



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

in der Fachrichtung
Wirtschaftsinformatik

T H E M A

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

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

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

Neubrandenburg, Februar 2015

Niels Gundermann

Inhaltsverzeichnis

Abbildungsverzeichnis	V
Tabellenverzeichnis	VII
Listings	VIII
1. Motivation	1
2. Problembeschreibung und Zielsetzung	3
2.1. Allgemeine Anforderungen an Benutzeroberflächen von pro- fil c/s	3
2.2. Umsetzung der Benutzerschnittstellen für mehrere Plattfor- men in der data experts GmbH (Ist-Zustand)	4
2.2.1. Multichannel-Framework	4
2.2.2. JWAM	6
2.3. Probleme des Multichannel-Frameworks	8
2.4. Zielsetzung	9
3. Domänenspezifische Sprachen	10
3.1. Begriffsbestimmungen	10
3.2. Anwendungsbeispiele	14
3.3. Model-Driven Software Development (MDSD)	14
3.4. Abgrenzung zu GPL	15
3.5. Vor- und Nachteile von DSLs gegenüber GPLs	16
3.6. Interne DSLs	22
3.6.1. Implementierungstechniken	23
3.7. Externe DSLs	25
3.7.1. Implementierungstechniken	25

3.8. Nicht-textuelle DSLs	29
4. GUI-DSL	30
4.1. Motivation des Ansatzes	30
4.2. Anforderungen an die GUI-DSL	31
5. Entwicklung einer Lösungsidee	33
5.1. Allgemeine Beschreibung der Lösungsidee	33
5.2. Konzept	33
6. Evaluation des Frameworks zur Entwicklung der DSL	35
6.1. Vorstellung ausgewählter Frameworks	35
6.1.1. PetitParser	35
6.1.2. Xtext	36
6.1.3. Meta Programming System	36
6.2. Vergleich und Bewertung der vorgestellten Frameworks	37
7. Grobkonzept für die Entwicklung des Prototyps	40
7.1. Vorgehensmodell	40
7.2. Konzeption der DSL-Umgebung (Vision)	42
7.2.1. 1. Iteration	42
7.2.2. 2. Iteration	46
7.2.3. 3. Iteration	48
8. Entwicklung einer DSL zur Beschreibung der GUI in profil c/s	50
8.1. 1. Iteration	50
8.2. 2. Iteration	56
8.3. 3. Iteration	63
9. Entwicklung des Generators zur Generierung von Klassen für das Multichannel-Framework	72
9.1. Beschreibung der GUI-, FP- und IP-Klassen	73
9.2. Umsetzung des frameworkspezifischen Generators	74
10. Zusammenfassung und Ausblick	85
A. Zuwendungsblatt für den Web- und Standalone-Client	X

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

Abbildungsverzeichnis

1.	Architektur des Multichannel-Frameworks	5
2.	Komplexes Werkzeug mit Benutzt-Beziehung	7
3.	Die grundlegende Idee hinter dem MDSD	15
4.	Parsen allgemein	26
5.	Funktionsweise von Parserkombinatoren	28
6.	Grundlegendes Konzept	34
7.	Grundlegende Idee für den Prototyp	34
8.	Inkrementelles Modell	40
9.	1. Iteration: Konzeption der DSL-Umgebung	42
10.	Beispiel: Suchfeld für Name	44
11.	Beispiel: Suchmaske	45
12.	45
13.	46
14.	46
15.	46
16.	2. Iteration: Konzeption der DSL-Umgebung	47
17.	3. Iteration: Konzeption der DSL-Umgebung	48
18.	1. Iteration: UIDescription	53
19.	1. Iteration: UsedUIDescription	53
20.	1. Iteration: ComponentDefinition	54
21.	1. Iteration: Interaction	54
22.	55
23.	2. Iteration: UIDescription	58
24.	2. Iteration: Property und Properties	58

25.	2. Iteration: UsedDescription	59
26.	2. Iteration: Definition	60
27.	2. Iteration: Refinement	60
28.	3. Iteration: UIDescription	66
29.	3. Iteration: Definition	67
30.	3. Iteration: Button	67
31.	3. Iteration: TabView	68
32.	3. Iteration: Tree	68
33.	3. Iteration: Table	69
34.	Aufbau eines Explorers	72
35.	Generierte GUI des Inhalts- und Verweisebaums	78
36.	Standalone-Client: Zuwendungsblatt	X
37.	Web-Client: Zuwendungsblatt	XI
38.	Multiselection-Komponente	XII
39.	Generierte Explorer-GUI	XIII

Tabellenverzeichnis

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

Listings

1.	Beispiel: Fluent Interface	24
2.	1. Iteration: Syntax	55
3.	2. Iteration: Properties	61
4.	2. Iteration: Interaktion	61
5.	2. Iteration: Definition komplexer Komponenten	61
6.	2. Iteration: Button und Label	61
7.	2. Iteration: Area-Zuweisung	62
8.	2. Iteration: Verändern von Komponenten	62
9.	2. Iteration: Verändern von Komponenten mit gleichen Namen	63
10.	3. Iteration: Properties- und Layout-Dateien	69
11.	3. Iteration: Eingebundene GUI-Komponenten	69
12.	3. Iteration: Button und Label	70
13.	3. Iteration: Textfield und Textarea	70
14.	3. Iteration: Table und Tree	70
15.	3. Iteration: TabView	71
16.	3. Iteration: TabView	71
17.	3. Iteration: Struktur	71
18.	GUI-Skript für den Inhaltsbaum	74
19.	GUI-Skript für den Verweisebaum	74
20.	Speichern der Importe und der globalen Variablen	75
21.	Generierung eines Innercomplex	75
22.	Generierung der init-Methode der GUI-Klassen	76
23.	Generierung eines Labels	76
24.	Generierung eines Trees	77
25.	GUI-Skript für die Explorer-GUI	78

26.	Generierung eines Windows	78
27.	Generierung der Einbindung anderer GUI-Skripte	79
28.	Generierung einer Interchangeable-Komponente	79
29.	Generierung der FP-Klasse	80
30.	Generierung der IP-Klasse	80
31.	Generierung der Interaktionsformen	81
32.	Generierung der Standardinteraktionsformen von Trees	82
33.	Generierung einer Interaktionsform	82
34.	Generierung der Kommandoinitialisierung	83
35.	Generierung der Aktionen bei einer Interaktion	83

1. Motivation

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

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

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

Daraus ergibt sich die Überlegung, ob diese beiden Ansätze zur Implementierung von *GUIs* (*traditionell* und *Model-Driven*) verbunden werden kön-

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

2. Problembeschreibung und Zielsetzung

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

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

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

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

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

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

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

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

Tabelle 1.: Prioritäten der Anforderungen an die GUI

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

2.2.1. Multichannel-Framework

Die Clients werden in der Programmiersprache *Java* entwickelt. Für die Realisierung des *Standalone-Clients* wird in der *data experts GmbH* das Framework *Swing* verwendet. Für den *Web-Client* wird auf *wingS* zurückgegriffen. Um eine Vorstellung des Ist-Zustands zu vermitteln, sind in Abbildung 36 und in Abbildung 37 (siehe Anhang A) die GUIs eines *Zuwendungsblatts* und eines *Förderantrags* für den *Web-Client* und den *Standalone-Client* abgebildet.

In diesen GUIs ist nur ein bestimmter Teil für die fachlichen Informationen relevant. Dies sind lediglich die Tabelle, die darunter stehenden Schaltflächen sowie das Bemerkungsfeld (im *Web-Client* auf der rechten Seite und im *Standalone-Client* in der Mitte). Dieser Bereich der GUI ist in beiden Clients gleich aufgebaut. Andere Teile der GUI haben derzeit keinen einheitlichen Aufbau, was den unterschiedlichen Frameworks für die Umsetzung von *Web-* und *Standalone-Client* geschuldet ist.

Dass der Aufbau der GUI in beiden Clients ähnlich ist, liegt an der Umset-

zung der *GUI*, die im Folgenden erläutert wird.

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

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

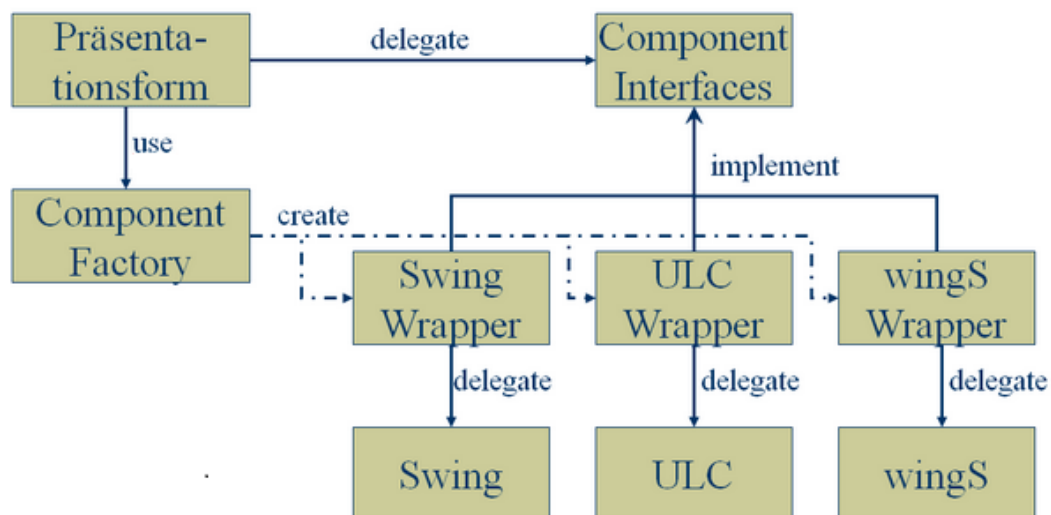


Abbildung 1.: Architektur des Multichannel-Frameworks (vgl. [Ste07])

Aus *Präsentationsformen* können mithilfe der *Component-Factories* *GUIs* erzeugt werden, die auf unterschiedlichen Frameworks basieren. Die *GUI-Komponenten* müssen dafür das *Component-Interface* implementieren. (Bei den verwendeten Frameworks handelt es sich um *Swing*, *ULC* und *wingS*. *ULC* wird in der *data experts GmbH* nicht mehr eingesetzt.) Das *Component-Interface* wird für die Interaktion mit den Komponenten der unterschiedlichen Frameworks benötigt. Mit dem *MCF* ist die *data experts GmbH* in der Lage ihre *GUIs* für *Swing* und für *wingS* mit nur einer *GUI-Beschreibung* zu erzeugen.

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

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

2.2.2. JWAM

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

Die Bezeichnung WAM steht für *Werkzeug & Material-Ansatz*. Es handelt sich dabei um einen Ansatz zur Softwarearchitektur, welcher den anwendungsorientierten Ansatz der Softwareentwicklung fördert. Der Benutzer der Software steht im Mittelpunkt, wodurch die Gestaltung der Funktionalitäten, Benutzerschnittstellen und die Schnittstelle des Entwicklungs- und Implementierungsprozesses beeinflusst werden (vgl. [Sch05, S.13]). Weiterhin werden Entwurfsmetaphern verwendet, die den Entwicklern und Anwendern das Entwickeln und Verstehen der Software vereinfachen sollen (vgl. [Sau03, S.30]). Diese Entwurfsmetaphern beschreiben „[...] *Elemente und Konzepte der Anwendung durch bildhafte Vorstellungen von realen Gegenständen* [...]“ [Sau03, S.30]. Die grundlegenden Metaphern werden im Folgenden kurz erläutert.

Ein *Material* kann nicht direkt vom Nutzer bearbeitet werden. Es besitzt jedoch eine Schnittstelle, die fachliche Operationen erlaubt. Diese Operationen können bspw. von einem *Werkzeug* aufgerufen werden, um den Zustand des *Materials* zu verändern. Dabei verfügen *Werkzeuge* über eine Präsentation und geben somit eine Handhabung vor (vgl. [Sau03, S.30]).

Es gibt zwei Arten von *Werkzeugen* - *monolithische* und *komplexe Werkzeuge* (vgl. [Hof06, S.5]). Da in der *data experts GmbH* hauptsächlich *komplexe Werkzeuge* Anwendung finden, werden die *monolithische Werkzeuge* in dieser Arbeit nicht weiter erläutert.

Komplexe Werkzeuge gliedern sich in Oberfläche, Interaktion und Fachlogik auf. Dabei muss sich die Funktionskomponente (*FunctionPart* - *FP*) vollständig von der *GUI* abstrahieren und sich somit auf die Fachlogik beschränken. Zwischen diesen Komponenten steht eine Interaktionskomponente (*InteractionPart* - *IP*), die für die Abstraktion Sorge trägt. Eine *Werkzeugklasse* umschließt diese drei Komponenten, wie Abbildung 2 zu entnehmen ist (vgl. [Hof06, S.5f]).

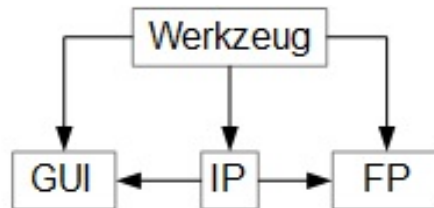


Abbildung 2.: Komplexes Werkzeug mit Benutz-Beziehung

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

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

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

2.3. Probleme des Multichannel-Frameworks

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

Ein weiteres Problem bezieht sich auf die integrierten Frameworks (*Swing* und *wingS*). Beide Frameworks sind veraltet und werden nicht mehr gewartet. (Analysen dazu wurden in [Gun13] und [Gun14b] durchgeführt.) Um auch in Zukunft den Anforderungen der Kunden nachkommen zu können, müssten beide Frameworks von den Entwicklern der *data experts GmbH* selbst weiterentwickelt werden. Eine andere Möglichkeit wäre es, andere und modernere Frameworks einzusetzen, um den nötigen Support der Entwickler des Frameworks nutzen zu können.

Das MCF ist in der Theorie so konzipiert, dass es leicht sein sollte, neue Frameworks zu integrieren (siehe Abbildung 1). In der Praxis wurde diese unkomplizierte Integration jedoch widerlegt. Ein Problem, welches bei der Integration neuer Frameworks aufkommt, ist, dass sich das MCF sehr stark an *Swing* orientiert und die *GUIs* vor allem vom *GridBagLayout* stark beeinflusst werden. Ein solcher Layoutmanager steht nicht in allen *GUI-Frameworks* zur Verfügung. Da die Beschreibung der *GUIs* über ein solches Layout vollzogen wird, ist der Umgang mit dem *GridBagLayout* innerhalb des Frameworks eine Voraussetzung für die Integration in das MCF.

Zusammenfassend sind folgende Probleme des *MCF* zu nennen:

P1 Verwendete Frameworks sind veraltet

P2 Starke Orientierung an *Swing*

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

R1 Vergebene Bezeichnungen für *GUI-Komponenten* müssen in unterschiedlichen Klassen (*IP-* und *GUI-Klassen*) gepflegt werden.

R2 Beim Erstellen von Tabellen müssen viele Methoden überschrieben werden.

R3 Die Werte für Aufschriften, Größeneinstellungen u. Ä. für *GUI-Komponenten* werden bei der *data experts GmbH* in *Properties-Dateien* festgehalten. Das Erstellen und Pflegen wird als fehleranfällig betrachtet.

2.4. Zielsetzung

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

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

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

3. Domänenspezifische Sprachen

3.1. Begriffsbestimmungen

Sprache/Programmiersprache

Formal betrachtet ist eine Sprache eine beliebige Teilmenge aller Wörter über einem Alphabet. „Ein Alphabet ist eine endliche, nicht leere Menge von Zeichen (auch Symbole oder Buchstaben genannt).“ [Hed12, S.6] Zur Verdeutlichung der Definition einer Sprache sei V ein Alphabet und $k \in \mathbb{N}$ (\mathbb{N} ist die Menge der natürlichen Zahlen einschließlich der Null) (vgl. [Hed12, S.6]). „Eine endliche Folge (x_1, \dots, x_k) mit $x_i \in V (i = 1, \dots, k)$ heißt Wort über V der Länge k “ [Hed12, S.6].

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

Konkrete Syntax

Die *konkrete Syntax* beschreibt die Notation der Sprache. Demnach bestimmt sie, welche Sprachkonstrukte der Nutzer einsetzen kann, um ein Programm in dieser Sprache zu schreiben (vgl. [Aho08, S.87]).

Abstrakte Syntax

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

Statische Semantik

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

Ausführungssemantik

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

General Purpose Language (GPL)

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

Domain Specific Language (DSL)

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

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

Es gibt verschiedene Arten von *DSLs*. Neben *internen* und *externen DSLs* (siehe Kapitel 3.6 und 3.7) findet eine Unterscheidung zwischen *technischen*

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

Grammatik

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

Lexikalische Analyse

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

Parser

Ein *Parser* ist ebenfalls ein Teil der *DSL-Umgebung* (vgl. [Gho11, S.211]). Er ist dafür verantwortlich, aus dem Ergebnis der *lexikalischen Analyse* ein Output zu generieren, mit dem weitere Aktionen durchgeführt werden können (vgl. [Gho11, S.212]). Der Output wird in Form eines *Syntaxbaums* (*AST*) generiert (vgl. [FP11, S.47]). Ein solcher Baum ist laut Martin Fowler et al. eine weitaus nutzbarere Darstellung dessen, was mit dem *DSL-Skript* beschrieben werden soll. Daraus lässt sich auch das *semantische Modell* generieren (vgl. [FP11, S.48]).

Semantisches Modell

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

Generator

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

1. Transformer Generation

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

2. Templated Generation

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

3.2. Anwendungsbeispiele

Die Anwendungsbereiche für *DSLs* sind breit gefächert. Die bekanntesten *DSLs* sind Sprachen wie *SQL* (zur Abfrage und Manipulation von Daten in einer relationalen Datenbank), *HTML* (als Markup-Sprache für das Web) oder *CSS* (als Layoutbeschreibung) (vgl. [Gho11, S.12]). Alle Sprachen sind in ihren Ausdrucksmöglichkeiten eingeschränkt und von der Abstraktion her direkt auf eine *Domäne* (jeweils dahinter in Klammern genannt) zugeschnitten (vgl. [Gho11, S.12f]).

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

3.3. Model-Driven Software Development (MDSD)

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

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

Daraus lassen sich folgende Ziele des *MDSD* ableiten:

- Schnelleres Entwickeln durch Automatisierungen
- Bessere Softwarequalität durch automatisierte Transformationen und formalen Modell-Definitionen

ne DSL hat diese Eigenschaft nicht. Da sie auf eine bestimmte *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 Domänenfokus als ein Schlüsselement der Definition einer DSL (vgl. [FP11, S.27f]).

