technische universität
dortmund

Projektgruppe 631

Responsive and Knowledge-Driven Business Modeling

30. September 2020

Betreuer:
Philip Zweihoff
Barbara Steffen
Stefan Naujokat
Prof. Bernhard Steffen
Prof. Falk Howar

In Kooperation mit:
Schulz Systemtechnik GmbH
Pollhornbogen 18
21107 Hamburg
https://www.schulz.st

Technische Universität Dortmund
Fakultät für Informatik

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation . . . 1

1.2 Ziele der Arbeit . . . 2

1.3 Wissenschaftliche Herausforderungen und Fragestellungen . . . 3

2 Grundlagen **5**

2.1 Modellgetriebene Softwareentwicklung . . . 5

2.1.1 Grundbegriffe . . . 5

2.1.2 Grundlagen und Aufgabe . . . 9

2.1.3 Erhalten eines ausführbaren Programms aus einer DSL . . . 12

2.1.4 Unified Modelling Language . . . 14

2.2 Domain-Specific Languages . . . 15

2.2.1 Textuelle Modellierungssprachen . . . 16

2.2.2 Grafische Modellierungssprachen . . . 17

2.2.3 Gegenüberstellung grafische und textbasierte DSL . . . 20

2.2.4 Auswahl von textlichen oder grafischen DSLs . . . 22

2.3 CINCO . . . 23

2.3.1 Cinco Product Definition . . . 23

2.3.2 Meta Graph Language . . . 23

2.3.3 Meta Style Language . . . 25

2.3.4 Meta Plug-ins . . . 27

2.3.5 Actions . . . 31

2.3.6 Prime-Reference . . . 32

2.4 Ontologie . . . 33

2.4.1 Definition . . . 33

2.4.2 Erstellung einer Ontologie . . . 33

2.4.3 Kriterien für Ontologien . . . 34

2.5 DIME . . . 35

2.5.1 Konzept . . . 35

2.5.2 Code Generierung . . . 35

2.5.3 Modellierung . . . 36
2.5.4 Benutzeroberfläche . . . 39
2.6 Canvas . . . 40
2.6.1 Canvas in einer Organisation . . . 40
2.6.2 Business Model Canvas . . . 41
2.6.3 Die neun Blöcke des Business Model Canvas . . . 41
2.6.4 Vor- und Nachteile des Business Model Canvas . . . 45

3 Konzepte Ontologie 47
3.1 Ontologie . . . 47
3.1.1 Aufbau Firmenstruktur . . . 48
3.1.2 Aufbau Ontologiemodell . . . 51
3.1.3 Erste Schritte in DIME . . . 51
3.1.4 Pattern extrahieren . . . 51
3.1.5 Umbenannt: Prototypen . . . 51
3.2 Pattern . . . 52
3.2.1 Prozess Pattern . . . 52
3.2.2 GUI Pattern . . . 59
3.3 Baumpattern/Baumtypen . . . 64
3.3.1 Definition / Regeln - Bäume im Kontext unserer Webapp / Ontologie 64
3.3.2 Schematische Darstellung der Baumverwaltung . . . 65
3.3.3 Pattern der Baumverwaltung . . . 71
3.3.4 Schematische Darstellung der Baumauswahl . . . 76
3.3.5 Pattern der Baumauswahl . . . 81
3.4 Transformation . . . 86
3.4.1 Modell-zu-Modell Transformation in CINCO . . . 86
3.4.2 Modell-Generatoren . . . 87
3.4.3 ModelProvider . . . 90
3.4.4 Transformationsphasen . . . 91

4 Implementierung Ontologie 97
4.1 DSL . . . 97
4.1.1 Basis der DSL . . . 97
4.1.2 Implementierung der Features in der DSL . . . 98
4.1.3 Custom Checks und Appearances . . . 99
4.1.4 Beispielhafte Umsetzung einer Firmenontologie in der Ontologie-DSL 101
4.1.5 OntologyTool-DSL . . . 102
4.2 Transformator . . . 103
4.2.1 DataModelGenerator . . . 103

4.2.2 Übersicht Prozess- und GUI-Generatoren . . . 104
4.2.3 CrudProcessGenerator . . . 106
4.2.4 OverviewGenerator . . . 108
4.2.5 DetailViewGenerator . . . 109
4.2.6 EditViewGenerator . . . 113
4.2.7 Baumstrukturen . . . 116

5 Konzepte Canvas **121**
5.1 Canvas . . . 121
5.1.1 Aufbau Canvasmodell . . . 121
5.1.2 Prototyp . . . 121
5.2 Pattern . . . 123
5.2.1 Prozess Pattern (Elementtypen) . . . 123
5.2.2 Prozess Pattern (Canvas) . . . 131
5.2.3 GUI Pattern . . . 142

6 Implementierung Canvas **147**
6.1 DSL . . . 147
6.1.1 Basis der Canvas-DSL . . . 147
6.1.2 Implementierung der Features in der DSL . . . 147
6.1.3 Custom Checks und Appearances . . . 149
6.1.4 Beispielhafte Umsetzung eines Canvases in der Canvas-DSL . . . 152
6.2 Transformator . . . 153
6.3 CanvasGeneratorFactory . . . 153
6.4 DataModelGenerator . . . 154
6.5 Übersicht Prozess- und GUI-Generatoren . . . 155
6.5.1 Prozess-Generatoren . . . 155
6.5.2 GUI-Generatoren . . . 158
6.5.3 CanvasGUIGenerator . . . 159
6.5.4 CanvasProcessGenerator . . . 161
6.5.5 InteractAddProcessGenerator . . . 162
6.5.6 SubInteractProcessGenerator . . . 162
6.5.7 ManagementProcessGenerator . . . 163
6.5.8 UpdateAllProcessGenerator . . . 163
6.5.9 CanvasCorrectPriorityProcessGenerator . . . 164
6.5.10 SearchProcessGenerator . . . 164

7 Ergebnisse **167**
7.1 Zusammenarbeit mit der Schulz Systemtechnik GmbH . . . 167
7.2 Überblick . . . 169

7.3 Zusammenfassung und Ausblick 170

A Dokumentation **173**

Abbildungsverzeichnis **222**

Literaturverzeichnis **226**

Kapitel 1

Einleitung

1.1 Motivation

Heutige Märkte unterliegen einem stetigen Wandel. Eine Planung über Jahrzehnte, wie in dem klassischen Geschäftsmodellansatz, wird so insbesondere für junge und Start-up Unternehmen fast unmöglich. Sie sind auf eine schnelle Planung und dynamische Umsetzung eines Geschäftsmodells angewiesen. Sinkende Markteintrittsbarrieren und neue Technologien erleichtern Ihnen dabei den schnellen Markteintritt und unter Umständen sogar das Durchsetzen gegen bereits etablierte Unternehmen, wenn sie schnell auf die veränderlichen Trends und damit einhergehenden Kundenwünsche reagieren können. Dies übt selbst auf erfolgreiche Unternehmen einen hohen Innovationsdruck aus, der noch durch kürzere Produktlebens- und Technologiezyklen der Konkurrenz verstärkt wird. Eine hohe Wandlungsfähigkeit kann so ein entscheidender Erfolgsfaktor für jedes Unternehmen sein.

Aus diesem Grund wird von vielen Unternehmen der *Business Model Canvas* (BMC) von Osterwalder und Pigneur [27] genutzt. Der BMC ist ein standardisiertes Modell, dass es ermöglicht, ganzheitliche Geschäftsmodelle schnell aufzustellen und damit die Wandlungsfähigkeit von Unternehmen voran zu treiben. Die Qualität des erzeugten Geschäftsmodells hängt bislang stark von dem Verständnis der Nutzer ab.

Ziel der Projektgruppe ist es daher, einen *Living Business Model Canvas* (LBMC) zu erzeugen, der den Nutzer gezielt durch den Erstellungsvorgang führt und auf der jeweiligen *Business Ontologie* (siehe Kapitel 2.4) beruht. Um eine gute Grundlage zu schaffen, die auf dem LBMC beruht, soll eine *Domain-Specific Language* (DSL, siehe Kapitel 2.2) zur Modellierung einer Business Ontologie in Zusammenarbeit mit der Firma Schulz Systemtechnik GmbH umgesetzt werden, sodass die Nutzer mithilfe einer grafischen Oberfläche die Grundlage für die Erzeugung einer Business Ontologie selbst modellieren können. Um der Firma, und eventuellen weiteren Nutzergruppen die Nutzung leicht zu machen, wird die so modellierte Applikation als Webapplikation generiert, sodass eine einfache Verwaltung online stattfinden kann. Mit diesem Schritt wird die Organisation des Unternehmens

festgelegt, die nötige Strukturwissen zur Erzeugung eines BMC darstellt. In einer weiteren DSL wird sich dem Prozesswissen gewidmet. Hierbei wird der Nutzer gezielt durch den Erstellungsvorgang eines BMC geleitet und z.B. eine Herangehensweise für das Ausfüllen des BMC aufgezeigt. Außerdem werden mögliche Fragestellungen zu den jeweiligen Zeitpunkten empfohlen.

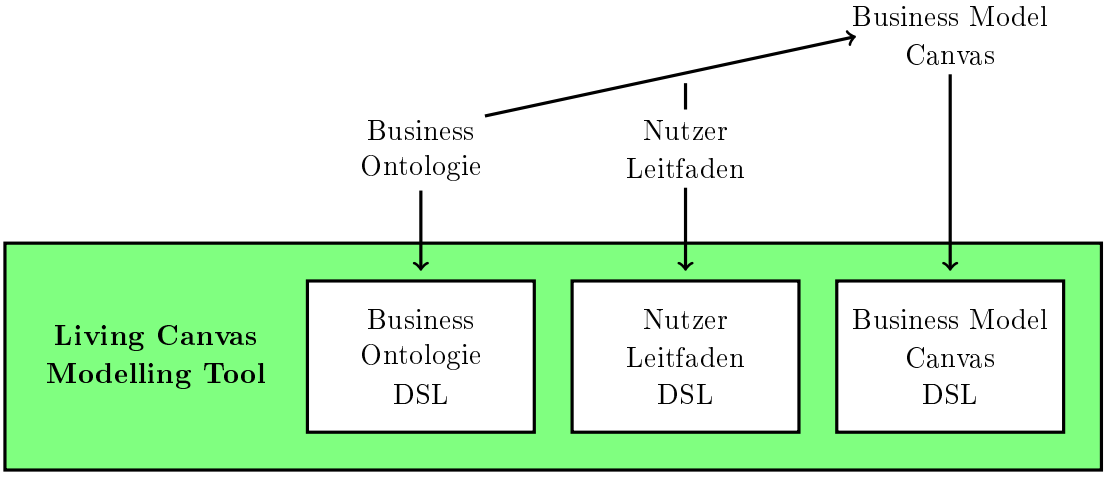


Abbildung 1.1: Konzept: Living Business Model Canvas

Als weiteren Schritt wurde im Laufe der Projektgruppe eine DSL zur Modellierung des BMC, basierend auf der Ontologie, erzeugt. Diese drei DSLs bilden, wie in Abbildung 1.1 zu sehen, den Living Business Model Canvas.

1.2 Ziele der Arbeit

Der Ablauf der Projektgruppe erstreckte sich über zwei Semester. Diese waren wie folgt strukturiert.
In dem ersten Semester war das Ziel die Planung und Erstellung einer DSL für die Darstellung von Business Ontologien. In Zusammenarbeit mit der Firma Schulz Systemtechnik GmbH wurde zunächst, mithilfe von *DIME (DyWA Integrated Modeling Environment*, siehe Kapitel 2.5), eine beispielhafte Webapplikation erstellt, die es dieser Firma ermöglicht, ihre Business Ontologie darzustellen. Aufbauend auf den Erkenntnissen bei der Entwicklung der Webapplikation wurde mithilfe von CINCO (siehe Kapitel 2.3) eine Ontologie DSL entwickelt, dessen Ergebnisse in DIME Modelle transformiert werden können, um damit wiederum eine entsprechende Webapplikation zu generieren.
Im zweiten Semester war das Ziel die Planung und Erstellung einer DSL für den Business Model Canvas. Es wurde eine beispielhafte Webapp entwickelt, die es ermöglicht, die Daten aus der Ontologie dem Canvas hinzuzufügen und parallel weiter an der Entwicklung der Business Ontologie gearbeitet. In Zusammenarbeit mit der Firma Schulz Systemtechnik GmbH wurde ein Konzept ausgearbeitet, wie diese Wahl die weiteren auswählbaren Daten

beeinflussen sollte. Mithilfe der so gewonnenen Erkenntnissen wurde eine Canvas DSL entwickelt, die es ermöglicht, eigene Canvases zu definieren, das Zusammenwirken der zu nutzenden Daten zu definieren und diese zu transformieren und als Webapp zu generieren.

1.3 Wissenschaftliche Herausforderungen und Fragestellungen

Ein wichtiger Punkt bei den Ergebnissen, die erzielt werden sollen, ist, dass ein großes Augenmerk darauf gelegt wird, möglichst viel automatisch zu generieren. Dabei kommen sowohl die Generatoren zum Einsatz, die in der genutzten Software enthalten sind, als auch selbstgeschriebene Generatoren. Dies bietet den großen Vorteil, dass nachträgliche Änderungen in einem Bereich sehr schnell durchgeführt werden können, falls die entsprechenden Generatoren darauf ausgelegt sind, da viele einzelne manuelle Schritte erspart bleiben. Ein Endprodukt, welches komplett darauf ausgelegt ist, generiert zu werden, beinhaltet allerdings auch verschiedene Probleme. Der Entwicklungsprozess kann eine gewisse Eingewöhnungsphase erfordern, da nicht mehr direkt mit ausführbarem Code gearbeitet wird, sondern nur noch mit Code, welcher ausführbaren Code generiert. Selbiges gilt auch für das Arbeiten mit und dem Generieren von Modellen in einer Modellierungsumgebung. Dies führt zusätzlich noch zu veränderten Anforderungen an die Art und Weise des Testens und der Fehlersuche.

Es gibt bereits verschiedene Ansätze, die sich mit der Nutzung von Sprachen befassen, um Codegenerierung oder Modelltransformationen mithilfe von *Language Workbenches* durchzuführen [36]. Damit verbunden sind sowohl umfangreich definierte Vorgaben, sowie erprobte Vorgehen, die Anhaltspunkte für die eigene Arbeit liefern können. Seltener kommt es dagegen vor, dass das Ziel der Language Workbenches nicht das Erhalten von ausführbarem Code, sondern lediglich das Modellieren selbst ist. Das kann dazu führen, dass gewisse Abweichungen von den bekannten Arbeitspattern erforderlich sind, worunter insbesondere Modifikationen und Auslassen von sonst üblichen Schritten zu verstehen sind. Das Herausfinden dieser Unterschiede und die Erstellung von Alternativplänen sind daher auch Aufgaben, die gelöst werden müssen.

Eine weitere Herausforderung ist der Austausch zwischen den beiden an der Entwicklung beteiligten Parteien aufgrund ihrer unterschiedlichen Interessen und Fähigkeiten. Zu den Parteien gehören zum einen die Teilnehmer der PG, welche hauptsächlich Programmierkenntnisse sowie Wissen über die Erstellung von Language Workbenches mitbringen. Die andere Partei sind Vertreter der Firma Schulz Systemtechnik GmbH, welche sich mit technischen Details nicht beschäftigen können oder wollen, aber dafür einen tiefen Einblick in die Prozesse der Firma geben können. Eine effiziente Form der Kommunikation, welche unter anderem durch eine relativ große Entfernung erschwert wird, sowie eine Möglichkeit,

das Auftreten eines *Semantic Gap* zu überwinden, sind also unumgänglich. Genauso sollten Prozesse erdacht werden, um funktionierende Feedbackzyklen zu erhalten.

