



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

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Neubrandenburg, Februar 2015

Niels Gundermann

Inhaltsverzeichnis

Abbildungsverzeichnis	V
Tabellenverzeichnis	VII
Listings	VIII
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 mehrere Plattfor- men in der <i>data experts GmbH</i> (Ist-Zustand)	4
2.2.1 Multichannel-Framework	4
2.2.2 JWAM	6
2.3 Probleme des Multichannel-Frameworks	8
2.4 Zielsetzung	9
3 Domänenspezifische Sprachen	10
3.1 Begriffsbestimmungen	10
3.2 Anwendungsbeispiele	14
3.3 Model-Driven Software Development (MDSD)	14
3.4 Abgrenzung zu GPL	15
3.5 Vor- und Nachteile einer DSL gegenüber einer GPL	16
3.6 Interne DSLs	23
3.6.1 Implementierungstechniken	23
3.7 Externe DSLs	25
3.7.1 Implementierungstechniken	25

3.8	Nicht-textuelle DSLs	29
4	GUI-DSL	30
4.1	Motivation des Ansatzes	30
4.2	Anforderungen an die GUI-DSL	31
5	Entwicklung einer Lösungsidee	33
5.1	Allgemeine Beschreibung der Lösungsidee	33
5.2	Konzept	33
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	Meta Programming System	36
6.2	Vergleich und Bewertung der vorgestellten Frameworks	37
7	Festlegungen für die Entwicklung des Prototypen	40
7.1	Vorgehensmodell	40
7.2	Grobkonzept der DSL-Umgebung (Vision)	42
7.2.1	1. Iteration	42
7.2.2	2. Iteration	46
7.2.3	3. Iteration	48
8	Entwicklung einer DSL zur Beschreibung der GUI in profil c/s	51
8.1	1. Iteration	51
8.2	2. Iteration	57
8.3	3. Iteration	63
9	Entwicklung des Generators zur Generierung von Klassen für das Multichannel-Framework	72
9.1	Beschreibung der GUI-, FP- und IP-Klassen	73
9.2	Umsetzung des frameworkspezifischen Generators	74
10	Zusammenfassung und Ausblick	85
A	Zuwendungsblatt für Web- und Standalone-Client	X

B	Generierte Explorer-GUI	XIII
C	Inhalt des beiliegenden Datenträgers	XIV
	Glossar	XV
	Literaturverzeichnis	XVIII

Abbildungsverzeichnis

1	Architektur des Multichannel-Frameworks	5
2	Komplexes Werkzeug mit Benutzt-Beziehung	7
3	Die grundlegenden Ideen hinter dem MDSD	15
4	Parsen allgemein	26
5	Funktionsweise von Parser-Kombinatoren	28
6	Grundlegendes Konzept	34
7	Grundlegende Idee für den Prototypen	34
8	Inkrementelles Modell	41
9	Konzeption der DSL-Umgebung (1. Iteration)	43
10	Beispiel: Suchfeld für Name	45
11	Beispiel: Suchmaske	45
12	Konzeption der DSL-Umgebung (2. Iteration)	47
13	Konzeption der DSL-Umgebung (3. Iteration)	48
14	1. Iteration: UIDescription	54
15	1. Iteration UsedUIDescription	54
16	Teil 3: GUI-Beschreibungsmodell Version 1	55
17	Teil 4: GUI-Beschreibungsmodell Version 1	55
18	Teil 1: GUI-Beschreibungsmodell Version 2	58
19	Teil 2: GUI-Beschreibungsmodell Version 2	58
20	Teil 3: GUI-Beschreibungsmodell Version 2	59
21	Teil 4: GUI-Beschreibungsmodell Version 2	60
22	Teil 5: GUI-Beschreibungsmodell Version 2	60
23	3. Iteration: UIDescription	66
24	3. Iteration: Definition	67

25	3. Iteration: Button	67
26	3. Iteration: TabView	68
27	3. Iteration: TabView	68
28	3. Iteration: TabView	69
29	Aufbau eines Explorers	72
30	Generierte GUI des Inhalts- und Verweisebaums	78
31	Standalone-Client: Zuwendungsblatt	X
32	Web-Client: Zuwendungsblatt	XI
33	Multiselection-Komponenten	XII
34	Generierte Explorer-GUI	XIII

Tabellenverzeichnis

1	Prioritäten der Anforderungen an die <i>GUI</i>	4
2	Implementierung einfacher Grammatikregeln mit einem RD- Parser	27
3	Bewertung der Frameworks für die Entwicklung von DSLs . .	38
4	Basiskomponenten mit spezifischen Metadaten	65

Listings

1	Beispiel: Fluent Interface	24
2	1. Iteration: Syntax	56
3	2. Iteration: Properties	61
4	2. Iteration: Interaktion	61
5	2. Iteration: Definition komplexer Komponenten	61
6	2. Iteration: Button und Label (1)	61
7	2. Iteration: Area-Zuweisung (2)	62
8	2. Iteration: Verändern von Komponenten eingebundener GUI-Beschreibungen	62
9	2. Iteration: Verändern von Komponenten eingebundener GUI-Beschreibungen mit Namensüberschneidung	63
10	3. Iteration: Properties- und Layout-Dateien	69
11	3. Iteration: Eingebundene <i>GUI-Komponenten</i>	70
12	3. Iteration: Button und Label	70
13	3. Iteration: Textfield und Textarea	70
14	3. Iteration: Table und Tree	70
15	3. Iteration: TabView	71
16	3. Iteration: TabView	71
17	3. Iteration: Struktur	71
18	GUI-Skript für den Inhaltsbaum	74
19	GUI-Skript für den Verweisebaum	74
20	Speichern der Importe und der globalen Variablen	74
21	Generierung eines Innercomplex	75
22	Generierung der Methode <i>init</i> der GUI-Klassen	76
23	Generierung eines Labels	76
24	Generierung eines Trees	77

25	GUI-Skript für das Explorer-GUI	78
26	Generierung eines Windows	78
27	Generierung der Einbindung anderer GUI-Skripte	79
28	Generierung einer Interchangeable-Komponente	79
29	Generierung der FP-Klasse	80
30	Generierung der IP-Klasse	80
31	Generierung der Interaktionsformen	81
32	Generierung der Standard-Interaktionsformen von Trees . . .	82
33	Generierung einer Interaktionsform	82
34	Generierung der Kommandoinitialisierung	83
35	Generierung der Methoden zur Bestimmung der auszufüh- renden Aktionen bei einer Interaktion	83

Kapitel 1

Motivation

In der heutigen Zeit werden Programme auf vielen unterschiedlichen Geräten wie z. B. PCs, Smartphones, Tablets ausgeführt. Deswegen ist die *Usability* ein wichtiger Faktor bei der Entwicklung von Anwendungen. Eine „[...] schlechte Usability führt zu Verwirrung und Miss- bzw. Unverständnis“ [Use12] beim Kunden, wodurch letztendlich Umsatz verloren geht. Die *Usability* wird hauptsächlich vom *Graphical User Interface (GUI)* bestimmt. Folglich ist die Benutzeroberfläche neben der internen Umsetzung ein wichtiger Faktor für den Erfolg einer Anwendung (vgl. [LW07]).

Wenn ein Programm auf unterschiedlichen Geräten ausgeführt wird, muss der Entwickler bei der *traditionellen GUI-Entwicklung* mehrere GUIs manuell implementieren. Demnach werden mehrere GUIs mit unterschiedlichen Toolkits oder Frameworks entworfen. Diese Frameworks haben einen starken imperativen Charakter, sind schwer zu erweitern und verhalten sich je nach Plattform unterschiedlich (vgl. [KB11]). Daraus folgt, dass die Entwickler bei dem *traditionellen* Ansatz, die GUI für jedes Framework explizit beschreiben müssen.

Ein anderer Ansatz zur Beschreibung von Benutzeroberflächen ist das *Model-Driven Development* (siehe Kapitel 3.3). Damit sollen bspw. GUIs anhand der modellierten Funktionalitäten automatisch erzeugt werden (vgl. [SKNH05]). Laut Myers et al. wurden die Darstellungen dieser generierten GUIs in der Vergangenheit von den Darstellungen *traditionell* implementierter Benutzerschnittstellen übertroffen (vgl. [MHP99]). Der Grund dafür ist, dass die Abstraktionsebene, auf der die Beschreibung der GUIs beim *Model-Driven De-*

velopment stattfindet, oft nicht an die Gestaltung der *GUI*, sondern an fachliche Konzepte der *Domäne* angepasst wird.

Daraus ergibt sich die Überlegung, ob diese beiden Ansätze zur Implementierung von *GUIs* (*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 unternommen (vgl. [BPL13]). Dort wurde nachgewiesen, dass eine *GUI* eine Sprache definiert.

In dieser Arbeit wird versucht, den kombinierten Ansatz in einem speziellen Fachbereich umzusetzen, um die Unterstützung einer *GUI* auf unterschiedlichen Plattformen zu gewährleisten. Bei dieser Umsetzung wird sich auf die *GUIs* der Anwendung *profil c/s* bezogen. *Profil c/s* ist eine *JEE-Anwendung*, die *InVeKoS* umsetzt und von der *data experts GmbH* entwickelt wird.

Kapitel 2

Problembeschreibung und Zielsetzung

2.1 Allgemeine Anforderungen an Benutzeroberflächen von *profil c/s*

Die erste Anforderung bezieht sich auf die Plattformen, auf welche der Client von *profil c/s* dargestellt werden muss. Dieser soll sowohl in Web-Browsern (*Web-Client*) als auch standalone auf einem PC (*Standalone-Client*) ausgeführt werden können.

Um die Darstellung auf unterschiedlichen Plattformen umzusetzen, werden unterschiedliche Frameworks zur Darstellung der *GUI* verwendet. Dabei besteht die Möglichkeit, dass verwendete Frameworks veralten, woraus sich der Bedarf an einer Überführungsmöglichkeit der *GUIs*, die mit älteren Frameworks dargestellt werden, in *GUIs*, die mit aktuellen Frameworks dargestellt werden, ergibt.

Da die Nutzer an einen bestimmte Aufbau *GUIs* gewöhnt sind, ist es von Vorteil, wenn die *GUI* des Clients auf unterschiedlichen Plattformen den gleichen Aufbau hat. Von daher ist die optische Ähnlichkeit der *GUIs* auf unterschiedlichen Plattformen eine weitere Anforderung.

Um eine effiziente Arbeitsweise zu bewirken, ist es wichtig, dass die verwendeten Frameworks um wiederverwendbare Komponenten erweitert werden können. So ist es möglich, redundante Implementierungen zu verallgemeinern und letztendlich zu reduzieren.

Abschließend ist die Ausdruckskraft der Syntax, für die Entwicklung der *GUIs*, als Kriterium zu nennen. Eine ausdrucksstarke Syntax fördert die Lesbarkeit des Quellcodes und damit letztlich dessen Verständnis sowie die Effizienz mit der die *GUIs* entwickelt werden (vgl. [VBK⁺13, S.70]).

Diese Anforderungen an die *GUIs* haben in der *data experts GmbH* unterschiedliche Prioritäten (1 = höchste Priorität, 3 = niedrigste Priorität), die in folgender Tabelle beschrieben werden.

Nr.	Anforderung	Priorität
AA1	Bereitstellung für den Standalone- und Web-Bereich	1
AA2	Überführungsmöglichkeit in andere Frameworks	2
AA3	Ähnlicher Aufbau auf unterschiedlichen Plattformen	2
AA4	Erweiterungsmöglichkeiten der verwendeten Frameworks	1
AA5	Ausdrucksstarke Syntax	3

Tabelle 1: Prioritäten der Anforderungen an die *GUI*

2.2 Umsetzung der Benutzerschnittstellen für mehrere Plattformen in der *data experts GmbH* (Ist-Zustand)

2.2.1 Multichannel-Framework

Die Clients werden in der Programmiersprache *Java* entwickelt. Für die Realisierung des *Standalone-Clients* wird in der *data experts GmbH* das Framework *Swing* verwendet. Für den *Web-Client* wird auf *wingS* zurückgegriffen. Um eine Vorstellung des Ist-Zustands zu vermitteln, sind in Abbildung 31 und in Abbildung 32 (siehe Anhang A) die *GUIs* eines *Zuwendungsblatts* und eines *Förderantrags* für den *Web-Client* und den *Standalone-Client* abgebildet.

In diesen *GUIs* ist nur ein bestimmter Teil für die fachlichen Informationen relevant. Dies sind lediglich die Tabelle, die darunter stehenden Schaltflächen sowie das Bemerkungsfeld (im *Web-Client* auf der rechten Seite und im *Standalone-Client* in der Mitte). Dieser Bereich der *GUI* ist in beiden Cli-

ents gleich aufgebaut. Andere Teile der *GUI* haben derzeit keine einheitlichen Aufbau, was den unterschiedlichen Frameworks für die Umsetzung von *Web*- und *Standalone-Client* geschuldet ist.

Dass der Aufbau der *GUI* in beiden Clients ähnlich ist, liegt an der Umsetzung der *GUI*, die im Folgenden erläutert wird.

Aufgrund von Anforderung *AA1* wurden in der Vergangenheit zwei *GUIs* mit unterschiedlichen Frameworks von der *data experts GmbH* entwickelt. Dieses Verfahren erwies sich mit komplexer werdenden *GUIs* als sehr ineffizient. Daher hat die *data experts GmbH* eine Lösung erarbeitet, mit der es möglich ist, eine einmal beschriebene *GUI* auf mehrere Plattformen zu portieren. Durch diese Abstraktion wird der Aufwand der Entwicklung neuer *GUIs* stark reduziert. Zugleich fördert die einmalige Beschreibung auch einen ähnlichen Aufbau der *GUIs* im *Web*- und *Standalone-Client*, was der Anforderung *AA3* nachkommt. Die Lösung der *data experts GmbH* ist das *Multichannel-Framework (MCF)*.

Die Architektur des *Multichannel-Frameworks* ist Abbildung 1 zu entnehmen. Innerhalb des *MCF* werden die *GUIs* mittels so genannter *Präsentationsformen* beschrieben.

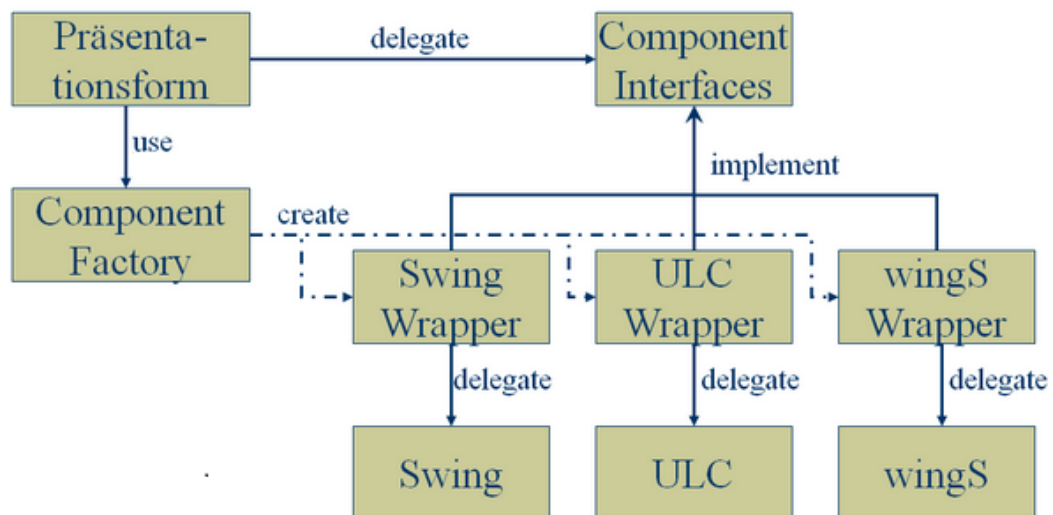


Abbildung 1: Architektur des Multichannel-Frameworks (vgl. [Ste07])

Aus *Präsentationsformen* können mithilfe der *Component-Factories* *GUIs* erzeugt werden, die auf unterschiedlichen Frameworks basieren und das *Component-Interface* implementieren. (Bei den verwendeten Frameworks handelt es sich

um *Swing*, *ULC* und *wingS*. *ULC* wird in der *data experts GmbH* nicht mehr eingesetzt.) Das *Component-Interface* wird für die Interaktion mit den Komponenten der unterschiedlichen Frameworks benötigt. Mit dem *MCF* ist die *data experts GmbH* in der Lage ihre *GUIs* für *Swing* und für *wingS* mit nur einer *GUI*-Beschreibung zu erzeugen.

Die Anbindung anderer Frameworks ist bei Betrachtung der Architektur des *MCF* unproblematisch. Dadruch scheint die *data experts GmbH* mit diesem Ansatz auch auf den Einsatz neuer Frameworks (siehe Anforderung AA2) vorbereitet zu sein.

Wie bereits erwähnt, werden die Clients mit *Java* entwickelt. Für die Architektur der Clients wird der *WAM-Ansatz* verwendet, welcher im Folgenden Kapitel kurz erläutert wird.

2.2.2 JWAM

„JWAM ist eine Realisierung des WAM-Ansatzes in der Programmiersprache *Java*“ [Sau03, S.30]

Die Bezeichnung *WAM* steht für *Werkzeug & Material-Ansatz*. Es handelt sich dabei um einen Ansatz zur Softwarearchitektur, welcher den anwendungsorientierten Ansatz der Softwareentwicklung fördert. Der Benutzer der Software steht im Mittelpunkt, wodurch die Gestaltung der Funktionalitäten, Benutzerschnittstellen und die Schnittstelle des Entwicklungs- und Implementierungsprozesses beeinflusst werden (vgl. [Sch05, S.13]). Weiterhin werden Entwurfsmetaphern verwendet, die den Entwicklern und Anwendern das Entwickeln und Verstehen der Software vereinfachen sollen (vgl. [Sau03, S.30]). Diese Entwurfsmetaphern beschreiben „[...] *Elemente und Konzepte der Anwendung durch bildhafte Vorstellungen von realen Gegenständen* [...]“ [Sau03, S.30]. Die grundlegenden Metaphern werden im Folgenden kurz erläutert. Ein *Material* kann nicht direkt vom Nutzer bearbeitet werden. Es besitzt jedoch eine Schnittstelle, die fachliche Operationen erlaubt. Diese Operationen können bspw. von einem *Werkzeug* aufgerufen werden, um den Zustand des Materials zu verändern. Dabei verfügen *Werkzeuge* über eine Präsentation und geben somit eine Handhabung vor.

Es wird zwischen zwei Arten von Werkzeugen - *monolithische* und *komplexe Werkzeuge* (vgl. [Hof06, S.5]). Da in der *data experts GmbH* hauptsächlich *komplexe Werkzeuge* Anwendung finden, werden die *monolithische Werkzeuge* in dieser Arbeit nicht weiter erläutert. *Komplexe Werkzeuge* gliedern sich in Oberfläche, Interaktion und Fachlogik auf. Dabei muss sich die Funktionskomponente (*FunctionPart* - *FP*) vollständig von der *GUI* abstrahieren und sich somit auf die Fachlogik beschränken. Zwischen diesen Komponenten steht eine Interaktionskomponente (*InteractionPart* - *IP*), die für die Abstraktion Sorge trägt. Eine *Werkzeugklasse* umschließt diese drei Komponenten, wie Abbildung 2 zu entnehmen ist (vgl. [Hof06, S.5f]).

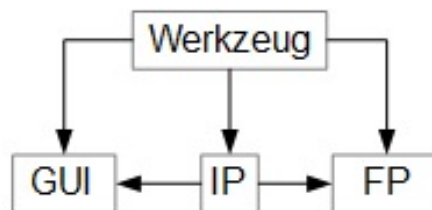


Abbildung 2: Komplexes Werkzeug mit Benutzt-Beziehung

Automaten können selbstständig Routinearbeiten erledigen (vgl. [Sau03, S.30]). Sie können ebenfalls *Materialien* bearbeiten (vgl. [Sch05, S.14]).

Gegenstände, die mit diesen Metaphern assoziiert werden, können in *Arbeitsumgebungen* abgelegt werden. Dabei werden zwei *Arbeitsumgebungen* unterschieden. Erstens die persönlichen Arbeitsplätze der Benutzer und zweitens Räume der *Arbeitsumgebung*, die für alle zugänglich sind und dort somit die Zusammenarbeit stattfinden kann (vgl. [Sau03, S.31], [Sch05, S.15]). Darüber hinaus gibt es *fachliche Services*, die als Dienstleister Funktionalität zur Verfügung stellen. Diese können *Materialien* verwalten, verfügen jedoch über keine eigene Präsentation (vgl. [Sau03, S.30f]).

Durch fachliche Services ist mit dem WAM-Ansatz auch *Multi-Channeling* möglich (vgl. [Sau03, S.42]). Das *Multi-Channeling* des WAM-Ansatzes setzt dabei auf einer anderen Ebene als das MCF an. Das *Multi-Channeling* des WAM-Ansatzes basiert darauf, dass „[...] Funktionalität unabhängig von der konkreten Handhabung, Präsentation und Technik bereitgestellt wird.“ [Sau03, S.42] Das MCF hingegen basiert darauf, dass eine Präsentation unabhängig von der Technik bereitgestellt werden kann. Es setzt demnach auf einer

höheren Ebene an, als das *Multi-Channeling* des WAM-Ansatzes. Die Mechanismen, die das *Multi-Channeling* des WAM-Ansatzes bietet, werden jedoch auch im MCF verwendet.

2.3 Probleme des Multichannel-Frameworks

Bezogen auf die Anforderungen werden AA3 und AA5 nicht durch das MCF umgesetzt. (Zu AA5 ist dabei zu erwähnen, dass jede Sprache eine gewisse Ausdruckskraft hat. Da in dieser Arbeit versucht wird, die GUI-Entwicklung mittels einer ausdrucksstarken *domänenspezifischen Sprache* (DSL), die sich auf die *Domäne* von *profil c/s* bezieht, umzusetzen, ist die Ausdruckskraft herkömmlicher Programmiersprachen oder Frameworks als unzureichend anzusehen).

Ein weiteres Problem bezieht sich auf die integrierten Frameworks (*Swing* und *wingS*). Beide Frameworks sind veraltet und werden nicht mehr gewartet. (Analysen dazu wurden in [Gun13] und [Gun14b] durchgeführt.) Um auch in Zukunft den Anforderungen der Kunden nachkommen zu können, müssten beide Frameworks von den Entwicklern der *data experts GmbH* selbst weiterentwickelt werden. Eine andere Möglichkeit wäre es, andere und modernere Frameworks einzusetzen, 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 1). In der Praxis wurde diese unkomplizierte 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 *GridBagLayout* stark beeinflusst werden. Ein solcher Layoutmanager steht nicht in allen GUI-Frameworks zur Verfügung. Da die Beschreibung der GUI über ein solches Layout vollzogen wird, ist der Umgang mit dem *GridBagLayout* innerhalb des Frameworks eine Voraussetzung für die Integration in das MCF. Zusammenfassend sind folgende Probleme des MCF zu nennen:

P1 Verwendete Frameworks sind veraltet

P2 Starke Orientierung an *Swing*

Dazu müssen in der *data experts GmbH* bei der Entwicklung von *GUIs* lästige Routinearbeiten durchgeführt werden, die im Folgenden genannt werden:

- R1** Vergebene Bezeichnungen für *GUI-Komponenten* müssen in unterschiedlichen Klassen (*IP*- und *GUI-Klassen*) gepflegt werden.
- R2** Beim Erstellen von Tabellen müssen viele Methoden überschrieben werden.
- R3** Die Werte für Aufschriften, Größeneinstellungen u. Ä. für *GUI-Komponenten* werden bei der *data experts GmbH* in Properties-Dateien festgehalten. Das Erstellen und Pflegen wird als fehleranfällig betrachtet.

2.4 Zielsetzung

Das langfristige Ziel der *data experts GmbH* bzgl. der *GUIs* ist es, eine Lösung zu entwickeln, welche das *MCF* ablösen kann. Anzustreben ist eine Lösung, die neben der Umsetzung der oben genannten Anforderungen, auch die genannten Routinearbeiten reduziert.

In dieser Arbeit wird ein Ansatz untersucht, bei dem es möglich ist, AA1 - AA5 besser umzusetzen. Kern des Ansatzes ist eine *DSL*, mit deren Hilfe die *GUIs* beschrieben werden sollen (im Folgenden als *GUI-DSL* bezeichnet). Eine *DSL* könnte so konzipiert werden, dass sie ausreichend abstrakt und erweiterbar ist und bzgl. der Ausdruckskraft das *MCF* übertrifft.

Die genaue Lösungsidee durch die *GUI-DSL*, welche in dieser Arbeit verfolgt wird, ist in Kapitel 5 beschrieben.

Kapitel 3

Domänenspezifische Sprachen

3.1 Begriffsbestimmungen

Sprache/Programmiersprache

Formal betrachtet ist eine Sprache eine beliebige Teilmenge aller Wörter über einem Alphabet. „Ein Alphabet ist eine endliche, nicht leere Menge von Zeichen (auch Symbole oder Buchstaben genannt).“ [Hed12, S.6] Zur Verdeutlichung der Definition einer Sprache sei V ein Alphabet und $k \in \mathbb{N}$ (\mathbb{N} ist die Menge der natürlichen Zahlen einschließlich der Null) (vgl. [Hed12, S.6]). „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].

Programmiersprachen werden dazu verwendet, um einem Computer Instruktionen zukommen zu lassen (vgl. [FP11, S.27], [VBK⁺13, S.27]). In diesem Kontext werden die Bestandteile einer Sprache wie folgt abgegrenzt:

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 (vgl. [Aho08, S.87]).

Abstrakte Syntax

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

Statische Semantik

Die statische Semantik beschreibt die Menge an Regeln bzgl. des Typsystems, die ein Programm befolgen muss (vgl. [VBK⁺13, S.26]).