3.5. Vor- und Nachteile von DSLs gegenüber GPLs

Vorteile:

Ausdruckskraft

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

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

Höhere Qualität

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

falls zu einem qualitativ höherwertigem Quellcode führt (vgl. [Gho11, S.21]).

Verbesserte Produktivität bei der Entwicklung der Software

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

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

Bessere Kommunikation mit Domänenexperten und Kunden

Aufgrund domänenspezifischer und präziser Ausdrücke, die in der Sprache verwendet werden, sind die Domänenexperten bzw. die Kunden vertrauter mit der Implementierung, als würde für die Umsetzung eine *GPL* verwendet werden (vgl. [VBK⁺13, S.42]). Die hohe Ausdruckskraft fördert das Verständnis dieser *DSL*. Damit ist es einfacher die Kunden in die Entwicklung mit einzubeziehen. Dabei sollten jedoch zusätzliche Hilfsmittel, wie Visualisierungen oder Simulationen verwendet werden (vgl. [FP11, S.34], [VBK⁺13, S.42]). Somit kann die oft vernachlässigte Kommunikation zwischen Kunden und Auftragnehmern verbessert werden. Martin Fowler et al. bezeichnen die Verwendung einer *DSL* als reine Kommunikationplattform als vorteilhaft (vgl. [FP11, S.34f]). Grund dafür ist, dass bereits bei der Entwicklung einer *DSL* das Verständnis des Auftragnehmers über die *Domäne*

gesteigert wird (vgl. [VBK⁺13, S.41]).

Plattformunabhängigkeit

Durch die Nutzung einer *DSL* kann bspw. ein Teil der Logik von der Kompilierung in den Ausführungskontext überführt werden. Die Definition der Logik findet dabei in der *DSL* statt, welche erst bei der Ausführung evaluiert wird. Ein solches Verfahren wird oft unter der Verwendung von *XML* genutzt (vgl. [FP11, S.35]). Dadurch ist es möglich die Logik auf unterschiedlichen Plattformen auszuführen (vgl. [VBK⁺13, S.43]). Dieser Vorteil ist besonders für den praktischen Teil dieser Arbeit interessant, allerdings weniger in Bezug auf Logik, denn in Bezug auf Benutzerschnittstellen.

Einfachere Validierung und Verifizierung

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

Unabhängigkeit von Technologien

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

Skalierung des Entwicklungsprozesses

Die Integration von neuen Mitarbeitern in ein Entwicklerteam fordert immer eine gewisse Einarbeitungszeit. Dieser Zeitraum kann durch die Nutzung einer *DSL* verkürzt werden, wenn die *DSL* einen hohen Abstraktionsgrad hat und dadurch leichter zu verstehen und zu erlernen ist (vgl. [Gho11, S.21]).

Innerhalb eines Entwicklerteams haben die Mitarbeiter oft einen unterschiedlichen Erfahrungsstand bzgl. einer speziellen Programmiersprache, die zur Entwicklung genutzt werden soll. Erfahrene Teammitglieder könnten sich mit der Implementierung der *DSL* befassen und die Grundlage für die anderen Teammitglieder schaffen. Diese wiederum nutzen die *DSL*, um die fachlichen Anforderungen der Kunden zu implementieren (vgl. [Gho11, S.21]). Das zu einer effizienteren Arbeitsweise, da sich nicht jeder Entwickler mit allem auskennen muss. Markus Völter et al. hingegen sehen die Teilung der Programieraufgaben als Gefahr bzw. Nachteil (vgl. [VBK⁺13, S.44]).

Nachteile:

Großes Know-How gefordert

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

Kosten für die Entwicklung der DSL

Bei wirtschaftlichen Entscheidungen wird der monetäre Input mit dem monetären Output verglichen. Investitionen führen dazu, dass der Input größer wird. Da eine *DSL* vor dem Einsatz zuerst entwickelt werden muss, ist es notwendig Investitionen für die Entwickler der *DSL* zu tätigen. Ob sich eine Investition lohnt, muss vorher durch entsprechende Analysen überprüft werden. Dabei muss festgestellt werden, ob die Entwicklung der *DSL* gerechtfertigt ist. Im Bereich der *technischen DSLs* fällt die Rechtfertigung einfach, da diese *DSLs* oft wiederverwendet werden können. *Fachliche DSLs* hingegen haben oft eine weitaus kompaktere Domäne, als eine *technische DSL*. Daher ergeben

sich die Möglichkeiten zur Wiederverwendung erst zu einem späteren Zeitpunkt und können nur schwer von der im Vorfeld durchgeführten Analyse wahrgenommen werden (vgl. [VBK⁺13, S.43]).

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

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

Investitionsgefängnis

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

Kakophonie

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

Normalerweise werden für eine Applikation mehrere *DSLs* benötigt. Diese unterschiedlichen *DSLs* haben i. d. R. unterschiedliche syntaktische Strukturen. Das führt dazu, dass Mitarbeiter unterschiedliche Sprachen beherrschen müssen. Das wiederum erfordert, dass die Entwickler öfter umdenken müssen, als würden sie fortwährend mit einer Sprache arbeiten. Dies macht den Entwicklungsprozess weitaus komplizierter (vgl. [FP11, S.37]).

Ghetto-Sprache

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

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

Borniertheit durch Abstraktion

Abstraktion ist von großer Wichtigkeit für eine *DSL*. Wenn ein Entwickler mit der Arbeit an einer *DSL* begonnen hat, hat dieser die Abstraktion in einem bestimmten Maß bereits festgelegt. Ein Problem tritt auf, wenn im Nachhinein etwas mit der Sprache beschrieben werden soll, dass nicht zu dieser Abstraktionsebene passt.

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

Kulturelle Herausforderungen

Die genannten Nachteile des Einsatzes von *DSLs*, führen zu Äußerungen wie der, dass die Entwicklung von Sprachen kompliziert ist, Domänenexperten keine Programmierer sind oder auch, dass nicht schon wieder eine neue Sprache gelernt werden will, was auch als

Yet-Another-Language-To-Learn Syndrom bezeichnet wird (vgl. [Gho11, S.22]).

Solche kulturellen Probleme entstehen i. d. R. dann, wenn etwas Neues eingeführt werden soll (vgl. [VBK⁺13, S.45]). Die Mitarbeiter müssen demnach entsprechend geschult und motiviert werden.

Unvollständige DSLs

Wenn ein Unternehmen viel Erfahrung bei der Entwicklung von *DSLs* aufgebaut hat und die Entwicklung durch die Einführung entsprechender Tools vereinfacht wurde, besteht die Gefahr der voreiligen Entwicklung einer neuen *DSLs*. Das heißt, dass die Notwendigkeit einer neuen *DSL* nicht ausreichend evaluiert wurde. Durch die einfachere Entwicklung scheint es weniger aufwendig eine neue *DSL* zu konstruieren, als nach bestehenden Ansätzen zur Lösung für das gleiche Problem zu suchen (vgl. [VBK⁺13, S.44f]). Die Aussicht auf die Amortisierung einer Investition in neue *DSLs*, unterstützt diese Vorgehensweise (vgl. [FP11, S.38]). Somit entstehen immer mehr *DSLs*, die auf gleichen Problemen basieren, aber untereinander nicht kompatibel sind. Außerdem fördert die Entwicklung einer *DSL* das Verstehen der *Domäne*, weshalb die Möglichkeit besteht, dass dies der einzige Grund für eben diese Entwicklung ist (vgl. [FP11, S.38]). Das wiederum hat zur Folge, dass mehrere unvollständige *DSLs* existieren. Markus Völter et al. nennen dieses Phänomen die „*DSL Hell*“ (vgl. [VBK⁺13, S.44f]).

Zusammenfassend ist zu sagen, dass der Aufwand für die Vorbereitung des Einsatzes einer *DSL* sehr hoch ist. Wurde eine *DSL* jedoch eingeführt, wird sich der Arbeitsaufwand um ein Vielfaches verringern und der Gewinn schließlich höher ausfallen (vgl. [Gho11, S.21]).

3.6. Interne DSLs

Bei einer *internen DSL* handelt es sich um eine *DSL*, die in eine *GPL* integriert ist. Sie übernimmt dabei das Typsystem der *GPL* (vgl. [VBK⁺13, S.50]). In Bezug auf die Ziele aus Kapitel 3.3 können einige dieser Absichten mit *Ap-*

plication Programming Interfaces (API) erreicht werden. In vielen Fällen ist eine *DSL* nicht mehr als ein *API*. Martin Fowler et al. sehen den größten Unterschied zwischen *API* und *DSL* darin, dass eine *DSL* neben einem abstrahierten Vokabular auch eine spezifische Grammatik nutzt (vgl. [FP11, S.29]). Ein *API* hingegen besitzt die gleichen syntaktischen Strukturen wie die *GPL*, in der das *API* bereitgestellt wurde. Somit werden überflüssige Notationsformen in das *API* übernommen, was bei einer *DSL* nicht der Fall ist (vgl. [VBK⁺13, S.30]). Weiterhin können *DSLs* so konstruiert werden, dass durch Restriktionen und Limitierungen nur korrekte Programme geschrieben werden können. Markus Völter et al. bezeichnen diese Eigenschaft als „*correct-by-construction*“ (vgl. [VBK⁺13, S.30]).

3.6.1. Implementierungstechniken

Parse Tree Manipulation

Allgemein betrachtet funktioniert diese Technik wie folgt:

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

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

Konstrukte, die in der Wirtsprache geschrieben wurden und nicht verändert werden sollen, spielen bei der *Parse Tree Manipulation* für die Erzeugung der *semantischen Modelle* keine Rolle (vgl. [FP11, S.46]).

Fluent Interfaces

In einem klassischen *API* hat jede Methode eine eigene Aufgabe und ist nicht von anderen Methoden in diesem *API* abhängig (vgl. [FP11, S.28]). In einer *internen DSL* hingegen ist es möglich, Methoden bereitzustellen, die verkettet werden können und somit komplette Sätze darstellen. Dadurch wird der Output einer Methode zum Input der folgenden Methode. Demzufolge wird die Lesbarkeit der *DSL* wesentlich verbessert, da es einer Sequenz von Aktionen gleicht, die in der *Domäne* ausgeführt werden (vgl. [Gho11, S.94]) und ohne eine Vielzahl von Variablen aufgerufen werden müssen (vgl. [FP11, S.68]). Eine solche Verkettung von Methoden wird als *Fluent Interface* bezeichnet. Das *Fluent Interface* steht laut Markus Völter et al. zwischen dem *API* und einer *internen DSL* (vgl. [VBK⁺13, S.50]). Ein einfaches Beispiel für ein *Fluent Interface* wird von Martin Fowler et al. beschrieben.

Listing 1: Beispiel: Fluent Interface

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

(vgl. [FP11, S.68])

Annotationen

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

Bevor eine *Annotation* verarbeitet werden kann, muss sie definiert werden. Die Definitionen von *Annotationen* variieren bei Verwendung unterschiedli-

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

3.7. Externe DSLs

Eine externe *DSL* ist eine separate Sprache, welche die Infrastruktur vorhandener Sprachen nicht nutzt (vgl. [Gho11, S.18]). Das bedeutet, dass eine *externe DSL* eine eigene *Syntax* sowie ein eigenes Typsystem besitzt. In der Regel wird mit einer *externen DSL* ein Skript geschrieben, welches von einem Programm gelesen wird. Dieser Vorgang wird auch als *Parsen* bezeichnet (vgl. [FP11, S.28]). Für den *Parser* und die *lexikalische Analyse* werden oft vorhandene Infrastrukturen genutzt (vgl. [Gho11, S.19]).

3.7.1. Implementierungstechniken

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

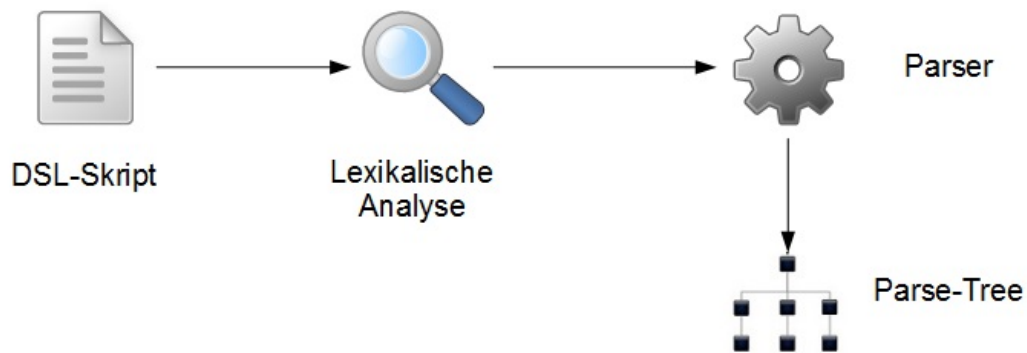


Abbildung 4.: Parsen allgemein

Parsergenerator

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

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

Recursive Decent Parser (RD-Parser)

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

Grammatikregel	Implementierung
$A \mid B$	<pre> if (A()) then true else if (B()) then true else false </pre>
$A B$	<pre> if (A()) then if (B()) then true else false else false </pre>
$A?$	<pre> A(); true </pre>
A^*	<pre> while (A()); true </pre>
A^+	<pre> if (A()) then while (A()); else false </pre>

Tabelle 2.: Einfache Grammatikregeln mit einem *RD-Parser* (vgl. [FP11, S.248])

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

Parserkombinator

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

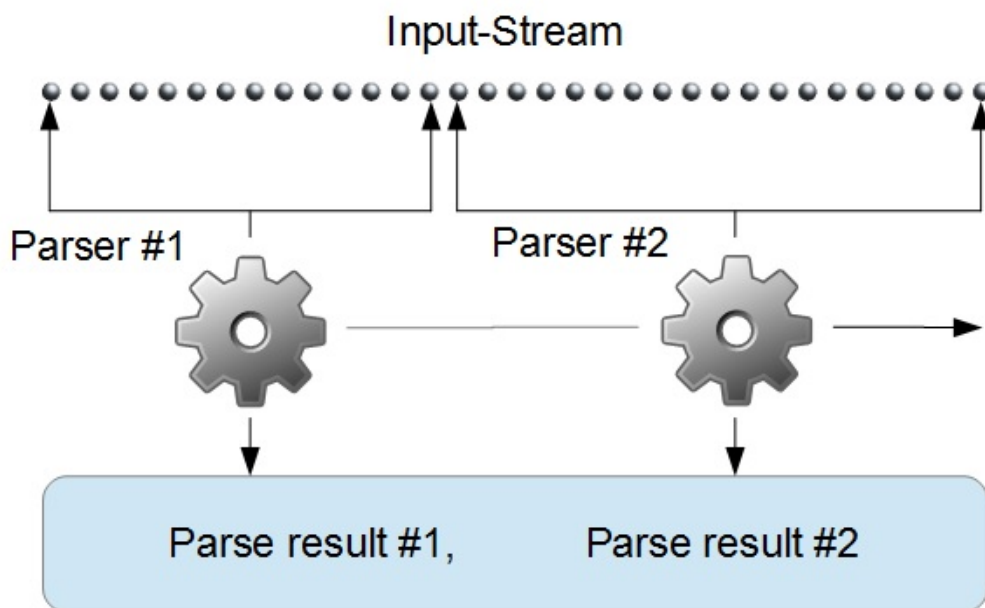


Abbildung 5.: Funktionsweise von Parserkombinatoren (in Anlehnung an [Gho11, S.243])

Bezogen darauf, dass ein *Parser* aus Funktionen besteht, sind diese *Parserkombinatoren* Funktionen erster Ordnung, die unterschiedlich kombiniert werden können (vgl. [Gho11, S.243], [FP11, S.256]). Durch diese Kombination wird eine Struktur gebildet, welche das *semantische Modell* repräsentiert (vgl. [FP11, S.256]). Ein großer Vorteil dieser Technik ist, dass einfache *Parsers* zu komplexeren *Parsern* zusammengefügt werden können.

Weiterhin wird durch die Kombination mehrerer grammatikbestimmender Komponenten auch die Lesbarkeit der Grammatik gefördert, was bei einem *RD-Parser* ein großer Nachteil ist. Daher bezeichnen Martin Fowler et al.

Parserkombinatoren auch als Mittelweg zwischen *RD-Parsern* und *Parsergeneratoren* (vgl. [FP11, S.261]).

3.8. Nicht-textuelle DSLs

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

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

(vgl. [Gho11, S.19])

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

4. GUI-DSL

4.1. Motivation des Ansatzes

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

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

Die Umsetzung des letztgenannten Ansatzes ist weitaus einfacher, da lediglich die Aspekte, welche die *GUI* betreffen, auf *MDSD* umgestellt werden müssen. Zudem soll den Entwicklern weiterhin die Möglichkeit gegeben werden die *GUIs* selbst zu entwerfen, was durch die im ersten Ansatz genannte Beschreibungsform nicht möglich ist, da die *GUIs* vollständig generiert werden. Wogegen unter der Verwendung des zweiten Ansatzes weiterhin *GUI-Komponenten* verwendet werden, was den Entwicklern die Anpassung der *GUIs* ermöglicht. Von daher wird dieser Ansatz weiter verfolgt und eine *GUI-DSL* konzipiert, welche die nachfolgenden Anforderungen erfüllen soll.

4.2. Anforderungen an die GUI-DSL

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

AS1 Beschreibung von *GUIs* über die Zusammensetzung von *GUI-Komponenten*

AS2 Wiederverwendung und Erweiterung bzw. Veränderung beschriebener *GUIs*

AS3 Verwendung einer abstrakten Layoutbeschreibung

AS4 Gebrauch von weniger Quellcode zur Beschreibung von *GUIs*

AS5 Beschreibung von Interaktionen mit den *GUI-Komponenten*

AS6 Erweiterung um neue *GUI-Komponenten*

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

In Bezug auf das Layout muss erwähnt werden, dass in der traditionellen *GUI-Entwicklung* die Strukturierung der *GUI-Komponenten* mit Hilfe von Layoutcontainern vorgenommen wird. In der Vergangenheit hat sich gezeigt, dass die Strukturierung über ein spezifisches Layout zu einer Orientierung an einem bestimmten Framework führt (Beispiel: *MCF* orientiert sich an *Swing*). Dies ist unvorteilhaft, da bestimmte Layouts auf anderen Plattformen (bspw. Web oder Mobil) nicht dargestellt werden können, weil entsprechende Layoutmanager nicht vorhanden sind. Somit ist das Layout in der *GUI-DSL* so zu beschreiben, dass es auf allen Plattformen gleichermaßen gut dargestellt werden kann (siehe Anforderung AS3).