Kapitel 2

Grundlagen

In diesem Kapitel werden die verschiedenen Grundlagen vorgestellt, welche für den Rest der Arbeit wichtig sind. Dafür wird zuerst auf die theoretischen Grundlagen in Form der modellgetriebenen Softwareentwicklung eingegangen. Danach werden DSLs und Ontologien abwechselnd mit der jeweils passenden Software CINCO und DIME vorgestellt. Am Schluss befinden sich Details zu Canvases und deren Nutzung.

2.1 Modellgetriebene Softwareentwicklung

Beim *Model-Driven Software Engineering* (kurz MDSE, auf deutsch „Modellgetriebene Softwareentwicklung“) geht es um die Erstellung von Modellen, die einen beliebigen Prozess oder Zustand abbilden, sowie der automatischen oder halbautomatischen Transformation dieser Modelle in ausführbaren Maschinencode.
Das MDSE baut auf verschiedenen Konzepten auf, die auch untereinander stark verknüpft sind. Diese werden hier zuerst vorgestellt. Danach wird auf die Grundidee und die Aufgaben des MDSE eingegangen. Zum Schluss wird noch die modellgetrieben Architektur betrachtet und die damit verbundene Modellierungssprache UML als Beispiel dafür genutzt.

2.1.1 Grundbegriffe

Eines der wichtigsten Konzepte des MDSE ist die *Abstraktion*. Abstraktion wird primär dazu genutzt, komplizierte oder komplexe Sachverhalte zu verstehen. Dabei wird versucht, bestimmte Gemeinsamkeiten zu anderen bekannten Themen zu finden und Unterschiede auszublenden.
Abstraktion wird sowohl im Alltag als auch in der Wissenschaft benutzt. Wenn Menschen sich oder anderen etwas beibringen wollen, nutzen sie häufig unbewusst Abstraktion, entweder um relevante Informationen von den weniger wichtigen zu trennen oder um nicht

von der Menge an zu lernenden Inhalten überwältigt zu werden [10]. In der Wissenschaft wird Abstraktion meist unter dem Begriff der *Modellierung* genutzt. Beim Modellieren wird versucht, eine abstrakte Version der Realität zu erzeugen, welche Modell genannt wird. Auch hier liegt der Schwerpunkt wieder an der Auswahl der Details, die abstrahiert werden sollen.
Ein simples Beispiel für das Erhalten und die Nutzung eines Modells wäre die Betrachtung eines Gegenstandes im Hinblick auf bestimmte Aspekte und den Vergleich mit anderen Gegenständen. Zu diesem Gegenstand werden sich nur diese Informationen gemerkt und dann bei allen zu vergleichenden Gegenständen ebenso nur diese Informationen betrachtet, sodass das Vergleichen stark vereinfacht wird. Die betrachteten Informationen wären hier dann das Modell der Gegenstände.
Allgemein gibt es zwei Gebrauchsarten von Modellen: Ein beschreibendes Modell kann dazu genutzt werden, die Realität zu beschreiben. Ein vorschreibendes Modell dagegen stellt etwas auf, was noch gar nicht existiert. Das Modell kann dann als Anleitung für etwas genutzt werden und dabei unter anderem spezifizieren, wie sich das Endprodukt verhalten soll bzw. welche Eigenschaften es haben soll. So ein Modell bietet dann die Grundlage, um die entsprechenden Inhalte korrekt generieren zu können [10].
Die Nutzung von Modellen in der Softwareentwicklung ist nicht nur effektiv, sondern auch wichtig. Die Größe von Software steigt mit den Aufgaben, die sie lösen soll, und den Funktionalitäten, die sie bereitstellt, an. Mit zunehmender Größe steigt auch die Komplexität der Software, was dazu führt, dass sie ab einem bestimmten Punkt nicht mehr ohne Hilfsmittel als Ganzes erfasst werden kann. Dieses Problem lässt sich durch eine Aufteilung in die einzelne Aufgaben- oder Themenbereiche und eine Darstellung in verkürzter oder abstrakter Form lösen. Dabei ist es sinnvoll, technische Details wegzulassen, was zusätzlich noch den Vorteil hat, dass die Möglichkeit besteht, mit anderen Beteiligten über die Software reden zu können, da nach einer gut durchgeführten Abstraktion nur noch relevante und leicht verständliche Inhalte übrig bleiben. Zu diesen Personengruppe gehören sowohl die Kunden und Nutzer der Software, als auch Experten auf dem Fachgebiet des Themas der Software, die in der Lage sind, mit den Entwicklern inhaltliche Fragen zu klären, sich aber nicht mit den technischen Details beschäftigen wollen oder können. Selbst unerfahrenen Entwicklern wird durch die reduzierte Menge an Komplexität die Möglichkeit gegeben, effizienter an der Software zu arbeiten, wenn die zu entwickelnden Teile sinnvoll von anderen abgekapselt wurden [10].

Gemeinsam mit den Modellen bilden *Transformationen* die Grundlage für das MDSE. Eine Transformation wird dazu genutzt, um Informationen von einem Modell auf ein anderes abzubilden. Bei diesem Vorgang werden normalerweise gezielt Informationen ausgelassen, wodurch meist weniger umfangreiche Modelle entstehen. Genauso ist es allerdings auch möglich, dass das aus der Transformation entstehende Modell mit bestimmten Informatio-

nen angereichert wird. Dies passiert insbesondere dann, wenn bestimmte Annahmen bei der Planung der Transformation getroffen werden können, z.B. wenn das transformierte Modell Informationen enthalten muss, mit denen sich der Ersteller des Ausgangsmodells nicht auseinandersetzen soll.

Im MDSE gilt der Grundsatz, dass allgemein alles ein Modell ist. Während dies insbesondere bei Modellen selbsterklärend ist, ist es auch für Transformationen und Regeln anwendbar. Eine Transformation kann als eine Menge von konkreten Änderungen an einem anderen Modell beschrieben werden, Regeln dagegen könnten als alle möglichen und erlaubten Transformationen modelliert werden. Dies führt dazu, dass die *Modellierungssprache* selbst als Modell bezeichnet werden kann. Dieser Vorgang wird *Metamodellierung* genannt und kann rekursiv wiederholt werden. Dafür würde das Metamodell der Modellierungssprache, also das Modell des Sachverhalts, gebildet werden, welches Meta-Metamodell genannt wird. Meistens ist ein Meta-Metamodell in der Lage, sich selbst zu beschreiben, was bedeutet, dass nicht noch weiter abstrahiert werden muss. Um herauszufinden, ob ein Modell das Metamodell eines anderen Modells ist, müssen sich alle Objekte des zweiten Modells durch Objekte des möglichen Metamodells beschreiben lassen, wie in Abbildung 2.1 zu sehen ist.

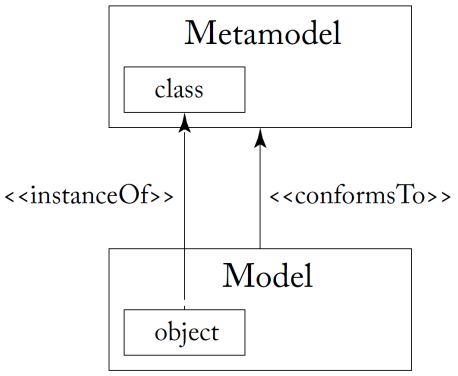


Abbildung 2.1: Anforderung an das Metamodell [10]

Transformationen werden meist händisch erstellt, wobei genauso die Möglichkeit besteht, diese durch Generierung automatisiert zu erhalten. Damit dies möglich ist, müssen die Modelle auf der Metamodellebene definiert sein, damit alle nötigen Metainformationen und Rahmenbedingungen über die Modelle verfügbar sind, welche für die Automatisierung benötigt werden [10].

Modellierungssprachen

Metamodelle können die Grundlage von Sprachen bilden, welche zur Modellierung oder Programmierung genutzt werden. Eine Modellierungssprache besteht aus einer *abstrakten Syntax*, einer *konkreten Syntax* und einer *Semantik*. Da diese drei Elemente aufeinander aufbauen und voneinander abhängig sind, sind sie alle für die Definition einer Sprache sowie deren Sinn und Nutzen notwendig [10].

Die abstrakte Syntax gibt vor, welche Struktur die Modellierungssprache hat. Damit legt sie den grundlegenden Aufbau der Regeln einer Sprache fest. Dazu gehört primär, welche Klassen von Syntaxbausteinen in welcher Situation verwendet werden dürfen.
Die konkrete Syntax beschreibt, auf welche Weise genau die Modellierungssprache dargestellt werden kann. Sie gibt alle nutzbaren Bauteile der Sprache vor und weist sie den entsprechenden Klassen der abstrakten Semantik zu. Sie kann auch als eine Art allgemeine Anleitung zur Verwendung der Sprache genutzt werden. Es gibt sie sowohl in textueller als auch in grafischer Form, wobei im letzten Fall ein von ihr dargestelltes Modell normalerweise die Form eines Diagramms hat. Sie muss immer in direkter Abhängigkeit von der abstrakten Syntax definiert werden.
Eine Abwägung, die beim Design der konkreten Syntax getroffen werden muss, ist die Komplexität der Sprache. Ist sie gering, wird es leichter, die Sprache zu lernen. Dadurch kann es allerdings nach dem Erlernen der Sprache mühevoller sein, gewisse Dinge in ihr darzustellen. Komplexe Sprachen mit mächtigen Sprachkonstrukten sind dagegen potentiell schwieriger zu erlernen, sollten aber nach dem Verinnerlichen der Sprache die Arbeitseffizienz erhöhen können.
Die Semantik wird dazu benutzt, die Bedeutung der mit der konkreten Syntax erstellten Texte oder Grafiken auszuwerten. Dies kann sowohl für einzelne Elemente, als auch für die Kombination aus mehreren Elementen nach den Regeln der abstrakten Syntax geschehen. Eine genaue Definition der Semantik ist wichtig, damit die Sprache richtig benutzt und verstanden werden kann.
Die Definition der Semantik kann noch weiter aufgeteilt werden. Die statische Semantik gibt die Regeln an, die sicherstellen, dass die Inhalte der Syntax überhaupt einen Sinn ergeben können; z. B. durch Vorschriften für die Verwendung von Daten mit bestimmten Typen. Die Ausführungssemantik bezieht sich auf die Bedeutung bei der Ausführung [35]. Semantiken können auf unterschiedliche Arten definiert werden. Eine Möglichkeit ist, dass sie alle Inhalte in mathematische Gleichungen übersetzen. Alternativ dazu können sie als Interpreter benutzt werden, der die Bedeutung der Sprache in das Verhalten des Modells überträgt. Außerdem können sie auch die Bedeutung der Sprache auf eine andere Sprache abbilden, welche selbst eine wohldefinierte Semantik besitzt, um diese dann die weitere Übersetzung durchführen zu lassen.

Um aus einem Modell ein nutzbares System zu machen, muss dieses Modell nach Durchlaufen der Transformationsschritte ausführbar sein. Dies ist eigentlich genau dann der Fall, wenn die für den Betrieb relevanten Semantiken vollständig spezifiziert wurden. In der Praxis kann es allerdings auch zwei andere Situationen geben. Zum einen kann die Ausführungsumgebung in der Lage sein, gewisse Lücken in einer noch nicht vollständig definierten Semantik selbst aufzufüllen, wodurch die nötige Vollständigkeit des Modells erreicht wird. Zum anderen können auch selbst sehr genau und ausführlich spezifizierte Modelle noch

nicht ausführbar sein, wenn die Anwendung, die sie auszuführen soll, entweder komplett fehlt oder nicht vollständig fertiggestellt wurde. Genauso kann die Anwendung noch nicht an das aktuelle Modell angepasst worden sein, z.B. wenn sie auf einer neuen Plattform ein bereits bestehendes Modell benutzen soll [10].

2.1.2 Grundlagen und Aufgabe

Das MDSE kann als Methodik angesehen werden, bei der Software durch die Nutzung von Modellierung erstellt wird.

Die wichtigsten Konzepte des MDSE sind Modelle und Transformationen, also Abbildungen von und zu Modellen. Während allgemein gesagt werden kann, dass sich mithilfe von Algorithmen und Datenstrukturen Computerprogramme schreiben lassen, kann analog dazu festgestellt werden, dass sich aus Modellen und dazu passenden Transformationen Software herstellen lassen kann. Die dafür nötige Notation wird *Modellierungssprache* genannt. Jede Modellierungssprache hat eine Menge an Regeln, welche festlegen, wie die Modellierungssprache benutzt werden kann, damit Modelle und Transformationen nur sinnvoll miteinander in Beziehung gesetzt werden können. Die Mittel, mit denen aus der Modellierungssprache eine nutzbare Software hergestellt werden kann, sind zum einen eine IDE, in der Konzepte den Regeln entsprechend aufgeschrieben werden können, sowie zum anderen ein Compiler oder ein Interpreter, der die aufgeschriebenen Konzepte ausführen oder ausführbar machen kann.

Die Hauptaufgabe des MDSE ist die Automatisierung der Softwareentwicklung durch die Nutzung von modellorientierten Lösungsansätzen.

Die Grundidee bei dem MDSE ist, dass ein Modell in kleinen Schritten so verändert werden kann, dass es am Ende eine Form hat, aus der ausführbarer Programmcode generiert werden kann. Diese Schritte, also die Transformation von einem Modell zu einem anderen, sollen möglichst automatisch passieren. Vor allem bei neuen Projekten wird die Automatisierung allerdings erst schrittweise hinzugefügt, sodass manuelle Änderungen von den Entwicklern sinnvoll und nötig sind, um die Modelle für eine weitere Transformation vorzubereiten [10].

Die Vorteile dieses Ansatzes sind, dass die Modelle am Anfang der Transformationskette wenig bis gar keine technischen Details enthalten, was die Kommunikation zwischen Entwicklern und Kunden, aber auch innerhalb des Teams, deutlich erleichtern kann. Die bessere Kommunikation zwischen den Entwicklern kann sich dann in Form von kürzeren Arbeitszeiten aufgrund von weniger Fehlern aufzeigen. Genauso reduziert sie Fehlplanungen, da die Kunden an den Modellen besser darstellen können, was sie wirklich benötigen.

Sowohl die Implementierung als auch die Konzeptualisierung des MDSE ist in verschiedene Ebenen aufgeteilt [10]. Die Implementierung besteht aus der Modellebene, der Realisierungsebene und der Automatisierungsebene. Auf der Modellebene wird die Definition der Modelle vorgenommen. Auf der Realisierungsebene befindet sich der Code, welcher den Inhalt der Modelle nutzen kann. Auf der Automatisierungsebene wird der Prozess der Abbildung von der Modellebene zur Realisierungsebene durchgeführt.
Die Konzeptualisierung besteht aus der Anwendungsebene, der Anwendungsdomain-Ebene und der Metaebene.

- Anwendungsebene: Hierzu gehören die konkreten Modelle, die Transformationsregeln sowie alle erstellten Codefragmente.
- Anwendungsdomain-Ebene: Sie beinhaltet die Modellierungssprache, die Transformationsdefinitionen sowie die Plattform, auf welcher der generierte Code ausgeführt werden kann
- Metaebene: Hier befindet sich das Metamodell der Modellierungssprache sowie die Sprache für die Transformationsdefinition.

Diese drei Ebenen lassen sich direkt auf die Ideen des Modells, dem Metamodell und dem Meta-Metamodell übertragen.
Ein großer Vorteil bei dieser Art der Softwareerstellung ist die Unabhängigkeit des Modells von der genutzten Zielplattform. Ändert sich die Zielplattform, reicht es aus, wenn die für die Zielplattform relevanten Transformationsregeln geändert werden. Auf diese Weise können auch mehrere Transformationsregeln für verschiedene Plattformen parallel zueinander existieren, wodurch das Modell nicht angepasst werden muss und somit eine Nutzung von mehreren Plattformen ermöglicht wird.

