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

in der Fachrichtung
Wirtschaftsinformatik

T H E M A

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

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

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Kapitel 1

Motivation

In der heutigen Zeit werden Programme auf vielen unterschiedlichen Geräten (bspw. Desktop, Smartphone, Tablet) ausgeführt. Die Useability ist ein wichtiger Faktor bei der Entwicklung von Anwendungen. Denn *„schlechte Usability führt zu Verwirrung und Miss- bzw. Unverständnis“* [Use12] beim Kunden, wodurch letztendlich Umsatz verloren geht. Die Useability wird hauptsächlich vom User Interface (UI) bestimmt. Folglich ist die Benutzeroberfläche neben der internen Umsetzung ein wichtiger Faktor für den Erfolg einer Anwendung. (vgl. [LW])
Wenn ein Programm auf unterschiedlichen Geräten ausgeführt wird, muss der Entwickler bei der traditionellen UI-Entwicklung mehrere Graphical User Inferfaces (GUIs) manuell implementieren. Folglich werden mehrere GUIs mit unterschiedlichen Toolkits oder Frameworks entworfen. Diese Framworks haben einen starken imperativen Charakter, sind schwer zu erweitern und verhalten sich je nach Plattform unterschiedlich. (vgl. [KB11]) Daraus folgt, dass Entwickler bei dem traditionellen Ansatz das 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. UIs anhand der modellierten Funktionalitäten automatisch erzeugt werden. (vgl. [SKNH05]) Laut Myers et. al. wurden die Darstellungen dieser generierten User Interfaces (UIs) in der Vergangenheit von den Darstellungen traditionell implementierter Benutzerschnittstellen übertroffen. (vgl. [MHP99]) Grund dafür ist, dass die Abstraktionsebene, auf der die Beschreibung der GUIs beim

Model-Driven Ansatz statt findet, 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 UIs (traditionell und Model-Driven) verbunden werden können (kombinierter Ansatz). Somit könnte die genaue Beschreibung der Darstellung mit einer höheren Abstraktion verbunden werden. Dieser Versuch wurde bspw. von Baciková im Jahr 2013 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 der Umsetzung wird sich auf die UIs der Anwendung *profil c/s* bezogen. Profil c/s ist eine JEE -Anwendung die InVeKoS umsetzt und von der deg entwickelt wird.

Kapitel 2

Problembeschreibung und Zielsetzung

2.1 Allgemeine Anforderungen an Benutzeroberflächen von profil c/s

Die erste Anforderung bezieht sich auf die GUI des Clients von profil c/s. Dieser soll sowohl in Web-Browsern (Web-Client) als auch standalone auf einem PC (Standalone-Client) ausgeführt werden können.
Da die Nutzer an einen bestimmten Aufbau der GUIs gewöhnt sind, ist es von Vorteil, wenn beide Clients ein ähnliches UI bieten. Von daher ist die Ähnlichkeit der UIs auf unterschiedlichen Plattformen eine weitere Anforderung.
Um eine effiziente Arbeitsweise zu ermöglichen, 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.
Darauf aufbauend sind weiterhin die Abstraktionsmöglichkeitenvon den verwendeten Frameworks von großer Bedeutung, da durch einen hohen Abstraktionsgrad ein hoher Komplexitätsgrad beherrscht werden kann. (vgl. [Bra03])
Abschließend ist die Ausdruckskraft der Syntax, in der die UIs entwickelt werden, als Kriterium zu nennen. Dies fördert die Lesbarkeit des Quell-Codes und damit letztendlich das Verständnis dessen, sowie die Effizienz

mit der die GUIs entwickelt werden. (vgl. [VBK+13, S.70])
Diese Anforderungen an die UIs haben in der deg unterschiedliche Prioritäten (1 = höchste Priorität, 3 = niedrigste Priorität), die in folgender Tabelle beschrieben werden.

Nr.	Anforderung	Priorität
1.	Bereitstellung für standalone und Web	1
2.	Ähnlicher Aufbau auf unterschiedlichen Plattformen	2
3.	Erweiterbarkeit der verwendeten Frameworks	1
4.	Verwendung abstrahierbarer Frameworks	2
5.	Ausdrucksstarke Syntax	3

Tabelle 2.1: Prioritäten der Anforderungen an die GUI

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

2.2.1 Multichannel-Framework

Die Clients werden in der Programmiersprache Java entwickelt. Für die Realisierung des Standalone-Clients wird in der deg das Swing -Framework verwendet. Für den Web-Client wird auf wingS zurückgegriffen. Um eine Vorstellung des Ist-Zustandes zu vermitteln, sind in Abbildung 2.1 und in Abbildung 2.2 die GUIs eines Zuwendungsblattes und eines Förderantrags für den Web-Client und den Standalone-Client abgebildet.
In diesen UIs ist nur ein bestimmter Teil für die fachlichen Informationen relevant. Dies sind lediglich die Tabelle und 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 Clients gleich aufgebaut. Andere Teile des GUIs haben derzeit einen unterschiedlichen Aufbau, was den unterschiedlichen Frameworks für die Umsetzung von Web- und Standalone-Client geschuldet ist.

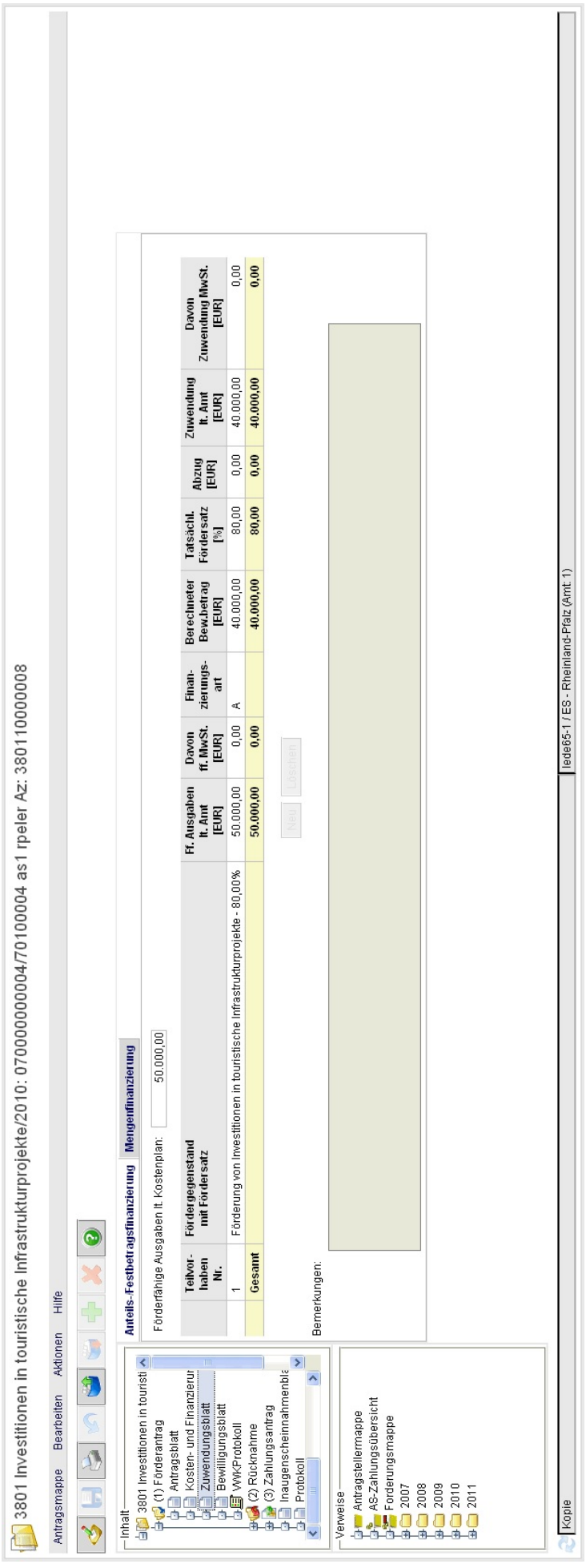

Abbildung 2.1: Web-Client: Zuwendungsblatt (vgl. [deG07])

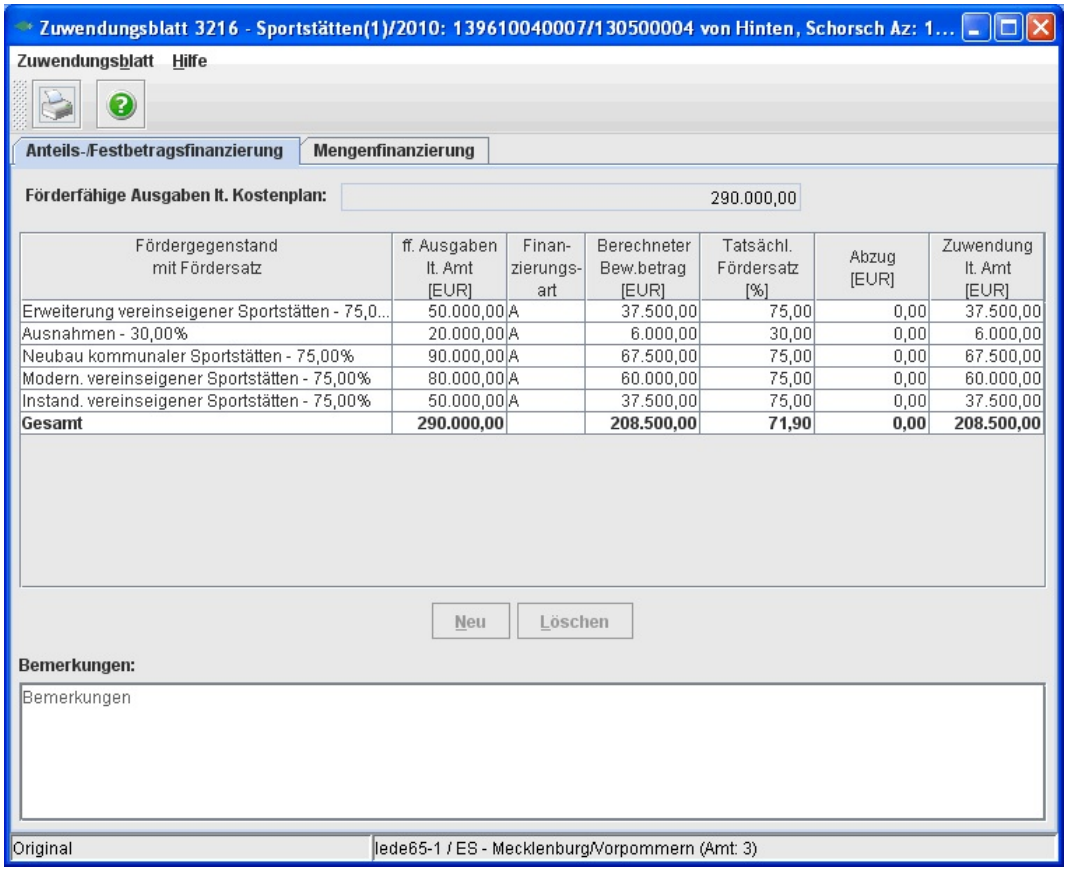

Abbildung 2.2: Standalone-Client: Zuwendungsblatt (vgl. [deG07])

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

Aufgrund von Anforderung Nr. 1 wurden in der Vergangenheit zwei GUIs mit unterschiedlichen Frameworks von der deg entwickelt. Dieses Verfahren erwies sich mit komplexer werdenden GUIs als sehr ineffizient. Daher hat die deg 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 Nr. 2 nachkommt. Die Lösung der deg ist das *Multichannel-Framework* (MCF).

Die Architektur des Multichannel-Frameworks ist Abbildung 2.3 zu entnehmen. Innerhalb des MCF werden die GUIs mittels so genannter *Präsentationsformen* beschrieben.

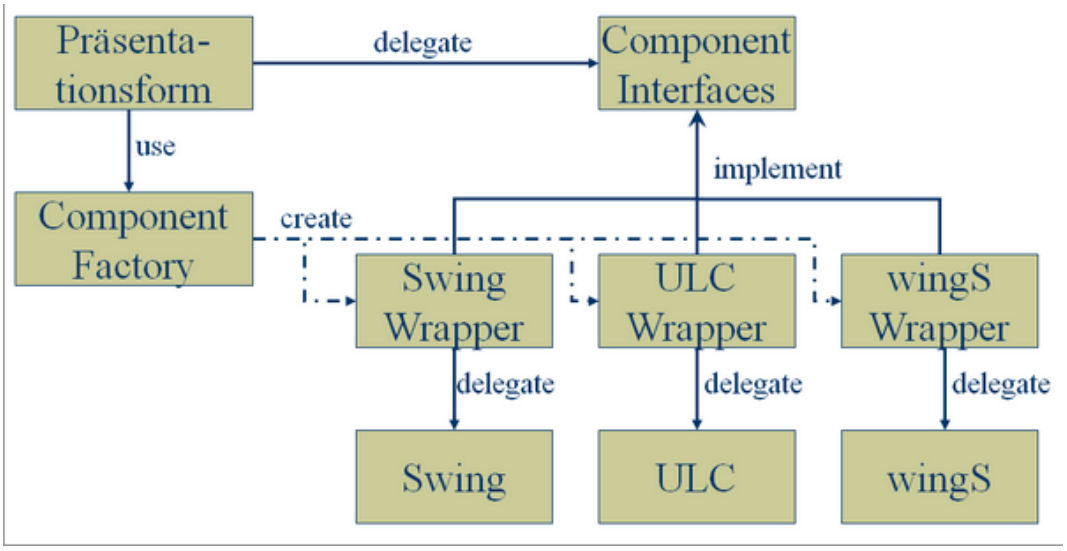

Abbildung 2.3: Architektur des Multichannel-Frameworks (vgl. [Ste07])

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 . Wobei ULC bei der deg nicht mehr im Einsatz ist.) Das *Component-Interface* wird für die Interaktion mit den Komponenten der unterschiedlichen Frameworks benötigt. Mit dem MCF ist die deg in der Lage ihre GUIs für das *Swing* -Framework und für das *wingS* -Framework mit nur einer GUI-Beschreibung zu erzeugen.
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“ [SdS03, S.30]

Die Bezeichnung *WAM* steht für Werkzeug und Material-Ansatz. Es handelt sich dabei um einen Ansatz zur Softwarearchitektur, welcher die anwendungsorienten 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. [SdS03, S.30]) Diese Entwurfsmetapher beschreiben *„[...] Elemente und Konzepte der Anwendung durch bildhafte Vorstellungen von realen Gegenständen [...]“* [SdS03, S.30] . Die grundlegenden Metaphern werden im Folgenden kurz erläutert. Ein *Material* kann nicht direkt vom Nutzer bearbeitet werden. Sie besitzen 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. Bei Werkzeugen werden zwei Arten unterschieden - *monolithische-* und *komplexe Werkzeuge*. (vgl. [Hof06, S.5]) Da in der deg hauptsächlich komplexe Werkzeuge Anwendung finden, werden die monolithischen Werkzeuge hier nicht weiter erläutert. Komplexe Werkzeuge gliedern sich in Oberfläche, Interaktion und Fachlogik auf. Dabei muss 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 Werkzeug-Klasse umschließt diese drei Komponenten (siehe Abbildung 2.4). (vgl. [Hof06, S.5f])

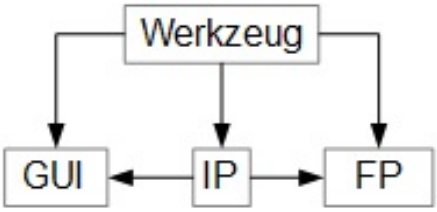

Abbildung 2.4: Architektur eines komplexen Werkzeugs mit Benutzt-Beziehung

Automaten können selbstständig Routinearbeiten erledigen. (vgl. [SdS03, 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 zwei-

tens Räume der Arbeitsumgebung, die für alle zugänglich sind und über die somit die Zusammenarbeit statt findet. (vgl. [SdS03, 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. [SdS03, S.30f])
Durch fachliche Services ist mit dem WAM-Ansatz Multi-Channeling möglich. (vgl. [SdS03, S.42]) Das Multi-Channeling des WAM-Ansatzes setzt dabei auf einer anderen Ebene an als das MCF. Das Multi-Channeling des WAM-Ansatzes basiert darauf, dass *[...] Funktionalität unabhängig von der konkreten Handhabung, Präsentation und Technik bereitgestellt wird.* [SdS03, S.42]
Das MCF hingegen basiert darauf, dass eine Präsentation unabhängig von der Technik bereitgestellt werden kann. Es setzt demnach an einer höheren Ebene an, als das Multi-Channeling des WAM-Ansatzes. Die Mechanismen, die das Multi-Channeling im WAM-Ansatz bietet, werden jedoch auch im MCF verwendet.

2.3 Probleme des Multichannel-Frameworks

Bezogen auf die Anforderungen wurden Anforderungen werden Nr. 3 und 5 nicht durch das MCF umgesetzt. (Fakt ist, dass jede Sprache eine gewisse Ausdruckskraft hat. Da in dieser Arbeit versucht wird die UI-Entwicklung mittels einer ausdrucksstarken DSL, die sich auf die Domäne von profil c/s bezieht, umzusetzen, ist die Ausdruckskraft herkömmlicher Programmiersprachen als unzureichend anzusehen).
Ein weiteres Problem bezieht sich auf die integrierten Framworks (Swing und wingS). Beide Frameworks sind veraltet und werden nicht mehr gewartet. (Analysen dafür wurden in [Gun13] und [Gun14b] durchgeführt.) Um auch in der Zukunft den Anforderungen der Kunden nachkommen zu können müssten beide Frameworks von den Entwicklern der deg selbst weiterentwickelt werden. Eine andere Möglichkeit wäre es, 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 2.3). In der Praxis wurde die Einfachheit einer solchen Integration jedoch widerlegt. Ein Problem, welches bei der Integration neuer Frameworks aufkommt, ist, dass sich das MCF sehr stark an Swing orientiert und die GUIs vor allem vom GridBagLayout stark beeinflusst werden. Ein solche Layoutmanager steht nicht in allen UI-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 inaktuell

P2 Starke Orientierung an Swing

P3 Oben genannte Anforderungen werden nicht komplett umgesetzt

Dazu kommen lästige Routinearbeiten, die in der deg bei der Entwicklung von UIs getätigt werden müssen.

R1 Vergebene Bezeichnungen für UI-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 UI-Komponenten werden bei der deg in Properties-Dateien festgehalten. Das Erstellen und Pflegen wird als Fehleranfällig betrachtet.

2.4 Zielsetzung

Das langfristige Ziel der deg bzgl. der UIs 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 mindert.
In dieser Arbeit wird ein Ansatz untersucht, bei dem es möglich ist, die oben genannten Anforderungen vollständig umzusetzen. Kern des Ansatzes ist

eine Domänenspezifische Sprache (DSL), mit deren Hilfe die UIs beschrieben werden sollen. Eine DSLs könnte so konzipiert werden, dass sie ausreichend abstrakt und erweiterbar ist und bzgl. der Ausdruckskraft das MCF übertrifft.
Die genaue Lösungsidee mittels DSL, welche in dieser Arbeit verfolgt wird, ist in Kapitel 4 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, nichtleere Menge von Zeichen (auch Symbole oder Buchstaben genannt).“* [Hed12, S.6] Zur Verdeutlichung der Definition einer Sprache sei V ein Alphabet und $k \in N$ (N ist die Menge der natürlichen Zahlen einschließig der Null). (vgl. [Hed12, S.6]) *„Eine endliche Folge $(x_1, \ldots, x_k)$ mit $x_i \in V (i = 1, \ldots, k)$ heißt Wort über V der Länge k“* [Hed12, S.6] .

Programmiersprachen werden dazu verwendet, um mit 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 bezüglich 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 Typ-Systems, die ein Programm befolgen muss. (vgl. [VBK+13, S.26])

Ausführungssemantik

Die Ausführungssemantik ist abhängig vom Compiler. Sie beschreibt wie ein Programm zu 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 Abstarktion 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 Abstraktionebene angepassen sein. Eine DSL ist demnach in ihren Ausdrucksmöglichkeiten eingeschränkt. Je stärker diese Einschränkung ist, desto besser ist die Unterstützung der Domäne sowie die Ausdruckskraft der DSL. (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 Personen, die keine Programmierer sind (bspw. 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 verweise ich auf den Praxisbericht [Gun14a]. Grammatiken können in einer Hierachrie dargestellt werden (*Chomsky-Hierarchie*). (vgl. [Hed12, S.32f]) Bei Programmiersprachen handelt es sich um *kontextfreie Sprachen*. Diese sind entscheibar und 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 sog. *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]) Allgemein 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 verantworlich 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 Syntax-Baums (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 dargestellt werden soll. Daraus lässt sich auch das semantische Model 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

In Anlehnung an Martin Fowler ist ein Generator der Teil der DSL Umgebung, welcher für das Erzeugen von Quellcode für die Zielumgebung zuständig ist. (vgl. [FP11, S.121]) Bei der Generierung von Code wird zwischen zwei Verfahren unterschieden.

