);}
    |   ID
        {
        Integer v = (Integer)memory.get($ID.text);
        if ( v!=null ) $value = v.intValue();
        else System.err.println("undefined variable "+$ID.text);
        }
    |   '(' e=expr ')' {$value = $e.value;}
    ;

ID  :   ('a'..'z'|'A'..'Z')+ ;
INT :   '0'..'9'+ ;
NEWLINE:'\r'? '\n' ;
WS  :   (' '|'\t')+ {skip();} ;
```

Codebeispiel 10: Anhang: Beispielgrammatik für eine DSL

[52] Uhrsprung der Grammatik: http://www.antlr3.org/works/help/tutorial/calculator.html (Stand 19.09.2016)

Nun wird in der, anhand der Grammatik definierten, Sprache der folgende Ausdruck[53] festgelegt:

X = 1
Y = 2
(X + Y) * 3

Der Parser Tree stellt den Inhalt des Ausdrucks dann wie folgt dar (siehe Abbildung 24).

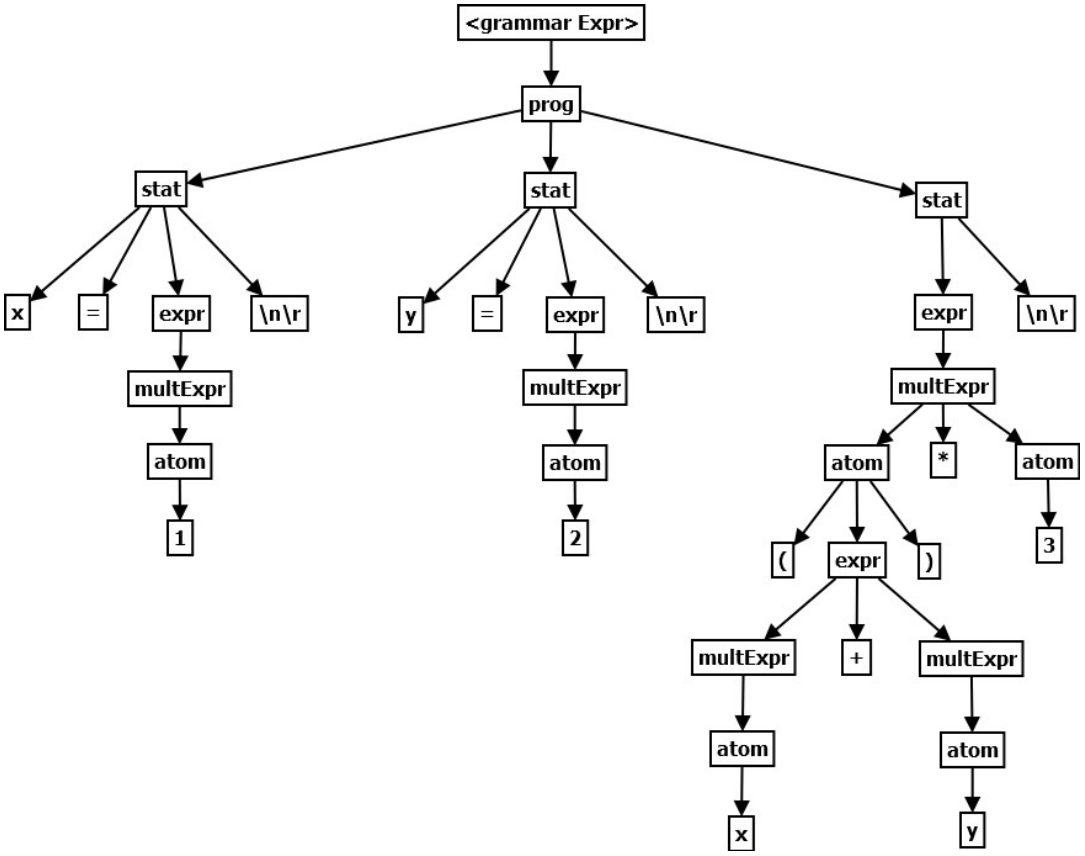

Abbildung 24: Anhang: Parse Tree (Eigene Darstellung)[54]

[53] Analog zum Beispiel von Guy Coder http://stackoverflow.com/questions/5026517/whats-the-difference-between-parse-tree-and-ast (zuletzt Besucht: 27.09.2016)

[54] Analog zum Beispiel von Guy Coder http://stackoverflow.com/questions/5026517/whats-the-difference-between-parse-tree-and-ast (zuletzt Besucht: 27.09.2016)

Der Abstrakte Syntax Tree des Ausdrucks hat indessen die Form von Abbildung 25.