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

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

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

5. Entwicklung einer Lösungsidee

5.1. Allgemeine Beschreibung der Lösungsidee

Eine Lösungsidee für die in Kapitel 2.3 beschriebenen Probleme wurde im Kapitel 2.4 bereits angedeutet. Kern dieser Idee ist, die im vorherigen Kapitel angesprochene *GUI-DSL* zur Beschreibung von *GUIs* zu nutzen. Diese *GUIs* sollen so beschrieben werden, dass sie in der Domäne von *profil c/s* für unterschiedliche *GUI-Frameworks* genutzt werden können. Diese Beschreibung soll weiterhin nur einmal stattfinden. Der Quellcode, welcher die *GUI* im entsprechenden Framework darstellt, wird frameworkspezifisch aus der *GUI-Beschreibung* generiert. Langfristig betrachtet könnte das *MCF* damit abgelöst werden.

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

5.2. Konzept

Die *GUI-DSL* wird für die abstrakte Beschreibung der *GUI* verwendet. Somit ist gewährleistet, dass die *GUI* weiterhin nur einmal beschrieben werden muss. Wie in Abschnitt 5.1 beschrieben, wird der Quellcode zur Darstellung der *GUI* mit Hilfe eines speziellen *Generators* erzeugt. Daraus folgt, dass die Integration eines neuen Frameworks (siehe Anforderung AA2) an

die Implementierung eines spezifischen *Generators* gekoppelt ist. In Abbildung 6 wird das grundlegende Konzept des Ansatzes schematisch dargestellt. Dabei wurden exemplarisch drei unterschiedliche *Generatoren* für bestimmte Frameworks verwendet.

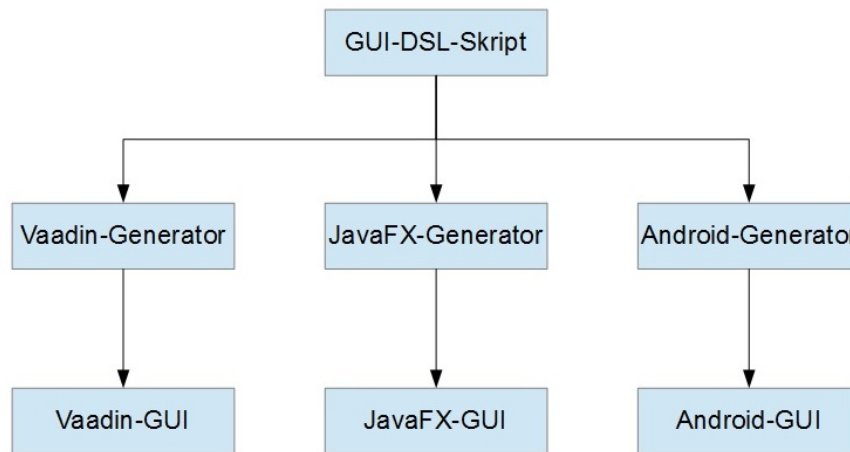


Abbildung 6.: Grundlegendes Konzept

Der Prototyp wird aus einem *GUI-Skript* den Quellcode erzeugen, welcher in das *MCF* eingebunden werden kann. Somit lässt sich prüfen, ob die *GUI-DSL* für existierende *GUIs* von *profil c/s* genügt. In Anlehnung an Abbildung 3 in Kapitel 3.3 wird zur Erzeugung des Quellcodes für das *MCF* auch individueller Quellcode von Nöten sein. In Abbildung 7 wird diese grundlegende Idee schematisch aufgezeigt.

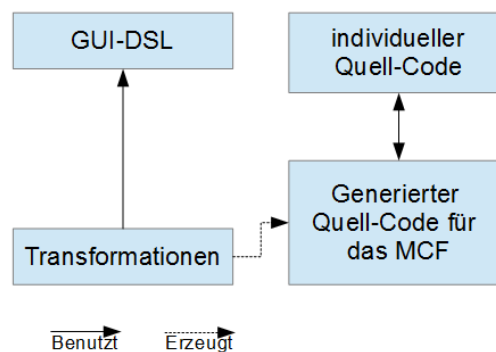


Abbildung 7.: Grundlegende Idee für den Prototyp

6. Evaluation des Frameworks zur Entwicklung der DSL

6.1. Vorstellung ausgewählter Frameworks

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

6.1.1. PetitParser

Dieses Framework arbeitet mit *Parserkombinatoren*. Somit ist es mit *PetitParser* einfach Grammatiken zusammenzustellen, zu transformieren oder zu erweitern sowie Teile dieser dynamisch wiederzuverwenden. Alles geschieht auf der Basis von *Pharo Smalltalk*, womit das Framework ursprünglich implementiert wurde (vgl. [RDGN10]). Es existieren auch Versionen des Frameworks für *Java*¹, *Dart*² und *PHP*³.

Einfache *Parser* bestehen aus Sequenzen von Funktionen, welche die Produktionsregeln 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 *Parser* einem Skript ähnelt. Alternativ können die zu kombinierenden *Parser* auch in Methoden von Unterklassen des *PetitParsers* implementiert werden (vgl. [BCK10, S.6]). Das fördert die Lesbarkeit, Übersichtlichkeit und erleichtert schließlich die Wartung des

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

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

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

Quellcodes.

Der Toolsupport ist für dieses Framework gewährleistet. Mithilfe dessen können Produktionsregeln editiert und grafisch abgebildet werden. Weiterhin ist es möglich Zufallsbeispiele für ausgewählte Produktionsregeln zu generieren, um somit Fehler in der Grammatik aufzudecken. Darüber hinaus wird die Effizienz einer Grammatik durch die Darstellung und Behebung direkter, überflüssiger Zyklen verbessert (vgl. [RDGN10]).

6.1.2. Xtext

Bei *Xtext* handelt es sich um eine Open-Source-Lösung für einen ANTLR-basierten *Parser- und Editorgenerator* mit dem *externe, textuelle DSLs* entwickelt werden können. Die Grammatiken für den *Parsergenerator* werden in der *EBNF* definiert. Durch die Integration in *Eclipse* kann der dazugehörige Editor für sämtliche Artefakte der Infrastruktur von *Xtext* verwendet werden. Aus der Grammatik wird der *Parser* sowie ein Modell generiert, mit dessen Hilfe weitere Artefakte der *DSL-Umgebung* implementiert werden können. Die Klassen für Validierungsregeln und den Generator werden ebenfalls vom Tool erzeugt. Diese müssen im Anschluss daran vom Nutzer entsprechend erweitert werden (vgl. [The14]). Zum Editieren der entsprechenden Dateien wird eine eigene *Syntax* verwendet, die stark an die *Java-Syntax* erinnert.

Wenn der *Parser* generiert wurde, ist es möglich einen in die *Eclipse IDE* integrierten Editor zu erzeugen (vgl. [VC09, S.1]). Dieser Editor ist in der Lage die Validierungsregeln auf die *DSL-Skripte* anzuwenden. Darüber hinaus wird auch Autovervollständigung des Quellcodes vom Editor angeboten.

6.1.3. Meta Programming System

Das *Meta Programming System (MPS)* ist ebenfalls eine Open-Source-Lösung, bei der die Entwicklung *externer DSLs* im Vordergrund steht. Bei der Entwicklung der Sprache mit *MPS* ist weder eine Grammatik noch ein *Parser* involviert. Die Sprache kann mit diesem Tool nicht nur in Textform definiert werden, sondern auch mittels Symbolen, Tabellen oder Grafiken (vgl. [VBK⁺13, S.16]).

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

Der *Generator* kann die Konstrukte der neuen Sprache in bestimmte Basissprachen überführen. Diese Basissprachen sind *C*, *Java* oder *XML*. Auch die Transformation in einfache Texte ist gewährleistet (vgl. [Völ11]).

Zudem wird für die Arbeit mit der *DSL* ein weiterer Editor zur Verfügung gestellt. Dieser bietet mehrere Funktionalitäten wie Autovervollständigung, Refactoring-Möglichkeiten oder einen Debugger (vgl. [PSV13]).

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

6.2. Vergleich und Bewertung der vorgestellten Frameworks

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

Machbarkeit der Integration in Eclipse

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

Erweiterung der Grammatik

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

Bereitstellung eines Editors für DSL-Skripte

Ein Editor soll die effiziente Entwicklung unterstützen. Features wie

⁴<https://www.jetbrains.com/mps/>

Autovervollständigung oder ausdrucksvolle Fehlermeldungen sind der *data experts GmbH* daher wichtig.

Erweiterung der Validierungen

Um ausdrucksvolle Fehlermeldungen verwenden zu können, ist es notwendig Validierungen durchzuführen, die entsprechende Fehler aufdecken können. Die Standardvalidierungen 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 *data experts GmbH* mit den vorgestellten Frameworks ebenfalls mit einbezogen.

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

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

Tabelle 3.: Bewertung der Frameworks für die Entwicklung von DSLs

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

Bei der Möglichkeit zur Erweiterung der Grammatik müssen beim 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 der *Eclipse IDE* verwendet werden kann.

Die Validierungen bzgl. der *DSL-Skripte* können bei *PetitParser* durch die

Parserkombinationen umgesetzt werden. Ausdrucksvolle Fehlermeldungen können jedoch nicht bereitgestellt werden. Die anderen Frameworks bieten dafür weitaus bessere Möglichkeiten.

Der letzte und entscheidende Punkt ist das vorhandene Know-How. Die *data experts GmbH* hat über *Xtext* eine geringe Sachkenntnis, wohingegen die Frameworks *PetitParser* und *MPS* in der Praxis noch nicht eingesetzt wurden.

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

7. Grobkonzept für die Entwicklung des Prototyps

7.1. Vorgehensmodell

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

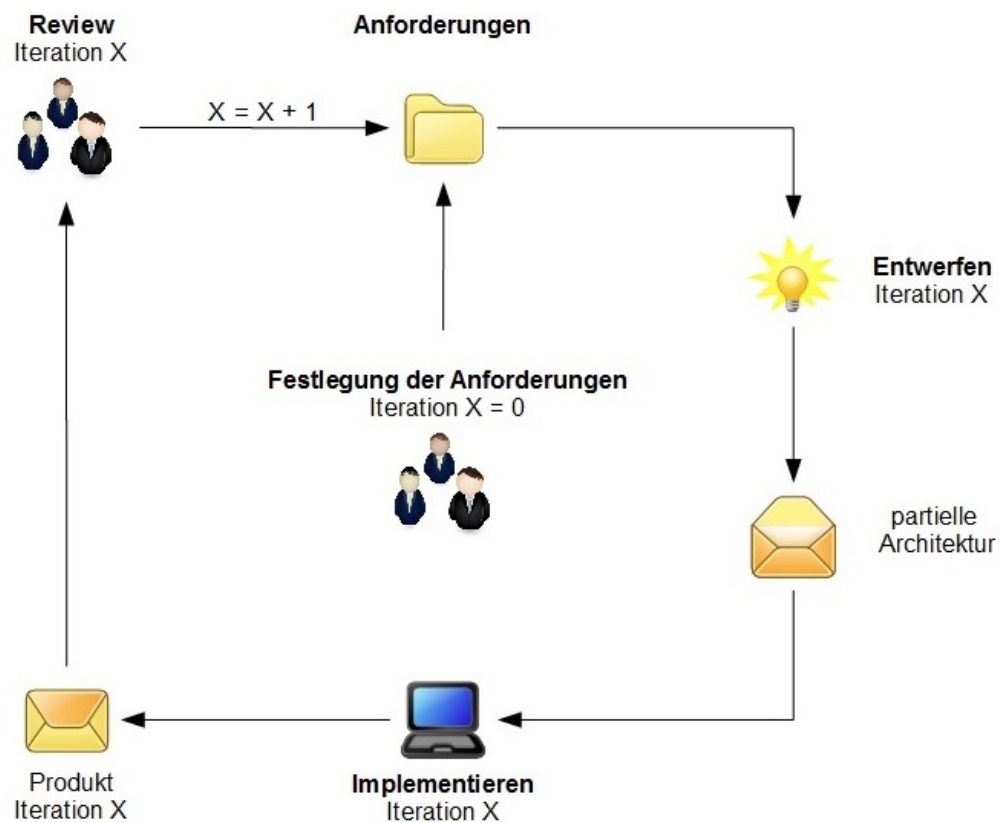


Abbildung 8.: Inkrementelles Modell

Nach der Definition der Anforderungen wird der Prototyp für die aktuelle Iteration entworfen und entwickelt. Im folgenden Verlauf werden diese beiden Phasen nicht separiert. An die Implementierung des Prototyps 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 vorgenommen. Das führt wiederum zu einem neuen Entwurf, woran sich eine weitere Implementierung anschließt. Dieser Zyklus wird somit mehrmals durchlaufen (Iteration) (vgl. [Sau10, S.5]).

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

Im weiteren Verlauf (Kapitel 7.2, 8 und 9) werden die durchgeführten Iterationen beschrieben. Dabei werden folgende Aspekte beleuchtet:

- **Vision** (siehe Kapitel 7.2)

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

- **Entwicklung**

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

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

7.2. Konzeption der DSL-Umgebung (Vision)

7.2.1. 1. Iteration

Die *DSL-Umgebung* soll sich in zwei Bereiche unterteilen.

- *DSL* zur Beschreibung der *GUI*
- *Generator* zur Generierung von Quellcodes

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

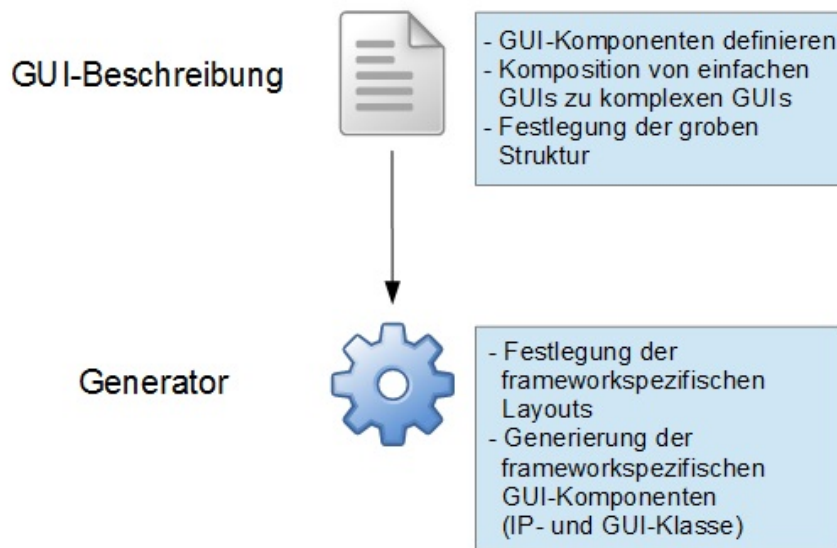


Abbildung 9.: 1. Iteration: Konzeption der DSL-Umgebung

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

- **Basiskomponenten**

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

- **Komplexe Komponenten**

Diese beschreiben domänenspezifische *GUI-Komponenten*, welche speziell für *profil c/s* entwickelt wurden. Interaktionen und Funktionalitäten wurden bereits festgelegt und können nicht verändert werden. Ein Beispiel für eine komplexe Komponente ist die *Multiselection-Komponente*, welche in Abbildung 38 in Anhang A dargestellt ist.

- **Layoutkomponenten**

Dabei handelt es sich um Komponenten, welche die Struktur der *GUI* bestimmen. In anderen *GUI-Frameworks* sind dies bspw. *Panel* (Swing), *Div* (HTML) oder *Pane* (JavaFX).

Bei den Beschreibungen der *Basiskomponenten* muss auch die Möglichkeit bestehen Interaktionen festzulegen (siehe Anforderung AS5). Zu diesen festgelegten Interaktionen gehören eine entsprechende Form und bestimmte Aktionen, die ausgeführt werden müssen. Die anderen *GUI-Komponenten* haben entweder feste (*komplexe Komponenten*) oder gar keine (*Layoutkomponenten*) Interaktionsmöglichkeiten.

Bei der Entwicklung der *komplexen Komponenten* ist darauf zu achten, dass sie für jedes verwendete *GUI-Framework* implementiert werden müssen. Damit wird verhindert, dass die Entwickler, die mit der *GUI-DSL* arbeiten, eigene *komplexe Komponenten* entwerfen, deren Wiederverwendungsgrad niedriger ist. Solche Komponenten sollten nur nach ausreichender Evaluation an einer zentralen Stelle implementiert und bereitgestellt werden. Die Notwendigkeit, dass die Quellcodes für diese *komplexen Komponenten* sowohl zur Entwicklungszeit als auch zur Laufzeit vorhanden sein müssen, ist ein Nachteil dieses Konzepts.

Bei den *Layoutkomponenten* ist besonders auf die Ausdruckskraft der für die Beschreibung dieser Komponenten verwendeten Bezeichnungen zu achten. Grund dafür ist, dass die *GUI* auf unterschiedlichen Plattformen abgebildet werden soll, die spezielle Layoutmanager unterstützen. Ein Programmfenster auf dem Desktop lässt sich beispielsweise als oberste Layoutkomponente festlegen. In der *data experts GmbH* ist in einem Web-Browser der Begriff *Programmfenster* als oberste *Layoutkomponenten* nicht geläufig. Dieses *Programmfenster* auf dem Desktop wird in der *data experts GmbH* mit dem

Browsertab assoziiert.

Darüber hinaus sollen die Skripte für die Beschreibung der *GUIs* so konzipiert werden, dass es möglich ist, andere *GUI-Beschreibungen* dort einzubinden (siehe Anforderung *AS1*). Somit werden eingebundene *GUIs* wiederum zu *GUI-Komponenten*. Ziel dessen ist es, dass die Entwickler aus mehreren einfachen *GUIs* eine komplexe *GUI* erstellen können (Modularisierung). Die Gefahr die dabei besteht, ist, dass mit der Zeit viele einfache *GUI-Beschreibungen* mit ähnlicher Struktur entwickelt werden. Weiterhin verfolgt die *data experts GmbH* bei den *GUIs* von *profil c/s* ein bestimmtes Schema (*Cooperate Design*), welches durch die Modularisierung unterstützt wird. Hierbei ist es wichtig zu betonen, dass die *DSL* keine Sprache sein soll, mit der jede *GUI* beschrieben werden kann, sondern lediglich *GUIs* für *profil c/s*. Der Vorteil, der sich aus dieser starken Modularisierung ergibt, ist, dass viele *GUI-Beschreibungen* wiederverwendet werden können. So ist es möglich, komplexe *GUI-Beschreibungen* zu entwickeln, die in Fachabteilungen mit fachlichen Konzepten assoziiert werden können.