Ausführungssemantik

Die Ausführungssemantik ist abhängig vom Compiler. Sie beschreibt, wie ein Programm zum Zeitpunkt seiner Ausführung funktioniert (vgl. [VBK⁺13, S.26]).

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-Maschine* berechenbar ist. Völter et al. behaupten, dass alle GPLs aufgrund dessen untereinander austauschbar sind. Dennoch sind Abstufungen bzgl. der Ausführung dieser Programmiersprachen zu machen. Unterschiedliche GPLs sind für spezielle Aufgaben optimiert (vgl. [VBK⁺13, S.27f]).

Domain Specific Language (DSL)

Eine *DSL* (zu dt. domänenspezifische Sprache) ist eine Programmiersprache, welche für eine bestimmte *Domäne* optimiert ist (vgl. [VBK⁺13, S.28]). Das Entwickeln einer *DSL* ermöglicht es, die Abstraktion der Sprache der Domäne anzupassen (vgl. [Gho11, S.10]). Das bedeutet, dass Aspekte, welche für die Domäne unwichtig sind, auch in der Sprache außer Acht gelassen werden können. Semantik und Syntax sollten demnach der jeweiligen Abstraktionsebene angepasst sein. Eine *DSL* ist demzufolge in ihren Ausdrucksmöglichkeiten eingeschränkt. Je stärker diese Einschränkung ist, desto besser sind die Unterstützung der Domäne sowie die Ausdruckskraft der *DSL* (vgl. [FP11, S.27f]).

Darüber hinaus sollte ein Programm, welches in einer *DSL* geschrieben wurde, alle Qualitätsanforderungen erfüllen, die auch bei einer Umsetzung des Programms mit GPLs realisiert werden (vgl. [Gho11, S.10f]).

Es gibt verschiedene Arten von DSLs. Neben internen und externen *DSLs*

(siehe Kapitel 3.6 und 3.7) findet eine Unterscheidung zwischen technischen *DSLs* und fachlichen *DSL* statt. Markus Völter et al. unterscheiden diese beiden Kategorien im Allgemeinen dahingehend, dass technische *DSLs* von Programmierern genutzt werden und fachliche *DSL* von Domänenexperten (z. B. Kunden) (vgl. [VBK⁺13, S.26]).

Grammatik

Grammatiken und insbesondere Grammatikregeln werden zur Beschreibung von Sprachen verwendet (vgl. [Hed12, S.23f]). Für die Definition einer Grammatik wird aus den Praxisbericht [Gun14a] verwiesen. Grammatiken können in einer Hierarchie dargestellt werden (*Chomsky-Hierarchie*) (vgl. [Hed12, S.32f]). Bei Programmiersprachen handelt es sich um *kontextfreie Sprachen*. Diese sind entscheidbar und können somit von einem Compiler verarbeitet werden (vgl. [Hed12, S.16f]).

Lexikalische Analyse

Bevor ein DSL-Skript (Text, der in der Syntax einer *DSL* geschrieben wurde) verarbeitet werden kann, muss das Skript vom sogenannten *Lexer* oder *Scanner* gelesen werden (vgl. [FP11, S.221]). Dabei wird ein Text aus diesem Skript als Input-Stream betrachtet. Der Lexer wandelt diesen Input-Stream in einzelne Tokens um (vgl. [Gho11, S.220]). Im Allgemeinen ist der *Lexer* die Instanz innerhalb der *DSL-Umgebung*, die für das Auslesen des DSL-Skriptes verantwortlich ist.

Parser

Ein Parser ist ebenfalls ein Teil der *DSL-Umgebung* (vgl. [Gho11, S.211]). Er ist dafür verantwortlich, aus dem Ergebnis der lexikalischen Analyse ein Output zu generieren, mit dem weitere Aktionen durchgeführt werden können (vgl. [Gho11, S.212]). Der Output wird in Form eines Syntaxbaums (AST) generiert (vgl. [FP11, S.47]). Ein solcher Baum ist laut Martin Fowler et al. eine weitaus nutzbarere Darstellung dessen, was mit dem DSL-Skript

beschrieben werden soll. Daraus lässt sich auch das semantische Modell generieren (vgl. [FP11, S.48]).

Semantisches Modell

Das semantische Modell ist ebenfalls eine Repräsentation dessen, was mit der *DSL* beschrieben wurde. Es wird laut Martin Fowler et al. auch als die Bibliothek betrachtet, welche von der *DSL* nach außen hin sichtbar ist (vgl. [FP11, S.159]). In Anlehnung an Ghosh sowie Martin Fowler et al. wird das semantische Modell als Datenstruktur betrachtet, deren Aufbau von der Syntax der *DSL* unabhängig ist (vgl. [Gho11, S.214], [FP11, S.48]).

Generator

Laut Martin Fowler et al. ist ein Generator der Teil der *DSL-Umgebung*, welcher für das Erzeugen eines Quellcodes für die *Zielumgebung* zuständig ist (vgl. [FP11, S.121]). Bei der Generierung von Quellcodes wird zwischen zwei Verfahren unterschieden.

1. Transformer Generation

Bei der *Transformer Generation* wird das semantische Modell als Input verwendet. Aus diesem Input wird der Quellcode für die Zielumgebung generiert (vgl. [FP11, S.533f]). Ein solches Verfahren wird oft verwendet, wenn ein Großteil des Outputs generiert wird und die Inhalte des semantischen Modells einfach in den Quellcode der Zielumgebung überführt werden können (vgl. [FP11, S.535]).

2. Templated Generation

Bei der *Templated Generation* wird eine Vorlage benötigt. In dieser Vorlage befinden sich Platzhalter. Diese dienen dazu, dass der vom Generator erzeugte Quellcode an diesen Stellen eingesetzt werden kann (vgl. [FP11, S.539f]). Dieses Codegenerierungsverfahren wird oft verwendet, wenn sich im zu generierenden Quellcode für die Zielumgebung viele statische Inhalte befinden und der Anteil dynamischer Inhalte sehr einfach gehalten ist (vgl. [FP11, S.541]).

3.2 Anwendungsbeispiele

Die Anwendungsbereiche für *DSLs* sind breit gefächert. Die bekanntesten *DSLs* 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) (vgl. [Gho11, S.12]). Alle Sprachen sind in ihren Ausdrucksmöglichkeiten eingeschränkt und von der Abstraktion her direkt auf eine Domäne (jeweils dahinter in Klammern genannt) zugeschnitten (vgl. [Gho11, S.12f]).

Weitere Beispiele für *DSLs* befinden sich im Bereich der Sprachen für Parser-Generatoren (*YACC*, *ANTLR*) oder im Bereich der Sprachen für das Zusammenbauen von Softwaresystemen (*Ant*, *Make*) (vgl. [Gho11, S.12]).

3.3 Model-Driven Software Development (MDSD)

In der Einleitung wurde schon der Model-Driven Ansatz in Verbindung mit GUI-Entwicklung erwähnt. Dieser Ansatz versucht den technischen Lösungen der IT-Industrie einen gewissen Grad an Agilität zu verleihen (vgl. [SKNH05]). Dies hängt damit zusammen, dass die Entwicklung von Softwareprodukten schneller und besser vonstatten geht und mit weniger Kosten realisierbar ist (vgl. [DM14, S.71]).

Erreicht wird dies, indem die Modelle formaler, strenger, vollständiger und konsistenter beschrieben werden (vgl. [VBK⁺13, S.31]). Die Grundidee ist, dass die Modelle Quellcodes oder Funktionalitäten beschreiben und diese in der Evolution der Software immer wiederverwendet werden können (vgl. [DM14, S.72]). Somit wird ein redundanter oder schematischer Quellcode vermieden und es ist möglich diese Modelle auch in anderen Anwendungen zu verwenden (vgl. [DM14, S.72]).

Daraus lassen sich folgende Ziele des MDSD ableiten:

- Schnelleres Entwickeln durch Automatisierungen
- Bessere Softwarequalität durch automatisierte Transformationen und formalen Modell-Definitionen
- Verhinderung von Wiederholungen und besseres Management von

veränderbaren Technologien durch die Trennung der Funktionsbereiche (Separation of Concerns)

- Architekturen, Modellierungssprachen (bspw. eine DSL) und Generatoren/Transformatoren können besser wiederverwendet werden
- Verringerte Komplexität durch höhere Abstraktion

(vgl. [SV06, S.13f])

Die Modelle sind somit nicht länger nur zur Dokumentation geeignet, sondern sind ein Teil der Software (vgl. [SV06, S.14f]). Dabei sind sie auf ein bestimmtes Domänenproblem angepasst. Die Beschreibung dieser Modelle kann bspw. über eine *DSL* erfolgen (vgl. [SV06, S.15]). In Abbildung 3.3 ist die Idee des MDSD schematisch dargestellt.

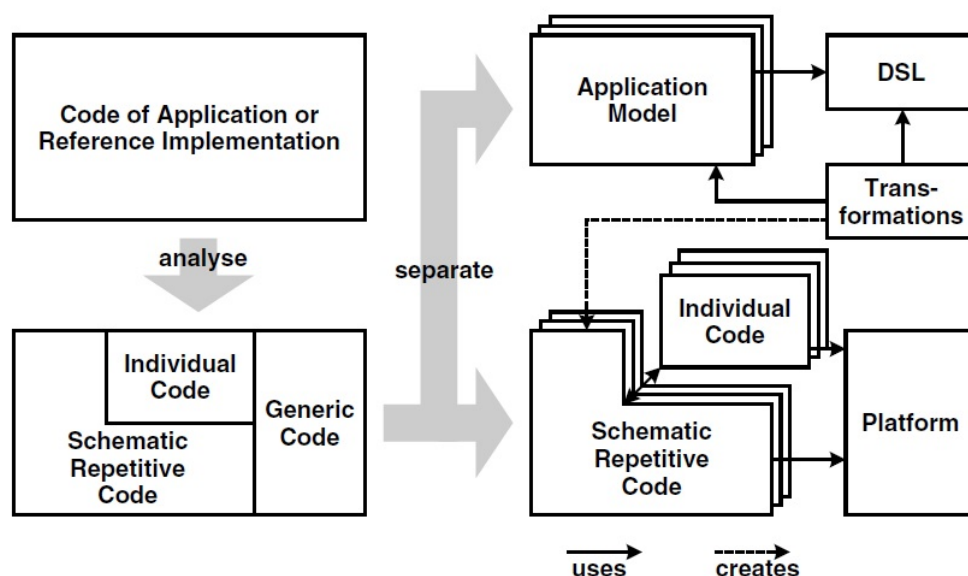


Abbildung 3: Die grundlegenden Ideen hinter dem MDSD (vgl. [SV06, S.15])

3.4 Abgrenzung zu GPL

Wie vorhergehend bereits erwähnt, sind GPLs Sprachen mit denen alles berechnet werden kann, was auch mit einer *Turing-Maschine* berechenbar ist. Folglich kann mit einer GPL jedes berechenbare Problem gelöst werden. Eine *DSL* hat diese Eigenschaft nicht. Da sie auf eine bestimmte Domäne zu-

geschnitten ist, können auch nur Probleme innerhalb dieser Domäne mit ihr gelöst werden (vgl. [VBK⁺13, S.28]). Martin Fowler et al. bezeichnen diese Eigenschaft des Domänenfokus als ein Schlüsselement der Definition einer *DSL* (vgl. [FP11, S.27f]).

3.5 Vor- und Nachteile einer DSL gegenüber einer GPL

Vorteile:

Ausdruckskraft

Laut Ghosh sollten *DSLs* so umgesetzt werden, dass sie präzise sind. Diese Präzision bedingt, dass eine *DSL* einfach zu verstehen ist. Bei der Verwendung einer *DSL* sollte demnach der von Dan Roam beschriebenen Prozess des visuellen Denkens (sehen - betrachten - verstehen - zeigen) (vgl. [Roa09]) so schnell wie möglich abzuarbeiten sein.

Weiterhin ist es wichtig bei der Entwicklung einer *DSL* darauf zu achten, dass sich die Abstraktion der Sprache an der Semantik der Domäne orientiert (vgl. [Gho11, S.20]). Sind diese Empfehlungen umgesetzt, wächst das Verständnis für das, was entwickelt werden soll, da die semantische Lücke zwischen Programm und Problem kleiner wird (vgl. [Gho11, S.20], [BCK08]). Zudem bleibt die Komplexität durch eine höhere Abstraktionsebene beherrschbar (vgl. [BCK08]).

Höhere Qualität

Bei der Entwicklung einer *DSL* werden die Sprachkonstrukte und Freiheitsgrade der Sprache festgelegt. Richtig konzipiert, schränken sie den Entwickler im Umgang mit dieser *DSL* so ein, dass die Möglichkeit redundanten Quellcode zu schreiben oder mehrfache Arbeit durchzuführen, kaum noch besteht. Zusätzlich wird die Anzahl von Syntaxfehlern verringert (vgl. [VBK⁺13, S.40f]). Ferner wird durch die starke Abstraktion einer *DSL* die Wiederverwendung gefördert, was ebenfalls zu einem qualitativ höherwertigem Quellcode führt (vgl. [Gho11,

S.21])).

Verbesserte Produktivität bei der Entwicklung der Software

Durch die Ausdruckskraft und die Abstraktion einer *DSL* muss i. d. R. weniger Quellcode für die Implementierung eines Programms geschrieben werden, als bei der Verwendung einer GPL benötigt wird. Mit einem entsprechenden Framework für GPLs könnte ähnliches erreicht werden (vgl. [VBK⁺13, S.40]).

Ebenso führt die stärkere Ausdruckskraft zu einer besseren Lesbarkeit von DSL-Code im Vergleich zu GPL-Code, wodurch jener einfacher zu verstehen ist. Dies erleichtern das Finden von Fehlern in diesem Quellcode sowie Veränderungen an dem System vorzunehmen. Bei einer GPL werden diese Vorteile durch Dokumentationen, ausdrucksvolle Variablenbezeichnungen und festgelegten Konventionen angestrebt (vgl. [FP11, S.33]). Allerdings ist der Entwickler zur Einhaltung dieser Vorschriften nicht gezwungen. Bei der Verwendung einer *DSL* hingegen kann dem Entwickler dieser Freiheitsgrad entzogen werden. Damit ist er gezwungen, lesbaren Quellcode zu schreiben, da die Sprache es nicht anders zulässt.

Bessere Kommunikation mit Domänenexperten und Kunden

Aufgrund domänenspezifischer und präziser Ausdrücke, die in der Sprache verwendet werden, sind die Domänenexperten bzw. die Kunden vertrauter mit der Implementierung, als würde für die Umsetzung eine GPL verwendet werden (vgl. [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 (vgl. [FP11, S.34], [VBK⁺13, S.42]). Somit kann die oft vernachlässigte Kommunikation zwischen Kunden und Auftragnehmern verbessert werden. Martin Fowler et al. bezeichnen die Verwendung einer *DSL* als reine Kommunikationplattform als vorteilhaft (vgl. [FP11, S.34f]). Grund dafür ist, dass bereits bei der Entwicklung einer *DSL* das Verständnis des Auftragnehmers über die Domäne gesteigert wird (vgl. [VBK⁺13, S.41]).

Plattformunabhängigkeit

Durch die Nutzung einer *DSL* kann bspw. 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 wird. Ein solches Verfahren wird oft unter der Verwendung von *XML* genutzt (vgl. [FP11, S.35]). Dadurch ist es möglich die Logik auf unterschiedlichen Plattformen auszuführen (vgl. [VBK⁺13, S.43]). Dieser Vorteil ist besonders für den praktischen Teil dieser Arbeit interessant, allerdings weniger in Bezug auf Logik, denn in Bezug auf Benutzerschnittstellen.

Einfachere Validierung und Verifizierung Da *DSLs* bestimmte Details der Implementierung ausblenden, sind sie auf semantischer Ebene reichhaltiger als *GPLs*. Das führt dazu, dass Analysen einfacher umzusetzen sind und Fehlermeldungen verständlicher gestaltet werden können, indem die Terminologie der Domäne verwendet wird. Dadurch und durch die vereinfachte Kommunikation mit den Domänenexperten, werden Reviews und Validierungen des *DSL*-Codes weitaus effizienter (vgl. [VBK⁺13, S.41]).

Unabhängigkeit von Technologien

Die Modelle, welche 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 weitestgehend entkoppelt werden (vgl. [VBK⁺13, S.41]).

Skalierung des Entwicklungsprozesses

Die Integration von neuen Mitarbeitern in ein Entwicklerteam fordert immer eine gewisse Einarbeitungszeit. Dieser Zeitraum kann durch die Nut-

zung einer *DSL* verkürzt werden, wenn die *DSL* einen hohen Abstraktionsgrad hat und dadurch leichter zu verstehen und zu erlernen ist (vgl. [Gho11, S.21]).

Innerhalb eines Entwicklerteams haben die Mitarbeiter oft einen unterschiedlichen Erfahrungsstand bzgl. einer speziellen Programmiersprache, die zur Entwicklung genutzt werden soll. Erfahrene Teammitglieder könnten 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 (vgl. [Gho11, S.21]). Das führt im ersten Moment zu einer effizienteren Arbeitsweise, da sich nicht jeder Entwickler mit allem auskennen muss. Markus Völter et al. hingegen sehen die Teilung der Programmieraufgaben als Gefahr bzw. Nachteil (vgl. [VBK⁺13, S.44]).

Nachteile:

Großes Know-How gefordert

Bevor die eine *DSL* genutzt werden kann, muss sie entwickelt werden (vgl. [VBK⁺13, S.43]). Das Designen einer Sprache ist eine komplexe Aufgabe, die nur schwer skalierbar ist (vgl. [Gho11, S.21]). Die Vorteile, die eine *DSL* bietet, können nur genutzt werden, wenn die *DSL* ausreichend gut konzipiert ist. Dazu muss einerseits der richtige Abstraktionsgrad gefunden und andererseits die Sprache so einfach wie möglich gehalten werden. Für beide Aufgaben werden Entwickler benötigt, die viel Erfahrung mit Sprachdesign haben (vgl. [VBK⁺13, S.44]).

Kosten für die Entwicklung der DSL

Bei wirtschaftlichen Entscheidungen wird der monetäre Input mit dem monetären Output verglichen. Investitionen führen dazu, dass der Input größer wird. Da eine *DSL* vor dem Einsatz zuerst entwickelt werden muss, ist es notwendig Investitionen für die Entwickler der *DSL* zu tätigen. Ob sich eine Investition lohnt, muss vorher durch entsprechende Analysen überprüft werden. Dabei muss festgestellt werden, ob die Entwicklung der *DSL* gerechtfertigt ist. Im Bereich der technischen *DSLs* fällt die Rechtfertigung

einfach, da diese *DSLs* oft wiederverwendet werden können. Fachliche *DSLs* 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 (vgl. [VBK⁺13, S.43]).

Weiterhin sind in der Phase, in der die *DSL* entwickelt wird, keine erhebliche Senkung der Kosten zu erwarten. Die Kosten reduzieren sich i. d. R. erst, wenn die *DSL* eingesetzt wird (vgl. [Gho11, S.21]).

Bevor eine *DSL* entwickelt werden kann, sollte ein entsprechendes Know-How aufgebaut werden. Der Aufbau dieses Wissens verursacht weitere Kosten (vgl. [FP11, S.37]).

Investitionsgefängnis

Der Begriff stammt von Markus Völter et al. Er beruht auf der Annahme, dass sich ein Unternehmen dessen bewusst ist, dass höhere Investitionen in wiederverwendbare Artefakte zu einer besseren Produktivität führen. Artefakte, die wiederverwendet werden können, führen dennoch zu Einschränkungen. Die Flexibilität geht dabei verloren. Weiterhin besteht dabei die Gefahr, dass bestimmte Artefakte aufgrund geänderter Anforderungen unbrauchbar werden. Darüber hinaus ist es gefährlich, Artefakte zu verändern die häufig wiederverwendet werden, weil dadurch unerwünschte Nebeneffekte auftreten können. Somit wäre das Unternehmen wiederum zu Investitionen gezwungen, um die Anforderungen umzusetzen. Von daher der verwendete Begriff *Investitionsgefängnis* (vgl. [VBK⁺13, S.45]).

Kakophonie

Eine *DSL* abstrahiert von einem Domänenmodell (vgl. [Gho11, S.22]). Je besser diese Abstraktion ist, desto euphonischer und ausdrucksstärker ist die Sprache.

Normalerweise werden für eine Applikation mehrere *DSLs* benötigt. Diese unterschiedlichen *DSLs* haben i. d. R. unterschiedliche syntaktische Strukturen. Das führt dazu, dass Mitarbeiter unterschiedliche Sprachen beherrschen müssen. Das wiederum erfordert, dass die Entwickler öfter umden-

ken müssen, als würden sie fortwährend mit einer Sprache arbeiten. Dies macht den Entwicklungsprozess weitaus komplizierter (vgl. [FP11, S.37]).

Ghetto-Sprache

Wenn ein Unternehmen nur mit eigenen *DSLs* arbeitet, gleichen diese Sprachen einer Ghetto-Sprache, die von keinem anderen Unternehmen verstanden wird. Dadurch ist es schwer, neue Technologien von anderen Unternehmen in den Bereichen, in denen vermehrt *DSLs* eingesetzt werden, zu integrieren. Denn diese Technologien werden nicht mit den eigenen *DSLs* kompatibel sein. Außerdem ist es schwer in diesem Bereich von neuen Mitarbeitern zu profitieren, da anzunehmen ist, dass sie diese *DSLs* und ihren Zweck nicht kennen (vgl. [FP11, S.38]).

Dieser Punkt ist auch in Verbindung mit dem *Investitionsgefängnis* 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. Eine andere Möglichkeit ist, weiter in die Entwicklung eigener *DSLs* zu investieren, um seine Systeme aufrecht zu erhalten.

Borniertheit durch Abstraktion

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 dieser Abstraktionsebene passt.

Dabei besteht die Gefahr, dass der Entwickler sich von der Abstraktion der Sprache gefangen nehmen lässt. Das bedeutet, dass er versucht, das Problem aus der realen Welt auf seine Abstraktion anzupassen. Der richtige Weg hingegen ist, die Sprache und deren Abstraktionsebene so anzupassen, dass das Problem beschrieben werden kann (vgl. [FP11, S.39]).

Kulturelle Herausforderungen

Die genannten Nachteile des Einsatzes von DSLs, führen zu Äußerungen wie der, dass die Entwicklung von Sprachen kompliziert ist, Domänenexperten keine Programmieren sind oder auch, dass nicht schon wieder eine neue Sprache gelernt werden will (*Yet-Another-Language-To-Learn Syndrom* (vgl. [Gho11, S.22])).

Solche kulturellen Probleme entstehen i. d. R. dann, wenn etwas Neues eingeführt werden soll (vgl. [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 die Einführung entsprechender Tools vereinfacht wurde, besteht die Gefahr der voreiligen Entwicklung einer neuen *DSLs*. Das heißt, dass die Notwendigkeit einer neuen *DSL* nicht ausreichend evaluiert wurde. Durch die einfachere Entwicklung scheint es weniger aufwendig eine neue *DSL* zu konstruieren, als nach bestehenden Ansätzen zur Lösung für das gleiche Problem zu suchen (vgl. [VBK⁺13, S.44f]). Die Aussicht auf die Amortisierung einer Investition in neue *DSLs*, unterstützt diese Vorgehensweise (vgl. [FP11, S.38]). Somit entstehen immer mehr *DSLs*, die auf gleichen Problemen basieren, aber untereinander nicht kompatibel sind. Außerdem fördert die Entwicklung einer *DSL* das Verstehen der Domäne, weshalb die Möglichkeit besteht, dass dies der einzige Grund für eben diese Entwicklung ist (vgl. [FP11, S.38]). Das wiederum hat zur Folge, dass mehrere unvollständige *DSLs* existieren. Markus Völter et al. nennen dieses Phänomen die „*DSL Hell*“ (vgl. [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 Gewinn schließlich höher ausfallen (vgl. [Gho11, S.21]).

3.6 Interne DSLs

Bei einer internen *DSL* handelt es sich um eine *DSL*, die in eine *GPL* integriert ist. Sie übernimmt dabei das Typsystem der *GPL* (vgl. [VBK⁺13, S.50]). In Bezug auf die Ziele aus Kapitel 3.3 können einige dieser Absichten mit Application Programming Interfaces (*API*) erreicht werden. In vielen Fällen ist eine *DSL* nicht mehr als ein *API*. Martin Fowler et al. sehen den größten Unterschied zwischen *API* und *DSL* darin, dass eine *DSL* neben einem abstrahierten Vokabular auch eine spezifische Grammatik nutzt (vgl. [FP11, S.29]). Ein *API* hingegen besitzt die gleichen syntaktischen Strukturen wie die *GPL*, in der das *API* bereitgestellt wurde. Somit werden überflüssige Notationsformen in das *API* übernommen, was bei einer *DSL* nicht der Fall ist (vgl. [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*“ (vgl. [VBK⁺13, S.30]).

3.6.1 Implementierungstechniken

Parse Tree Manipulation

Allgemein betrachtet funktioniert diese Technik wie folgt:

Ein Codefragment, welches erst gelesen und zu einem späteren Zeitpunkt ausgewertet werden soll, 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 Codefragment in einen Parse Tree umzuformen und diesen zu bearbeiten. Diese Möglichkeit existiert nur in wenigen Sprachen. Martin Fowler et al. nennen hierzu nur die Beispiele C#, ParseTree (Ruby) und Lisp (vgl. [FP11, S.45f]).

Anders als Lisp bieten die anderen Beispiele die Möglichkeit über den Parse Tree zu iterieren. Bei Lisp-Codes handelt es sich bereits 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 Parse Trees einbezogen werden (vgl. [FP11, S.46]).

