



B A C H E L O R A R B E I T

in der Fachrichtung
Wirtschaftsinformatik

T H E M A

Konzeption einer DSL zur Beschreibung von Benutzeroberflächen für profil c/s auf der Grundlage des Multichannel-Frameworks der deg

Eingereicht von:	Niels Gundermann (Matrikelnr. 5023) Willi-Bredel-Straße 26 17034 Neubrandenburg E-Mail: gundermann.niels.ng@googlemail.com
Erarbeitet im:	7. Semester
Abgabetermin:	16. Februar 2015
Gutachter:	Prof. Dr.-Ing. Johannes Brauer
Co-Gutachter:	Prof. Dr. Joachim Sauer
Betrieblicher Gutachter:	Dipl.-Ing. Stefan Post Woldegker Straße 12 17033 Neubrandenburg Tel.: 0395/5630553 E-Mail: stefan.post@data-experts.de

Inhaltsverzeichnis

Abbildungsverzeichnis	ii
Tabellenverzeichnis	iii
Listings	v
1 Motivation	1
2 Problembeschreibung und Zielsetzung	3
2.1 Allgemeine Anforderungen an Benutzeroberflächen von pro- fil c/s	3
2.2 Umsetzung der Benutzerschnittstellen für mehreren Plattfor- men in der deg (Ist-Zustand)	5
2.3 Probleme des Multichannel-Frameworks	6
2.4 Zielsetzung	7
3 Domänenspezifische Sprachen	8
3.1 Begriffsbestimmungen	8
3.2 Anwendungsbeispiele	13
3.3 Model-Driven Software Development (MDSD)	13
3.4 Abgrenzung zu GPL	15
3.5 Vor- und Nachteile von DSL gegenüber GPL	15
3.6 Interne DSL	22
3.6.1 Implementierungstechniken	23
3.7 Externe DSL	25
3.7.1 Implementierungstechniken	25
3.8 Nicht-Textuelle DSL	29

4	Entwicklung einer Lösungsidee	30
4.1	Allgemeine Beschreibung der Lösungsidee	30
4.2	Architektur	30
4.3	Vorteile gegenüber dem Multichannel-Framework	31
5	Anforderung an die GUI-DSL	32
6	Evaluation des Frameworks zur Entwicklung der DSL	35
6.1	Vorstellung ausgewählter Frameworks	35
6.1.1	PetitParser	35
6.1.2	Xtext	36
6.1.3	MPS	37
6.2	Vergleich und Bewertung der vorgestellten Frameworks . . .	37
7	Aufteilung der Anforderungen auf Sprache und Generator	38
7.1	Anforderung an die neue DSL	38
7.2	Anforderung an den Generators	38
8	Entwicklung einer DSL zur Beschreibung der GUI in profil c/s	39
8.1	Analyse der Metadaten der GUI	39
8.2	GUI-Beschreibungs-Model zur Verarbeitung im Generator . .	44
8.3	Konkrete Syntax	50
9	Entwicklung des Generators für das Generieren von Klassen für das Multichannel-Framework	55
9.1	WAM-GUI Architektur	55
9.2	Syntax und Semantik für die Beschreibung der GUIs	56
9.3	Umsetzung des frameworkspezifischen Generators	56
10	Zusammenfassung und Ausblick	57
	Anhang	xi
	Glossar	xv
	Literaturverzeichnis	xvi

Abbildungsverzeichnis

2.1	Web-Client	4
2.2	Standalone-Client	5
2.3	MC-Framework	6
3.1	mdsd	14
3.2	Parsing	26
3.3	parsercombinator	28
4.1	neuerAnsatz	31
8.1	Teil1-1	44
8.2	Teil2-1	45
8.3	Teil1-2	46
8.4	Teil2-2	47
8.5	Teil3-2	48
8.6	Teil4-2	49
8.7	Teil5-2	50

Tabellenverzeichnis

3.1 Implementierung einfacher Grammatikregeln mit einem RD-Parser [FP11, S.248]	27
---	----

Listings

3.1	Beispiel: Fluent Interface	24
8.1	Syntax Version 1	50
8.2	Properties in Version 2	51
8.3	Interaktion in Version 2	52
8.4	Komplexe Komponenten in Version 2	52
8.5	Area-Zuweisung Möglichkeit 1 Version 2	52
8.6	Area-Zuweisung Möglichkeit 2 Version 2	53
8.7	Überschreiben einer eingebundenen Komponente Version 2	53
8.8	Überschreiben einer eingebundenen Komponente mit Titel der Komponente Version 2	54

Kapitel 1

Motivation

In der heutigen Zeit werden Programme auf vielen unterschiedlichen Geräten (bspw. Desktop, Smartphone, Tablet) ausgeführt. Die Usability¹ ist ein wichtiger Faktor bei der Entwicklung einer Anwendung. Denn eine [...] *schlechte Useability führt zu Verwirrung und Miss- bzw. Unverständnis beim Kunden* [Use12]. Dadurch geht letztendlich Umsatz verloren. Die Usability wird hauptsächlich von der Benutzerschnittstelle bestimmt. Folglich ist die Benutzeroberfläche neben der internen Umsetzung immer ein wichtiger Faktor, der für den Erfolg einer Anwendung eine große Rolle spielt. [LW]

Wenn ein Programm auf unterschiedlichen Geräten ausgeführt wird, muss der Entwickler bei der traditionellen Entwicklung² mehrere Graphical User Interfaces (GUI)³ manuell implementieren. Folglich werden mehrere GUIs mit unterschiedlichen Toolkits oder Frameworks entworfen. Diese Frameworks haben einen starken imperativen Charakter, sind schwer zu erweitern und sie verhalten sich je nach Plattform unterschiedlich. [KB11] Daraus folgt, dass Entwickler bei dem traditionellen Ansatz das GUI für jedes Framework explizit beschreiben muss.

Ein anderer Ansatz zur Beschreibung von Benutzeroberflächen ist das Model-Driven Development (siehe Kapitel 3.3). Damit sollen bspw. UIs anhand der modellierten Funktionalitäten automatisch erzeugt werden. [SKNH05] Laut Myers et. al. wurde die Darstellung dieser generierten UIs in der Vergangenheit von der Darstellung traditionell implementierter Benutzerschnittstellen

¹Siehe Glossar: Usability

²Siehe Glossar: Traditionelle UI-Entwicklung

³Siehe Glossar: GUI

übertraffen. [MHP99] Grund dafür ist, dass die Abstraktionsebene, auf der die GUIs beschrieben wurden, oft nicht an die Gestaltung der GUI, sondern an fachliche Konzepte der Domäne angepasst wurde.

Daraus ergibt sich die Überlegung, ob diese beiden Ansätze zur Implementierung von UIs (traditionell und Model-Driven) verbunden werden können (kombinierter Ansatz). Somit könnte die genaue Beschreibung der Darstellung mit einer höheren Abstraktion verbunden werden. Dieser Versuch wurde bspw. von Baciková im Jahr 2013 (siehe [BPL13]) unternommen. Dort wurde nachgewiesen, dass ein GUI eine Sprache definiert.

In dieser Arbeit wird versucht den kombinierten Ansatz in einem anderen Fachbereich umzusetzen, um die Unterstützung eines GUIs auf unterschiedlichen Plattformen zu gewährleisten. Bei der Umsetzung wird sich auf die UIs der Anwendung *profil c/s*. Profil c/s ist eine JEE⁴-Anwendung die InVeKoS⁵ umsetzt und von der deg entwickelt wird.

⁴Siehe Glossar: JEE

⁵Siehe Glossar: InVeKoS

Kapitel 2

Problembeschreibung und Zielsetzung

2.1 Allgemeine Anforderungen an Benutzeroberflächen von profil c/s

Die wichtigste (primäre) Anforderung für diese Arbeit bezieht sich auf den Client von profil c/s. Dieser soll sowohl in Web-Browsern (Web-Client) als auch standalone auf einem PC (Standalone-Client) ausgeführt werden können.

Um eine effiziente Arbeitsweise zu ermöglichen, kamen (sekundäre) Anforderungen wie *Erweiterbarkeit der Frameworks*, *Abstraktion* und die *Ausdrucks-kraft*¹ der Sprachkonstrukte, die zur Entwicklung verwendet werden.

In Abbildung 2.1 und Abbildung 2.2 ist das GUI eines Zuwendungsblatt² eines Förderantrag³ zu sehen.

¹Siehe Glossar: Ausdruckskraft

²Siehe Glossar: Zuwendungsblatt

³Siehe Glossar: Förderantrag

3801 Investitionen in touristische Infrastrukturprojekte/2010: 07000000000004/70100004 as1 rpeler Az: 3801100000008

Antragsmappe Bearbeiten Aktionen Hilfe

Inhalt

- 3801 Investitionen in touristische Infrastrukturprojekte/2010: 07000000000004/70100004 as1 rpeler Az: 3801100000008
 - (1) Förderantrag
 - Kosten- und Finanzierungsmappe
 - Zuwendungsübersicht
 - Bewilligungsblatt
 - WVKProtokoll
 - (2) Rücknahme
 - (3) Zahlungsantrag
 - Inaugenscheinnehmende
 - Protokoll

Verweise

- Antragstellermappe
- AS-Zahlungsübersicht
- Förderungsmappe
- 2007
- 2008
- 2009
- 2010
- 2011

Anteils-Festbetragsfinanzierung **Mengenfinanzierung**

Förderfähige Ausgaben lt. Kostenplan: 50.000,00

Teilvorhaben Nr.	Fördergegenstand mit Fördersatz	Ft. Ausgaben lt. Amt [EUR]	Davon lt. MwSt. [EUR]	Finanzierungsart	Berechneter Bew. betrag [EUR]	Tatsächl. Fördersatz [%]	Abzug [EUR]	Zuwendung lt. Amt [EUR]	Davon Zuwendung MwSt. [EUR]
1	Förderung von Investitionen in touristische Infrastrukturprojekte - 80,00%	50.000,00	0,00	A	40.000,00	80,00	0,00	40.000,00	0,00
Gesamt		50.000,00	0,00		40.000,00	80,00	0,00	40.000,00	0,00

Neu Löschen

Bemerkungen:

Kopie

led665-1 / ES - Rheinland-Pfalz (Amt 1)

Abbildung 2.1: Web-Client: Zuwendungsblatt [deG07]

Fördergegenstand mit Fördersatz	ff. Ausgaben lt. Amt [EUR]	Finanzierungsart	Berechneter Bew.betrag [EUR]	Tatsächl. Fördersatz [%]	Abzug [EUR]	Zuwendung lt. Amt [EUR]
Erweiterung vereinseigener Sportstätten - 75,0...	50.000,00	A	37.500,00	75,00	0,00	37.500,00
Ausnahmen - 30,00%	20.000,00	A	6.000,00	30,00	0,00	6.000,00
Neubau kommunaler Sportstätten - 75,00%	90.000,00	A	67.500,00	75,00	0,00	67.500,00
Modern. vereinseigener Sportstätten - 75,00%	80.000,00	A	60.000,00	75,00	0,00	60.000,00
Instand. vereinseigener Sportstätten - 75,00%	50.000,00	A	37.500,00	75,00	0,00	37.500,00
Gesamt	290.000,00		208.500,00	71,90	0,00	208.500,00

Abbildung 2.2: Standalone-Client: Zuwendungsblatt [deG07]

Für den Aufbau sind nur die Tabelle und die darunter stehenden Buttons, sowies das Bemerkungsfeld (im Web-Client auf der rechten Seite und im Standalone-Client in der Mitte) von Bedeutung. Dass der Aufbau der GUI in beiden Clients ähnlich ist, liegt an der Umsetzung der GUI.

2.2 Umsetzung der Benutzerschnittstellen für mehreren Plattformen in der deg (Ist-Zustand)

Für die Umsetzung der primären Anforderung wäre es möglich gewesen für den Web-Client und dem Standalone-Client separate GUIs mit unterschiedlichen Frameworks zu entwickeln. Die deg hat jedoch eine Lösung erarbeitet mit der es möglich ist, ein einmal beschriebenes GUI auf mehrere Plattformen zu portieren. Das reduziert den Aufwand zur Entwicklung neuer GUIs, durch eine höhere Abstraktion. Zugleich fördert die einmalige Beschreibung auch einen ähnlichen Aufbau der GUI im Web- und

Standalone-Client. Die Lösung der deg ist das *Multichannel-Framework* (MCF). Die Architektur des Multichannel-Frameworks ist Abbildung 2.3 zu entnehmen. Innerhalb dieses Frameworks werden die GUIs mittels so genannter *Präsentationsformen* beschreiben.

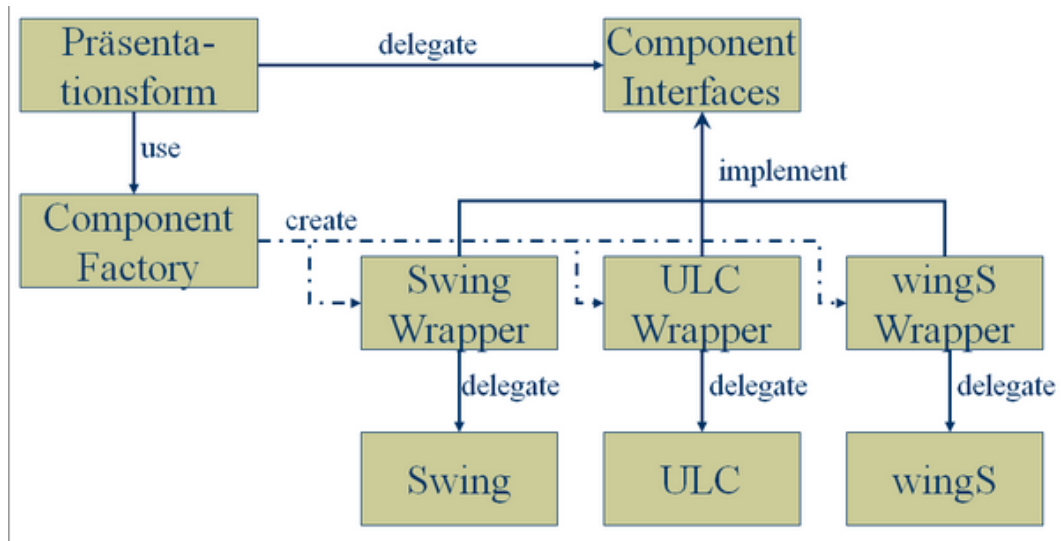


Abbildung 2.3: Architektur des Multichannel-Frameworks [Ste07]

Aus Präsentationsformen mithilfe der *Component-Factories* GUIs erzeugt werden, die auf unterschiedlichen Frameworks basieren⁴ und das *Component-Interface* implementieren. Das *Component-Interface* wird für die Interaktion mit den Komponenten der unterschiedlichen Frameworks benötigt. Mit dem MCF ist die deg in der Lage ihre GUIs für das *Swing*⁵-Framework und für das *wingS*⁶-Framework mit nur einer GUI-Beschreibung zu erzeugen.

2.3 Probleme des Multichannel-Frameworks

Beim Einsatz des MCF treten jedoch Probleme auf. Das erste Problem bezieht sich auf die integrierten Frameworks (Swing und wingS). Beide Frameworks sind verwaltet und werden nicht mehr gewartet. Um auch in der Zukunft den Anforderungen der Kunden nachkommen zu können müssten beide Frameworks von den Entwicklern der deg selbst weiterentwickelt werden. Eine andere Möglichkeit wäre es, wenn die deg andere und moder-

⁴Hier: Swing, ULC und WingS. Wobei ULC bei der deg nicht mehr im Einsatz ist.