Transformer Generation

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

Templated Generation

Bei der Templated Generation wird eine Vorlage benötigt. In dieser Vorlage befinden sich Platzhalter. Diese dienen dazu, dass vom Generator erzeugter 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 realtionalen 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. [gho11]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 UI-Entwicklung erwähnt. Dieser Ansatz versucht den technischen Lösungen der IT-Industrie einen gewissen Grad an Agilität zu verleihen. (vgl. [SKNH05]) Das ist damit verbunden, dass die Produktion von Softwareprodukten schneller und besser vonstatten geht und mit weniger Kosten verbunden ist. (vgl. [DM14, S.71])
Erreicht wird dies, indem die Modelle formaler, strenger, vollständiger und konsistenter beschrieben werden. (vgl. [VBK+13, S.31]) Die Kernidee ist, dass die Modelle Quellcode oder Funktionalitäten beschreiben und diese in der Evolution der Software immer wiederverwendet werden können. (vgl. [DM14, S.72]) Somit wird 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 Entwicklen 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 Concers)

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

(vgl. [mds06, S.13f])

Die Modelle sind somit nicht länger nur zur Dokumentation geeignet, sondern sind ein Teil der Software. (vgl. [mds06, S.14f]) Dabei sind sie auf ein bestimmtes Domänenproblem angepasst. Die Beschreibung dieser Modelle kann bspw. über eine DSL erfolgen. (vgl. [mds06, S.15]) In Abbildung 3.3 ist die Idee des MDSD schematisch dargestellt.

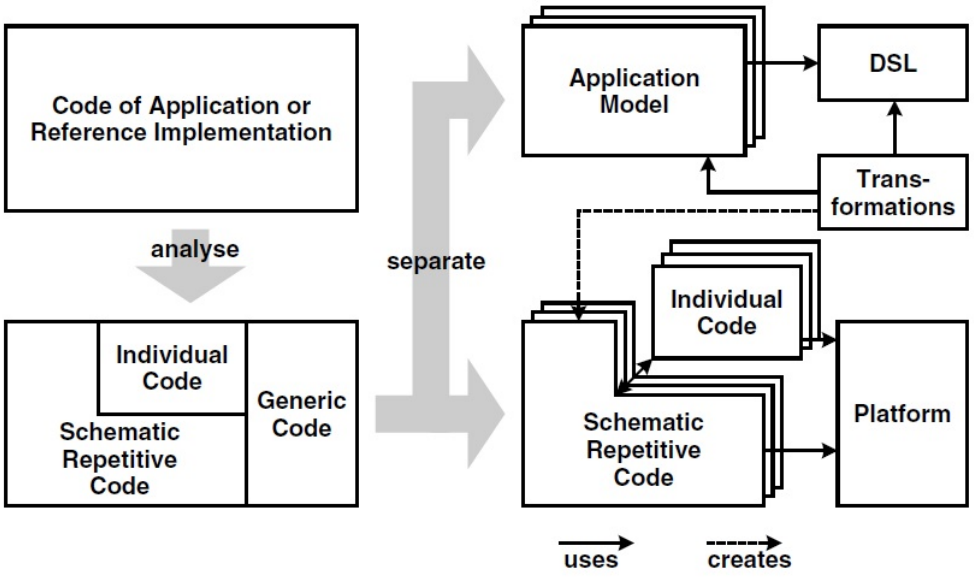

Abbildung 3.1: Die grundlegenden Ideen hinter dem MDSD (vgl. [mds06, S.15])

3.4 Abgrenzung zu GPL

Wie zu Beginn dieses Kapitels 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 bestimm-

te Domäne zugeschnitten 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 Domänen-Fokus als ein Schlüsselelement der Definition einer DSL. (vgl. [FP11, S.27f])

3.5 Vor- und Nachteile von DSL gegenüber GPL

Vorteile

- **Ausdruckskraft**
  Laut Ghosh sollten DSLs so umgesetzt werden, dass sie präsize sind. Diese Präzision bedingt, dass eine DSL einfach zu verstehen ist. Sie sollte demnach den von Dan Roam beschriebenen Prosess des visuellen Denkens (sehen - betrachten - verstehen - zeigen) (vgl. [Roa09]) so schnell wie möglich abzuarbeiten.
  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 ist wächst das Verständis für das, was entwickelt werden soll, da die semantische Lücke zwischen Programm und Problem kleiner wird. (vgl. [gho11, S.20], [BCK08]) Außerdem bleibt die Komplexität durch eine höhere Absatrktionsebene beherrschbar. (vgl. [BCK08])

- **Höhere Qualität**
  Bei der Entwicklung einer DSL werden Sprachkonstrukte und Freiheitsgrade der Sprache festgelegt. Richtig konzipiert, schränken sie den Entwickler beim Umgang mit dieser DSL so ein, dass die Möglichkeit doppelten Code zu schreiben oder doppelte Arbeit durchzuführen kaum noch besteht. Zusätzlich wird die Anzahl von Fehlern verringert. (vgl. [VBK+13, S.40f]) Auch durch die starke Abstraktion einer DSL wird die Wiederverwendung gefördert, was ebenso zu qualitativ höherwertigen Quellcode führt. [gho11]21

- **Verbesserte Produktivität bei der Entwicklung der Software**
  Durch die Ausdruckskraft und die Abstraktion der Sprache muss i.d.R.

auch weniger DSL-Code für die Implementierung eines Programms geschrieben werden, als wenn dieses Programm mit einer GPL implementiert wird. Wobei man mit einem entsprechenden Framework für GPLs ähnliches erreichen könnte. (vgl. [VBK+13, S.40])
Die stärkere Ausdruckskraft führt zu einer bessere Lesbarkeit von DSL-Code im Verlgiech zu GPL-Code, wordurch DSL-Code einfacher zu verstehen ist. Dadurch ist es auch einfacher Fehler in diesem Code zu finden, sowie Apassungen an dem System vorzunehmen. Bei einer GPL werden diese Vorteile durch Dokumentationen, ausdrucksvolle Variablenbezeichnungen und festgeleten Konventionen angestrebt. (vgl. [FP11, S.33]) Allerdings ist der Entwickler zur Einhaltung diese 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 Domänen-Experten und Kunden**
  Aufgrund domänenspezifischer- und präziser Ausdrücke, die in der Sprache verwendet werden, sind die Domänen-Experten bzw. die Kunden vertrauter mit der Implementiertung, als wenn für die Umsetzung eine GPL verwendet werden würde. (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. beschreiben sogar den Einsatz 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 De-

finition 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 verwendet. (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, sondern 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 Analsyen einfacher umzusetzten sind und Fehlermeldungen verständlicher gestaltet werden können, indem die Terminologie der Domäne verwendet wird. Dadruch und durch die vereinfachte Kommunikation mit den Domänen-Experten 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 Abstarktionsniveau ereicht, welches an die Domäne angepasst ist. Dadruch kann die Beschreibung der Modelle von den genutzten Technologien weitgehend entkoppelt werden. (vgl. [VBK+13, S.41])

- **Skalierbarkeit des Entwicklungsprozesses**
  Die Integration von neuen Mitarbeitern in ein Entwicklerteam fordert immer eine gewisse Einarbeitungszeit. Dieser Zeitraum kann durch die Nutzung einer DSL verküzt 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 unterschiedlicher Erfahrungstand 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 Anforderung der Kunden zu implementieren. (vgl. [gho11, S.21]) Das führt im ersten Moment zu einer effizienteren Arbeitsweise, als wenn sich jeder Entwickler mit allem auskennen muss. Markus Völter hingegen sieht die Teilung der Programmieraufgaben als Gefahr bzw. Nachteil. (vgl. [VBK+13, S.44])

Nachteile

- **Großes Know-How gefordert**
  Bevor die Vorteile einer DSL genutzt werden können, muss die DSL 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 geboten werden, wenn die DSL ausreichend gut konzipiert ist. Dazu muss einerseits der richtige Abstraktionsgrad gefunden werden und andererseits die Sprache so einfach wie möglich gehalten werden. Für beide Aufgaben werden Entwickler benötigt, die viel Erfahrung mit Sprach-Design 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 voher 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 DSL hingegen haben oft eine weitaus kompaktere Domäne, als eine technische DSL. Daher ergeben sich die Möglichkeiten zur Wiederverwendung erst zu einem späteren Zeitpunkt und können nur schwer von der im Vorfeld durchgeführten Analysen wahrgenommen werden. (vgl. [VBK+13, S.43])
  Weiterhin ist in der Phase, in der die DSL entwickelt wird, keine große

positive Änderungen bzgl. 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])

- **Investitions-Gefängnis**

  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 dadruch unerwünschte Nebeneffekte auftreten können. Somit wäre das Unternehmen wiederum zu Investitionen gezwungen, um die Anforderungen umzusetzen. Von daher der verwendete Begriff *Investitions-Gefängnis*. (vgl. [VBK+13, S.45])

- **Kakophonie**

  Eine DSL abstrahiert von Domänen-Model. (vgl. [gho11, S.22]) Je besser diese Absatrktion 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 führt dazu, dass die Entwickler öfter umdenken, als wenn sie fortwährend mit einer Sprache arbeiten würden. Das 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. Dadruch ist es schwer neue Technologien von anderen Unternehmen in den Bereichen, wo vermehrt DSLs eingesetzt werden, zu integrieren. Denn diese Technologien werden kaum mit den eigenen DSLs kompatibel sein. Außerdem ist es kaum möglich von neuen Mitarbeitern in diesem Bereichen zu provitieren, da sie sich höchstwahrscheinlich nicht einmal diese DSLs und ihren Zweck kennen werden. [FP11]38

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

- **Engstirnigkeit 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 bestimmen Maß bereits festgelegt. Ein Problem tritt auf, wenn im Nachhinein etwas mit der Sprache beschrieben werden soll, dass nicht zu dieser Abstraktionebene passt.

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

- **Kulturelle Herausforderungen**

  Die genannten Nachteile den Einsatzes von DSLs führen zu Äußerungen wie *Die Entwicklung von Sprachen ist kompliziert*, *Domänen-Experten sind keine Programmieren* oder *Ich möchte nicht schon wieder eine neue Sprache lernen* (*Yet-Another-Language-To-Learn Syndrom* (vgl. [gho11, S.22])).

  Solche kulturellen Probleme i.d.R. dann, wenn etwas neues eingeführt

werden soll. [VBK+13]45 Die Mitarbeiten müssen demnach entsprechend geschult und motiviert werden.

- **Unvollständige DSLs**
  Wenn ein Unternehmen viel Erfahrung bei der Entwicklung von DSLs aufgebaut hat und die Entwicklung durch entsprechende Tools vereinfacht wird, besteht die Gefahr, dass neue DSLs zu frühzeitig entwickelt werden. Damit ist gemeint, dass die Überlegungen über die Notwendigkeit einer neuen DSL nicht ausreichend durchgeführt werden. Durch die einfache Entwicklung scheint es weniger aufwendig eine neue DSL zu entwickeln, als nach bestehenden Ansätzen für das gleiche Problem zu suchen. (vgl. [VBK+13, S.44f]) Der Gedanke daran, dass sich die Investition in die Entwicklung einer DSL zu einem späteren Zeitpunkt amortisieren wird, unterstützt dies. (vgl. [FP11, S.38]) Dadurch entstehen immer mehr DSLs, die auf gleichen Problemen basieren, aber untereinander inkompatibel sind. Außerdem führt der Fakt, dass die Entwicklung einer DSL zum Verstehen der Domäne sehr hilfreich ist, dazu, dass eine DSL nur aus diesem Grund entwickelt wird. (vgl. [FP11, S.38]) Das wiederum führt dazu, dass mehrere halb-fertige DSLs existieren. Markus Völter et al. nennen dieses Phanomän 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 letzendliche Gewinn 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 übernehmen dabei das Typ-System der GPL. (vgl. [VBK+13, S.50]) In Bezug auf die Ziele aus Kapitel 3.3 können einige dieser mit Application Programming Interfaces (API) erreichen werden. In vielen Fällen ist eine DSL nicht mehr als ein API. Martin Fowler sieht den größten Unterschied zwischen API und DSL darin, dass eine DSL neben einem ab-

strahierten 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. [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 Code-Fragment, welches zu einem späteren Zeitpunkt ausgewertet werden soll, als es gelesen wurde, wird in einem Parse-Tree hinterlegt. Dieser Parse-Tree wird noch vor der Ausführung modifiziert. Um diese Implementierungstechnik nutzen zu können, muss eine Umgebung vorliegen in der es möglich ist ein Code-Fragment in einen Parse-Tree umzuformen und diesen zu bearbeiten. Diese Möglichkeit existiert nur in wenigen Sprachen. Martin Fowler et al. geben hierzu nur die Beispiele C#, ParseTree (Ruby) und Lisp. (vgl. [FP11, S.45f])
Anders als Lisp bieten die anderen Beispiele die Möglichkeit über den Parse-Tree zu iterieren. Bei Lisp-Code handelt es sich schon um einen Parse-Tree von verschachtelten Listen. Bei der Iteration über den Parse-Tree ist aufgrund der Performance darauf zu achten, dass möglichst nur die notwendigen Teile des Baumes beachtet werden. (vgl. [FP11, S.46])
Konstrukte, die in der Wirtsprache geschrieben wurden und nicht verändert werden sollen, spielen bei der Parse-Tree Manipulation keine Rolle, um das semantische Modell zu erzeugen. (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 hingenen ist es möglich Methoden bereitzustellen,

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

Listing 3.1: Beispiel: Fluent Interface

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

(vgl. [FP11, S.68])

Annotationen

Annotationen sind ein Teil der Informationen über ein Programmelement, wie Methoden oder Variablen. Diese Informationen können zur Laufzeit oder zur Übersetzungzeit (wenn die Umgebung die Möglichkeit dazu bietet) manipuliert werden. (vgl. [FP11, S.445])
Bevor eine Annotation verarbeitet werden kann, muss sie definiert werden. Die Definition von Annotationen variiert bei unterschiedlichen Sprachen. (vgl. [FP11, S.446]) Die Verarbeitung von Annotationen findet normalerweise zu frei bestimmten Zeitpunkten statt. Zum Zeitpunkt der Übersetzung, zum Zeitpunkt des Ladens des Programms oder zum Zeitpunkt der Ausführung des Programms. (vgl. [FP11, S.447]) Verarbeitungen während der Laufzeit beeinflussen i.d.R. das Verhalten von Objekten. Beim Laden

des Programms werden meist Validierungs-Annotationen verwendet. Solche Annotationen werden bspw. verwendet, um Mapping für Datenbanken auszulesen. Somit wird die Definition von Elementen von der Verarbeitung getrennt, was zu einem übersichtlichen und lesbaren Code 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 Typsytem 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 die lexikalische Analyse werden oft vorhandene Infrastrukturen genutzt. (vgl. [gho11, S.19])

3.7.1 Implementierungstechniken

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

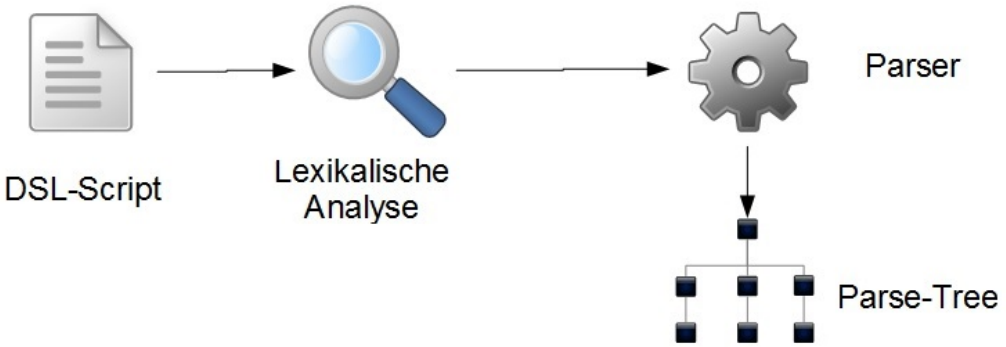

Abbildung 3.2: Parsen allgemein

Parser Generator

Bei der Generierung von Parsern muss der Parser nicht manuell implementiert werden. Diese Aufgabe wird an den Generator delegiert. Damit dies möglich ist, müssen zwei Artefakte definiert werden. Erstens muss eine Grammatik in der EBNF beschrieben werden. Zweitens werden bestimmte Aktionen benötigt, die bei der Bestätigung bestimmter Grammatik-Regeln ausgeführt werden sollen (Validierungs-Regeln). (vgl. [gho11, S.218]) Wird der Parser Generator 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 Parser Generatoren mit Code-Generierung, wodruch der Build-Prozess komplexer wird. (vgl. [FP11, S.272]) Vorteile dieser Technik im Vergleich dazu, dass der Parser manuell entwickelt wird, sind die folgenden.

- Möglichkeit des programmierens auf einem höheren Abstraktionsniveau (vgl. [gho11, S.218])
- Weniger Code zum Implementieren des Parsers benötigt (vgl. [gho11, S.218])
- Möglichkeit des Generieren eines Parser 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 Parsing-Regeln für die nonterminalen Symbole der Grammatik. [FP11]245 Die Funktionen geben dabei einen Boolean-Wert zurück, der Auskunft darüber gibt, ob die Symbole aus dem DSL-Skript mit den Symbolen übereinstimmen, die laut Grammatik erwartet werden. (vgl. [FP11, S.246]) Tabelle 3.1 enthält die Implementierungsmöglichkeiten von einfachen Grammatikregeln.

Grammatik-Regel	Implementierung

A | B	`if(A())`
	`  then true`
	`  else if(B())`
	`  then true`
	`  else false`

A B	`if(A())`
	`  then if(B())`
	`  then true`
	`  else false`
	`  else false`

A?	`A();` `true`
A*	`while(A());` `true`

A+	`if(A())`
	`  then while(A());`
	`  else false`

Tabelle 3.1: 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. Das 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 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 fehlerhaft verarbeitet wurde, kann der Rest den Input-Streams an einen anderen Parser übergeben werden. Somit ist es möglich Parser beliebig zu verketten. (vgl. [gho11, S.242]) Die Elemente, die verkettet werden können werden *Parser-Kombinatoren* genannt. Abbildung 3.3 stellt schematisch diese Funktionsweise dar.