Konstrukte, die in der Wirtsprache geschrieben wurden und nicht verändert werden sollen, spielen bei der *Parse Tree Manipulation* für die Erzeugung der semantischen Modelle keine Rolle (vgl. [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 (vgl. [FP11, S.28]). In einer internen *DSL* hingegen ist es möglich, Methoden bereitzustellen, die verkettet werden können und somit komplette Sätze darstellen. Dadurch wird der Output einer Methode zum Input der folgenden Methode. Demzufolge wird die Lesbarkeit der *DSL* wesentlich verbessert, 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 internen *DSL* (vgl. [VBK⁺13, S.50]). Ein einfaches Beispiel für ein Fluent Interface wird von Martin Fowler et al. beschrieben.

Listing 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 Teile der Informationen über ein Programmelement, wie Methoden oder Variablen. Diese Informationen können während der Laufzeit oder der Übersetzungszeit - wenn die Umgebung die Möglichkeit dazu

bietet - manipuliert werden (vgl. [FP11, S.445]).

Bevor eine Annotation verarbeitet werden kann, muss sie definiert werden. Die Definitionen von Annotationen variieren bei Verwendung unterschiedlicher Sprachen (vgl. [FP11, S.446]). Die Verarbeitung von Annotationen kann zu drei bestimmten Zeitpunkten, nämlich zum Zeitpunkt der Übersetzung, des Ladens oder der Ausführung des Programms, stattfinden (vgl. [FP11, S.447]). Die Interpretation und Ausführung von Annotationen während der Laufzeit beeinflussen i. d. R. das Verhalten von Objekten. Beim Laden des Programms werden meist Validierungsannotationen verwendet. Solche Annotationen finden bspw. beim Auslesen des Mappings für Datenbanken Anwendung. Somit wird die Definition der Elemente von der Verarbeitung getrennt, was zu einem übersichtlichen und lesbaren Quellcode beiträgt (vgl. [FP11, S.449]).

3.7 Externe DSLs

Eine externe *DSL* ist eine separate Sprache, welche die Infrastruktur vorhandener Sprachen nicht nutzt (vgl. [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 *Parsen* bezeichnet (vgl. [FP11, S.28]). Für den Parser und die lexikalische Analyse werden oft vorhandene Infrastrukturen genutzt (vgl. [Gho11, S.19]).

3.7.1 Implementierungstechniken

Bei den Implementierungstechniken von externen *DSLs* geht es um die Art und Weise, wie der DSL-Code vom Parser in ein semantisches Modell oder einen *AST* überführt wird (vgl. [FP11, S.89]). Die allgemeine Vorgehensweise bei der Verwendung von Parsern ist der Abbildung 4 zu entnehmen.

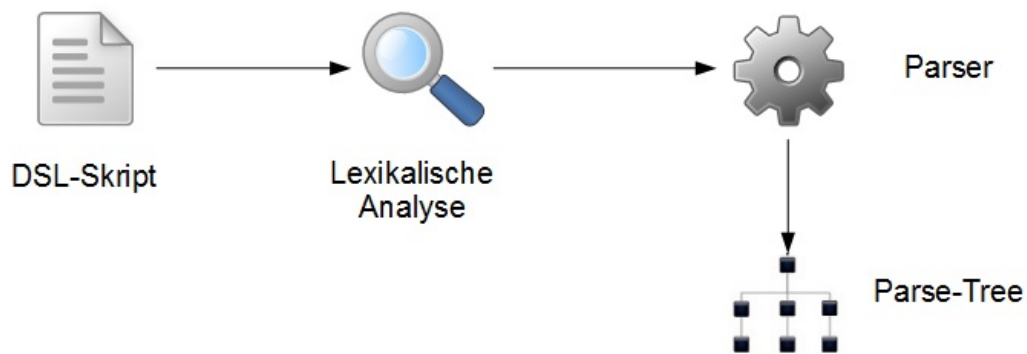


Abbildung 4: Parsen allgemein

Parsergenerator

Bei der Generierung von Parsern muss dieser nicht manuell implementiert werden. Diese Aufgabe wird an den Generator delegiert. Damit dies möglich ist, müssen zwei Artefakte definiert werden. Zuerst muss eine Grammatik in der erweiterten Backus-Naur Form (EBNF) beschrieben werden. Anschließend werden bestimmte Aktionen benötigt, die bei der Bestätigung bestimmter Grammatikregeln ausgeführt werden sollen (Validierungsregeln) (vgl. [Gho11, S.218]). Wird der Parsergenerator ausgetauscht, führt dies auch häufig dazu, dass die notwendigen Artefakte (Grammatik und Aktionen) neu definiert werden müssen (vgl. [FP11, S.269]). Weiterhin arbeiten die meisten Parsergeneratoren mit Code-Generierung, wodurch der Erstellungsprozess komplexer wird (vgl. [FP11, S.272]). Vorteile dieser Technik im Vergleich zur manuellen Implementierung des Parsers, sind die Folgenden:

- Möglichkeit des Programmierens auf einem höheren Abstraktionsniveau (vgl. [Gho11, S.218])
- Gebrauch von weniger Quellcode zur Implementierung des Parsers (vgl. [Gho11, S.218])
- Möglichkeit des Generierens eines Parsers in unterschiedlichen Sprachen (vgl. [Gho11, S.218], [FP11, S.270])
- Validierung der Grammatik durch Fehlererkennung und -behandlung (vgl. [FP11, S.272])

Recursive Decent Parser (RD-Parser)

Dieser Parser basiert auf Funktionen, die rekursiv aufgerufen werden. Es handelt sich dabei um einen *Top-Down Parser* (vgl. [Gho11, S.226]). Die Funktionen implementieren dabei die Regeln des Parsens für die nonterminalen Symbole der Grammatik (vgl. [FP11, S.245]). Die Funktionen geben dabei einen Boolean-Wert zurück, der Auskunft darüber gibt, ob die Symbole aus dem DSL-Skript mit den laut Grammatik zu erwartenden Symbolen übereinstimmen (vgl. [FP11, S.246]). Tabelle 2 enthält die Implementierungsmöglichkeiten von einfachen Grammatikregeln.

Grammatikregel	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 2: Implementierung einfacher Grammatikregeln mit einem RD-Parser (vgl. [FP11, S.248])

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

Parser-Kombinator

Bei der Kombination von Parsern wird die Grammatik mittels einer Struktur von Parser-Objekten implementiert (vgl. [FP11, S.256]). Wenn ein Teil des Input-Streams von einem Parser erfolgreich oder auch fehlerhaft verarbeitet wurde, kann der Rest des Input-Streams an einen anderen Parser übergeben werden. Somit ist es möglich, Parser-Objekte beliebig zu verketteten (vgl. [Gho11, S.242]). Die Elemente, die verkettet werden können, werden *Parser-Kombinatoren* genannt. Abbildung 5 stellt schematisch diese Funktionsweise dar.

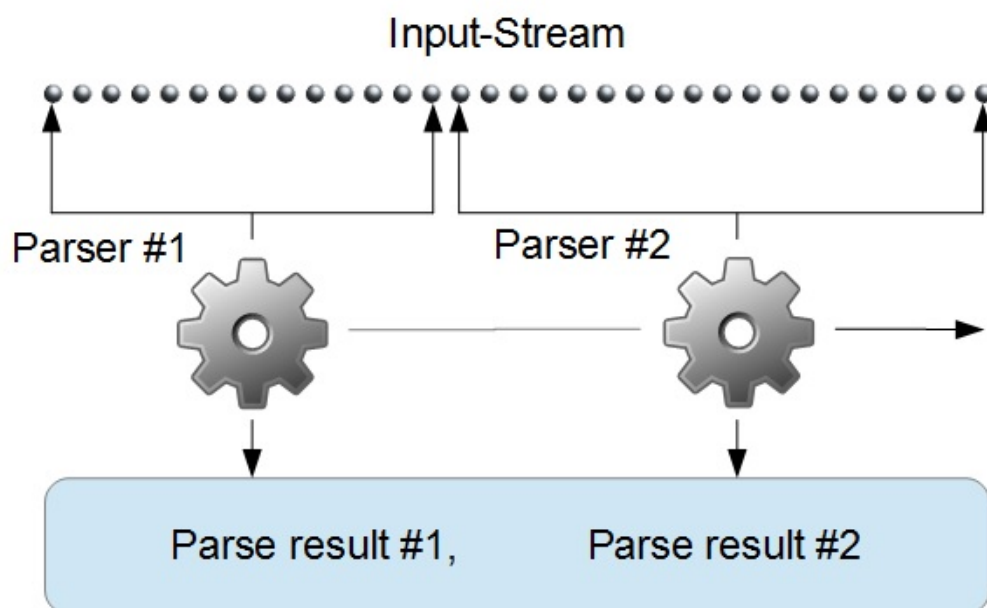


Abbildung 5: Funktionsweise von Parser-Kombinatoren (in Anlehnung an [Gho11, S.243])

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 Modell repräsentiert (vgl. [FP11, S.256]). Ein großer Vorteil dieser Technik ist, dass einfache Parser zu komplexeren Parsern zusammengefügt werden können. Weiterhin wird durch die Kombination mehrerer grammatikbestimmender Komponenten auch die Lesbarkeit der Grammatik gefördert, was bei einem RD-Parser ein großer Nachteil ist. Daher bezeichnen Martin Fowler et al. Parser-Kombinatoren auch als Mittelweg zwischen RD-Parsern und Parser-Generatoren

(vgl. [FP11, S.261]).

3.8 Nicht-textuelle DSLs

Die Kapitel 3.6 und 3.7 bezogen sich auf textuelle DSLs. Auch wenn eine *DSL* eine bestimmte Domäne repräsentiert, bedeutet dies nicht, dass diese Repräsentation immer textuell erfolgen muss (vgl. [Gho11, S.19]). Es gibt einige Gründe, mit einer nicht-textuellen *DSL* zu arbeiten:

- Viele Domänenprobleme können von den Domänennutzern besser durch Tabellen oder grafische Darstellungen erklärt werden
- Domänenlogik ist in textueller Form oft zu komplex und enthält zu viele syntaktische Strukturen
- Visuelle Modelle sind von Domänenexperten einfacher zu durchdringen und zu verändern

(vgl. [Gho11, S.19])

Für diesen Ansatz muss der Domänennutzer die Repräsentation des Wissens über eine Domäne innerhalb eines auf Projektionen basierenden Editors visualisieren. Mit diesem Editor kann der Domänennutzer die Sicht auf die Domäne verändern, ohne Quellcode schreiben zu müssen. Im Hintergrund generiert dieser Editor den Quellcode, welcher die Sicht auf die Domäne modelliert (vgl. [Gho11, S.19f]).

Kapitel 4

GUI-DSL

4.1 Motivation des Ansatzes

Eine GUI ermöglicht die Interaktion mit einem Programm durch unterschiedliche GUI-Komponenten (vgl. [Gal07, S.4]). Mit Hilfe dieser Komponenten werden Informationen dargestellt oder Eingaben vom Nutzer getätigt. Um die Zusammensetzung der GUI über eine DSL zu beschreiben, gibt unterschiedliche Ansätze, wovon zwei im Folgenden vorgestellt werden:

1. Dieser Ansatz zeichnet sich dadurch aus, dass die GUI auf der Grundlage von fachlichen Modellen generiert wird. (vgl. [SKNH05]) Das bedeutet, dass in der Beschreibung der *GUI-DSL* keine GUI-Komponenten verwendet werden, wie es bei traditionellen GUI-Frameworks (JavaFX, Swing) der Fall ist.
2. Der zweite Ansatz beschreibt ein Modell der GUI, welches in andere GUI-Modelle überführt werden kann. Somit wird die Trennung zwischen der GUI und den fachlichen Aspekten in der Software erhalten und das Domänenproblem auf die GUI reduziert.

Die Umsetzung des letztgenannten Ansatzes ist weitaus einfacher, da lediglich die Aspekte, welche die GUI betreffen, auf *MDSD* umgestellt werden müssen. Zudem soll den Entwicklern weiterhin die Möglichkeit gegeben werden die GUIs selbst zu entwerfen, was durch die im ersten Ansatz genannte Beschreibungsform nicht möglich ist, da die *GUIs* vollständig generiert werden. Wogegen unter der Verwendung des zweiten Ansatzes wei-

terhin GUI-Komponenten verwendet werden, was den Entwicklern die Anpassung der GUIs ermöglicht. Von daher wird dieser Ansatz weiter verfolgt und eine *GUI-DSL* konzipiert, welche die nachfolgenden Anforderungen erfüllen soll.

4.2 Anforderungen an die GUI-DSL

Die allgemeinen Anforderungen an die GUI und an die Sprache zur Beschreibung dieser, wurden in Kapitel 2.1 erläutert. Die folgenden Festlegungen beziehen sich auf die Ausdrucksmöglichkeit der *GUI-DSL*, wodurch eine GUI beschrieben werden soll.

AS1 Beschreibung von GUIs über die Zusammensetzung von GUI-Komponenten

AS2 Wiederverwendung und Erweiterung bzw. Veränderung beschriebener GUIs

AS3 Verwendung einer abstrakten Layoutbeschreibung

AS4 Gebrauch von weniger Quellcode zur Beschreibung von GUIs

AS5 Beschreibung von Interaktionen mit den GUI-Komponenten

AS6 Erweiterung um neue GUI-Komponenten

Wie im vorherigen Abschnitt bereits erwähnt, sollen die Entwickler in der Lage sein, eine GUI relativ frei zu gestalten. Daraus folgt, dass die einzelnen GUI-Komponenten unterschiedlich kombinierbar sein müssen (siehe Anforderung AS1). Zudem sollten die beschriebenen GUIs auch in anderen GUI-DSL-Skripten (GUI-Skripten) wiederverwendet werden können (siehe Anforderung AS2)), da viele GUIs in *profil c/s* ähnlich aufgebaut sind.

In Bezug auf das Layout muss erwähnt werden, dass in der traditionellen GUI-Entwicklung die Strukturierung der GUI-Komponenten mit Hilfe von Layoutcontainern vorgenommen wird. In der Vergangenheit hat sich gezeigt, dass die Strukturierung über ein spezifisches Layout zu einer Orientierung an einem bestimmten Framework führt (Beispiel: MCF orientiert sich an Swing). Dies ist unvorteilhaft, da bestimmte Layouts auf anderen Plattformen (Bspw. Web oder Mobil) nicht dargestellt werden können, weil

entsprechende Layoutmanager nicht vorhanden sind. Somit ist das Layout in der *GUI-DSL* so zu beschreiben, dass es auf allen Plattformen gleichermaßen gut dargestellt werden kann (siehe Anforderung AS3).

Für eine Steigerung der Effizienz in der *data experts GmbH* ist bei der Einführung neuer Technologien außerdem darauf zu achten, dass weniger Quellcode geschrieben werden muss als zuvor. Die Qualität darf darunter jedoch nicht leiden (siehe Anforderung AS4).

Da eine GUI ohne Interaktionsmöglichkeiten ihren Zweck nicht erfüllen kann, ist die Beschreibung der Interaktionen unabdingbar (siehe Anforderung AS5).

Darüber hinaus darf die Erweiterung um neue GUI-Komponenten nicht vernachlässigt werden (siehe Anforderung AS6), da anderenfalls die Gefahr besteht, dass die *GUI-DSL* unbrauchbar wird.

Kapitel 5

Entwicklung einer Lösungsidee

5.1 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, die im vorherigen Kapitel angesprochene GUI-DSL zur Beschreibung von GUIs zu nutzen. Diese GUIs sollen so beschrieben werden, dass sie in der Domäne von profil c/s für unterschiedliche GUI-Frameworks genutzt werden können. Diese Beschreibung soll weiterhin nur einmal stattfinden. Der Quellcode, welcher die GUI im entsprechenden Framework darstellt, wird frameworkspezifisch aus der GUI-Beschreibung generiert. Langfristig betrachtet könnte das MCF damit abgelöst werden.

Die Anforderungen für die GUI-DSL wurden bereits beschrieben. Diese sollen so weit wie möglich im Prototypen, welcher im Zuge dieser Arbeit entwickelt wird, umgesetzt werden. Der Prototyp soll Quellcode erzeugen, der eine GUI mit den syntaktischen Strukturen des MCF beschreibt. Somit kann geprüft werden, ob sich der generierte Quellcode in profil c/s einbinden lässt.

5.2 Konzept

Die *GUI-DSL* wird für die abstrakte Beschreibung der GUI verwendet. Somit ist gewährleistet, dass die GUI weiterhin nur einmal beschrieben werden muss. Wie in Abschnitt 5.1 beschrieben, wird der Quellcode zur Dar-

stellung der GUI frameworkspezifisch, mit Hilfe eines speziellen Generators, erzeugt. Daraus folgt, dass die Integration neuer Frameworks (siehe Anforderung AA2) an die Implementierung eines spezifischen Generators gekoppelt ist. Abbildung 6 zeigt das grundlegende Konzept für diesen Ansatz auf. Dabei wurden exemplarisch drei unterschiedliche Generatoren für bestimmte Frameworks verwendet.

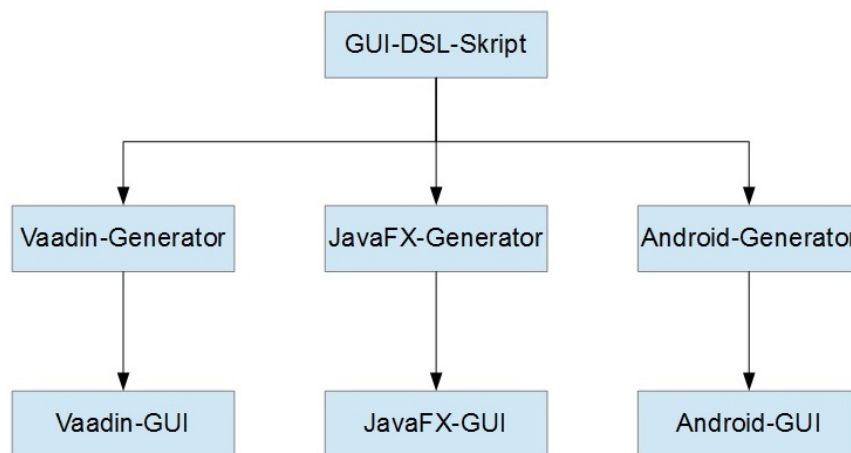


Abbildung 6: Grundlegendes Konzept

Der Prototyp wird aus einem GUI-Skript Quellcode erzeugen, welcher in das MCF eingebunden werden kann. Somit lässt sich prüfen, ob die GUI-DSL für existierende GUIs von profil c/s genügt. In Anlehnung an Abbildung 3 in Kapitel 3.3 wird eine GUI mit sinnvollen Interaktionen nicht allein über die *GUI-DSL* nicht umsetzbar sein, da die dafür notwendigen Informationen nicht in der *GUI-DSL* beschrieben werden. Folglich ist zumindest zur Erzeugung dieser Informationsquellen individueller Quellcode von Nöten. Abbildung 7 zeigt diese grundlegende Idee schematisch auf.

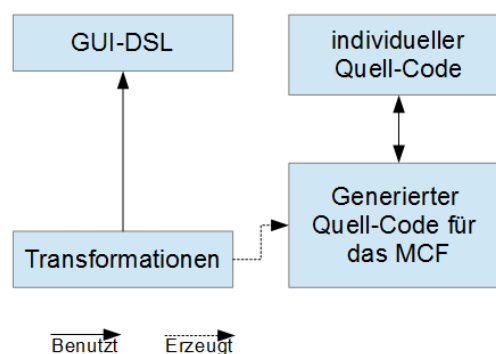


Abbildung 7: Grundlegende Idee für den Prototypen

Kapitel 6

Evaluation des Frameworks zur Entwicklung der DSL

6.1 Vorstellung ausgewählter Frameworks

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

6.1.1 PetitParser

Dieses Framework 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 (vgl. [RDGN10]). Es existieren auch Versionen des Frameworks für Java¹, Dart² und PHP³.

Einfache Parser bestehen aus Sequenzen von Funktionen, welche die Produktionsregeln (Produktionen) der Grammatik abbilden. Komplexe Parser werden durch die Kombination anderer Parser implementiert (vgl. [RDGN10]).

Die Implementierung dieser Kombination kann in einer einzelnen Methode

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

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

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

vorgenommen werden, wodurch der Parser einem Skript ähnelt. Alternativ können die zu kombinierenden Parser auch in Methoden von Unterklassen des `PetitParsers` implementiert werden (vgl. [BCK10, S.6]). Das fördert die Lesbarkeit, Übersichtlichkeit und schließlich die Wartbarkeit des Codes. Tool Support ist für dieses Framework gewährleistet. Mithilfe dessen können Produktionen editiert und grafisch abgebildet werden. Weiterhin können Zufallsbeispiele für ausgewählte Produktionen generiert werden, um somit Fehler in der Grammatik aufzudecken. Darüber hinaus wird die Effizienz einer Grammatik durch die Darstellung und Behebung 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 dem 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 von *Xtext* verwendet werden. Aus der Grammatik wird der Parser sowie ein Modell, mit dessen Hilfe weitere Artefakte der DSL-Umgebung implementiert werden können, generiert. Die Klassen für Validierungs-Regeln und den Generator werden ebenfalls vom Tool erzeugt. Diese müssen im Anschluss daran vom Nutzer entsprechend erweitert werden (vgl. [The14]). Zum Editieren der entsprechenden Dateien wird eine eigene Syntax verwendet, die meiner Meinung stark an die Java-Syntax erinnert.

Wenn der Parser generiert wurde, ist es möglich einen in Eclipse integrierenden Editor zu erzeugen (vgl. [VC09, 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 Meta Programming System

Das Meta Programming System (MPS) ist ebenfalls eine Open-Source-Lösung, bei der die Entwicklung externer DSLs im Vordergrund stehen. Bei der Entwicklung der Sprache mit MPS ist weder eine Grammatiken noch ein Parser

involviert. Die Sprache wird mit diesem Tool projektional entworfen. Das bedeutet, dass die Sprache nicht nur in Text-Form definiert werden kann, sondern auch mittels Symbolen, Tabellen oder Grafiken (vgl. [VBK⁺13, S.16]).

Zur Unterstützung der Entwicklung wird von der Firma JetBrains ein projektionaler Editor zur Verfügung gestellt⁴.

Der Generator kann die Konstrukte der neuen Sprache in bestimmte Basis-Sprachen überführen. Diese Basis-Sprachen sind *C*, *Java* oder *XML*. Auch die Transformation in einfachen Text ist gewährleistet (vgl. [Völ11]).

Des Weiteren wird ein Editor für die Arbeit mit der DSL zur Verfügung gestellt. Dieser bietet mehrere Funktionalitäten wie Code-Completion, Refactoring-Möglichkeiten oder einen Debugger (vgl. [PSV13]).

Die Integration in Eclipse war laut Pech et al. Ende 2013 geplant (vgl. [PSV13]). Im Oktober 2014 wies Vaclav Pech jedoch im Jet-Brains-Forum darauf hin, dass die Integration von MPS in Eclipse aufgrund von wichtigeren Features zurückgestellt wurde (vgl. [Pec14]). Ein Plugin für Eclipse ist demnach in absehbarer Zeit nicht zu erwarten.

6.2 Vergleich und Bewertung der vorgestellten Frameworks

In diesem Abschnitt wird das Framework für die Umsetzung des Prototypen evaluiert. Dabei sind folgende Kriterien von Belang.

Machbarkeit der Integration in Eclipse

Dieser Punkt ist wichtig, da die deg hauptsächlich mit Eclipse arbeitet und so wenig wie möglich von anderen Tools Gebrauch machen möchte. Grund dafür ist, dass die gewohnte Arbeitsweise der Entwickler bzgl. des Tools nicht beeinträchtigt werden soll.

Erweiterung der Grammatik

Die Grammatik muss erweiterbar sein, weil die GUI in profil c/s kein abgeschlossenes Konzept ist. Es ist davon auszugehen, dass neue Kom-

⁴<https://www.jetbrains.com/mps/>

ponenten in Zukunft benötigt werden

Bereitstellung eines Editors für DSL-Skripte

Ein Editor soll die effiziente Entwicklung unterstützen. Features wie Code-Completion oder ausdrucksvolle Fehlermeldungen sind der deg daher wichtig.

Erweiterung der Validierungen

Um ausdrucksvolle Fehlermeldungen verwenden zu können, ist es notwendig Validierungen durchzuführen, die entsprechende Fehler aufdecken können. Die Standard-Validierungen prüfen i.d.R. nur die Syntax der Sprache und keine fachlichen Zusammenhänge.

Vorhandenes Know-How

Um eine DSL effizient zu entwickeln ist neben dem Sprachdesign auch der Umgang mit dem Framework wichtig. Von daher werden die Erfahrungen der deg mit den vorgestellten Frameworks ebenfalls mit einbezogen.

Die Bewertung ist Tabelle 3 zu entnehmen. Dabei wurden drei Bewertungsstufen (Gut (+), Ausreichend (O) und Ungenügend (-)) verwendet.

Kriterium	PetitParser	Xtext	MPS
Machbarkeit der Intergration in Eclipse	+	+	-
Erweiterung der Grammatik	+	+	+
Bereitstellung eines Editors für DSL-Skripte	+	+	O
Erweiterung der Validierungen	O	+	+
Vorhandenes Know-How	-	O	-

Tabelle 3: Bewertung der Frameworks für die Entwicklung von DSLs

Die Machbarkeit der Integration von MPS in Eclipse ist derzeit noch nicht gewährleistet. Da für Xtext ein entsprechendes Plugin und für PetitParser eine Java-Version existiert, ist auch die Integration dieser beiden Frameworks in Eclipse möglich.

Bei der Möglichkeit zur Erweiterung der Grammatik müssen nirgends Abstriche gemacht werden.