⁵Siehe Glossar: Swing

⁶Siehe Glossar: wingS

nere Frameworks einsetzt um den nötigen Support der Framework-Entwickler nutzen zu können.

Das MCF ist in der Theorie so konzipiert, dass es leicht sein sollte neue Frameworks zu integrieren (siehe Abbildung 2.3). In der Praxis wurde die Einfachheit einer solchen Integration jedoch widerlegt. Ein Problem, welches bei der Integration neuer Frameworks aufkommt, ist, dass sich das MCF sehr stark an Swing orientiert und die GUIs vor allem vom GridBagLayout⁷ stark beeinflusst sind. Ein solches Layout steht nicht in allen Frameworks zur Verfügung. Da die Beschreibung der GUI über ein solches Layout vollzogen wird, ist es der Umgang mit dem GridBagLayout⁸ innerhalb des Frameworks eine Voraussetzung für die Integration in das MCF. Zusammenfassend sind folgende Probleme des MCF zu nennen:

- verwendeten Frameworks sind inaktuell
- Starke Orientierung an Swing

Routinearbeiten

-

2.4 Zielsetzung

Das langfristige Ziel der deg bzgl. des MCF ist es, eine Lösung zu entwickeln, welche das MCF ablösen kann. Anzustreben ist eine Lösung, die neben den primären Anforderungen die sekundären Anforderung besser umsetzt als das MCF.

In dieser Arbeit wird ein Ansatz untersucht, bei dem es möglich ist, die oben genannten sekundären Anforderungen besser umzusetzen. Kern des Ansatzes ist eine DSL, mit deren Hilfe die UIs beschrieben werden sollen. Eine DSLs kann so konzipiert werden, dass sie ausreichend abstrakt, erweiterbar und ausdrückstärker ist als das MCF. Die primären Anforderungen dürfen dabei nicht außer Acht gelassen werden.

Die genaue Lösungsidee mittels DSL, welche in dieser Arbeit verfolgt wird, ist in Kapitel 4 beschrieben.

⁷Siehe Glossar: GridBagLayout

⁸Siehe Glossar: GridBagLayout

Kapitel 3

Domänenspezifische Sprachen

3.1 Begriffsbestimmungen

Sprache/Programmiersprache

Rein formal betrachtet ist eine Sprache ist eine beliebige Teilmenge aller Wörter über einem Alphabet. *Ein Alphabet ist eine endliche, nichtleere Menge von Zeichen (auch Symbole oder Buchstaben genannt)* [Hed12, S.6]. Zur Verdeutlichung der Definition einer Sprache sein V ein Alphabet und $k \in \mathbb{N}^1$. *Eine endliche Folge (x_1, \dots, x_k) mit $x_i \in V (i = 1, \dots, k)$ heißt Wort über V der Länge k* [Hed12, S.6].

Bei Programmiersprachen grenzt die Bestandteile einer Sprache wie folgt ab:

abstrakte Syntax

Die abstrakte Syntax ist eine Datenstruktur, welche die Kerninformationen eines Programms beschreibt. Sie enthält keinerlei Informationen über Details bezüglich der Notation. Zur Darstellung dieser Datenstruktur werden abstrakte Syntaxbäume genutzt. [VBK⁺13, S.179].

¹ \mathbb{N} ist die Menge der natürlichen Zahlen einschließlich der Null. [Hed12, S. 6]

konkrete Syntax

Die konkrete Syntax beschreibt die Notation der Sprache. Demnach bestimmt sie, welche Sprachkonstrukte der Nutzer einsetzen kann, um ein Programm in dieser Sprache zu schreiben. Die konkrete Syntax wird in so genannten Parse-Bäumen (konkrete Syntaxbäume) dargestellt. [Aho08, S.87]

statische Semantik

Die statische Semantik beschreibt die Menge an Regeln bezüglich des Typ-Systems, die ein Programm befolgen muss. [VBK⁺13, S.26]

ausführbare Semantik

Die ausführbare Semantik ist abhängig vom Compiler. Sie beschreibt wie ein Programm zu seiner Ausführung funktioniert. [VBK⁺13, S.26]

Programmiersprachen werden dazu verwendet, um mit einem Computer Instruktionen zukommen zu lassen. [FP11, S.27] [VBK⁺13, S.27]

General Purpose Language (GPL)

Bei GPLs handelt es sich um Programmiersprachen, die Turing-vollständig sind. Das bedeutet, dass mit einer GPL alles berechnet werden kann, was auch mit einer Turing-Maschine² berechenbar ist. Völter et. Al. behaupten, dass alle GPLs aufgrund dessen untereinander austauschbar. Dennoch sind Abstufungen bei der Ausführung dieser Programmiersprachen zu machen. Unterschiedliche GPL sind für spezielle Aufgaben optimiert. [VBK⁺13, S.27f]

²Siehe Glossar: Turing-Maschine

Domain Specific Language (DSL)

Eine DSL ist eine Programmiersprache, welche für eine bestimmte Domain³ optimiert ist. [VBK⁺13, S.28] Das Entwickeln einer DSL ermöglicht es, die Abstraktion der Sprache der Domäne anzupassen. [gho11, S.10] Das bedeutet, dass Aspekte, welche für die Domäne unwichtig sind, auch von der Sprache außer Acht gelassen werden können (Abstraktion). Die Semantik und Syntax sollten dieser Abstraktionsebene angepasst sein. Darüber hinaus sollte ein Programm, welches in einer DSL geschrieben wurde, alle Qualitätsanforderungen erfüllen, die auch bei einer Umsetzung des Programms mit anderen Programmiersprachen realisiert werden. [gho11, S.10f] Eine DSL ist demnach in ihren Ausdrucksmöglichkeiten eingeschränkt. Je stärker diese Einschränkung ist, desto besser ist die Unterstützung der Domäne sowie die Ausdruckskraft der DSL. [FP11, S.27f] In manchen Fällen findet eine Unterscheidung zwischen technischen DSLs und fachlichen DSL statt. Markus Völter unterscheidet diese beiden Kategorien im Allgemeinen dahingehend, dass technische DSLs von Programmierern genutzt werden und fachliche DSL von Personen, die keine Programmierer sind (bspw. Kunden bzw. Personen, die sich in der Domäne auskennen). [VBK⁺13, S.26]

Grammatik

Grammatiken und insbesondere Grammatikregeln können dazu verwendet um Sprachen zu beschreiben und somit auch den Aufbau eines Computerprogramms. [Hed12, S.23f] Für die Definition einer Grammatik verweise ich auf den Praxisbericht [Gun14, S.5ff]. Grammatiken können in einer Hierarchie dargestellt werden (*Chomsky-Hierarchie*). [S.32f] Bei Programmiersprachen handelt es sich dabei um *kontextfreie Sprachen*, da diese Sprachen entscheidbar sind und somit von einem Compiler⁴ verarbeitet werden. [Hed12, S. 16f]

³Siehe Glossar: Domäne

⁴Siehe Glossar: Compiler

Lexikalische Analyse

Bevor ein DSL-Script verarbeitet werden kann, muss es vom so genannten *Lexer* oder *Scanner* gelesen werden. [FP11, S.221] Dabei wird ein Text (DSL-Script) als Input-Stream betrachtet. Der Lexer wandelt diesen Input-Stream in einzelne Tokens. vgl. [gho11, S.220] Allgemein ist der Lexer die Instanz innerhalb der Infrastruktur einer DSL, die für das Auslesen des DSL-Scriptes verantwortlich ist.

Parser

Ein Parser ist ein Teil der Infrastruktur der DSL. [gho11, S.211] Er ist dafür verantwortlich aus dem DSL-Script ein Output zu generieren, mit dem weitere Aktionen durchgeführt werden können. [gho11, S.212] Der Output wird in Form eines Syntax-Baums (Parse-Baum) (AST⁵) generiert. [FP11, S.47] Ein solcher Baum ist laut Martin Fowler eine weitaus nutzbarere Darstellung dessen, was mit dem DSL-Script dargestellt werden soll. Daraus lässt sich auch das semantische Model generieren. [FP11, S.48]

Semantisches Model

Das semantische Model ist eine Repräsentation dessen, was mit der DSL beschrieben wurde. Es wird laut Martin Fowler auch als das Framework oder die Bibliothek betrachtet, welche von der DSL nach außen hin sichtbar ist. [FP11, S.159] In Anlehnung an Ghosh ist das semantische Model mit dem AST gleichzusetzen, der durch eine lexikalische Analyse des DSL-Scripts mithilfe eines Parsers erzeugt wird. Somit wird es als Datenstruktur betrachtet, dessen Struktur von der Syntax der DSL unabhängig ist [gho11, S.214]. Das Gleichsetzen des semantischen Models mit dem AST ist laut Martin Fowler in den meisten Fällen nicht effektiv. Grund dafür ist, dass der AST sehr stark an die Syntax der DSL gebunden, wohingegen das semantische Model von der Syntax unabhängig ist. [FP11, S.48]

⁵Siehe Glossar: Abstrakter Syntax Baum

Generator

In Anlehnung an Martin Fowler ist ein Generator ist ein Teil der einer DSL Umgebung⁶, der für das Erzeugen von Quellcode für die Zielumgebung⁷ zuständig ist. [FP11, S.121] Bei der Generierung von Code wird zwischen zwei Arten unterschieden.

Transformer Generation

Bei der Transformer Generation wird das semantische Model als Input verwendet, woraus Quellcode für die Zielumgebung generiert wird. [FP11, S.533f] Eine solte Generation wird oft verwendet, wenn ein Großteil des Output generiert wird und die Inhalte des semantischen Models einfach in den Quellcode der Zielumgebung überführt werden können. [FP11, S.535]

Templated Generation

Bei der Templated Generation wird eine Vorlage benötigt. In dieser Vorlage befinden sich Platzhalter, an deren Stelle der Generator speziellen Code generiert. [FP11, S.539f] Diese Art der Cod degenerierung wird oft verwendet, wenn sich in der generierte Quellcode für die Zielumgebung viele statische Inhalte befinden und der dynamisch generierte Anteil sehr einfach gehalten ist. [FP11, S.541]

⁶Siehe Glossar: DSL Umgebung

⁷Siehe Glossar: Zielumgebung

3.2 Anwendungsbeispiele

Die Anwendungsbereiche für DSLs sind sehr unterschiedlich. Die bekanntesten DSL sind Sprachen wie *SQL* (zur Abfrage und Manipulation von Daten in einer relationalen Datenbank), *HTML* (als Markup-Sprache für das Web) oder *CSS* (als Layoutbeschreibung). [gho11, S.12] Alle Sprachen besitzen eine eingeschränkte Ausdrucksmöglichkeiten und sind von der Abstraktion her direkt auf eine Domäne (jeweils dahinter in Klammern genannt) zugeschnitten. [gho11, S.12f]

Weitere Beispiele für DSL befinden sich im Bereich der Sprachen für Parser-Generatoren (*YACC*, *ANTLR*) oder im Bereich der Sprachen für das Zusammenbauen von Softwaresystemen (*Ant*, *Make*). [gho11, S.12]

Für den Bereich der UI-Entwicklung gibt es ebenfalls Anwendungsbeispiele. Diese werden in Kapitel 5 genauer beleuchtet.

3.3 Model-Driven Software Development (MDSD)

In der Einleitung wurde schon der Model-Driven Ansatz in Verbindung mit UI-Entwicklung erwähnt. Dieser Ansatz versucht den technischen Lösungen der IT-Industrie einen gewissen Grad an Agilität zu verleihen. [SKNH05] Das ist damit verbunden, dass die Produktion von Softwareprodukten schneller und besser von statten geht und mit weniger Kosten verbunden ist. [DM14, S.71] Erreicht wird dies indem die Modelle formaler, strenger, vollständiger und konsistenter beschrieben werden. [VBK⁺13, S.31] Die Kernidee ist, dass die Modelle Quellcode oder Funktionalitäten beschreiben und diese in der Evolution der Software immer wiederverwendet werden können. [DM14, S.72] Somit wird wiederkehrender oder schematischer Quellcode vermieden und es ist möglich diese Modelle auch in anderen Anwendungen zu verwenden. [DM14, S.71] Daraus lassen sich folgende Ziele des MDSD ableiten:

- schnelleres Entwickeln durch Automatisierungen
- bessere Softwarequalität durch automatisierte Transformationen (Generation) und formalen Model-Definitionen

- Verhinderung von Wiederholungen und besseres Management von veränderbare Technologien durch die Trennung der Funktionsbereiche (Separation of Concerns).
- Architekturen, Modellierungssprachen⁸ und Generatoren/Transformatoren können besser wiederverwendet werden
- Verringerte Komplexität durch höhere Abstraktion

[mds06, S.13f]

Die Modelle sind somit nicht länger nur zur Dokumentation geeignet. Sie sind Teil der Software. [mds06, S.14f] Die Modelle sind dabei auf ein bestimmtes Domänenproblem angepasst. Um diese Modelle zu beschreiben wird eine DSL benötigt. [mds06, S.15] In Abbildung 3.3 ist die Idee des MDSD schematisch dargestellt.

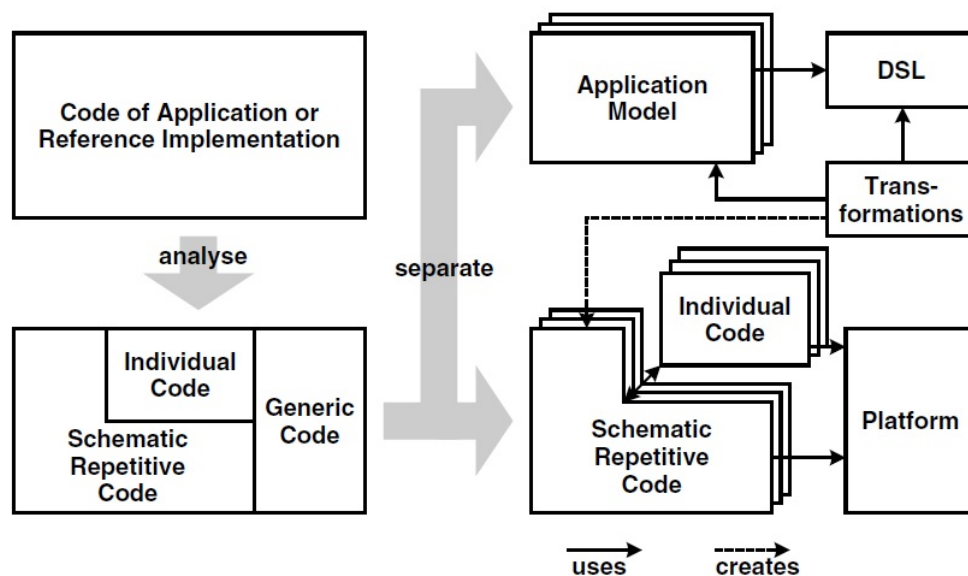


Abbildung 3.1: Die grundlegenden Ideen hinter dem MDSD [mds06, S.15]

⁸bspw. eine DSL

3.4 Abgrenzung zu GPL

Wie zu Beginn dieses Kapitels schon erwähnt sind GPLs. Sprachen mit denen alles Berechnet werden kann, was auch mit einer Turing-Maschine berechenbar ist. [VBK⁺13, S.27] Folglich kann mit einer GPL jedes berechenbare Problem gelöst werden. Eine DSL hat diese Eigenschaft nicht. Da sie auf eine bestimmte Domäne zugeschnitten ist, können auch nur Probleme innerhalb dieser Domäne mit ihr gelöst werden. [VBK⁺13, S.28] Martin Fowler bezeichnet diese Eigenschaft des Domänen Fokus als ein Schlüsselement der Definition einer DSL. [FP11, S.27f]