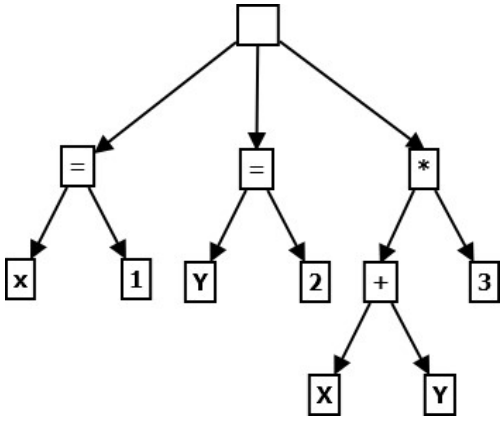

Abbildung 25: Anhang: Abstract Syntax Tree (Eigene Darstellung)[55]

[55] Analog zum Beispiel von Guy Coder http://stackoverflow.com/questions/5026517/whats-the-difference-between-parse-tree-and-ast (zuletzt Besucht: 27.09.2016)

Visual Studio DSL Tools

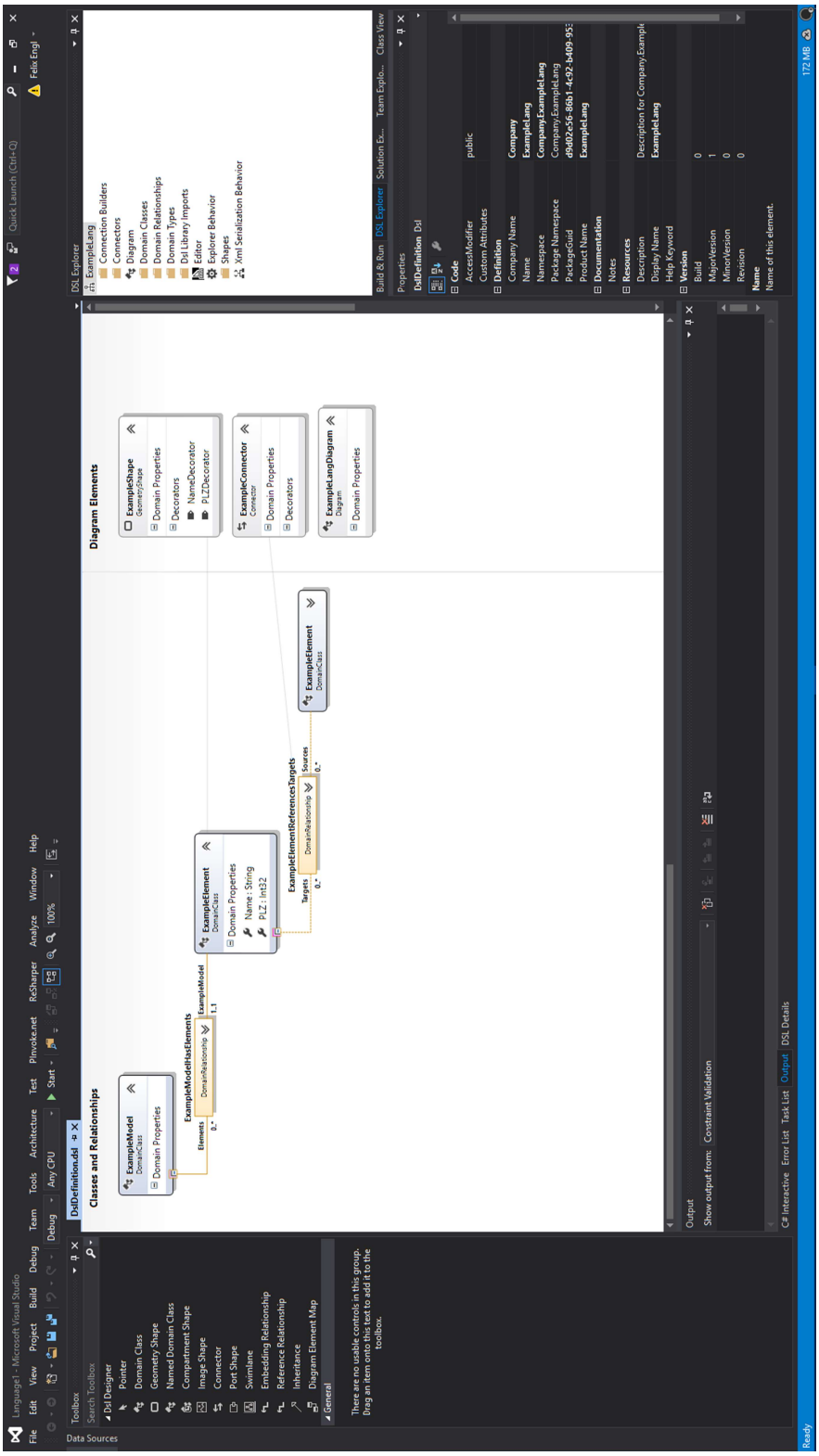

Abbildung 26: Anhang: Visual Studio DSL Tools Screenshot

Liste der einzelnen Elemente der Visual Studio DSL Tool GUI (siehe Abbildung 26):

- *Toolbox*:
  Alle Sprachelemente der Editor DSL, mit denen die Grammatik der Grafischen DSL festgelegt werden kann.