Bei der Erstellung von Software mithilfe des MDSE-Ansatzes sollte zwischen dem Problembereich und dem Lösungsbereich unterschieden werden. Während des Entwicklungsprozesses wird sich in einer Analysephase durch die Inhalte des Problems gearbeitet. Dabei müssen die einzelnen Objekte des Problems sowie ihre Eigenschaften und Zusammenhänge zwischeneinander dargestellt werden. Dies kann dann als eine feste Basis benutzt werden, um die Kommunikation über das Thema zu vereinfachen. Für den Lösungsbereich müssen zuerst alle benötigten Anforderungen gesammelt werden, um zu ermitteln, was am Ende das Ergebnis sein soll. Danach wird in der Designphase beschlossen, wie die Zielverfolgung auszusehen hat. Aus den Ergebnissen dieser beiden Phasen kann dann Software hergestellt werden.
Das MDSE ist in der Lage, sich beiden Bereichen gleichermaßen anzunehmen und entsprechende Lösungen für diese zu finden. Als Gegenüberstellung dazu könnte eine beliebige Programmiersprache gewählt werden, die ausschließlich dazu geeignet ist, eine Lösung zu

erarbeiten, nicht aber ein Problem zu analysieren [10].

Modellgetriebene Architektur

Die *modellgetriebene Architektur* (kurz MDA) basiert auf verschiedenen Grundsätzen und Definitionen, die von der *Object Management Group* (kurz OMG) für verschiedene Abläufe bei der Softwareerstellung vorgeschlagen wurden[10]. Diese sind:

- System: Die Anwendung, die abhängig von der modellgetriebenen Architektur gebaut werden soll.
- Modell: Soll entweder das System bzw. Teile davon oder die Umgebung des Systems repräsentieren.
- Architektur: Die Spezifikation für die Teile des Systems. Legt außerdem die Regeln fest, auf welche Art diese Teile miteinander interagieren können und sollen.
- Plattform: Die Zusammennahme aller Technologien und Subsysteme, welche eine Umgebung für das Erstellen und Ausführen des Systems bereitstellen.
- Standpunkt: Die Beschreibung des Systems mit besonderem Blick auf ausgewählte Details.
- Sicht: Der Teil eines System, der abhängig von einem konkreten Standpunkt als wichtig angesehen wird.
- Transformation: Der Übergang eines Modells in ein anderes Modell.

Bei der modellgetriebenen Architektur werden drei Möglichkeiten für verschiedene Abstraktionslevel der Modelle vorgegeben, wobei sich die Abstraktion hier auf die Möglichkeit bezieht, ausführbaren Code zu erhalten. Zu der abstraktesten Ebene gehören die berechnungsunabhängigen Modelle, die auch Business Model genannt werden. Diese sollen potentiell jedes Detail im Bezug auf Kontext, Voraussetzungen und Zweck des zu modellierenden Sachverhalts darstellen, was dazu führt, dass manche Details in späteren Modelltransformationen herausfallen werden. Damit können sie im Idealfall perfekt darstellen, wie eine entsprechende Lösung des Problems auszusehen hat. Allerdings enthalten sie keinerlei Informationen über eine technische Umsetzung dieser Lösung.
Auf der nächsten Ebene befindet sich das plattformunabhängige Modell. Dieses ist im Bezug zu der fertigen Anwendung weniger abstrakt als das Business Model. Neu hinzugekommen sind jetzt technische Details für den allgemeinen Aufbau und das gewünschte Verhalten der Software.
Auf der letzten und am wenigsten abstrakten Ebene befindet sich das plattformspezifische Modell. Dieses enthält jetzt Anforderungen an das Endprodukt, damit es auf der gewählten

Plattform problemlos genutzt werden kann. Hiermit sind alle Vorraussetzungen erfüllt, ausführbaren Code zu generieren.
Allgemein ist es möglich, aus einem Modell von einer der abstrakteren Ebenen ein Modell aus einer weniger abstrakten Ebene mithilfe von Transformationen abzubilden. Dabei ist eine Abbildung eine Definition der Zusammenhänge zwischen zwei Objekten. Diese wird im Kontext der modellgetriebenen Architektur vor allem für die zwei Metamodelle der Modelle, zwischen denen die Abbildung entstehen soll, vorgenommen, damit daraus automatisiert die Transformation der Modelle durchgeführt werden kann. Gleichzeitig ist es auch möglich, direkt von einem Modell auf ein anderes abzubilden, wodurch es dann allerdings schwieriger bis unmöglich wird, diesen Vorgang zu verallgemeinern und damit automatisch durchführen zu lassen. Speziell bei den Abbildungen zwischen den Ebenen der modellgetriebenen Architektur müssen schon auf den sehr abstrakten Ebenen Details, die für die tieferen Ebenen wichtig sind, beachtet werden.

Interoperabilität

Bei der *Interoperabilität* geht es um die Möglichkeit von mehreren Systemen, untereinander so Informationen auszutauschen, dass diese von den anderen Systemen nutzbar sind. Die Schwierigkeit bei der Interoperabilität besteht darin, dass unterschiedliche Plattformen gleiche Informationen nicht nur unterschiedlich verstehen können, sondern normalerweise auch auf unterschiedliche Weisen darstellen. Gesucht ist also eine Möglichkeit, sowohl die Semantik als auch die Syntax von verschiedenen Systemen passend abzubilden.
Die *modellgetriebene Interoperabilität* basiert auf den Ansätzen der modellgetriebenen Architektur und ist in der Lage, dieses Problem zu lösen. Dies wird erreicht, indem sowohl von dem Modell, von dem die Informationen stammen, als auch von dem Modell, welches die Informationen erhalten soll, jeweils das Metamodell betrachtet wird. Zwischen den beiden Metamodellen wird dann eine Art Brücke gezogen; es wird also geschaut, welche Informationen jeweils von den Modellen benötigt werden und wie diese dargestellt oder gespeichert werden sollen. Durch die Nutzung der Metamodelle wird dieser Prozess stark vereinfacht. Dadurch kann es sehr leicht werden, die entsprechenden Abbildungen vorzunehmen, wodurch es dann sogar möglich ist, dies automatisiert zu erreichen [10].

2.1.3 Erhalten eines ausführbaren Programms aus einer DSL

Modellierungssprachen sind Anwendungen, mit denen Entwickler Modelle erstellen können. *Domain Specific Languages* (kurz DSL) sind Modellierungssprachen, die nur für einen bestimmten Kontext entwickelt und in diesem benutzt werden. Durch diese Art der Spezialisierung können sie effizienter benutzt werden als eine Sprache, die für kein bestimm-

tes Aufgabengebiet erstellt wurde. Das Erstellen einer DSL wird dadurch allerdings auch schwieriger: Es wird nun nicht mehr nur ein Experte im Gebiet der Erstellung von Modellierungssprachen benötigt, sondern auch ein Experte in dem Gebiet, mit dem sich die DSL befassen soll [35].
Für die Erstellung von DSLs gibt es verschiedene Richtlinien. Eine DSL sollte eine ausreichend gute Abstraktion bieten, sodass sich der Nutzer der DSL um weniger Details kümmern muss und ihm das Modellieren insgesamt erleichtert wird. Zusätzlich sollten DSLs, ähnlich wie andere Software, grundsätzlich von mehr als einer Person angefertigt werden. Dadurch wird verhindert, dass die Entwicklung der Sprache durch das Ausfallen einer einzelnen Person komplett gestoppt wird. Außerdem werden generell bessere Designentscheidungen getroffen, wenn sie vorher in einer Gruppe besprochen wurden [10].
DSLs sollten grundsätzlich nie als etwas Vollendetes oder etwas, das vollendet werden kann, betrachtet werden. Stattdessen sollten sie immer mit dem Gedanken, dass sie veränderbar bleiben müssen, erstellt werden. Die wichtigsten Gründe für eine nachträgliche Änderung der DSL sind meist eine Veränderung des Kontextes, für den die Sprache erstellt wurde, oder neue Bedürfnisse der Nutzer der Sprache [35].

Für die Erzeugung von ausführbaren Programmen aus einer DSL wird oft *Codegenerierung* genutzt.
Bei der Codegenerierung geht es darum, aus einem Modell nutzbaren Quellcode zu erstellen. Dieser Vorgang ist vergleichbar mit der Erstellung von Binärdateien aus Quellcode, weshalb die Art Programme, die dafür zuständig ist, auch Modellcompiler genannt werden. Da das endgültige Ziel eigentlich ist, aus einem Modell ein ausführbares Programm zu erhalten, sind diese beiden Vorgänge eng miteinander verknüpft, denn sie werden bei einer vollständig funktionierenden automatischen Codegenerierung direkt nacheinander durchgeführt [10].
Während es grundsätzlich möglich ist, eine Mischung aus generiertem und handgeschriebenen Code zu benutzen, ist es meist empfehlenswert, sich nur auf einen der beiden Ansätze festzulegen oder diese zumindest eindeutig zu trennen.
Zu den Vorteilen der Codegenerierung zählt die Möglichkeit, gewisse Vorgaben an den generierten Code zu stellen, die zwingend eingehalten werden müssen, was durch eine korrekte Generierung garantiert werden kann. Ein Beispiel für einen Grund für solche Vorgaben wäre leistungsschwache Hardware des Kunden. Ein weiterer Vorteil ist, dass bereits generierter Code sehr leicht wiederverwendet werden kann, indem er beispielsweise mit leichten Änderungen neu generiert wird, wobei dieser Prozess beliebig oft wiederholt werden kann. Dies hat den zusätzlichen Vorteil, dass die Wiederverwendung eine korrekte Funktion des Codes teilweise garantieren kann.
Generierter Code kann den Nachteil haben, dass er Entwicklern nicht bekannt vorkommt, was die Fehlersuche oder das Verstehen des Codes erschwert. Beim Arbeiten mit generiertem Code über einen längeren Zeitraum sollte dieses Problem allerdings an Bedeutung

verlieren. Dieses Problem wird auch abgeschwächt, wenn das Ergebnis der Codegenerierung sehr nah an vorher händisch erstelltem Code oder Modellen angelehnt ist.

2.1.4 Unified Modelling Language

Die *Unified Modelling Language* (UML) ist eine Modellierungssprache, um Spezifikationen und Zusammenhänge in Softwaresystemen darzustellen. Sie wird wird auch von der OMG entwickelt. Es handelt sich dabei um keine echte DSL. Trotzdem ist UML aber relativ ähnlich zu grafischen DSLs, sodass sie als guter Einstiegspunkt genutzt werden kann, um DSLs an Beispielen erklären zu können. Klassendiagramme sind vermutlich die bekannteste Form von UML.

Klassendiagramme

Klassendiagramme werden dazu genutzt, um Objekte, Klassen und deren Zusammenhänge in einer objektorientierten Programmierumgebung darzustellen. Die Aufgabe von Klassendiagrammen ist unter anderem, die Struktur der Implementierung in einer objektorientierten Programmiersprache zu definieren.
Klassendiagramme bestehen aus drei verschiedenen Teilen:[18]

- Klassen: Objekte und daraus abgeleitete Klassen und Interfaces entsprechen einer Objektdeklaration in der jeweiligen Programmiersprache. Sie werden als Rechteck dargestellt. Die Klassen können Attribute mit entsprechenden Modifiern wie „Public“, „Private“ oder „Protected“ besitzen.
- Beziehungen beschreiben alle verschiedenen Beziehungen, die es zwischen Objekten geben kann. Dazu gehören Vererbung und Implementierung, aber auch Relationen, welche darstellen, wie viele Klassen von einem Typ in einem anderen enthalten sind. Die Beziehungen zwischen den Klassen werden mit verschiedenen Pfeilen dargestellt.
- Operationen: Mit ihnen werden Funktionen der jeweiligen Klassen modelliert. Sie stehen, falls vorhanden, unter den Attributen einer Klasse.

Abbildung 2.2 zeigt das Modell eines Backofens. Der Backofen wird hier als Objekt einer objektorientierten Programmiersprache gezeigt. In diesem Klassendiagramm steht die Beziehung zwischen den Entitäten im Vordergrund. Somit ist darstellbar, wie viel Essen gleichzeitig in den Backofen passt, dass der Backofen ein Elektrogerät ist (also von der entsprechenden abstrakten Klasse erbt) und welche Methoden und Attribute diesen Backofen ausmachen.

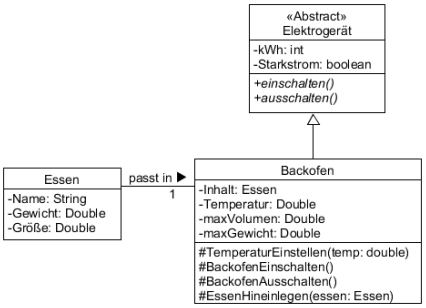


Abbildung 2.2: Klassendiagramm Backofen

Einige Java IDEs, wie zum Beispiel BlueJ, ermöglichen es, die Grundgerüste der Klassen aus einem Klassendiagramm zu generieren. Der generierte Java-Code kann dann zwar noch nicht sinnvoll ausgeführt werden, da die Methoden nur aus ihrer Signatur bestehen und noch nicht implementiert sind, aber auch die Generierung eines Codegerüsts kann für Entwickler schon ein große Hilfe sein, womit UML den Ansprüchen der Codegenerierung genügt. Hätten Klassendiagramme die Möglichkeit, die Funktionsweise von Methoden ebenfalls zu bestimmen und damit zu generieren, wäre es eine grafische Modellierungssprache für die entsprechende zugrundeliegende objektorientierte Sprache. Damit wäre es allerdings immer noch keine DSL, da es nicht domänenspezifisch wäre.

2.2 Domain-Specific Languages

Domain-Specific Languages sind wie in 2.1.3 beschrieben Design- und Entwicklungssprachen, die darauf ausgelegt sind, ganz spezifische Anforderungen einer Anwendungsdomäne umzusetzen.
DSLs können aus eigenen Markup-, Entwicklungs- und Modellierungssprachen bestehen. Somit gibt es DSLs, die darauf ausgelegt sind, als möglichst präzise Programmiersprachen zu agieren und somit Textbasiert sind, sowie andere, die darauf ausgelegt sind, Modelle zu erstellen, aus denen Code generiert wird. Diese werden grafische Sprachen genannt. [**?**]
DSLs sind besonders dafür geeignet, dass Domänenexperten eigene spezialisierte Anwendungen schreiben können, ohne sich in GPLs einarbeiten zu müssen. Es wird also eine Sprache oder ein Modellierungswerkzeug bereitgestellt, dass für den Endanwender in seiner spezifischen Domäne leicht zu verstehen und zu benutzen ist, damit er sich eigene neue Domänenspezifische Anwendungen erstellen kann. [**?**]

Textuelle und grafische DSL

Wie bereits angemerkt, gibt es einen Unterschied zwischen textuellen und grafischen DSL. Beide Sprachen haben ihre Vorteile und Nachteile. Werden sich die Richtlinien für DSL angeschaut, dann gibt es zwei Richtlinien, für die sich grafische besonders DSL gut eignen. Grafische Modelle bieten sich besonders dafür an, dem Entwickler gute Abstraktionen seiner vorliegenden Arbeit zu geben. Zusammenhänge zwischen Objekten lassen sich in Grafiken häufig auf einen Blick erkennen, ohne sich tiefgehender mit allen Einzelheiten beschäftigt zu haben. Gleichzeitig lassen sich Werkzeuge zum Erstellen der Modelle entwickeln, die möglichst leicht und intuitiv verständlich sind. So ist es für eine Person ohne Programmierkenntnisse, die sich aber in der Domäne dieser DSL auskennt, leichter, Objekte über ein Interface zu erstellen.