3.5 Vor- und Nachteile von DSL gegenüber GPL

Vorteile

- **Ausdruckskraft**

Laut Ghosh sollten DSLs so umgesetzt werden, dass sie präzise sind. Das bedeutet, dass eine DSL zu sehen, zu betrachten, vorzustellen und es zu zeigen ist.⁹ Weiterhin ist es wichtig bei der Entwicklung einer DSL darauf zu achten, dass sich die Abstraktion der Sprache präzise an der semantik der Domäne orientiert. [gho11, S.20] Resultat dessen ist es, dass das Verständnis für das, was entwickelt wird verbessert wird und die Abstraktion auf einer höheren Ebene beherrschbar bleibt. (vgl. [BCK08])

- **Höhere Qualität** [VBK⁺13]

Bei der Entwicklung einer DSL werden Sprachkonstrukte und Freiheitsgrade der Sprache festgelegt. Richtig konzipiert, schränken sie den Entwickler beim Umgang mit dieser DSL so ein, dass doppelter Code und doppelte Arbeit verhindert wird. Zusätzlich wird die Anzahl von Fehlern verringert. [VBK⁺13, S.40f] Auch durch die hohe Abstraktion einer DSL wird die Wiederverwendung von Code gefördert, was ebenso zu qualitativ höherem Code führt. [gho11, S.21]

⁹Diese vier Schritte beschreiben den Prozess des visuellen Denkens. [Roa09]

- **Verbesserte Produktivität bei der Entwicklung der Software**

Durch die Ausdruckskraft und der Abstraktion der Sprache muss i.d.R. auch weniger DSL-Code für die Implementierung eines Programms geschrieben werden, als wenn dieses Programm mit einer GPL implementiert wird. Wobei man mit einem entsprechenden Framework für GPLs ähnliches erreichen könnte. [VBK⁺13, S.40]

Die stärkere Ausdruckskraft führt zu einer bessere Lesbarkeit von DSL-Code im Vergleich zu GPL-Code, wodurch DSL-Code einfacher zu verstehen ist. Dadurch ist es auch einfacher Fehler in diesem Code zu finden, sowie Anpassungen an dem System vorzunehmen. Bei einer GPL werden diese Vorteile durch Dokumentationen, ausdrucksvolle Variablenbezeichnungen und festgelegten Konventionen angestrebt. [FP11, S.33] Dies ist jedoch mit einem höheren Aufwand verbunden. Zumal der Entwickler diese Vorschriften von sich aus einhalten muss. Bei der Verwendung einer DSL ist er zur Umsetzung einer besseren Lesbarkeit gezwungen, da die Sprache es nicht anders zulässt.

- **Bessere Kommunikation mit Domänen-Experten und Kunden**

Aufgrund domänenspezifischer und präziser Ausdrücke, die in der Sprache verwendet werden, sind die Domänen-Experten bzw. die Kunden vertrauter mit der Implementierung, als wenn für die Umsetzung eine GPL verwendet werden würde. [VBK⁺13, S.42] Die hohe Ausdruckskraft fördert das Verständnis dieser DSL. Damit ist es einfacher, die Kunden in die Entwicklung mit einzubeziehen. Dabei sollten jedoch zusätzliche Hilfsmittel wie Visualisierungen oder Simulationen verwendet werden. [FP11, S.34], [VBK⁺13, S.42] Das fördert die Kommunikation zwischen Kunden und Auftragnehmern, die oft vernachlässigt wird. Martin Fowler beschreibt sogar den Einsatz einer DSL als reine Kommunikationplattform als vorteilhaft. [FP11, S.34f] Grund dafür ist auch, dass die Entwicklung einer DSL das Verständnis der Domäne des Auftragnehmers steigert. [VBK⁺13, S.41]

- **Plattformunabhängigkeit [FP11]**

Durch die Nutzung einer DSL kann ein Teil der Logik von der Kompilierung in den Ausführungskontext überführt werden. Die Definition der Logik findet dabei in der DSL statt, welche erst bei der Ausführung evaluiert werden. Das wird auch oft unter der Verwendung von XML umgesetzt. [FP11, S.35] Dadurch ist es möglich die Logik auf unterschiedlichen Plattformen auszuführen. [VBK⁺13, S.43] Dieser Vorteil ist besonders für den praktischen Teil dieser Arbeit interessant.

- **Einfachere Validierung und Verifizierung**

Da DSLs Details der Implementierung ausblenden sind sie auf semantischer Ebene reichhaltiger als GPLs. Das führt dazu, dass Analysen einfacher umzusetzen sind und Fehler-Meldungen verständlicher gestaltet werden können, indem die Terminologie der Domäne verwendet wird. Dadurch und durch die vereinfachte Kommunikation mit den Domänen-Experten werden Reviews und Validierungen des Codes weitaus effizienter. [VBK⁺13, S.41]

- **Unabhängigkeit von Technologien**

Die Modelle, die zur Beschreibung von Systemen verwendet werden, können so gestaltet werden, dass sie von Implementierungstechniken unabhängig sind. Dies wird durch ein hohes Abstraktionsniveau erreicht, welches an die Domäne angepasst ist. Dadurch kann die Beschreibung der Modelle von den genutzten Technologien weitgehend entkoppelt werden. [VBK⁺13, S.41]

- **Skalierbarkeit des Entwicklungsprozess**

Die Integration von neuen Mitarbeitern in ein Entwicklerteam fordert immer eine gewisse Einarbeitungszeit. Diese Einarbeitungszeit kann durch die Nutzung einer DSL verkürzt werden, wenn die DSL einen hohen Abstraktionsgrad hat und dadurch leichter verstanden und zu erlernen ist. [gho11, S.21] Innerhalb eines Entwickler-Teams haben die Mitarbeiter oft einen unterschiedlichen Erfahrungsstand bzgl. einer speziellen Programmiersprache, die zur Entwicklung genutzt werden soll. Erfahrene Teammitglieder können sich mit der Implementierung der DSL befassen und die Grundlage für die anderen Teammitglieder schaffen. Diese wiederum nutzen die DSL, um die fachlichen Anforderungen der Kunden zu implementieren. [gho11, S.21] Markus Völter hingegen sieht die Teilung der Programmieraufgaben als Gefahr bzw. Nachteil. [VBK⁺13, S.44]

Nachteile

- **Großes Know-How gefordert**

Bevor die Vorteile einer DSL genutzt werden können, muss die DSL entwickelt werden. [VBK⁺13, S.43] Das Designen einer Sprache ist eine komplexe Aufgabe, die nur schwer skalierbar ist. [gho11, S.21] Die Vorteile, die eine DSL bietet, können nur geboten werden, wenn diese DSL auch entsprechend gutes Konzept hat. Dazu muss zum einen der richtige Abstraktionsgrad gefunden werden und zum anderen die Sprache so einfach wie möglich gehalten werden. Für beide Aufgaben werden Entwickler benötigt, die viel Erfahrung mit Sprach-Design haben. [VBK⁺13, S.44]

- **Kosten für die Entwicklung der DSL** Bei wirtschaftliche Entscheidungen wird der Input mit dem Output verglichen. Investitionen führen dazu, dass der Input größer wird. Da eine DSL vor dem Einsatz zuerst entwickelt werden muss, ist notwendig Investitionen für die Entwickler der DSL zu tätigen. Ob sich eine Investition lohnt, wird mittels vorher durchzuführenden Analysen überprüft. Dabei muss festgestellt werden, ob die Entwicklung der DSL gerechtfertigt ist. Im Bereich der technischen DSLs ist fällt die Rechtfertigung einfach, da diese DSLs oft wiederverwendet werden können. Fachliche DSL hingegen haben oft eine weitaus kompaktere Domäne, als eine technische DSL. Daher ergeben sich die Möglichkeiten zur Wiederverwendung erst zu einem späteren Zeitpunkt und können nur schwer von der im Vorfeld durchgeführten Analyse wahrgenommen werden. [VBK⁺13, S.43]
Weiterhin ist in der Phase, in der die DSL entwickelt wird, keine große positive Änderungen in den Kosten zu erwarten. Die Kosten reduzieren sich i.d.R erst wenn die DSL eingesetzt wird. [gho11, S.21]
Bevor eine DSL entwickelt werden kann, sollte ein entsprechendes Know-How aufgebaut werden. Der Aufbau dieses Wissens erfordert wiederum Kosten. [FP11, S.37]
- **Investitions-Gefängnis** [VBK⁺13]
Der Begriff stammt von Markus Völter. Er beruht auf der Annahme, dass sich ein Unternehmen dessen bewusst ist, dass mehr Investitionen in wiederverwendbare Artefakte zu einer besseren Produktivität führen. [VBK⁺13, S.45] Artefakte, die wiederverwendet werden können, führen dennoch zu Einschränkungen. Die Flexibilität geht dabei verloren. Dabei besteht die Gefahr, dass die Artefakte aufgrund geänderter Anforderungen, unbrauchbar werden. Weiterhin ist es auch gefährlich Artefakte zu Verändern, die häufig wiederverwendet werden, da durch diese Veränderung Nebeneffekte auftreten können, die nicht erwünscht sind. Somit muss das Unternehmen wiederum mehr investieren um die Anforderungen umzusetzen. Von daher der verwendete Begriff *Investitions-Gefängnis*.

- **Kakophonie**

Die Kakophonie beschreibt einen schlechten Klang einer Sprache. Eine DSL abstrahiert von Domänen-Modell. [gho11, S.22] Je besser diese Abstraktion ist, desto euphonischer und ausdrucksstärker ist die Sprache. Dass dafür viel Erfahrung benötigt wird, wurde bereits erwähnt. Normalerweise werden für eine Applikation mehrere DSLs benötigt. Diese unterschiedlichen DSLs haben i.d.R. unterschiedliche syntaktische Strukturen. Das führt dazu, dass Mitarbeiter immer wieder neue Sprachen lernen müssen. Das wiederum führt zu höheren Kosten. Weiterhin müssen die Entwickler bei der Verwendung mehrerer Sprachen öfter umdenken, als wenn sie fortwährend mit einer Sprache arbeiten würden. Das macht den Entwicklungsprozess weitaus komplizierter. [FP11, S.37]

- **Ghetto Slang**

Dieser Nachteil steht im Kontrast zum Punkt *Kakophonie*. Wenn ein Unternehmen nur mit eigenen DSLs arbeitet, die niemand sonst kennt, oder einsetzt, gleichen diese Sprachen einem Ghetto Slang, den niemand sonst versteht. Dadurch ist es schwer neue Technologien in den Bereichen, wo vermehrt DSLs eingesetzt werden, zu integrieren. Außerdem ist es kaum möglich, von neuen Mitarbeitern in diesem Bereich zu profitieren, da diese sich höchswahrscheinlich nicht einmal diese DSLs kennen. [FP11, S.38]

Dieser Punkt ist auch in Verbindung mit dem *Investitions-Gefängnis* zu betrachten. Durch die Verwendung übermäßig vieler DSLs ist das Unternehmen gezwungen, diese durch eine große Investition abzusetzen und allgemein bekannte Technologien einzuführen, um von diesen zu profitieren, oder das Unternehmen investiert weiter in die Entwicklung eigener DSLs, um seine Systeme aufrecht zu erhalten.

- **Abstraktion als Scheuklappen**

Abstraktion ist von großer Wichtigkeit für eine DSL. Wenn ein Entwickler mit der Arbeit an einer DSL begonnen hat, hat dieser die Abstraktion in einem bestimmten Maß bereits festgelegt. Ein Problem tritt auf, wenn im Nachhinein etwas mit der Sprache beschrieben werden soll, dass nicht zu der Abstraktion der Sprache passt. Dabei besteht die Gefahr, dass der Entwickler sich von der Abstraktion der Sprache gefangen nehmen lässt. Das bedeutet, dass der Entwickler versucht, das Problem aus der realen Welt auf seine Abstraktion anzupassen. Der richtige Weg hingegen ist es, die Sprache und deren Abstraktionsgrad so anzupassen, dass das Problem mit beschrieben werden kann. [FP11, S.39]

- **Kulturelle Herausforderungen**

Die genannten Nachteile den Einsatzes von DSLs führen zu Äußerungen wie *Die Entwicklung von Sprachen ist kompliziert, Domänen-Experten sind keine Programmierer* oder *Ich möchte nicht schon wieder eine neue Sprache lernen* (*Yet-Another-Language-To-Learn Syndrom* [gho11, S.22]). Solche kulturellen Probleme entstehen immer, wenn etwas neues eingeführt werden soll. [VBK⁺13, S.45] Die Mitarbeiter müssen demnach entsprechend geschult und motiviert werden.

- **Unvollständige DSLs**

Wenn ein Unternehmen viel Erfahrung bei der Entwicklung von DSLs aufgebaut hat und die Entwicklung durch entsprechende Tools vereinfacht wurde, besteht die Gefahr, dass DSLs zu schnell entwickelt werden. Durch die Einfachheit der Entwicklung scheint es einfacher eine neue DSL zu entwickeln, als nach bestehenden Ansätzen für dasselbe Problem zu suchen. [VBK⁺13, S.44f] Der Gedanke daran, dass sich die Investition in die Entwicklung einer DSL zu einem späteren Zeitpunkt amortisieren wird, bestätigt diese Haltung. [FP11, S.38] Dadurch entstehen immer mehr DSLs, die auf gleichen Problemen basieren, aber inkompatibel zueinander sind. Außerdem führt der Fakt, dass die Entwicklung einer DSL bei dem Verstehen der Domäne und dem Entwerfen des Models sehr hilfreich ist, dazu, dass eine DSL nur zum Verständnis des Problems oder der Domäne genutzt wird. [FP11, S.38] Das wiederum führt dazu, dass mehrere halb-fertige DSLs existieren. Markus Völter et. Al. nennen dieses Phänomän die *DSL Hell*. [VBK⁺13, S.44f]

Zusammenfassend ist zu sagen, dass der Aufwand für die Vorbereitung des Einsatzes einer DSL sehr hoch ist. Wurde eine DSL jedoch eingeführt, wird sich der Arbeitsaufwand um ein Vielfaches verringern und der letztendliche Gewinn fällt höher aus. [gho11, S.21]

3.6 Interne DSL

Bei einer internen DSL handelt es sich um eine DSL, die in eine GPL integriert sind. Sie übernehmen dabei das Typ-System der GPL in die sie integriert sind. [VBK⁺13, S.50] In Betrachtung der Ziele aus Kapitel 3.3 können einige dieser mit Application Programming Interfaces (API) erreichen werden. In vielen Fällen ist eine DSL nicht mehr als ein API. Martin Fowler sieht den größten Unterschied zwischen API und DSL darin, dass das eine DSL neben einem abstrahierten Vokabular auch eine spezifische Grammatik nutzt ([FP11, S.29]), welche die Syntax der DSL bestimmt. Ein API hingegen besitzt die gleichen syntaktischen Strukturen wie die GPL, in der das API

bereitgestellt wurde. Somit werden überflüssige Notationsformen in das API mit übernommen, was bei einer DSL nicht der Fall ist. [VBK⁺13, S.30] Weiterhin können DSLs so konstruiert werden, dass durch Restriktionen und Limitierungen nur korrekte Programme geschrieben werden können. Markus Völter et. al. bezeichnen diese Eigenschaft als *correct-by-construction*. [VBK⁺13, S.30]

3.6.1 Implementierungstechniken

Parse-Tree Manipulation

Allgemein betrachtet funktioniert diese Technik wie folgt.