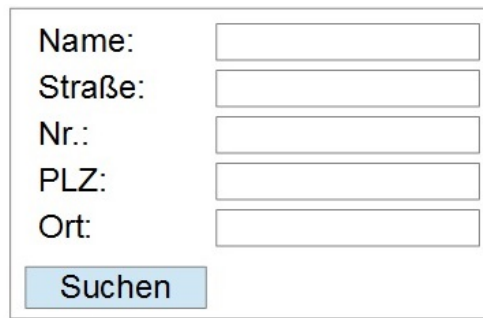
Suchmasken können bspw. nach diesem Konzept gestaltet, beliebig wiederverwendet und kombiniert werden. Hierzu werden unterschiedliche Suchfelder definiert, die aus einem *Label* und einem *Textfeld* bestehen (Beispiel siehe Abbildung 10).



The image shows a simple user interface element for a search field. It consists of a rectangular container with a thin border. Inside the container, on the left side, is the text 'Name:'. To the right of this text is a smaller, empty rectangular box, which is the text input field.

Abbildung 10.: Beispiel: Suchfeld für Name

In einer Suchmaske können mehrere dieser Suchfelder beliebig kombiniert werden (Beispiel siehe Abbildung 11).



The image shows a search mask (Suchmaske) with five input fields and a search button. The fields are labeled 'Name:', 'Straße:', 'Nr.:', 'PLZ:', and 'Ort:'. The 'Suchen' button is located at the bottom left of the form.

Abbildung 11.: Beispiel: Suchmaske

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

Um die miteinander kombinierten *GUI-Komponenten* zu strukturieren, ist es notwendig, dass die *GUI-DSL* Informationen über die Anordnung der *GUI-Komponenten* enthält. Diese Informationen müssen ausreichend abstrakt sein, damit sich diese Struktur auf unterschiedliche *GUI-Frameworks* beziehen lässt. Dazu wird die Struktur innerhalb einer *GUI* als Anordnung von Bereichen betrachtet. In der *GUI-DSL* werden diesen Bereichen die *GUI-Komponenten* zugeordnet. Genauere Informationen über die Anordnung dieser Bereiche dürfen nicht enthalten sein, da dies zu einer Orientierung an ein bestimmtes Layout führen würde (siehe Anforderung AS3). Zudem ist weiterhin wichtig, dass den Bereichen jeweils nur eine Komponente zugeordnet werden kann, wodurch der Zwang zur vorher beschriebenen Kombination von eingebundenen *GUI-Skripten* und definierten *GUI-Komponenten* verstärkt wird.

Die Suchfelder aus dem vorherigen Beispiel (siehe Abbildung ??) bestehen aus zwei Bereichen (siehe Abbildung ??). Einem Bereich wurde das *Label* zugewiesen und dem anderen der *Button*.

Abbildung 12.:

Die *GUI* der Suchmaske (siehe Abbildung ??) bestehe aus sechs Bereichen, denen die verwendeten Komponenten (Suchfelder und *Button*) zugeordnet

wurden (siehe Abbildung ??).

Abbildung 13.:

Der *Generator* übernimmt beim Erzeugen des frameworkspezifischen Quellcodes die konkrete Anordnung der beschriebenen Bereiche. Grund dafür ist, dass die Anordnung der Komponenten frameworkspezifisch ist und teilweise unterschiedliche Layoutmanager in unterschiedlichen Frameworks unterstützt werden. Da für jedes eingesetzte Framework ein eigener Generator implementiert werden muss (siehe Kapitel 5), ist es theoretisch möglich diese Aufgabe weitgehend unabhängig von der Beschreibung der verwendeten *GUI-Komponenten* zu erfüllen.

Der Generator bestimmt bspw. wie die einzelnen Bereiche, die in dem Suchfeld und der Suchmaske verwendet wurden, anzuordnen sind. Abbildung ?? und ?? sind Beispiele für die Anordnung der Bereiche zu entnehmen. Dabei ist zu betonen, dass der Generator keinen Einfluss auf den Inhalt der Bereiche hat.

Abbildung 14.:

Abbildung 15.:

7.2.2. 2. Iteration

Das grundsätzliche Konzept muss in dieser Iteration um ein Artefakt erweitert werden (siehe Abbildung 16).

Eine Änderung, die in dieser Grafik nicht dargestellt wird, ist, dass die Aktionen, die bei Interaktionen mit *GUI-Komponenten* ausgeführt werden, nicht in der *GUI-DSL* definiert werden sollen. Dies liegt an der starken Variation der Aktionen, wodurch diese kaum abstrahiert werden können.

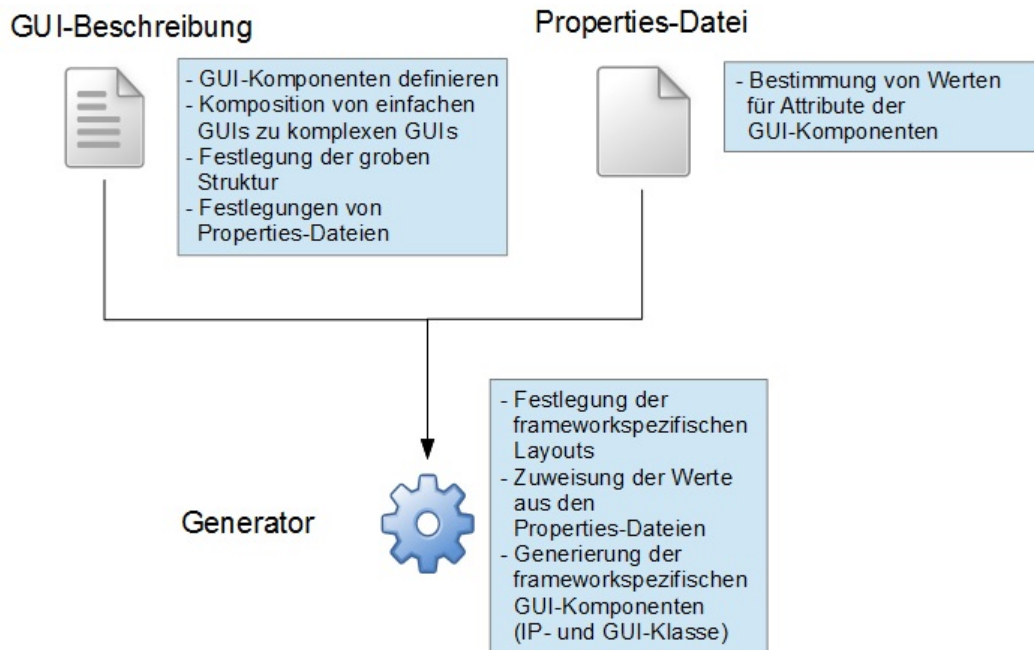


Abbildung 16.: 2. Iteration: Konzeption der DSL-Umgebung

Eine weitere Änderung ist, dass die *GUI-Komponenten*, die in einem *GUI-Skript* direkt definiert werden, in einer anderen Beschreibungen verändert werden können. Die Voraussetzung dafür ist, dass dieses *GUI-Skript* in die andere Beschreibung eingebunden ist. Dadurch können die wiederverwendeten Komponenten angepasst werden, was die Flexibilität steigert (siehe Anforderung AS2).

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

Der *Generator* muss die festgelegten *Properties-Dateien* in die Generierung mit einbeziehen und ihnen die entsprechenden Werte entnehmen. Dabei gilt, dass nicht mehr nach einem bestimmten Attribut einer Komponente in der *Properties-Datei* gesucht wird, wenn diesem Attribut innerhalb des *GUI-Skripts* ein bestimmter Wert zugewiesen wurde.

7.2.3. 3. Iteration

Das grundsätzliche Konzept wird in dieser Iteration nochmals erweitert. Neben den bekannten Artefakten der *DSL-Umgebung* kommt ein weiteres Artefakt hinzu, durch welches das Layout beschrieben werden soll. Das führt dazu, dass die Beschreibung des Layouts nicht mehr im Generator stattfindet, sondern nur noch frameworkspezifisch generiert werden muss. In Abbildung 17 ist das neue Konzept schematisch dargestellt.

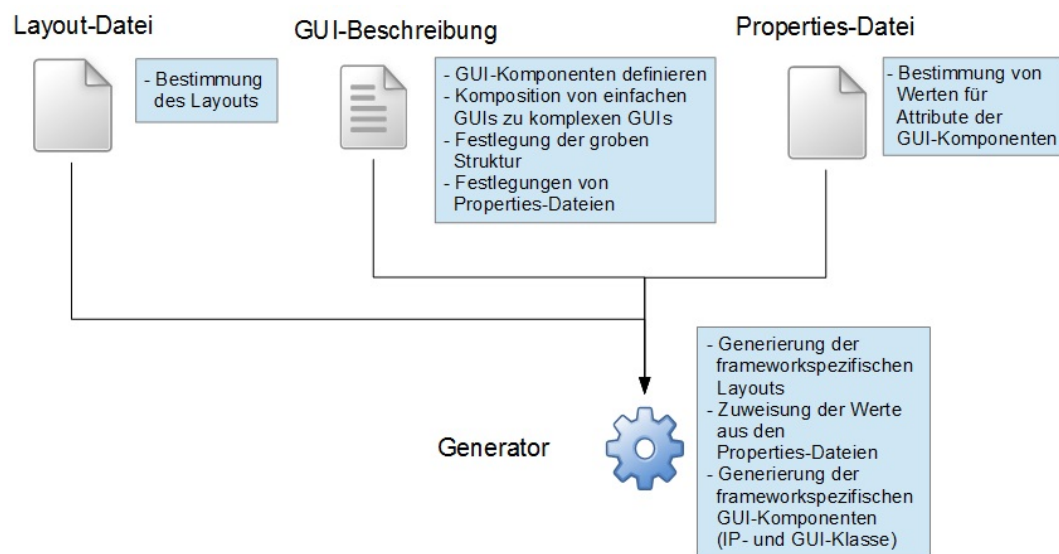


Abbildung 17.: 3. Iteration: Konzeption der DSL-Umgebung

Da die Layoutbeschreibung in eine separate Datei ausgelagert wird, müssen die *GUI-Komponenten* aus dem *GUI-Skript* mit den Festlegungen in der *Layout-Datei* referenziert werden können.

Bezogen auf die im *GUI-Skript* definierten Bereiche ist aufgefallen, dass die Angabe der Anzahl der Bereiche nicht notwendig ist, da dieser Wert doppelt definiert werden würde. Zum einen würde dies direkt in der Angabe der Anzahl festgelegt werden und andererseits indirekt durch die Zuweisung der *GUI-Komponenten* zu den Bereichen. Auf Basis dieser doppelten Angabe sollten ursprünglich Validierungen ausgeführt werden. Dies widerspricht jedoch der Anforderung, dass so wenig Codezeilen wie möglich geschrieben werden sollen, um eine *GUI* zu beschreiben (siehe Anforderung AS4).

Weiterhin ist aufgefallen, dass in einem *GUI-Skript* nur eine *Properties-Datei* angegeben werden kann, wohingegen in den *GUI-Klassen* der *data experts*

GmbH jedoch mehrere dieser Dateien festgelegt werden können. Dies ist auch in der *GUI-DSL* zu beachten.

Neben den Komponenten *Button* und *Label* müssen folgende weitere *Basis-komponenten* definiert werden können:

- **Textfield**

Ein Feld, in dem ein einzeliger Text editiert werden kann.

- **Textarea**

Ein Feld, in dem ein mehrzeiliger Text editiert werden kann.

- **Tree**

Eine Baumstruktur, in der mehrere Elemente eingebunden werden können.

- **Table**

Eine Tabellenstruktur, in der mehrere Elemente eingebunden werden können.

- **TabView**

Eine Ansicht, in der aus mehreren Taben ein Tab visualisiert werden kann.

- **Interchangeable**

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

Weiterhin müssen die Bezeichnungen der Interaktionstypen an die Bezeichnungen der *Interaktionsformen (IF)* der *data experts GmbH* angepasst werden. Darüber hinaus sollen folgenden Standardinteraktionen für entsprechende *GUI-Komponenten* verwendet werden, die für die Umsetzung des Prototyps notwendig sind.

- **Label:** *IfTextDisplay*

- **Tree:** *IfTree, IfActivator*

Diese Standardinteraktionstypen müssen nicht in den jeweiligen Komponenten definiert werden.

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

8.1. 1. Iteration

Analyse der Metadaten

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

Für die Beschreibung der Layoutkomponenten wird zwischen zwei Typen (Layouttypen) unterschieden (siehe *semantisches Model TypeDefinition*), um zwischen obersten *Layoutkomponenten* und Anderen zu differenzieren.

Ein weiterer Aspekt in einem *GUI-Skript*, ist die Verwendung anderer *GUI-Skripte* (siehe *Semantisches Modell: Use*).

Zusammenfassend sind für die Beschreibung der *GUI* folgende Metadaten nötig:

- Anzahl der Bereiche
- Zuweisung der GUI-Komponenten zu den Bereichen

- **Angabe des Layouttyps**
- **Angabe der verwendeten GUI-Beschreibungen**
- **Definition von GUI-Komponenten**

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

An den *Basiskomponenten* können darüber hinaus Interaktionen beschrieben werden, wofür Informationen über den Interaktionstyp von Nöten sind. Der einzige Interaktionstyp, der in dieser Iteration umgesetzt wurde, ist ein manuelles Betätigen der Komponente durch den Nutzer. Zudem können an dieser Interaktion Aktionen definiert werden, wodurch andere Komponenten verändert werden können.

Zusammenfassend ergeben sich folgende Metadaten der *Basiskomponenten* :

- **Typ**
- **Bezeichnung**
- **Text**
- **Interaktion** (siehe *Semantisches Modell: Interaction*)

Die Interaktion benötigt folgende Attribute, die beschrieben werden müssen:

- **Bezeichnung**
- **Interaktionstyp**
- **Aktion**

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

- **Aktionstyp**

Dient der Unterscheidung von Interaktionen mit anderen *GUI-Komponenten* und fachlichen Modellen.

- **Element**

Ist ein Verweis auf das Objekt, mit dem interagiert werden soll.

- **Attribute** (siehe *Semantisches Modell: Property*)

Sind zu verändernde Eigenschaften des Elements.

Die *komplexen Komponenten* werden in einer eigenen Komponentendefinition beschrieben, da neben den vordefinierten Funktionalitäten der *komplexen Komponenten* auch weitere optionale Wertzuweisungen möglich sein sollen. Dazu wird nach der Implementierung einer Komponente, ein neues Schlüsselwort für eine Komponentendefinition in die Grammatik eingebaut. Jede komplexe Komponente benötigt darüber hinaus eine Bezeichnung, um referenziert zu werden. In dieser Iteration ist die *Multiselection-Komponente* umgesetzt, welche generisch implementiert ist. Der generische Typ wird innerhalb der Komponente in der *GUI-Beschreibung* definiert. Ebenso muss der Entwickler die Werte angeben, welche in dieser Komponente selektiert werden können. Zusätzlich soll die Möglichkeit bestehen, Werte zu beschreiben, die bereits ausgewählt wurden.

Semantisches Modell

Das Artefakt, welches bei diesem Modell im Mittelpunkt steht, ist die *UIDescription* (siehe Abbildung 19). Die Methoden werden zum Erhalt der Übersichtlichkeit nur in den Interfaces abgebildet. In den Klassen sind lediglich die globalen Variablen dargestellt. Die aggregierten Artefakte, auf die schon im vorherigen Abschnitt hingewiesen wurde, sind dem Diagramm gut zu entnehmen.

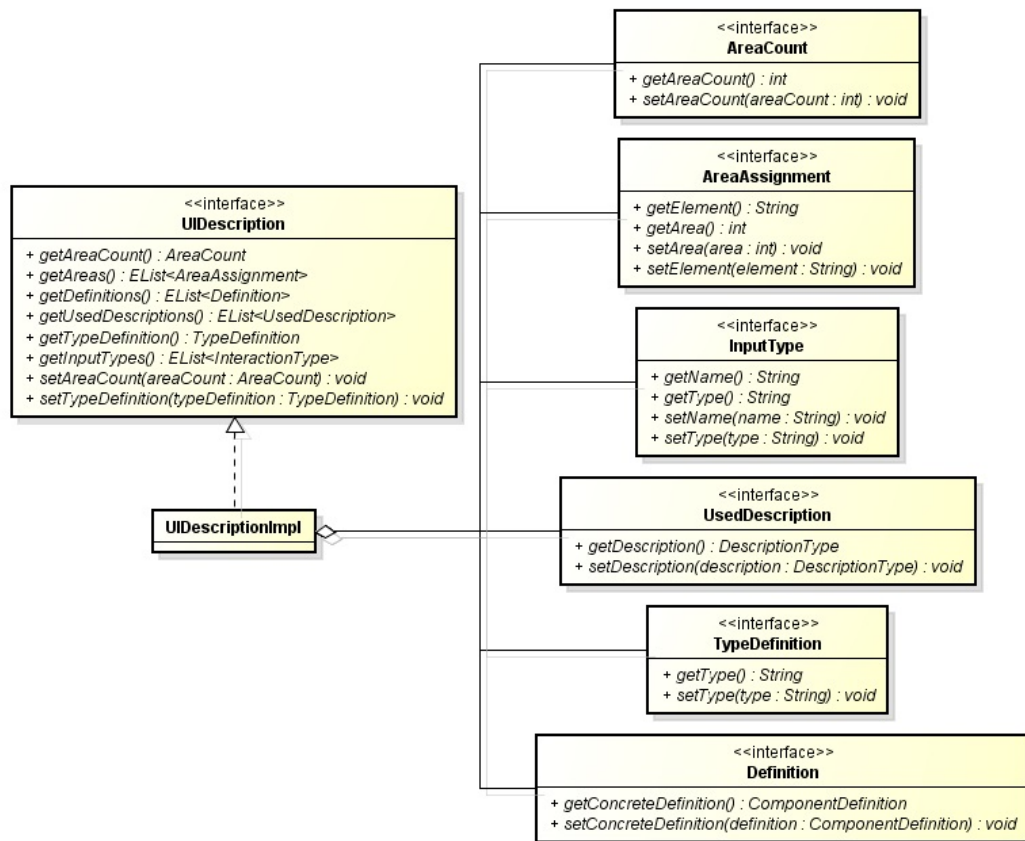


Abbildung 18.: 1. Iteration: UIDescription

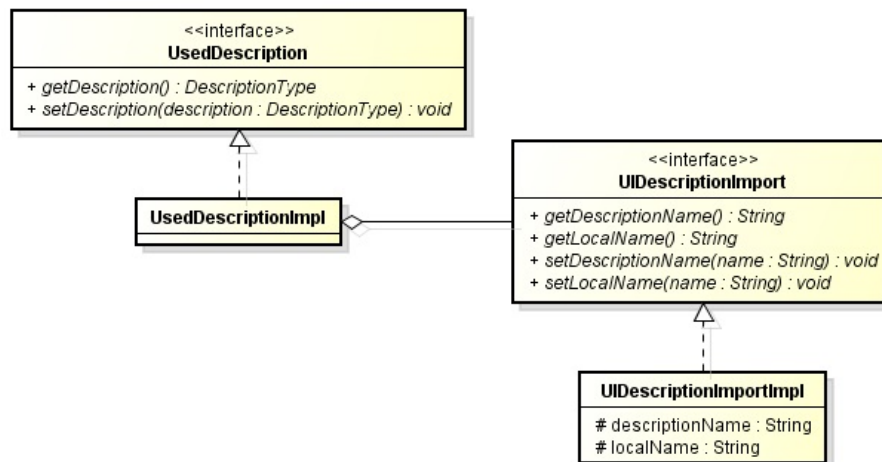


Abbildung 19.: 1. Iteration: UsedUIDescription

Die Klasse *DefinitionImpl* aggregiert weitere Artefakte des Modells. Diese sind, um die Übersicht zu wahren, Abbildung 21 zu entnehmen.