Die Bereitstellung eines Editors für DSL-Skripte ist bei der Verwendung von

MPS schlechter ausgefallen, als bei den anderen Frameworks. Grund dafür ist, dass der Editor nicht in Eclipse verwendet werden kann.

Die Validierungen bzgl. der DSL-Skripte können bei PetitParser durch die Parser-Kombinationen umgesetzt werden. Ausdrucksvolle Fehlermeldungen können jedoch nicht bereitgestellt werden. Die anderen Frameworks bieten dafür weitaus bessere Möglichkeiten.

Der letzte und entscheidende Punkt ist das vorhandene Know-How. Die deg hat bzgl. PetitParser und MPS keine Erfahrungen. Die Erfahrungen mit Xtext halten sich zwar in Grenzen, übersteigen aber dennoch die Affinität mit den anderen Frameworks.

Nach dieser Analyse ist Xtext vor allem aufgrund des vorhandenen Know-Hows auszuwählen.

Kapitel 7

Festlegungen für die Entwicklung des Prototypen

7.1 Vorgehensmodell

Das Vorgehensmodell für die Entwickler des DSL-Prototypen ist ein inkrementelles Modell. Das bedeutet, dass mehrere Iterationen durchlaufen werden (inkrementell), in denen unterschiedliche Versionen des Prototyps entwickelt werden (vgl. [Sau10, S.5]). In Abbildung 8 ist das Vorgehensmodell schematisch dargestellt.

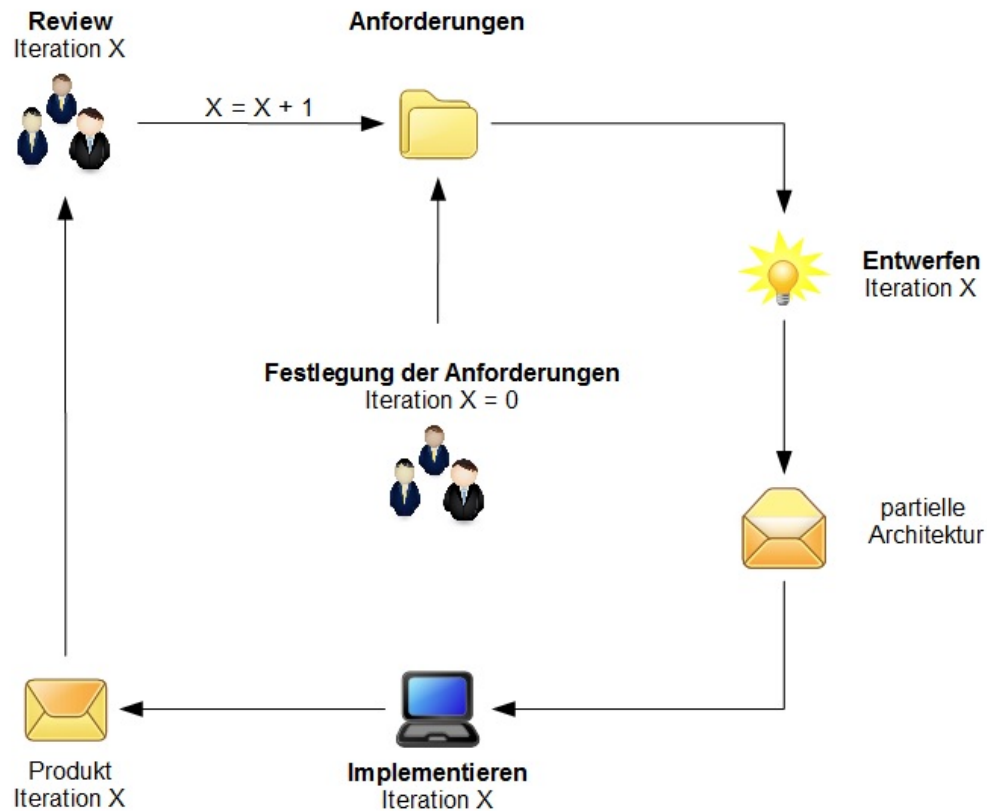


Abbildung 8: Inkrementelles Modell

Nach der Definition der Anforderungen wird der Prototyp für die aktuelle Iteration entworfen und entwickelt. Im folgenden Verlauf werden diese beiden Phasen nicht separiert. An die Implementierung des Prototypen der aktuellen Iteration schließt sich ein Review an. Innerhalb des Reviews wird der Prototyp der aktuellen Iteration vorgestellt und weitere Anforderungen festgelegt, bestehende Anforderungen geändert oder Änderungen am gesamten Konzept gemacht. Das führt wiederum zu einem neuen Entwurf, woran sich eine weitere Implementierung anschließt. Dieser Zyklus wird somit mehrmals durchlaufen (Iteration) (vgl. [Sau10, S.5]).

Grund für dieses Vorgehen ist, dass in der deg bzgl. DSL-Entwicklung wenig Know-How existiert. Aus den Iterationen soll so viel Erfahrung und Wissen wie möglich geschöpft werden. Außerdem werden somit auch Irrwege aufgezeigt, die bei Entwicklung anderer DSLs Beachtung finden können. Weiterhin können Anforderungen flexibel angepasst und Missverständnisse reduziert werden, da der Entwicklungsprozess transparent ist. (vgl. [Sau10, S.67])

Der weitere Verlauf (Kapitel 7.2, 8 und 9) wird die durchgeführten Iterationen beschrieben. Dabei werden folgende Aspekte beleuchtet.

Vision (siehe Kapitel 7.2)

Dadurch werden die Anforderungen an die DSL-Umgebung beschrieben. Die in Kapitel 4 vorgestellten Anforderungen sind das Resultat der in dieser Arbeit durchgeführten Iterationen.

Entwicklung

Die Entwicklung beschreibt die Phasen *Entwerfen* und *Implementieren*. Sie teilt sich in zwei weitere Bereiche auf.

- Entwicklung der DSL (siehe Kapitel 8)
- Entwicklung eines Generators (siehe Kapitel 9)

7.2 Grobkonzept der DSL-Umgebung (Vision)

7.2.1 1. Iteration

Die DSL-Umgebung soll sich in zwei Bereiche unterteilen.

- DSL zur Beschreibung der GUI
- Generator zur Generierung von Quellcode

In Abbildung 9 sind die Artefakte mit den Funktionen (blauer Kasten), die von ihnen umgesetzt werden sollen dargestellt.

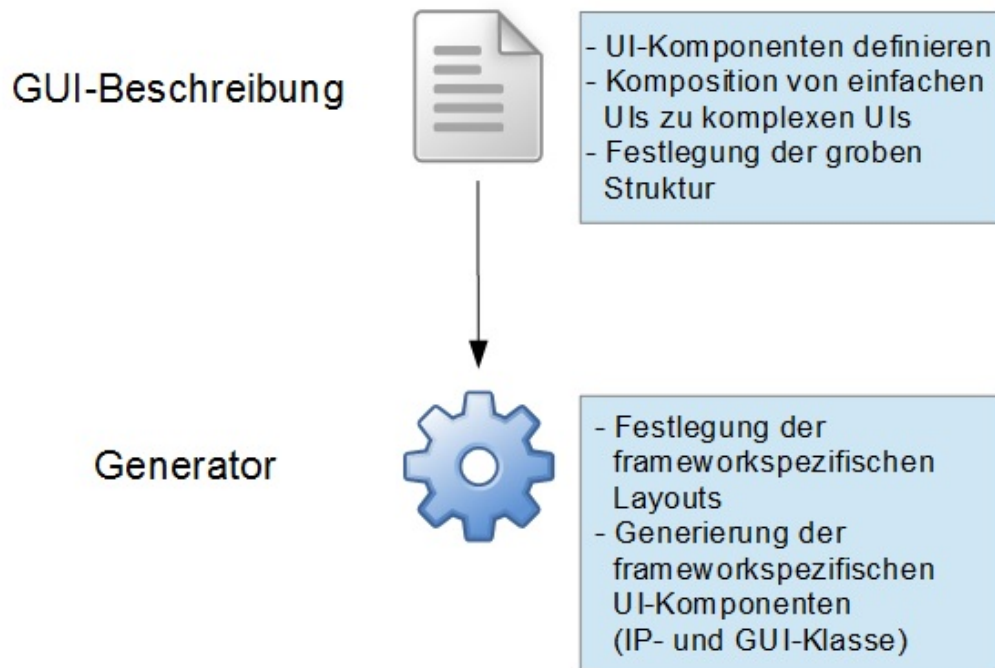


Abbildung 9: Konzeption der DSL-Umgebung (1. Iteration)

Die elementare Aufgabe der GUI-DSL ist es, die *GUI-Komponenten*, die in dem GUI verwendet werden sollen, zu definieren (siehe Anforderung AS1). Ausgehend von den für profil c/s verwendeten *GUI-Komponenten*, können diese in drei Kategorien unterteilt werden.

Basiskomponenten

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

Komplexe Komponenten

Diese zeichnen sich dadurch aus, dass sie domänenspezifisch sind und speziell für profil c/s entwickelt wurden und Interaktionen und Funktionalitäten bereits festgelegt sind und nicht verändert werden können. Ein Beispiel für eine komplexe Komponenten ist die *Multiselection-Komponente*.

Layout-Komponenten

Dabei handelt es sich um Komponenten, welche die Struktur der GUI

bestimmen. In anderen GUI-Frameworks sind dies bspw. *Panel* (Swing), *Div* (HTML) oder *Pane* (JavaFX).

Bei den Beschreibungen der *Basiskomponenten* muss auch die Möglichkeit bestehen Interaktionen festzulegen (siehe Anforderung AS5). Zu einer Interaktion gehört die Art der Interaktion und bestimmte Aktionen, die bei der Interaktion ausgeführt werden. Die anderen *GUI-Komponenten* haben festgelegte Interaktionsmöglichkeiten (*komplexe Komponenten*) oder es ist nicht möglich mit ihnen zu interagieren (Layout-Komponenten).

Bezüglich der *komplexen Komponenten* ist darauf zu achten, dass sie für jedes verwendete GUI-Framework implementiert werden müssen. Damit wird verhindert, dass die Entwickler, die mit der *GUI-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. Die Notwendigkeit dessen, dass die Quellen für diese *komplexen Komponenten* sowohl zur Entwicklungszeit, als auch zur Laufzeit vorhanden sein müssen, ist ein Nachteil dieses Konzeptes.

Bei den Layout-Komponenten ist besonders auf die Ausdruckskraft der für die Beschreibung dieser Komponenten verwendeten Bezeichnungen zu achten. Grund dafür ist, dass die GUI auf unterschiedlichen Plattformen abgebildet werden soll, die spezielle Layout-Bereiche unterstützen. Beispielsweise lässt sich ein Programmfenster auf dem Desktop als oberste Layout Komponente festlegen. In einem Web-Browser ist der Begriff *Fenster* als oberste Layout Komponenten in der *data experts GmbH* nicht geläufig. Die Komponente, die in einem Browser mit dem Programmfenster auf dem Desktop assoziiert wird, ist in der *data experts GmbH* das Tab.

Darüber hinaus sollen die Skripte für die Beschreibung der *GUIs* so konzipiert werden, dass es möglich ist andere GUI-Beschreibungen dort einzubinden (siehe Anforderung AS1). Somit werden eingebundene GUIs wiederum zu *GUI* -Komponenten. Ziel dessen ist es, dass die Entwickler aus mehreren einfachen *GUIs* ein komplexes *GUI* erstellen können (Modularisierung). Die Gefahr die dabei besteht ist, dass mit der Zeit viel einfache GUI-Beschreibungen mit ähnlicher Struktur entwickelt werden. Da die

deg bei den GUIs von profil c/s ein bestimmtes Schema verfolgt (Cooperate Design), sollte dieses Problem dadurch ausreichend eingedämmt sein. Wichtig ist hierbei zu betonen, dass die DSL keine Sprache sein soll, mit der jedes GUI beschrieben werden kann. Sie soll lediglich UIs für profil c/s beschreiben können. Der Vorteil, der sich aus dieser starken Modularisierung ergibt, ist dass viele GUI-Beschreibungen wiederverwendet werden können. So ist es meiner Meinung nach möglich komplexe GUI-Beschreibungen zu entwickeln, die in Fachabteilungen mit fachlichen Konzepten assoziiert werden können.

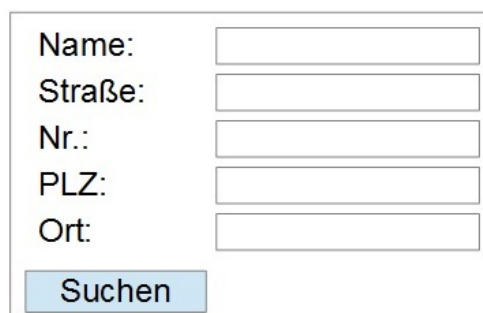
Beispielsweise können Suchmasken nach diesem Konzept gestaltet, beliebig wiederverwendet und kombiniert werden. Hierzu werden unterschiedliche Suchfelder definiert, die aus einem Label und einem Textfeld bestehen (Beispiel siehe Abbildung 10).



The image shows a simple search field for a name. It consists of a rectangular container with a thin border. Inside, on the left, is the text 'Name:' followed by a small gap, and then a rectangular text input field.

Abbildung 10: Beispiel: Suchfeld für Name

In eine Suchmaske können mehrere dieser Suchfelder beliebig komponiert werden. (Beispiel siehe Abbildung 11).



The image shows a search mask form. It is a rectangular container with a thin border. Inside, there are five rows, each with a label on the left and a text input field on the right. The labels are 'Name:', 'Straße:', 'Nr.:', 'PLZ:', and 'Ort:'. Below these input fields is a blue button with the text 'Suchen' in white.

Abbildung 11: Beispiel: Suchmaske

Durch diese Möglichkeit der Komposition können auch komplexere fachliche Konzepte auf die GUI bezogen werden, wie z.B. *Personensuche*.

Um die miteinander komponierten *GUI-Komponenten* zu strukturieren ist es

notwendig, dass die *GUI-DSL* Informationen über die Anordnung der *GUI-Komponenten* enthält. Diese Informationen müssen ausreichend abstrakt sein, damit sich diese Struktur auf unterschiedliche GUI-Frameworks beziehen lässt. Dazu wird die Struktur innerhalb einer GUI als Anordnung von Bereichen betrachtet. In der *GUI-DSL* werden diesen Bereichen die *GUI-Komponenten* zugeordnet. Genauere Informationen über die Anordnung dieser Bereiche dürfen nicht enthalten sein, da dies eine Orientierung an bestimmte Layouts bedingt (siehe Anforderung AS3). Dazu ist weiterhin wichtig, dass den Bereichen jeweils nur eine Komponente zugeordnet werden kann. Dadurch wird der Zwang zur vorher beschriebenen Komposition von eingebundenen *GUI-DSL* -Skripten und definierten *GUI-Komponenten* verstärkt werden. Der Generator übernimmt beim Erzeugen des frameworkspezifischen Quellcodes die konkrete Anordnung der beschriebenen Bereiche. Grund dafür ist, dass die Anordnung der Komponenten frameworkspezifisch ist und teilweise unterschiedliche Layoutmanager in unterschiedlichen Frameworks unterstützt werden. Da für jedes eingesetzte Framework ein eigener Generator implementiert werden muss (siehe Kapitel 5), ist es theoretisch möglich diese Aufgabe weitgehend unabhängig von der Beschreibung der verwendeten *GUI* -Komponenten zu erfüllen.

7.2.2 2. Iteration

Das grundsätzliche Konzept muss um ein Artefakt erweitert werden (siehe Abbildung 12).

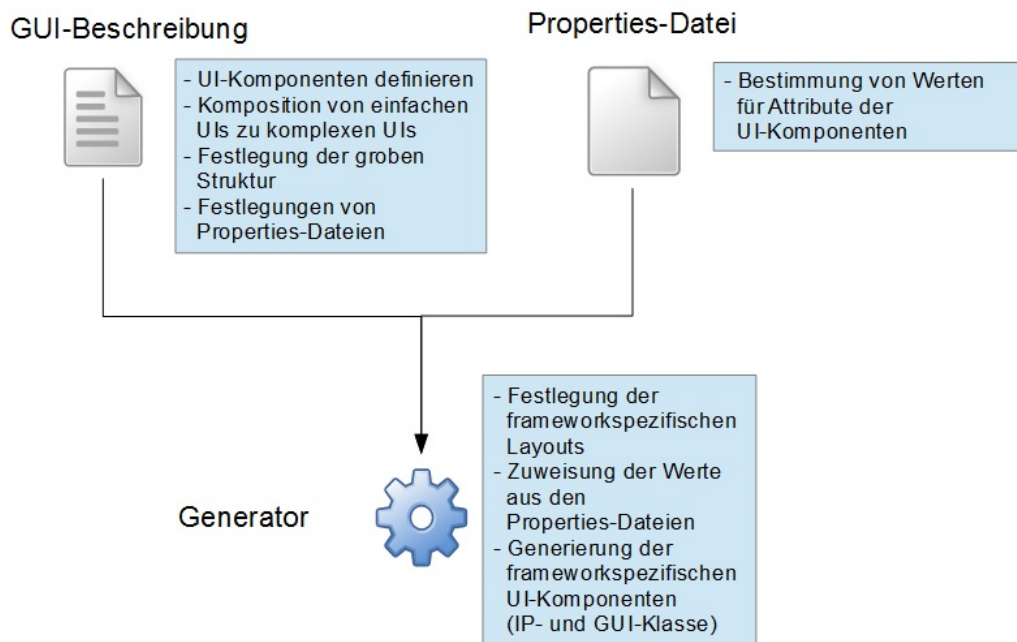


Abbildung 12: Konzeption der DSL-Umgebung (2. Iteration)

Eine Änderung, die in dieser Grafik nicht dargestellt wird, ist, dass die Aktionen, die bei Interaktionen mit *GUI-Komponenten* ausgeführt werden, nicht in der GUI-DSL definiert werden sollen. Grund dafür ist, dass diese Aktionen sehr unterschiedlich sind und somit kaum abstrahiert werden können. Eine weitere Änderung ist, dass die *GUI-Komponenten*, die einem GUI-DSL-Skript direkt definiert werden, in einer anderen Beschreibung, wo jenes GUI-DSL-Skript eingebunden ist, verändert werden können. Dadurch werden die wiederverwendeten Beschreibungen anpassbar, was die Flexibilität enorm steigert (siehe Anforderung AS2).

Darüber hinaus soll die Möglichkeit bestehen, bestimmte Werte, welche die Attribute von *GUI-Komponenten* annehmen können, in Properties-Dateien auszulagern, wodurch die GUI-DSL weitgehend entlastet wird. Allerdings muss für die Zuweisung von *GUI-Komponenten* zu Wert-Beschreibung in der Properties-Datei und dem DSL-Skript ein eindeutiger Schlüssel definiert werden.

Der Generator muss die festgelegten Properties-Dateien in die Generierung mit einbeziehen und ihnen die entsprechenden Werte entnehmen. Dabei gilt die Festlegung, dass wenn in dem GUI-DSL-Skript einem Attribut ein bestimmter Wert zugewiesen ist, wird in der Properties-Datei (wenn die-

se nebst Schlüssel festgelegt wurde) nicht mehr nach diesem Attribut der Komponente gesucht.

7.2.3 3. Iteration

Das grundsätzliche Konzept wird nochmals erweitert. Neben den bekannten Artefakten der DSL-Umgebung kommt ein weiteres Artefakt hinzu. Dabei handelt es sich um eine Layout-Beschreibung. Somit findet die Beschreibung des Layout nicht mehr im Generator statt, sondern muss nur noch frameworkspezifisch generiert werden. In Abbildung 13 ist das neue Konzept schematisch dargestellt.

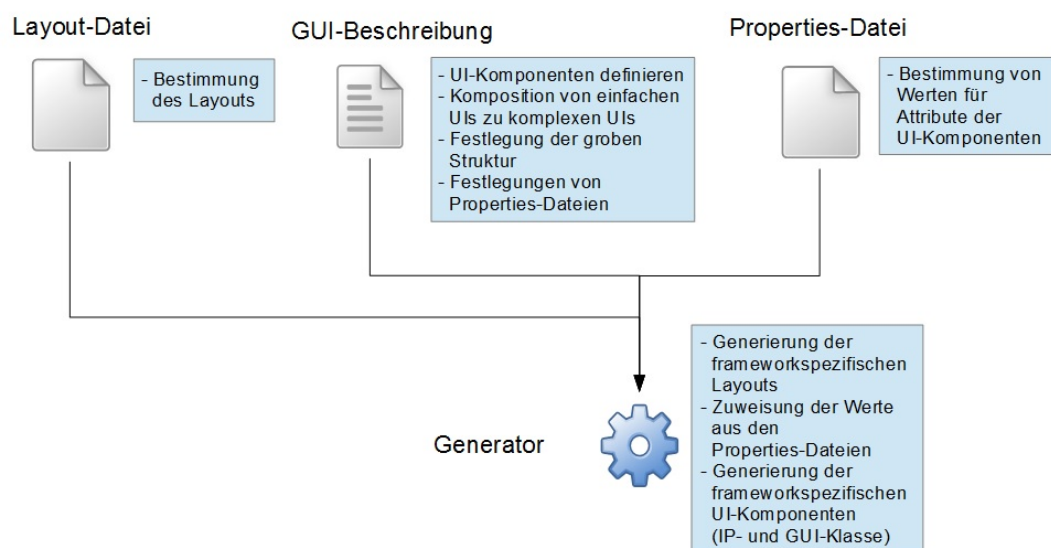


Abbildung 13: Konzeption der DSL-Umgebung (3. Iteration)

Da die Layout-Beschreibung in eine separate Datei ausgegliedert wurde, muss die Referenzierung von *GUI-Komponenten* aus dem GUI-Skript mit den Festlegungen in der Layout-Datei referenziert werden können.

Bezogen auf die im GUI-Skript definierten Bereiche ist aufgefallen, dass die Angabe der Anzahl der Bereiche nicht notwendig ist. Grund dafür ist, dass dieser Wert doppelt definiert werden würde. Einerseits direkt in der Angabe der Anzahl und andererseits indirekt durch die Zuweisung der *GUI-Komponenten* und eingebundenen GUI-Skripte zu den Bereichen. Ursprünglich war die Idee, dass auf Basis dieser doppelten Angabe Validierungen ausgeführt werden können. Die doppelte Deklaration widerspricht jedoch

der Anforderung, dass so wenig Codezeilen wie möglich geschrieben werden sollen, um eine GUI zu beschreiben (siehe Anforderung AS4).

Weiterhin ist aufgefallen, dass in einem GUI-Skript nur eine Properties Datei angegeben werden kann. In den GUI-Klassen der deg können jedoch mehrere Properties-Dateien angegeben werden. Dies ist auch in der GUI-DSL zu beachten.

Neben den *Basiskomponenten* *Button* und *Label* müssen folgende weitere *Basiskomponenten* definiert werden können.

Textfield

Ein Feld in dem ein einzeliger Text editiert werden kann.

Textarea

Ein Feld in dem ein mehrzeiliger Text editiert werden kann.

Tree

Eine Baumstruktur in der mehrere Element eingebunden werden können.

Table

Eine Tabellenstruktur in der mehrere Elemente eingebunden werden können.

TabView

Eine Ansicht in der aus mehreren Taben eine Tab betrachtet werden kann.

Interchangeable

Ein Bereich, in dem während der Laufzeit unterschiedliche GUIs eingebunden werden können.

Weiterhin müssen die Bezeichnungen der Interaktionstypen den Bezeichnungen der Interaktionsformen (IF) der deg angeglichen werden. Darüber hinaus sollen folgende Standard-Interaktionstypen in den *GUI-Komponenten*, die für den Prototypen benötigt werden, verwendet werden. (Zuerst wird die Komponente genannt und anschließend die Standard-Interaktionstypen)

- Label: IfTextDisplay

- Tree: IfTree, IfActivator

Diese Standard-Interaktionstypen müssen nicht in den jeweiligen Komponenten definiert werden.

Kapitel 8

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

8.1 1. Iteration

Analyse der Metadaten

Die Beschreibung einer GUI wird in der *GUI-DSL* als eigener Komplex betrachtet (siehe semantisches Modell *UIDescription*). Innerhalb dieses Komplexes werden die entsprechenden Komponenten definiert. Die Bereiche die innerhalb einer Beschreibung festgelegt werden sollen, müssen *GUI-Komponenten* zugeordnet werden können (siehe semantisches Modell *AreaAssignment*). Diese Bereiche sollten vor der Entwicklung bereits festgelegt werden. Um abzusichern, dass die Anzahl der festgelegten Bereiche genau eingehalten wird, muss diese Anzahl in der GUI-Beschreibung angegeben werden (siehe semantisches Modell *AreaCount*).

Für die Beschreibung der Layout-Komponenten werden zwei Typen unterschieden (siehe semantisches Modell *TypeDefinition*), um zwischen obersten Layout-Komponenten und anderen zu differenzieren.

Ein weiterer Aspekt in dem GUI-Skript ist die Verwendung von anderen GUI-Skripten (siehe semantisches Modell *Use*).

Zusammenfassend sind für die Beschreibung der GUI folgende Metadaten nötig.

- Anzahl der Bereiche

- Zuweisung der *GUI-Komponenten* zu den Bereichen
- Angabe des Layout-Typs
- Angabe der Verwendeten GUI-Beschreibungen
- Definition von *GUI-Komponenten*

Die Definitionen der *GUI-Komponenten* nehmen einen eigenen Komplex innerhalb der GUI-Beschreibung ein. Bezogen auf die *Basiskomponenten* der GUI ist die Beschreibung eines Textes wichtig. Im Falle eines Buttons oder eines Labels (andere *Basiskomponenten* sind in dieser Iteration nicht umgesetzt) beschreibt dieser die Aufschrift der Komponente. Weiterhin ist es für die Zuweisung zu einem Bereich wichtig, dass diese Komponenten innerhalb der Datei referenziert werden können. Daher muss für jede *GUI-Komponente* eine Bezeichnung definiert werden, die innerhalb der Datei eindeutig ist.