Ein Code-Fragment, welches zu einem Späteren Zeitpunkt ausgewertet werden soll, als es gelesen wurde, wird in einem Parse-Tree hinterlegt. Dieser Parse-Tree wird noch vor der Ausführung modifiziert. Um diese Implementierungstechnik nutzen zu können, muss eine Umgebung vorliegen in der es möglich ist ein Code-Fragment in einen Parse-Tree umzuformen und diesen zu bearbeiten. Diese Möglichkeit existiert nur in wenigen Sprachen. Martin Fowler et. al. geben hierzu nur die Beispiele C#, ParseTree (Ruby) und Lisp. [FP11, S.45f]

Anders als Lisp bieten die anderen Beispiele die Möglichkeit über den Parse-Tree zu iterieren. Bei Lisp-Code handelt es sich schon um einen Parse-Tree von verschachtelten Listen. Bei der Iteration über den Parse-Tree ist aufgrund der Performance darauf zu achten, dass möglichst nur die notwendigen Teile des Baum beachtet werden. [FP11, S.46]

Konstrukte, die in der Hostsprache geschrieben wurden und nicht verändert werden sollen, spielen bei der Parse-Tree Manipulation keine Rolle, um das semantische Model zu erzeugen. [FP11, S.46]

Fluent Interfaces

In einem klassischen API hat jede Methode eine eigene Aufgabe und ist nicht von anderen Methoden in diesem API abhängig. [FP11, S.28] In einer internen DSL hingegen ist es möglich Methoden bereitzustellen, die hintereinander gekettet werden können und somit komplette Sätze darstellen. Somit wird der Output einer Methode zum Input der folgenden Methode. Die Lesbarkeit der DSL wird dadurch weitaus besser, da es einer Sequenz von Aktionen gleicht, die in der Domäne ausgeführt werden (vgl. [gho11, S.94]) und ohne eine Vielzahl von Variablen aufgerufen werden müssen (vgl. [FP11, S.68]). Eine solche Verkettung von Methoden wird als *Fluent Interface* bezeichnet. Das Fluent Interface steht laut Voelter et. al. zwischen dem API und einer DSL. [VBK⁺13, S.50] Ein Beispiel für ein Fluent Interface bietet Fowler et. al. Dabei wird ein Computer mit einem Processor und zwei Festplatten beschrieben.

Listing 3.1: Beispiel: Fluent Interface

```
1 computer()
2     .processor()
3         .cores(2)
4         .speed(2500)
5         .i386()
6     .disk()
7         .size(150)
8     .disk()
9         .size(75)
10        .speed(7200)
11        .sata()
12    .end()
```

vgl. [FP11, S.68]

Annotationen

Annotationen sind ein Teil der Informationen über ein Programmelement, wie eine Methoden oder Variablen. Diese Informationen können zur Laufzeit oder zur Übersetzungszeit (wenn die Umgebung die Möglichkeit dazu bietet) manipuliert werden. [FP11, S.445]

Bevor eine Annotation verarbeitet werden kann muss sie definiert werden. Die Definition von Annotationen variiert zwischen unterschiedlichen Sprachen. [FP11, S.446] Die Verarbeitung von Annotationen findet normalerweise während der Übersetzung, während des Ladens des Programms oder während der Ausführung des Programms statt. [FP11, S.447] Verarbeitungen während der Laufzeit beeinflussen i.d.R. das Verhalten von Objekten. Beim Laden des Programms werden meist Validierungs-Annotationen verwendet. Solche Annotationen werden bspw. dazu verwendet das Mapping für die Datenbanken auszulesen. Somit wird die Definition von Elementen von der Verarbeitung getrennt, was zu einem übersichtlichen und lesbaren Code beiträgt. [FP11, S.449]

3.7 Externe DSL

Eine externe DSL ist eine separate Sprache, welche die Infrastruktur vorhandener Sprachen nicht nutzt. [gho11, S.18] Das bedeutet, dass eine externe DSL eine eigene Syntax sowie ein eigenes Typsystem besitzt. In der Regel wird mit einer externen DSL ein Skript geschrieben, welches von einem Programm gelesen wird. Dieser Vorgang wird auch als *parse* bezeichnet. [FP11, S.28] Für den Parser sowie den lexikalischen Analysen werden oft vorhandene Infrastrukturen genutzt. [gho11, S.19]

3.7.1 Implementierungstechniken

Bei den Implementierungstechniken von externen DSL geht es um die Art und Weise, wie der DSL-Code vom Parser in ein semantisches Model oder einem AST überführt wird. [FP11, S.89] Die Allgemeine Vorgehensweise bei der Verwendung von Parsern ist Abbildung 3.2 zu entnehmen.

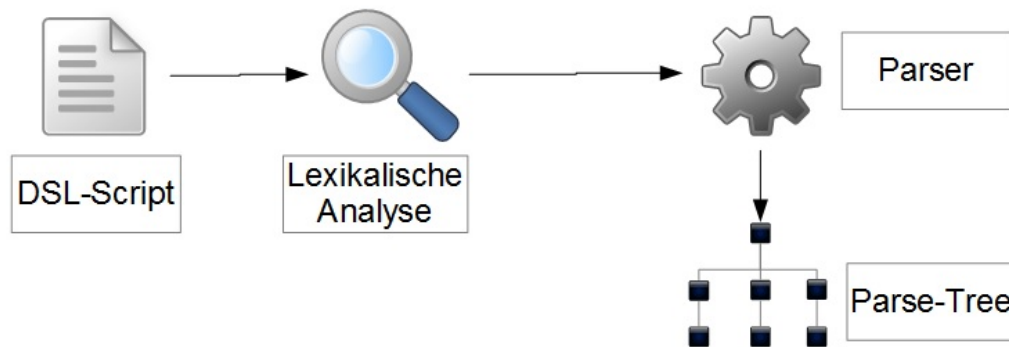


Abbildung 3.2: Parsen allgemein

Parser Generator

Bei der Generierung von Parsern muss der Parser nicht manuell implementiert werden. Diese Aufgabe wird an den Generator delegiert. Damit dies möglich ist, muss eine Grammatik in der EBNF¹⁰, sowie bestimmte Aktionen, die bei der Bestätigung bestimmter Grammatik-Regeln ausgeführt werden sollen (Validierungs-Regeln), definiert werden. [gho11, S.218] Wird der Parser Generator ausgetauscht führt dies auch häufig dazu, dass die notwendigen Artefakte (Grammatik und Aktionen bei der Bestätigung von Grammatikregeln) neu definiert werden müssen. [FP11, S.269] Weiterhin arbeiten die meisten Parser Generatoren mit Code-Generierung, wodurch der Build-Process komplexer wird. [FP11, S.272] Vorteile dieser Technik im Vergleich dazu, dass der Parser manuell entwickelt wird, sind die folgenden.

- Programmieren auf einem höheren Abstraktionsniveau [gho11, S.218]
- Weniger Code zum Implementieren des Parsers [gho11, S.218]
- Möglichkeit des Generieren eines Parser in unterschiedlichen Sprachen [gho11, S.218], [FP11, S.270]
- Validierung der Grammatik durch Fehlererkennung und -behandlung [FP11, S.272]

¹⁰Siehe Glossar: EBNF

Recursive Decent Parser (RD-Parser)

Dieser Parser basiert auf Funktionen, die rekursiv aufgerufen werden. Es handelt sich dabei um einen Top-Down Parser¹¹. [gho11, S.226] Die Funktionen implementieren dabei die Parsing-Regeln für die nonterminalen Symbole der Grammatik. [FP11, S.245] Die Funktionen geben dabei einen Boolean-Wert zurück, der Auskunft darüber gibt, ob die Symbole aus dem DSL-Script mit den Symbolen übereinstimmen, die laut Grammatik erwartet werden. [FP11, S.246] Tabelle 3.1 zeigt die Implementierungsmöglichkeiten von einfachen Grammatikregeln auf.

Grammatik-Regel	Implementierung
$A \mid B$	<pre> 1 if (A()) 2 then true 3 else if (B()) 4 then true 5 else false </pre>
$A B$	<pre> 1 if (A()) 2 then if (B()) 3 then true 4 else false 5 else false </pre>
$A?$	<pre> 1 A() ; 2 true </pre>
A^*	<pre> 1 while (A()) ; 2 true </pre>
A^+	<pre> 1 if (A()) 2 then while (A()) ; 3 else false </pre>

Tabelle 3.1: Implementierung einfacher Grammatikregeln mit einem RD-Parser [FP11, S.248]

Da dieser Parser direkt implementiert werden kann, ist es ebenso möglich diesen Parser zu Debuggen. Das ist neben der einfachen Implementierung (solange es sich um eine einfache Grammatik handelt) ein großer Vorteil dieser Technik. [FP11, S.249] Ein großer Nachteil ist, dass keine Grammatik definiert wird. Laut Fowler et. al. wird dadurch einer DSL ein großer Vorteil entzogen. [FP11, S.249]

¹¹Siehe Glossar: Top-Down Parser

Parser-Kombinator

Bei der Kombination von Parsern wird die Grammatik mittels einer Struktur von Parser Objekten implementiert. [FP11, S.256] Wenn ein Teil des Input-Streams von einem Parser erfolgreich oder fehlerhaft verarbeitet wurde, kann der Rest den Input-Streams an einen anderen Parser übergeben werden. Somit ist es möglich Parser beliebig zu verketteten. [gho11, S.242] Die Elemente, die verkettet werden können werden *Parser-Kombinatoren* genannt. Abbildung 3.3 stellt schematisch diese Funktionsweise dar.

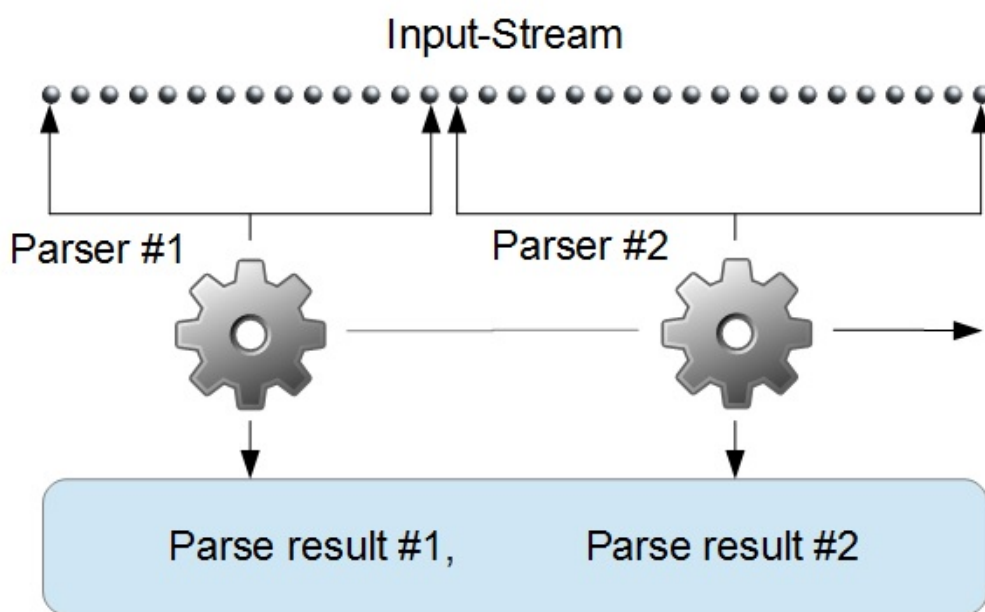


Abbildung 3.3: Funktionsweise von Parser-Kombinatoren (in Anlehnung an [gho11, S.243])

Bezogen darauf, dass ein Parser aus Funktionen besteht, sind diese Parser-Kombinatoren Funktionen erster Ordnung, die unterschiedlich kombiniert werden können. (vgl. [gho11, S.243], [FP11, S.256]) Durch diese Kombination wird eine Struktur gebildet, welche das semantische Model repräsentiert. [FP11, S.256] Ein großer Vorteil dieser Technik ist, dass einfache Parser zu komplexeren Parsern zusammengefügt werden können. Weiterhin wird durch die Kombination mehrerer Grammatik-bestimmender Komponenten auch die Lesbarkeit der Grammatik gefördert, was bei einem RD-Parser ein großer Nachteil war. Daher bezeichnen Fowler et. al. Parser-Kombinatoren auch als Mittelweg zwischen RD-Parsern und Parser Generatoren. [FP11,

S.261]

3.8 Nicht-Textuelle DSL

Bei den in den letzten Kapiteln vorgestellten internen und externen DSLs handelt es sich um textuelle DSLs. Auch wenn eine DSL eine bestimmte domäne repräsentiert, bedeutet dies nicht, dass diese Repräsentation immer textuell erfolgen muss. [gho11, S.19] Es gibt einige Gründe, mit einer nicht-textuellen DSL zu arbeiten:

- Viele Domänenprobleme können durch die Domänen-Nutzer besser durch Tabellen oder grafischen Darstellungen erklärt werden
- Domänenlogik ist in textueller Form oft zu komplex und enthält zu viele syntaktische strukturen
- visuelle Modelle sind einfacher zu durchdringen und zu verändern durch Domänenexperten

[gho11, S.19]

Für diesen Ansatz muss der Domänen-Nutzer die Repräsentation des Wissens über eine Domäne in einem Editor (Projection Editor) visualisieren. Mit diesem Editor kann der Domänen-Nutzer die Sicht auf die Domäne verändern, ohne auch nur eine Zeile code schreiben zu müssen. Im Hintergrund generiert dieser Editor den Code, welcher Sicht auf die Domäne modelliert. [gho11, S.19f]

Kapitel 4

Entwicklung einer Lösungsidee

4.1 Allgemeine Beschreibung der Lösungsidee

Eine Lösungsidee für die in Kapitel 2.3 beschriebenen Probleme wurde im Kapitel 2.4 bereits angedeutet. Kern dieser Idee ist, eine DSL zur Beschreibung von GUIs zu nutzen. Diese GUIs sollen so beschrieben werden, dass sie in der Domäne von profil c/s für unterschiedliche UI-Frameworks genutzt werden können. Diese Beschreibung soll weiterhin nur ein mal statt finden. Der Quell-Code, welcher das GUI im entsprechenden Framework darstellt, wird frameworkspezifisch aus der GUI-Beschreibung generiert. Langfristig betrachtet kann das MCF damit theoretisch abgelöst werden.

4.2 Architektur

In diesem Lösungsansatz ist die DSL der Ausgangspunkt. Mit deren Hilfe wird eine abstrakte Beschreibung der GUI vorgenommen. Somit ist gewährleistet, dass die GUI weiterhin nur einmal beschrieben werden muss. Ein Generator kann nach dem parsen dieser Beschreibung, frameworkspezifischen Quell-Code generieren. Somit ist die Integration neuer Frameworks an die Implementierung eines spezifischen Generators gekoppelt. Abbildung 4.1 zeigt die Architektur für diesen Ansatz auf. Dabei wurden exemplarisch drei unterschiedliche Generatoren für unterschiedliche Frameworks mit aufgenommen.