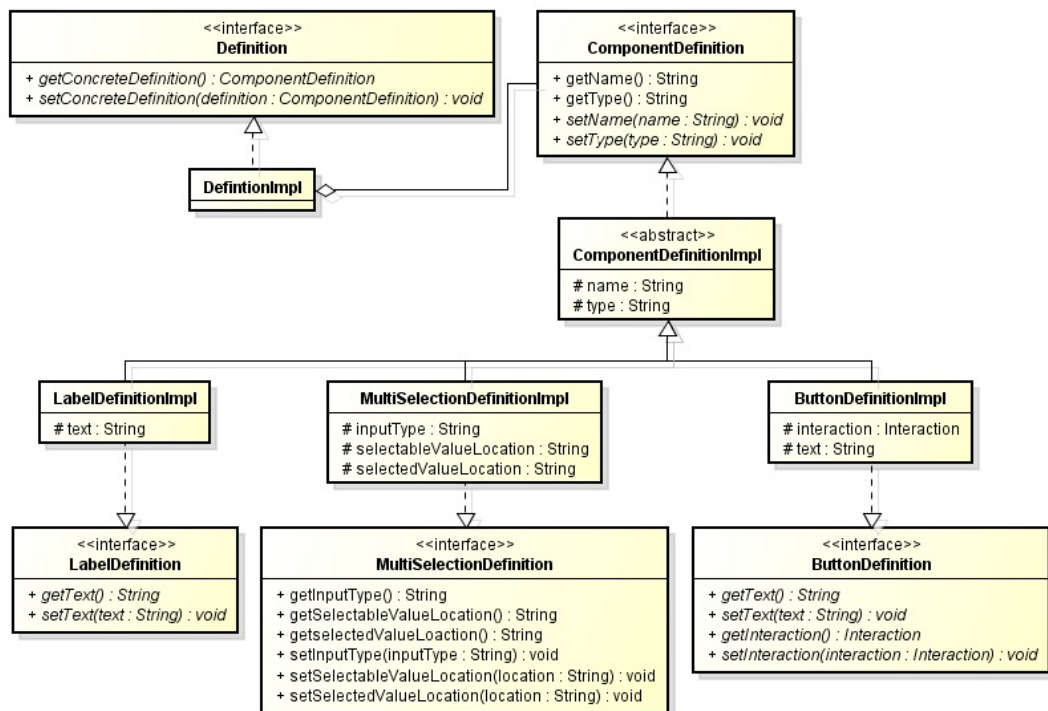


Abbildung 20.: 1. Iteration: ComponentDefinition

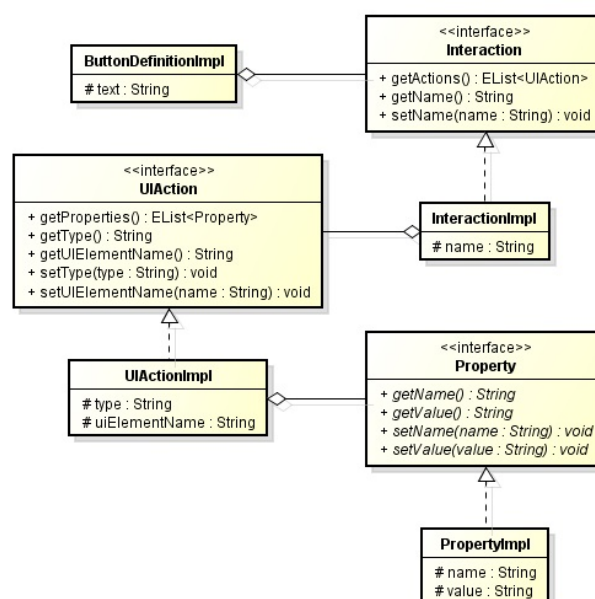


Abbildung 21.: 1. Iteration: Interaction

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 Interaktion (*Interaction*) enthalten kann. Das Interface *Property* wird benötigt, um bestimmte Eigenschaften der *GUI-Komponenten* festzulegen, ohne wissen zu müssen, um welchen Komponen-

tentyp es sich handelt. Dazu wurden die allgemein gültigen Einstellungsmöglichkeiten von *Basiskomponenten* in *CommonProperty* zusammengefasst.

Konkrete Syntax

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

Listing 2: 1. Iteration: Syntax

```

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

```

Mit diesem *GUI-Skript* soll eine *GUI* wie in Abbildung ?? dargestellt werden.

Abbildung 22.:

Die Beschreibung der *GUI* beginnt mit der Deklaration der Anzahl der verwendeten Bereiche. Die Bezeichnung *Area* wurde bewusst gewählt, da dieser Begriff abstrakter ist als die in verschiedenen *GUI-Frameworks* verwendeten Begriffe, wie *Panel* oder *Pane*. Die *Syntax* dieser *DSL* soll sich bzgl. des Aufbaus der *GUI* an keinem *GUI-Framework* orientieren.

Nach dieser Deklaration muss der Layouttyp angegeben werden. In diesem Fall wurde der Typ *WINDOW* verwendet, welcher die oberste Layoutkomponente darstellt. Andere Layoutkomponenten werden durch den Typ *INNERCOMPLEX* abgebildet. Der Begriff wurde gewählt, weil eine *GUI*, die

als *INNERCOMPLEX* beschrieben wurden, in eine andere *GUI* eingebunden werden soll und somit einen Komplex innerhalb dieser einnimmt.

An die Definition des Layouttyps schließt sich die Angabe der eingebundenen *GUI-Skripte* an. In der oben stehenden Beschreibung wurde ein *GUI-Skript* eingebunden, welches die Bezeichnung *AnotherDescription* trägt.

Anschließend werden die einzelnen Komponentendefinitionen durch das Schlüsselwort *DEF* eingeleitet und durch das Schlüsselwort *END DEF* abgeschlossen. Der Definitionskopf wird durch das Zeichen „:“ beendet. Dort sind die Pflichtfelder *Titel* („Interactbt“) und *Typ* (*Button*) der Komponentendefinition zu finden. Weiterhin wurde eine Interaktion für den *Button* definiert, deren Aktion ausgelöst wird, wenn er betätigt wird (*CLICK*). In diesem Fall wird die Aufschrift des *Buttons* in „Du hast interagiert“ geändert.

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 auf dem beiliegenden Datenträger (siehe *Anlage*).

8.2. 2. Iteration

Analyse der Metadaten

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

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

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

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

Alle anderen Metadaten für die *Basiskomponenten* bleiben bestehen.

Bezogen auf die *komplexen Komponenten* ist es lediglich notwendig, den Typ des Inputs anzugeben. Dadurch wird die Definition einer *Multiselection-Komponente* weitaus einfachen, da die Festlegung über Elemente, die selektiert werden können oder bereits ausgewählt sind, nicht benötigt wird. Das ermöglicht die *komplexen Komponenten* mittels *use* (siehe *Semantisches Modell: UsedDefinitions*) in die *GUI-Beschreibung* einzubinden (siehe *Konkrete Syntax*).

Semantisches Modell

In dieser Iteration wurden an den Artefakten *AreaCount*, *TypeDefinition* und *AreaAssignment* keine Änderungen vorgenommen. Die Artefakte *Property* und *Refinement* sind hinzugekommen. Weitere Artefakte, die von *UIDescriptionImpl* aggregiert werden (siehe Abbildung 21), wurden verändert.

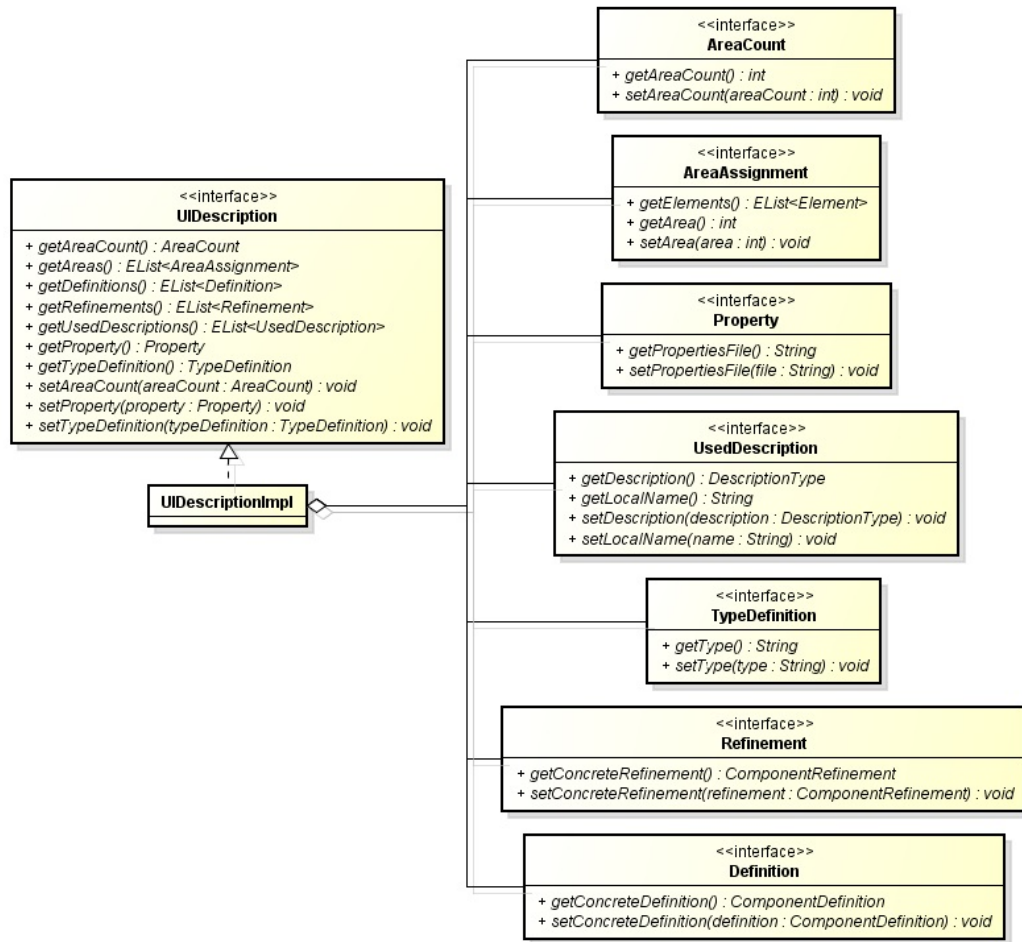


Abbildung 23.: 2. Iteration: UIDescription

Das Artefakt *Property* bildet die *Property-Datei* ab. Sie ist nicht zu verwechseln mit dem Artefakt *Properties*, welches die Eigenschaften von Komponenten abbildet. Abbildung 24 zeigt beide Artefakte auf.

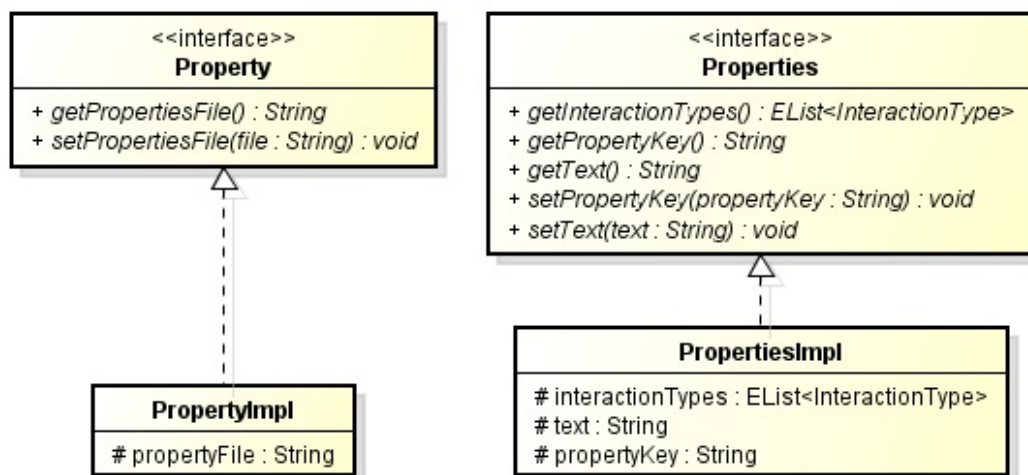


Abbildung 24.: 2. Iteration: Property und Properties

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

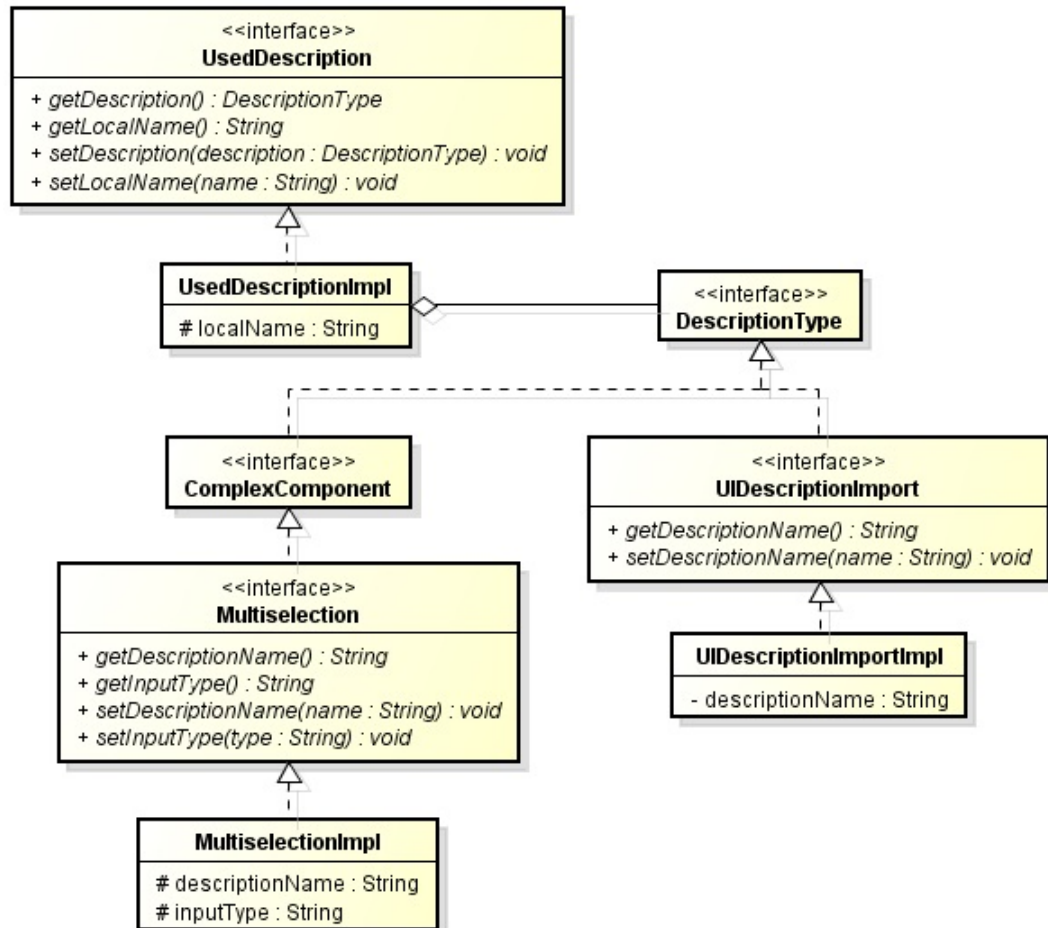


Abbildung 25.: 2. Iteration: UsedDescription

Weiterhin wird zwischen *Definition* und *Refinement* unterschieden. Die *Definition* bildet neu definierte Komponenten für die GUI ab. Ein *Refinement* hingegen beschreibt die veränderten Komponenten importierter GUI-Skripte (siehe Abbildung 26 und Abbildung 27).