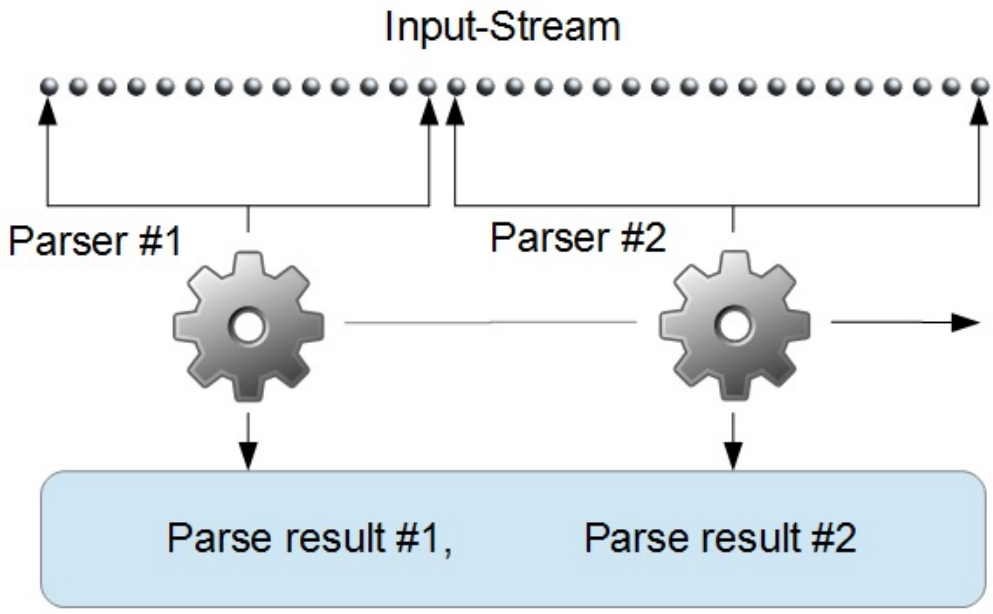

Abbildung 3.3: Funktionsweise von Parser-Kombinatoren (in Anlehnung an [gho11, S.243])

Bezogen darauf, dass ein Parser aus Funktionen besteht, sind diese Parser-Kombinatoren Funktionen erster Ordnung, die unterschiedlich kombiniert werden können. (vgl. [gho11, S.243], [FP11, S.256]) Durch diese Kombination wird eine Struktur gebildet, welche das semantische Model repräsentiert. (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 Grammatik-bestimmender Komponenten auch die Lesbarkeit der Grammatik gefördert, was bei einem RD-Parser ein großer Nachteil war. Daher bezeichnen Fowler et al. Parser-Kombinatoren auch als Mittelweg zwischen RD-Parsern und Parser Generatoren. [FP11]261

3.8 Nicht-Textuelle DSL

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

Kapitel 4

Entwicklung einer Lösungsidee

4.1 Allgemeine Beschreibung der Lösungsidee

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

4.2 Architektur

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

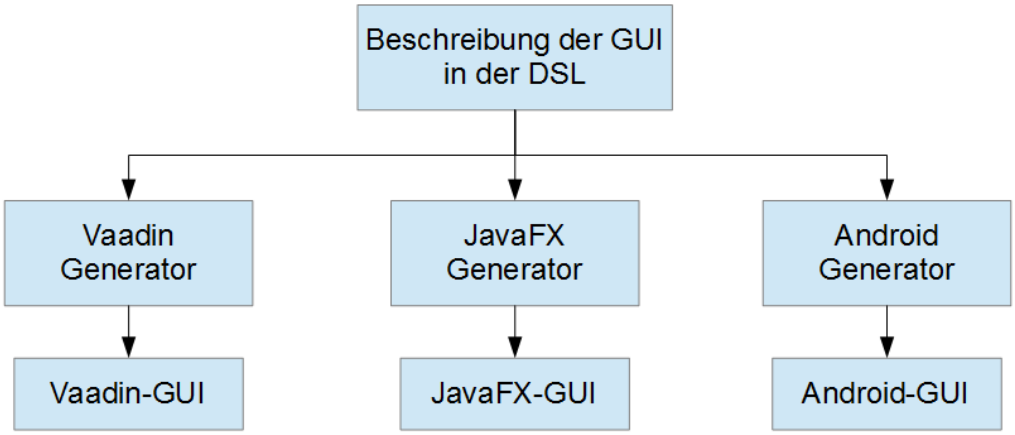

Abbildung 4.1: DSL-Ansatz für gleich GUIs auf unterschiedlichen Plattformen

4.3 Vorteile gegenüber dem Multichannel-Framework

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

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

Darüber hinaus werden dem Entwickler durch die Codegenerierung die in Kaptiel 2.3 fehleranfällige Routinearbeit abgenommen.

Kapitel 5

Anforderung an die GUI-DSL

Die allgemeinen Anfroderung an die GUI wurden in Kapitel 2.1 erläutert. Die folgenden Anforderungen beziehen sich auf die Aspekte der GUIs die beschrieben werden müssen. Ein UI ermöglicht die Interaktion mit einem Programm mit Hilfe unterschiedlicher UI-Komponenten (vgl. [Gal07, S.4]). Mithilfe dieser Komponenten werden Informationen dargestellt, oder Eingaben vom Nutzer getätigt. Um die Zusammensetzung dieser Komponenten zu beschreiben gibt es zwei Ansätze.
Beim ersten Ansatz wird die GUI durch fachliche Modelle beschrieben. (Motivation für MDSD [SKNH05]) Das bedeutet, dass in der Beschreibung der GUI keine UI-Komponenten verwendet werden, wie es in allgemein bekannten UI-Frameworks (JavaFX, Swing) der Fall ist. In der deg soll den Entwicklern weiterhin die Möglichkeit gegeben werden, die UIs selbst zu entwerfen. Ein Grund dafür ist, dass es ein zu großer Aufwand wäre alle Module von profil c/s auf MDSD umzustellen und die GUIs generieren zu lassen. Von daher ist dieser Ansatz vorerst nicht umsetzbar.
Die Komplexität des zweiten Ansatzes schein weitaus geringer zu sein. Dabei werden weiterhin UI-Komponenten in der GUI-Beschreibung verwendet. Die Entwickler haben somit die Möglichkeit die UIs in einem gewissen Grad anzupassen. Die GUI von profil c/s wird dabei als Domäne betrachtet und nicht die fachlichen Hintergründe. Durch die Festlegung dieser Domäne lässt sich bei der Beschreibung von GUIs eine höhere Absatrktionsebene nutzen, als es bei den verwendeten UI-Frameworks der Fall ist.
Der Entwickler soll somit in der Lage sein ein UI relatic frei zu gestalten.

Außerdem sollte er dadurch die Möglichkeit haben bestimmte Konzepte einfach wieder zu verwenden. Für die GUI-DSL wird bzgl. der Anfroderungen an die Komponenten zwischen drei Kategorien unterschieden.
Die erste Kategorie sind umfasst *Basiskomponenten* . Dabei handelt es sich um UI-Komponenten, deren Funktionen in unterschiedlichen UI-Frameworks ähnlich sind und in unterschiedlichen Anwendungen eingesetzt werden könne. Das bedeutet, dass sie nicht als domänenspezifisch angesehen werden können. Beispiele hierfür sind UI-Komponenten wie der *Button* oder das *Label*.
Die zweite Kategorie umfasst *komplexe Komponenten*. Diese zeichnen sich dadurch aus, dass sie domänenspezifisch sind und speziell für profil c/s entwickelt wurden. Ein Beispiel für eine komplexe Komponenten ist die Multiselection-Komponente . Die komplexen Komponentenmüssen für jedes verwendete UI-Framework implementiert werden. Damit wird verhindert, dass die Entwickler, die bzgl. der GUI nur mit der DSL arbeiten, eigene komplexe Komponenten entwerfen, deren Wiederverwendungsgrad nierdirger 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 Komponentensowohl zur Entwicklungszeit, als auch zur Laufzeit vorhanden sein müssen, ist ein Nachteil dieses Konzeptes.
Die dritte Kategorie umfasst *Layout Komponenten*. Dabei handelt es sich um strukturgebende Komponenten. In anderen UI-Frameworks sind dies bspw. *Panel*, *Div* oder *Pane*. In der GUI-DSL müssen auch solche Komponenten verfügbar sein. Dabei ist besonders auf die Ausdruckskraft der für die Beschreibung dieser Komponenten verwendeten Bezeichnungen zu achten. Grund dafür ist, das sich bspw. auf dem Desktop ein Fenster als oberste Layout Komponente festlegen lässt. In einem Web-Browser ist der Begriff *Fenster* als oberste Layout Komponenten jedoch nicht geläuft. Die Komponente, die in einem Browser dem Fester auf dem Desktop am nähsten kommt, ist meiner Meinung nach das Tab.
Die Attribute, die für die einzelnen Komponenten beschrieben werden müssen, werden in Kaptiel 8 genauer analysiert.
Bezüglich des Layouts ist neben den Layout Komponenten eine weitere An-

forderung zu nennen. Hierzu muss erwähnt werden, dass in der traditionellen UI-Entiwcklung GUIs mit Hilfe von Layout-Containern strukturiert werden. In der Vergangenheit hat sich gezeigt, dass die strukturierung über ein spezifisches Layout zu einer Orientierung an ein bestimmtes Framework führt (Beispiel: MCF orientiert sich an Swing). Das ist ein Problem, da bestimmte Layouts im Web oder auf mobilen Plattformen nicht genauso dargestellt werden können, wie auf einer Desktop-Anwendung. Von daher ist das Layout in der GUI-Beschreibung so zu beschreiben, dass es auf allem Plattformen gleichermaßen gut dargestellt werden kann.
Darüber hinaus ist es für die Effizens in deg der wichtig, dass mit dem neuen Ansatz weniger Codezeilen (LOC) geschrieben werden müssen als mit dem alten Ansatz.
Weiterhin ist für ein effizientes Arbeiten auch die Bereitstellung eines Editors für die DSL von großer Bedeutung. Dieser Editor soll nach Möglichkeit auch Validierungen durchführen können und Code-Completion anbieten. Eine Integration dieses Editors in die von der deg verwendete Entwicklungsumgebung (Eclipse) wäre wünschenswert.
Bezüglich der Anforderungen an die Komponenten ist abschließend zu sagen, dass bei den trivialen und den komplexen UI-Komponenten die Möglichkeit bestehen muss Interaktionen festzulegen. Außerdem muss die GUI-DSL um weitere triviale und komplexe Komponenten erweiterbar sein.
Zusammenfassen bestehen die Sprache folgenden Anforderungen

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

Für die Infrastruktur der DSL bestehend die folgenden Anforderungen.

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

Kapitel 6

Evaluation des Frameworks zur Entwicklung der DSL

6.1 Vorstellung ausgewählter Frameworks

Zur Umsetzung der DSL und der Generatoren wird ein Framework benötigt, welches die dafür notwendige 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 Patit-Parser einfach Grammatiken zusammenzustellen, zu transformieren oder zu erweitern, sowie Teile dieser dynamisch wiederzuverwenden. Alles geschieht auf der Basis von Pharo Smalltalk, womit das Framework ursprünglich implementiert wurde. Es existieren auch Versionen des Frameworks für Java[1], Dart[2] und PHP[3]. 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 vorgenommen werden, wodurch der Par-

[1] https://github.com/petitparser/java-petitparser
[2] https://github.com/petitparser/dart-petitparser
[3] https://github.com/mindplay-dk/petitparserphp

ser einem Skript ähnelt. Alternativ können die zu kombinierenden Parser auch in Methoden von Unterklassen des PetitParsers implementiert werden. (vgl. [bra10, S.6]) Das fördert die Lesbarkeit, Überschichtlichkeit und schließlich die Wartbarkeit des Codes.
Tool Support ist für dieses Framework gewährleistet. Mithilfe dessen können Produktionen editiert und grafisch abgebeildet werden. Weiterhin können Zufallsbeispiele für ausgewählte Produktionen generiert werden, um so 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 der externe, textuelle DSLs entwickelt werden können. Die Grammatiken für den Parser-Generator werden in der EBNF definiert. Durch die Intergration in Eclipse kann der Eclipse-Editor für sämtliche Artefakte der Infrastruktur von Xtext verwendet werden. Aus der Grammatik wir der Parser sowie ein Model, mit dessen Hilfe der Generator implementiert werden kann, generiert. Die Klassen für Validierungs-Reglen und den Generator werden ebenfalls vom Tool erzeugt. Diese müssen im Anschluss daran vom Nutzer entsprechend erweitert werden. (vgl. [114]) Zur Editierung der entsprechenden Dateien wird eine eigene Syntax verwendet, die meiner Meinung stark an die Java-Syntax erinnert. Wenn der Parser generiert wurde, ist Xtext in der Lage einen in Eclipse integrierten Editor zu erzeugen. (vgl. [ML09, 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 dem 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 gestellt[4].
Der Generator kann die Konstrukte der neuen Sprach in bestimmte Basis-Sprachen überführen. Diese Basis-Sprachen sind *C*, *Java* oder *XML*. Auch die Transformation in einfachen Text ist gewährleistet. (vgl. [Vol11])
Des Weiteren wird ein Editor für die Arbeit mit der DSL zur Verfügung gestellt. Dieser bietet mehrere Funktionalitäten wie Code-Completion, Refactorings oder einen Debugger. (vgl. [PSV13])
Die Integration in Eclipse war laut Pech et al. Ende 2013 geplant. (vgl. [PSV13]) Im Oktober 2014 wieß 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 Umsetung des Prototypen evaluiert. Dabei sind folgende Kriterien von Belang.

Machbarkeit der Intergration 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.

Erweiterbarkeit der Grammatik
Die Grammtik muss erweiterbar sein, weil das GUI in profil c/s kein abgeschlossenes Konzept ist. Es ist davon auszugehen, dass neue Komponenten in Zukunft benötigt werden

Bereistellung eines Editors für DSL-Skripte

[4] https://www.jetbrains.com/mps/

Ein Editor unterstützt die effiziente Entiwcklung. Features wie Code-Completion oder ausdrücksvolle Fehlermeldungen sind der deg daher wichtig.

Erweiterbarkeit der Validierungen

Um ausdrucksvolle Fehlermeldungen verwenden zu können, ist es notwendig Validierungen durchzuführen, die entsprechende Fehler aufdecken können. Die Standart-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 vorgestellen Frameworks ebenfalls mit einbezogen.

Die Bewertung ist Tabelle 8.1 zu entnehmen. Dabei wurden drei Bewertungsstufen (Gut (+), Ausreichend (O) und Ungenügend (-)) verwendet.

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

Tabelle 6.1: Bewertung der Frameworks für die Entwicklung von DSLs

Die Machbarkeit der Integration von MPS in Eclipse ist derzeit noch nicht gewährleistet. Für Xtext gibt es ein entsprechendes Plugin und da es eine Java-Version von PetitParser existiert, ist auch die Integration nach Eclipse möglich.
Bei der Möglichkeit zur Erweiterbarkeit der Grammatik müssen bei keinem Framework 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 entscheidene 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]) Inkrementelle Modelle können wie in Abbildung 7.1 dargestellt werden.
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])

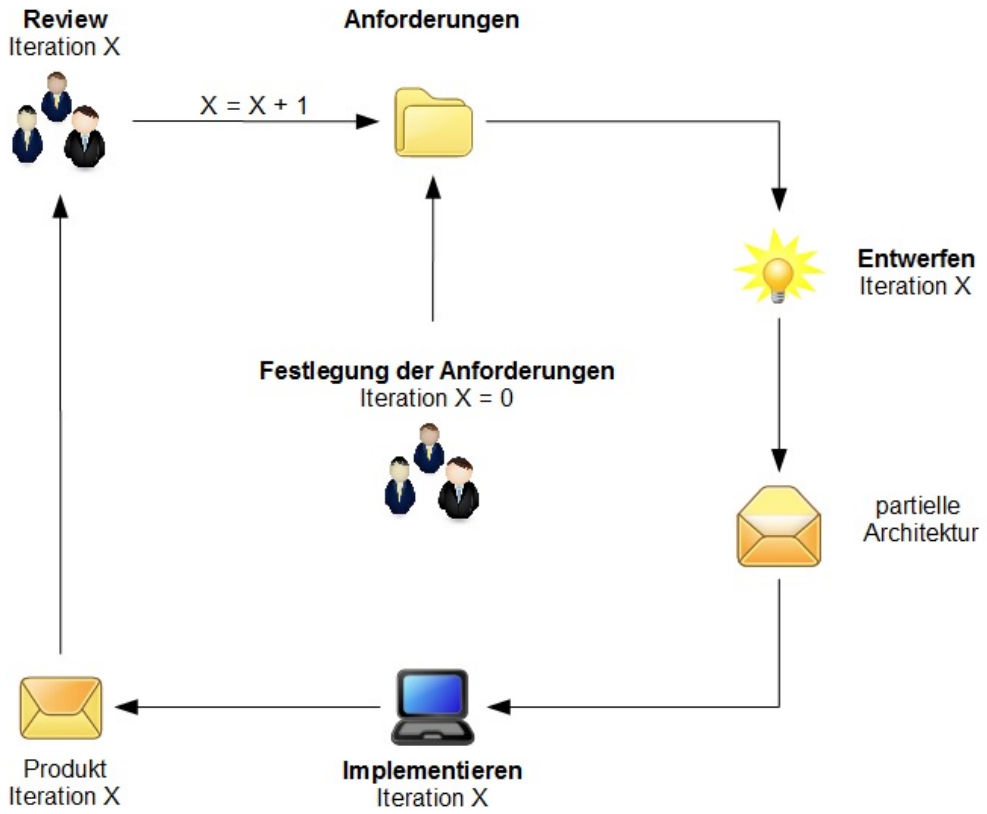

Abbildung 7.1: Inkrementelles Modell

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)

  Dabei wird das Konzpt der DSL beschrieben. Bezogen auf die schematische Darstellung eines inkrementellen Modells (siehe Abbildung 7.1) ist die Vision als *Anforderung* zu betrachten.

- **Entwicklung**

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

- **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 des GUI
- Generator zur Generierung von Code

In Abbildung 7.2 sind die Artefaktion mit den Funktionen (Blauer Kasten), die von ihnen umgesetzt werden sollen dargestellt.

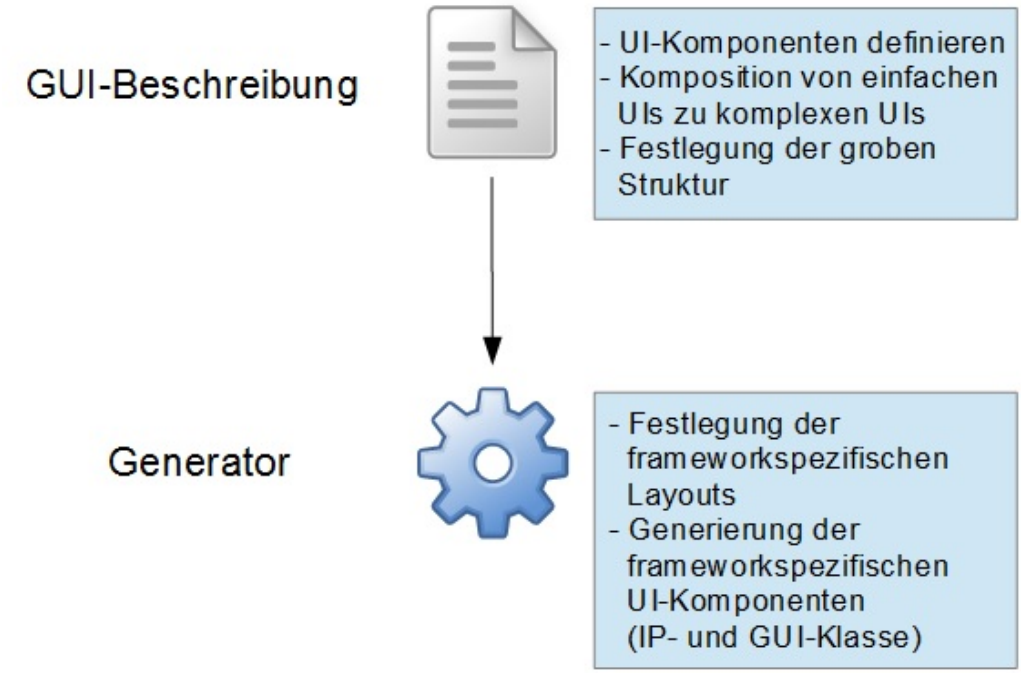

Abbildung 7.2: Konzeption der DSL-Umgebung (1. Iteration)

Die elementare Aufgabe der GUI-Beschreibung ist es, die UI-Komponenten, die in dem GUI verwendet werden sollen, zu definieren. Dazu gehören auch Beschreibungen bzgl. der Art der Interaktion und den Aktionen, die bei der Interaktion ausgeführt werden sollen. Weiterhin sind hierbei die allgemeinen Anforderungen aus Kapitel 2.1 zu beachten.

Darüber hinaus sollen die Skripte für die Beschreibung der UIs so konzipiert werden, dass es möglich ist andere GUI-Beschreibungen dort einzubinden. Somit werden eingebundene GUIs wiederum zu UI-Komponenten. Ziel dessen ist es, dass die Entwickler aus mehreren einfachen UIs ein komplexes UI erstellen können (Modularisierung). Die Gefahr die dabei besteht ist, dass mit der Zeit viel einfache GUI-Beschreibungen mit ähnlicher Strukturiert entwickelt werden. Da die deg bei den GUIs von profil c/s ein bestimmtes Schema verfolgt (Cooperate Design), sollte dieses Problem dadurch ausreichend eingedemmt 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 und beliebig wiederverwendet und kombiniert werden. Hierzu werden unterschiedliche Suchfelder definiert, die aus einem Label und einem Textfeld bestehen (Beispiel siehe Abbildung 7.3).

Abbildung 7.3: Beispiel: Suchfeld für Name

In eine Suchmaske können mehrere dieser Suchfelder beliebig komponiert werden. (Beispiel siehe Abbildung 7.4).

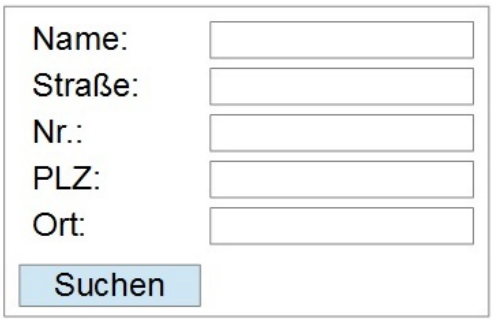

Abbildung 7.4: Beispiel: Suchmaske

Durch diese Möglichkeit der Komposition können auch komplexere fachliche Konzepte auf das UI bezogen werden, wei z.B. *Personensuche*.