Zusätzlich können dem Nutzer Funktionen zu den Objekten vorgeschlagen werden, von denen er, aufgrund seiner Kenntnisse in der entsprechenden Domäne, annehmen kann, dass sie existieren. Dafür muss die genaue Syntax der Objekte nicht bekannt sein.[14]

Ausblick

Im weiteren Verlauf wird es eine kurze Übersicht über textuelle DSLs gegeben. Anschließend soll die Idee und die Umsetzung von grafischen DSLs anhand von UML betrachtet werden. UML befindet sich auf der Grenze von Modellierungssprache und DSL, bietet aber viele Ansatzpunkte, an denen sich DSLs erklären lassen. Danach werden einige Workbenches zur Erstellung und Nutzung von DSLs erläutert. Anhand dieser Beispiele wird auf die verschiedenen Umsetzungen und Zielgruppen eingegangen, die die verschiedenen Modellierungswerkzeuge anstreben. Als letztes wird die Frage erörtert, ob und wann eine grafische DSL einen Vorteil gegenüber einer rein textlichen DSL bietet.

Zusätzlich ist noch relevant, dass DSLs nicht nur darauf beschränkt sind, konkrete Dinge zu generieren. Mit einer entsprechenden DSL lassen sich auf Metaebene auch andere DSL spezifizieren und erstellen. Somit gibt es auch DSLs zur DSL-Erstellung.

2.2.1 Textuelle Modellierungssprachen

Textuelle Modellierungssprachen sind, wie der Name bereits sagt, DSLs, die Modellierungen über Text ermöglichen. Diese sind in ihrer Handhabung vergleichbar mit GPLs. Es wird Code mit einer bestimmten Sprache geschrieben und dieser am Ende in ein entsprechendes Programm, Objekt oder Ähnliches umgewandelt oder kompiliert. Sie ähneln in ihrer Funktionsweise stark den GPLs.

HTML

Hyper-Text Markup Language (kurz HTML) ist eine DSL zur Generierung von Webseiten. HTML funktioniert, indem innerhalb eines Textes der dargestellt werden soll, sogenannte Tags eingebunden werden. Diese können ganz einfache Textformatierungen sein, wie z. B. Text für fett gedruckten Text oder Text für kursiven Text.
HTML definiert aber auch die Struktur des darzustellenden Textes. Auch dafür werden Tags verwendet. <html> um festzulegen von wo bis wo der Textteil als HTML-Dokument interpretiert werden soll, <head> für den Titel der Seite, <body> für den Inhalt, <table> um Tabellen darzustellen, und viele mehr. Tags können zudem Attribute besitzen, die weitere Einstellungen vornehmen können.[30]
Wie in 2.3 zu sehen ist, enthält HTML sehr schnell sehr viel zusätzlichen Text der eigenen Sprache, zusätzlich zu dem darzustellenden Text. HTML ist im Vergleich zu einer GPL zwar leichter zu erlernen, allerdings gibt es auch in HTML sehr viele verschiedene Tags und