An den *Basiskomponenten* können darüber hinaus Interaktionen beschrieben werden. Hierzu sind Informationen über den Interaktionstyp nötig. Der einzige, in dieser Iteration umgesetzte, Interaktionstyp ist ein Klick auf die Komponente. An dieser Interaktion können ebenso Aktionen definiert werden, die Auswirkungen auf andere Komponenten haben. Zusammenfassend ergeben sich folgende Metadaten der *Basiskomponenten*.

- Typ
- Bezeichnung
- Text
- Interaktion (siehe semantisches Modell *Interaction*)

Die Interaktion benötigt folgende Attributen, die beschrieben werden müssen.

- Bezeichnung
- Interaktionstyp
- Aktion

Aktionen nehmen wiederum einen eigenen Komplex innerhalb der Komponentendefinition ein. Dabei werden folgende Informationen benötigt.

- **Aktionstyp**
Zur Unterscheidung zwischen Interaktionen mit anderen *GUI-Komponenten* oder fachlichen Modellen
- **Element**
Ein Verweis auf das Element, mit dem interagiert werden soll.
- **Attribute** (siehe semantisches Modell *Property*)
Die zu verändernden Attribute des Elements.

Die *komplexen Komponenten* werden in einer eigenen Komponentendefinition beschrieben. Grund dafür ist, dass neben den vordefinierten Funktionalitäten der *komplexen Komponenten* auch weitere optionale Wertzuweisungen möglich sein sollen. Dazu wird nach der Implementierung der Komponente für jedes Framework ein neues Schlüsselwort für eine Komponentendefinition in die Grammatik eingebaut. Jede komplexe Komponente benötigt darüber hinaus eine Bezeichnung um referenziert zu werden. In dieser Iteration ist eine *Multiselection-Komponente* umgesetzt. Diese Komponente ist generisch implementiert. Der generische Typ muss innerhalb der Komponenten in der GUI-Beschreibung definiert werden. Ebenso müssen die Werte, die in dieser Komponente selektiert werden können, angegeben werden. Zusätzlich sollen optional auch die Werte angegeben werden, die bereits selektiert wurden.

Semantisches Modell

Das Artefakt, welches beim diesem Modell im Mittelpunkt steht ist die *UI-Description* (siehe Abbildung 15). Die Methoden werden zum Erhalt der Übersichtlichkeit nur in den Interfaces abgebildet. In den Klassen sind lediglich die globalen Variablen dargestellt. Die aggregierten Artefakte auf die schon im vorherigen Abschnitt verwiesen wurde, sind aus dem Diagramm gut zu entnehmen.

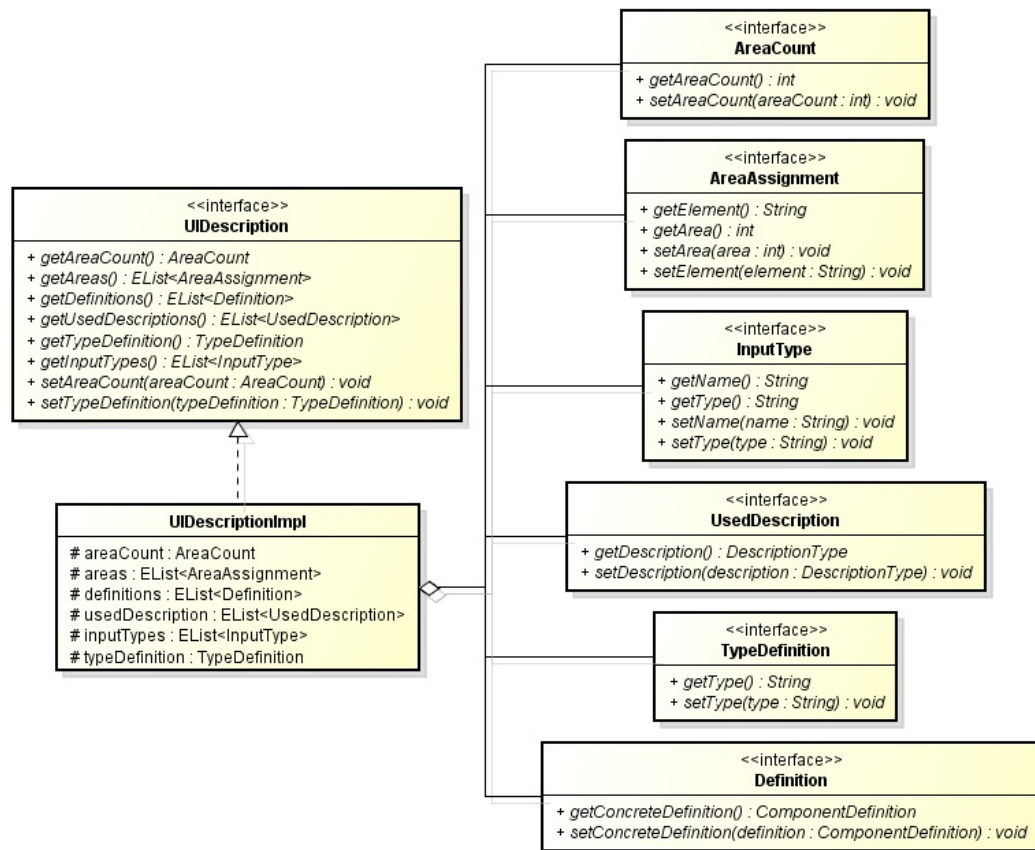


Abbildung 14: 1. Iteration: UIDescription

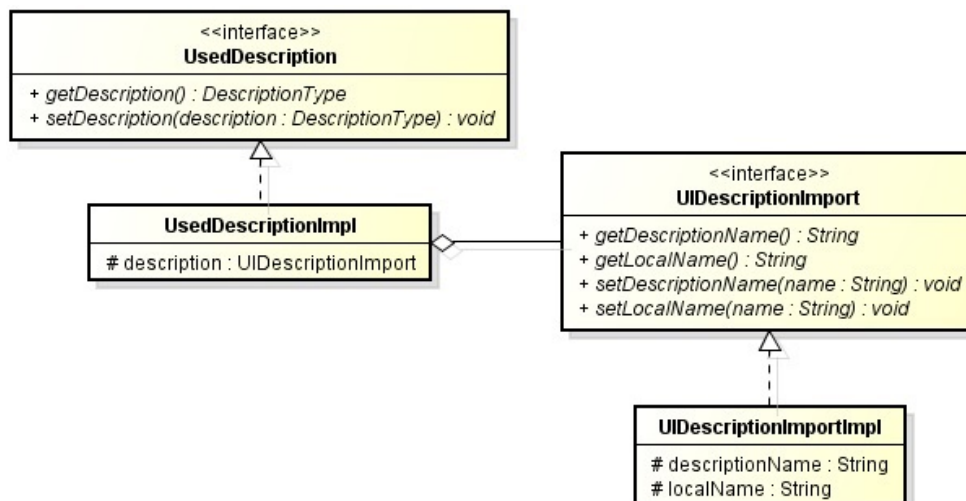


Abbildung 15: 1. Iteration UsedUIDescription

Die Klasse *DefinitionImpl* aggregiert weitere Artefakte des Modells. Diese sind um die Übersicht zu wahren Abbildung 17 zu entnehmen.

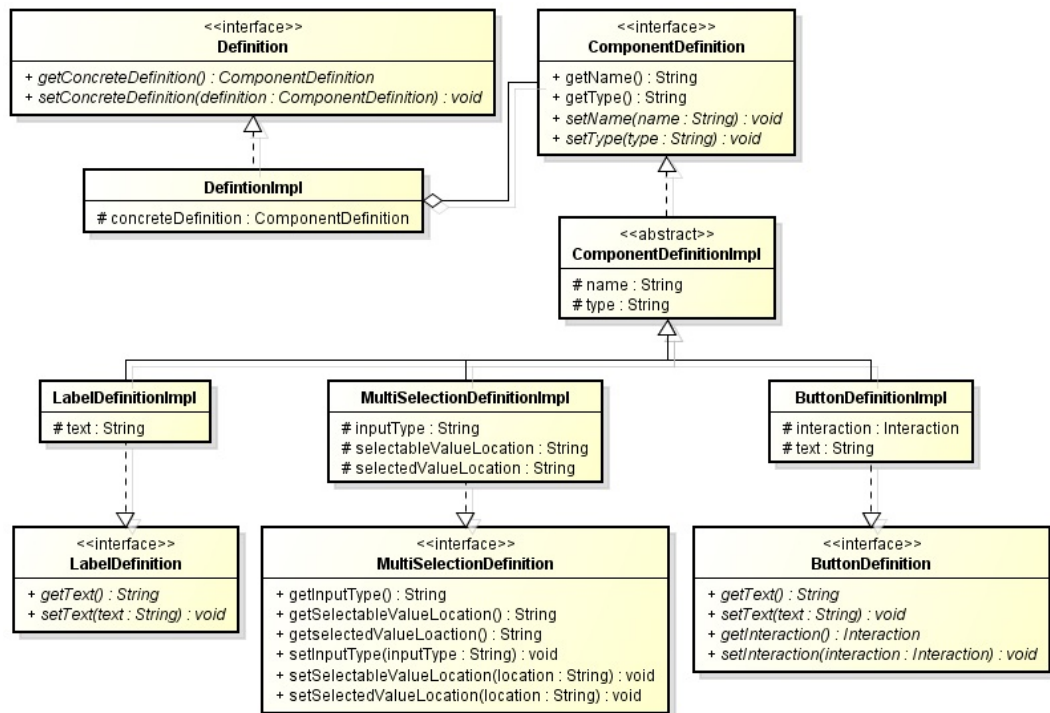


Abbildung 16: Teil 3: GUI-Beschreibungsmodell Version 1

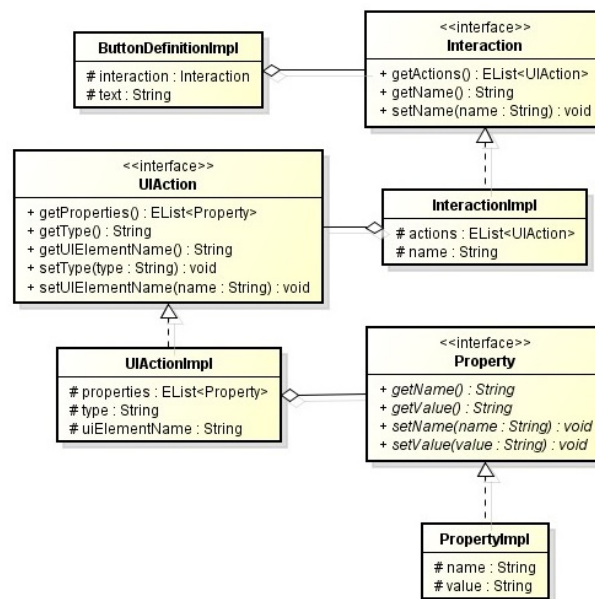


Abbildung 17: Teil 4: GUI-Beschreibungsmodell Version 1

Dort sind die drei umgesetzten Ausprägungen einer *Definition* zu erkennen. Dabei handelt es sich um *Label*, *Button* und *MultiSelection*. Weiterhin ist zu erkennen, dass nur der *Button* eine *Interaction* enthalten kann. Das Interface *Property* wird benötigt um bestimmte Werte an *GUI-Komponenten* zu setzen, ohne wissen zu müssen um welchen Komponententyp es sich han-

delt. Dazu wurden die allgemein gültigen Einstellungsmöglichkeiten von *Basiskomponenten* in *CommonProperty* zusammengefasst.

Konkrete Syntax

Folgender Auszug aus einem GUI-Skript enthält sämtliche Features, die im Prototypen der ersten Iteration umgesetzt wurden.

Listing 2: 1. Iteration: Syntax

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

Die Bezeichnung *Area* wurde bewusst so gewählt, da dieser Begriff abstrakter ist als die in verschiedenen GUI-Frameworks verwendeten Begriffe wie, Panel oder Pane. In der Syntax dieser DSL gilt es sich vor allem bzgl. des Aufbaus der GUI an keinem GUI-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 Referenz-Wert verwendet wird, der in dieser Beschreibung nicht deklariert wurde (*valuepackage.Values*). Dabei handelt es sich um einen qualifizierten Namen einer Klasse.

Die dazugehörige Grammatik befindet sich im Anhang ??.

8.2 2. Iteration

Analyse der Metadaten

In den Metadaten der GUI-Beschreibung muss in dieser Iteration eine Properties-Datei angegeben werden, in der bestimmte Werte für die Attribute der *GUI-Komponenten* enthält.

Da die Möglichkeit bestehen soll die, in den eingebundenen GUI-Skripte definierten, Komponenten in dieser Iteration zu verändert, wird eine weitere Ergänzung für die GUI-Beschreibung benötigt. Diese Veränderungen sollen sich sowohl semantisch als auch syntaktisch von der Komponentendefinition abgrenzen (siehe semantisches Modell *Refinement*). Um die eindeutige Referenzierung zu ermöglichen muss bei der Bezeichnung der eingebundenen GUI-Skripte sowie bei der veränderten Komponente der qualifizierte Name angegeben werden.

Bei den Interaktionen der *Basiskomponenten* fällt die Aktion komplett weg. Somit muss nur noch der Interaktionstyp angegeben werden.

In den Definitionen der *Basiskomponenten* muss aufgrund des Properties-Konzeptes die Möglichkeit bestehen, einen Property-Schlüssel anzugeben. Alle anderen Metadaten für die *Basiskomponenten* bleiben bestehen.

Bezogen auf die *komplexen Komponenten* ist es lediglich notwendig den Input-Typ anzugeben. Die Festlegung über selektierbare und selektierte Elemente in der Multiselection-Komponente wird nicht benötigt. Das ermöglicht, die *komplexen Komponenten* mittels *use* (siehe semantisches Modell *UsedDefinitions*) in die GUI-Beschreibung einzubinden (siehe konkrete Syntax).

Semantisches Modell

In dieser Iteration wurden an den Artefakten *AreaCount*, *TypeDefinition* und *AreaAssignment* keine Änderungen vorgenommen. Artefakte wie *Property* und *Refinement* sind hinzugekommen. Die weiteren Artefakte, die von *UIDescriptionImpl* aggregiert werden (siehe Abbildung 17), wurden verändert.

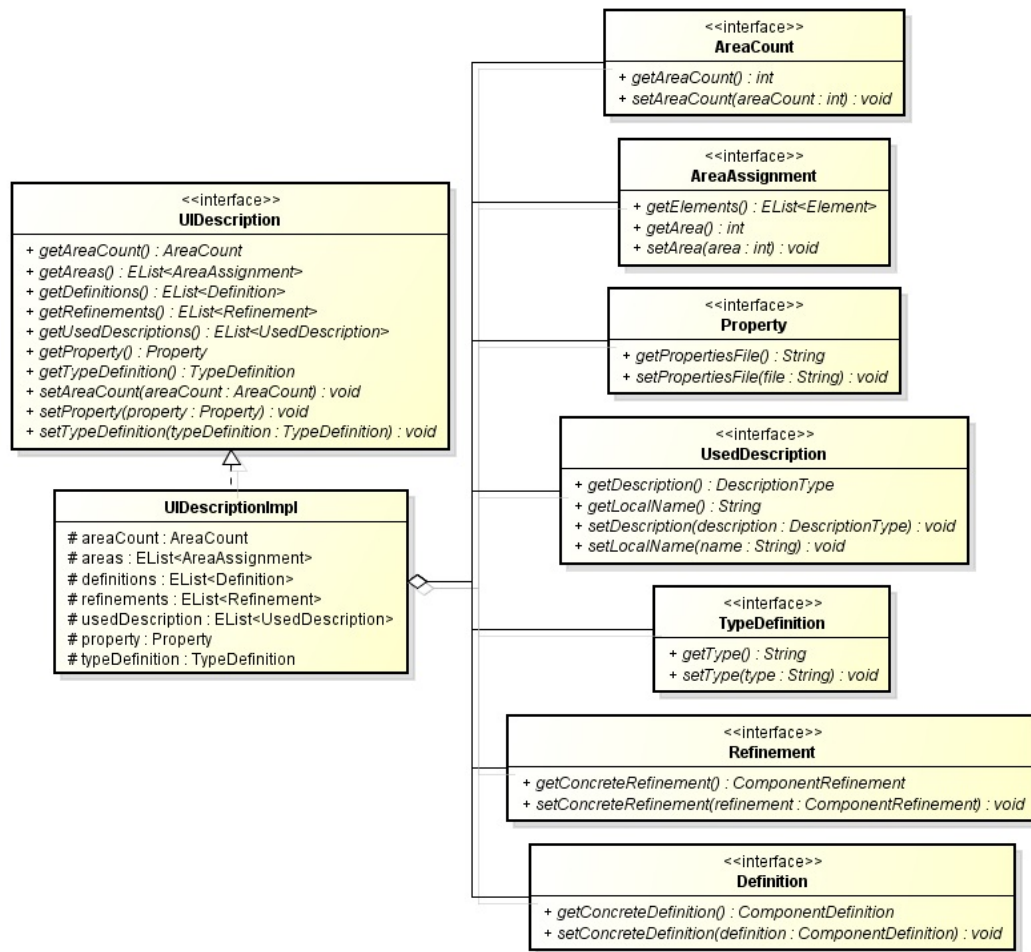


Abbildung 18: Teil 1: GUI-Beschreibungsmodell Version 2

Das Artefakt *Property* bildet die Property-Datei ab. Sie ist nicht zu verwechseln mit dem Artefakt *Properties*, welches die Eigenschaften von Komponenten abbildet. Abbildung 19 zeigt beide Artefakte auf.

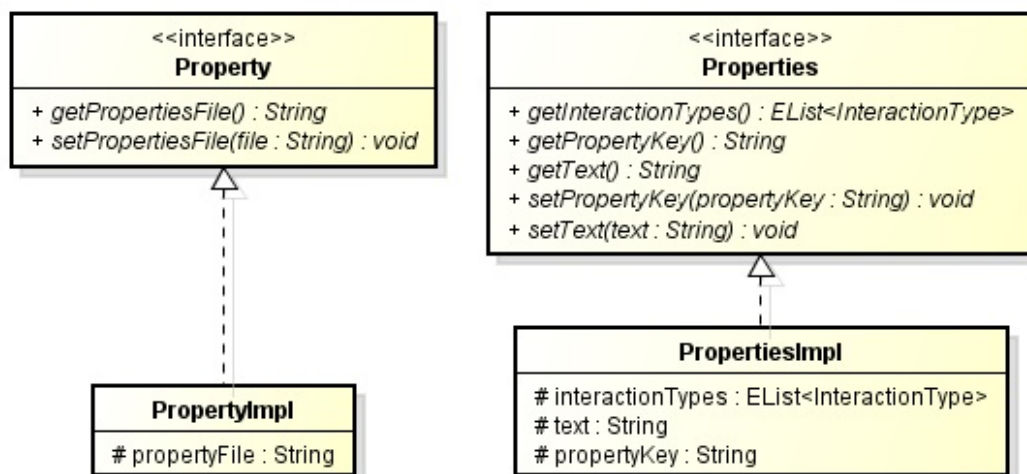


Abbildung 19: Teil 2: GUI-Beschreibungsmodell Version 2

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

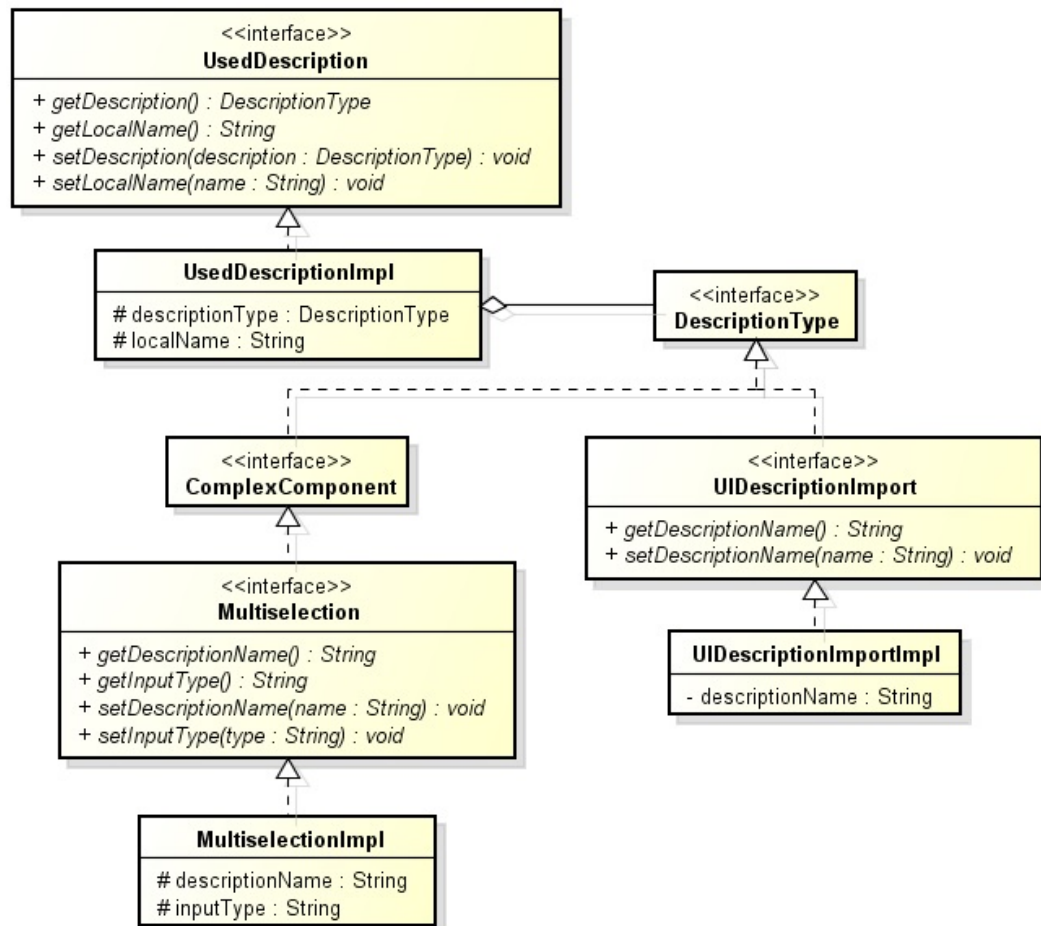


Abbildung 20: Teil 3: GUI-Beschreibungsmodell Version 2

Weiterhin wird eine Unterscheidung zwischen *Definition* und *Refinement* vorgenommen. Die *Definition* bildet neu definierte Komponenten für das GUI ab. Ein *Refinement* hingegen beschreibt die veränderten Komponenten importierter GUI-Skripte (siehe Abbildung 21 und Abbildung 22).

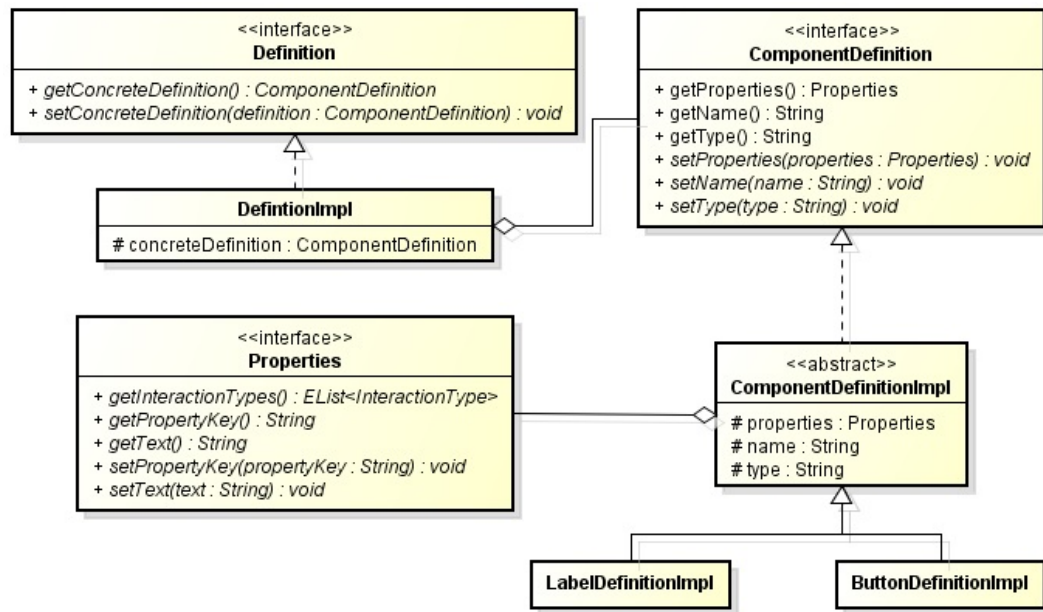


Abbildung 21: Teil 4: GUI-Beschreibungsmodell Version 2

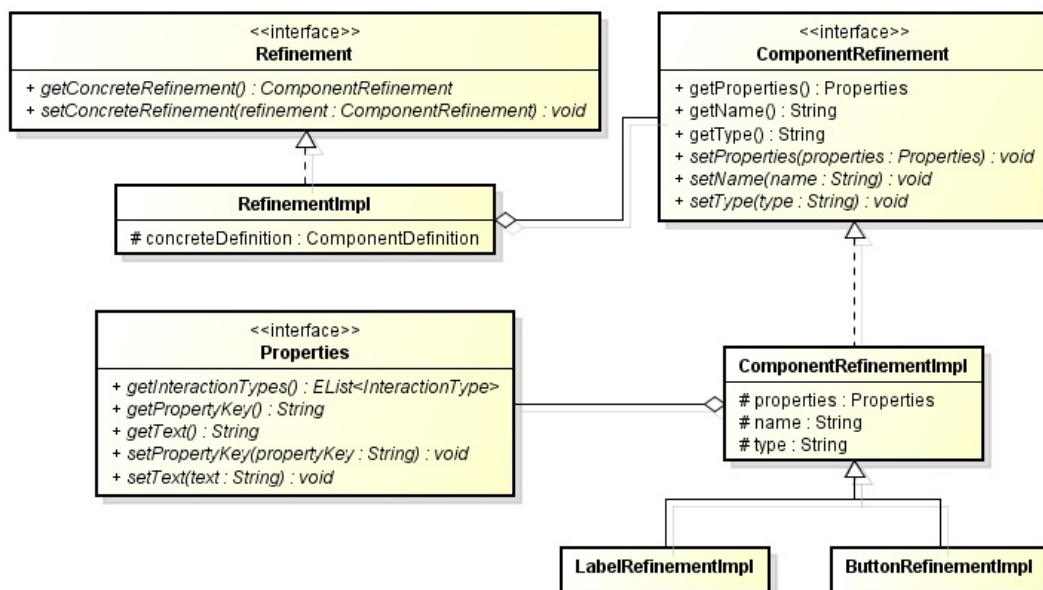


Abbildung 22: Teil 5: GUI-Beschreibungsmodell Version 2

Konkrete Syntax

Eine Veränderung der Syntax in der zweiten Iteration ist ein neues Schlüsselwort zur Festlegung der Properties-Dateien. Um eine Properties-Datei einzubinden muss, wie in Listing 3, eine entsprechende Datei angegeben werden und in den Komponentendefinitionen entsprechende Schlüssel deklariert werden. Das Label mit der Bezeichnung *OneLabel* enthält keinen

Property-Key. In diesem Fall wird der Titel als solcher verwendet.