- *Classes und Relationships*:
  Bereich für die Festlegung aller DSL Klassen und deren Beziehungen untereinander.
- *Diagram Elements*:
  Modifikationen für die Darstellung der einzelnen Instanzen der DSL Klassen im Editor der fertigen DSL.
- *DSL Explorer*:
  Explorer für die gesamte DSL.

Project Files Aufbau

Im Folgenden werden die XML-Strukturen der beiden Files ClickerProject.clproj und GeneratorPaths.clproj in UML Diagrammen dargestellt. Attribute der einzelnen Knoten werden wie Klassen-Attribute dargestellt, ein leeres Attribute-Feld bedeutet, dass der Knoten keine Attribute hat. Sollte das Attribut-Feld fehlen wird der Knoten als eigenes Element in einer anderen UML Dargestellt.

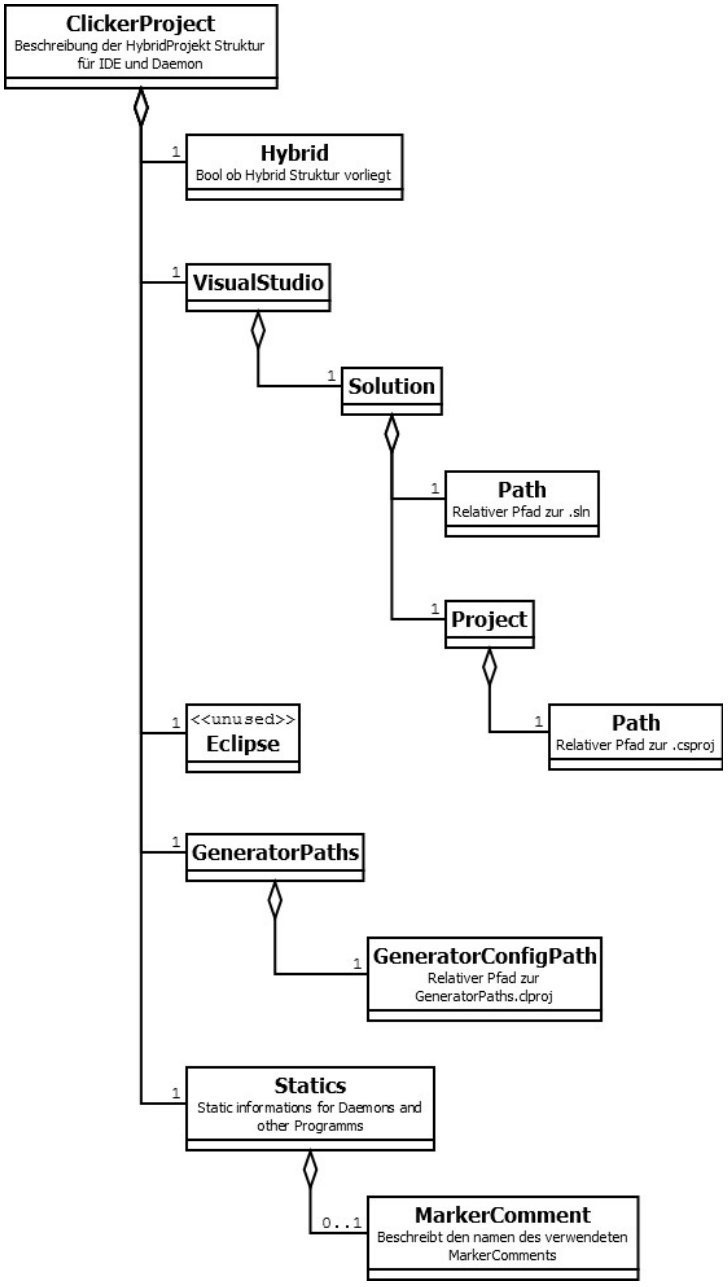

Abbildung 27: Anhang: Struktur ClickerProject File (Eigene Darstellung)

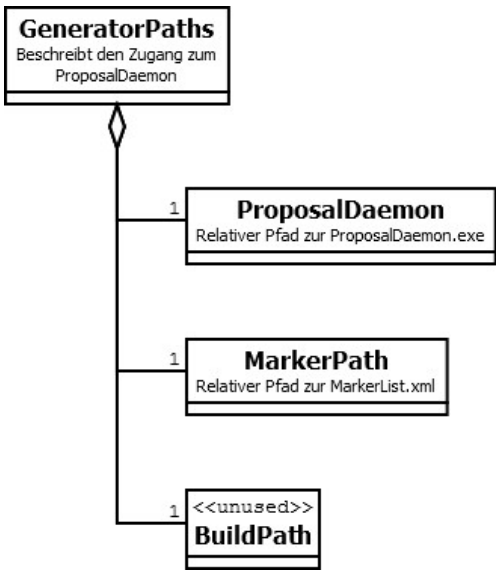

Abbildung 28: Anhang: Struktur GeneratorPaths File (Eigene Darstellung)