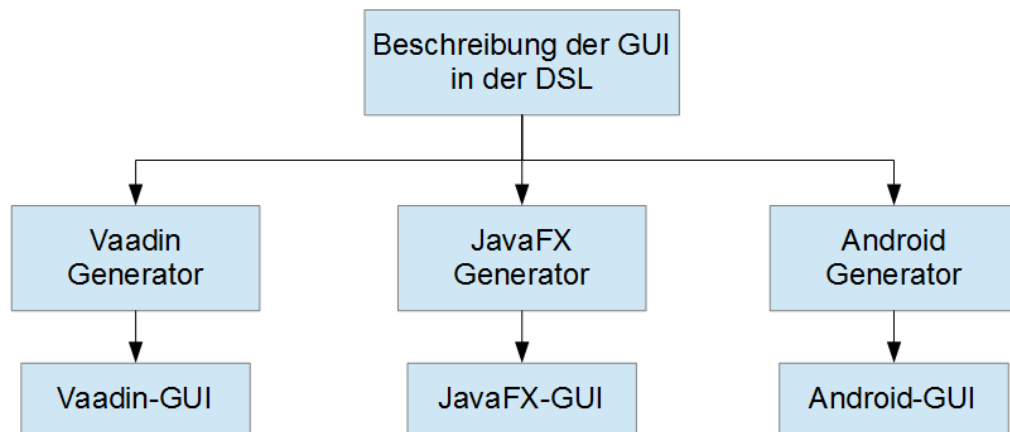


Abbildung 4.1: DSL-Ansatz für gleich GUIs auf unterschiedlichen Plattformen

4.3 Vorteile gegenüber dem Multichannel-Framework

Wie in Kapitel 2.3 erläutert, weist das MCF einige Probleme auf. Mit der vorgestellten Lösung kann das Problem der inaktuellen Frameworks und das Problem der starken Orientierung an Swing (oder an ein anderes Framework) beseitigt werden. Eine DSL sollte sich nicht an Besonderheiten bestehender Frameworks orientieren, sondern an dem Domänenproblem. (vgl. [mds06, S.15]) Von daher sollte bei korrekter Umsetzung sichergestellt sein, dass die Integration von unterschiedlichen Frameworks gleichermaßen gut funktioniert.

Ein weiterer Vorteil ist, dass durch die wegfallende Orientierung an Swing auch die Beschreibungsform ausdrucksstärker wird. Grund dafür ist, dass die syntaktischen Strukturen, die in Swing vorhanden sind, nicht mehr benötigt werden und die DSL auf einer höheren Abstraktionsebene konzipiert werden kann. Das erweitern der DSL um fachliche Konzepte aus der c/s-Welt sollte somit ermöglicht werden. Mit diesen fachlichen Konzepten wird eine Annäherung an das Model-Driven Development (siehe Kapitel 3.3) erreicht.

Darüber hinaus werden dem Entwickler durch die Codegenerierung die in Kapitel 2.3 fehleranfällige Routinearbeiten abgenommen.

Kapitel 5

Anforderung an die GUI-DSL

Die allgemeinen Anforderung an die GUI wurden in Kapitel 2.1 erläutert. Die folgenden Anforderungen beziehen sich auf die Aspekte der GUIs die beschrieben werden müssen. Ein UI ermöglicht die Interaktion mit einem Programm mit Hilfe unterschiedlicher UI-Komponenten (vgl. [Gal07, S.4]). Mithilfe dieser Komponenten werden Informationen dargestellt, oder Eingaben vom Nutzer getätigt. Um die Zusammensetzung dieser Komponenten zu beschreiben gibt es zwei Ansätze.

Beim ersten Ansatz wird die GUI durch fachliche Modelle beschrieben. (Motivation für MDSD [SKNH05]) Das bedeutet, dass in der Beschreibung der GUI keine UI-Komponenten verwendet werden, wie es in allgemein bekannten UI-Frameworks (JavaFX, Swing) der Fall ist. In der deg soll den Entwicklern weiterhin die Möglichkeit gegeben werden, die UIs selbst zu entwerfen. Ein Grund dafür ist, dass es ein zu großer Aufwand wäre alle Module von profil c/s auf MDSD umzustellen und die GUIs generieren zu lassen. Von daher ist dieser Ansatz vorerst nicht umsetzbar.

Die Komplexität des zweiten Ansatzes scheint weitaus geringer zu sein. Dabei werden weiterhin UI-Komponenten in der GUI-Beschreibung verwendet. Die Entwickler haben somit die Möglichkeit die UIs in einem gewissen Grad anzupassen. Die GUI von profil c/s wird dabei als Domäne betrachtet und nicht die fachlichen Hintergründe. Durch die Festlegung dieser Domäne lässt sich bei der Beschreibung von GUIs eine höhere Abstraktionsebene nutzen, als es bei den verwendeten UI-Frameworks der Fall ist.

Der Entwickler soll somit in der Lage sein ein UI relativ frei zu gestalten.

Außerdem sollte er dadurch die Möglichkeit haben bestimmte Konzepte einfach wieder zu verwenden. Für die GUI-DSL wird bzgl. der Anforderungen an die Komponenten zwischen drei Kategorien unterschieden.

Die erste Kategorie umfasst *triviale UI-Komponenten*. Dabei handelt es sich um UI-Komponenten, deren Funktionen in unterschiedlichen UI-Frameworks ähnlich sind und in unterschiedlichen Anwendungen eingesetzt werden können. Das bedeutet, dass sie nicht als domänenspezifisch angesehen werden können. Beispiele hierfür sind UI-Komponenten wie der *Button* oder das *Label*.

Die zweite Kategorie umfasst *komplexe UI-Komponenten*. Diese zeichnen sich dadurch aus, dass sie domänenspezifisch sind und speziell für profil c/s entwickelt wurden. Ein Beispiel für eine komplexe Komponente ist die Multiselection-Komponente¹.

Die dritte Kategorie umfasst *Layout Komponenten*. Dabei handelt es sich um strukturgebende Komponenten. In anderen UI-Frameworks sind dies bspw. *Panel*, *Div* oder *Pane*. In der GUI-DSL müssen auch solche Komponenten verfügbar sein. Dabei ist besonders auf die Ausdruckskraft der für die Beschreibung dieser Komponenten verwendeten Bezeichnungen zu achten. Grund dafür ist, dass sich bspw. auf dem Desktop ein Fenster als oberste Layout Komponente festlegen lässt. In einem Web-Browser ist der Begriff *Fenster* als oberste Layout Komponenten jedoch nicht geläufig. Die Komponente, die in einem Browser dem Fenster auf dem Desktop am nächsten kommt, ist meiner Meinung nach das *Tab*.

Die Attribute, die für die einzelnen Komponenten beschrieben werden müssen, werden in Kapitel 8 genauer analysiert.

Bezüglich des Layouts ist neben den Layout Komponenten eine weitere Anforderung zu nennen. Hierzu muss erwähnt werden, dass in der traditionellen UI-Entwicklung GUIs mit Hilfe von Layout-Containern strukturiert werden. In der Vergangenheit hat sich gezeigt, dass die Strukturierung über ein spezifisches Layout zu einer Orientierung an ein bestimmtes Framework führt (Beispiel: MCF orientiert sich an Swing). Das ist ein Problem, da bestimmte Layouts im Web oder auf mobilen Plattformen nicht genauso dargestellt werden können, wie auf einer Desktop-Anwendung. Von daher ist

¹Siehe Glossar: Multiselection-Komponente

das Layout in der GUI-Beschreibung so zu beschreiben, dass es auf allem Plattformen gleichermaßen gut dargestellt werden kann.

Darüber hinaus ist es für die Effizienz in der Entwicklung wichtig, dass mit dem neuen Ansatz weniger Codezeilen (LOC) geschrieben werden müssen als mit dem alten Ansatz.

Weiterhin ist für ein effizientes Arbeiten auch die Bereitstellung eines Editors für die DSL von großer Bedeutung. Dieser Editor soll nach Möglichkeit auch Validierungen durchführen können und Code-Completion anbieten. Eine Integration dieses Editors in die von der Entwicklungsumgebung (Eclipse) verwendete Entwicklungsumgebung (Eclipse) wäre wünschenswert.

Bezüglich der Anforderungen an die Komponenten ist abschließend zu sagen, dass bei den trivialen und den komplexen UI-Komponenten die Möglichkeit bestehen muss Interaktionen festzulegen. Außerdem muss die GUI-DSL um weitere triviale und komplexe Komponenten erweiterbar sein.

Zusammenfassend bestehen die Sprache folgenden Anforderungen

- Beschreibung trivialer UI-Komponenten
- Beschreibung komplexer UI-Komponenten
- Beschreibung von Layout Komponenten
- Verwendung einer abstrakten Layoutbeschreibung
- Weniger LOC zur Beschreibung von UIs
- Beschreibung von Interaktionen an trivialen und komplexen UI-Komponenten
- Erweiterung um neue triviale und komplexe UI-Komponenten

Für die Infrastruktur der DSL bestehend die folgenden Anforderungen.

- Bereitstellung eines Editors mit Code-Completion und Validierungsmöglichkeiten
- Integration in die Eclipse IDE

Kapitel 6

Evaluation des Frameworks zur Entwicklung der DSL

6.1 Vorstellung ausgewählter Frameworks

Zur Umsetzung der DSL und der Generatoren wird ein Framework benötigt, welches dafür notwendige Funktionalitäten bereit stellt. Hierzu werden die Frameworks *PetitParser*, *Xtext* und *MPS* kurz vorgestellt und im Anschluss verglichen.

6.1.1 PetitParser

Dieses Frameworks arbeitet mit Parser-Kombinatoren. Somit ist es mit PetitParser einfach Grammatiken zusammenzustellen, zu transformieren oder zu erweitern, sowie Teile dieser dynamisch wiederzuverwenden. Alles geschieht auf der Basis von Pharo Smalltalk, womit das Framework ursprünglich implementiert wurde. Es existieren jedoch auch Versionen des Frameworks für Java¹, Dart² und PHP³. Einfache Parser bestehen aus Sequenzen von Funktionen, welche die Produktionsregeln (Produktionen) der Grammatik abbilden. Komplexere Parser werden durch die Kombination anderer Parser implementiert. (vgl. [RDGN10]) Die Kombination kann in einer einzelnen Methode implementiert werden, was dazu führt, dass man den Par-

¹<https://github.com/petitparser/java-petitparser>

²<https://github.com/petitparser/dart-petitparser>

³<https://github.com/mindplay-dk/petitparserphp>

ser in einer Skript-Form erhält. Alternativ können die zu kombinierenden Parser auch in Methoden von Unterklassen des `PetitParsers` implementiert werden. ([bra10, S.6]) Das fördert die Lesbarkeit, Übersichtlichkeit und schließlich die Wartbarkeit des Codes.

Tool Support ist für dieses Framework gewährleistet. Mithilfe dessen können Produktions editiert und grafisch abgebildet werden. Aus Zufallsbeispiele für ausgewählte Produktionen werden generiert um so Fehler in der Grammatik aufzudecken. Darüberhinaus wird die Effizienz einer Grammatik durch die Darstellung direkter, ineffizienter Zyklen in der Grammatik verbessert. (vgl. [RDGN10])

6.1.2 Xtext

Bei *Xtext* handelt es sich um eine Open-Source-Lösung für einen *ANTLR*-basierten Parser- und Editorgenerator mit der externe, textuelle DSLs entwickelt werden können. Die Grammatiken für den Parser-Generator werden in der EBNF definiert. Durch die Integration in Eclipse kann der Eclipse-Editor für sämtliche Artefakte der Infrastruktur verwendet werden. Aus der Grammatik wird der LL(k)-Parser, sowie ein Model, mit dessen Hilfe der Generator implementiert werden kann, generiert. Die Klassen für Validierungs-Regeln und den Generator werden ebenfalls vom Tool erzeugt. Diese müssen im Anschluss daran vom Nutzer entsprechend erweitert werden. Zur Editierung der entsprechenden Dateien wird eine eigene Syntax verwendet, die meiner Meinung jedoch stark an die Java-Syntax erinnert.

Wenn der Parser generiert wurde, ist *Xtext* in der Lage einen in Eclipse integrierten Editor zu erzeugen. [?, S.1] Dieser Editor ist in der Lage die Validierungs-Regeln auf die DSL-Skripte anzuwenden. Darüber hinaus wird auch Code-Completion vom Editor angeboten.

6.1.3 MPS

6.2 Vergleich und Bewertung der vorgestellten Frameworks

Know-How Erlernbarkeit Machbarkeit der Intergration Verwendung eines
Editors für DSL-Skripte Erweiterbarkeit der Grammatik Erweiterbarkeit der
Validierungen

Kapitel 7

Aufteilung der Anforderungen auf Sprache und Generator

Die Aufteilung der Anforderung in diesem Kapitel ist nach einigen Umsetzungsversuchen entstanden. Grund dafür ist, dass es in der deg keine Vorstellungen darüber gab, wie die konkrete Syntax der Sprache letztendlich aussehen sollte. Die Überlegungen, Irrtümer und Versuche, die zu dieser Aufteilung geführt haben, sind in Kapitel 8 und Kapitel 9 festgehalten.

7.1 Anforderung an die neue DSL

7.2 Anforderung an den Generators

Kapitel 8

Entwicklung einer DSL zur Beschreibung der GUI in profil c/s

Die Entwicklung der DSL und die Entwicklung des Generators (siehe Kapitel 9) gliedern sich in mehrere Versionen. In jeder Version sind Ideen umgesetzt, die nach der Fertigstellung einer Version in der deg begutachtet wurden und wenn nötig abgewandelt oder verändert.

8.1 Analyse der Metadaten der GUI

Version 1

Überlegungen bzgl. des Aufbaus der UIs gingen dahin, dass sich die DSL an dem Komponentenmodel von RCP4 orientieren soll. Das bedeutet, dass es Meta-Ebene gibt, die den groben Aufbau der GUI beschreibt und eine Implementierungs-Ebene, welche spezifische Komponenten innerhalb der GUI umsetzen soll. Beim Aufbau der GUI wurde zwischen zwei Typen unterschieden (siehe semantisches Model *TypeDefinition*). Für die Beschreibung des Aufbaus enthält jede beschriebene GUI eine bestimmte Anzahl von Bereichen (*Area*), denen genau eine andere UI-Komponent zugeordnet werden kann (siehe semantisches Model *AreaCount* und *AreaAssingment*). Weiterhin können von einer GUI-Beschreibung andere GUI-Beschreibungen verwendet werden (siehe semantisches Model *Use*). Die verwendeten GUI-Beschreibungen können jedoch nicht erweitert werden. Den Kern der GUI-Beschreibung jedoch die Kompnentendefinition (siehe semantisches Model

Definition). Dort werden einzelne Komponenten der GUI durch die Meta-Daten beschrieben.

Bezogen auf die trivialen Komponenten des UIs ist die Beschreibung eines Textes wichtig. Im Falle eines Buttons oder eines Labels (andere triviale Komponenten sind in dieser Version nicht umgesetzt) beschreibt dieser die Aufschrift der Komponente. Weiterhin war es für die Zuweisung zu den entsprechenden Bereich wichtig, dass diese Komponenten innerhalb der Datei referenziert werden können. Das wurde durch den Titel umgesetzt, der für jede Komponente definiert werden muss. An den trivialen Komponenten können darüber hinaus Interaktionen beschrieben werden. Hierzu ist ein Interaktionstyp nötig. Eine einfacher Klick auf die Komponente ist der einzige Interaktionstyp in dieser Version. An dieser Interaktion können ebenso Aktionen definiert werden, die Auswirkungen auf andere Komponenten haben. Zusammenfassend ergeben sich folgende Meta-Daten der trivialen Komponenten.