Listing 3: 2. Iteration: Properties

```
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 Menge der Metadaten für eine Interaktion stand die Frage offen, ob die Interaktionstypen einfach hintereinander mit Komma, oder untereinander mit dem entsprechenden Schlüsselwort aufgezählt werden sollen. Aufgrund der Anforderung AS4, wird die erste Variante bevorzugt (siehe Listing 4).

Listing 4: 2. Iteration: Interaktion

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

Die komplexen Komponenten werden wie in Listing 5 mit der Komponente Multiselection gezeigt ist, über das Schlüsselwort *use* eingebunden werden. Der Input-Typ kann dabei optional innerhalb der Zeichen < und > angegeben werden.

Listing 5: 2. Iteration: Definition komplexer Komponenten

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

Für die Zuweisung mehrerer Komponenten zu den Bereichen (*Area*) kamen zwei Lösungen in Betracht. Bei der einen finden die Definitionen der Komponenten zusammen mit der Zuweisung zu dem Bereich statt. Dies könnte bspw. wie in Listing 6 dargestellt werden.

Listing 6: 2. Iteration: Button und Label (1)

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

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 Bereich zugewiesen werden müssten. Da die mit *use* eingebundenen Komponenten auch Bereichen zugeordnet werden, würde für dieses Verfahren ein zusätzliches syntaktisches Konzept innerhalb der Zuweisung benötigt werden. Um dies zu umgehen wurde die Entscheidung getroffen, das alte Verfahren zu verfeinern. Listing 7 zeigt ein Beispiel für die Zuweisung von drei Komponenten zu einem Bereich.

Listing 7: 2. Iteration: Area-Zuweisung (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 der Werte von Komponenten, die in einem eingebundenen GUI-Skript definiert wurden, können über das Schlüsselwort *REFINE* getätigt werden. Der erste Teil von Listing 8 zeigt die Originaldatei, deren Beschreibung eingebunden wird. Diese trägt den Namen *LabelAndButton* und befindet sich im Package *guidescription*. Der zweite Teil zeigt, wie die Aufschrift einer Komponente *Button* überschrieben wird.

Listing 8: 2. Iteration: Verändern von Komponenten eingebundener GUI-Beschreibungen

```

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

```



```

14 Area count: 1
15 type: WINDOW
16 use: "guidescription.LabelAndButton" as: 'Embedded'
17 REFINE Button with name: 'Button':
18     text='NewText'
19 END REFINE
20 Area:1<- 'Embedded'

```

Sollten mehrere GUI-Skripte eingebunden sein, in denen Komponenten mit demselben Namen definiert sind, muss die Bezeichnung der eingebundenen Ressource zur eindeutigen Identifikation in der Referenz stehen (siehe Listing 9).

Listing 9: 2. Iteration: Verändern von Komponenten eingebundener GUI-Beschreibungen mit Namensüberschneidung

```

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

```

Die dazugehörige Grammatik, welche in der zweiten Iteration entwickelt wurde, ist im Anhang ?? zu finden.

8.3 3. Iteration

Analyse der Metadaten

Um eine Referenzierung von *GUI-Komponenten* und dem entsprechenden Layout aus der Layout-Datei zu ermöglichen, ist es wie bei den Properties-Dateien notwendig, die Layout-Datei innerhalb des GUI-Skripts anzugeben. Weiterhin wird in den einzelnen Komponentendefinitionen ein Schlüssel benötigt, über den die Komponente eindeutig referenziert werden kann. Dafür kann die Bezeichnung der *GUI-Komponente* verwendet werden. Um in der Layout-Datei nicht alle einzelnen *GUI-Komponenten* unterscheiden zu müssen, wird innerhalb der GUI-Beschreibung ein optionales Feld benötigt, mittels dessen unterschiedlichen *GUI-Komponenten* derselbe Layout-Schlüssel zugeordnet werden kann.

Da die Anzahl der Bereiche nicht mehr angegeben werden muss, fällt dies aus den Metadaten heraus. Die Zuweisung der *GUI-Komponenten* zu den

Bereichen entfällt ebenfalls. Die Strukturierung wird über eine Aufzählung der *GUI-Komponenten* vorgenommen.

Zusammenfassend werden folgende Metadaten für die Beschreibung einer GUI mit der GUI-DSL benötigt.

- Typ
- Properties-Dateien
- Layout-Dateien
- Eingebundene *GUI-Komponenten*
- Veränderte eingebundene Komponentendefinitionen
- Komponentendefinitionen
- Struktur

Alle Komponentendefinitionen und die Veränderten eingebundenen Komponentendefinitionen benötigen durch die genannten Änderung folgende Metadaten.

- Bezeichnung
- Property-Schlüssel (optional)
- Layout-Schlüssel (optional)
- Interaktionen (optional)

Über die konkreten Komponentendefinitionen müssen mehrere *Basiskomponenten* beschrieben werden können. *Basiskomponenten* die spezielle Metadaten benötigen sind mit diesen in folgender Tabelle aufgelistet.

Basiskomponente	Spezifische Metadaten
Label	Aufschrift
Button	Aufschrift
Textfield	Text, Editierbarkeit
Textarea	Text, Editierbarkeit
Tree	Input-Modell
Table	Input-Modell
TabView	GUI-Beschreibungen für die einzelnen Tabe

Tabelle 4: Basiskomponenten mit spezifischen Metadaten

Mit Ausnahme der Metadaten des TabViews sind alle Angaben zu anderen Basiskomponenten optional.

Semantisches Modell

Das semantische Modell hat sich durch die vielen Änderungen in dieser Iteration ebenso stark verändert. Bei der Betrachtung der *UIDescription* (siehe Abbildung 23) fällt auf, dass *Area* und *AreaCount* nicht mehr vorhanden sind. Hinzugekommen sind *Structure* (worin die Anordnung der *GUI-Komponenten (Element)* in einer Liste abgelegt wird) und *Layout* (worin die zu verwendeten Layout-Dateien in einer Liste abgelegt werden).

Eine weitere Änderungen ist bei *Property* zu finden. Dort ist ebenfalls eine Liste vorhanden und kein allein stehender Wert.

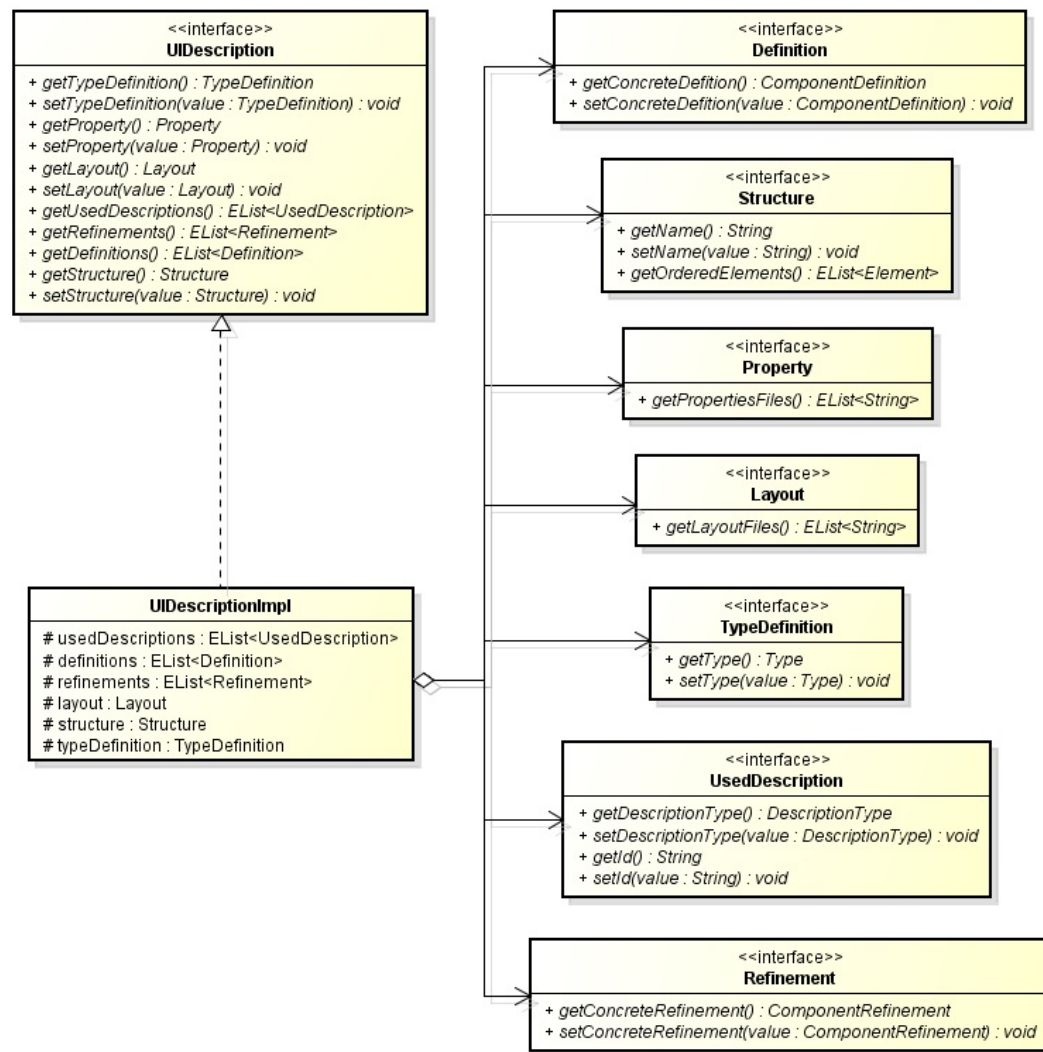


Abbildung 23: 3. Iteration: UIDescription

An *TypeDefinition* und *UsedDescription* wurden keine signifikanten Änderungen vorgenommen. Die meisten Änderungen wurden bei den Artefakten *Refinement* und *Definition* vorgenommen. Der Aufbau der dieser Artefakte ist ähnlich. Das Interface (*Refinement* und *Definition*) wird von einer Klasse implementiert (*RefinementImpl* und *DefinitionImpl*), die mehrere Objekte bestimmter Klassen, die das Interface *ComponentRefinement* oder *ComponentDefinition* implementieren, enthält. In Abbildung 24 ist diese Struktur für die *Definition* abgebildet.

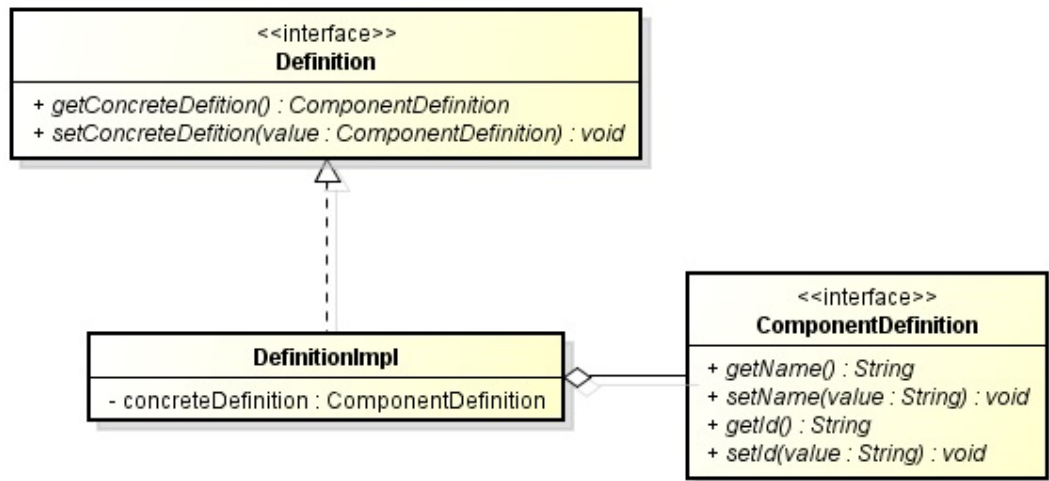


Abbildung 24: 3. Iteration: Definition

Die benannten Klassen bilden die unterschiedlichen *Basiskomponenten* ab, die definiert oder verändert werden können. Jede dieser Klassen aggregiert ein Objekt des Typen *CommonProperties* ein. Dieses Interface bildet die allgemeinen Properties ab. Bei den *Basiskomponenten* Label, Button, Textfield und Textarea ist diese Aggregation transitiv, da die speziellen Properties dieser *Basiskomponenten* als eigenes Artefakt implementiert sind. Abbildung 25 zeigt dies für einen Button auf.

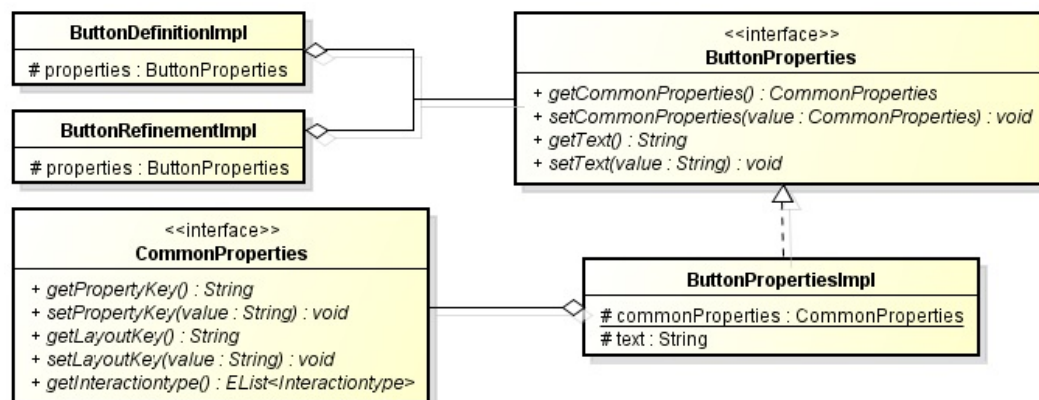


Abbildung 25: 3. Iteration: Button

Eine direkte Aggregation der *CommonProperties* findet bei den *Basiskomponenten* TabView, Table und Tree statt. Für die speziellen Properties dieser *Basiskomponenten* existieren keine einzelnen Klassen. Die folgenden Abbildungen (26, 27 und 28) zeigen diese drei *Basiskomponenten* (Refinement und Definition) mit der Aggregation der *CommonProperties*.

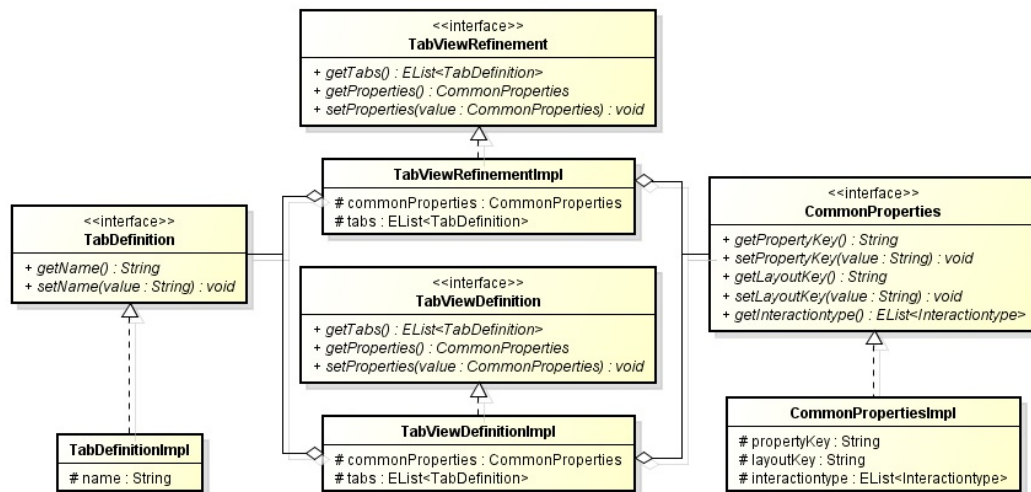


Abbildung 26: 3. Iteration: TabView

Die Basiskomponente *TabView* benötigt eine Menge von *TabDefinitions*. Diese Klasse bildet die Referenz zu den in die *TabView* einzubindenden *GUI-Komponenten*. Die *Basiskomponenten* *Tree* und *Table* benötigen lediglich eine Referenz auf den Input-Typen, den sie abbilden sollen, in Form einer Zeichenkette (siehe Abbildung 27 und 28).

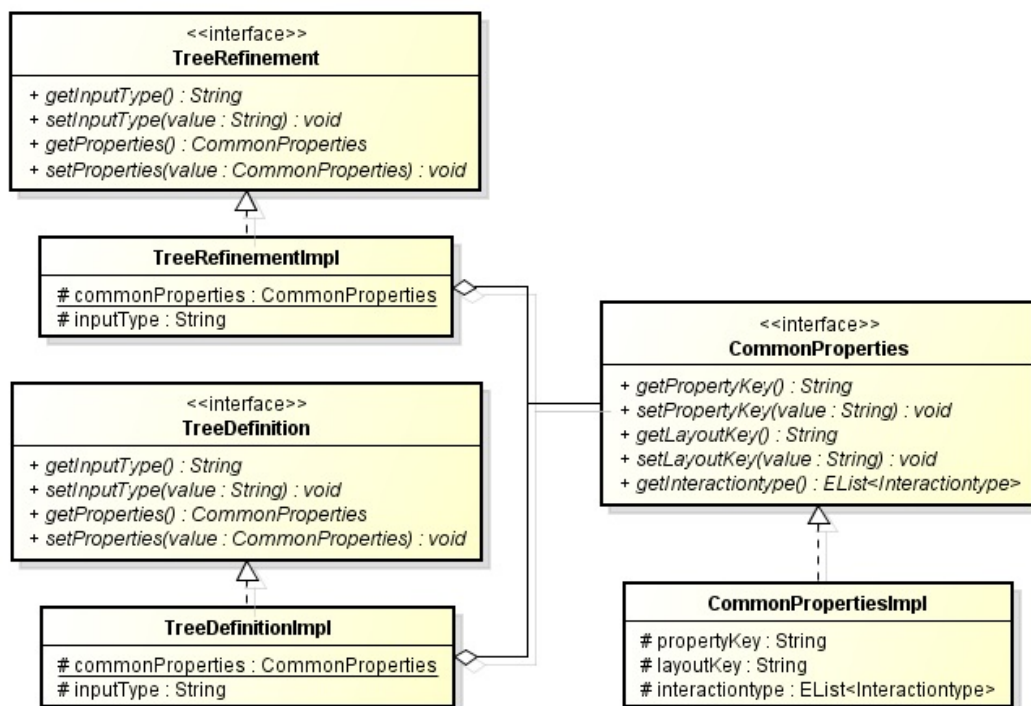


Abbildung 27: 3. Iteration: TabView

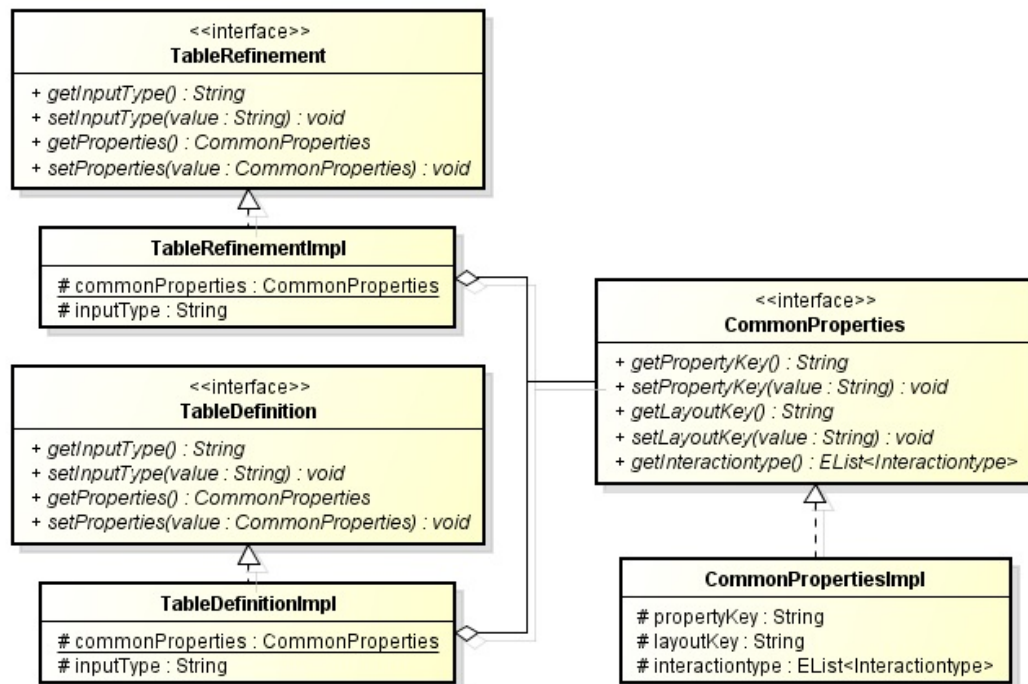


Abbildung 28: 3. Iteration: TabView

Konkrete Syntax

Die syntaktischen Konstrukte wurden in dieser Iteration stark vermehrt. Das liegt vor allem daran, dass viele neue *Basiskomponenten* hinzugekommen sind. Der grundsätzliche syntaktische Aufbau eines GUI-Skripts hat sich jedoch nicht verändert. Eingeleitet wird die Beschreibung weiterhin mit der Typ-Definition, gefolgt von der Angabe der Properties-Dateien. Sofern mehrere Properties-Dateien vorhanden sind, werden diese mit Komma getrennt voneinander aufgezählt.

Dasselbe Prinzip wird bei der sich anschließenden Angabe der Layout-Dateien verwendet. Das Semikolon gilt ab sofort als Trennzeichen für einen abgeschlossen definierten Komplex (siehe Listing 10).

Listing 10: 3. Iteration: Properties- und Layout-Dateien

```

1 type: INNERCOMPLEX;
2 get properties from: 'properties1', 'properties2';
3 get layout from: 'layout1', 'layout2';

```

Für die eingebundenen *GUI-Komponenten* wurden keine großen syntaktischen Veränderungen vorgenommen. Lediglich das Semikolon wird für den Abschluss des Komplexes benötigt (siehe Listing 11).

Listing 11: 3. Iteration: Eingebundene GUI-Komponenten

```
1 use: Multiselection<Input> as: "multi";
```

Die Definitionen der einzelnen *Basiskomponenten* hat sich syntaktisch stark verändert. Grund dafür ist vor allem, dass die Anforderungen AA5 und AS4 mehr Beachtung finden sollen. Somit werden die bekannten *Basiskomponenten* wie folgt definiert (siehe Listing 12).

Listing 12: 3. Iteration: Button und Label

```
1 Button as: "Button" -> propertyKey='buttonproperty' layoutKey='buttonlayout'
2 interactiontype=IfViewImage text='buttontext';
3
4 Label as: 'Label' -> propertyKey='labelproperty' layoutKey='labellayout'
5 interactiontype=IfActivator text='labeltext';
```

Bei den Definitionen der anderen *Basiskomponenten* werden die allgemeinen Properties wie im vorherigen Beispiel zugewiesen. In den Fällen der *Basiskomponenten* Textfield und Textarea werden die speziellen Properties nach demselben Prinzip definiert (siehe Listing 13).

Listing 13: 3. Iteration: Textfield und Textarea

```
1 Textfield as: 'Textfield' -> propertyKey='textfieldproperty'
2 layoutKey='textfieldlayout' interactiontype=IfActivator
3 text='textfieldtext' editable=TRUE;
4
5 Textarea as: 'Textarea' -> propertyKey='textareaproperty'
6 layoutKey='textarealayout' interactiontype=IfActivator
7 text='textareatext' editable=TRUE;
```

Die Syntax für die Definition der *Basiskomponenten* Table und Tree sind ähnlich. Der benötigte Input-Typ wird nach dem Schlüsselwort angegeben, welches die *GUI-Komponente* bestimmt (siehe Listing 14).

Listing 14: 3. Iteration: Table und Tree