Um die miteinander komponierten UI-Komponenten zu strukturieren ist es notwendig, dass die GUI-Beschreibung Informationen über die Anordnung der UI-Komponenten enthält. Diese Inforamtionen sollen ausreichd abstrakt sein, damit sich diese Struktur auf unterschiedliche UI-Frameworks beziehen lässt. Dazu wird die Struktur innerhalb eines GUIs als Anordnung von Bereichen betrachtet. In der GUI-Beschreibung werden diesen Bereichen die UI-Komponenten zugeordnet. Genauere Informationen über die Anordnung dieser Bereiche dürfen nicht enthalten sein, da dies eine Orientierung an bestimmte Layouts bedingt (was per Anforderung ausgeschlossen ist - siehe Kapitel 2.1). Dazu ist weiterhin wichtig, dass den Bereichen jeweils nur eine Komponente zugeordnet werden kann. Dadruch wird der Zwang zur vorher beschriebenen Komposition von eingebundenen GUI-Beschreibungen und definierten UI-Komponenten verstärkt werden.
Der Generator übernimmt die konkrete Anordnung der beschriebenen Bereiche. Grund dafür ist, dass die Anordung der Komponenten frameworkspezifisch ist (teilweise werden unterschiedliche Layoutmanager in unterschiedlichen Frameworks unterstützt). Da für jedes eingesetzte Framework ein eigener Generator implementiert werden muss (siehe Kapitel 4), ist es theoretisch möglich diese Aufgabe weitgehend unabhängig von der Beschreibung der verwendeten Komponenten zu erfüllen.
Die wichtigste Aufgabe des Generator ist jedoch die Generierung des frameworkspezifischen Quell-Codes.

7.2.2 2. Iteration

Nach dem Review der ersten Iteration des Prototypen wurde das grundsätzliche Konzept um ein Artefakt erweitert (siehe Abbildung 7.5).

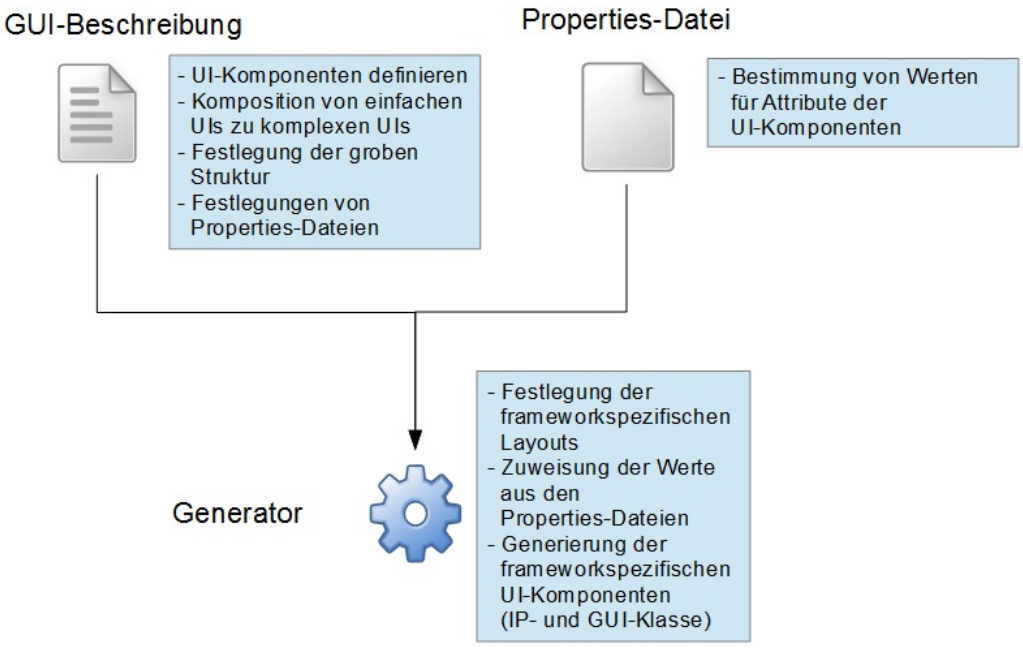

Abbildung 7.5: Konzeption der DSL-Umgebung (2. Iteration)

Eine Änderung, die in dieser Grafik nicht dargestellt wird, ist, dass die Aktionen, die bei Interaktionen mit UI-Komponenten ausgeführt werden sollen, nicht in der GUI-Beschreibung definiert werden sollen. Grund dafür ist, dass diese Aktionen sehr unterschiedlich bei profil c/s sind und somit kaum abstrahiert werden können.
Eine weitere Änderung ist, dass die UI-Komponenten, die einer GUI-Beschreibungen direkt definiert werden, in einer anderen Beschreibung, wo jene GUI-Beschreibung eingebunden ist, verändert werden können. Dadruch werden die wiederverwendeten Beschreibungen anpassbar, was die Flexibilität enorm steigert.
Darüber hinaus soll die Möglichkeit bestehen, bestimmte Werte, welche die Attribute von UI-Komponenten annehmen können, in Properties-Dateien auszulagern. Dadruch wird die GUI-Beschreibung weitgehend entlastet. Allerdings muss für die Zuweisung von UI-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 entsprechende Werte entnehmen. Dabei gilt die Festlegung, dass wenn in der GUI-Beschreibung einem Attribut ein bestimmter Wert zugewiesen ist, wird in der Properties-Datei (wenn eine festgelegt wurde) nicht mehr nach diesem Attribut der Komponente gesucht.

7.2.3 3. Iteration

In dem Review, welches vor der 3. Iteration durchführt wurde, entstanden die letzten Änderungen, die bzgl. der Sprache innerhalb dieser Arbeit Beachtung finden. Das grundsätzliche Konzept wurde 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 7.6 ist das neue Konzept schematisch dargestellt.

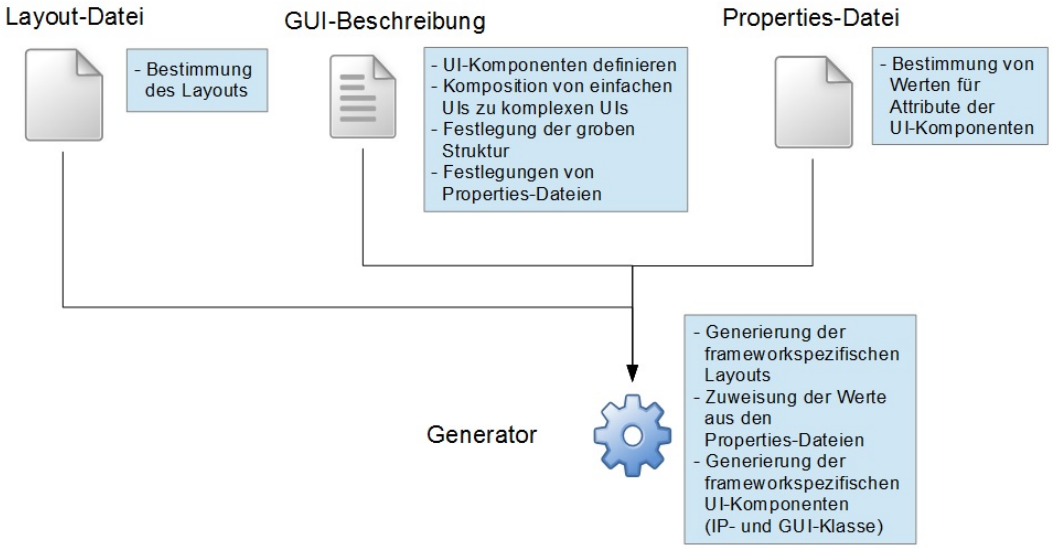

Abbildung 7.6: Konzeption der DSL-Umgebung (3. Iteration)

Da die Layout-Beschreibung in eine separate Datei augegliedert wurde, muss die Referenzierung von UI-Komponenten und Bereichen aus der GUI-Beschreibung mit den Festlegungen in der Layout-Datei referenziert werden können.
Bezogen auf die in der GUI-Beschreibung definierten Bereiche ist aufgefallen, dass die Angabe der Anzahl der Bereiche nicht notwendig ist. Grund dafür ist, dass dieser Wert doppelt definiert weden würde. Einerseits direkt in der Angabe der Anzahl und andererseits indirekt durch die Zuweisung

der UI-Komponenten und eingebundenen GUI-Beschreibungen 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 LOCs wie möglich geschrieben werden sollen, um eine GUI zu beschreiben (siehe Kapitel 2.1).
Weiterhin ist aufgefallen, dass in einer GUI-Beschreibung 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-Beschreibung via DSL zu beachten.
Neben den Basiskomponenten *Button* und *Label* müssen folgende weitere Basiskomponenten definiert werden können.

- Textfield
- Textarea
- Tree
- Table
- TabPane
- Interchangeable
  Ein Bereich, in dem während der Laufzeit unterschiedliche UIs eingebunden werden können.

Weiterhin müssen die Bezeichnungen der Interaktionstypen den Bezeichnungen der Interaktionsformen (IF) der deg angeglichen werden. Darüber hinaus sollen folgende Standard-Interaktionstypen verwendet werden. (Zuerst wird die Komponente genannt und anschließend die Standard-Interaktionstypen)

- Button: IfActivator, IfTextDisplay
- Interchangeable: Änderung des GUIs
- Tree: Doppelklick

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 des GUIs

Die Beschreibung einer GUI wird in der Sprache als eigener Komplex betrachtet (siehe semantisches Modell *Definition*). Innerhalb dieses Komplexes werden die entsprechenden Komponenten definiert. Die Bereiche die innerhalb einer Beschreibung festgelegt werden sollen, müssen UI-Komponenten zugeordnet werden können (siehe semantisches Modell *AreaAssignment*). Diese Bereiche sollten vor der Entwicklung bereits festgelegt werden. Um abzusichern, dass die 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 Model *TypeDefinition*).
Ein weiterer Aspekt in GUI-Beschreibung ist die Verwendung von anderen GUI-Beschreibungen (siehe semantisches Modell *Use*).
Zusammenfassend sind für die Beschreibung der GUI folgende Metadaten nötig.

Anzahl der Bereiche

Zuweisung der UI-Komponenten zu den Bereichen

Angabe des Layout-Typs

Angabe der Verwendeten GUI-Beschreibungen

Definition von UI-Komponenten

Die Definition der UI-Komponenten nehmen einen eigenen Komplex innerhalb der GUI-Beschreibung ein. Bezogen auf die Basiskomponenten des UIs 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 UI-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 Meta-Daten 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.

Aktions-Typ
Zur Unterscheidung zwischen Interaktionen mit anderen UI-Komponenten oder fachlichen Modellen

Element
Ein Verweis auf das Element, mit dem interagiert werden soll.

Attribute (siehe semantisches Modell *Property*)
Die zu veränderden Attribute des Elements.

Die komplexen Komponentenwerden in einer eigenen Komponentendefinition beschrieben. Grund dafür ist, dass neben den vordefinierten Funktionalitäten der komplexen Komponentenauch 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 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

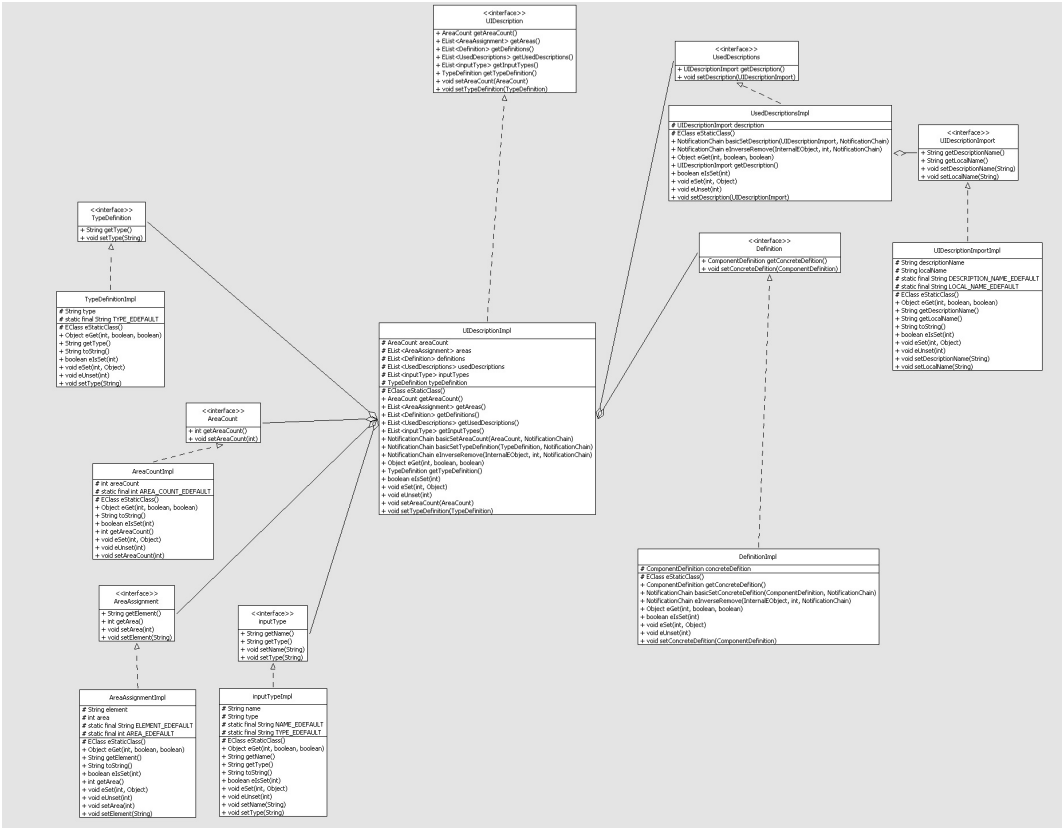

Abbildung 8.1: Teil 1: GUI-Beschreibungs-Model Version 1

Das Artefakt, welches beim dieserm Model im Mittelpunkt steht ist die *UI-Description* (siehe 8.1). Bezüglich der Methoden sind im Allgemeinen nur Getter- und Setter-Methoden des semantischen Modells für diese Arbeit von Belang. Die aggregierten Artefakte sind aus dem Diagramm gut zu entnehmen. Die Klasse *DefinitionImpl* aggregiert weitere Artefakte des Modells. Diese sind um die Übersicht zu wahren Abbildung 8.2 zu entnehmen.

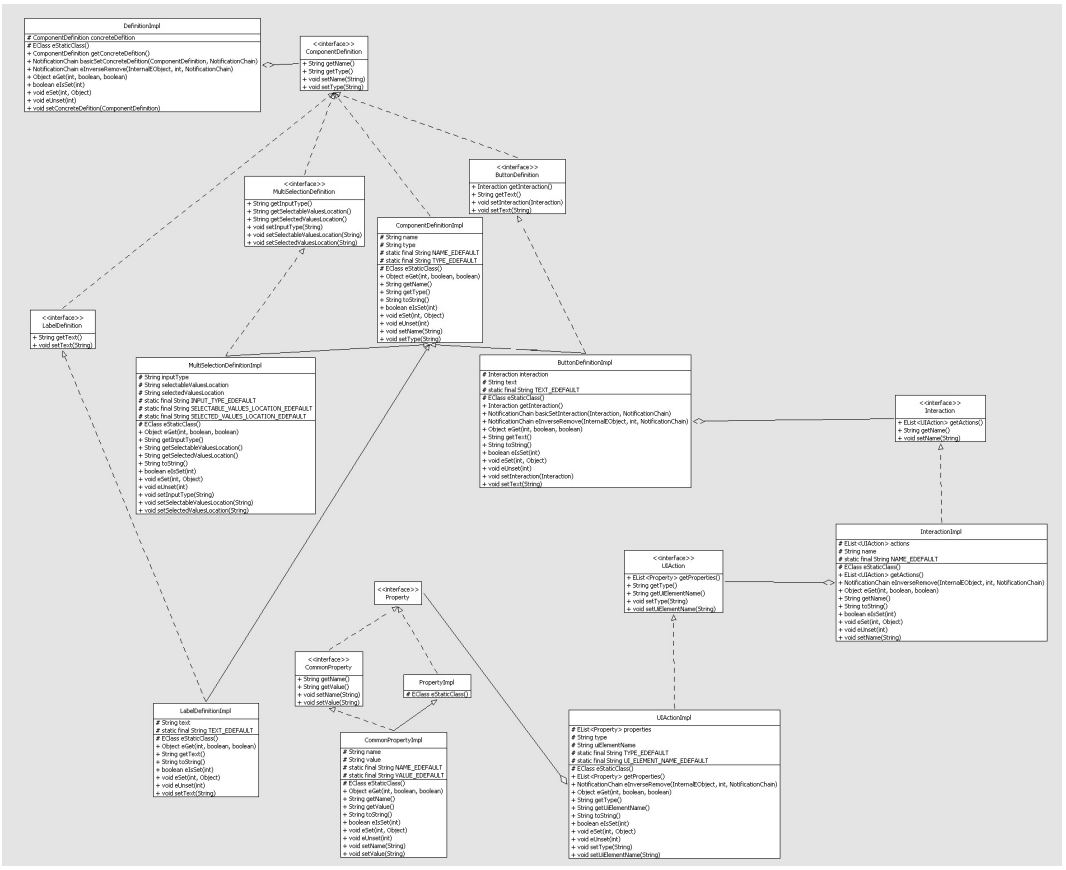

Abbildung 8.2: Teil 2: GUI-Beschreibungs-Model Version 1

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* aggregieren kann. Das bedeutet, dass nur an dieser Komponente eine Interaktion beschrieben werden kann. Das Artefakt *Property* ist der letzte interessante Teil. Dieses Interface wird benötigt um bestimmte Werte an Komponenten zu setze, ohne das Wissen zu müssen um welchen Komponententyp es sich handelt. Dazu wurden die allgemein gültigen Einstellungsmöglichkeiten von Basiskomponenten in *CommonProperty* zusammengefasst. Die Klasse *PropertyImpl* ist ein Artefakt, welches zur Vollständigkeit des Modells erzeugt wurde. Es erfüllt für diese Version jedoch keinen weiteren Zweck.

Konkrete Syntax

Listing 8.1: 1. Iteration: Syntax

```
1 Area count: 4
2 type: WINDOW use: "AnotherDescription"
```

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

Die Bezeichnung *Area* wurde bewusst so gewählt, da dieser Begriff abstrakter ist als die in verschiedenen UI-Frameworks verwendeteten Begrifft wie, Panel oder Pane. In der Syntax dieser DSL gilt es sich vor allem bzgl. des Aufbaus der GUI an keinem UI-Framework zu orientieren. Die einzelnen Komponentendefinitionen werden durch das Schlüsselwort *DEF* eingeleitet und durch das Schlüsselwort *END DEF* abgeschlossen. Der Definitionskopf wird durch das Zeichen *:* beendet. Dort sind die Pflichtfelder der Komponentendefinition zu finden (*Titel* und *Typ*). Bei der Multiselection-Komponente fällt auf, dass ein Referenzwert verwendet wird, der in dieser Beschreibung nicht deklariert wurde (*valuespackage.Values*). Dabei handelt es sich um einen qualifizierten Namen einer Klasse.
Die dazugehörige Grammatik befindet sich im Anhang 10.

8.2 2. Iteration

Analyse der Metadaten des GUIs

Für die Metadaten der GUI-Beschreibung werden muss in dieser Iteration eine Properties-Datei angegeben werden, in der bestimmte Werte für die Attribute der UI-Komponenten enthält.
Da die Möglichkeit bestehen soll die, in den eingebundenen GUI-Beschreibungen definierten, Komponenten in dieser Iteration zu verändert, wird eine weitere Ergänzung für die GUI-Beschreibung benötigt. Diese Veränderungen sollen sich von der Komponentendefinition abgrenzen, damit für nicht über-

schriebene Werte auf die eingebundene GUI-Beschreibung zurückgegriffen werden kann (siehe semantisches Modell *Refinement*). Um die eindeutige Referenzierung zu ermöglichen muss bei der Bezeichnung der eingebundenen GUI-Beschreibung sowie bei der veränderten Komponente der qulifizierte 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 Komponentenist 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 Komponentenmittels *use* (siehe semantisches Model *UsedDefinitions*) in die GUI-Beschreibung einzubinden (siehe konkrete Syntax).