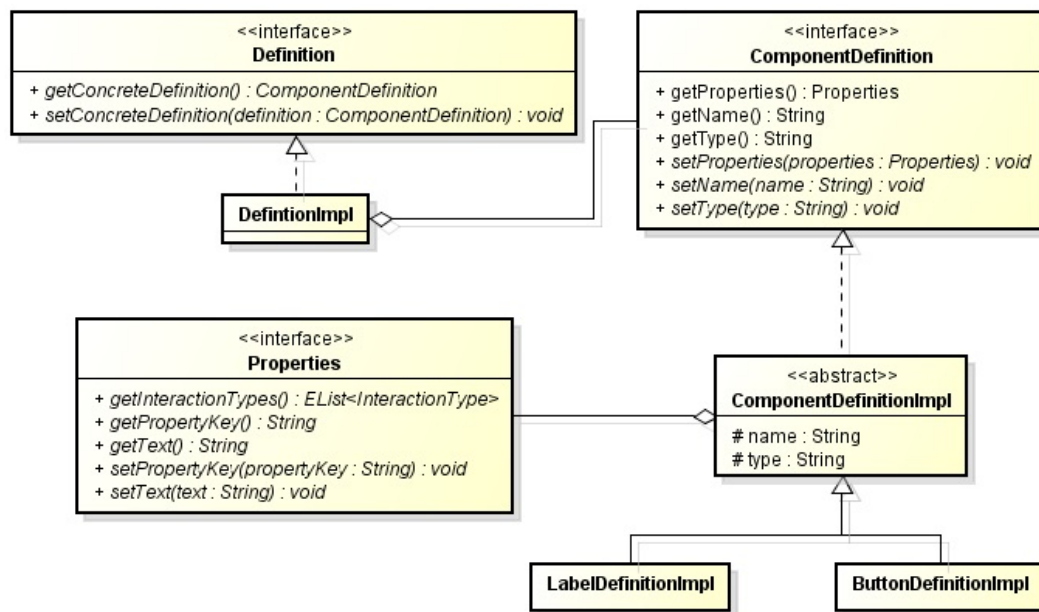


Abbildung 26.: 2. Iteration: Definition

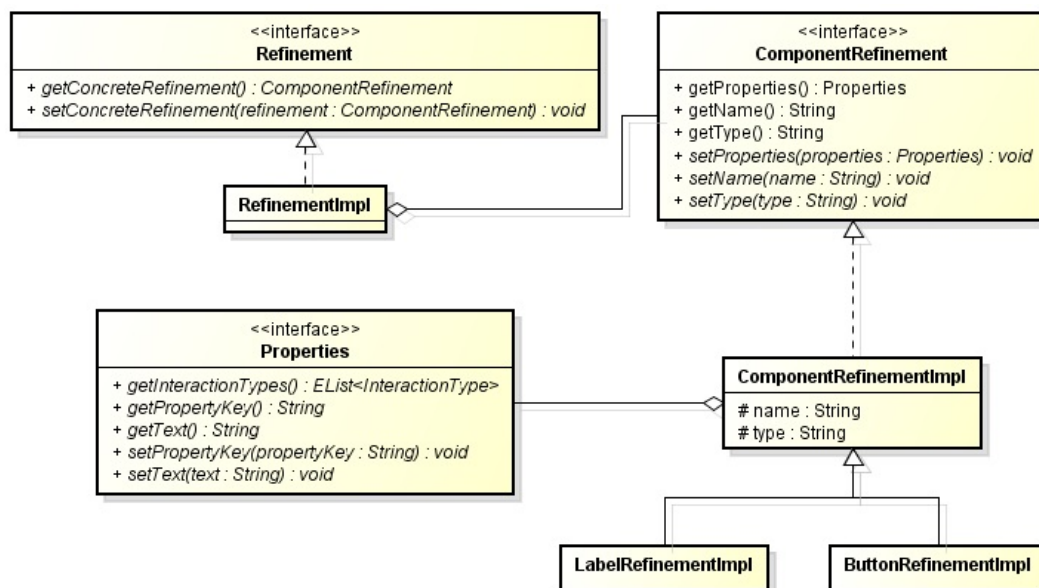


Abbildung 27.: 2. Iteration: Refinement

Konkrete Syntax

Zur Umsetzung des Properties-Konzepts ist in der zweiten Iteration ein neues Schlüsselwort hinzugekommen. Um eine *Properties-Datei* einzubinden, muss, wie in Listing 3, eine entsprechende Datei angegeben und in den Komponentendefinitionen entsprechende Schlüssel deklariert werden. Das *Label* mit der Bezeichnung *OneLabel* enthält keinen Property-Schlüssel

(*PropertyKey*). In diesem Fall wird der Titel als solcher verwendet.

Listing 3: 2. Iteration: Properties

```
1 type: WINDOW
2 get properties from: 'sources.ui.properties '
3 DEF Label as "OneLabel" END DEF
4 DEF Label as "AnotherLabel":
5     propertyKey='AnotherLabel2 '
6 END DEF
```

Aufgrund der Reduktion der Metadaten für eine Interaktion, musste festgelegt werden, ob die Interaktionstypen hintereinander durch Kommasetzung oder untereinander durch entsprechende Schlüsselwörter deklariert werden sollen. Unter Beachtung der Anforderung *AS4* wird die erste Variante bevorzugt (siehe Listing 4).

Listing 4: 2. Iteration: Interaktion

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

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

Listing 5: 2. Iteration: Definition komplexer Komponenten

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

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

Listing 6: 2. Iteration: Button und Label

```
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 durch Kommas voneinander getrennt aufzuzählen. Wenn nur die in der Datei definierten Komponenten einem Bereich zugewiesen werden müssten, würde sich die erstgenannte Lösungsmöglichkeit anbieten. Jedoch können die mit *use* eingebundenen Komponenten auch Bereichen zugeordnet werden, weshalb für dieses Verfahren ein zusätzliches syntaktisches Konzept innerhalb der Zuweisung benötigt werden würde. Um dies zu umgehen, wurde die Entscheidung getroffen, das alte Verfahren zu verfeinern. Listing 7 zeigt ein Beispiel für die Zuweisung von drei Komponenten zu einem Bereich.

Listing 7: 2. Iteration: Area-Zuweisung

```
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"
```

Die Eigenschaften von Komponenten, welche in einem eingebundenen *GUI-Skript* definiert wurden, können durch die Verwendung des Schlüsselworts *REFINE* überschrieben werden. Der erste Teil von Listing 8 zeigt die Originaldatei, welche eingebunden wird. Diese trägt den Namen *LabelAndButton* und befindet sich im Package *guidescription*. Der zweite Teil von Listing 8 zeigt, wie die Aufschrift der Komponente *ButtonToOverride* überschrieben wird.

Listing 8: 2. Iteration: Verändern von Komponenten

```
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 Area count: 1
```



```

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

```

Sollten die Bezeichnungen von eingebundenen Komponenten nicht eindeutig sein, muss der Name des dazugehörigen *GUI-Skriptes* in der Referenz mit angegeben werden (siehe Listing 9).

Listing 9: 2. Iteration: Verändern von Komponenten mit gleichen Namen

```

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

```

Die dazugehörige Grammatik, welche in der zweiten Iteration entwickelt wurde, befindet sich auf dem beiliegenden Datenträger (siehe *Anlage*).

8.3. 3. Iteration

Analyse der Metadaten

Um das Referenzieren von *GUI-Komponenten* und dem entsprechenden Layout aus der *Layoutdatei* zu ermöglichen, ist es wie bei den *Properties-Dateien* notwendig, die *Layoutdatei* innerhalb des *GUI-Skripts* anzugeben. Weiterhin wird in den einzelnen Komponentendefinitionen ein Schlüssel benötigt, über den die Komponente eindeutig referenziert werden kann. Dafür ist es möglich die Bezeichnung der *GUI-Komponente* zu verwenden. Um in der *Layoutdatei* nicht jede *GUI-Komponenten* einzeln aufführen zu müssen, wird innerhalb der *GUI-Beschreibung* ein optionales Feld benötigt, wodurch unterschiedlichen *GUI-Komponenten* derselbe Layout-Schlüssel zugeordnet werden kann.

Da die Anzahl der Bereiche nicht mehr angegeben werden muss, fällt dies aus den Metadaten heraus. Die Zuweisung der *GUI-Komponenten* zu den Bereichen entfällt ebenfalls. Die Strukturierung wird über eine Aufzählung der *GUI-Komponenten* vorgenommen.

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

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

Alle Komponentendefinitionen und die veränderten eingebundenen Komponentendefinitionen benötigen durch die genannten Änderungen folgende Metadaten:

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

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

Basiskomponente	Spezifische Metadaten
Label	Aufschrift
Button	Aufschrift
Textfield	Text, Veränderungsmöglichkeit des Textes
Textarea	Text, Veränderungsmöglichkeit des Textes
Tree	Input-Modell
Table	Input-Modell
TabView	GUI-Beschreibungen der einzelnen Tab

Tabelle 4.: Basiskomponenten mit spezifischen Metadaten

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

Semantisches Modell

Das *semantische Modell* hat sich durch die vielen Änderungen in dieser Iteration stark verändert. Bei der Betrachtung der *UIDescription* (siehe Abbildung 28) fällt auf, dass *Area* und *AreaCount* nicht mehr vorhanden sind. Hinzugekommen sind die Artefakte *Structure*, in dem die Anordnung der GUI-Komponenten (*Element*) in einer Liste beschrieben wird und *Layout*, in dem die verwendeten *Layoutdateien* in einer Liste abgelegt werden.

Eine weitere Änderung ist bei *Property* zu finden. Dort ist jetzt ebenfalls eine Liste vorhanden und kein alleinstehender Wert mehr.

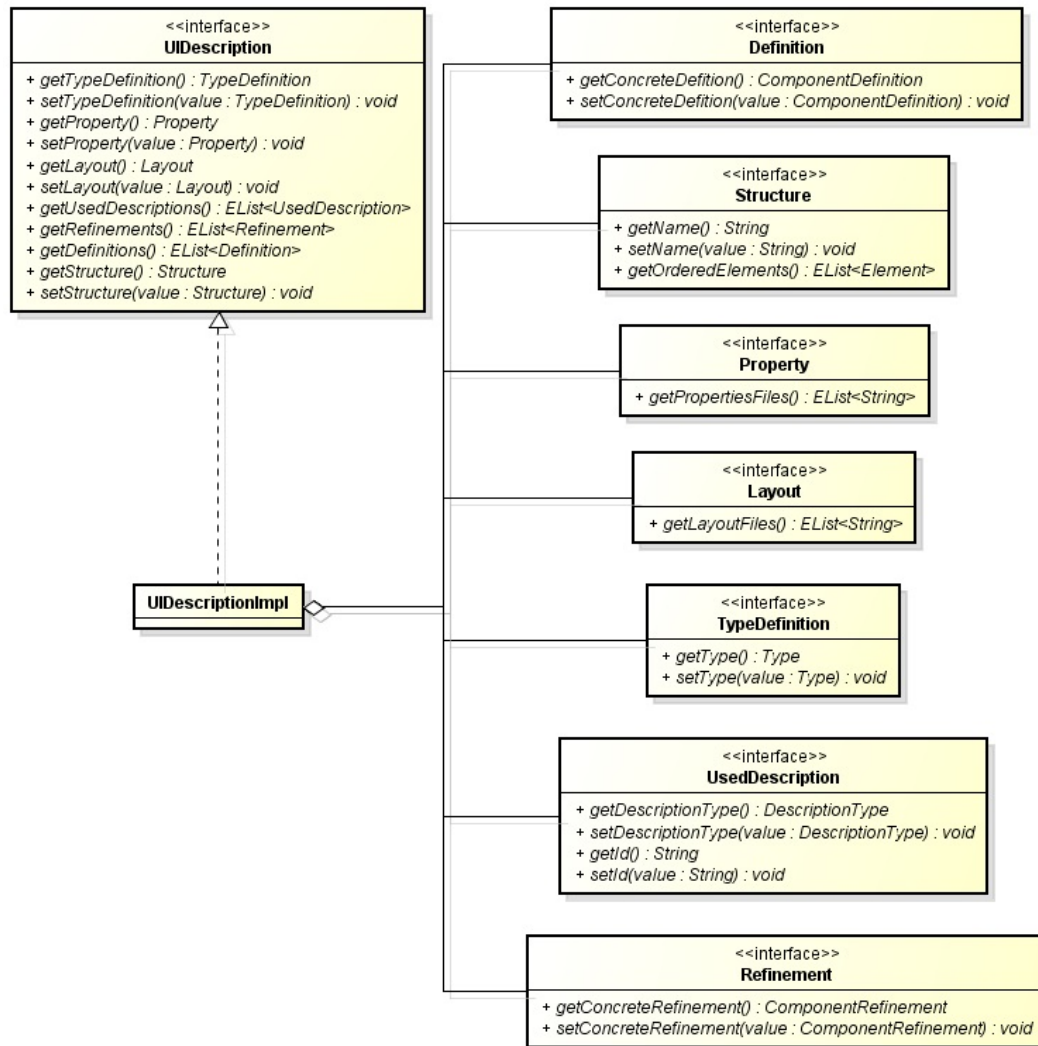


Abbildung 28.: 3. Iteration: UIDescription

An *TypeDefinition* und *UsedDescription* wurden keine signifikanten Änderungen vorgenommen. Die meisten Änderungen wurden bei den Artefakten *Refinement* und *Definition* vorgenommen. Der Aufbau dieser Artefakte ist ähnlich. Das Interface (*Refinement* und *Definition*) wird von einer Klasse implementiert (*RefinementImpl* und *DefinitionImpl*), die mehrere Objekte enthält. Die Klassen dieser Objekte implementieren das Interface *ComponentRefinement* oder *ComponentDefinition*. Abbildung 29 ist diese Struktur für die *Definition* zu entnehmen.

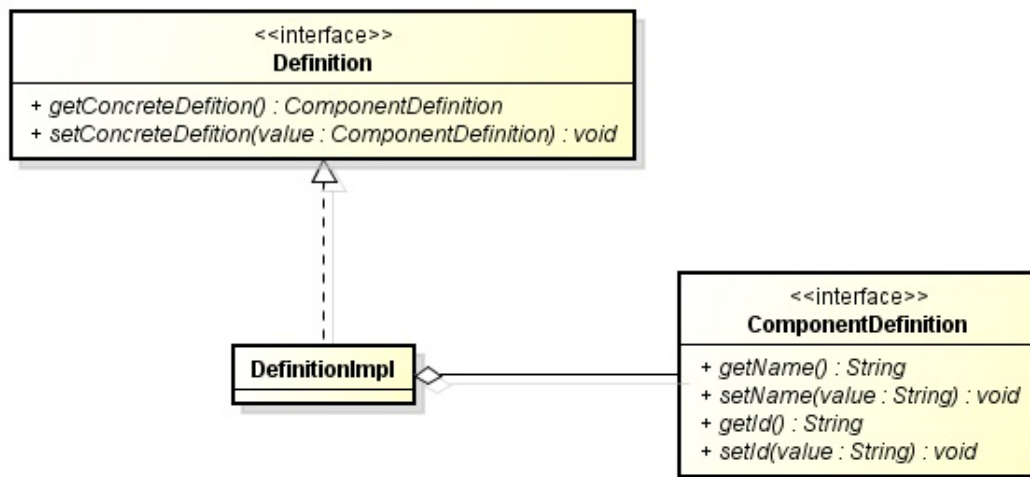


Abbildung 29.: 3. Iteration: Definition

Die benannten Klassen bilden die unterschiedlichen *Basiskomponenten* ab, die definiert oder verändert werden können. Jede dieser Klassen aggregiert ein Objekt des Typs *CommonProperties*. Dieses Interface bildet die allgemeinen Properties ab. Bei den *Basiskomponenten* *Label*, *Button*, *Textfield* und *Textarea* ist diese Aggregation transitiv, da die speziellen Properties dieser Komponenten als eigenes Artefakt implementiert sind. Abbildung 30 zeigt dies für einen *Button* auf.

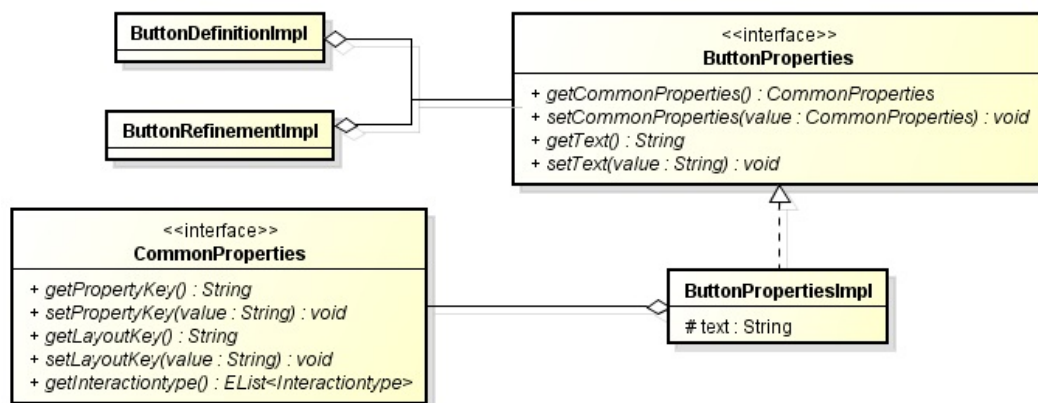


Abbildung 30.: 3. Iteration: Button

Eine direkte Aggregation der *CommonProperties* findet bei den *Basiskomponenten* *TabView*, *Table* und *Tree* statt, für die keine einzelnen Klassen existieren. Die Abbildungen 31, 32 und 33 zeigen diese drei Komponenten (*Refinement* und *Definition*) mit der Aggregation der *CommonProperties*.

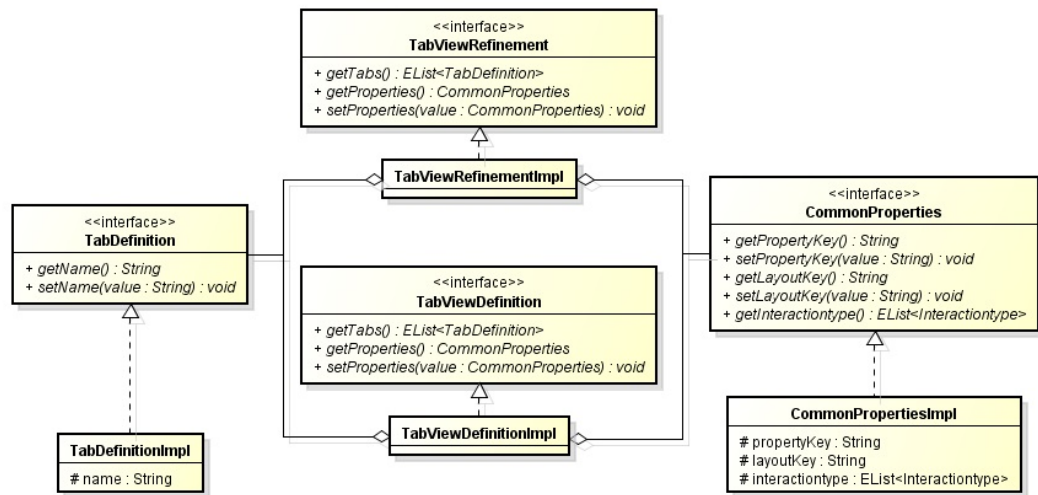


Abbildung 31.: 3. Iteration: TabView

Die Basiskomponente *TabView* benötigt mehrere Objekte des Typs *TabDefinition*. Diese Klasse bildet die Referenz zu den GUI-Komponenten, welche in die *TabView* einzubinden sind. Die Basiskomponenten *Tree* und *Table* benötigen lediglich eine Referenz zum Input-Modell (siehe Abbildung 32 und 33).

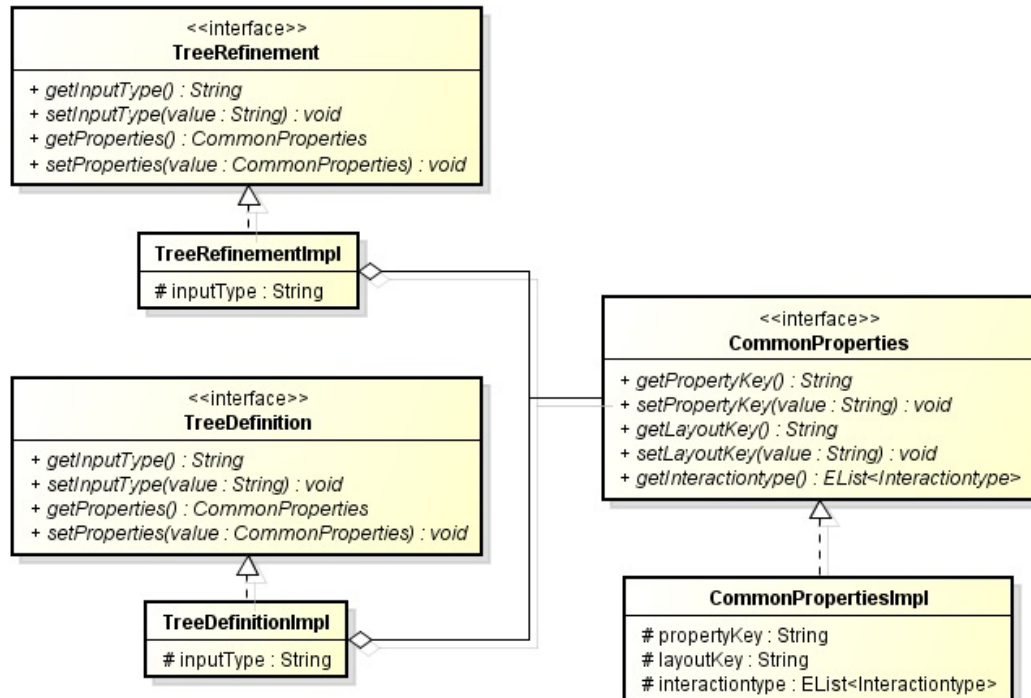


Abbildung 32.: 3. Iteration: Tree

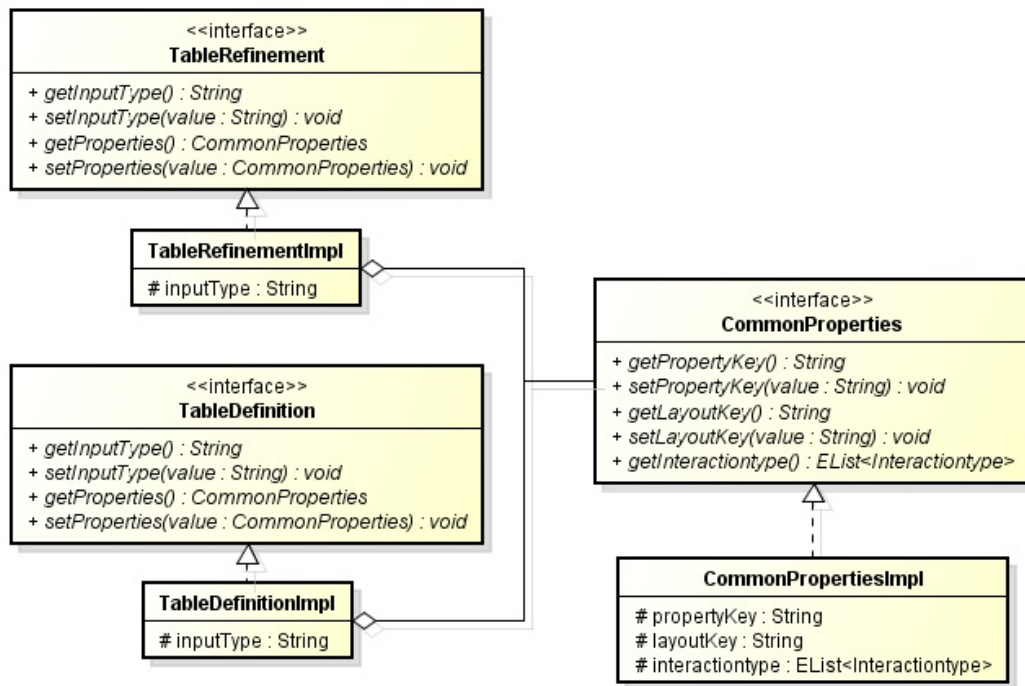


Abbildung 33.: 3. Iteration: Table

Konkrete Syntax

In dieser Iteration sind viele *Basiskomponenten* hinzugekommen. Der grundsätzliche syntaktische Aufbau eines *GUI-Skripts* hat sich jedoch nicht verändert. Eingeleitet wird die Beschreibung weiterhin mit der Typdefinition, gefolgt von der Angabe der *Properties-Dateien*. Sofern mehrere *Properties-Dateien* vorhanden sind, werden diese durch Kommas getrennt.

Dasselbe Prinzip wird bei der sich anschließenden Angabe der *Layoutdateien* verwendet. Das Semikolon gilt ab sofort als Trennzeichen für einen abgeschlossenen Komplex (siehe Listing 10).

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