- Typ
- Titel
- Text
- Interaktion

Die Interaktion benötigt folgende Attributen, die beschrieben werden müssen.

- Titel
- Interaktionstyp
- Aktion

Die Aktion benötigt einen *ActionType*, das *Element* auf das sich die Interaktion auswirken soll und die Veränderung der Attribute des entsprechenden Elements (*Properties*).

Die komplexen Komponenten müssen für jedes verwendete UI-Framework implementiert werden. Das hat zur Folge, dass die Implementierung dieser Komponenten nicht so stark abstrahiert wird, dass sie nur einmal entwickelt

werden müssen. Damit wird jedoch auch verhindert, dass die Entwickler, die bzgl. der GUI nur mit der DSL arbeiten, eigene komplexe Komponenten entwerfen, deren Wiederverwendungsgrad niedriger ist, als wenn diese Komponenten nach ausreichender Evaluation an einer zentralen Stelle implementiert und bereitgestellt werden. Ein Nachteil dieses Konzeptes ist es, dass gewährleistet sein muss, dass die Quellen für diese komplexen Komponenten sowohl zur Entwicklungszeit, als auch zur Laufzeit vorhanden sind.

Da für die komplexe Komponenten eine Klasse im Classpath vorliegen muss, könnten diese Komponenten in eine GUI-Beschreibung wie andere verwendete GUI-Beschreibungen über *use* eingebunden werden. An komplexen Komponenten sollen jedoch weitere optionale Wertzuweisungen möglich sein. Deshalb werden komplexe Komponenten wie die trivialen Komponenten in einer Komponentendefinition beschrieben. Dazu wird nach der Implementierung der Komponente für jedes Framework ein neues Schlüsselwort für eine Komponentendefinition eingebaut. Jede komplexe Komponente benötigt wiederum einen Titel um referenziert zu werden. In dieser Version ist eine Multiselection-Komponente¹ umgesetzt. Diese Komponente ist generisch implementiert. Der generische Typ kann in der DSL an dem Schlüsselwort *InputType* beschrieben werden. Die Werte, die in dieser Komponente selektiert werden können, werden über das Schlüsselwort *selectableValues* gesetzt und die Werte, die selektiert sind am Schlüsselwort *selectedValues*.

Version 2

Aus der Begutachtung der ersten Version kam heraus, dass die Aktionen, die bei Interaktionen ausgeführt werden, nicht in der DSL beschrieben werden müssen. Grund dafür ist, dass diese Aktionen teilweise sehr komplex sind und kaum abstrahiert werden können. Daher fällt der gesamte Teil zur Beschreibung von Aktionen bei Interaktionen weg.

Bezogen auf die komplexen Komponenten hat sich ergeben, dass lediglich nur den Input-Typ angegeben werden muss. Die Festlegung über selektier-

¹Siehe Glossar: Multiselection-Komponente

bare und selektierte Elemente ist ebenso wie die Aktion einer Interaktion teilweise zu komplex und schwer abstrahierbar. Das ermöglicht, die komplexen Elemente mittels *use* (siehe semantisches Model *UsedDefinitions*) in die GUI-Beschreibung einzubinden (siehe konkrete Syntax). Da für komplexe Komponenten immer noch der Input-Typ angegeben werden kann, werden komplexen internen Komponenten und anderen komplexen externen Komponenten zwischen zwei Artefakten unterschieden (siehe semantisches Model *UsedDescription*)

Bezüglich der Komponenten, die mittels *use* eingebunden werden, ist es für die Lokalisierung der entsprechenden Quellen besser, wenn in der GUI-Beschreibung der qualifizierte Name angegeben wird. Dadurch wird die Beschreibung zwar länger, im Gegenzug dazu jedoch auch eindeutig.

Die in diesen eingebundenen GUI-Beschreibungen definierten Komponenten können in dieser Version weiter verfeinert werden. Dabei überschreiben die Werte, die in der bearbeiteten Beschreibung definiert wurden, die Werte die in der Originaldatei definiert sind. Werte die nicht überschrieben werden, werden aus der Originaldatei übernommen.

Eine weitere Diskussion regte die Art und Weise der Zuweisungen von Komponenten zu einem Area an. Bei der Lösung aus Version 1 ist es nicht möglich mehrere Komponenten einem Area zuzuweisen. Für die Meta-Daten einer Areazuweisung (siehe semantisches Model *AreaAssignment*) bedeutet dies, dass diese nicht nur mit einer Komponenten umgehen können muss, sondern mit einer Vielzahl von Komponenten.

Ein weiterer wichtiger Punkt, welcher in der ersten Version keine Beachtung fand, ist die Art und Weise, wie an den Komponenten bestimmte Werte wie bspw. die Aufschrift gesetzt werden. In der deg werden dazu so genannte *Properties*-Dateien verwendet, die mittels eines Frameworks ausgelesen werden und über einen Schlüssel den Komponenten zugewiesen werden. Damit wird die eigentliche Klasse zur Beschreibung der GUI in der deg entlastet. Auch die GUI-Beschreibung mittels DSL kann damit entlastet werden. Dazu wird jedoch ein Konzept benötigt, wie diese *Properties*-Dateien in die GUI-Beschreibung eingebunden und verwendet werden. Hierzu müssen die Meta-Daten für die GUI angepasst werden. Neben der Anzahl an Areas sowie der Zuweisung von Komponenten zu diesen und den Einbin-

dung anderer Komponenten, wird die benutzte Properties-Datei mit angegeben. Der benötigte Schlüssel für die Wertzuweisung wird innerhalb der Komponenten angegeben. Sind in einer Properties-Datei mehrere Werte für unterschiedliche Attribute einer Komponente angegeben, werden sie entsprechend zugeordnet, sodass in der GUI-Beschreibung nur der Schlüssel angegeben werden muss. Das hat den Vorteil, dass die GUI-Beschreibung dadurch weitaus verkürzt wird, und dass bei Fehlern bzgl. der Werten, die den Attributen der Komponenten zugewiesen wurden, nur die Properties-Datei verändert werden muss und nicht die GUI-Beschreibung. Das Verändern der GUI-Beschreibung würde dazu führen, dass die Klassen neu generiert werden müssten. Abgesehen von den Properties-Dateien besteht in der eigentlichen GUI-Beschreibung weiterhin die Möglichkeit, Werte festzulegen. Bei der Generation müssen die in der GUI-Beschreibung festgelegten Werte vorrangig behandelt werden. Grund dafür ist, dass im Vorfeld geprüft werden kann, ob die Werte in der GUI-Beschreibung definiert sind. Somit muss die Properties-Datei nicht zwingend nach dem richtigen Schlüssel durchsucht werden.

8.2 GUI-Beschreibungs-Model zur Verarbeitung im Generator

Abbildung 8.1: Teil 1: GUI-Beschreibungs-Model Version 1

Das Artefakt, welches beim dieserm Model im Mittelpunkt steht ist die *UI-Description* (siehe 8.1). Die aggregierten Artefakte sind aus dem Diagramm gut zu entnehmen. Die wichtigen Methoden umfassen nur die Getter- und Setter-Methoden der Attributer einer Klasse. Die anderen Methoden werden von Xtext ebenfalls generiert. Diese sind für diese Arbeit jedoch nicht von Belang. Die Klasse *DefinitionImpl* aggregiert weitere Artefakte des Models. Diese sind um die Übersicht zu wahren Abbildung 8.2 zu entnehmen.



Version 2

In dieser Version wurden an den Artefakten *AreaCount*, *TypeDefinition* und *AreaAssignment* keine Änderungen vorgenommen. Artefakte wie *Property* und *Refinement* sind hinzugekommen. Die weiteren Artefakte, die von UI-

DescriptionImpl aggregiert werden (siehe Abbildung 8.2), wurden verändert.

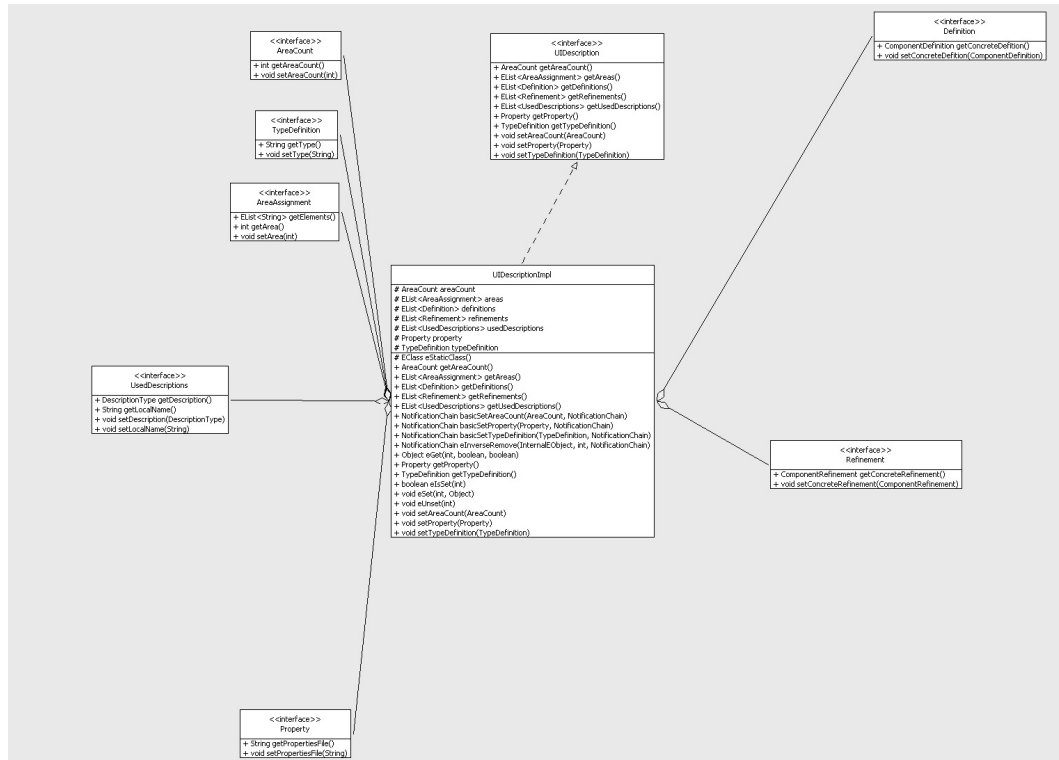


Abbildung 8.3: Teil 1: GUI-Beschreibungs-Model Version 2

Das Artefakt *Property* bildet die Property-Datei ab. Sie ist nicht zu verwechseln mit dem Artefakt *Properties*, welches die Eigenschaften von Komponenten abbildet. Abbildung 8.4 zeigt beide Artefakte auf.

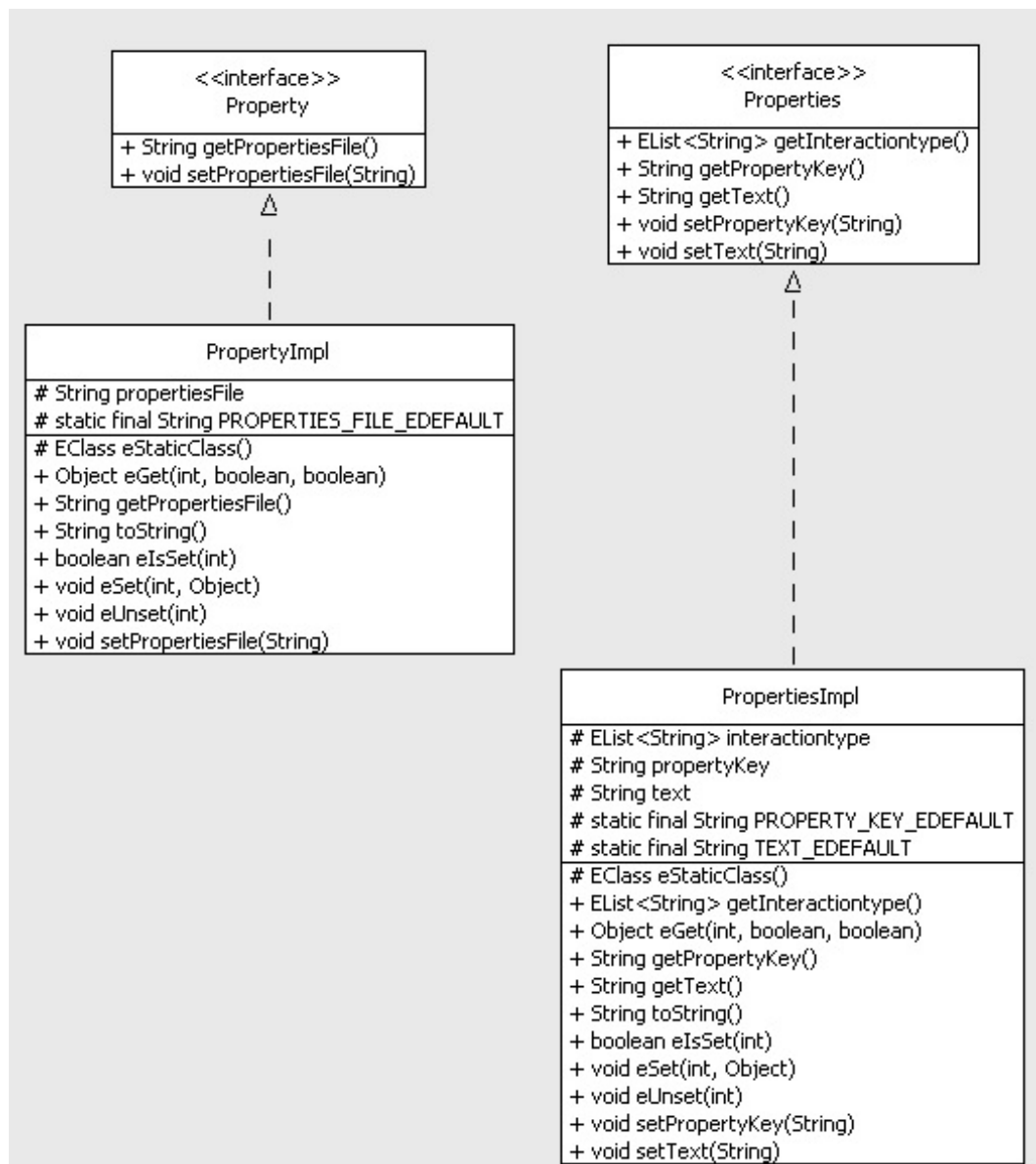


Abbildung 8.4: Teil 2: GUI-Beschreibungs-Model Version 2

Die *UsedDescription* enthält in dieser Version ein *DefinitionType*. Dieser bestimmt, ob es sich bei der importierten Komponente um ein beschriebenes GUI handelt, oder um eine komplexe Komponente, für die ein Input-Typ (*inputType*) festgelegt werden kann.