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta charset="utf-8">
    <title>Beispiel HTML</title>
  </head>
  <body style="font-family: sans-serif;">
    <h1>Eine Beispielhafte HTML Webseite</h1>
    <p>
      Text kann in Absätzen zusammengefasst werden,
      unabhängig der Zeilenumbrüche
    </p>
    <p>
      <span style="color: #ff0000;">Die Textfarbe lässt sich anpassen
        und Tabellen sind ebenfalls einfach erstellt.</span>
    </p>
    <table>
      mit Tabellenüberschrift
      <tr>
        <td><em>Zeile 1 Spalte 1</em></td>
        <td>Zeile 1 Spalte 2</td>
      </tr>
      <tr>
        <td>Zeile 2 Spalte 1</td>
        <td>Zeile 2 Spalte 2</td>
      </tr>
    </table>
  </body>
</html>
```

(a) Text

(b) Webseite

Abbildung 2.3: HTML Als Text und als Webseite

Eigenschaften, die bekannt sein müssen, um funktionale Webseiten erstellen zu können. Hier kristallisiert sich einer der größten Nachteile der textlichen DSL heraus. Obwohl die DSL im Falle von HTML nur dazu da ist, Webseiten zu gestalten, besteht sie trotzdem aus einer großen Menge an HTML-eigenen Elementen. Somit muss ein Webseitendesigner die Webseite nicht jedes mal neu erfinden, er muss aber trotzdem eine ganze Sprache erlernen um diese zu erstellen.

2.2.2 Grafische Modellierungssprachen

Grafische Domain Specific Languages verfolgen die Idee, dass zur Nutzung der Sprache grafische Objekte miteinander verbunden werden. Es wird also wenig bis gar kein Text geschrieben. Die resultierenden Modelle werden nach der Erstellung in das gewünschte Format übersetzt. Das Hauptaugenmerk liegt dementsprechend darauf, möglichst intuitiv grafische Modelle zu erstellen, die eine Aufgabe so präzise wie möglich abbilden. Grafische Modellierungssprachen wurden in 2.1 Modellgetriebene Softwareentwicklung angesprochen. Aus diesen Modellierungssprachen lassen sich auch Domain Specific Languages erstellen. Als einführendes Beispiel wird ein Teil von UML gedanklich in eine DSL verwandelt.

Von UML zur DSL

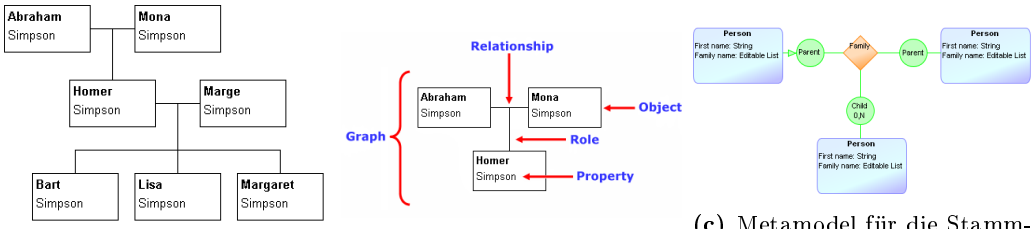
UML bietet bereits einige Grundsätze, an denen sich grafische DSLs orientieren können. Bei Klassendiagrammen und deren angesprochenen Einsatz in BlueJ, ist zu beobachten, dass zumindest Klassendiagramme eine DSL für die Erstellung eines Grundgerüstes von Softwaresystemen sind. Gleichzeitig ist das Gegenargument, dass sich mit UML (und auch mit Klassendiagrammen) im Grunde alles modellieren lässt. Damit ist UML nicht domänenspezifisch. Außerdem lässt sich aus den wenigsten UML-Diagrammen etwas generieren. Bei der sogenannte Model Driven Architecture (MDA), geht es darum, aus Modellen mög-

lichst automatisch Quellcode generieren zu lassen. MDA sieht die Trennung von Technik und Funktionalität vor. Dies hat den gleichen Zweck wie die Domänenspezifizierung bei DSL. Es geht darum, dass die Funktionalitäten und die Konversion von Modell in andere Modelle oder Quellcode definiert werden und somit als Grundgerüst dienen. Das Modell ist dann die eigentliche Entwicklungsarbeit an der Problemstellung. Die Idee der MDA ist, dass möglichst alles aus Modellen generiert wird und Modelle so häufig wie möglich wiederverwertet werden können. Kann kein Quellcode aus Modellen generiert werden, so sollen die Modelle so selbsterklärend und eindeutig sein, dass die Umsetzung kein Problem mehr darstellen soll.
Würde eine gesamte Software nach MDA erstellt werden, und dabei der gesamte Quellcode aus den Modellen generiert wird, so würde de facto eine DSL für eine Problemstellung mit UML als grafische Grundlage und der resultierenden Programmiersprache als Wirtssprache entstehen.
Dies bedeutet im Umkehrschluss, dass sich zum Beispiel eine DSL mit Java als Wirtssprache für die Erstellung von Tourenplanungen schreiben ließe, die einen UML Diagrammtyp als grafische Grundlage nehmen könnte.
Im Folgenden werden einige Language Workbenches betrachtet, mit denen grafische DSL erstellt werden können. Bei den Spezifikationen dieser Language Workbenches handelt es sich häufig ebenfalls um DSLs, was diese Language Workbenches effektiv zu Meta-DSLs macht.

MetaEdit+

MetaEdit+ ist eine Entwicklungsumgebung für domänenspezifische Modellierungssprachen von MetaCase[5]. Die zugrundeliegende Forschung am MetaPHOR Projekt wurde zwischen den 1980 und 1990er Jahren an der Universität Jyväskylä betrieben. MetaEdit+ besteht aus zwei Anwendungen. MetaEdit+ Workbench ist eine Arbeitsumgebung um grafische DSL zu erstellen. Diese können entweder von Grund auf neu entworfen werden, oder von bereits vorhandenen DSL abgeleitet werden. Mit MetaEdit+ Modeler lassen sich dann anhand der erstellten DSL Produkte erzeugen.
Wie dem Namen MetaEdit zu entnehmen ist, handelt es sich bei der Spezifikation von MetaEdit ebenfalls um eine DSL. Mit MetaEdit ist es also ebenfalls wiederum möglich, eine DSL zu erstellen, die zum Erstellen von DSLs benutzt werden soll.
MetaEdit benutzt zum Erzeugen von neuen Metamodellen die GOPPRR- (Graph, Object, Port, Property, Relationship, Role) Modellierungssprache. Der Name enthält bereits alles, was MetaEdit benötigt, um neue Metamodelle zu erstellen.

- Graph: ein individuelles Modell
- Object: die Hauptelemente eines Graphs



(a) Abbildung eines Stammbaumes

(b) GOPPRR Metatypen

(c) Metamodel für die Stammbaum DSL

Abbildung 2.4: Von der Idee zur DSL mit MetaEdit+

- Property: Eigenschaften, die ein Object haben kann
- Relationship: Verbindung von Objects und wie diese miteinander interagieren
- Role: Art des Objects in einer Relationship
- Port: Definition von zusätzlichen Eigenschaften, wie die Role, mit dem Object zusammenhängt

In einem einfachen Beispiel, um mit MetaEdit ein Stammbaum anzulegen, müssen aus einem Stammbaum zunächst die Typen extrahiert werden. Im Falle des Stammbaumes ist die Umwandlung in GOPPRR denkbar einfach. Alle Mitglieder des Stammbaumes werden als das gleiche Object abgebildet. Dieses Object beschreibt einfach nur eine Person mit ihrem Vor- und Familiennamen als Property. Ein Stammbaum verfügt wegen seiner Struktur über klar definierte Relationen zwischen allen Personen. Die einfachste Art, diese Relationen darzustellen, ist zu definieren, dass zwei Personen zusammen eine Relation haben, aus der eine beliebige Anzahl an Kindern entspringen können. Jeder Person in dieser Relationship muss dann eine Rolle zugewiesen werden. Dabei genügt es, Mutter, Vater und Kind als Rollen zu haben. Geschwister lassen sich anhand der Relationship gleicher Eltern erkennen, und die Familiengrade und Generationen über die Kinders-Kinder oder Eltern-Eltern.
Mit dem auf diesen Überlegungen basierenden Metamodell aus Abbildung 2.4c lassen sich im MetaEdit+ Modeler einfach Stammbäume erzeugen, die immer den Spezifikationen der entworfenen Sprache entsprechen. Aufgrund des Aufbaus von GOPPRR ist es mit MetaEdit+ dementsprechend leicht, DSL für Graphen zu spezifizieren. Es ist aber natürlich ebenfalls möglich, DSLs zu entwerfen, die nicht auf Graphen basieren und wesentlich komplexere Relationship zwischen den Objects haben.[29]

Marama

Marama ist eine DSL Workbench, die als Eclipse Plugin entwickelt wurde. Die Idee von Marama ist, dass die zur Verfügung gestellten Werkzeuge so einfach zu benutzen sind,

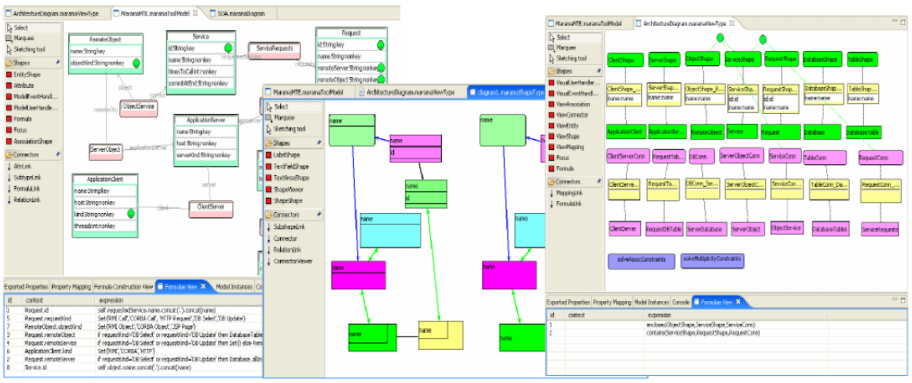


Abbildung 2.5: Marama Tools: Meta-Model (links), Shape Designer (mitte), View Definer (rechts)

dass erfahrene Modellierer nach nur einem Tag Arbeit in der Lage sind, einfache grafische Modelle zu gestalten. Die Arbeitszeit beinhaltet aber noch keine Codegeneration oder Ähnliches.
Marama legt dabei besonders Wert darauf, Werkzeuge anzubieten, die eigenen Werkzeuge zu spezifizieren, Modelltransformationen und Codegeneration zu erleichtern und eigene neue Views zu erstellen. Bei der Erstellung und Integration neuer Werkzeuge versucht Marama, diese Änderungen auch sofort in den jeweiligen Modellen vorzunehmen und somit möglichst viel Arbeit zu automatisieren.
Marama ist unterteilt in drei Arbeitsbereiche. Im Meta-Model Tool können auf Basis des Enhanced Entity-Relationship Model (EER) mithilfe einer Object Constrain Language (OCL) neue Metamodelle erzeugt werden. Mit dem Visual Shape Designer werden die Spezifikationen der Relationen für das Modell erzeugt. Im View Designer wird als letztes festgelegt, welche visuellen Elemente in welchen Viewtype gehören, und wie diese visuellen Elemente mit dem Metamodell zusammenhängen. Marama bringt zusätzlich ein Werkzeug zur Modelltransformation mit. Mit diesem Werkzeug lassen sich Modellschemata in andere umwandeln, ohne zusätzlichen Zeitaufwand zu benötigen[16].

2.2.3 Gegenüberstellung grafische und textbasierte DSL

Im vorherigen Abschnitt wurden Workbenches zur Erstellung und Benutzung von DSL vorgestellt. Um die Vorteile von grafischen Modellierungswerkzeugen zu erläutern, soll aber noch ein Schritt zurück gemacht werden. In 2.2.1 wurde HTML kurz eingeführt, und in 2.3a ist eine kleine beispielhafte HTML-Webseite zu sehen. HTML lässt sich komplett in einem Texteditor und frei von Werkzeugen oder gar grafischen Werkzeugen schreiben. Allerdings wird es auch bei einer kleinen Webseite ein wenig aufwändig, die gesamte Struktur schnell zu überblicken.

Eine einfache Abhilfe, die in jeder IDE eingesetzt wird, ist das sogenannte *Syntax-Highlighting*, also das Hervorheben der Syntax. Syntax-Highlighting erleichtert das Lesen des HTML-Dokumentes, sorgt aber beim Schreiben noch nicht dafür, dass die Namen der Tags oder Attribute automatisch angezeigt werden. Automatisches Vorschlagen und Ausfüllen von Befehlen ist eine weitere Maßnahme, die viele IDEs ergreifen, um das Schreiben von Anwendungen in einer Sprache zu erleichtern. Diese Maßnahmen helfen erfahrenen Entwicklern erheblich dabei, schnell neue Anwendungen zu schreiben, da sie sich nicht an jeden Befehl in seinem genauem Wortlaut erinnern müssen. Existieren jedoch keine Kenntnisse einer Sprache oder wird eine neue Bibliothek oder Erweiterung zum ersten Mal benutzt, ist es häufig nicht vermeidbar, die Dokumentationen eben jener zu lesen.
Um zu dem Beispiel von HTML zurückzukehren, müsste sich also jemand, der nur eine einfache Webseite in HTML erstellen möchte und keine Vorkenntnisse hat, die Dokumentation von HTML oder zumindest eine Einführung durchlesen. Damit gerade das nicht der Fall ist, gibt es HTML Workbenches, die es ermöglichen, HTML grafisch zu erlernen. Eine dieser Möglichkeiten ist die Webseite HTMLBausteine. [4]
Dort lassen sich, wie in 2.6 zu sehen ist, HTML-Tags und Attribute als Blöcke miteinander verbinden und mit dem entsprechenden Inhalt füllen. HTML ist als Beispiel nicht ideal, da HTML noch vergleichsweise leicht zu erlernen ist. Werden aber weitere DSLs wie SQL betrachtet, dann wird unweigerlich deutlich, dass ein sogenannter Datenbankarchitekt nicht daran vorbei kommt, SQL und die zugrundeliegende relationale Algebra zu erlernen. Somit gibt es bei einigen heutigen großen DSLs das Problem, dass diese so groß und mächtig geworden sind, dass die Nutzer dieser Sprachen in erster Linie Experten der Sprache sind und nicht Domänenexperten, die eine auf sie zugeschnittene DSL benutzen. Sollte es jedoch das Ziel sein, dass ein Domänenexperte möglichst einfach und selbstständig neue Probleme bewältigen und Lösungen finden kann, dann ist es unerlässlich, dass die resultierenden DSLs grafisch einsetzbar sind. Anhand des HTML-Beispiels und der Workbench-Beispiele ist sehr deutlich zu erkennen, wie groß der Vorteil ist, Modelle in einer visuellen Repräsentation erstellen zu können. Grafische DSL haben also den klaren Vorteil, dass die Einstiegshürde wesentlich geringer ist als bei rein textuellen DSL. Außerdem ist das Arbeiten mit grafischen Werkzeugen einfacher und intuitiver als mit reinen Textsprachen.
Andererseits muss natürlich auch in Betracht gezogen werden, ob die Nachteile von grafischen DSL die Vorteile nicht überschatten. Als erstes muss bedacht werden, dass die Entwicklung grafischer DSLs einen erhöhten Aufwand verlangt. Es muss nicht nur eine Sprache entwickelt werden, sondern auch die entsprechenden grafischen Werkzeuge und Oberflächen erstellt werden. Grafische DSLs verlangen vom Entwickler der Sprache außerdem erweitertes abstraktes Verständnis ab, damit die Konzepte der neuen Sprache auch in grafischer Darstellung Sinn ergeben. Diese beiden Hürden zur Erstellung von grafischen DSL können verringert werden, wenn entsprechende Workbenches zu Erstellung von DSLs eingesetzt werden. In Kapitel 2.2.2 wurden drei Workbenches vorgestellt, die alle in verschiedenen

DSL-Arten spezialisiert sind. Das Entwerfen von DSLs mit Hilfe von Workbenches, beziehungsweise DSLs zur Erstellung von DSL, erleichtert die Arbeit auf die gleiche Weise, wie es die erstellten DSLs für den Endanwender erleichtern sollen.

2.2.4 Auswahl von textlichen oder grafischen DSLs

Domänenspezifische Sprachen sind eine sinnvolle Art, um Entwicklungsarbeit aufzuteilen. Werden grafische DSLs gesondert betrachtet, dann wurde im Verlauf festgestellt, dass sie einige nennenswerte Vorteile gegenüber klassischen, textlichen DSL haben. Werden grafische DSLs als Anwendung der MDA und dem damit zusammenhängenden MDSE betrachtet, dann bieten diese eine Erweiterung der MDA in die vollautomatische Codegenerierung. Zusätzlich sind mit guten grafischen DSLs die Endanwender (in Unternehmen also Mitarbeiter, die nicht Teil einer Softwareabteilung sind) in der Lage, neue Probleme zu bewältigen, ohne dass ihnen ein vollständig neues Programm geschrieben werden muss.

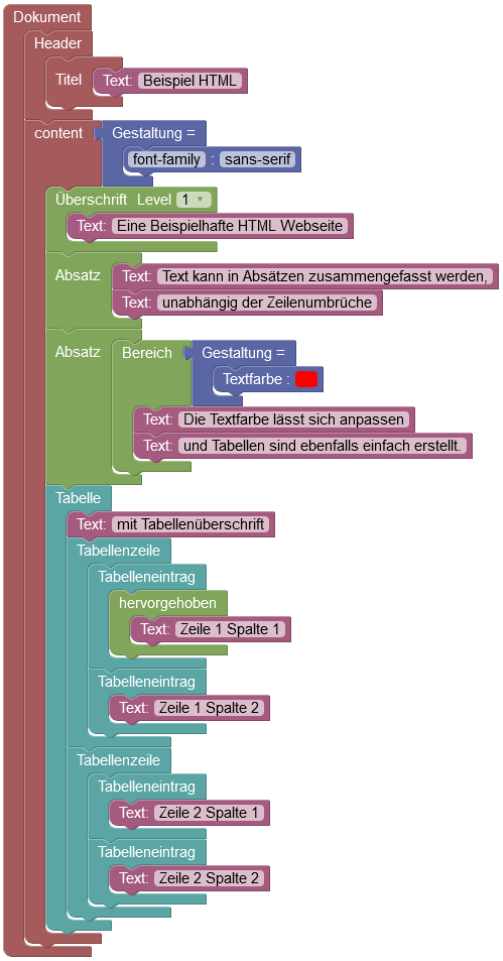


Abbildung 2.6: Das HTML-Dokument aus Abbildung 2.2.1 als grafische Darstellung

Die Entwicklung von grafischen DSL hat dementsprechend einen hohen Aufwand, bevor sie eingesetzt werden kann. Sobald diese aber ausgereift ist, lassen sich viele Aufgaben dieser Domäne wesentlich schneller und leichter bewältigen, ohne dass ein dedizierter Entwickler diese Aufgaben übernehmen muss.

Abschließend lässt sich also Folgendes sagen: Wenn eine domänenspezifische Sprache erstellt werden soll, die von möglichst programmierunerfahrenen Personen eingesetzt wird, und die Möglichkeit besteht, diese als grafische Sprache zu erstellen, dann lohnt sich der zusätzliche Aufwand, dies zu tun.

2.3 CINCO

Die „Cinco SCCE Meta Tooling Suite“ ist eine Entwicklungsumgebung, welche schnelles und einfaches Entwickeln von domänen-spezifischen grafischen Modellierungswerkzeugen ermöglicht. Cinco beinhaltet verschiedene Frameworks des Eclipse Modeling Product. Auf diese Frameworks greift Cinco weitestgehend im Hintergrund zu, sodass der Entwickler davon relativ wenig mitbekommt. Cinco unterstützt drei Sprachen (CPD, MGL und MSL) mit denen es möglich ist die strukturellen und grafischen Features des entwickelten Tools zu beschreiben. Mit der Hilfe dieser drei Sprachen kann Cinco das Modellierungtool zum größten Teil selber generieren. Dies geschieht durch Codegenerierung, was ein entscheidendes Merkmal von Cinco ist. Dieses generierte Tool wird auch „Cinco Product“ genannt. [22]