```
1 Table<tablemodel> as: 'Table' -> propertyKey='tableproperty'
2 layoutKey='tablelayout' interactiontype=IfActivator;
3
4 Tree<treemodel> as: 'Tree' -> propertyKey='treeproperty'
5 layoutKey='treelayout' interactiontype=IfActivator;
```

Die *Basiskomponenten* TabView werden die verwendeten GUI-Komponenten der einzelnen Tab ebenfalls nach dem Schlüsselwort angegeben, welches die *GUI-Komponente* festlegt. Hierbei können anderes als bei den *Basiskomponenten* Table und Tree mehrere Angaben getätigt werden (siehe Listing

15).

In den folgenden Beispielen wird davon ausgegangen, dass die vorher genannten Beispiele für die *Basiskomponenten* Tree, Table und Textarea in demselben GUI-Skript definiert wurden. Durch das syntaktische Konstrukt in Listing 15 wird eine Tab-Ansicht mit drei Taben beschrieben. Das erste Tab enthält den Tree, das zweite die Table und das dritte die Textarea.

Listing 15: 3. Iteration: TabView

```
1 TabView[ Tree ][ Table ][ Textarea ] as: 'tabview' -> propertyKey='tabviewproperty'
2 layoutKey='tabviewlayout' interactiontype=IfViewImage;
```

Dabei ist zu erwähnen, dass die Angabe der Properties in den meisten Fällen optional ist. Lediglich die speziellen Properties der *Basiskomponenten* Table, Tree und TabView müssen zwingend angegeben werden.

Die Syntax zur Veränderung einer *Basiskomponente* eines eingebundenen GUI-Skripts ähnelt der Definition einer *Basiskomponente* des gleichen Typs. Es muss lediglich angegeben werden, welche *GUI-Komponenten* in welcher GUI-Beschreibung verändert werden soll.

Folgendes Beispiel zeigt, wie die Aufschrift eines Buttons eines eingebundenen GUI-Skripts verändert wird (siehe Listing 16). Der Button trägt die Bezeichnung *EmbeddedButton*. Das GUI-Skript liegt im Package *guidescription* und trägt die Bezeichnung *Embedded*.

Listing 16: 3. Iteration: TabView

```
1 use: "guidescription.Embedded" as: "embedded";
2 Button change: 'embedded.EmbeddedButton' -> text='Neuer Text';
```

Der letzte Teil einer GUI-Beschreibung beinhaltet die Angabe der Struktur. Diese Angabe ähnelt der Zuweisung von *GUI-Komponenten* zu Bereichen, wie es auch der ersten und zweiten Iteration bekannt ist. Da dieses Konstrukt vereinfacht werden sollte, werden die *GUI-Komponenten* in der richtigen Reihenfolgen nach dem Schlüsselwort *Struktur* aufgezählt (siehe Listing 17).

Listing 17: 3. Iteration: Struktur

```
1 Structure: 'Button', 'Label', 'Textfield', 'tabview';
```

Die damit für diese Arbeit endgültige Grammatik ist, wie die Grammatiken der anderen Iterationen, im Anhang ?? zu finden.

Kapitel 9

Entwicklung des Generators zur Generierung von Klassen für das Multichannel-Framework

Alle Umsetzungen die in diesem Kapitel beschrieben werden, fanden in der 3. Iteration statt. Von daher wird nicht mehr zwischen Iterationen unterschieden.

Ziel ist es mit dem Generator und einem entsprechenden GUI-Skript eine Exploreransicht zu erstellen, wie sie in profil c/s geläufig ist. Abbildung 29 zeigt den Aufbau einer solchen GUI. Darin enthalten sind zwei Bäume (auf der linken Seite) und ein Interchangeable (auf der rechten Seite), worin der Inhalt der ausgewählten Elemente des Inhaltsbaums angezeigt werden soll. Das Anzeigen des Inhalts wird jedoch nicht umgesetzt.

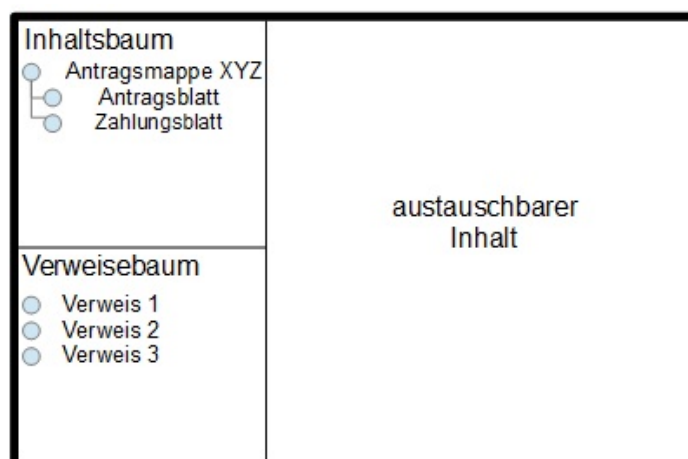


Abbildung 29: Aufbau eines Explorers

Die notwendigen GUI-Skripte werden bei der Umsetzung vorgestellt.

9.1 Beschreibung der GUI-, FP- und IP-Klassen

Durch die Beschreibung der GUIs mit der entwickelten *GUI-DSL*, ist es möglich die GUI-, FP- und die IP-Klassen (siehe Kapitel 2) zu generieren. Die GUI-Klasse soll dabei vollständig generiert werden. Dafür werden Informationen über das Layout aus der Layout-Datei benötigt. Die Umsetzung und Einbindung der Layout-Datei wird nicht in dieser Arbeit behandelt. Aus diesem Grund wird im Generator ein Layout festgelegt. Die IP- und FP-Klassen sollen nur teilweise generiert werden. Somit unterteilt sich das folgende Kapitel in drei Abschnitte (GUI-Klassen, FP-Klassen und IP-Klassen).

In den GUI-Klassen der *deg* werden die *GUI-Komponenten* durch *Präsentationsformen* beschrieben (siehe Kapitel 2). Alle *GUI-Komponenten* sind in diesen Klassen global verfügbar. Die globalen Variablen stehen bei der *deg* am Ende der Klasse. Die bestehende Struktur der Klassen soll nicht verändert werden. Da die Interaktion von der Präsentation getrennt ist, müssen zur Referenzierung von Interaktionen zu *GUI-Komponenten* entsprechende Schlüssel vergeben werden (siehe Routinearbeit R1).

Die IP-Klassen ordnen den *GUI-Komponenten* mit Hilfe dieses Schlüssels entsprechende Interaktionen und darauf folgende Kommandos zu. Was genau bei der Interaktion gesehen soll, kann vom Generator nicht erzeugt werden. Dies muss vom Entwickler nachgepflegt werden.

In den FP-Klassen ist die Funktionalität des Werkzeugs beschrieben. Eine komplette Generierung der FP-Klassen kann mit der DSL nicht angestrebt werden, weil dafür entsprechende Informationen fehlen. Dennoch kann der Klassenrumpf und der Konstruktor erzeugt werden.

Für die Generierung der Klassen wird in dieser Arbeit Transformer Generation (siehe Kapitel 3) verwendet. Für die Generierung der IP- und FP-Klassen ist auch die Templated Generation (siehe Kapitel 3) in Erwägung zu ziehen. Um bei der Art der Generation einheitlich zu sein, wird auf diese Möglichkeit nicht weiter eingegangen.

9.2 Umsetzung des frameworkspezifischen Generators

GUI-Klassen

Bei Betrachtung der Präsentation aus Abbildung 29 können die Bäume auf der linken Seite als einzelne GUI-Beschreibungen betrachtet werden, die jeweils mit einem fachlichen Konzepten assoziiert werden können. Der obere Baum kann mit dem fachlichen Konzept des *Inhaltsbaumes* in Verbindung gebracht werden und der untere mit dem des *Verweisebaums*. Da sich die fachlichen Konzepte erkennen lassen, sollten separate GUI-Skripte erstellt werden (siehe Listing 18 und 19).

Listing 18: GUI-Skript für den Inhaltsbaum

```
1 type: INNERCOMPLEX;
2 Label as: 'kopfzeile' -> text = 'Inhaltsbaum';
3 Tree[ testwerkzeuge.modelle.InhaltsModell ] as: 'inhaltsbaum';
4 Structure: 'kopfzeile', 'inhaltsbaum';
```

Listing 19: GUI-Skript für den Verweisebaum

```
1 type: INNERCOMPLEX;
2 Label as: 'kopfzeile' -> text = 'Verweisebaum';
3 Tree[ testwerkzeuge.modelle.VerweiseModell ] as: 'verweisebaum';
4 Structure: 'kopfzeile', 'verweisebaum';
```

Bei der Generierung betrachtet Xtext eine GUI-Beschreibung im Ganzen. Für jede GUI-Beschreibung soll eine eigene GUI-Klasse angelegt werden. Die Klassen enthalten in diesen beiden Fällen zwei globale Variablen. Darüber hinaus enthalten sie Importe, die am Anfang der beiden Klassen stehen. Um das GUI-Skript für jede zu generierende Datei nur einmal analysieren zu müssen, ist es notwendig Importe, Methoden und globale Variablen zwischen zu speichern. Die Methoden in Listing 20 realisieren das Speichern der Importe und globalen Variablen beim generieren der Methoden. So ist es möglich die bestehende Struktur der Klassen der deg beizubehalten.

Listing 20: Speichern der Importe und der globalen Variablen

```
1 def addImport(String newImport) {
2     if (!imports.contains(newImport)) {
3         imports.add(newImport)
```

```

4         }
5     }
6
7 def addGlobalVar(String globalVar) {
8     if (!globalVars.contains(globalVar))
9         globalVars.add(globalVar)
10 }

```

Zu Beginn einer Generierung muss zwischen den Typen der GUI-Beschreibung unterschieden werden. Je nachdem ob die Beschreibung als *Window* oder *Innercomplex* definiert ist, werden entsprechende Importe benötigt. In den Fällen der oben genannten Bäume wird ein *Innercomplex* definiert (siehe Listing 21).

Listing 21: Generierung eines Innercomplex

```

1 def compileComplex(UIDescription description) '''
2     «addImport("import DE.data_experts.jwammc.core.pf.PfPanel;")»
3     public class «guiFilename» extends PfPanel{
4         «description.genRest»
5     }
6 '''
7
8 def genRest(UIDescription description) '''
9     «addImport("import java.awt.BorderLayout;")»
10    public «guiFilename»() {
11        super( new BorderLayout() );
12        try {
13            init();
14        }
15        catch ( Exception e ) {
16            e.printStackTrace();
17        }
18    }
19    «description.init»
20    «genGlobalVars»
21 '''

```

In der Methode *compileComplex* wird festgelegt, dass sich die *GUI-Komponenten* auf einem *Border-Layout* anordnen. Wenn die *Layout-Datei* verwendet wird, muss der Generator aus dem Inhalt dieser Datei auf einen entsprechenden *Layout-Container* schließen. Zum Abschluss der Methode *genRest* werden zwei weitere Methoden aufgerufen. Die erste Methode (*description.init*) generiert die im Konstruktor aufgerufene Methode *init*. Die andere Methode *genGlobalVars* ist für die Generierung der globalen Variablen zuständig. In der Methode *init* werden alle *GUI-Komponenten* und das *Layout* definiert.

Da für das Layout in dieser Arbeit keine Beschreibung existiert, müssen diese Angaben nachgepflegt werden. Die Definition der *GUI-Komponenten* mit ihren Properties können jedoch erzeugt werden. Listing 22 zeigt die Methode *init* ohne Berücksichtigung des Layouts.

Listing 22: Generierung der Methode *init* der GUI-Klassen

```

1 def getInit(UIDescription description) '''
2     public void init() {
3         «FOR def : description.definitions»
4             «def.compileComponent»
5         «ENDFOR»
6     }
7 '''

```

In der Methode *compileComponent* wird geprüft um welche *GUI-Komponente* es sich handelt. Diese wird anschließend compiliert, wobei der entsprechende Quellcode zur frameworkspezifischen Definition der Komponente generiert wird. Im Fall des Labels werden der entsprechende Import und die globale Variable hinzugefügt. Weiterhin muss die Referenz für die IP-Klasse definiert werden, da Labels Standardinteraktionen besitzen. Wenn vorhanden, müssen letztlich die Properties am Label gesetzt werden (siehe Listing 23).

Listing 23: Generierung eines Labels

```

1 def compileLabel(LabelDefinition definition) '''
2     «addImport("import DE.data_experts.jwammc.core.pf.PfLabel;")»
3     «addGlobalVar("PfLabel " + definition.id + ";" »)
4     «definition.id» = new PfLabel();
5     «definition.id».setIfName("«definition.id»");
6     «IF definition.properties != null»
7         «genProperty(definition.id, 'setText', definition.properties.text,
8             true)»
9     «ENDIF»
10
11 def genProperty(String id, String method, String value, Boolean isString) '''
12     «IF value != null»
13         «IF isString»
14             «id».«method»("«value»");
15         «ELSE»
16             «id».«method»(«value»);
17         «ENDIF»
18     «ENDIF»
19     '''
20 '''

```

Die Eigenschaften aus den Properties-Dateien müssten an dieser Stelle zusätzlich berücksichtigt werden. Dies wurde aus Zeitgründen nicht umgesetzt.

Der Quellcode für den Baum wird ähnlich generiert. Wenn der Input-Typ des Baumes nicht gesetzt ist, muss ein Standard-Wert dafür eingesetzt werden. Des Weiteren muss für den Baum ein *CellRenderer* definiert werden (siehe Listing 24).

Listing 24: Generierung eines Trees

```

1 def compileTree(TreeDefinition definition) '''
2     «addImport("import DE.data_experts.jwammc.core.pf.PfTree;")»
3     «addImport("import DE.data_experts.jwammc.core.pf.TreeCellRenderer;")»
4     «addImport("import DE.data_experts.jwammc.core.pf.PfTree;")»
5     «addImport("import javax.swing.tree.DefaultTreeModel;")»
6     «addGlobalVar("PfTree " + definition.id + ";")»
7     «definition.id» = new PfTree();
8     «definition.id».setIfName("«definition.id»");
9     «IF definition.inputType == null»
10         «addImport("import DE.data_experts.util.ObjectNode;")»
11         «definition.id».setTreeModel( new DefaultTreeModel( new ObjectNode
12             () ) );
13     «ELSE»
14         «definition.id».setTreeModel( new DefaultTreeModel(
15             new «definition.inputType.substring(1, definition.inputType.length
16                 -1)»() ) );
17     «ENDIF»
18     «definition.id».setCellRenderer( new TreeCellRenderer() );
19 '''

```

Durch die genannten Methoden kann der Generator die beiden GUI-Skripte zur Beschreibung des Inhalts- und des Verweisebaums in GUI-Klassen transformieren, die innerhalb der MCF ausgeführt werden können.

Die generierten Dateien (*GuiInhaltsbaum.java* und *GuiVerweisebaum.java*) befinden sich auf dem beiliegenden Datenträger (siehe Anlage) im Projekt *Explorer*. Abbildung 30 zeigt die beiden GUIs, welche bei der Ausführung der generierten Dateien im Kontext von profil c/s erzeugt werden.



Abbildung 30: Generierte GUI des Inhalts- und Verweisebaums

Um die Explorer-GUI zu generieren, müssen diese beiden Bäume zusammen mit einem austauschbaren Bereich in einem GUI-Skript definiert werden (siehe Listing 25).

Listing 25: GUI-Skript für das Explorer-GUI

```

1 type: WINDOW;
2 use: "Inhaltsbaum" as: 'inhaltsbaum';
3 use: "Verweisebaum" as: 'verweisebaum';
4 Interchangeable as: "austauschbarerBereich";
5 Structure: 'inhaltsbaum', 'verweisebaum', 'austauschbarerBereich';

```

Da es sich um ein *Window* handelt, wird in diesem Fall die Methode *compileWindow* aufgerufen. Der einzige Unterschied zur Methode *compileComplex* ist in diesem Fall die Oberklasse (siehe Listing 26).

Listing 26: Generierung eines Windows

```

1 def compileWindow(UIDescription description) '''
2     «addImport("import DE.data_experts.jwammc.core.pf.PfRootPane;")»
3     public class «guiFilename» extends PfRootPane{
4         «description.genRest»
5     }
6 '''

```


Ebenso wie bei der Generierung der ersten beiden GUIs, wird hier als Layout-Container ein Border-Layout verwendet. Um die Anordnung der Bäume wie gewünscht zu erhalten, wird ein weiterer Layout-Container benötigt. Da sich diese Information auf das Layout bezieht, muss sie normalerweise von der Layout-Datei geliefert werden.

Die GUI-Klassen der eingebundenen Gui-Skripte für die Bäume werden in der zu generierenden GUI-Klasse für den Explorer deklariert. Dabei muss der Typ der *UsedDescription* im Vorfeld überprüft werden. Handelt es sich um den Typ *UIDescriptionImport*, ist lediglich die Deklaration der *GUI-Komponente* und die Einbindung in einen Layout-Container nötig (siehe Listing 27). Anderenfalls müssen die spezifischen Eigenschaften komplexer Komponenten untersucht werden. Darauf wird in dieser Arbeit jedoch nicht weiter eingegangen.

Listing 27: Generierung der Einbindung anderer GUI-Skripte

```

1 def compile(UsedDescription description) '''
2     «IF description.descriptionType instanceof UIDescriptionImport»
3         «var castedDescriptionType = description.descriptionType
4           asUIDescriptionImport»
5         «var usedQualifiedClassName = castedDescriptionType.
6           descriptionName.genGuiFileName»
7         «addGlobalVar(usedQualifiedClassName + ' ' + description.id + ';' )
8           »
9         «description.id» = new « usedQualifiedClassName » ();
10    «ELSE»
11        «genComplexComponent( description ) »
12    «ENDIF»
13 '''

```

Die letzte Komponente, die damit noch nicht in der GUI-Klasse deklariert wurde ist die Interchangeable-Komponente. Da diese *GUI-Komponenten* keine speziellen Properties besitzt, ist die Methode zur Generierung des Quellcodes recht einfach gehalten (siehe Listing 28).

Listing 28: Generierung einer Interchangeable-Komponente

```

1 def compileInterchangeable(InterchangeableDefinition definition) '''
2     «addImport("import DE.data_experts.jwammc.core.pf.PfCardPanel;" ) »
3     «addGlobalVar("PfCardPanel " + definition.id + ";" ) »
4     «definition.id» = new PfCardPanel ();
5     «definition.id».setIfName(" «definition.id» ");
6 '''

```

Die generierten Dateien sind auf dem beiliegenden Datenträger (siehe Anlage) zu finden. Es handelt sich um die Dateien *GuiInhaltsbaum.java*, *GuiVerweisebaum.java* und *GuiExplorer.java*. Abbildung 34 (in Anhang B) zeigt die GUI, welche durch die Klasse *GuiExplorer.java* erzeugt wird.

FP-Klassen

Da für die FP-Klassen lediglich der Klassenrumpf und der Konstruktor generiert wird, ist diese Aufgabe entsprechend einfach. Dafür werden lediglich eine Oberklasse und entsprechende Importe benötigt, wie Listing 29 zu entnehmen ist.

Listing 29: Generierung der FP-Klasse

```

1 def genFpSource() '''
2     «addImport("import DE.data_experts.jwam.tools.FpObject;")»
3     «addImport("import de.jwam.handling.toolconstruction.request.
        RequestHandler;")»
4     public class «specificFilename» extends FpObject{
5         public «specificFilename»( RequestHandler parent ) {
6             super( parent , "«descriptionname»" );
7         }
8     }
9 '''

```

IP-Klassen

Um ein Objekt der IP-Klasse zu instantiieren wird ein Objekt der dazugehörigen FP-Klasse benötigt. Die Generierung der IP-Klasse beginnt wiederum mit der Generierung des Klassenkopfs und entsprechend benötigten Imports (siehe Listing 30).

Listing 30: Generierung der IP-Klasse

```

1 def genIpSource(UIDescription description) '''
2     «addImport("import DE.data_experts.jwam.tools.IpObject;")»
3     «addImport("import de.jwamx.technology.iafpf.guimanagement.IAFContext;")»
4     «addImport("import DE.data_experts.util.degexception.ExceptionManager;")»
5     public class «specificFilename» extends IpObject{
6         public «specificFilename»( IAFContext iafContext , final
            Fp«descriptionname» fp ) {
7             super( iafContext , fp );
8             «addGlobalVar("Fp" + descriptionname + ' fp;')»
9             this.fp = fp;
10            try {

```

```

11         initCommands();
12         initIAFs( iafContext );
13     }
14     catch ( Exception ex ) {
15         ExceptionManager.getManager().addAndShow( ex );
16     }
17 }
18 «description.genIAF»
19 «description.genCommands»
20 «description.genCommandMethods»
21 «genGlobalVars»
22 }
23 '''

```

Wie bei den GUI-Klassen müssen hier ebenfalls die globalen Variablen am Ende der Klasse stehen. Zwischen dem Konstruktor und den globalen Variablen werden die Interaktionsformen mit den Kommandos bestimmt (*genIAF* - siehe Listing 31), den Kommandos bestimmte Methoden zugeordnet (*genCommands* - siehe Listing 34) und die Rümpfe für diese Methoden generiert (*genCommandMethods* - siehe Listing 35).

Bei der Bestimmung der Interaktionsformen werden die im GUI-Skript definierten *GUI-Komponenten* durchlaufen und deren Standard-Interaktionsformen und spezielle Interaktionsformen übersetzt (siehe Listing 31).

Listing 31: Generierung der Interaktionsformen

```

1 def genIAF(UIDescriptiondescription)'''
2     protected void initIAFs( IAFContext iafContext ) {
3         «FOR definition : description.definitions»
4             «definition.compileIAF»
5         «ENDFOR»
6     }
7 '''
8
9 def compileIAF(Definition definition)'''
10     «IF definition.concreteDefition.name == 'Label'»
11         «(definition.concreteDefition as LabelDefinition).
12             compileLabelStandardIAF»
13     «ELSEIF definition.concreteDefition.name == 'Tree'»
14         «(definition.concreteDefition as TreeDefinition).
15             compileTreeStandardIAF»
16     «ENDIF»
17     «««
18         Spezielle Interaktionsformen
19     «definition.compileSpecificInteractionTypes»
20     '''

```

Für die Standard-Interaktionsformen muss unterschieden werden um welche *GUI-Komponente* es sich handelt.

Zu jeder Interaktionsform gibt es ein entsprechendes Kommando, welches zur Generierung des Quellcodes relevant ist. Die Zuordnung von Kommando zu Interaktionsform wird vom Generator vorgenommen (Beispiel Tree - siehe Listing 32). Die dafür verwendeten Methoden (z.B. *genIAFAActivator* oder *genIAFTree*) können häufig wiederverwendet werden.

Listing 32: Generierung der Standard-Interaktionsformen von Trees

```

1 def compileTreeStandardIAF(TreeDefinition definition) '''
2         «definition.id.genIAFAActivator»
3         «definition.id.genIAFTree»
4     '''
5
6 def genIAFTree(String id) '''
7         «genIAFSource("DE.data_experts.jwam.gui.interaction.IfTree","de.
8             jwamx.technology.iafpf.command.cmdSelect",id)»
9
10 def genIAFAActivator(String id) '''
11         «genIAFSource("de.jwamx.technology.iafpf.interaction.ifActivator",
12             "de.jwamx.technology.iafpf.command.cmdActivate", id)»
13     '''

```

Die Methode *genIAFSource* hat ebenfalls einen sehr hohen Wiederverwendungsgrad. Sie wird von allen Methoden, die für die Zuordnung von Interaktionsform zu Kommando zuständig sind verwendet. In dieser Methode wird der Quellcode letztendlich generiert (siehe Listing 33).

Listing 33: Generierung einer Interaktionsform

```

1 def genIAFSource(String iafSource, String commandSource, String id) '''
2     «addImport("import "+ iafSource + ";")»
3     «var iafNameWithPrefix = iafSource.split("\\.").last»
4     «var iafName = iafNameWithPrefix.substring(2)»
5     «addGlobalVar(iafNameWithPrefix + " "+id+iafName+';')»
6     «id+iafName» = («iafNameWithPrefix») iafContext.interactionForm(
7         «iafNameWithPrefix».class, "«id»" );
8     «IF !commandSource.equals("")»
9         «addImport("import "+ commandSource + ";")»
10        «var commandNameWithPrefix = commandSource.getClassOfSource»
11        «var commandName = commandNameWithPrefix.substring(3)»
12        «id + iafName».attach«commandName»Command( «id +
13            commandName»Command );
14        «addCommand(id, commandName)»
15    «ENDIF»
16 '''
17
18 def addCommand(String id, String commandName){
19     addGlobalVar("cmd"+commandName + ' ' + id + commandName + "Command;")

```

```

18         commands.put(id+commandName, commandName);
19         return ""
20     }

```

Die Einzelheiten dieser Methode sind unter anderem abhängig von den Konventionen die bzgl. Namensgebung in der deg getroffen wurden. Der Aufruf der Methode *addCommand* ist für weitere Generierungen von Belang. Nach Abschluss der Generierung des Quellcodes zur Bestimmung der Interaktionsformen und Kommandos, müssen den Kommandos entsprechende Methoden zugeordnet werden. Die Referenz auf die Kommandos bietet die Liste *commands*, die durch den Aufruf der Methode *addCommand* (siehe Listing 33) gefüllt wird. Die Methode *genCommands* ist für die beschriebene Zuordnung zuständig (siehe Listing 34).

Listing 34: Generierung der Kommandoinitialisierung