Proposal Aufbau

Ergänzend zu Abbildung 12 in Kapitel 6.4.2.2 „Programmablauf“ werden im Folgenden die beiden Knoten *Proposal* und *Combine* in ihrer XML-Struktur beschrieben.

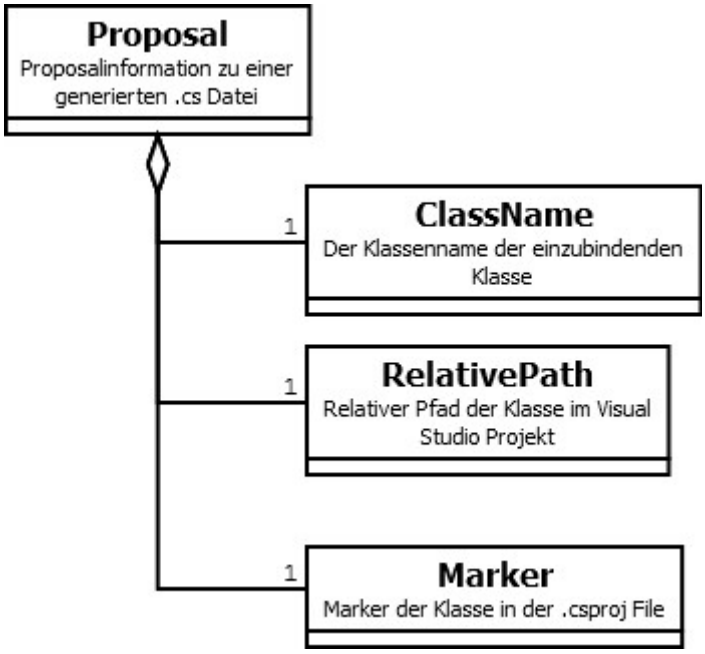

Abbildung 29: Anhang: XML-Struktur Proposal (Eigene Darstellung)

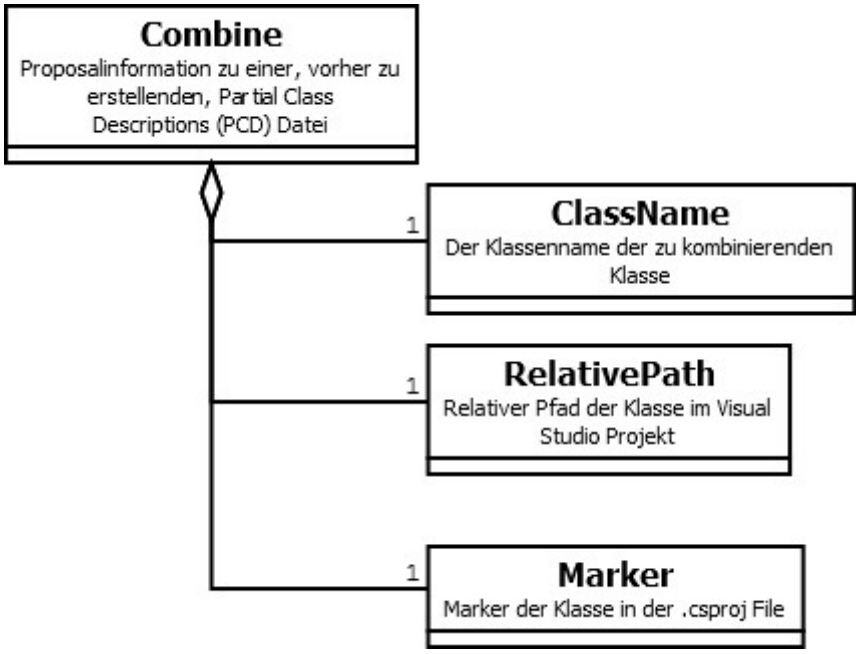

Abbildung 30: Anhang: XML-Struktur Combine (Eigene Darstellung)

Kanban Board

Die erste Abbildung (siehe Abbildung 31) zeigt den gesamten Aufbau des Kanban Boards für das Pilotprojekt „Incremental Game Prototyp DSL“.

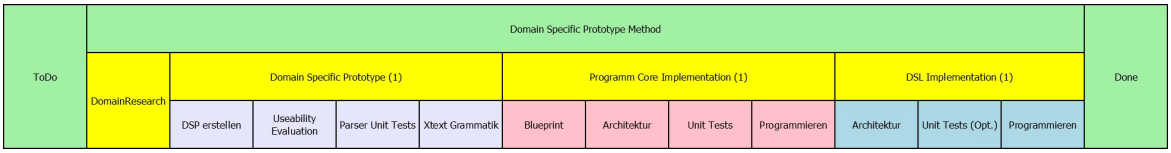

Abbildung 31: Anhang: Kanban Board (vollst.) (Eigene Darstellung)

Abbildung 32: Anhang: Kanban Board (Ebene 1) (Eigene Darstellung)

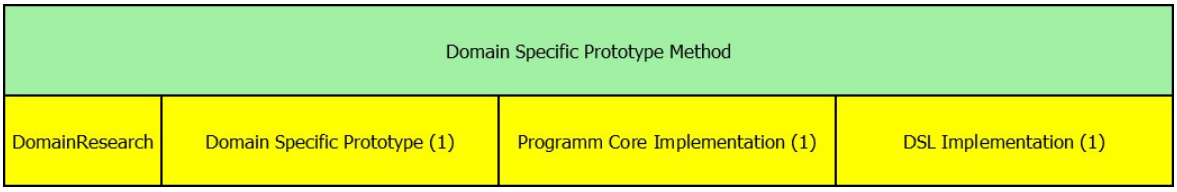

Abbildung 33: Anhang: Kanban Board (Ebene 2) (Eigene Darstellung)

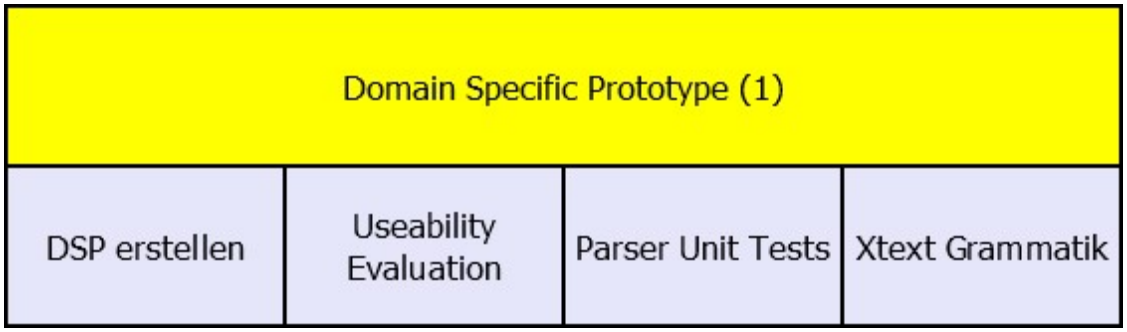