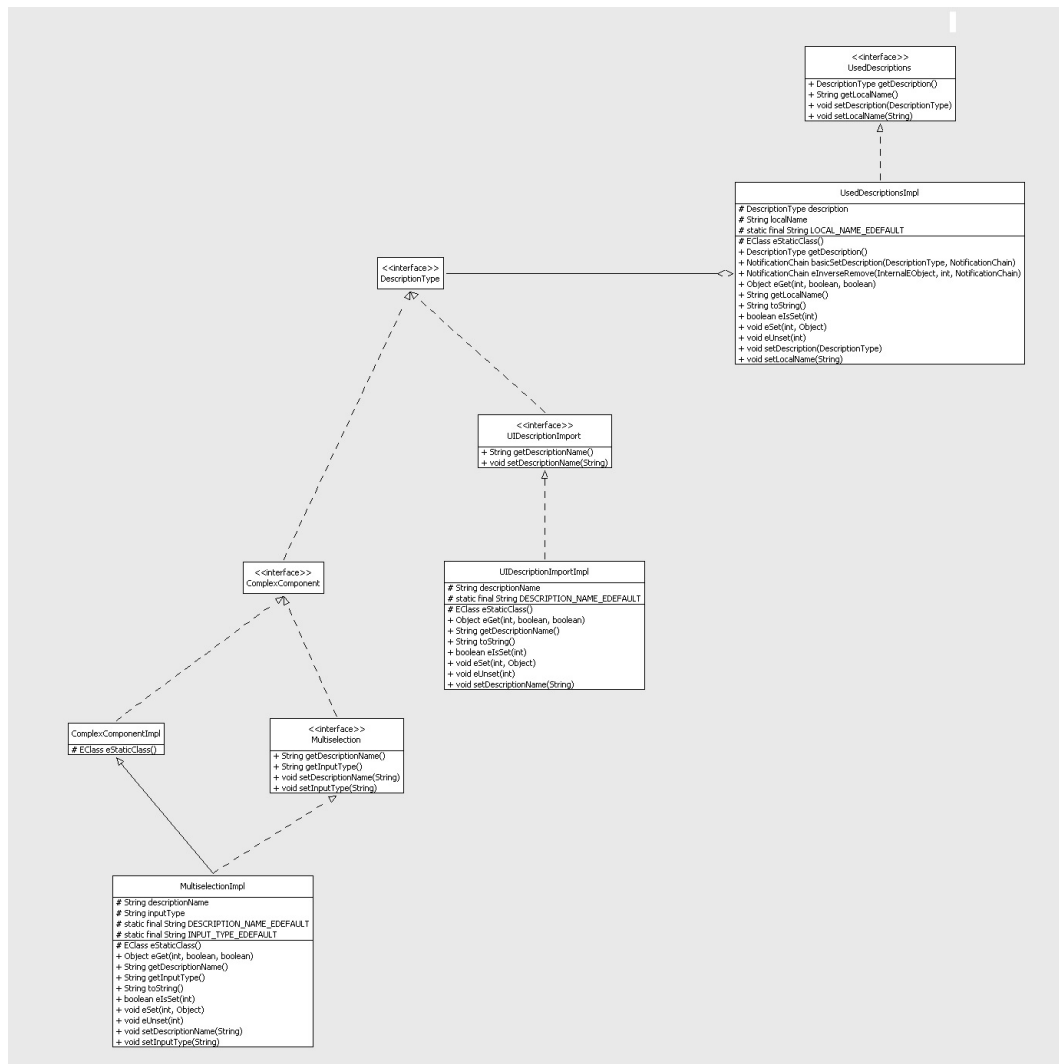


Abbildung 8.5: Teil 3: GUI-Beschreibungs-Model Version 2

Zwischen *Definition* und *Refinement* wird unterschieden. Die *Definition* bildet neu definierte Komponenten für das GUI ab. Ein *Refinement* hingegen bildet die Veränderten Komponenten importierter Komponenten ab (siehe Abbildung 8.6 und Abbildung 8.7).

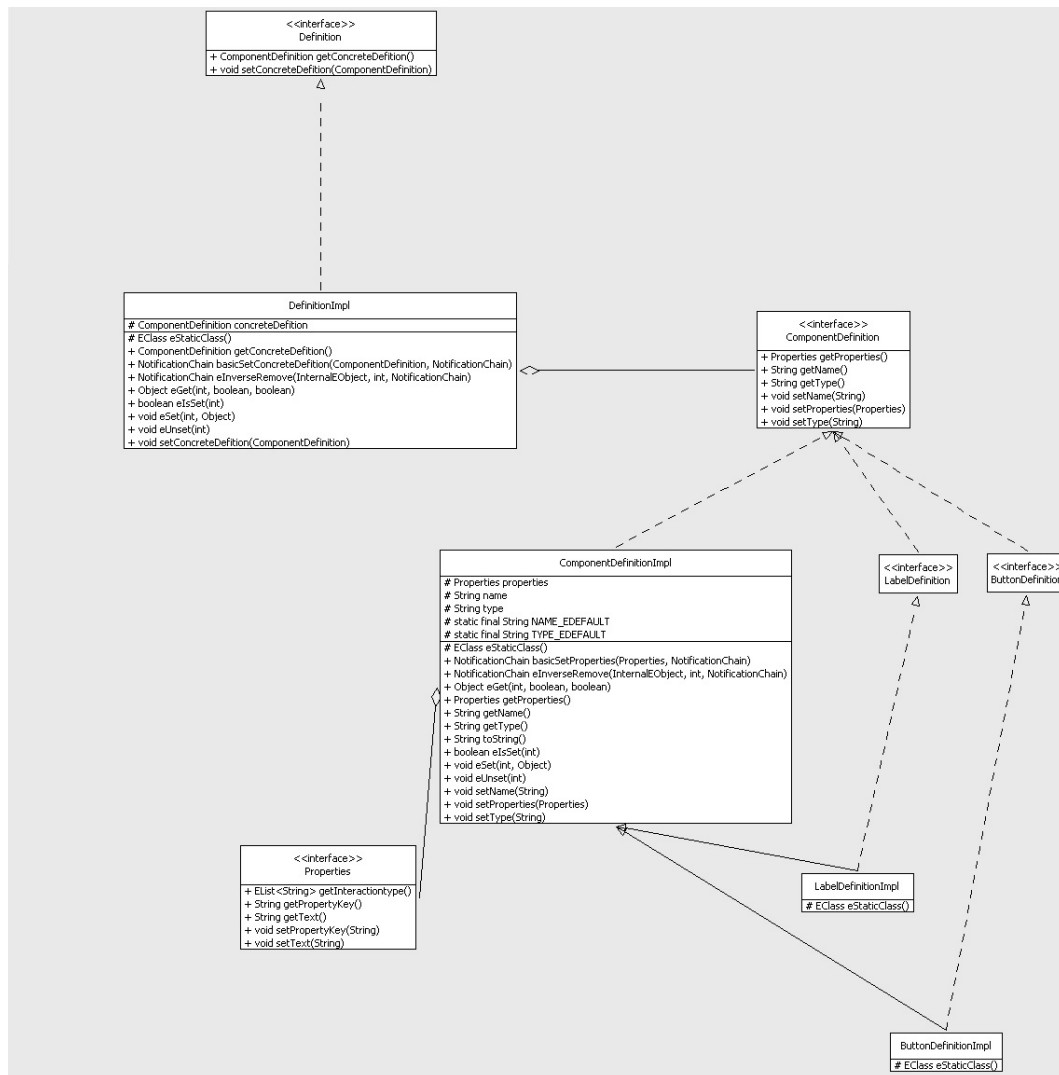


Abbildung 8.6: Teil 4: GUI-Beschreibungs-Model Version 2

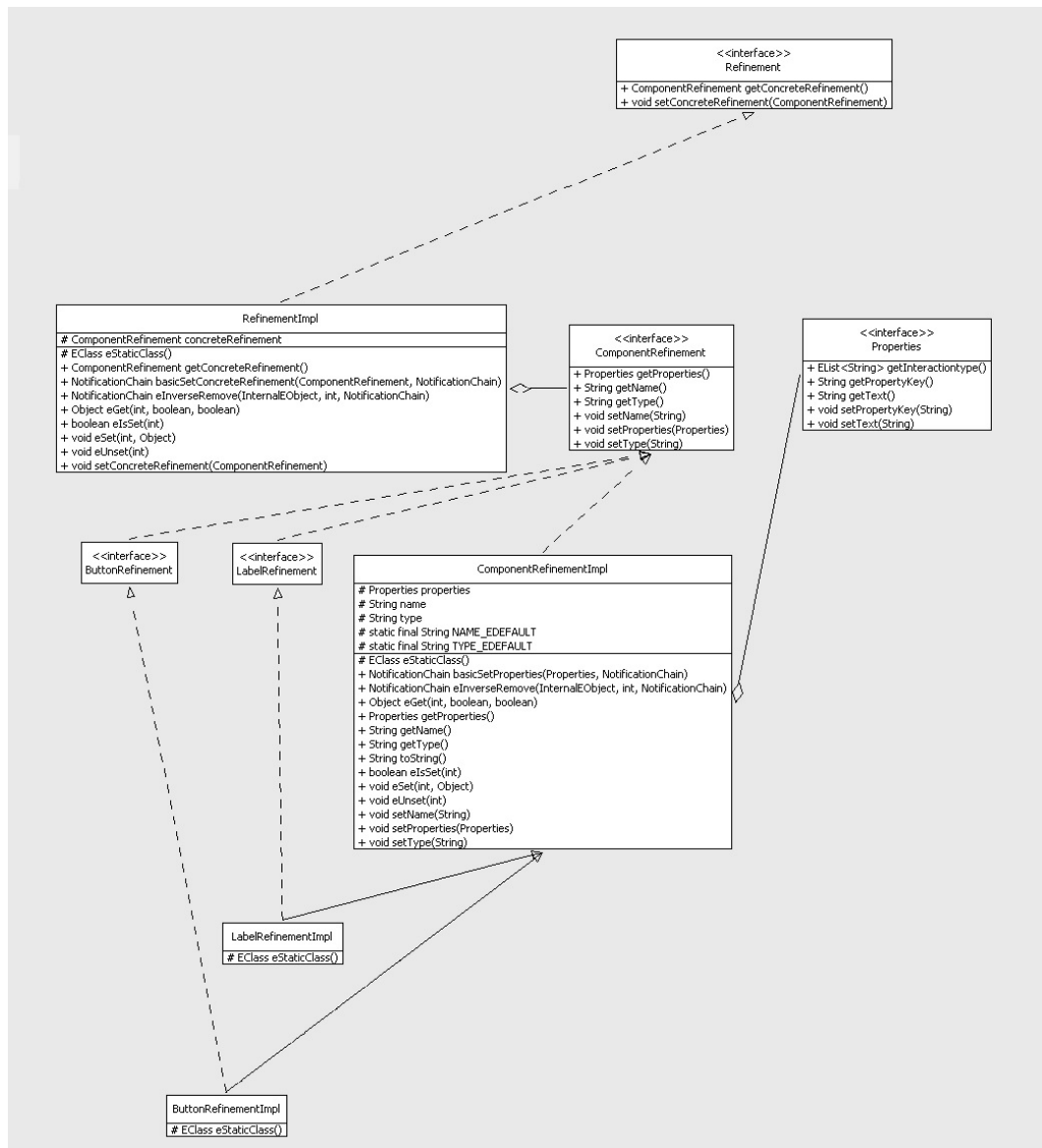


Abbildung 8.7: Teil 5: GUI-Beschreibungs-Model Version 2

Version 3

8.3 Konkrete Syntax

Die Syntax wird durch Beispiele beschreiben. Zu jeder Version ist ein minimaler DSL-Code zu finden. Besondere Änderungen bzgl. der konkreten Syntax sind jeweils nachfolgend genannt. Die Grammatiken befinden sich im Anhang 10. Sie sind dort ebenfalls in Versionen aufgeteilt.

Version 1

Listing 8.1: Syntax Version 1

```

1 Area count: 4
2 type: WINDOW use: "AnotherDescription"
3 DEF Label as "HEAD" :
4 END DEF
5 DEF Button as "Interactbt":
6     text="Interagiere"
7     interaction="btinteraction" type=CLICK with actions:type=UiAction element=
        "HEAD":Text="Du_hast_interagiert"
8 END DEF
9 DEF MultiSelection as "Multiselect" :
10     inputType="valuepackage.Values"
11     selectableValues="valuepackage.Values.asList()"
12 END DEF
13 Area:1<- "HEAD"
14 Area:2<- "AnotherDescription"
15 Area:3<- "Interactbt"
16 Area:4<- "Multiselect"

```

Die Bezeichnung *Area* wurde bewusst so gewählt, da dieser Begriff abstrakter ist als die in verschiedenen UI-Frameworks verwendeten Begriffe wie, Panel oder Pane. In der Syntax dieser DSL gilt es sich vor allem bzgl. des Aufbaus der GUI an keinem UI-Framework zu orientieren. Die einzelnen Komponentendefinitionen werden durch das Schlüsselwort *DEF* eingeleitet und durch das Schlüsselwort *END DEF* abgeschlossen. Der Definitionskopf wird durch das Zeichen *:* beendet. Dort sind die Pflichtfelder der Komponentendefinition zu finden (*Titel* und *Typ*). Bei der *MultiSelection*-Komponente fällt auf, dass ein Referenzwert verwendet wird, der in dieser Beschreibung nicht deklariert wurde (*valuepackage.Values*). Dabei handelt es sich um einen qualifizierten Namen einer Klasse.

Version 2

Die einfachste der Veränderungen bzgl. der Syntax in Version 2 ist die Festlegung der Properties-Dateien. In Listing 8.2 ist zu erkennen, dass eine entsprechende Datei festgelegt wurde und in den Komponenten entsprechende Schlüssel vergeben wurden. Das Label mit der Bezeichnung *OneLabel* enthält keinen Property-Key. In diesem Fall wird der Titel als solcher verwendet.

Listing 8.2: Properties in Version 2

```

1 type: WINDOW
2 get properties from: 'sources.ui.properties'

```

```

3 DEF Label as "OneLabel" END DEF
4 DEF Label as "AnotherLabel":
5     propertyKey='AnotherLabel2 '
6 END DEF

```

Aufgrund der Reduzierung der Meta-Daten für eine Interaktion stand die Frage offen, ob die Interaktionstypen einfach hintereinander mit Komma aufgezählt werden sollen, oder ob sie untereinander und jedes Mal wieder mit dem entsprechenden Schlüsselwort aufgezählt werden sollen. Aufgrund der Tatsache, dass in der deg höchstens 4 Interaktionstypen in einer Komponente verwendet werden, werden diese in der GUI-Beschreibung per DSL hintereinander mit Komma aufgezählt, wie in Listing 8.3 zu erkennen ist.

Listing 8.3: Interaktion in Version 2

```

1 DEF Button as "InteractButton":
2     interactiontype=Click , ChangeText
3 END DEF

```

Die komplexen Komponenten werden wie in Listing 8.4 mit der Komponente Multiselection gezeigt ist, über das Schlüsselwort *use* eingebunden werden. Der Input-Typ kann dabei optional innerhalb der Zeichen < und > angegeben.

Listing 8.4: Komplexe Komponenten in Version 2

```

1 type: WINDOW
2 use: Multiselection<'valuepackage.Values'> as: 'Multi'

```

Für die Zuweisung mehrerer Komponenten zu den Areas kamen zwei Lösungen in Betracht. Bei der einen finden die Definitionen der Komponenten zusammen mit der Zuweisung zu dem Area statt. Dies könnte bspw. wie in Listing ?? dargestellt werden.

Listing 8.5: Area-Zuweisung Möglichkeit 1 Version 2

```

1 Area count: 1
2 type: WINDOW
3 Area:1={
4 DEF Button as "Button:
5     text="Button"
6 END_DEF
7 DEF_Label_as_"Label":
8     text="Label"
9 END_DEF
10 }

```

Eine andere Möglichkeit wäre es, die aktuelle Form der Zuweisung zu verfeinern und somit die Komponenten bei der Zuweisung mit Komma getrennt von einander aufzählen. Die erste Möglichkeit würde sich sehr gut eignen, wenn nur die in der Datei definierten Komponenten dem Area zugewiesen werden müssten. Da die eingebundenen Komponenten auch Areas zugewordnet werden, würde für dieses Verfahren ein zusätzliches syntaktisches Konzept innerhalb der Area-Zuweisung benötigt werden. Um dies zu umgehen wurde die Entscheidung getroffen, das alte Verfahren zu verfeinern. Listing 8.6 ist ein Beispiel für die Area-Zuweisung von drei Komponenten zu entnehmen.