CPD, MGL und MSL werden anschließend in den Abschnitten 2.3.1 bis 2.3.3 thematisiert.Im folgenden werden Features von Cinco vorgestellt, welche die Modellierungsmöglichkeiten noch umfangreicher machen. 2.3.4 gibt einen Überblick über die Meta Plug-ins, mit denen die Modellierungselemente ausgestattet werden können. Einige dieser Meta Plug-ins benötigen Actions, die das darauf folgende 2.3.5 vorstellt. Das Cinco Product ermöglicht es entwickelte Tools in anderen Tools zu verwenden. Dies geschieht mit Hilfe der Prime-Reference, welche in Kapitel 2.3.6 beschrieben wird.

2.3.1 Cinco Product Definition

Die *Cinco Product Definition* wird in der .cpd Datei definiert. Sie wird genutzt, um die Modellierungsumgebung zu erstellen. Es muss ein Name für das Projekt und ein Verweis auf mindestens eine *Meta Graph Language* angegeben werden. Außerdem kann die erzeugte Umgebung mit optionalen Parametern individualisiert werden [1].

2.3.2 Meta Graph Language

Die Meta Graph Language wird in die .mgl Datei des Cinco Projekts geschrieben. Sie wird dazu genutzt, um die Elemente des Modells zu definieren.[1]

Graphmodell

Das Graphmodell ist das Wurzelelement jeder .mgl Datei. Es benötigt die folgenden Parameter:

- *package* ist das Präfix der generierten Klassen.
- *nsURI* ist der für die MGL genutzte Namensraum.
- *diagrammExtension* gibt die Endung erstellter Modelle an.

Es kann Attribute und Modellelemente enthalten.

Attribute

Das Graphmodell und dessen Elemente können Attribute besitzen. Diese können beispielsweise vom Typ

- *EString*
- *EChar*
- *EBoolean*
- *EInt*
- *EFloat*
- *EDouble*

sein. Sie werden mit der Syntax *attr type as name[min,max]* deklariert, wobei min/ max angeben, wie viele Werte das Attribut minimal bzw. maximal enthalten darf. Ohne angegebene Kardinalität enthält das Attribut genau einen Wert.
Zusätzlich können Enumerationen und selbst definierte Typen erstellt werden.

Knoten

Für Knoten kann spezifiziert werden, welche Kantentypen in einem Knoten ein- bzw ausgehen dürfen und in welcher Häufigkeit dies erlaubt ist. Dies geschieht mit den Attributen *incomingEdges* und *outgoingEdges*. Möglichkeiten dafür sind:

- *incomingEdges(∗)*: In den Knoten darf eine beliebige Anzahl beliebiger Kanten eingehen.
- *incomingEdges(∗[0,3])*: In den Knoten dürfen beliebige Kanten eingehen, aber maximal 3.
- *incomingEdges(Kantentyp1[0,5],Kantentyp2[0,∗])*: In den Knoten dürfen maximal 5 Kanten von Kantentyp1 und eine beliebige Anzahl von Kantentyp2 eingehen.
- *incomingEdges({Kantentyp1,Kantentyp2}[0,5])*: In den Knoten dürfen insgesamt maximal 5 Kanten von Kantentyp1 und Kantentyp2 eingehen.

Analog können *outgoingEdges* definiert werden.
Knoten können außerdem durch *extent* von anderen Knoten erben und dadurch die Attribute der Superklasse übernehmen. Analog können auch die Attribute der im folgenden vorgestellten Kanten und Container vererbt werden.

Kanten

Kanten werden genutzt, um die Beziehungen zwischen Knoten auszudrücken. Diese Beziehungen wurde bereits durch *incomingEdges/outgoingEdges* in den Knoten definiert. Daher müssen bei Kanten keine weiteren Angaben dazu gemacht werden.

Container

Container können genau wie Knoten eingesetzt werden. Zusätzlich kann ein Container Knoten oder andere Container enthalten. Dies kann mit dem Attribut *containableElements* definiert werden. Wie bereits bei ein- bzw ausgehenden Kanten gibt es folgende Möglichkeiten:

- *containableElements(∗)*: Der Container darf eine beliebige Anzahl beliebiger Knoten enthalten.
- *containableElements(∗[0,3])*: Der Container darf beliebige Knoten enthalten, aber maximal 3.
- *containableElements(Knotentyp1[0,5],Knotentyp2[0,∗])*: Der Container darf maximal 5 Knoten von Knotentyp1 und eine beliebige Anzahl von Knotentyp2 enthalten.
- *containableElements({Knotentyp1,Knotentyp2}[0,5])*: Der Container darf insgesamt maximal 5 Knoten von Knotentyp1 und Knotentyp2 enthalten.

2.3.3 Meta Style Language

Die *Meta Style Language* beschreibt die grafische Darstellung der Modellelemente und wird in die .style Datei geschrieben. Jedes Modellelement besitzt eine *@Style* Annotation, die auf eine Definition der MSL verweist. Im folgenden werden diese vier Möglichkeiten aus [1] vorgestellt, um das Aussehen von Modellelementen zu verändern:

NodeStyle

Der *nodeStyle* wird genutzt, um das Aussehen von Knoten und Containern zu definieren. Jeder nodeStyle muss mindestens eine Containerform enthalten. Erlaubte Formen dafür sind:

- *rectangle*
- *roundedRectangle*
- *ellipse*
- *polygon*

Für alle vier können ein Name, sowie die Attribute *size* und *appearance* angegeben werden. Gerundete Rechtecke benötigen zusätzlich das Attribut *corner*, mit dem die Form der Ecken verändert werden kann. Polygone werden über ihre Eckpunkte definiert, die im Attribut *points* angegeben werden.
Die Formen lassen sich hierarchisch erweitern. Jede Containerform kann weitere Formen enthalten. Bei diesen kann durch das Attribut *position* zusätzlich die Position innerhalb der Elternform angeben werden.
Einige Elemente können ausschließlich in andere Formen eingefügt werden und können selbst keine Formen enthalten. Diese sind:

- *text*/*multitext*: Es wird ein Textfeld mit dem String im Parameter *value* erzeugt.
- *image*: Es wird ein Bild aus *path* mit der Größe *size* eingefügt.
- *polyline*: Es wird ein Kantenzug aus den Punkten im Parameter *points* erstellt.

Das Textelement kann außerdem dazu genutzt werden, um Attribute der Modellelemente anzuzeigen. Dafür muss das Modellelement seiner Styledefinition das anzuzeigende Attribut als Parameter übergeben. Auf diese Weise wird die Ausgabe mit dem Attribut gebunden, so dass auch bei manueller Änderung immer der aktuelle Wert ausgegeben wird.

EdgeStyle

Der *edgeStyle* wird genutzt, um das Aussehen von Kanten zu definieren. Anders als bei Knoten wird die Form nicht hierarchisch aufgebaut. Stattdessen werden Dekoratoren genutzt, um Formen an Kanten zu platzieren. Dekoratoren benötigen eine Form und einen Punkt auf der Kante. Das Attribut *location* gibt diesen Punkt relativ zur Kante an. Es benötigt einen Wert zwischen 0 und 1, wobei 0 der Beginn und 1 das Ende der Kante ist. Als Form können die im vorherigen Abschnitt vorgestellten Elemente genutzt werden. Zusätzlich sind diese vordefinierten Formen nutzbar:

- *ARROW*
- *DIAMOND*
- *CIRCLE*
- *TRIANGLE*

Außerdem kann ein Dekorator durch *movable* bewegbar gemacht werden.

Appearance

Appearances können genutzt werden, um das Aussehen von geometrischen Formen zusätzlich zu gestalten. Sie können entweder separat definiert und anschließend über ihren Namen

referenziert werden oder direkt in einer Styledefinition definiert werden. Im folgenden sind die möglichen Attribute und ihre Bedeutung aufgelistet.

- *angle* setzt den Winkel der Form.
- *background* setzt die Hintergrundfarbe nach dem RGB-Farbraum.
- *foreground* setzt die Linienfarbe nach dem RGB-Farbraum.
- *filled* aktiviert bzw. deaktiviert die Hintergrundfarbe.
- *font* setzt die Linienform durch (Schriftart,Schriftgröße).
- *imagePath* enthält den Pfad zu einem Bild.
- *lineStyle* setzt die Art der Linie durch: *DASH*, *DASHDOT*, *DASHDOTDOT*, *DOT*, *SOLID*.
- *lineWidth* setzt die Breite der Linien.
- *transparency* setzt die Transparenz.

Appearanceprovider

Durch Appearanceprovider kann das Aussehen von Modellelementen auch während der Laufzeit verändert werden. Ein Appearanceprovider ist eine Javaklasse, die das *StyleAppearanceProvider* Interface implementiert. Dieses fordert die Implementierung der *getAppearance* Methode, die das zu verändernde Modellelement als Parameter erwartet. Cinco erzeugt aus den in der MGL erstellten Modellelementen Javaklassen. Auf die Attribute kann also mit get- und set-Methoden zugegriffen werden.
Die Methode erstellt das neue Aussehen und gibt es zurück.

2.3.4 Meta Plug-ins

Im folgenden Abschnitt werden Meta Plug-ins vorgestellt, welche Cinco erweitern und für jedes mit Cinco entwickelte Modellierungswerkzeug verwendet werden können. Sie ermöglichen neue Features für das Modellierungswerkzeug und können dessen Verhalten beeinflußen. Durch die Verwendung von Meta Plug-ins wird unter anderem Model-Checking, Layout-Anpaßung, Bearbeitung der Ansicht und Code-Generierung ermöglicht. Die Meta Plug-ins helfen dem zu Folge, das Modell zu verändern. [31]

Core Meta Plug-ins

@color(parameter)

Kann einem EString-Attribut hinzugefügt werden und ermöglicht einen Color-Picker in der Cinco Properties View. Gültige Parameter sind: rgb, rgba und hex.

@contextMenuAction(parameter)

Kann einer Node, einer Edge, einem Container oder einem Graphmodell hinzugefügt werden. Als Parameter wird eine Klasse verlangt, die im Kontext-Menü des Objektes erscheint. Diese Klasse muß bestimmte Methoden, wie z.B. getName() enthalten.

@disable(parameter)

Schränkt die Funktionalität von Modelelementen ein. Mögliche Parameter sind: move, select, create, resize, delete.

@disableHighlight

Schaltet die Highlight-Funktion eines Objektes aus. Generell ist in Cinco für die Modellierungselemente die Highlight-Funktion aktiviert, die z.B. erkennbar wird, wenn sich ein Objekt mit einer Kante verbinden läßt.

@doubleClickAction(parameter)

Kann einer Node, einer Edge, einem Container oder einem Graphmodell hinzugefügt werden. Der Parameter enthält die Aktions-Klasse, welche bei einem Doppelklick auf das jeweilige Element ausgeführt wird.

@file(parameter,...)

Kann einem EString-Attribut hinzugefügt werden und ermöglicht einen File-Chooser in der Cinco Properties View. Gültige Parameter sind Datei-Typen, die lediglich angezeigt werden sollen.

@grammar(parameter1, parameter2)

Kann einem EString-Attribut hinzugefügt werden und ermöglicht einen xtext-Editor in der Cinco Properties View. Der erste Parameter enthält den Pfad der xtext-Grammatik und der zweite Parameter den Pfad des Grammatik-Activators.

@icon(parameter)

Kann einer Node, einer Edge oder einem Container hinzugefügt werden. Ermöglicht dem Element ein neues Außehen. Der Parameter enthält den Pfad zu einer Bild-Datei.

@label(parameter)

Kann einen benutzerdefinierten Typ, welcher zu einem Modellelement gehört, um ein Label erweitern. Der als Parameter angegebene Name, wird angezeigt.

@multiLine

Ermöglicht, daß Testeingaben in der Cinco Properties View über mehrere Zeilen gehen. Wird meistens bei Strings verwendet. Bei Verwednung in einer Node, einer Edge oder einem Container, muß multiLine ebenfalls beim Style hinzugefügt werden.

@palette(parameter)

Gruppiert Elemente. Die Gruppierungen kriegen einen im Parameter definierten Namen.

@possibleValuesProvider(parameter)

Kann für Attribute benutzt werden und zeigt alle möglichen Werte, die dieses Attribut annehmen kann, in Form einer Combo-Box. Der Parameter muß den Pfad zu der Klasse, die die Funktionalität beschreibt, als String enthalten.

@propertiesViewHidden

Wenn ein Attribut dieses Meta Plug-In besitzt, wird es nicht in der Cinco Properties Viewangezeigt.

@readOnly

Wenn ein Attribut dieses Meta Plug-in besitzt, kann der Wert vom Attribut nicht mehr verändern sondern nur eingesehen werden.

@style(parameter1,...)

Kann für Nodes, Edges, Container und Graphmodels benutzt werden. Der erste Parameter muß ein Style aus der Style-Datei sein. Falls dieser Style Text erwartet, kann dieser über weitere Parameter mitgegeben werden. Sowohl als normaler String (Text") oder als Attribut(${name}).

@wizard

Kann einem Graphmodel hinzugefügt werden, um die Benutzung eines Wizards zu ermöglichen.

[21]

Hooks

Hooks ermöglichen das Reagieren des Models auf bestimmte Events, die eintreffen können. [23]

@postAttributeChange(parameter)

Kann für Nodes, Edges, Container, Types und dem Graphmodel benutzt werden und benötigt als Parameter den Pfad zu einer Klassen, welche die Klasse „CincoPosrtValueChangeListener“ erweitert. Ermöglicht das Reagieren auf Werteveränderung der Attribute.

@postCreate(parameter)

Kann für Nodes, Edges, Types, Graphmodel oder Container benutzt werden. Die als Parameter übergebene Klasse wird nach Erstellen des jeweiligen Elementes aufgerufen.

@postDelete(parameter)

Kann benutzt werden für Nodes, Container und Edges. Nach dem Löschen des Model-Elementes wird eine Aktion ausgeführt, welche in der als Pfad übergebenen Klasse, implementiert ist. Diese Klasse muß die Super-Klasse " CincoPostDeleteHook" erweitern

@postMove(parameter)

Kann benutzt werden für Nodes oder Container. Nach dem Bewegen des Model-Elementes, wird eine Aktion ausgeführt, welche in der als Pfad übergebenen Klasse, implementiert ist.

@postResize(parameter)

Kann benutzt werden für Nodes oder Container. Nach dem Verändern der Größe des Model-Elementes, wird eine Aktion ausgeführt, welche in der als Pfad übergebenen Klasse, implementiert ist.

@postSave(parameter)

Kann benutzt werden für das Graphmodel. Nach dem Speichern des Models, wird eine Aktion ausgeführt, welche in der als Pfad übergebenen Klasse, implementiert ist. Diese Klasse muß die Klasse " CincoPostSaveHook<>" erweitern.

@postSelect(parameter)

Kann benutzt werden für Nodes, Container und Edges. Nach dem Auswählen des jeweiligen Model-Elementes, wird eine Aktion ausgeführt, welche in der als Pfad übergebenen Klasse, implementiert ist. Diese Klasse muß die Klasse " CincoPostSelectHook<>" erweitern

@preDelete(parameter)

Kann benutzt werden für Nodes oder Container. Vor dem Löschen von Nodes, Edges oder Contaniern, wird eine Aktion ausgeführt, welche in der als Pfad übergebenen Klasse, implementiert ist.

[21]

Weitere Meta Plug-Ins

@generatable(parameter1,parameter2)

Kann benutzt werden, um Code zu generieren und wird einem Graphmodel hinzuge-

fügt. Paramter1 gibt die Klasse an, die den Code generiert und Parameter2 den Ort wo der Code generiert wird.

@mcam(parameter)

Wird dem Graphmodel hinzugefügt und aktiviert ohne Parameter mcam". Für Validierung wird der Parameter „check“ verwendet und für das Zusammenführen von Modellen „merge“.

@mcam_checkmodule(parameter)

Wird dem Graphmodel hinzugefügt und aktiviert ein Check-Modul, welches die Klasse „Check“ erweitert. Der Parameter enthält den Pfad zu Checkmodul.

@mcam_changemodule(parameter)

Wird dem Graphmodel hinzugefügt und aktiviert ein Changemodul, welches das Interface „info.scce.mcam.framework.modules.ChangeModule“ implementiert hat. Der Parameter enthält den Pfad zum Changemodul.

@mcam_label

Wird einem Attribut eines Modelelementes hinzugefügt. Der „Name“ repräsentiert nun das Modelelement.

@mcam_mergestrategy(parameter)

Wird dem Graphmodel hinzugefügt und sorgt dafür, dass die eigene Merge-Strategy, welche das Interface „info.scce.mcam.framework.strategies.merge.MergeStrategy“ besitzt, genutzt wird. Der Parameter enthält den Pfad zur MergeStrategy

[21]

2.3.5 Actions

Verwendung von Actions

Meta Plug-ins, wie „@contextMenuAction()“ oder „@doubleClickAction()“ benötigen als Parameter eine Action. Cinco liefert dafür die parametrisierbare abstrakte Klasse „CincoCustomAction“Der Typ-Parameter, kann jedes Modellelement aus der MGL-Klaße sein, also auch das gesamte Modell. Die abstrakte Klaße „CincoCustomAction“ verlangt drei Methoden, die implementiert werden müßen.

canExecute

Überprüft, ob die Action ausgeführt werden kann.

execute

Ausführung der Action

getName

Name der Action zum Anzeigen, in der Modellierungsumgebung.