```

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

```

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

Listing 11: 3. Iteration: Eingebundene GUI-Komponenten

```

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

```

Da die Anforderungen *AA5* und *AS4* mehr Beachtung finden sollten, haben sich die Definitionen der einzelnen *Basiskomponenten* syntaktisch stark verändert. Somit werden die bekannten *Basiskomponenten* wie folgt definiert (siehe Listing 12):

Listing 12: 3. Iteration: Button und Label

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

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

Listing 13: 3. Iteration: Textfield und Textarea

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

Die *Syntax* für die Definition der *Basiskomponenten Table* und *Tree* ist ähnlich. Das benötigte Input-Modell wird zusammen mit dem jeweiligen Schlüsselwort angegeben, welches die *GUI-Komponente* bestimmt (siehe Listing 14).

Listing 14: 3. Iteration: Table und Tree

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

Für die *Basiskomponenten TabView* werden die verwendeten *GUI-Komponenten* der einzelnen *Tab* ebenfalls zusammen mit dem Schlüsselwort angegeben, welches die *GUI-Komponente* festlegt. Hierbei können, anders als bei den *Basiskomponenten Table* und *Tree*, mehrere Angaben gemacht werden (siehe Listing 15).

In den folgenden Beispielen wird davon ausgegangen, dass die vorher be-

schriebenen *Basiskomponenten* *Tree*, *Table* und *Textarea* in demselben *GUI-Skript* definiert wurden. Durch das syntaktische Konstrukt in Listing 15 wird eine *TabView* mit drei Taben beschrieben. Das erste Tab enthält die Komponente *Tree*, das zweite die Komponente *Table* und das dritte die Komponente *Textarea*.

Listing 15: 3. Iteration: TabView

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

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

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

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

Listing 16: 3. Iteration: TabView

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

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

Listing 17: 3. Iteration: Struktur

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

Die für diese Arbeit endgültige Grammatik ist, wie die Grammatiken der anderen Iterationen, ebenfalls auf dem beiliegenden Datenträger (siehe *Anlage*) zu finden.

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

Alle Umsetzungen die in diesem Kapitel beschrieben werden, fanden in der dritten Iteration statt.

Ziel ist es, mit dem *Generator* und einem entsprechenden *GUI-Skript* eine Exploreransicht zu erstellen, die für *profil c/s* charakteristisch ist. Abbildung 34 zeigt den Aufbau einer solchen *GUI*. Darin enthalten sind auf der linken Seite zwei *Trees* und auf der rechten Seite ein Komponente vom Typ *Interchangeable*, in der das selektierte Elemente des *Inhaltsbaums* visualisiert werden soll. Das Anzeigen des Inhalts wird im Prototypen nicht umgesetzt.

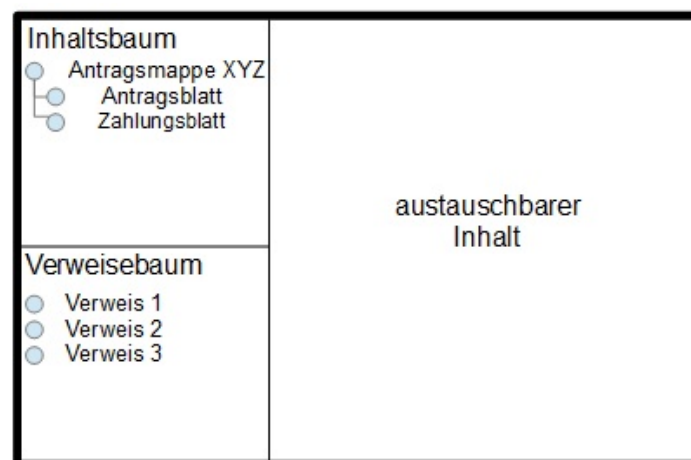


Abbildung 34.: Aufbau eines Explorers

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

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

GUI-Klasse

In den *GUI-Klassen* der *data experts GmbH* werden die *GUI-Komponenten* durch *Präsentationsformen* beschrieben (siehe Kapitel 2). Alle *GUI-Komponenten* sind in diesen Klassen als globale Variablen verfügbar. Diese stehen bei der *data experts GmbH* am Ende der Klasse. Die bestehende Struktur der Klassen soll nicht verändert werden. Da die Interaktion von der Präsentation getrennt ist, müssen für das Referenzieren von Interaktionen zu *GUI-Komponenten* entsprechende Schlüssel vergeben werden (siehe Routinearbeit R1).

IP-Klasse

Die *IP-Klassen* ordnen den *GUI-Komponenten* mit Hilfe dieses Schlüssels entsprechende Interaktionen und darauf folgende Kommandos zu. Die Funktionalitäten der Interaktion können vom *Generator* nicht erzeugt werden. Dies muss vom Entwickler nachgepflegt werden.

FP-Klasse

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

Für die Generierung der *IP*- und *FP*-Klassen ist die *Templated Generation* (siehe Kapitel 3) in Erwägung zu ziehen. Da die Generierung in dieser Arbeit möglichst einheitlich gehalten werden soll, ist die *Transformer Generation* (siehe Kapitel 3) für alle Klassen zu verwenden.

9.2. Umsetzung des frameworkspezifischen Generators

GUI-Klassen

Bei der Präsentation aus Abbildung 34 können die *Trees* auf der linken Seite als einzelne *GUI-Beschreibungen* betrachtet werden, die jeweils mit einem fachlichen Konzept assoziiert werden können. Der obere *Tree* kann mit dem fachlichen Konzept des *Inhaltsbaums* in Verbindung gebracht und der untere mit dem des *Verweisebaums* assoziiert werden. Da sich die fachlichen Konzepte erkennen lassen, sollten separate *GUI-Skripte* erstellt werden (siehe Listing 18 und 19).

Listing 18: GUI-Skript für den Inhaltsbaum

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

Listing 19: GUI-Skript für den Verweisebaum

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

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

es möglich die bestehende Struktur der Klassen der *data experts GmbH* beizubehalten.

Listing 20: Speichern der Importe und der globalen Variablen

```

1 def addImport(String newImport) {
2     if (!imports.contains(newImport)) {
3         imports.add(newImport)
4     }
5 }
6 def addGlobalVar(String globalVar) {
7     if (!globalVars.contains(globalVar))
8         globalVars.add(globalVar)
9 }

```

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

Listing 21: Generierung eines Innercomplex

```

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

```

In der Methode *compileComplex* wird festgelegt, dass sich die *GUI-Komponenten* auf einem *BorderLayout* anordnen. Wenn die *Layout-Datei* verwendet wird, muss der *Generator* aus dem Inhalt dieser Datei auf einen entsprechenden *Layoutcontainer* schließen. Zum Abschluss der Methode *genRest* werden zwei weitere Methoden aufgerufen. Die Erste (*description.init*) generiert

die im Konstruktor aufgerufene Methode *init*. Die Andere (*genGlobalVars*) ist für die Generierung der globalen Variablen zuständig.

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

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

```

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

```

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

Listing 23: Generierung eines Labels

```

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

```

```

17             «ENDIF»
18         «ENDIF»
19         ' ' '
20 ' ' '

```

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

Der Quellcode für den *Tree* wird ähnlich generiert. Wenn das Input-Modell des *Trees* nicht definiert ist, muss ein Standardwert dafür eingesetzt werden. Des Weiteren muss für den Baum ein *CellRenderer* definiert werden (siehe Listing 24).

Listing 24: Generierung eines Trees

```

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

```

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

Die generierten Dateien (*GuiInhaltsbaum.java* und *GuiVerweisebaum.java*) befinden sich auf dem beiliegenden Datenträger (siehe *Anlage*) im Projekt *Explorer*.

Abbildung 35 zeigt die beiden *GUIs*, welche bei der Ausführung der generierten Dateien im Kontext von *profil c/s* erzeugt werden.



Abbildung 35.: Generierte GUI des Inhalts- und Verweisebaums

Um die *Explorer-GUI* zu generieren, müssen *Inhalts-* und *Verweisebaum* zusammen mit einem austauschbaren Bereich in einem *GUI-Skript* definiert werden (siehe Listing 25).

Listing 25: GUI-Skript für die Explorer-GUI

```

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

```

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

Listing 26: Generierung eines Windows

```

1 def compileWindow(UIDescription description) '''
2     «addImport("import DE.data_experts.jwamc.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 Layoutcontainer ein *BorderLayout* verwendet. Um die Anordnung der *Trees* wie gewünscht zu erhalten, wird ein weiterer Layoutcontainer benötigt. Da sich diese Information auf das Layout bezieht, muss sie von der *Layout-Datei* geliefert werden.

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

Listing 27: Generierung der Einbindung anderer GUI-Skripte

```

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

```

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

Listing 28: Generierung einer Interchangeable-Komponente

```

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

```

Die generierten Dateien *GuiInhaltsbaum.java*, *GuiVerweisebaum.java* und *GuiExplorer.java* sind auf dem beiliegenden Datenträger (siehe *Anlage*) zu finden. Abbildung 39 (in Anhang B) ist die *GUI* zu entnehmen, welche durch die Klasse *GuiExplorer.java* erzeugt wird.

FP-Klassen

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

Listing 29: Generierung der FP-Klasse

```

1 def genFpSource() '''
2     «addImport("import DE.data_experts.jwam.tools.FpObject;")»
3     «addImport("import
4         de.jwam.handling.toolconstruction.request.RequestHandler;")»
5     public class «specificFilename» extends FpObject{
6         public «specificFilename»( RequestHandler parent ) {
7             super( parent , "«descriptionname»" );
8         }
9     }
10 '''

```

IP-Klassen

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

Listing 30: Generierung der IP-Klasse

```

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

```

```

11         try {
12             initCommands();
13             initIAFs( iafContext );
14         }
15         catch ( Exception ex ) {
16             ExceptionManager.getManager().addAndShow( ex );
17         }
18     }
19     «description.genIAF»
20     «description.genCommands»
21     «description.genCommandMethods»
22     «genGlobalVars»
23 }
24 '''

```

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

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

Listing 31: Generierung der Interaktionsformen

```

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

```

Für die Standardinteraktionsformen muss evaluiert werden, um welche *GUI-Komponente* es sich handelt.