Semantisches Modell

In dieser Version wurden an den Artefakten *AreaCount*, *TypeDefinition* und *AreaAssignment* keine Änderungen vorgenommen. Artefakte wie *Property* und *Refinement* sind hinzugekommen. Die weiteren Artefakte, die von *UIDescriptionImpl* aggregiert werden (siehe Abbildung 8.2), wurden verändert.

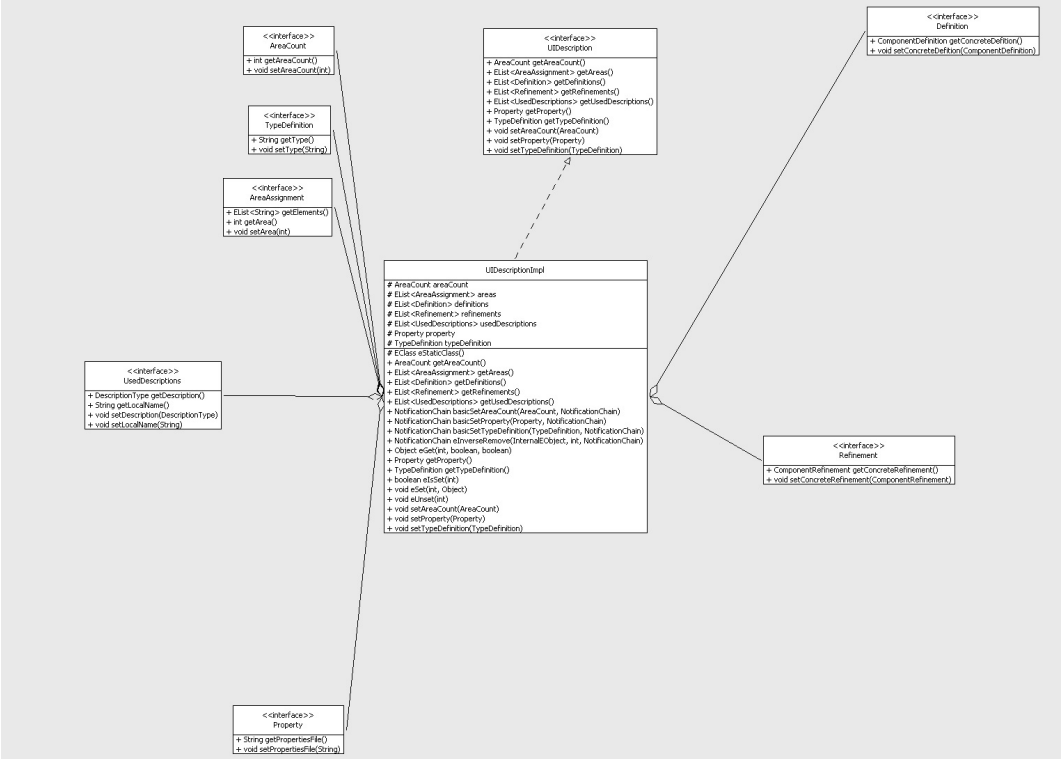

Abbildung 8.3: Teil 1: GUI-Beschreibungs-Model Version 2

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

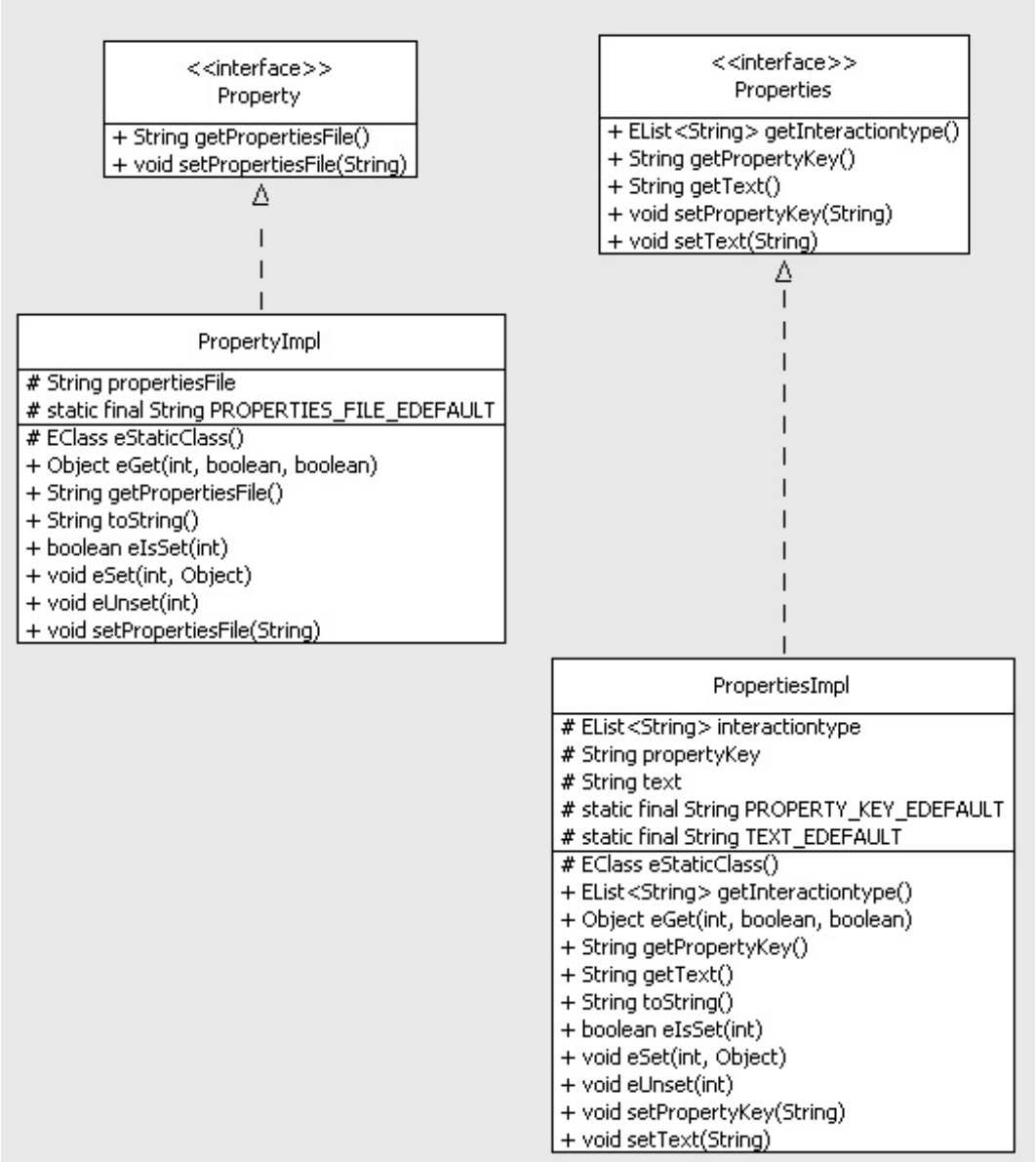

Abbildung 8.4: Teil 2: GUI-Beschreibungs-Model Version 2

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

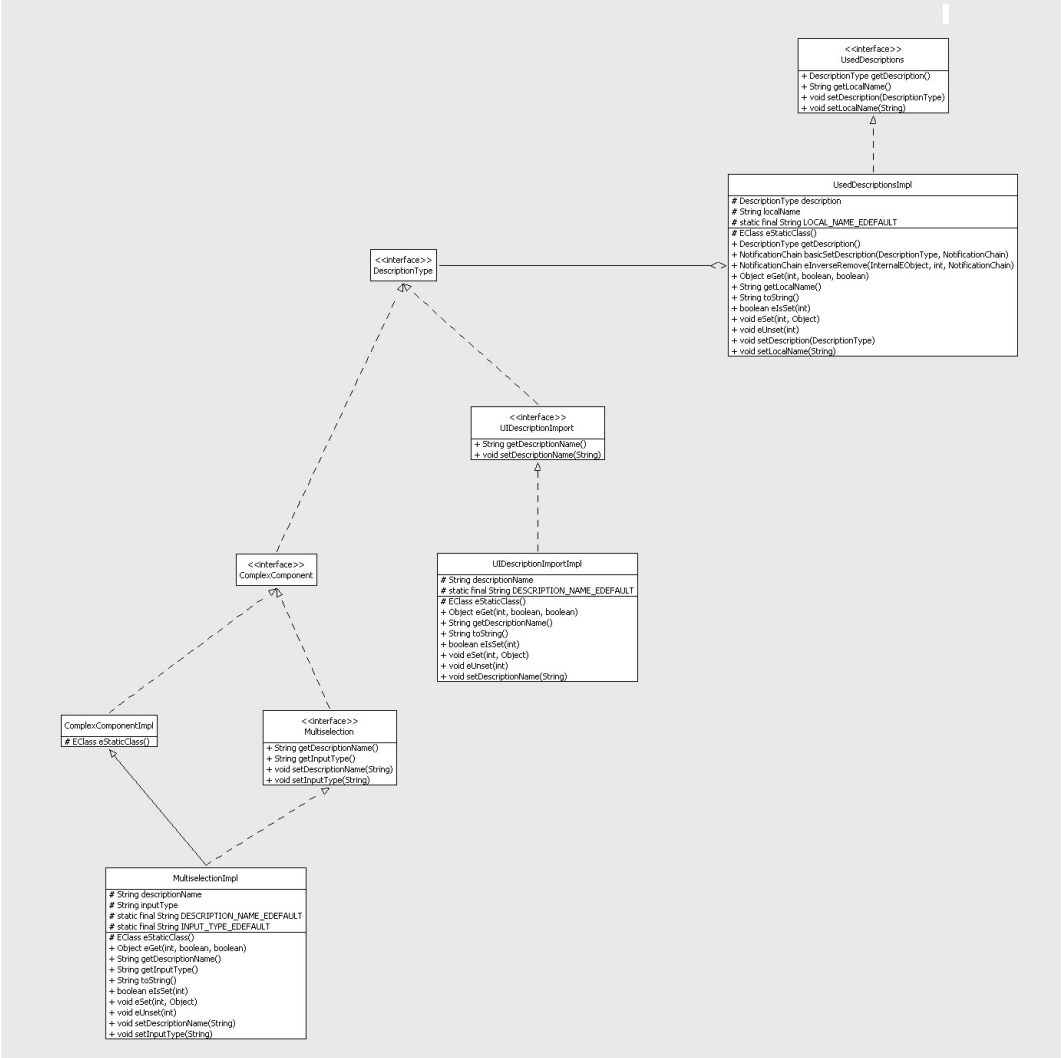

Abbildung 8.5: Teil 3: GUI-Beschreibungs-Model Version 2

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

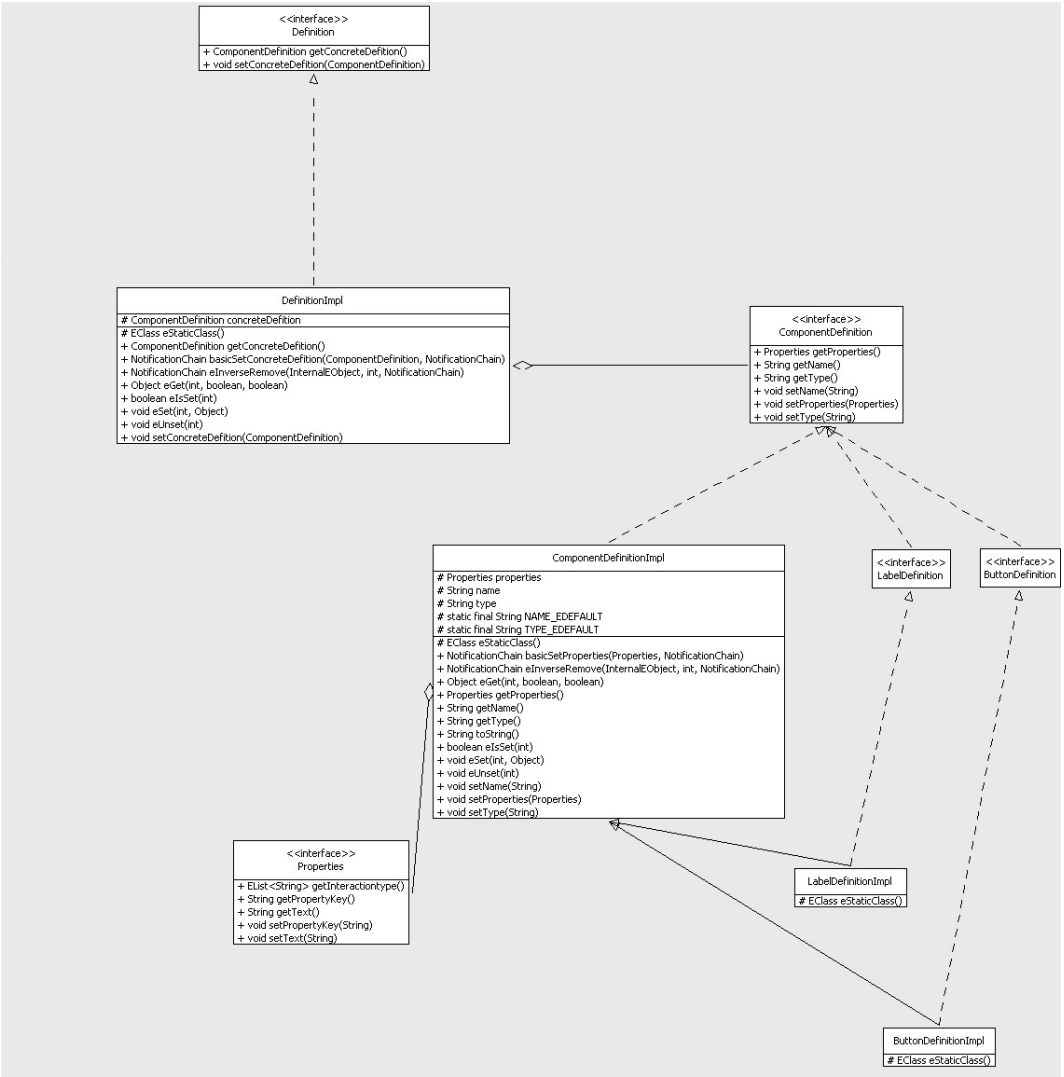

Abbildung 8.6: Teil 4: GUI-Beschreibungs-Model Version 2

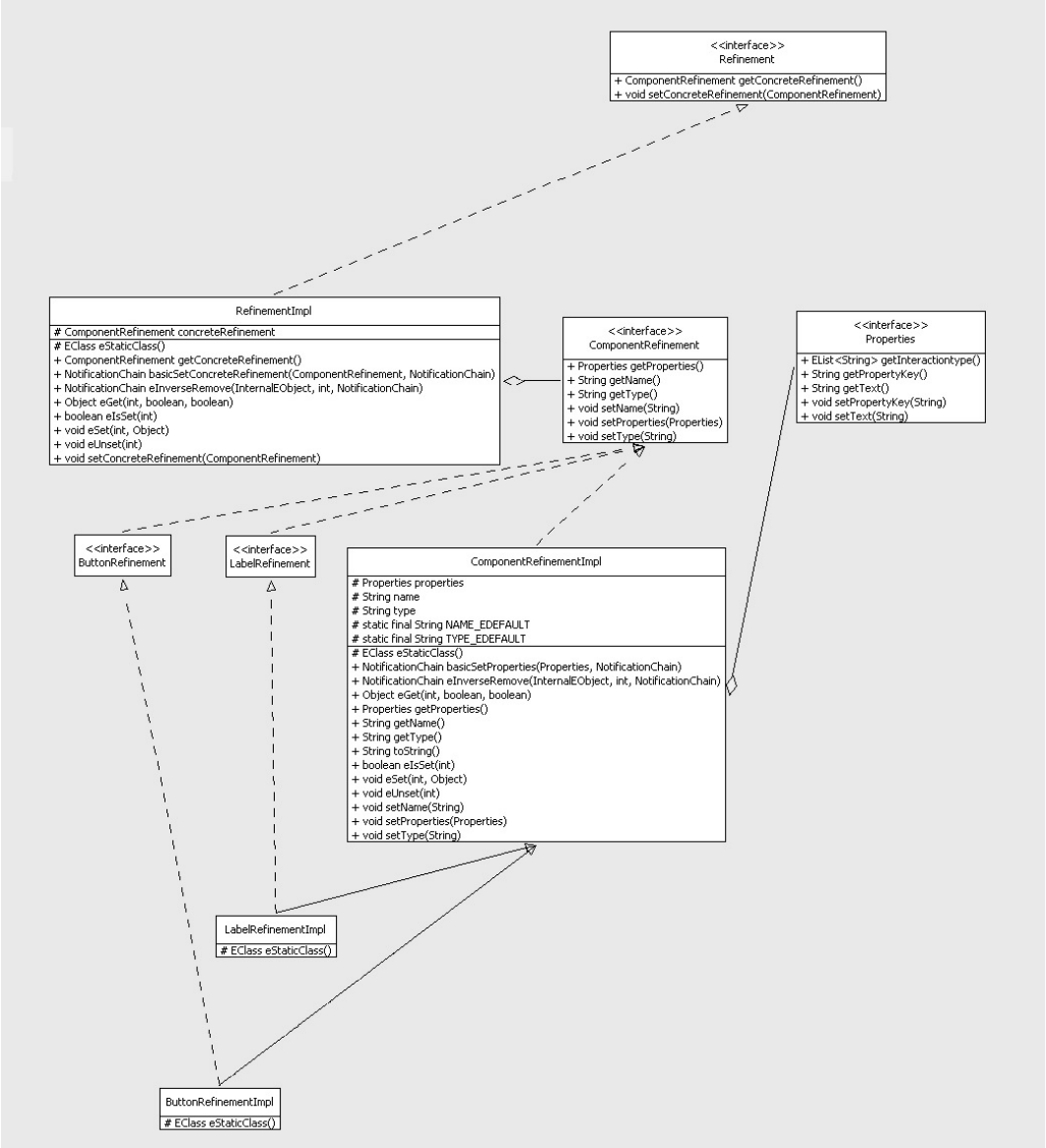

Abbildung 8.7: Teil 5: GUI-Beschreibungs-Model Version 2

Konkrete Syntax

Die einfachste der Veränderungen bzgl. der Syntax in Version 2 ist die festlegung der Properties-Dateien. Um eine Properties-Datei einzubinden muss, wie in Listing 8.2, eine entsprechende Datei festgelegt und in den Komponenten entsprechende Schlüssel vergeben werden. Das Label mit der Bezeichnung *OneLabel* enthält keinen Property-Key. In diesem Fall wird der Titel als solcher verwendet.

Listing 8.2: 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 Meta-Daten 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 Tatsache, dass in der deg höchstens 4 Interaktionstypen in einer Komponente verwendet werden, werden diese in der GUI-Beschreibung per DSL hintereinander mit Komma aufgezählt, wie in Listing 8.3 zu erkennen ist.

Listing 8.3: 2. Iteration: Interaktion

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

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

Listing 8.4: 2. Iteration: Definition komplexer Komponenten

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

Für die Zuweisung mehrer Komponenten zu den Areas kamen zwei Lösungen in Betracht. Bei der einen finden die Definitionen der Komponenten zusammen mit der Zuweisung zu dem Area statt. Dies könnte bspw. wie in Listing 8.5 dargestellt werden.

Listing 8.5: 2. Iteration: Area-Zuweisung (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 ge-

trennt von einander aufzählen. Die erste Möglichkeit würde sich sehr gut eignen, wenn nur die in der Datei definierten Komponenten dem Area zugewiesen werden müssten. Da die mit *use* eingebundenen Komponenten auch Areas zugeordnet werden, würde für dieses Verfahren ein zusätzliches syntaktisches Konzept innerhalb der Area-Zuweisung benötigt werden. Um dies zu umgehen wurde die Entscheidung getroffen, das alte Verfahren zu verfeinern. Listing 8.6 ist ein Beispiel für die Area-Zuweisung von drei Komponenten zu entnehmen.

Listing 8.6: 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 einer eingebundenen GUI-Beschreibung definiert wurden, können über das Schlüsselwort *REFINE* getätigt werden. Der erste Teil von Listing 8.7 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 8.7: 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-Beschreibungen 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 8.8)

Listing 8.8: 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 ist im Anhang 10 zu finden.