Alternativ kann eine Action auch in Xtend geschrieben werden. Xtend ist komplett mit Java kompatibel und ermöglicht das Schreiben von Code, welcher einfacher zu lesen ist. [23]

2.3.6 Prime-Reference

Cinco besitzt ein Feature namens „Prime-Reference“ welches „Many-To-One“ Beziehungen erlaubt, so ist es zum Beispiel möglich ganze Modelle per „Drag-And-Drop“ in andere Modelle hineinzuziehen und dort zu verwenden. Dort können sie zu wiederverwendbaren und austauschbaren Komponenten-Bibliotheken zusammengefasst werden. Dadurch kann z.B. ein Attribut einer Node ein ganzes Modell referenzieren. Dabei wird das Attribut automatisch beim Erstellen der Node gesetzt. Dies führt zu einer Modell-Struktur in der besondere Nodes Modelle wiederspiegeln, in denen separat von verschiedenen Modellierern auf verschiedenen Abstraktionsleveln gearbeitet werden kann.[23]

Verwendung der Prime-Reference

Cinco's „Meta Graph Language“, kurz MGL, unterstützt drei Arten Prime-References bei Nodes zu definieren:

- **Nodes, die sich auf Modellelemente in der gleichen MGL-Klasse beziehen**
  Wird benutzt, wenn eine Node das ganze Graphmodell wiederspiegeln soll, wie zum Beispiel in hierarchischen Modellen. In diesem Fall enthält das Prime-Statement das Wort „this“ um die eigene Klaße referenzieren zu können.
- **Nodes, die sich auf Modellelemente einer anderen MGL-Klaße beziehen**
  Wird benutzt um verschiedene Modelle zusammenzuführen, um das Zielprodukt zu erhalten. In diesem Fall muß „import“ und der Pfad der anderen MGL-Klaße benutzt werden. Dieses Statement enthält anschließend noch einen Namen, damit es für das setzen der Prime-Reference genutzt werden kann.
- **Nodes, die sich auf Modellelemente beziehen, welche in einem Ecore Metamodell definiert sind**
  Ermöglicht die Kompabilität mit anderen Eclipse-basierten Frameworks, wie zum Beispiel nicht-grafische Modelle, die mit einem Xtext-Editor erstellt wurden. In diesem Fall muss „import“ und der Pfad des Ecore Metamodells benutzt werden. Dieses Statement enthält anschließend noch einen Namen, damit es für das setzen der Prime-Reference genutzt werden kann.

[23][28]

Meta Plug-ins für die Prime-Reference

Ein Graphmodell, kann mit „@primeviewer“ das Modell in der Modellierungsumgebung angezeigen und ermöglicht dadurch das Ziehen per „Drag-And-Drop“.Mit „@pvFileExtension(parameter1)“ können nur Dateien mit der Endung die im Parameter übergeben wurde, geöffnet werden. Bei Verwendung von „@pvLabel(parameter1)“ haben alle geöffneten Dateien den in parameter1 übergebenen Namen. [21]

2.4 Ontologie

Mittels einer Ontologie kann Wissen zusammengefasst und dargestellt werden. Sie gehört somit zu dem Gebiet der Wissensrepräsentation, in der es darum geht, explizites Wissen zu modellieren, zu ordnen und darzustellen.

2.4.1 Definition

Eine Ontologie ist eine Sammlung von geteiltem Wissen über eine bestimmte Domäne. Die dort beteiligten Menschen einigen sich auf das Wissen, das in dieser erlangt wurde. Somit entsteht eine Konzeptualisierung über das Thema. Dabei ist in einer Ontologie keine bestimmte Form vorgeschrieben. Häufig besteht sie aus einem Vokabular von Termen und Definitionen. Diese können wiederum in Relationen zueinander stehen. Die Terme und Definitionen dürfen in vier verschiedenen Formen vorliegen: Hoch informell, semi-informell, semi-formell und formell.
Ist eine Definition informell, besteht sie aus natürlicher Sprache. Die semi-informelle Definition beschränkt die Ausdrucksweise der natürlichen Sprache auf strukturierte Formen. Bei semi-formellen Sprachen wird eine künstliche formal definierte Sprache verwendet. Die formelle Definition benutzt definierte Terme mit formaler Semantik, Theoremen und dem Beweisen von Eigenschaften [34].
Eine Ontologie kann auch als Graph vorliegen. Diese Art der graphenbasierten Wissensrepräsentation verwendet Knoten, um Daten oder Konzepte darzustellen und Kanten, um Beziehungen zwischen Knoten abzubilden. Dabei dürfen unterschiedliche Knoten- und Kantentypen verwendet werden.

2.4.2 Erstellung einer Ontologie

Eine Ontologie kann für eine bestimmte Domäne auf zwei verschiedenen Wegen erstellt werden: Zum einen kann eine Ontologie ohne andere Ontologien gebaut, zum anderen können bestehende Ontologien für die eigene Domäne angepasst werden. Für diesen Prozess gibt es Frameworks, die unterstützend wirken.

Um eine Ontologie komplett neu zu erstellen, werden nach der Uschold und King's Methode vier Schritte durchlaufen [13]. Zunächst müssen der Zweck und der Anwendungsbereich identifiziert werden. Danach beginnt die Erstellung der Ontologie, die die Erfassung, die Codierung und die Integration bereits existierender Ontologien umfasst.
Bei der Erfassung werden die Schlüsselkonzepte und Beziehungen der Domäne erfasst und genaue und eindeutige Definitionen in Text und Termen, die zu den Konzepten und Beziehungen passen, erstellt.
In der dritten Phase, der Codierungsphase, geht es darum eine Repräsentation für die Konzepte, die im vorherigen Schritt identifiziert wurden, in eine formale Sprache zu bringen. Das beinhaltet das Einigen auf Basisterme, die dafür benutzt werden die Ontologie zu spezifizieren. Darunter fallen beispielsweise Klassen, Entitäten oder Relationen. Häufig wird dies die Meta-Ontologie genannt, weil sie fähig ist, die gebaute Ontologie zu beschreiben. Als Nächstes muss in der Codierungsphase eine Repräsentationssprache gewählt werden, die die Meta-Ontologie unterstützt. Zuletzt wird der Code geschrieben.
Im Anschluss oder während der Phase kann es sinnvoll sein, andere bereits bestehende Ontologien zu benutzen und für die Eigene zu verwenden.
In der vierten Phase wird die entstandene Ontologie im Bezug auf die Spezifikationen, die Kompetenzen, die sie erfüllen soll, oder die Realität evaluiert und dokumentiert [34]
Ein weiterer Ansatz geht über die Prädikatenlogik der ersten Stufe. Hier werden zuerst die Hauptszenarien, in denen die Ontologie genutzt wird, identifiziert. Danach wird eine Menge von in natürlicher Sprache formulierten Fragen, den Kompetenzfragen, aufgestellt. Mit ihnen ist es möglich, den Nutzen der Ontologie zu erfassen. Mit den Fragen und den daraus resultierenden Antworten werden die Konzepte, Eigenschaften, Relationen und Axiome erschlossen und in Prädikatenlogik dargestellt [13].

2.4.3 Kriterien für Ontologien

Nachdem eine Ontologie erstellt wurde, kann sie bewertet werden. Dabei gibt es nach Gruber fünf Kriterien eine Ontologie objektiv zu bewerten [15]:
Eindeutigkeit, Konsistenz, Erweiterbarkeit und ein minimaler Verschlüsselungsbias sind die ersten vier. Unter letzterem ist zu verstehen, dass die Konzeptualisierung des Wissenslevels möglichst unabhängig zu dem spezifischen Symbollevel ist. So soll es möglich sein, dass mehrere Systeme, die unterschiedliche Repräsentationssysteme haben, die Ontologie nutzen können.
Das letzte Kriterium ist die minimale ontologische Verpflichtung. Damit ist gemeint, dass die erstellte Ontologie einige wenige, aber nicht zu viele Annahmen über ihre Welt haben darf, damit der Benutzer diese zur individuellen Anpassung nutzt.
Diese Kriterien können nicht gleichzeitig erfüllt werden, da sich beispielsweise Eindeutigkeit und minimale ontologische Verpflichtung ausschließen. Zum einen soll das Wissen möglichst

detailiert definiert sein, zum anderen sollen nur Grundannahmen gemacht werden. Somit sollen diese Kriterien nur eine Orientierung darstellen. Was für die jeweilige Ontologie relevant ist, muss individuell bewertet werden [15].

2.5 DIME

Das *DyWA Integrated Modeling Environment*[9] (DIME) ist eine Entwicklungsumgebung für die modellgetriebene Entwicklung von Webapplikationen. In den folgenden Kapiteln wird zuerst eine Einführung in das Konzept von DIME geliefert. Anschließend folgt eine Erläuterung des Ablaufes der Codegenerierung und eine Übersicht über die Benutzeroberfläche. Abschließend wird die Modellierung von Daten, Prozessen und GUIs vorgestellt.

2.5.1 Konzept

DIME soll die Umsetzung von Anwendungen auch Personen ermöglichen, die keine Programmiererfahrungen haben. Hierfür kann der Nutzer verschiedene Modelle erstellen, die zusammen das gewünschte Aussehen und Verhalten der Applikation abbilden. Die Generierung der lauffähigen Anwendung erfolgt durch die Software, die im nächsten Kapitel genauer erläutert wird.
Der Nutzer muss im Kern nur drei Modelle erstellen:

1. Datenmodell mit den Verknüpfungen der Daten untereinander
2. Prozessmodell, welches die Abläufe und die Logik der Applikation abbildet
3. GUI-Modell, das die gewünschte GUI beschreibt

Die einzelnen Modelle können während der Erstellung und Bearbeitung validiert werden, sodass das gewünschte Verhalten erreicht wird. Hierfür unterstützt DIME den Entwickler mit Hinweisen und Warnungen zum aktuellen Stand der Modelle. Nach der erfolgten Modellierung durch den Nutzer, und die Generierung durch DIME erfolgt das Deployment mithilfe des Frameworks DyWa (Dynamic Web Application) [25].
Insgesamt folgt DIME den Paradigmen *One Thing Approach* (OTA) [33] und *eXtreme Model-Driven Design* (XMDD) [19], erlaubt also eine agile nutzerfokussierte Entwicklung mit dem Schwerpunkt auf ein einzelnes umfangreiches Modell. Im Falle von DIME ist dieses in die drei oben genannten Untermodelle aufgeteilt.

2.5.2 Code Generierung

Jedem Modelltypen ist eine eigene Modellierungssprache zugeordnet. Diese Sprachen bestehen alle aus verschiedenen Elementen, die mit Kanten verbunden, und verschachtelt aufgebaut werden können. Ergänzt werden diese Modelle durch eine *.dad* Datei, die einen

Einstiegspunkt für die Generierung darstellt. Aus den hier hinterlegten Informationen kann außerdem die Landingpage der Webapplikation extrahiert werden.
DIME-Anwendungen verwenden für die Speicherung von Informationen Datenbanken. Der Zugriff auf diese Datenbank erfolgt mit dem DyWA Framework. Da das Konzept von DIME gebietet, dass auch Nutzer ohne Programmiererfahrung eine Anwendung entwickeln können, muss keine Datenbank manuell angelegt werden, sondern diese wird im Hintergrund aus dem Datenmodell generiert.
Um das Aussehen der Anwendung festzulegen und zu Konfigurieren, wird das Oberflächenmodell verwendet. Auch hier wird der Einstieg in die Entwicklung für ungeübte Nutzer vereinfacht, indem vorgegebene GUI Elemente zu den gewünschten Oberflächen zusammengefügt werden können. Diese können wiederverwendet und verschachtelt werden, um eine schnellere Entwicklung zu ermöglichen.
Die Logikmodelle in DIME basieren auf *jABC4*, welches ein Framwork für für die modellgetriebene Anwendungsentwicklung ist. Dabei wird die Koordination von allen Prozessen durch einen Graphen modelliert [32]. Dieser Graph besteht aus *Service Independent Building Blocks* (SIBs), die jeweils beliebige Funktionen enthalten können. Aus diesem Modell wird die Logik der generierten Anwendung abgeleitet, die die Funktionen und Datenflüsse beinhaltet.

2.5.3 Modellierung

In diesem Kapitel werden die einzelnen Modelle präzisiert und deren Aufbau und Zusammenspiel erläutert. Beispiele und Tutorials zur konkreten Umsetzung von Applikationen sind auf der Webseite der DIME-Dokumentation[1] zu finden.

Daten

Daten werden in DIME in mehrere Datentypen aufgeteilt:

1. *Primitive Types*: Simple Daten, die wiederum in die Varianten Text, Integer, Real, Boolean, Timestamp und File eingeteilt werden. Diese werden im Datenmodell als Attribute von anderen Typen dargestellt.

2. *Complex Types*: Komplexere Typen, die mehrere verschiedene Attribute beinhalten können und außerdem mit anderen Typen durch Assoziationen verknüpft werden können. Assoziationen werden zusätzlich zu einer optionalen Kante zwischen den Typen auch als Attribut im beinhaltenden Typ angezeigt.

3. *Abstract Types*: Abstrakte Typen, die nicht instanziiert werden können, sondern die nur erstellt werden können, damit andere Typen von ihnen die Verbindungen oder Assoziationen erben.

[1] https://projekte.itmc.tu-dortmund.de/projects/scce-documentation/wiki/DIMEDocumentation

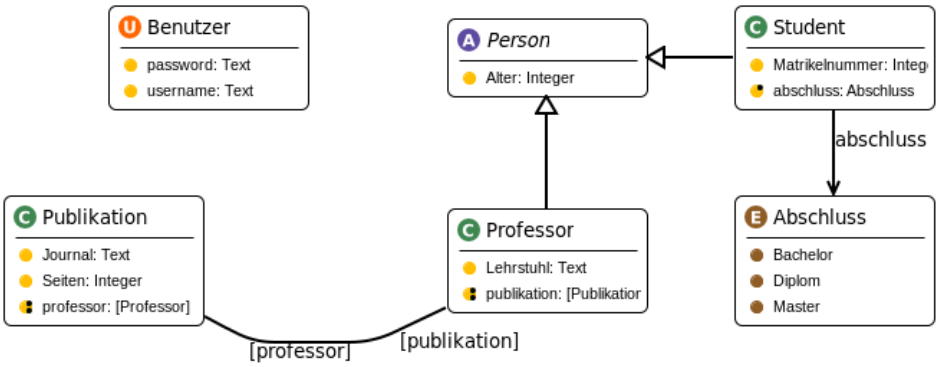


Abbildung 2.7: Beispielhaftes Datenmodell mit den verschiedenen Datentypen

4. *Enums*: Typen, die einen klar definierten, explizit aufgelisteten Definitionsraum haben. Diese können keine weiteren Attribute beinhalten.

5. *User Types*: Benutzertypen, die sich mit festgelegten Attributen (bspw. Name und Passwort) ins System einloggen können.

Die Attribute der Typen können als einzelne Elemente, oder als Listen vorliegen, was vom Nutzer konfiguriert werden kann. Die Assoziationen (uni- oder bidirektional) werden als Linien, die Vererbungskanten als Pfeile dargestellt. Die Richtung der Verbindung wird optisch bei den Assoziationen durch einen an der Kante aufgeführten Attributnamen, und bei den Pfeilen durch den Zeiger des Pfeils ausgedrückt. Den verschiedenen Datentypen ist ein farbiges Symbol mit einem entsprechenden Buchstaben angefügt, um eine intuitive Unterscheidung auch bei flüchtiger Betrachtung zu ermöglichen. Ein beispielhaftes Datenmodell ist in Abbildung 2.7 dargestellt.

Prozesse

Für die Erstellung des Logikmodells unterstützt DIME die Integration von verschiedenen Prozesstypen. Diese leiten sich von den SIBs ab und werden in einem Prozessgraphen aufgelistet. Zu diesen Prozesstypen gehören die *Basic-Prozesse*, die als kleinste Einheit die simplen Prozesse abbilden. Außerdem existieren noch *Interactable-Prozesse*, deren Ein- und Ausgabe-Interfaces auf nicht-native Typen eingeschränkt sind, da die Interaction-SIBs als Verbindung zwischen Frontend und Backend von DIME fungieren, und das Frontend keine nativen Typen kennt. Vervollständigt werden die Prozesse durch *Interaction Prozesse* für die Verknüpfungslogik, *Long-Running-Prozesse*, die einen gesamten Life-Cycle von Einheiten beschreiben, sowie *Security-Prozesse* für Zugriffskontrolle [9].
Der Prozessgraph besteht neben den genannten SIBs auch aus Start- und Endelementen und aus einem Datenmodell. Die Eingaben und Ausgaben der einzelnen Prozesse können durch (gestrichelte) Pfeile verbunden werden. Der Prozessfluss wird mit (durchgängigen) Pfeilen dargestellt. Die Ausgaben von Elementen werden mit orangefarbigen Ausgabeports

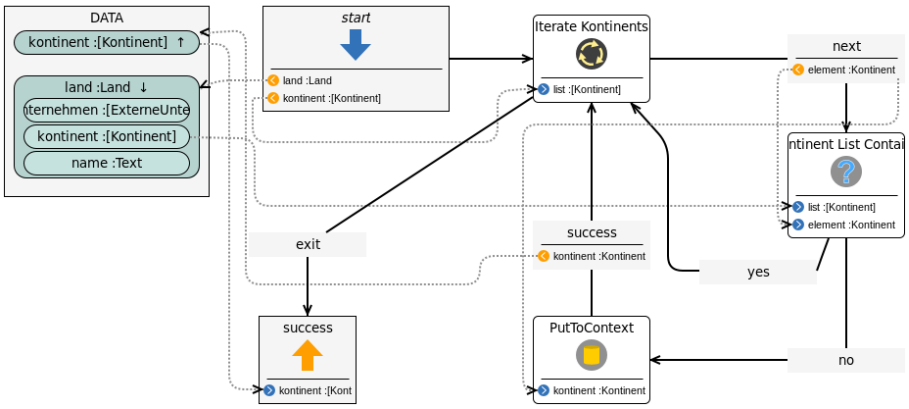


Abbildung 2.8: Ein Prozessmodell mit Datenflüssen und dem Prozessfluss von *start* zu *success*

ausgedrückt. Eingabeports werden als Kontrast dazu blau dargestellt. Ein Beispiel für ein so modelliertes Logikmodell ist in Abbildung 2.8 visualisiert.

GUIs

Die grafischen Oberflächen der Anwendung können mit eigenen GUI-Modellen konfiguriert werden. Diese bestehen aus einer Verknüpfung zum Datenmodell und GUI-Komponenten (*Components*). Die Komponenten erlauben eine grundlegende Datendarstellung und Eingabeverarbeitung. Außerdem kann das Layout der Webapplikation angepasst werden.
Die Komponenten sind in vier Kategorien aufgeteilt:

1. *Structure*: Diese Strukturkomponenten erlauben die Anordnung und Gruppierung von Komponenten. Die Oberfläche kann dabei in Reihen, Spalten und Panels eingeteilt werden. Außerdem erlaubt DIME die Gruppierung in Tabs.

2. *Interaction*: Interaktionskomponenten sind Bausteine, die eine Bearbeitung von Daten in Form von (Radio) Buttons, Textfeldern, Comboboxen und Checkboxen ermöglicht. Zusätzlich dazu können Formulare erstellt werden, die einen gesammelten Datentransfer ermöglichen.