Listing 8.6: Area-Zuweisung Möglichkeit 2 Version 2

```

1 Area count: 1
2 type: WINDOW
3 DEF Label as "OneLabel" END DEF
4 DEF Label as "AnotherLabel" END DEF
5 DEF Button as "InteractButton":
6     interactiontype=Click ,ChangeText
7 END DEF
8 Area:1<- "OneLabel" , "InteractButton" , "AnotherLabel"

```

Das Überschreiben Werte der Komponenten, die in einer eingebundenen GUI-Beschreibung definiert wurden, können über das Schlüsselwort *REFINE* getätigt werden. Der erste Teil von Listing ?? zeigt die Originaldatei, deren Beschreibung eingebunden wird. Der zweite Teil zeigt, wie die Aufschrift einer Komponente *OverriddenButton* überschrieben wird.

Listing 8.7: Überschreiben einer eingebundenen Komponente Version 2

```

1 PART 1
2 Area count: 2
3 type: INNERCOMPLEX
4 DEF Label as "Label" :
5     text="Text"
6 END DEF
7 DEF Button as "Button":
8     text="AlterText"
9 END DEF
10 Area:1<- 'Label'
11 Area:2<- "Button"
12
13
14 PART 2
15 Area count: 1
16 type: WINDOW

```

```
17 use: "guidescription.LabelAndButton" as: 'Embedded'  
18 REFINE Button with name: 'OverriddenButton':  
19     text='NewText'  
20 END REFINE  
21 Area:1<- 'Embedded'
```

Sollten mehrere Komponenten eingebunden sein, in denen Komponenten mit den selben Namen definiert sind, muss der Titel der eingebundenen Ressource zur eindeutigen Identifikation stehen (siehe Listing 8.8)

Listing 8.8: Überschreiben einer eingebundenen Komponente mit Titel der Komponente Version 2

```
1 use: "guidescription.LabelAndButton" as: 'Embedded1'  
2 use: "guidescription.LabelAndTwoButton" as: 'Embedded2'  
3 REFINE Button with name: 'Embedded2.OverriddenButton':  
4     text='NewText'  
5 END REFINE
```

Kapitel 9

Entwicklung des Generators für das Generieren von Klassen für das Multichannel-Framework

9.1 WAM-GUI Architektur

Die GUI bei der deg teilt sich in drei Bereiche. Diese Dreiteilung entspricht dem WAM-Ansatz entnommen. Auf diesen Ansatz wird in Kapitel 9 etwas genauer eingegangen. Der erste Teil übernimmt die Beschreibung des Aufbaus des UIs (GUI-Part). Der zweite Teil enthält die Interaktionsformen mit den Komponenten, welche im GUI-Part deklariert wurden. Dieser Teil wird als *Interaction Part* (IP) bezeichnet. Der dritte Teil enthält den funktionalen Teil der GUI und wird *Functionally Part* (FP) genannt.

Bei den Überlegungen darüber, wie die DSL umzusetzen ist, wurde frühzeitig entschieden, dass der FP in der DSL nicht beschrieben wird. Es soll nur der GUI-Part neben einigen Inhalten aus dem IP beschrieben.

9.2 Syntax und Semantik für die Beschreibung der GUIs

9.3 Umsetzung des frameworkspezifischen Generators

Kapitel 10

Zusammenfassung und Ausblick

Test und Ausführung

Anhang

Grammatiken

Version 1

```

1 UIDescription :
2     areaCount=AreaCount
3     typeDefinition=(TypeDefinition)
4     usedDescriptions+=(UsedDescriptions)* &
5     inputTypes+=(inputType)* &
6     definitions+=(Definition)*
7     areas+=(AreaAssignment)*;
8
9 inputType :
10    'inputType=' type=STRING '_as_' name=STRING;
11
12 UsedDescriptions :
13    'use:_' description=UIDescriptionImport;
14
15 AreaCount :
16    'Area_count:_' areaCount=INT;
17
18 Definition :
19    'DEF_' concreteDefition=ComponentDefinition 'END_DEF';
20
21 TypeDefinition :
22    'type:_' type=TYPE;
23
24 TYPE :
25    ( 'WINDOW' | 'INNERCOMPLEX' );
26
27 UIDescriptionImport :
28    descriptionName=STRING ( '_As:_' localName=STRING )?;
29
30 AreaAssignment :
31    'Area:' area=INT '<-' element=STRING
32    | element=STRING '->' 'Area:' area=INT ;
33
34 ComponentDefinition :
35    LabelDefinition | ButtonDefinition | MultiSelectionDefinition;

```

```

36
37 MultiSelectionDefinition :
38     type='MultiSelection' ' _as_ ' name=STRING ':' ( 'inputType=' inputType=STRING
39     ( 'selectableValues=' selectableValuesLocation=STRING ( 'selectedValues='
        selectableValuesLocation=STRING)?)?)?;
40
41 ButtonDefinition :
42     type='Button' ' _as_ ' name=STRING ':' '
43     ( 'text=' text=STRING)?
44     ( 'interaction=' interaction=Interaction)?;
45
46 Interaction :
47     name=STRING ' _type=' Interactiontype ' _with_actions=' actions+=(UIAction)
        *;
48
49 LabelDefinition :
50     type='Label' ' _as_ ' name=STRING ':' '
51     ( 'text=' text=STRING)?;
52
53
54 UIAction :
55     'type=' type='UiAction'
56     'element=' uiElementName=STRING ':' '
57     properties+=(Property)*;
58
59 Property :
60     CommonProperty;
61
62 CommonProperty :
63     (name=CommonPropertyType '=' value=STRING);
64
65 CommonPropertyType :
66     'Text' ;
67
68 Interactiontype :
69     'CLICK' ;

```

Version 2

```

1 UIDescription :
2     areaCount=AreaCount
3     typeDefinition=(TypeDefinition)
4     (property=(Property))?
5     usedDescriptions+=(UsedDescriptions)*
6     refinements+=(Refinement)*
7     definitions+=(Definition)*
8     areas+=(AreaAssignment)*;
9
10 Refinement :
11     'REFINE' concreteRefinement=ComponentRefinement 'END_REFINE';
12

```

```

13 ComponentRefinement:
14     LabelRefinement | ButtonRefinement;
15
16 ButtonRefinement:
17     type='Button' '_with_name:_' name=STRING
18     properties=(Properties)?;
19
20 LabelRefinement:
21     type='Label' '_with_name:_' name=STRING
22     properties=(Properties)?;
23
24 Property:
25     'get_properties_from:' propertiesFile=STRING;
26
27 UsedDescriptions:
28     'use:_' description=DescriptionType ('_as:_' localName=STRING)?;
29
30 DescriptionType:
31     UIDescriptionImport | ComplexComponent;
32
33 AreaCount:
34     'Area_count:_' areaCount=INT;
35
36 Definition:
37     'DEF_' concreteDefition=ComponentDefinition 'END_DEF';
38
39 TypeDefinition:
40     'type:_' type=Type;
41
42 Type:
43     ('WINDOW' | 'INNERCOMPLEX');
44
45 UIDescriptionImport:
46     descriptionName=(STRING);
47
48 ComplexComponent:
49     (Multiselection);
50
51 Multiselection:
52     descriptionName='Multiselection' ('<' inputType=STRING '>')?;
53
54 AreaAssignment:
55     'Area:' area=INT '<-' elements+=(STRING)+
56     | elements+=(STRING)+ '->' 'Area:' area=INT;
57
58 ComponentDefinition:
59     LabelDefinition | ButtonDefinition;
60
61 ButtonDefinition:
62     type='Button' '_as_' name=STRING
63     properties=(Properties)?;

```

64

65 Properties :

66 ': ' ('propertyKey=' propertyKey=STRING)?

67 ('text=' text=STRING)?

68 ('interactiontype=' interactiontype+=(Interactiontype)+)?;

69

70 LabelDefinition :

71 type='Label' 'as' name=STRING

72 properties=(Properties)?;

73

74 Interactiontype :

75 'Click' | 'ChangeText';

76

77 terminal WS:

78 ('_' | '\t' | '\r' | '\n' | ',')+;

Glossar

Förderantrag [...] ist ein Antrag, den der Begünstigte einreicht, wenn er sich eine Maßnahme fördern lassen möchte [dat14]. 5

GridBagLayout ist ein Layout Manager innerhalb von Swing, welcher die Komponenten horizontal, vertical und entlang der Grundlinie anordnet. Dabei müssen die Komponenten nicht die gleiche Größe haben [Oraa]. 9

GUI ist die Schnittstelle zwischen dem Benutzer und dem Programm. 1

. 5

Swing ist ein UI-Framework für Java Applikationen [Orab]. xiii, 8, 9

Traditionelle UI-Entwicklung Bei der traditionellen UI-Entwicklung wird mit traditionellen UI-Toolkits gearbeitet. Bei diesen Toolkits wird Aufbau der GUI genau beschrieben. Für die Interaktion mit den UI-Widgets, werden Listener implementiert, die auf andere Events reagieren, die von anderen Widgets erzeugt generiert wurden. Events können zu unterschiedlichen Zeitpunkten generiert werden und es wird nicht festgelegt in welcher Reihenfolge sie bei anderen Widgets ankommen. [KB11]. 1

Usability beschreibt die Nutzerfreundlichkeit einer GUI, sowie auch die Nutzerfreundlichkeit einer Software. 1

. 8, 9

Zuwendungs-Berechner ist ein Werkzeug innerhalb von profil c/s. Mit diesem Werkzeug kann der Sachbearbeiter die Zuwendung, die dem Antragsteller bewilligt werden soll, nach einem standardisierten Verfahren berechnen (siehe Abschnitt "Algorithmen"). Das Ergebnis wird im Zuwendungsblatt dokumentiert, das auch später mit demselben Werkzeug angesehen werden kann [deG07]. xiv

Zuwendungsblatt ist die grafische Dokumentation der Ergebnisse des Zuwendungs-Berechners innerhalb von profil c/s. xiv, 5

Literaturverzeichnis

- [Aho08] AHO, ALFRED V: *Compiler: Prinzipien, Techniken und Werkzeuge*. Pearson Studium, 2008.
- [BCK08] BRAUER, JOHANNES, CHRISTOPH CRISEMAN und HARTMUT KRISEMAN: *Auf dem Weg zu idealen Programmierwerkzeugen - Bestandsaufnahme und Ausblick*. Informatik Spektrum, 31(6):580–590, 2008.
- [BPL13] BACIKOVÁ, MICHAELA, JAROSLAV PORUBÁN und DOMINIK LAKATOS: *Defining Domain Language of Graphical User Interfaces*. In: *OASIS-OpenAccess Series in Informatics*, Band 29. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.
- [bra10] *Eine DSL für Harel-Statecharts mit PetitParser*. Arbeitspapiere der Nordakademie. Nordakad., 2010.
- [dat14] DATA EXPERTS GMBH: *Förderantrag*. Profil Wiki der deg, März 2014. Zuletzt eingesehen am 02.12.2014.
- [deG07] GMBH DATA EXPERTS: *Detaillkonzept ELER/i-Antragsmappe*, Januar 2007. Letzte Änderung am 01.12.2014.
- [DM14] DANIEL, FLORIAN und MARISTELLA MATERA: *Model-Driven Software Development*. In: *Mashups, Data-Centric Systems and Applications*, Seiten 71–93. Springer Berlin Heidelberg, 2014.
- [FP11] FOWLER, MARTIN und REBECCA PARSON: *Domain-Specific Languages*. Addison-Wesley, 2011.

- [Gal07] GALITZ, WILBERT O.: *The Essential Guide to User Interface Design - An Introduction to GUI Design Principles and Techniques*. John Wiley Sons, New York, 2007.
- [gho11] *DSLs in Action*. Manning Publications Co., 2011.
- [Gun14] GUNDERMANN, NIELS: *Entwicklung einer Grammatik für eine DSL mit xText am Beispiel einer Sprache zur Definition von Pflichtprüfungen in profil c/s*, 2014. Praxisbericht.
- [Hed12] HEDTSTUECK, ULRICH: *Einführung in die Theoretische Informatik*, Band 5. Auflage. Oldenbourg Verlag, 2012.
- [KB11] KRISHNASWAMI, NEELAKANTAN R. und NICK BENTON: *A Semantic Model for Graphical User Interfaces*. Microsoft Research, September 2011. Verfügbar unter URL:.
- [LW] LU, XUDONG und JIANCHENG WAN: *Model Driven Development of Complex User Interface*. Technischer Bericht, Shandong University. Verfügbar unter URL: <http://ceur-ws.org/Vol-297/paper7.pdf>.
- [mds06] *Model-Driven Software Development*. John Wiley Sons Ltd, Februar 2006.
- [MHP99] MYERS, BRAD, SCOTT E. HUDSON und RANDY PAUSCH: *Past, Present and Future of User Interface Software Tools*. Technischer Bericht, Carnegie Mellon University, September 1999. Verfügbar unter URL: <http://www.cs.cmu.edu/~amulet/papers/futureofhci.pdf>.
- [Oraa] ORACLE: *Class GridBagLayout*. URL: <https://docs.oracle.com/javase/7/docs/api/java/awt/GridBagLayout.html>. Zuletzt eingesehen am 02.12.2014.
- [Orab] ORACLE: *Swing*. URL: <https://docs.oracle.com/javase/jp/8/technotes/guides/swing/index.html>. Zuletzt eingesehen am 02.12.2014.

- [RDGN10] RENGGLI, LUKAS, STEPHANE DUCASSE, TUDOR GÎBRA und OSCAR NIERSTRASZ: *Practical Dynamic Grammars for Dynamic Languages*. In: *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, 2010. Online verfügbar unter URL: http://bergel.eu/download/Dyla2010/dyla10_submission_4.pdf. Zuletzt eingesehen am 6.1.2015.
- [Roa09] ROAM, DAN: *The Back of the Napkin (Expanded Edition) - Solving Problems and Selling Ideas with Pictures*. Penguin, New York, Expanded Auflage, 2009.
- [SKNH05] SUKAVIRIYA, NOI, SANTHOSH KUMARAN, PRABIR NANDI und TERRY HEATH: *Integrate Model-driven UI with Business Transformations: Shifting Focus of Model-driven UI*. Technischer Bericht, IBM T.J. Watson Research Center, Oktober 2005. Verfügbar unter URL: <http://www.research.ibm.com/people/p/prabir/MDDAUI.pdf>.
- [Ste07] STECHOW, DIRK: *JWAMMC - Das Multichannel-Framework der data-experts gmbh*. Vortrag, Dezember 2007.
- [Use12] USERLUTIONS GMBH: *3 Gründe, warum gute Usability wichtig ist*. URL: <http://rapidusertests.com/blog/2012/04/3-gute-grunde-fuer-usability-tests/>, April 2012. Zuletzt eingesehen am 01.12.2014.
- [VBK⁺13] VÖLTER, MARKUS, SEBASTIAN BENZ, LENNART KATS, MATS HELANDER, EELCO VISSER und GUIDO WACHSMUTH: *DSL Engineering*. CreateSpace Independent Publishing Platform, 2013.