8.3 3. Iteration

Analyse der Metadaten des GUIs

Um eine Referenzierung von UI-Komponenten und dem entsprechenden Layout aus der Layout-Datei zu ermöglichen, ist es wie bei den Properties notwendig, die Layout-Datei innerhalb der GUI-Beschreibung anzugeben. Weiterhin wird in den einzelnen Komponentendefinitionen ein Schlüssel benötigt, über den die Komponente eindeutig referenziert werden kann. Dafür kann die Bezeichnung der UI-Komponete verwendet werden. Um in der Layout-Datei nicht alle einzelnen UI-Komponenten unterscheiden zu müssen, wird innerhalb der GUI-Beschreibung ein optionales Feld benötigt, mittels dessen unterschiedlichen UI-Komponenten dasselbe Layout zugeordnet werden kann. Dieser Schlüssel wird Layout-Schlüssel genannt.
Da die Anzahl der Bereiche nicht mehr angegeben werden muss, fällt dies auch aus den Metadaten den GUI-Beschreibung heraus. Die Zuweisung der UI-Komponenten zu den Bereichen entfällt ebenfalls. Die Strukturierung wird über eine Aufzählung der UI-Komponenten vorgenommen.

Zusammenfassend die benötigten Metadaten für die GUI-Beschreibung die folgenden.

- Typ
- Properties-Dateien
- Layout-Dateien
- Eingebundene GUI-Beschreibungen
- Veränderte eingebundene Komponentendefinitionen
- Komponentendefinitionen
- Struktur

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

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

Mit den konkreten Kompnentendefinitionen müssen mehrere Basiskomponenten beschrieben werden können. Basiskomponenten die spezielle Metadaten benötigen sind mit diesen in folgender Tabelle aufgelistet.

Basiskomponente	**Spezielle Metadaten**
Label	Aufschrift
Button	Aufschrift
Textfield	Text, Editierbarkeit
Textarea	Text, Editierbarkeit
Tree	Input-Modell
Table	Input-Modell
TabPane	GUI-Beschreibungen für die einzelnen Tabs

Tabelle 8.1: Bewertung der Frameworks für die Entwicklung von DSLs

Von diesen Metadaten sind alle bis auf die des TabPanes 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 8.8) fällt auf, dass *Area* und *AreaCount* nicht mehr vorhanden sind. Hinzugekommen sind *Structure* (worin die Anordnung der UI-Komponenten (*Element*) in einer Liste abgelegt wird) und Layout (worin die zu verwendenen 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.

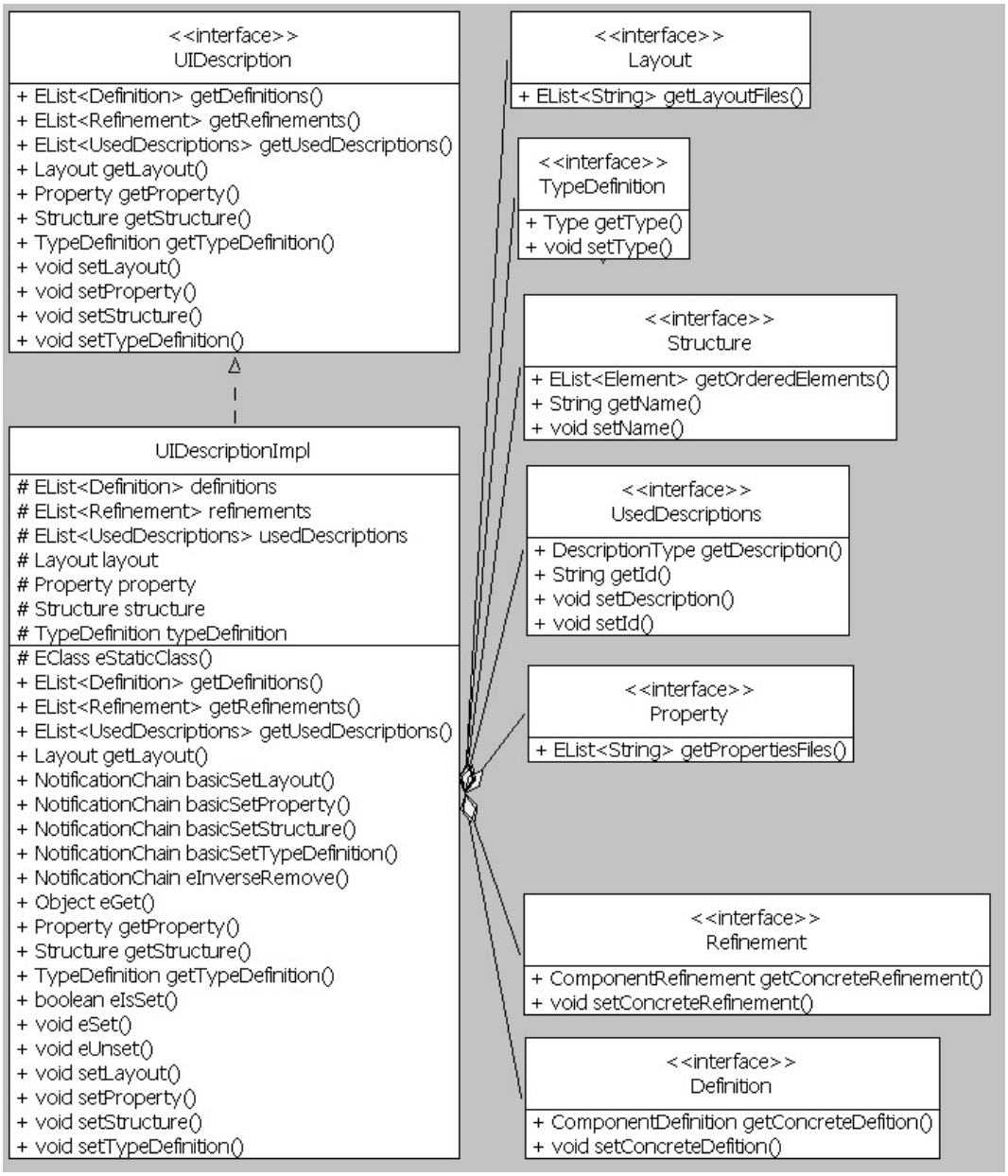

Abbildung 8.8: 3. Iteration: UIDescription

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 eines bestimmter Klassen, die das Interface *ComponentRefinement* oder *ComponentDefinition* implementieren, enthält. In Abbildung 8.9 ist diese Struktur für die *Definition* abgebildet.

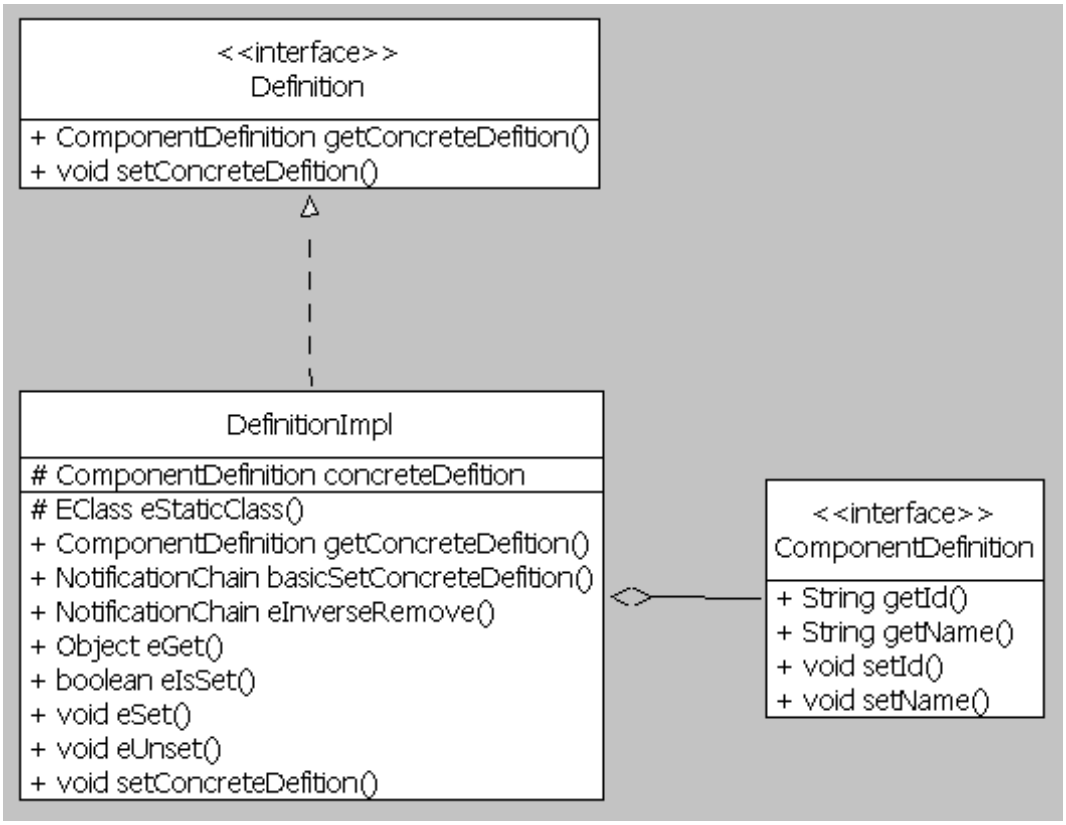

Abbildung 8.9: 3. Iteration: Definition

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 bilder 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 8.10 zeigt dies für einen Button auf.

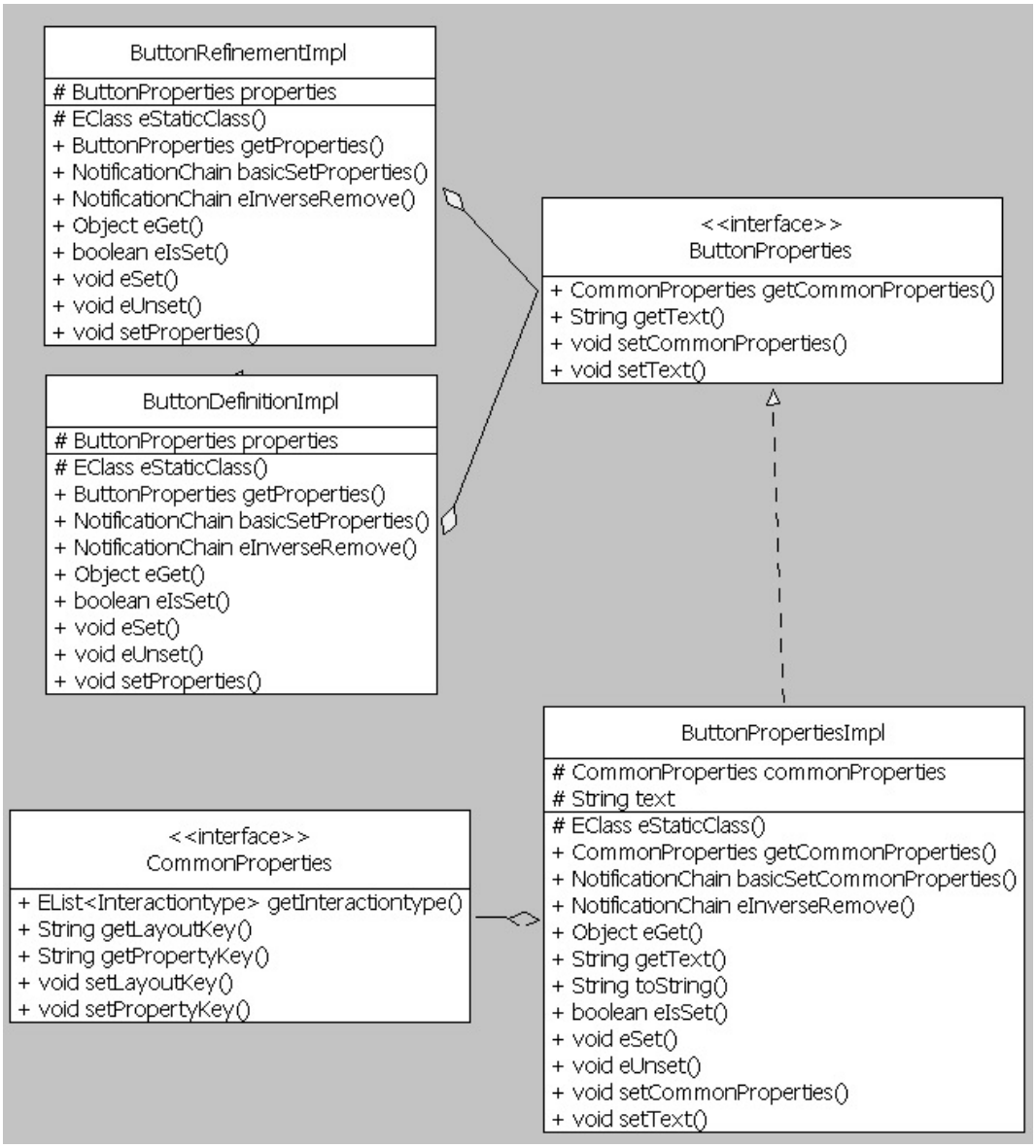

Abbildung 8.10: 3. Iteration: Button

Eine direkte Aggergation der *CommonProperties* findet bei den Basiskomponenten TabView, Table und Tree statt. Für die speziellen Properties dieser Basiskomponenten existieren keine einelnen Klassen. Die folgenden Abbildungen zeigen diese drei Basiskomponenten (*Refinement* und *Definition*) mit der Aggregation der *CommonProperties*. Um die Übersicht zu bewahren werden die Methoden der Klassen nicht mit aufgeführt.

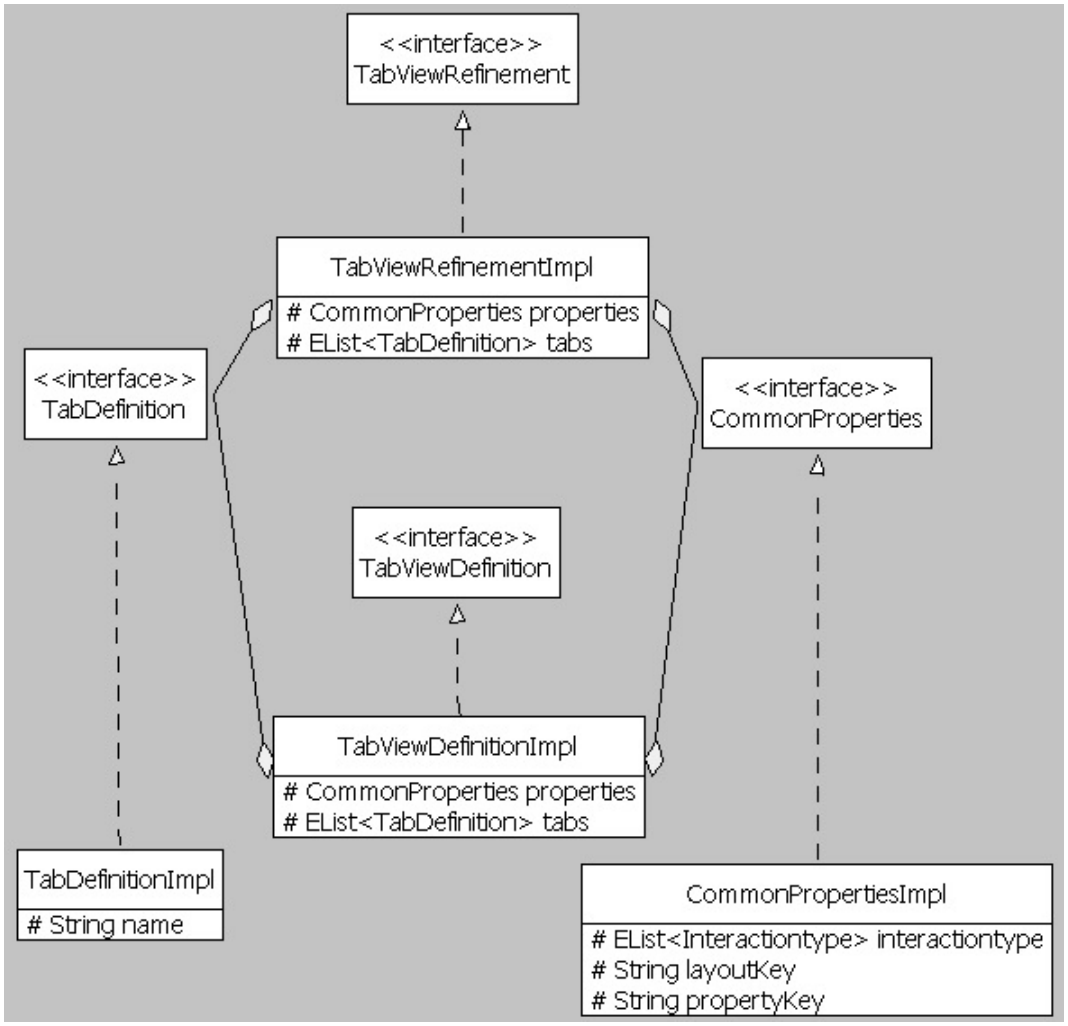

Abbildung 8.11: 3. Iteration: TabView

Die Basiskomponente TabView benötigt eine Menge von *TabDefinition*s. Diese Klasse bildet die Referenz zu den in die TabView einzubindenen UI-Komponenten. Die Basiskomponenten Tree und Table benötigen lediglich eine Referenz auf den Input-Typen, den sie abbilden sollen, in Form einer Zeichenkette (siehe Abbildung 8.12 und 8.13).