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

Listing 32: Generierung der Standardinteraktionsformen von Trees

```

1 def compileTreeStandardIAF(TreeDefinition definition) '''
2         «definition.id.genIAFAActivator»
3         «definition.id.genIAFTree»
4     '''
5 def genIAFTree(String id) '''
6         «genIAFSource("DE.data_experts.jwam.gui.interaction.IfTree",
7             "de.jwamx.technology.iafpf.command.cmdSelect", id)»
8     '''
9 def genIAFAActivator(String id) '''
10         «genIAFSource("de.jwamx.technology.iafpf.interaction.ifActivator",
11             "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 33).

Listing 33: Generierung einer Interaktionsform

```

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

```

```

19         commands.put(id+commandName, commandName);
20         return ""
21     }

```

Die Einzelheiten dieser Methode sind unter anderem abhängig von den Konventionen, die bzgl. der Namensgebung in der *data experts GmbH* getroffen wurden. Der Aufruf der Methode *addCommand* ist für weitere Generierungen von Belang.

Nach Abschluss der Generierung des Quellcodes zur Bestimmung der Interaktionsformen und Kommandos, müssen den Kommandos entsprechende Methoden zugeordnet werden. Über die Liste *commands*, die durch den Aufruf der Methode *addCommand* (siehe Listing 33) gefüllt wird, können die Kommandos referenziert werden. Die Methode *genCommands* ist für die beschriebene Zuordnung zuständig (siehe Listing 34).

Listing 34: Generierung der Kommandoinitialisierung

```

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

```

Die genauen Bezeichnungen ergeben sich wiederum aus den Konventionen für die Bezeichnungen innerhalb des MCF der *data experts GmbH*.

Um eventuelle Fehler bei der Ausführung des Programms zu vermeiden, müssen die Methoden, die vom generierten Quellcode verwendet werden, ebenfalls erzeugt werden. Dafür ist die Methode *genCommandMethods* zuständig, in der die Liste *commands* noch einmal benutzt wird, um die Methodenrümpfe zu erzeugen (siehe Listing 35). Die Implementierung der generierten Methoden muss der Entwickler übernehmen.

Listing 35: Generierung der Aktionen bei einer Interaktion

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

Der beiliegende Datenträger enthält die Generatorklasse (siehe *Anlage*). In den Methoden zur Generierung von *GUI-Komponenten* für die *GUI-Klassen*, sind die layoutbestimmenden Codeabschnitte durch entsprechende Kommentare gekennzeichnet.

10. Zusammenfassung und Ausblick

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

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

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

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

Die Ausdruckskraft der *GUI-DSL* (siehe Anforderung AA5) kann nicht objektiv bewertet werden, weswegen zur Umsetzung dieser Anforderung keine allgemeine Aussage getroffen werden kann.

Die Anforderungen an die Sprache (siehe Anforderung AS1 bis AS6) wurden bis auf Anforderung AS3 und AS5 vollständig umgesetzt.

Die Beschreibung von Interaktionen erwies sich als schwierig, da die dazugehörigen Aktionen, nur schwer abstrahiert werden können. Von daher konnte diese Anforderung nicht vollständig umgesetzt werden. Es können lediglich die Interaktionsformen mit *GUI-DSL* beschrieben werden.

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

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

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

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

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

Das Problem, dass die verwendeten *GUI-Frameworks* nicht aktuell sind (siehe Problem P1), wird durch die *GUI-DSL* alleine nicht gelöst. Entscheidungen über die Integration neuer Frameworks müssen von der *data experts GmbH* getroffen werden.

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

Das Pflegen der korrekten Bezeichnungen in den *IP-* und *GUI-Klassen* (siehe

Routinearbeit *R1*) könnte beispielsweise vom *Generator* übernommen werden. Dies wurde mit den Umsetzungen in Kapitel 9 realisiert und kann in den generierten Klassen, die sich auf dem beiliegenden Datenträger befinden, nachvollzogen werden.

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

Die letztgenannte Routinearbeit (*R3*) wird vorerst unumgänglich bleiben, da die Pflege der Attribute der *GUI-Komponenten* weiterhin innerhalb der Komponentendefinition oder über die *Properties-Dateien* erfolgen muss.

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

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

Zur Steigerung der Effektivität, wäre es von Vorteil, würde eine beschriebene *GUI* vor der Generierung getestet werden. Bei der Entwicklung eines solchen Test-Konzepts ist auch die Entwicklung eines grafischen Editors für die *GUI-DSL* in Betracht zu ziehen. Damit wäre es nicht nötig, dass das Framework, welches zur Entwicklung der *GUI-DSL* verwendet wird, einen Editor mit anpassbaren Validierungen bereitstellt.

Für einen solchen Test wären auch die *Layout-Dateien* relevant. Der Aufbau dieser Dateien muss noch festgelegt werden. Die *data experts GmbH* könnte dafür bestehende Sprachen wie CSS verwenden, oder definiert eine neue *DSL*, mit der es ausschließlich möglich ist das Layout zu beschreiben.

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

Zusammenfassend ist zu sagen, dass zum jetzigen Zeitpunkt noch unklar ist, ob die *GUI-DSL* in der *data experts GmbH* eingesetzt werden kann. Zuerst sollte geprüft werden, ob der *Generator* in der Lage ist, alle *GUI-Komponenten* zu erzeugen. Darüber hinaus wird die Weiterentwicklung des *Generators* für das *MCF* und die Implementierung von *Generatoren* für andere *GUI-Frameworks* zeigen, ob die *GUI-DSL* ausreichend abstrakt für *profil c/s* und die verwendeten *GUI-Frameworks* ist. Das Konzept, welches im Zuge dieser Arbeit entwickelt wurde (siehe Kapitel 5), ist im direkten Vergleich mit dem Konzept des *MCF* vielversprechend. Das beweist die Umsetzung der allgemeinen Anforderungen. Allerdings ist aufgrund des Entwicklungsstandes an eine kurzfristige Ablösung des *Multichannel-Frameworks* durch die *GUI-DSL* nicht zu denken.

A. Zuwendungsblatt für den Web- und Standalone-Client

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

Zuwendungsblatt Hilfe

Anteils-/Festbetragsfinanzierung Mengenfinanzierung

Förderfähige Ausgaben lt. Kostenplan: 290.000,00

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

Bemerkungen:

Bemerkungen

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

Abbildung 36.: Standalone-Client: Zuwendungsblatt (vgl. [dat07])

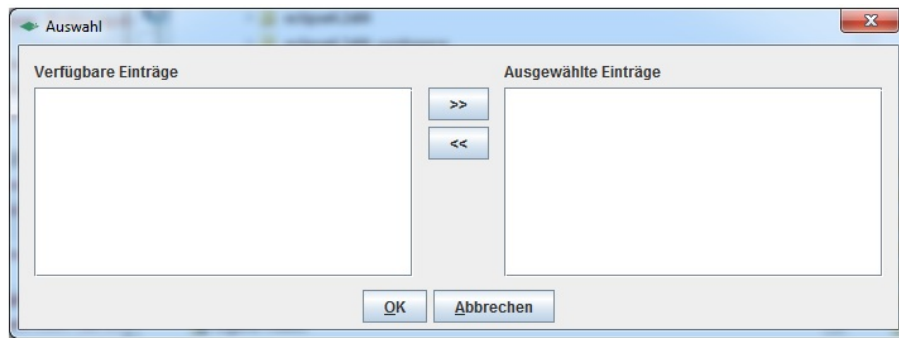


Abbildung 38.: Multiselection-Komponente

B. Generierte Explorer-GUI

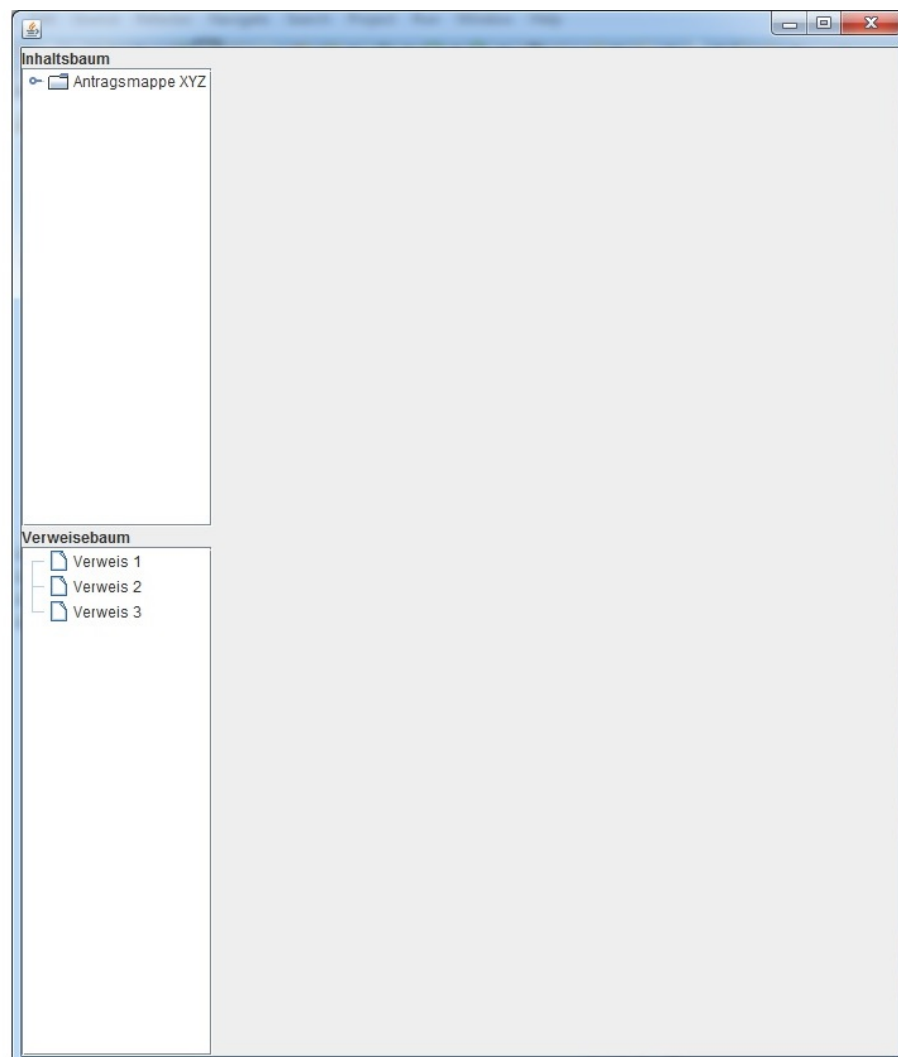


Abbildung 39.: Generierte Explorer-GUI

Inhalt des beiliegenden Datenträgers

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

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

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

Glossar

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

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

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

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

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

Inhaltsbaum - enthält die Dokumente die in dieser Antragsmappe vorliegen.

InVeKoS - bezeichnet das *Integriertes Verwaltungs- und Kontrollsystem*. Mit einem solchen Systemen wird im Allgemeinen sichergestellt, dass die durch den Europäischen Garantiefonds für die Landwirtschaft finanzierten Maßnahmen ordnungsgemäß umgesetzt wurden. Im Speziellen bedeutet dies die Absicherung von Zahlungen, die korrekte Behandlung von Unregelmäßigkeiten und die Rückforderung von zu unrecht gezahlten Beiträgen (vgl. [Gen14]).

Multiselection-Komponente - stellt in *profil c/s* ein Werkzeug zur Auswahl mehrerer Objekte dar. Es wird zwischen zwei Containern unterschieden. Der Container auf der linken Seite enthält der die Objekt, die zur Auswahl stehen und der Container auf der rechten Seite enthält die Objekte, die bereits ausgewählt sind. Mit den in der Mitte befindlichen Schaltflächen ist es möglich, die Objekte von einem Container in den anderen zu navigieren.

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

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

Traditionelle GUI-Entwicklung - beschreibt die *GUI-Entwicklung* unter Verwendung von *traditionellen GUI-Toolkits*. Bei diesen Toolkits wird der Aufbau der *GUI* genau beschrieben. Für die Interaktion mit den *GUI-Komponenten*, werden Listener implementiert, die auf andere Events reagieren, die wiederum von anderen Komponenten erzeugt wurden. Events können zu unterschiedlichen Zeitpunkten generiert werden, wobei die Reihenfolge, wie sie bei anderen Komponenten ankommen, nicht festgelegt ist (vgl. [KB11]).

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

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

Verweisebaum - enthält Verweise auf Dokumente, die mit der Antragsmappe in Verbindung stehen.

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

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

Zuwendungsberechner - ist ein *Werkzeug* innerhalb von *profil c/s*. „Mit diesem Werkzeug kann der Sachbearbeiter die Zuwendung, die dem Antragsteller bewilligt werden soll, nach einem standardisierten Verfahren berechnen [...] . Das Ergebnis wird im Zuwendungsblatt dokumentiert, das auch später mit demselben Werkzeug angesehen werden kann“ [dat07].

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

Literaturverzeichnis

- [Aho08] AHO, ALFRED V.: *Compiler: Prinzipien, Techniken und Werkzeuge*. Pearson Studium Informatik. Pearson Education Deutschland, 2008.
- [BCK08] BRAUER, JOHANNES, CHRISTOPH CRISEMAN und HARTMUT KRISEMAN: *Auf dem Weg zu idealen Programmierwerkzeugen - Bestandsaufnahme und Ausblick*. Informatik Spektrum, 31(6):580–590, 2008.
- [BCK10] BRAUER, JOHANNES, CHRISTOPH CRISEMAN und HARTMUT KRISEMAN: *Eine DSL für Harel-Statecharts mit PetitParser*. Arbeitspapiere der Nordakademie. Nordakademie, 2010.
- [BPL13] BACIKOVÁ, MICHAELA, JAROSLAV PORUBÁN und DOMINIK LAKATOS: *Defining Domain Language of Graphical User Interfaces*. In: *OASIS-OpenAccess Series in Informatics*, Band 29. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.
- [DAT] DATACOM BUCHVERLAG GMBH: *GUI (graphical user interface)*. URL: www.itwissen.info/definition/lexikon/graphical-user-interface-GUI-Grafische-Benutzeroberflaeche.html. Zuletzt eingesehen am 29.01.2015.
- [dat07] DATA EXPERTS GMBH: *Detaillkonzept ELER/i-Antragsmappe*, Januar 2007. Zuletzt eingesehen am 20.12.2014.
- [dat14] DATA EXPERTS GMBH: *Förderantrag*. Profil Wiki der data experts GmbH, März 2014. Zuletzt eingesehen am 20.12.2014.
- [Dir00] DIRNBAUER, KURT: *Usability*. Libri Books on Demand, 2000.

- [DM14] DANIEL, FLORIAN und MARISTELLA MATERA: *Model-Driven Software Development*. In: *Mashups, Data-Centric Systems and Applications*, Seiten 71–93. Springer Berlin Heidelberg, 2014.
- [FP11] FOWLER, MARTIN und REBECCA PARSON: *Domain-Specific Languages*. Addison-Wesley, 2011.
- [Gal07] GALITZ, WILBERT O.: *The Essential Guide to User Interface Design - An Introduction to GUI Design Principles and Techniques*. John Wiley Sons, New York, 2007.
- [Gen14] GENERALDIREKTION LANDWIRTSCHAFT UND LÄNDLICHE ENTWICKLUNG: *Das Integrierte Verwaltungs- und Kontrollsystem (InVeKoS)*. URL: http://ec.europa.eu/agriculture/direct-support/iacs/index_de.htm, November 2014. Zuletzt eingesehen am 20.12.2014.
- [Gho11] GHOSH, DEBASISH: *DSLs in Action*. Manning Publications Co., 2011.
- [Gun13] GUNDERMANN, NIELS: *Prototypische Implementierung eines JavaFX-Channels zur Integration ins Multichannel-Framework der deg*, 2013. Praxisbericht.
- [Gun14a] GUNDERMANN, NIELS: *Entwicklung einer Grammatik für eine DSL mit xText am Beispiel einer Sprache zur Definition von Pflichtprüfungen in profil c/s*, 2014. Praxisbericht.
- [Gun14b] GUNDERMANN, NIELS: *Prototypische Implementierung eines JavaFX/Web-Channels zur Integration ins Multichannel-Framework der deg*, 2014. Praxisbericht.
- [Hed12] HEDTSTUECK, ULRICH: *Einführung in die Theoretische Informatik*, Band 5. Auflage. Oldenbourg Verlag, 2012.
- [Hof06] HOFER, STEFAN MICHAEL: *Refactoring-Muster der WAM-Modellarchitektur*. Diplomarbeit, FH Oberösterreich, 2006.

- [KB11] KRISHNASWAMI, NEELAKANTAN R. und NICK BENTON: *A Semantic Model for Graphical User Interfaces*. Microsoft Research, September 2011.
- [LW07] LU, XUDONG und JIANCHENG WAN: *Model Driven Development of Complex User Interface*. In: *MoDELS'07 Workshop on Model Driven Development of Advanced User Interfaces*. Shandong University, 2007.
- [MHP99] MYERS, BRAD, SCOTT E. HUDSON und RANDY PAUSCH: *Past, Present and Future of User Interface Software Tools*. Technischer Bericht, Carnegie Mellon University, September 1999.
- [Oraa] ORACLE: *Class GridBagLayout*. URL: <https://docs.oracle.com/javase/7/docs/api/java/awt/GridBagLayout.html>. Zuletzt eingesehen am 20.12.2014.
- [Orab] ORACLE: *Swing*. URL: <http://docs.oracle.com/javase/8/docs/technotes/guides/swing/>. Zuletzt eingesehen am 20.12.2014.
- [Pec14] PECH, VACLAV: *Can MPS be integrated in Eclipse as Eclipse Plugin similar to Xtext?* URL: <http://forum.jetbrains.com/thread/Meta-Programming-System-1033>, Oktober 2014. Zuletzt eingesehen am 12.01.2015.
- [PSV13] PECH, VACLAV, ALEX SHATALIN und MARKUS VÖLTER: *JetBrains MPS as a tool for extending Java*. In: *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, Seiten 165–168. ACM, 2013.
- [RDGN10] RENGGLI, LUKAS, STEPHANE DUCASSE, TUDOR GÎBRA und OSCAR NIERSTRASZ: *Practical Dynamic Grammars for Dynamic Languages*. In: *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, 2010.
- [Roa09] ROAM, DAN: *The Back of the Napkin (Expanded Edition) - Solving Problems and Selling Ideas with Pictures*. Penguin, New York, Expanded Auflage, 2009.

- [Sau03] SAUER, JOACHIM: *Gestaltung von Anwendungssoftware nach dem WAM-Ansatz auf mobilen Geräten*. Diplomarbeit, Fachbereich Informatik, Universität Hamburg, 2003.
- [Sau10] SAUER, JOACHIM: *Architekturzentrierte agile Anwendungsentwicklung in global verteilten Projekten*. Doktorarbeit, Universität Hamburg, 2010.
- [Sch05] SCHMELZER, ROBERT FRANZ: *Realisierung teilautomatisierter Prozesse durch die Kombination von Ablaufsteuerung und unterstützter Kooperation*. Diplomarbeit, FH Oberösterreich, 2005.
- [Sch07] SCHMID, BENJAMIN: *Get your wingS back!* URL: <http://jaxenter.de/artikel/Get-your-wingS-back>, Dezember 2007. Zuletzt eingesehen am 20.12.2014.
- [SKNH05] SUKAVIRIYA, NOI, SANTHOSH KUMARAN, PRABIR NANDI und TERRY HEATH: *Integrate Model-driven UI with Business Transformations: Shifting Focus of Model-driven UI*. IBM T.J. Watson Research Center, Oktober 2005.
- [Ste07] STECHOW, DIRK: *JWAMMC - Das Multichannel-Framework der data-experts gmbh*. Vortrag in der data experts GmbH, Dezember 2007.
- [SV06] STAHL, THOMAS und MARKUS VÖLTER: *Model-Driven Software Development*. John Wiley & Sons Ltd, Februar 2006.
- [The14] THE ECLIPSE FOUNDATION: *Xtext Documentation*, September 2014.
- [Use12] USERLUTIONS GMBH: *3 Gründe, warum gute Usability wichtig ist*. URL: <http://rapidusertests.com/blog/2012/04/3-gute-grunde-fuer-usability-tests/>, April 2012. Zuletzt eingesehen am 20.12.2014.
- [VAC⁺08] VOGEL, OLIVER, INGO ARNOLD, ARIF CHUGHTAI, EDMUND IHLER, TIMO KEHRER, UWE MEHLIG und UWE ZDUN: *Software-*

- Architektur*. Springer Science Business Media (Berlin Heidelberg), 2008.
- [VBK⁺13] VÖLTER, MARKUS, SEBASTIAN BENZ, LENNART KATS, MATS HELANDER, EELCO VISSER und GUIDO WACHSMUTH: *DSL Engineering*. CreateSpace Independent Publishing Platform, 2013.
- [VC09] VÖLTER, MARKUS und LARS CORNELIUSSEN: *Carpe Diem*. Dot-NetPro, 5 2009.
- [Völ11] VÖLTER, MARKUS: *From programming to modeling-and back again*. Software, IEEE, 28(6):20–25, 2011.