Abbildung 34: Anhang: Kanban Board (Ebene 3A) (Eigene Darstellung)

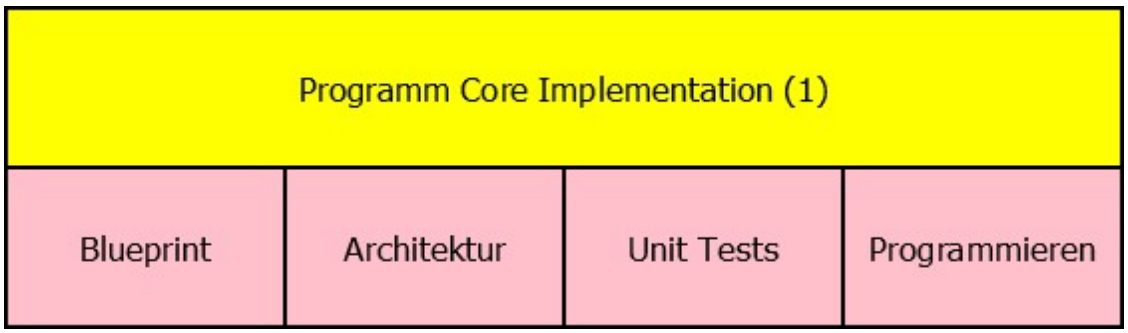

Abbildung 35: Anhang: Kanban Board (Ebene 3B) (Eigene Darstellung)

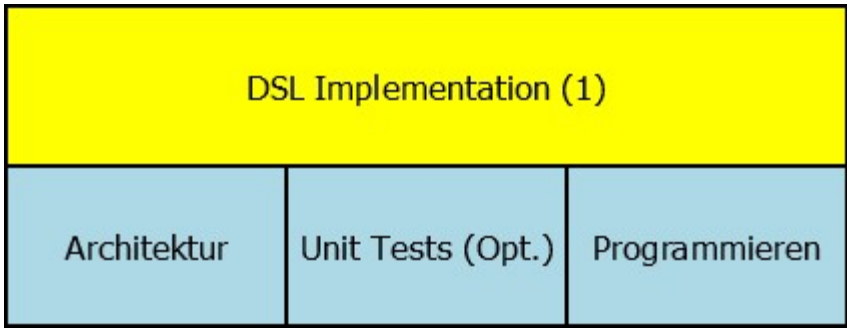

Abbildung 36: Anhang: Kanban Board (Ebene 3C) (Eigene Darstellung)

Hybrid Projekt Folder Tree

Der nachfolgende Textblock ist eine ASCII Darstellung des gesamten Hybrid Projekts. Die vollständige Grafik von Kapitel „6.4.1 Ordnerstruktur“ ist auf der beiliegenden CD als Vektorgrafik unter /Images/HybridProjektStruktur.SVG auffindbar.

```
[RootFolder]
|   ClickerProject.clproj
|   GeneratorPaths.clproj
|
+---ClickerProject
|   \---ClickerGame
|       |
|       +---ClickerGame
|       |   ClickerGame.csproj
|       |
|       \---ProposalDaemon
|           |
|           \---bin
|               \---Release
|                       MarkerList.xml
|                       ProposalDaemon.exe
|
\---IdleDSL
    |   .project
    |
    \---[ProjectFolder]
        |   .project
        |   *.incre
        |
        \---src-gen
            \---CSharp
                |   *.prop
                |
                +---Currency
                |       [Currency].cs
                |
                +---Development
                |   |   [Development].cs
                |   |
                |   \---Price
```

```
|         [DevelopmentPrice].cs
|
+---Generator
|   |   [GeneratorName].cs
|   |
|   +---GeneratorOrder
|   |       [GeneratorOrder].cs
|   |
|   +---Influence
|   |       [OrderInfluence].cs
|   |
|   \---Price
|           [GeneratorPrice].cs
|
+---Generic
|       Generated*.cs
|       g_*.prop.part
|       Program.cs
|       Proposal_*.prop
|
+---Player
|       GeneratedPlayer.cs
|
\---Transform
        [Transform].cs
```

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Luhe-Wildenau, den 28.09.2016

Felix Engl