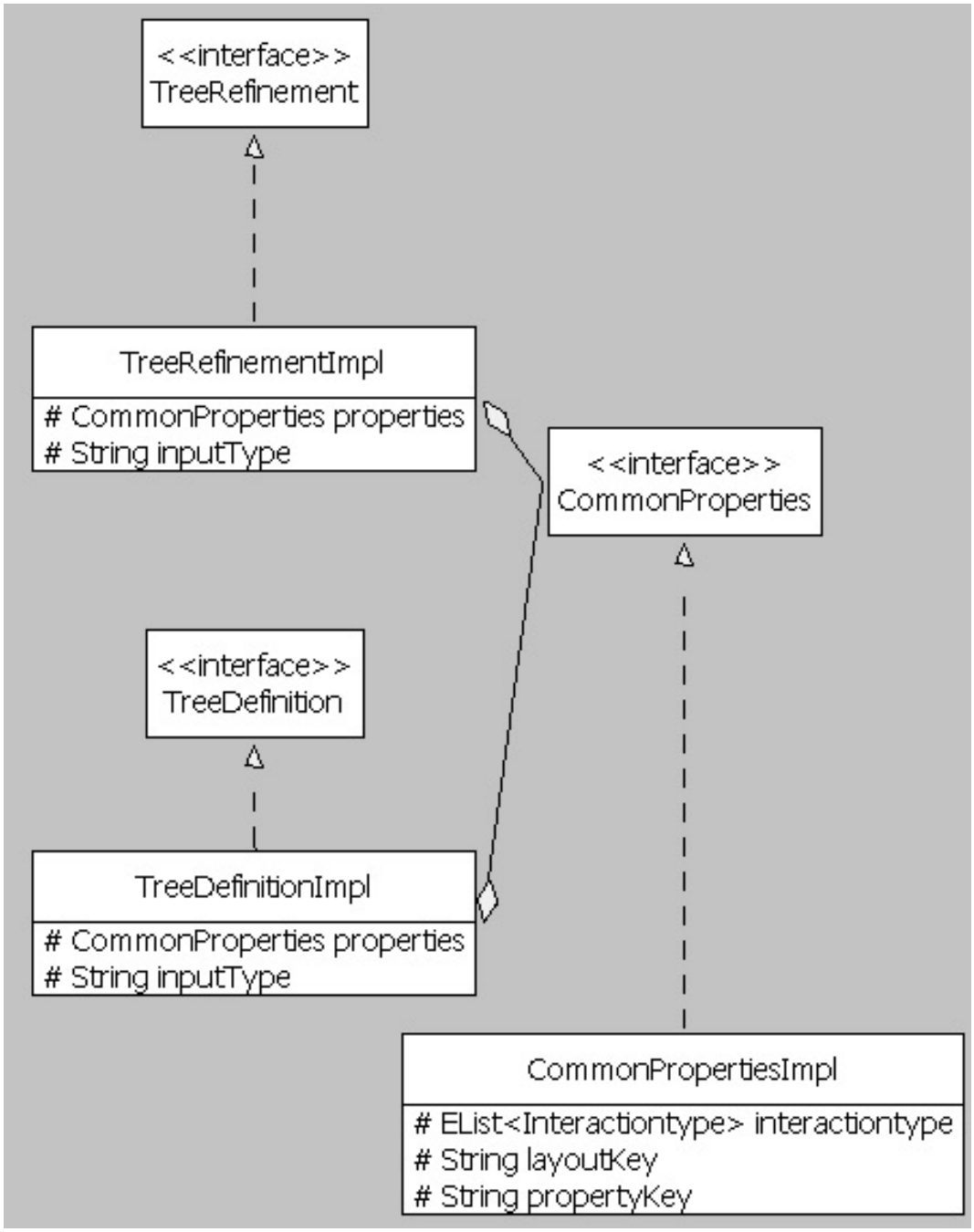

Abbildung 8.12: 3. Iteration: TabView

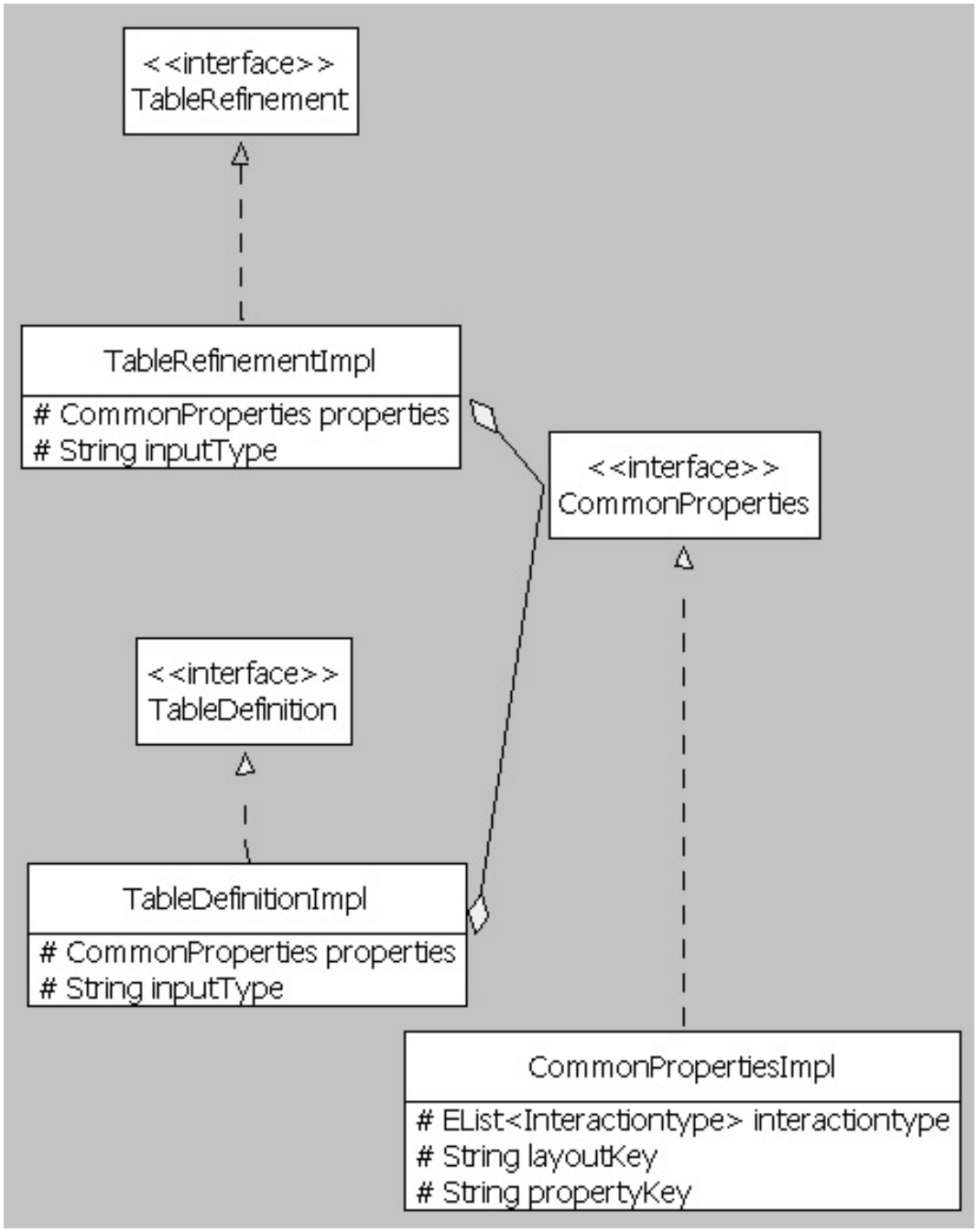

Abbildung 8.13: 3. Iteration: TabView

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 einer GUI-Beschreibung 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 Kom-

ma 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 (sieh Listing 8.9).

Listing 8.9: 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-Beschreibungen bzw. komplexe Komponentenwurden keine großen syntaktischen Veränderungen vorgenommen. Lediglich das Semikolon wird für den Abschluss des Komplexes benötigt (siehe Listing 8.10).

Listing 8.10: 3. Iteration: Eingebundene UI-Komponenten

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

Die Definitionen der einzelnen Basiskomponenten hat sich syntaktisch stark verändert. Grund dafür ist vor allem, dass die Anforderung Nr. 5 (siehe Kapitel 2.1) mehr Beachtung finden soll. Somit werden die bekannten Basiskomponenten wie folgt definiert (siehe Listing 8.11).

Listing 8.11: 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 allgemienen Properties wie im vorherigen Beispiel zugewiesen. In den Fällen der Basiskomponenten Textflied und Textarea werden die speziellen Properties nach demselben Prinzip definiert (siehe Listing **??**)

Listing 8.12: 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 ebenfalls ähnlich. Der benötigte Input-Typ wird nach dem Schlüsselwort angegeben, welches die UI-Komponente bestimmt (siehe Listing 8.13).

Listing 8.13: 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 UI-Komponenten der einzelnen Tabs ebenfalls nach dem Schlüsselwort angegeben, welches die UI-Komponente festlegt. Hierbei können anderes als bei den Basiskomponenten Table und Tree mehrere Angaben getätigt werden (siehe Listing 8.14). In den folgenden Beispiele wird davon ausgegange, dass die vorher genannten Beispiele für die Basiskomponenten Tree, Table und Textarea in derselben GUI-Beschreibung definiert wurde. Durch das syntaktische Konstrukt in Listing 8.14 wird eine Tab-Ansicht mit drei Tabe beschrieben. Das erste Tab enthält den Tree, das zweite die Table und das dritte die Textarea.

Listing 8.14: 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 einer eingebundenen GUI-Beschreibung ähnelt der Definition einer Basiskomponente des gleichen Typs. Es muss lediglich angegenben werden, welche UI-Komponenten in welcher GUI-Beschreibung verändert werden soll. Folgendes Beispiel zeigt, wie die Aufschrift eines Buttons einer eingebundenen GUI-Beschreibung verändert wird (siehe Listing 8.15). Der Button trägt die Bezeichnung *EmbeddedButton* und die GUI-Beschreibung liegt im Package *guidescription* und trägt die Bezeichnung *Embedded*.

Listing 8.15: 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 UI-Komponenten zu Bereichen, wie es auch der ersten und zweiten Iteration bekannt ist. Da dieses Konstrukt vereinfacht werden sollte, werden die UI-Komponenten in der richtigen Reihenfolgen nach dem Schlüsselwort *Structur* aufgezählt (siehe Listing 8.16).

Listing 8.16: 3. Iteration: Struktur

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

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 DSL-Skript eine Exploreransicht zu erststellen, wie sie in profil c/s geläufig ist. Abbildung 9.1 zeigt eine solche 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.

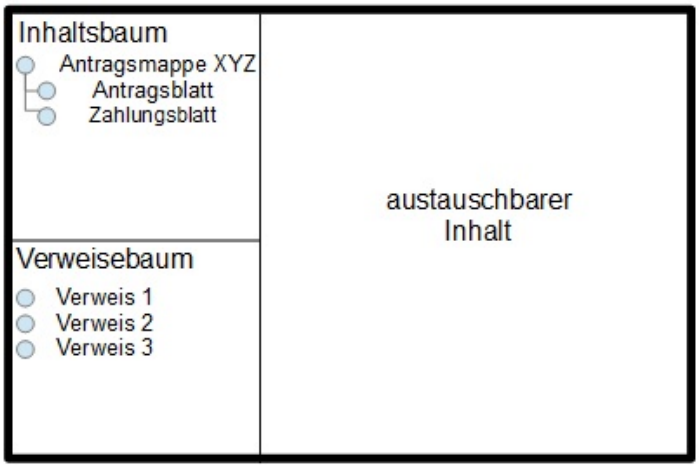

Abbildung 9.1: Einfacher Explorer

Die notwendigen DSL-Skripte werden bei der Umsetzung vorgestellt.

9.1 Beschreibung der GUI- und IP-Klassen

Durch die Beschreibung der GUIs mit der entwickelten 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 LayoutDatei 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 UI-Komponenten durch *Präsentationsformen* beschrieben (siehe Kapitel 2). Alle UI-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 (siehe Kapitel 2 - R1). Da die Interaktion von der Präsentation getrennt ist, müssen zur Referenzierung von Interaktionen zu UI-Komponenten entsprechende Schlüssel vergeben werden.
Die IP-Klassen ordnen den UI-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 Werkzeugt beschrieben. Eine komplette Generierung der FP-Klassen kann mit der DSL nicht angestrebt werden, da 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. In Bezug auf die IP-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 9.1 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-Beschreibungen erstellt werden (siehe Listing 9.1 und 9.2).

Listing 9.1: GUI-Beschreibung Inhaltsbaum

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

Listing 9.2: GUI-Beschreibung 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 DSL-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 9.3 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 9.3: Speichern der Importe

```
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. Der Klassenkopf und der Konstruktor werden wie folgt erzeugt (siehe Listing 9.4).

Listing 9.4: Klassenkopfgenerierung für einen 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 UI-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 (*descrition.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 UI-Komponenten und das Layout definiert.

Da für das Layout in dieser Arbeit keine Beschreibung existiert, müssen diese Angaben nachgepflegt werden. Die Definition der UI-Komponenten mit ihren Properties können jedoch erzeugt werden. Listing 9.5 zeigt die Methode *init* ohne Berücksichtigung des Layouts.

Listing 9.5: 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 UI-Komponente es sich handelt. Diese wird anschließend compiliert, wobei der entsprechende Quell-Code zur frameworkspezifischen Definition der Komponente generiert wird. Im Fall des Labels werden der entsprechende Import, 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 9.6).

Listing 9.6: 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,
                      true)»
8         «ENDIF»
9 '''
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 Quell-Code 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 9.7).

Listing 9.7: Label-Generierung

```
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        «ELSE»
13                «definition.id».setTreeModel( new DefaultTreeModel(
14                new «definition.inputType.substring(1,definition.inputType.length
                      -1)»() ) );
15        «ENDIF»
16                «definition.id».setCellRenderer( new TreeCellRenderer() );
17 '''
```

Allein durch die genannten Methoden kann der Generator die beiden DSL-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 im Anhang im Projekt *Explorer*. Abbildung 9.2 zeigt die beiden GUIs, welche bei der Ausführung der generierten Dateien im Kontext von profil c/s erzeugt werden.

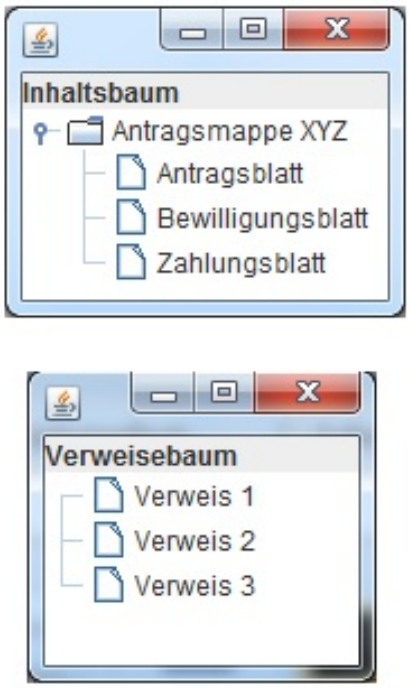

Abbildung 9.2: Generierte GUI des Inhalts- und Verweisebaums

Um die Explorer-GUI zu generieren, müssen diese beiden Bäume zusammen mit einem austauschbaren Bereich in einer GUI-Beschreibung definiert werden. Ein DSL-Skript, welches dies umsetzt, sieht wie folgt aus (siehe Listing 9.8).

Listing 9.8: DSL-Skript für das Exlporer-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 9.9).

Listing 9.9: Klassenkopfgenerierung für ein Window

```
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 von der Layout-Datei geliefert werden.
Die GUI-Klassen der eingebundenen Gui-Beschreibungen der 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 UI-Komponente und die Einbindung in einen Layout-Container nötig (siehe Listing 9.10). Anderenfalls müssen die spezifischen Eigenschaften komplexer Komponenten untersucht werden. Darauf wird in dieser Arbeit jedoch nicht weiter eingegangen.

Listing 9.10: Übersetzung eingebundener GUI-Beschreibungen

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

Die letzte Komponente, die damit noch nicht in der GUI-Klasse deklariert wurde ist die Interchangeable-Komponente. Da diese UI-Komponenten keine speziellen Properties besitzt, ist die Methode zur Generierung des Quell-Code recht einfach gehalten (siehe Listing 9.11).

Listing 9.11: Generierung der 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 im Anhang **??** zu finden. Es handelt sich um die Dateien *GuiInhaltsbaum.java*, *GuiVerweisebaum.java* und *GuiExplorer.java*. Abbildung 9.3 zeigt das GUI, welches durch die Klasse *GuiExplorer* erzeugt wird.

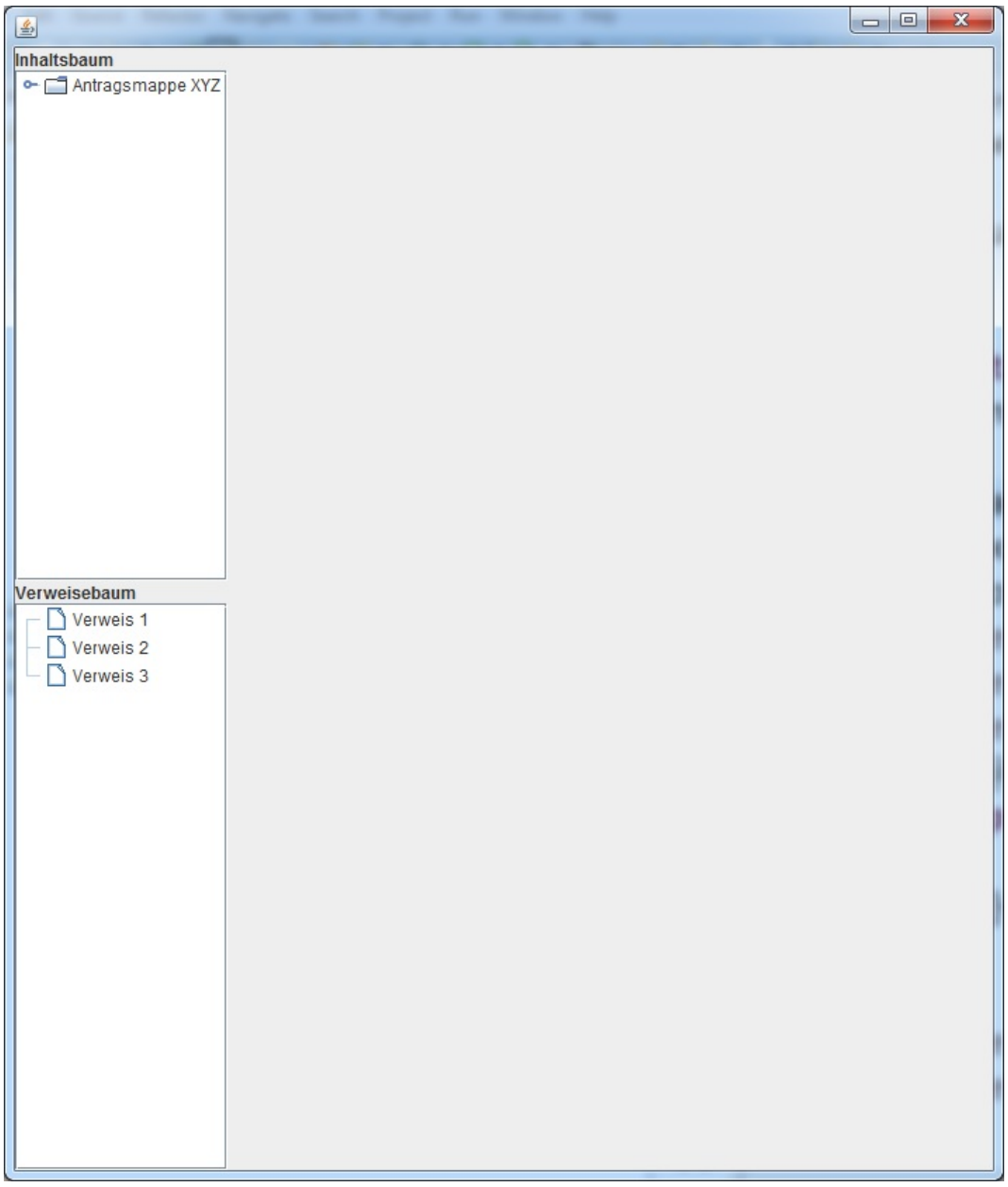

Abbildung 9.3: Generierte Explorer-GUI

FP-Klassen

Da für die FP-Klassen lediglich der Klassenrumpf und der Konstruktor generiert wird, ist diese Aufgabe entsprechend einfach. Es werden wiederum eine Oberklasse und entsprechende Importe benötigt, wie Listing 9.12 zu

entnehmen ist.

Listing 9.12: Generierung der FP-Klasse

```
def genFpSource() '''
        «addImport("import DE.data_experts.jwam.tools.FpObject;")»
        «addImport("import de.jwam.handling.toolconstruction.request.
            RequestHandler;")»
        public class «specificFilename» extends FpObject{
                 public «specificFilename»( RequestHandler parent ) {
                        super( parent, "«descriptionname»" );
                }
        }
'''
```

IP-Klassen

Um ein Objekt der IP-Klasse zu instanziieren wird ein Objekt der dazugehörigen FP-Klasse benötigt. Die Generierung der IP-Klasse beginnt wiederum mit der Generierung des Klassenkopfes mit entsprechend benötigten Imports (siehe Listing 9.13).

Listing 9.13: Generierung des Klassenkopfs der IP-Klasse

```
def genIpSource(UIDescription description)'''
        «addImport("import DE.data_experts.jwam.tools.IpObject;")»
        «addImport("import de.jwamx.technology.iafpf.guimanagement.IAFContext;")»
        «addImport("import DE.data_experts.util.degexception.ExceptionManager;")»
        public class «specificFilename» extends IpObject{
                public «specificFilename»( IAFContext iafContext, final
                    Fp«descriptionname» fp) {
                super( iafContext, fp );
                «addGlobalVar("Fp" + descriptionname + ' fp;')»
                        this.fp = fp;
                        try {
                        initCommands();
                            initIAFs( iafContext );
                    }
                catch ( Exception ex ) {
                        ExceptionManager.getManager().addAndShow( ex );
                }
                }
                «description.genIAF»
                «description.genCommands»
                «description.genCommandMethods»
                «genGlobalVars»
        }
'''
```

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 9.14), den Kommandos bestimmte Methoden zugeordnet (*genCommands* - siehe Listing 9.17) und die Rümpfe für diese Methoden generiert (*genCommandMethods* - siehe Listing 9.18).
Bei der Bestimmung der Interaktionsformen werden die in der GUI-Beschreibung definierten UI-Komponenten durchlaufen und deren Standard-Interaktionsformen und spezielle Interaktionsformen übersetzt (siehe Listing **??**).