```

1 def genCommands(UIDescription description) '''
2     protected void initCommands() {
3         «FOR id : commands.keySet»
4             «var commandName = commands.get(id)»
5             «addImport(" import DE.data_experts.jwam.util.CmdAusfuehrer
6                 " + commandName + ";" )»
7             «id»Command = new CmdAusfuehrer«commandName»(
8                 getAusfuehrer() ) {
9                 @Override
10                public void ausfuehren() {
11                    «id»();
12                }
13            };
14        «ENDFOR»
15    }
16 '''

```

Die genauen Bezeichnungen ergeben sich wiederum aus den Konventionen für die Bezeichnungen innerhalb des MCF der deg.

Um Compilierungsfehler zu vermeiden, müssen die Methoden, die vom generierten Quellcode verwendet werden, ebenfalls generiert werden. Dafür ist die Methode *genCommandMethods* zuständig. In dieser Methode wird die Liste *commands* nochmals benutzt, um die Methoden-Rümpfe zu erzeugen (siehe Listing 35). Die Implementierung der generierten Methoden muss der Entwickler übernehmen.

Listing 35: Generierung der Methoden zur Bestimmung der auszuführenden Aktionen bei einer Interaktion

```
1 def genCommandMethods(UIDescription description) '''
2     «FOR commandId : commands.keySet»
3         public void «commandId»() {}
4     «ENDFOR»
5     '''
```

Die Generator-Klasse ist auf dem beiliegenden Datenträger zu finden (siehe Anlage). In den Methoden zur Generierung von *GUI-Komponenten* für die GUI-Klassen sind die Code-Abschnitte, die das Layout bestimmen, entsprechend gekennzeichnet.

Kapitel 10

Zusammenfassung und Ausblick

Mit der *GUI-DSL* ist es möglich *GUIs* zu beschreiben, die für *profil c/s* verwendet werden können. Die Entwicklung der *GUI-DSL* ist mit der 3. Iteration bei weitem nicht abgeschlossen. Es wurde jedoch gezeigt, dass es möglich ist, aus einem GUI-Skript die für *profil c/s* relevanten *GUI*-, *IP*- und *FP-Klassen* zu generieren. Dabei ist es gelungen, die allgemeinen Anforderung (*AA1* bis *AA5*) durch die Konzeption der *GUI-DSL* umzusetzen.

Durch die Verwendung unterschiedlicher Generatoren, ist es möglich verschiedene GUI-Frameworks einzusetzen, mit denen die Darstellung im Web- und Standalone-Bereich gelingt (siehe Anforderung *AA1* und *AA2*).

Dass die *GUIs* auf beiden Plattformen ähnlich aufgebaut sind, hängt von der Verwendung ähnlicher Layout-Dateien ab. Insofern ist ein ähnlicher Aufbau nicht erzwungen worden, aber dennoch umsetzbar (siehe Anforderung *AA3*).

Auf die Möglichkeiten der Erweiterung verwendeter Frameworks, hat die *GUI-DSL* keinen Einfluss (siehe Anforderung *AA4*). Diese Anforderung muss bei der Evaluation neuer GUI-Frameworks beachtet werden.

Die Ausdruckskraft der *GUI-DSL* (siehe Anforderung *AA5*) kann nicht objektiv bewertet werden, weshalb unterschiedliche Meinungen darüber existieren werden, ob diese Anforderung mit der *GUI-DSL* umgesetzt wurde. Die Anforderungen an die Sprache (siehe Anforderung *AS1* bis *AS6*) wurden bis auf Anforderung *AS3* und *AS5* vollständig umgesetzt.

Die Beschreibung von Interaktionen erwies sich als schwierig, da die Aktionen, welche bei den Interaktionen ausgeführt werden sollen, nur schwer abstrahiert werden können. Von daher mussten bei dieser Anforderung Abstriche gemacht werden, sodass mit der *GUI-DSL* nur die Interaktionsform beschrieben werden kann.

Ob die Abstraktionsebene der Layoutbeschreibung (siehe Anforderung AS3) ausreicht, kann erst entschieden werden, wenn die *GUI-DSL* weiter eingesetzt wird und die Layout-Dateien vom Generator verwendet werden.

Die Beschreibung von *GUIs* durch die Zusammensetzung von *GUI-Komponenten* (siehe Anforderung AS1), sowie auch die Wiederverwendung und Erweiterung von *GUI-Komponenten* (siehe Anforderung AS2) wurde umgesetzt und in Kapitel 8 demonstriert.

Dass die *GUI-DSL* um neue *GUI-Komponenten* erweitert werden kann (siehe Anforderung AS6), wurde ebenfalls gezeigt.

Der bei der Demonstration verwendete Quellcode erwies sich als weitaus kürzer, als der generierte Quellcode. In der *GUI-DSL* wurden lediglich 13 Codezeilen benötigt. Der Generator erzeugte daraus ca. 200 Codezeilen. Ob Anforderung AS4 damit umgesetzt ist, ist wiederum eine subjektive Entscheidung. Dabei muss jedoch erwähnt werden, dass der Quellcode, der für die Implementierung des Generators und der Grammatik bei dieser Anforderung nicht berücksichtigt wird.

Die in Kapitel 2 beschriebenen Probleme des *MCF* werden durch die *GUI-DSL* nicht vollständig gelöst. Die wegfallende Orientierung an ein spezifisches GUI-Framework (siehe Problem P2), führt dazu dass die Überführung der GUI-Skripte in andere GUI-Frameworks einfacher erscheint. Das setzt jedoch voraus, dass die Implementierung eines entsprechenden Generators einfacher ist, als die Anpassung eines neuen GUI-Frameworks an Swing, ist diese Eigenschaft ein Vorteil gegenüber dem *MCF*. Der in dieser Arbeit entwickelte Prototyp realisiert das Generieren von Klassen für das *MCF*. Es wird jedoch nicht geprüft, ob die Entwicklung von Generatoren, die Quellcode für andere GUI-Frameworks generieren, mehr oder weniger Aufwand erfordert.

Das Problem, dass die verwendeten GUI-Frameworks nicht aktuell sind (siehe Problem P1), wird durch die *GUI-DSL* alleine nicht gelöst. Entschei-

dungen über die Integration neuer Frameworks müssen von der *data experts GmbH* getroffen werden.

Allerdings ist es möglich durch die Verwendung der *GUI-DSL* und eines spezifischen Generators für das *MCF* den Entwicklern bestimmte Routineaufgaben abzunehmen.

Beispielsweise kann das Pflegen der korrekten Bezeichnungen in den *IP*- und *GUI-Klassen* (siehe Routinearbeit *R1*) vom Generator übernommen werden. Dies wurde mit den Umsetzungen in Kapitel 9 realisiert und kann in den generierten Klassen, die sich auf dem beigelegten Datenträger befinden, nachvollzogen werden.

Über das Erstellung von Klassen, die für die Darstellung von Tabellen benötigt werden (betrifft Routinearbeit *R2*), kann an dieser Stelle keine Auskunft gegeben werden, da im Prototypen diese Klassen nicht generiert werden.

Die dritte genannte Routinearbeit (*R3*) wird den Entwicklern nicht abgenommen, da die Pflege der Attribute der *GUI-Komponenten* weiterhin über die Properties-Dateien erfolgen kann.

Abgesehen von der Umsetzung der Anforderungen sind für den Einsatz der *GUI-DSL* weitere Aspekte in Betracht zu ziehen, die in dieser Arbeit nicht angesprochen wurden.

Beispielsweise ist die Frage, wie die *GUI-DSL* in den Entwicklungsprozess der *data experts GmbH* eingebettet werden kann, offen. Mit *Xtext* ist es zwar möglich einen Editor zu generieren, aber wie dieser in die *Eclipse IDE* dauerhaft eingebunden werden kann, wurde nicht beschrieben.

Wenn eine *GUI* beschrieben wurde, wäre es von Vorteil, wenn diese im Anschluss daran getestet werden kann, bevor es zur Generierung kommt. Bei der Entwicklung eines solchen Test-Konzeptes, ist auch die Entwicklung eines grafischen Editors für die *GUI-DSL* in Betracht zu ziehen. Damit wäre es nicht nötig, dass das zur Entwicklung der *GUI-DSL* verwendete Framework einen Editor mit anpassbaren Validierungen bereitstellt.

Für einen solchen Test wäre auch die Layout-Dateien relevant. Der Aufbau dieser Dateien muss noch festgelegt werden. Entweder die *data experts GmbH* verwendet dafür bestehende Sprachen wie *CSS*, oder Definiert eine neue *DSL*, mit der es ausschließlich möglich ist das Layouts zu beschreiben. Bei der Generierung der Klassen sind Überlegungen über die Art der Trans-

formation (*Transformer Generation* oder *Templated Generation*) anzustellen. In Kapitel 9 wurde festgelegt, dass für den Prototypen *Transformer Generation* verwendet wird. Andererseits wurde auch darauf hingewiesen, dass bei der Generierung der *IP*- und *FP*-Klassen die *Templated Generation* in Betracht gezogen werden sollte, da nur bestimmte Teile des Quellcodes verändert werden müssen.

Zusammenfassend ist zu sagen, dass noch viele Fragen geklärt werden müssen, bevor entschieden werden kann, ob die *GUI-DSL* in der *data experts GmbH* eingesetzt werden kann. Ein erster Schritt wäre es zu prüfen, ob der Generator in der Lage ist, alle anderen GUI-Komponenten, deren Generierung nicht demonstriert wurde, genauso gut erzeugen, wie die, welche bereits im Prototypen verwendet wurden. Darüber hinaus wird die Weiterentwicklung des Generators für das *MCF* und die Implementierung von Generatoren für andere GUI-Frameworks zeigen, ob die *GUI-DSL* ausreichend abstrakt, oder nicht abstrakt genug für *profil c/s* und die verwendeten GUI-Frameworks ist. Das Konzept welches im Zuge dieser Arbeit entstand (siehe Kapitel 5) ist jedoch vielversprechend, wenn es mit dem Konzept des *MCF* verglichen wird. Das beweist vor allem die Umsetzung der allgemeinen Anforderungen. Allerdings ist aufgrund des Entwicklungsstandes an eine kurzfristigen Ablösung des *MCF* durch die *GUI-DSL* nicht zu denken.

Anhang A

Zuwendungsblatt für Web- und Standalone-Client

Zuwendungsblatt 3216 - Sportstätten(1)/2010: 139610040007/130500004 von Hinten, Schorsch Az: 1...

Zuwendungsblatt Hilfe

Anteils-/Festbetragsfinanzierung Mengenfinanzierung

Förderfähige Ausgaben lt. Kostenplan: 290.000,00

Fördergegenstand mit Fördersatz	ff. Ausgaben lt. Amt [EUR]	Finanzierungsart	Berechneter Bew.betrag [EUR]	Tatsächl. Fördersatz [%]	Abzug [EUR]	Zuwendung lt. Amt [EUR]
Erweiterung vereinseigener Sportstätten - 75,00%	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

Neu Löschen

Bemerkungen:

Bemerkungen

Original | Jede65-1 / ES - Mecklenburg/Vorpommern (Amt: 3)

Abbildung 31: Standalone-Client: Zuwendungsblatt (vgl. [dat07])

Abbildung 32: Web-Client: Zuwendungsblatt (vgl. [dat07])

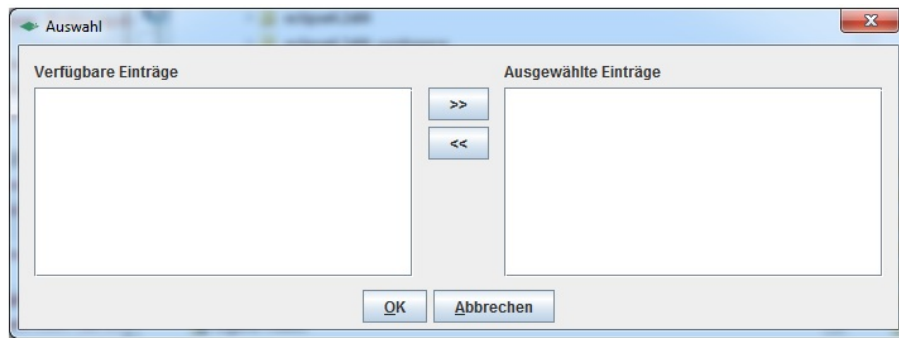


Abbildung 33: Multiselection-Komponenten

Anhang B

Generierte Explorer-GUI

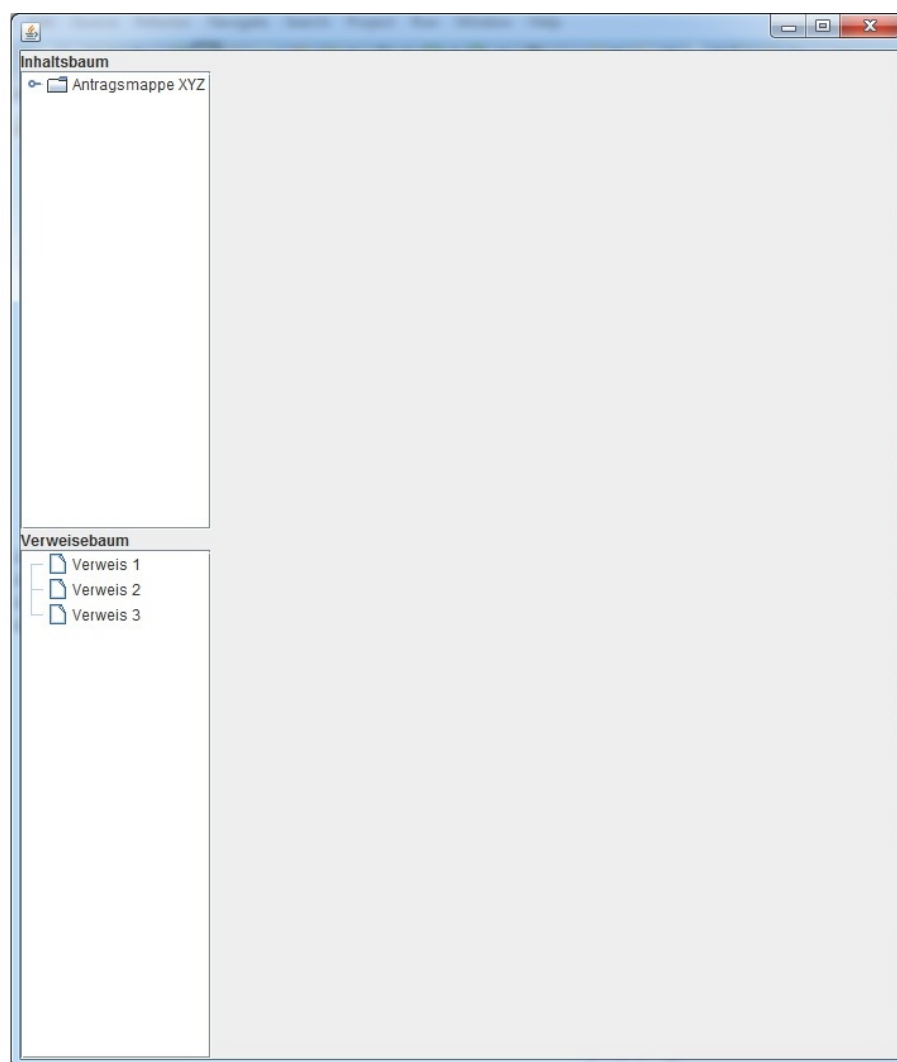


Abbildung 34: Generierte Explorer-GUI

Anhang C

Inhalt des beiliegenden Datenträgers

Grammatik aus der 1. Iteration: Iterationen/1/Grammatik.xtext

Grammatik aus der 2. Iteration: Iterationen/2/Grammatik.xtext

Grammatik aus der 3. Iteration: Iterationen/3/Grammatik.xtext

Glossar

Domäne oder Anwendungsdomäne beschreibt ein abgegrenztes Wissen- oder Interessengebiet (vgl. [VAC⁺08, S.170]).

DSL-Umgebung beinhaltet den Parser, den Lexer und die Verarbeitungslogik (vgl. [Gho11, S.211]).

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

Graphical User Interface ist die Bezeichnung für die Schnittstelle zwischen dem Benutzer und dem Programm (vgl. [DAT]). Die Kurzform *GUI* wird in dieser Arbeit aufgrund des allgemeinen Sprachgebrauchs als feminines Nomen verwendet (die GUI).

GridBagLayout ist ein Layoutmanager innerhalb von Swing, welcher die Komponenten horizontal, vertikal und entlang der Grundlinie anordnet. Dabei müssen die Komponenten nicht die gleiche Größe haben (vgl. [Oraa]).

Inhaltsbaum einer Antragsmappe enthält die Dokumente die in dieser Antragsmappe vorliegen.

InVeKoS ist die Abkürzung für Integriertes Verwaltungs- und Kontrollsystem. Mit einem solchen Systemen wird im allgemeinen sichergestellt, dass die durch den Europäischen Garantiefonds für die Landwirtschaft finanzierten Maßnahmen ordnungsgemäß umgesetzt wurden. Im speziellen bedeutet dies die Absicherung, von Zahlungen, die

korrekte Behandlung von Unregelmäßigkeiten und das wieder Einziehen von zu unrecht gezahlter Beiträge (vgl. [Gen14]).

Multiselection-Komponente stellt in *profil c/s* ein Werkzeug zur Auswahl mehrerer Objekte dar. Die *GUI* einer Multiselection-Komponenten ist in Abbildung 33 in Anhang A dargestellt. Es wird zwischen einem Container, der die Objekten, die zur Auswahl zur Verfügung stehen (linke Seite), enthält und einem Container, der die Objekten, die bereits ausgewählt sind, enthält (rechte Seite) unterschieden. Mit den in der Mitte befindlichen Schaltflächen ist es möglich die Objekte von einem Container in den anderen zu navigieren. In der Abbildung bestehen keine Auswahlmöglichkeiten.

Swing ist ein GUI-Framework für Java Applikationen (vgl. [Orab]).

Top-Down Parser erzeugen den Parse-Tree ausgehend von der Wurzel. Im Gegensatz dazu stehen Bottom-Up Parser, welche den Parse-Tree von den Blätter aus erzeugen (vgl. [Gho11, S.225]).

Traditionelle GUI-Entwicklung beschreibt die GUI-Entwicklung unter Verwendung von traditionellen GUI-Toolkits gearbeitet. Bei diesen Toolkits wird Aufbau der GUI genau beschrieben. Für die Interaktion mit den GUI-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 (vgl. [KB11]).

Turing-Maschine ist ein Automatenmodell welches vom Alan M. Turing 1936 vorgestellt wurde. Die Turing-Maschine abstrahiert die generelle Arbeitsweise heutiger Rechner (vgl. [Hed12, S.145ff]).

Usability beschreibt die Nutzerfreundlichkeit einer Software (vgl. [Dir00, S.10]).

Verweisebaum einer Antragsmappe enthält Verweise auf Dokumente, die mit dieser Antragsmappe in Verbindung stehen.

wingS ist ein Framework für die komponentenorientierte Entwicklung von Webapplikationen (vgl. [Sch07]).

Zielumgebung einer DSL beschreibt, die Infrastruktur, in der die Generierten Artefakte integriert werden können (vgl. [VBK⁺13, S.26]).

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 [...] . Das Ergebnis wird im Zuwendungsblatt dokumentiert, das auch später mit demselben Werkzeug angesehen werden kann“* [dat07].

Zuwendungsblatt ist die grafische Dokumentation der Ergebnisse des Zuwendungs-Berechners innerhalb von *profil c/s* (vgl. [dat07]).

Literaturverzeichnis

- [Aho08] AHO, ALFRED V.: *Compiler: Prinzipien, Techniken und Werkzeuge*. Pearson Studium Informatik. Pearson Education Deutschland, 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.
- [BCK10] BRAUER, JOHANNES, CHRISTOPH CRISEMAN und HARTMUT KRISEMAN: *Eine DSL für Harel-Statecharts mit PetitParser*. Arbeitspapiere der Nordakademie. Nordakademie, 2010.
- [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.
- [DAT] DATACOM BUCHVERLAG GMBH: *GUI (graphical user interface)*. URL: www.itwissen.info/definition/lexikon/graphical-user-interface-GUI-Grafische-Benutzeroberflaeche.html. Zuletzt eingesehen am 29.01.2015.
- [dat07] DATA EXPERTS GMBH: *Detaillkonzept ELER/i-Antragsmappe*, Januar 2007. Zuletzt eingesehen am 20.12.2014.
- [dat14] DATA EXPERTS GMBH: *Förderantrag*. Profil Wiki der data experts GmbH, März 2014. Zuletzt eingesehen am 20.12.2014.

- [Dir00] DIRNBAUER, KURT: *Usability*. Libri Books on Demand, 2000.
- [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.
- [Gen14] GENERALDIREKTION LANDWIRTSCHAFT UND LÄNDLICHE ENTWICKLUNG: *Das Integrierte Verwaltungs- und Kontrollsystem (InVeKoS)*. URL: http://ec.europa.eu/agriculture/direct-support/iacs/index_de.htm, November 2014. Zuletzt eingesehen am 20.12.2014.
- [Gho11] GHOSH, DEBASISH: *DSLs in Action*. Manning Publications Co., 2011.
- [Gun13] GUNDERMANN, NIELS: *Prototypische Implementierung eines JavaFX-Channels zur Integration ins Multichannel-Framework der deg*, 2013. Praxisbericht.
- [Gun14a] GUNDERMANN, NIELS: *Entwicklung einer Grammatik für eine DSL mit xText am Beispiel einer Sprache zur Definition von Pflichtprüfungen in profil c/s*, 2014. Praxisbericht.
- [Gun14b] GUNDERMANN, NIELS: *Prototypische Implementierung eines JavaFX/Web-Channels zur Integration ins Multichannel-Framework der deg*, 2014. Praxisbericht.
- [Hed12] HEDTSTUECK, ULRICH: *Einführung in die Theoretische Informatik*, Band 5. Auflage. Oldenbourg Verlag, 2012.
- [Hof06] HOFER, STEFAN MICHAEL: *Refactoring-Muster der WAM-Modellarchitektur*. Diplomarbeit, FH Oberösterreich, 2006.

- [KB11] KRISHNASWAMI, NEELAKANTAN R. und NICK BENTON: *A Semantic Model for Graphical User Interfaces*. Microsoft Research, September 2011.
- [LW07] LU, XUDONG und JIANCHENG WAN: *Model Driven Development of Complex User Interface*. In: *MoDELS'07 Workshop on Model Driven Development of Advanced User Interfaces*. Shandong University, 2007.
- [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.
- [Oraa] ORACLE: *Class GridBagLayout*. URL: <https://docs.oracle.com/javase/7/docs/api/java/awt/GridBagLayout.html>. Zuletzt eingesehen am 20.12.2014.
- [Orab] ORACLE: *Swing*. URL: <http://docs.oracle.com/javase/8/docs/technotes/guide>. Zuletzt eingesehen am 20.12.2014.
- [Pec14] PECH, VACLAV: *Can MPS be integrated in Eclipse as Eclipse Plugin similar to Xtext?* URL: <http://forum.jetbrains.com/thread/Meta-Programming-System-1033>, Oktober 2014. Zuletzt eingesehen am 12.01.2015.
- [PSV13] PECH, VACLAV, ALEX SHATALIN und MARKUS VÖLTER: *JetBrains MPS as a tool for extending Java*. In: *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, Seiten 165–168. ACM, 2013.
- [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.
- [Roa09] ROAM, DAN: *The Back of the Napkin (Expanded Edition) - Solving Problems and Selling Ideas with Pictures*. Penguin, New York, Expanded Auflage, 2009.

- [Sau03] SAUER, JOACHIM: *Gestaltung von Anwendungssoftware nach dem WAM-Ansatz auf mobilen Geräten*. Diplomarbeit, Fachbereich Informatik, Universität Hamburg, 2003.
- [Sau10] SAUER, JOACHIM: *Architekturzentrierte agile Anwendungsentwicklung in global verteilten Projekten*. Doktorarbeit, Universität Hamburg, 2010.
- [Sch05] SCHMELZER, ROBERT FRANZ: *Realisierung teilautomatisierter Prozesse durch die Kombination von Ablaufsteuerung und unterstützter Kooperation*. Diplomarbeit, FH Oberösterreich, 2005.
- [Sch07] SCHMID, BENJAMIN: *Get your wingS back!* URL: <http://jaxenter.de/artikel/Get-your-wingS-back>, Dezember 2007. Zuletzt eingesehen am 20.12.2014.
- [SKNH05] SUKAVIRIYA, NOI, SANTHOSH KUMARAN, PRABIR NANDI und TERRY HEATH: *Integrate Model-driven UI with Business Transformations: Shifting Focus of Model-driven UI*. IBM T.J. Watson Research Center, Oktober 2005.
- [Ste07] STECHOW, DIRK: *JWAMMC - Das Multichannel-Framework der data-experts gmbh*. Vortrag in der data experts GmbH, Dezember 2007.
- [SV06] STAHL, THOMAS und MARKUS VÖLTER: *Model-Driven Software Development*. John Wiley & Sons Ltd, Februar 2006.
- [The14] THE ECLIPSE FOUNDATION: *Xtext Documentation*, September 2014.
- [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 20.12.2014.
- [VAC⁺08] VOGEL, OLIVER, INGO ARNOLD, ARIF CHUGHTAI, EDMUND IHLER, TIMO KEHRER, UWE MEHLIG und UWE ZDUN: *Software-*

Architektur. Springer Science Business Media (Berlin Heidelberg), 2008.

- [VBK⁺13] VÖLTER, MARKUS, SEBASTIAN BENZ, LENNART KATS, MATS HELANDER, EELCO VISSER und GUIDO WACHSMUTH: *DSL Engineering*. CreateSpace Independent Publishing Platform, 2013.
- [VC09] VÖLTER, MARKUS und LARS CORNELIUSSEN: *Carpe Diem*. Dot-NetPro, 5 2009.
- [Völ11] VÖLTER, MARKUS: *From programming to modeling-and back again*. Software, IEEE, 28(6):20–25, 2011.