Listing 9.14: Übersetzung 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).
                      compileLabelStandardIAF»
12        «ELSEIF definition.concreteDefition.name == 'Tree'»
13                «(definition.concreteDefition as TreeDefinition).
                      compileTreeStandardIAF»
14        «ENDIF»
15        «««             Spezielle Interaktionsformen
16                «definition.compileSpecificInteractionTypes»
17        '''
```

Für die Standard-Interaktionsformen unterschieden werden um welche UI-Komponente es sich handelt.
Zu jeder Interaktionsform gibt es ein entsprechendes Kommando, welches zur Generierung des Quell-Codes relevant ist. Die Zuordung von Kommando zu Interaktionsform wird vom Generator vorgenommen (Beipsiel Tree - siehe Listing 9.15). Die dafür verwendeten Methoden (z.B. *genIAFActivator* oder *genIAFTree*) können häufig wiederverwendet werden.

Listing 9.15: Übersetzung der Standard-Interaktionsform von Trees

```
1 def compileTreeStandardIAF(TreeDefinition definition)'''
2                 «definition.id.genIAFActivator»
3                 «definition.id.genIAFTree»
```

```
4         '''
5
6 def genIAFTree(String id) '''
7                 «genIAFSource("DE.data_experts.jwam.gui.interaction.IfTree","de.
                      jwamx.technology.iafpf.command.cmdSelect",id)»
8         '''
9
10 def genIAFActivator(String id) '''
11                «genIAFSource("de.jwamx.technology.iafpf.interaction.ifActivator",
                       "de.jwamx.technology.iafpf.command.cmdActivate", id)»
12        '''
```

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 9.16).

Listing 9.16: 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(
              «iafNameWithPrefix».class, "«id»" );
7         «IF !commandSource.equals("")»
8                 «addImport("import "+ commandSource +  ";")»
9                 «var commandNameWithPrefix = commandSource.getClassOfSource»
10                «var commandName = commandNameWithPrefix.substring(3)»
11                «id + iafName».attach«commandName»Command( «id +
                      commandName»Command );
12                «addCommand(id, commandName)»
13        «ENDIF»
14 '''
15
16 def addCommand(String id, String commandName) {
17        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. Namensgebund in der deg getroffen wurden. Der Aufruf der Methode *addCommand* ist für weitere Generierungen von Belang. Nach Abschluss der Generierung des Quell-Codes 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 9.16) gefüllt wird. Die Methode *genCommands* ist für die beschriebene Zuordnung zuständig (siehe Listing 9.17).

Listing 9.17: 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
                              " + commandName + ";")»
6                         «id»Command = new CmdAusfuehrer«commandName»(
                              getAusfuehrer() ) {
7                         @Override
8                         public void ausfuehren() {
9                                 «id»();
10                        }
11                }};
12                «ENDFOR»
13        }
14 '''
```

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 Quell-Code verwendet werden, ebenfalls generiert werden. Dafür ist die Methode *genCommandMethods* zuständig. In dieser Methode wird die Liste *commands* nochmals benutzt, um die Methodenrümpfe zu erzeugen (siehe Listing 9.18). Die Implementierung der generierten Methoden muss der Entwickler übernehmen.

Listing 9.18: 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 im Anhang **??** zu finden. In den Methoden zur Generierung von UI-Komponenten für die GUI-Klassen sind die Codezeilen, die das Layout bestimmen, entsprechend gekennzeichnet.

Kapitel 10

Zusammenfassung und Ausblick

Durch die Generierung der GUI-, IP- und FP-Klassen muss die Routinearbeit R1 (siehe Kapitel 2) nicht mehr durchgeführt werden. Die Generierung von Tabellen in Hinblick auf Vermeidung der Routinearbeit R2 (siehe Kapitel 2) konnte aus Zeitgründen nicht mehr umgesetzt werden. Es ist jedoch auf Basis der gezeigten Generierungen davon auszugehen, dass den Entwicklern auch diese Arbeit vom Generator abgenommen werden kann. Dafür bedarf es jedoch die Umsetzung und Einbindung der Layout-Datei.
Bezüglich der Sprache kann die Anforderung *weniger LOC zur Beschreibung von UIs* (siehe Kapitel 5) erst an dieser Stelle als umgesetzt betrachtet werden. Zum Vergleich sind die generierten Klassen und die DSL-Skripte auf deren Basis jene erzeugt wurden im Anhang **??** zu finden. Die DSL-Skripte umfassen dabei lediglich 13 Codezeilen. Die daraus generierten Klassen umfassen mehr als 200 LOC.
Test und Ausführung

Einbettung in den Entwicklungsprozess

protected regions

Generierung Eclipse Editor

Anhang

Grammatiken

1. Iteration

```
1 UIDescription:
2         areaCount=AreaCount
3         typeDefinition=(TypeDefinition)
4         usedDescriptions+=(UsedDescriptions)* &
5         inputTypes+=(inputType)* &
6         definitions+=(Definition)*
7         areas+=(AreaAssignment)*;
8
9 inputType:
10        'inputType=' type=STRING ' as ' name=STRING;
11
12 UsedDescriptions:
13        'use: ' description=UIDescriptionImport;
14
15 AreaCount:
16        'Area count: ' areaCount=INT;
17
18 Definition:
19        'DEF ' concreteDefition=ComponentDefinition 'END DEF';
20
21 TypeDefinition:
22        'type: ' type=TYPE;
23
24 TYPE:
25        ('WINDOW' | 'INNERCOMPLEX');
26
27 UIDescriptionImport:
28        descriptionName=STRING (' As: ' localName=STRING)?;
29
30 AreaAssignment:
```

```
31          'Area:' area=INT '<-' element=STRING
32          | element=STRING '->' 'Area:' area=INT ;
33
34 ComponentDefinition:
35          LabelDefinition | ButtonDefinition | MultiSelectionDefinition;
36
37 MultiSelectionDefinition:
38          type='MultiSelection' ' as ' name=STRING':' ('inputType=' inputType=STRING
39          ('selectableValues=' selectableValuesLocation=STRING ('selectedValues='
                selectedValuesLocation=STRING)?)?)?;
40
41 ButtonDefinition:
42          type='Button' ' as ' name=STRING ':'
43          ('text=' text=STRING)?
44          ('interaction=' interaction=Interaction)?;
45
46 Interaction:
47          name=STRING ' type=' Interactiontype ' with actions:' actions+=(UIAction)
                *;
48
49 LabelDefinition:
50          type='Label' ' as ' name=STRING ':'
51          ('text=' text=STRING)?;
52
53
54 UIAction:
55          'type=' type='UiAction'
56          'element=' uiElementName=STRING ':'
57          properties+=(Property)*;
58
59 Property:
60          CommonProperty;
61
62 CommonProperty:
63          (name=CommonPropertyType '=' value=STRING);
64
65 CommonPropertyType:
66          'Text';
67
68 Interactiontype:
69          'CLICK';
```

2. Iteration

```
1 UIDescription:
2          areaCount=AreaCount
3          typeDefinition=(TypeDefinition)
4          (property=(Property))?
5          usedDescriptions+=(UsedDescriptions)*
6          refinements+=(Refinement)*
```

```
7         definitions+=(Definition)*
8         areas+=(AreaAssignment)*;
9
10 Refinement:
11        'REFINE' concreteRefinement=ComponentRefinement 'END REFINE';
12
13 ComponentRefinement:
14        LabelRefinement | ButtonRefinement;
15
16 ButtonRefinement:
17        type='Button' ' with name: ' name=STRING
18        properties=(Properties)?;
19
20 LabelRefinement:
21        type='Label' ' with name: ' name=STRING
22        properties=(Properties)?;
23
24 Property:
25        'get properties from:' propertiesFile=STRING;
26
27 UsedDescriptions:
28        'use: ' description=DescriptionType (' as: ' localName=STRING)?;
29
30 DescriptionType:
31        UIDescriptionImport | ComplexComponent;
32
33 AreaCount:
34        'Area count: ' areaCount=INT;
35
36 Definition:
37        'DEF ' concreteDefition=ComponentDefinition 'END DEF';
38
39 TypeDefinition:
40        'type: ' type=Type;
41
42 Type:
43        ('WINDOW' | 'INNERCOMPLEX');
44
45 UIDescriptionImport:
46        descriptionName=(STRING);
47
48 ComplexComponent:
49        (Multiselection);
50
51 Multiselection:
52        descriptionName='Multiselection' ('<' inputType=STRING '>')?;
53
54 AreaAssignment:
55        'Area:' area=INT '<-' element=STRING
56        | element=STRING '->' 'Area:' area=INT;
57
```

```
58 ComponentDefinition:
59         LabelDefinition | ButtonDefinition;
60
61 ButtonDefinition:
62         type='Button' ' as ' name=STRING
63         properties=(Properties)?;
64
65 Properties:
66         ':' ('propertyKey=' propertyKey=STRING)?
67         ('text=' text=STRING)?
68         ('interactiontype=' interactiontype+=(Interactiontype)+)?;
69
70 LabelDefinition:
71         type='Label' ' as ' name=STRING
72         properties=(Properties)?;
73
74 Interactiontype:
75         'Click' | 'ChangeText';
76
77 terminal WS:
78         (' ' | '\t' | '\r' | '\n' | ',')+;
```

3. Iteration

Listing 1: Grammatik der 3. Iteration

```
1
2 grammar org.deg.xtext.gui.GuiDSL with org.eclipse.xtext.common.Terminals
3
4 generate guiDSL "http://www.deg.org/xtext/gui/GuiDSL"
5
6 UIDescription:
7         typeDefinition=(TypeDefinition)
8         (property=(Property))?
9         (layout=(Layout))?
10        usedDescriptions+=(UsedDescriptions)*
11        refinements+=(Refinement)*
12        definitions+=(Definition)*
13        structure=Structure;
14
15 TypeDefinition:
16        'type: ' type=Type ';';
17
18 Type:
19        id=('WINDOW' | 'INNERCOMPLEX');
20
21 Property:
22        'get properties from:' propertiesFiles+=(STRING)+ ';';
23
24 Layout:
```

```
25          'get layout from:' layoutFiles+=(STRING)+ ';';
26
27 UsedDescriptions:
28          'use:' description=DescriptionType ('as:' id=STRING)? ';';
29
30 DescriptionType:
31          UIDescriptionImport | ComplexComponent;
32
33 ComplexComponent:
34          Multiselection;
35
36 UIDescriptionImport:
37          descriptionName=(STRING);
38
39 Refinement:
40          concreteRefinement=ComponentRefinement ';';
41
42 ComponentRefinement:
43          LabelRefinement | ButtonRefinement | TextfieldRefinement |
                TextareaRefinement | TableRefinement | TabViewRefinement |
44          TreeRefinement | InterchangeableRefinement;
45
46 Definition:
47          concreteDefition=ComponentDefinition ';';
48
49 ComponentDefinition:
50          LabelDefinition | ButtonDefinition | TextfieldDefinition |
                TextareaDefinition | TreeDefinition | TableDefinition |
51          TabViewDefinition | InterchangeableDefinition;
52
53 Structure:
54          name="Structure" ":" orderedElements+=(Element)* ';';
55
56 Element:
57          id=STRING;
58
59 TableRefinement:
60          name='Table' (inputType=INPUT_DESCRIPTION)? 'change:' id=STRING
61          (properties=(CommonProperties))?;
62
63 TabViewRefinement:
64          name='TabView' (tabs+=(TabDefinition)+)? 'change:' id=STRING
65          (properties=(CommonProperties))?;
66
67 TreeRefinement:
68          name='Tree' (inputType=INPUT_DESCRIPTION)? 'change:' id=STRING
69          (properties=(CommonProperties))?;
70
71 InterchangeableRefinement:
72          name='Interchangeable' 'change:' id=STRING
73          (properties=(CommonProperties))?;
```

```
74
75 TextareaRefinement:
76         name='Textarea' 'change:' id=STRING
77         (properties=(TextareaProperties))?;
78
79 TextfieldRefinement:
80         name='Textfield' 'change:' id=STRING
81         (properties=(TextfieldProperties))?;
82
83 ButtonRefinement:
84         name='Button' 'change:' id=STRING
85         (properties=(ButtonProperties))?;
86
87 LabelRefinement:
88         name='Label' 'change:' id=STRING
89         (properties=(LabelProperties))?;
90
91 TabViewDefinition:
92         name='TabView' tabs+=(TabDefinition)* 'as:' id=STRING
93         (properties=(CommonProperties))?;
94
95 TabDefinition:
96         name=TABNAME;
97
98 Multiselection:
99         name='Multiselection' (inputType=INPUT_DESCRIPTION)?;
100
101 TextfieldDefinition:
102         name='Textfield' 'as:' id=STRING
103         (properties=(TextfieldProperties))?;
104
105 TextareaDefinition:
106         name='Textarea' 'as:' id=STRING
107         (properties=(TextareaProperties))?;
108
109 TreeDefinition:
110         name='Tree' (inputType=INPUT_DESCRIPTION)? 'as:' id=STRING
111         (properties=(CommonProperties))?;
112
113 TableDefinition:
114         name='Table' (inputType=INPUT_DESCRIPTION)? 'as:' id=STRING
115         (properties=(CommonProperties))?;
116
117 InterchangeableDefinition:
118         name='Interchangeable' 'as:' id=STRING
119         (properties=(CommonProperties))?;
120
121 ButtonDefinition:
122         name='Button' 'as:' id=STRING
123         (properties=ButtonProperties)?;
124
```

```
125 LabelDefinition:
126         name='Label' 'as:' id=STRING
127         (properties=(LabelProperties))?;
128
129 Interactiontype:
130         id=('IfActivator' | 'IfTextDisplay' | 'IfViewImage');
131
132 TextareaProperties:
133         commonProperties=CommonProperties
134         ('text=' text=STRING)?
135         ('editable=' editable=BOOLEAN)?;
136
137 TextfieldProperties:
138         commonProperties=CommonProperties
139         ('text=' text=STRING)?
140         ('editable=' editable=BOOLEAN)?;
141
142 LabelProperties:
143         commonProperties=CommonProperties
144         ('text=' text=STRING )?;
145
146 ButtonProperties:
147         commonProperties=CommonProperties
148         ('text=' text=STRING)?;
149
150 CommonProperties:
151         '->'
152         ('propertyKey' '=' propertyKey=STRING)?
153         ('layoutKey' '=' layoutKey=STRING)?
154         ('interactiontype' '=' interactiontype+=(Interactiontype)+)?;
155
156 terminal WS:
157         (' ' | '\t' | '\r' | '\n' | ',')+;
158
159 terminal TABNAME:
160         '[' (!('[' | ']'))* ']';
161
162 terminal INPUT_DESCRIPTION:
163         '<' (!('<' | '>'))* '>';
164
165 terminal BOOLEAN:
166         'TRUE' | 'FALSE';
```

Glossar

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

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

. 1

. 5

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

. 1

. 1

. 8, 9

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

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

Literaturverzeichnis

[114] *Xtext Documentation*. URL: http://www.eclipse.org/Xtext/cumentation/2.6.0/Xtext Documentation.pdf, September 2014. Zuletzt eingesehen am 24.11.2014.

[Aho08] AHO, ALFRED V: *Compiler: Prinzipien, Techniken und Werkzeuge*. Pearson Studium, 2008.

[BCK08] BRAUER, JOHANNES, CHRISTOPH CRASEMANN und HARTMUT KRASEMANN: *Auf dem Weg zu idealen Programmierwerkzeugen - Bestandsaufnahme und Ausblick.* Informatik Spektrum, 31(6):580–590, 2008.

[BPL13] BACIKOVÁ, MICHAELA, JAROSLAV PORUBÄN und DOMINIK LAKATOS: *Defining Domain Language of Graphical User Interfaces*. In: *OASIcs-OpenAccess Series in Informatics*, Band 29. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

[Bra03] BRAUER, JOHANNES: *Grundkurs Smalltalk-Objektorientierung von Anfang an*. Vieweg+ Teubner Verlag, 3, 2003.

[bra10] *Eine DSL für Harel-Statecharts mit PetitParser*. Arbeitspapiere der Nordakademie. Nordakad., 2010.

[dat14] DATA EXPERTS GMBH: *Förderantrag*. Profil Wiki der deg, März 2014. Zuletzt eingesehen am 02.12.2014.

[deG07] GMBH DATA EXPERTS: *Detailkonzept ELER/i-Antragsmappe*, Januar 2007. Letzte Änderung am 01.12.2014.

[DM14] DANIEL, FLORIAN und MARISTELLA MATERA: *Model-Driven Software Development*. In: *Mashups*, Data-Centric Systems and Applications, Seiten 71–93. Springer Berlin Heidelberg, 2014.

[FP11] FOWLER, MARTIN und REBECCA PARSON: *Domain-Specific Languages*. Addison-Wesley, 2011.

[Gal07] GALITZ, WILBERT O.: *The Essential Guide to User Interface Design - An Introduction to GUI Design Principles and Techniques*. John Wiley Sons, New York, 2007.

[gho11] *DSLs in Action*. Manning Publications Co., 2011.

[Gun13] GUNDERMANN, NIELS: *Prototypische Implementierung eines JavaFX-Channels zur Integration ins MulitChannel-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. Online verfügbar URL:http://swt-www.informatik.uni-hamburg.de/uploads/media/DA_StefanHofer.pdf . Zuletzt eingesehen am 09.01.2015.

[KB11] KRISHNASWAMI, NEELAKANTAN R. und NICK BENTON: *A Semantic Model for Graphical User Interfaces*. Microsoft Research, September 2011. Verfügar unter URL:.

[LW] LU, XUDONG und JIANCHENG WAN: *Model Driven Development of Complex User Interface*. Technischer Bericht, Shandong University. Verfügar unter URL: http://ceur-ws.org/Vol-297/paper7.pdf.

[mds06] *Model-Driven Software Development*. John Wiley Sons Ltd, Februar 2006.

[MHP99] MYERS, BRAD, SCOTT E. HUDSON und RANDY PAUSCH: *Past, Present and Future of User Interface Software Tools*. Technischer Bericht, Carnegie Mellon University, September 1999. Verfügar unter URL: http://www.cs.cmu.edu/ amulet/papers/futureofhci.pdf.

[ML09] MARKUS VOELTER und LARS CORNELIUSSEN: *Carpe Diem*. DotNetPro, 5 2009.

[Oraa] ORACLE: *Class GridBagLayout*. URL: https://docs.oracle.com/javase/7/docs/api/java/awt/GridBagLayout.html. Zuletzt eingesehen am 02.12.2014.

[Orab] ORACLE: *Swing*. URL: https://docs.oracle.com/javase/jp/8/technotes/guides/swing/index.html. Zuletzt eingesehen am 02.12.2014.

[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 VOELTER: *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. Online verfügbar unter URL: http://bergel.eu/download/Dyla2010/dyla10_submission_4.pdf Zuletzt eingesehen am 6.1.2015.

[Roa09] ROAM, DAN: *The Back of the Napkin (Expanded Edition) - Solving Problems and Selling Ideas with Pictures*. Penguin, New York, Expanded Auflage, 2009.

[Sau10] SAUER, JOACHIM: *Architekturzentrierte agile Anwendungsentwicklung in global verteilten Projekten*. Doktorarbeit, Universität Hamburg, 2010. Online verfügbar URL:http://ediss.sub.uni-hamburg.de/volltexte/2011/4959/pdf/Dissertation_Sauer.pdf Zuletzt eingesehen am 08.01.2015.

[Sch05] SCHMELZER, ROBERT FRANZ: *Realisierung teilautomatisierter Prozesse durch die Kombination von Ablaufsteuerung und unterstützter Kooperation*. Diplomarbeit, FH Oberösterreich, 2005. Online verfügbar URL:http://www.schmelzer.cc/Downloads/Files/ Diplomarbeit_Schmelzer.pdf . Zuletzt eingesehen am 09.01.2015.

[SdS03] SAUER, JOACHIM und AUSGEWÄHLTE THEMEN DER SOFTWARETECHNIK: *Gestaltung von Anwendungssoftware nach dem WAM-Ansatz auf mobilen Geräten*. Diplomarbeit, Fachbereich Informatik, Universität Hamburg, 2003. Online verfügbar URL:http://swt-www.informatik.uni-hamburg.de/uploads/media/Diplomarbeit_Joachim_Sauer.pdf . Zuletzt eingesehen am 09.01.2015.

[SKNH05] SUKAVIRIYA, NOI, SANTHOSH KUMARAN, PRABIR NANDI und TERRY HEATH: *Integrate Model-driven UI with Business Transformations: Shifting Focus of Model-driven UI*. Technischer Bericht, IBM T.J. Watson Research Center, Oktober 2005. Verfügar unter URL: http://www.research.ibm.com/people/p/prabir/MDDAUI.pdf.

[Ste07] STECHOW, DIRK: *JWAMMC - Das Multichannel-Framework der data-experts gmbh*. Vortrag, Dezember 2007.

[Use12] USERLUTIONS GMBH: *3 Gründe, warum gute Usability wichtig ist*. URL: http://rapidusertests.com/blog/2012/04/3-gute-grunde-fuer-usability-tests/, April 2012. Zuletzt eingesehen am 01.12.2014.

[VBK+13] VÖLTER, MARKUS, SEBASTIAN BENZ, LENNART KATS, MATS HELANDER, EELCO VISSER und GUIDO WACHSMUTH: *DSL Engineering*. CreateSpace Independent Publishing Platform, 2013.

[Vol11] VOLTER, MARKUS: *From programming to modeling-and back again*. Software, IEEE, 28(6):20–25, 2011.