3. *Content*: Inhaltskomponenten stellen Daten dar. Die typischen Arten der Textdarstellung als Überschrift, Textfeld und Liste werden durch Möglichkeiten zur Einbettung von Tabellen, Bildern und Statusleisten vervollständigt.

Die einzelnen Komponenten können individuell durch Attribute, wie bspw. die Farbe, angepasst werden. Die Anordnung der Komponenten erfolgt durch Drag&Drop und ermöglicht eine vage Vorschau auf die generierte GUI.
Der zur jeweiligen GUI zegehörige Teil des Datenmodells ist in dieser Ansicht ebenfalls vorhanden. Zwischen den Daten und GUI-Objekten können verschiedene Verknüpfungen

erstellt werden. Die beiden wohl wichtigsten Kontrollflusskanten sind die *IF*- und die *FOR*-Kanten.
Eine IF-Kante verbindet eine GUI-Komponente mit einem Datenelement. Sie erlaubt das Ein- und Ausblenden der Komponente abhängig von dem Wert des Datenelements. Ein solches Verhalten ist beispielsweise gewünscht, wenn der Wert von einem Datenelement in der Oberfläche nur dargestellt werden soll, wenn dieser gesetzt ist.
Die FOR-Kante erlaubt das Durchlaufen einer Liste mit der Verarbeitung der einzelnen Elemente. Eine FOR-Kante von einer Liste im Datenkontext zu einem Panel hat beispielsweise zur Folge, dass für jedes Element der Liste ein Panel angelegt wird, und dieses in diesem Panel genauer dargestellt werden kann.

2.5.4 Benutzeroberfläche

DIME ist ein *Eclipse RCP (Rich Client Platform) Produkt* [20] und baut somit auf dem Eclipse Framework auf. Dieses wurde durch Plug-Ins erweitert, um die Bearbeitung von den zuvor vorgestellten Modellen zu erleichtern [9]. Die Entwicklung von DIME erfolgt mit der CINCO SCCE Meta Tooling Suite, welche es erlaubt, aus der Beschreibung von Metamodellen aus Modellierungssprachen RCPs generiert werden können [24] (Siehe auch Kapitel 2.3).
Verschiedene Views (also eigenständige Ansichten innerhalb des Anwendungsfensters) werden zur Ansicht und Bearbeitung von Informationen bereitgestellt. Neben den Eclipse-typischen Views wie dem *Projekt Explorer* bietet DIME also folgende fachbezogene Views:

1. *Diagram-View*: Stellt die verschiedenen Modellgraphen in Diagrammform dar und erlaubt die Bearbeitung dieser. Die Modellelemente werden dafür auf einem Raster angeordnet und können mit den jeweiligen Kanten verbunden werden.

2. *Model-Views*: Für jeden der Modelltypen existiert eine Model-View, welche die zu dieser Modellart passenden Dateien anzeigt. Die drei Views können beliebig angewählt, und für die Navigation zu den entsprechenden Daten verwendet werden.

3. *Properties*: Wenn eine Komponente oder eine Kante in der Diagramm-View ausgewählt wurden, werden in der Properties View die Attribute von diesem Element angezeigt. Die Attribute können außerdem bearbeitet werden.

4. *Model-Validation*: Um dem Nutzer eine Validierung der erstellten Modelle zu ermöglichen, ohne aus diesen erst die ausführbare Anwendung generieren zu müssen, bietet DIME eine Model-Validation-View. Wenn DIME Fehler in den Modellen erkennt, wird in diesem integrierten Fenster eine Warnung oder eine Fehlermeldung angezeigt, die eine Fehlersuche und -behebung erleichtern soll.

Alle aufgeführten Views können vom Benutzer beliebig ein- und ausgeschaltet werden. Außerdem kann die Größe und Position verändert werden. Damit kann der Nutzer den Aufbau

von der DIME-GUI an seine Bedürfnisse anpassen.
Mit den genannten GUI-Elementen wird der Anwender der DIME Software bei der Entwicklung der drei Modelle unterstützt und auf eventuelle Fehler hingewiesen. Auch Nutzer ohne Programmiererfahrung können eine Modellierung durchführen.

2.6 Canvas

Im Folgenden soll das Prinzip eines Canvas weiter erläutert werden. Dabei geht es zunächst um die allgemeine Definition eines Canvas und dessen Nutzen innerhalb einer Organisation. Eingegangen wird auf die Funktionen, die zur Übersicht von Abläufen und Daten einer Organisation dienen. Auf der Grundlage der für einen Canvas verwendeten Daten, kann dieser einen für die Organisation spezifischen Nutzen haben. Darauf aufbauend wird der Business Model Canvas erläutert. Bei diesem handelt es sich um einen speziellen Canvas, der im weiteren Verlauf dieser Arbeit noch eine zentrale Rolle spielen wird.

2.6.1 Canvas in einer Organisation

Unternehmen sehen sich in ihren Bestehen mit verschiedensten Einflüssen von außerhalb der eigenen Organisation konfrontiert. Diese reichen von Finanzkrisen, die das eigene Bestehen auf dem Mrkt gefährden, bis hin zu dem Verlangen nach einer nachhaltigeren und umweltschützenden Produktion, die von Seiten des Marktes verlangt wird. Bei gewissen Aspekten kann ein durch einen Canvas aufgestelltes Business Model dabei helfen, diese Ziele zu ordnen und auf Einflüsse koordiniert reagieren zu können. Das Business Modell definiert sich dabei als das rationale Wissen, wie die Organisation Werte kreiert, damit umgeht und nach außen verkörpert. Dabei stehen drei Schlüsselfaktoren bereit, nach denen die Organisation versucht, ihr Handeln auszurichten:

- Wie werden die Hauptkomponenten und Funktionen der Organisation integriert, um Werte nach außen zum Kunden hin zu vermitteln?
- Wie werden diese anteilig innerhalb der Organisation, deren Zuliefererkette und den Interessenvertretern miteinander verknüpft?
- Wie generiert die Organisation aus diesen Verbindungen für sich einen Mehrwert, bzw. profitiert davon?

Wenn das Business Modell der Organisation richtig verstanden, gibt es neben dem Einblick in die High Level Strategien hinaus Informationen darüber bekannt, welche Aktionen getroffen werden können, Strategien beizubehalten oder entsprechend verändern zu können. Überführt in einen Canvas, können die Verbindungen, die innerhalb der Organisation bestehen und bisher auf rein impliziter Ebene vorhanden sind, in explizites Wissen umgewandelt und das Business Modell für alle beteiligten klarer herausgestellt werden. So kann

es dazu kommen, dass bisher unbekannte Strategien und Innovationen zu ermitteln sind. Diese können den Nutzen haben, Werte zu steigern. Was bedeutet, die Transformation von bisherigen Aktionen und Interaktionen der Unternehmensbereiche in neue Wege zu leiten. Eines dieser Canvas Modelle, ist das von Alexander Osterwalder entwickelte Business Model Canvas.

2.6.2 Business Model Canvas

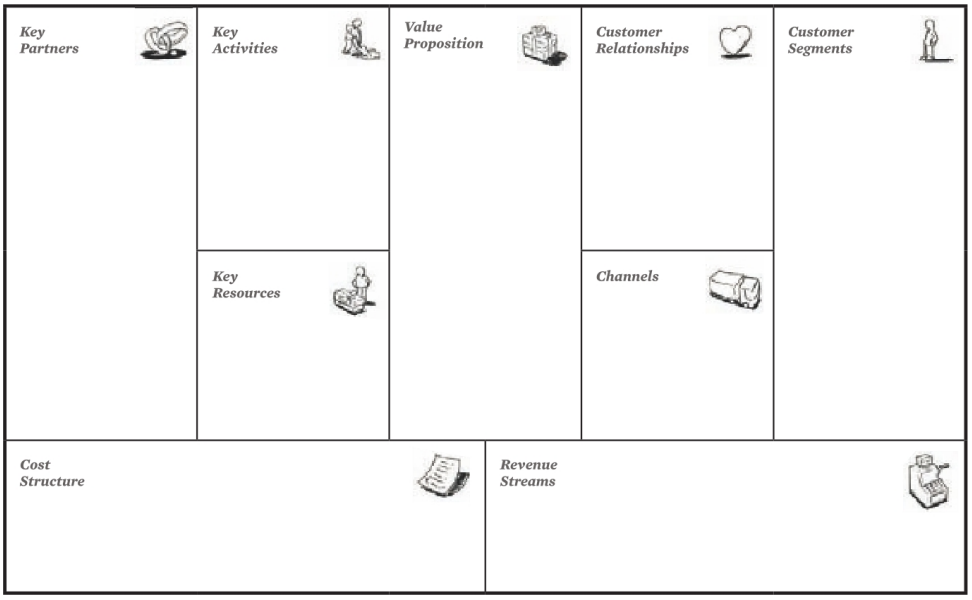


Abbildung 2.9: Das Business Model Canvas

Das Business Model Canvas ist ein von Alexander Osterwalder entwickeltes Hilfsmittel zur Erstellung und Analyse von Geschäftsmodellen.[26] Während klassische Businesspläne oft unübersichtlich und mehrere Seiten lang sind, erlaubt das Business Model Canvas eine schnelle Erstellung und Analyse eines Geschäftsmodells auf einen Blick durch Verwendung eines Tabellensystems.

2.6.3 Die neun Blöcke des Business Model Canvas

Das Business Model Canvas ist in neun Blöcke aufgeteilt[26], die sich wiederum in zwei Hälften aufteilen lassen.
Während die linke Seite, mit den Blöcken *Key Partners*, *Key Activities*, *Key Resources* und *Cost Structure*, sich mehr auf den geschäftlichen Teil konzentriert, beschäftigt sich die rechte Seite, mit den Blöcken *Customer Relationships*, *Customer Segments*, *Channels* und *Revenue Streams*, eher mit den Kunden. Die linke Hälfte wird daher als die interne und die rechte als externe Ansicht bezeichnet. Beide Hälften werden im Block *Value Proposition* zusammengeführt, der den zentralen Punkt des Business Model Canvas darstellt. In ihm

wird die konkrete Wertschöpfung des Geschäftsmodells definiert. Im Folgenden werden die neun Blöcke und deren vorgesehenen Inhalte genauer erläutert:[26]

Customer Segments

Das Herzstück eines jeden Geschäftsmodells ist die Kundenbasis, denn ohne zahlende Kundschaft hat ein Unternehmen keinen sicheren finanziellen Standfuß. In einem Geschäftsmodell sollten eine oder mehrere Kundensegmente herausgearbeitet werden. Unterschiedliche Kundengruppen können sich zum Beispiel ergeben, wenn die Kunden unterschiedliche Interessen haben, über unterschiedliche Kanäle angesprochen werden oder einfach nur unterschiedlich viel für einen Service oder ein Produkt zahlen wollen. Die Kundensegmentierung kann so granular wie nötig gewählt werden. Es kann sich sowohl auf den Massenmarkt konzentriert werden, wo alle angesprochenen Kundengruppen im Großen und Ganzen die selben Interessen verfolgen, bis hin zum extremen Nischensegment. Darüber hinaus gibt es auch sogenannte *Multi-Sided Markets*, bei der eine von zwei Kundengruppen ohne die Andere nicht existieren würde. Werbeagenturen würden zum Beispiel keine Werbung auf Nachrichtenseiten schalten, die keine Nutzer haben.

Value Propositons

Ein weiterer zentraler Punkt eines Geschäftsplans ist das angebotene Produkt oder Service. Es muss herausgearbeitet werden, warum sich Kunden für das Produkt entscheiden sollten. Anreize dafür können entweder qualitativer oder quantitativer Natur sein. Quantitative Anreize sind zum Beispiel ein günstigeres Produkt als bisher Verfügbare oder eine schnellere Befriedigung der Wünsche eines Kunden, wobei qualitative Anreize eher ein Produkt mit einem ansprechenderen Design oder eine auf den jeweiligen Kunden maßgeschneiderte Benutzererfahrung eines Services sein kann. Des Weiteren kann auch ein komplett neues Produkt auf den Markt gebracht werden, das Kundenwünsche erfüllt, die sie bisher noch nicht einmal in Betracht gezogen haben.

Channels

Wenn ein Unternehmen sich auf eine oder mehrere Kundengruppen fokussiert und ein passendes Produkt entwickelt hat, stellt sich die Frage, wie das Produkt an den Kunden gebracht wird. Um einen geeigneten Vertriebsweg zu erstellen, sollten fünf Phasen durchlaufen werden:

1. Zunächst sollte erarbeitet werden, wie der Kunde auf das Produkt aufmerksam wird.
2. Daraufhin muss der Kunde von dem Produkt überzeugt werden.
3. Dann muss geregelt werden, wie der Kunde das Produkt kaufen kann.

4. Als nächstes muss überlegt werden, wie dem Kunden das Produkt geliefert wird.

5. Schlussendlich muss aufgezeigt werden, wie dem Kunden Service angeboten wird, nachdem der Kauf stattgefunden hat.

Zu beachten ist dabei, dass es zwei Typen von Channeln gibt: Den direkten Channel und den indirekten Channel. Beim direkten Channel wird das Produkt zum Beispiel über eine eigene Webseite vertrieben und beworben. Beim indirekten Channel hingegen wird das Produkt über Dritte angeboten, zum Beispiel wird es in einem Geschäft einer Supermarktkette oder in großen Internetkaufhäusern angeboten. Zwar sind die Gewinnmargen beim direkten Vertrieb höher, jedoch auch die Kosten und möglichen Risiken. Eine gute Mischung aus beiden Channeltypen muss also gefunden werden.

Customer Relationships

Im Geschäftsmodell sollte auch die Kundenbeziehung genauer definiert werden. Fragen, die sich dabei stellen könnten wären: Wie werden Kunden gehalten, wie werden neue Kunden gewonnen und wie werden die Verkaufszahlen erhöht. Dazu gibt es verschiedene Möglichkeiten über die Kundenbeziehung. Zum Beispiel kann der Kunde durch Anbieten eines Call-Centers oder durch einen Chat mit einem Kundenservice-Mitarbeiter auf der eigenen Webseite beim Kauf des Produkts aktiv unterstützt werden. Weiterhin kann zum Produkt auch ein Internetforum angeboten werden, in dem sich Kunden gegenseitig Hilfestellung leisten. Ferner ist auch das Prinzip der *Co-Creation* zu beachten, bei dem die Kunden aktiv das Produkt, zum Beispiel auf der eigenen Webseite, durch Bewertungen bewerben. Auch sollten vollständig automatisierte Kundenbeziehungen in Betracht gezogen werden. Dem Kunden können beispielsweise Produkte angeboten werden, die gut mit bisherigen Produkten harmonisieren oder aber auch Ressourcen bereitgestellt werden, mit denen der Kunde sich bei Problemen selbst helfen kann, wie zum Beispiel eine Seite mit häufig auftretenden Problemen des Produktes und deren Lösungen (FAQ).

Revenue Streams

Ein weiterer zentraler Punkt eines jeden Geschäftsmodells ist der Revenue Stream, der sich mit der Frage beschäftigt: Wie wird mit dem Produkt überhaupt Geld verdient? Zunächst gibt es den traditionellen Weg: Das Produkt ist physischer Natur und wird nach einer Einmalzahlung an den Kunden übergeben. Je nachdem, wie das Produkt beschaffen ist, gibt es weiterhin die Möglichkeit der Vermietung. Hierbei hat der Kunde den Vorteil, nicht den gesamten Produktwert zahlen zu müssen und trotzdem das Produkt solange verwenden zu können, wie er es benötigt. Weiterhin gibt es bei digitalen Produkten die Möglichkeit einer Nutzungsstaffelung. Je mehr ein Kunde das Produkt benutzt, desto mehr muss er auch zahlen (siehe: Mobilfunkverträge). Weitere Einnahmequellen sind ein Abonnement-basiertes

Modell bei digitalen Gütern: Der Kunde zahlt also nur solange für das Produkt, wie er es auch benutzt. Ferner kann er das Produkt auch lizenzieren. Schlussendlich muss ein Preismodell erarbeitet werden. Als erste Möglichkeit kann ein Fixpreis angegeben werden, der von der Ausstattung des Produktes, des avisierten Kundensegmentes oder aber der abgenommenen Produktmenge abhängig ist. Zweitens kann ein dynamischer Preis angegeben werden, der durch Verhandlungen, Auktionen oder aber durch Angebot und Nachfrage bestimmt wird.

Key Resources

In einem Geschäftsmodell sollten auch immer die benötigten Ressourcen bedacht werden. Es sind vier unterschiedliche Ressourcen zu beachten, von denen ein Unternehmen abhängig sein kann: physische, intellektuelle, menschliche und finanzielle Ressourcen. Physische Ressourcen sind zum Beispiel Lagerräume, um das fertige Produkt aufzubewahren oder Fertigungsstraßen, um das Produkt herzustellen. Intellektuelle Ressourcen sind unter anderem eine bekannte Marke, internes Firmenwissen oder Copyrights, die dann lizenziert werden können. Menschliche Ressourcen sind hingegen etwa sachkundige Facharbeiter, die für eine fortwährende Weiterentwicklung des Produktes sorgen können oder überzeugende Verkäufer, die das Produkt gut verkaufen können. Finanzielle Ressourcen sind schlussendlich Bargeld oder Kredite, von denen das Unternehmen direkt abhängig ist.

Key Activities

Weiterhin muss im Geschäftsmodell erörtert werden, was ein Unternehmen wirklich machen muss, damit das Produkt erfolgreich wird. Dies ist hauptsächlich vom Sektor, in dem das Produkt vertrieben wird, abhängig. Handelt es sich um ein physisches Produkt, wird eher die Produktion im Vordergrund stehen. Sind die Kunden hingegen an Problemlösungen interessiert, wird die Hauptaufgabe des Unternehmens eher in eine Forschungsrichtung gehen: Wie können bestehende Prozesse beschleunigt werden oder gibt es allgemein bessere Prozesse, um ein Ziel zu erreichen? Bietet das Unternehmen hingegen eine Plattform an, liegt das Hauptaugenmerk auf der Optimierung der Plattform, damit mehr Kunden gewonnen werden können.

Key Partnerships

Es ist für Unternehmen nicht praktikabel, alle Ressourcen im Vertriebsprozess selbst zu besitzen. Die Hauptargumente für Unternehmenspartnerschaften sind die Kostenreduzierung und Risikominimierung. Zum Beispiel kann die Zulieferung von Teilen für die Produktion oder die Logistik der Produkte an einen Partner ausgelagert werden. Weiterhin kann es von Vorteil sein, mit Mitbewerbern eine Partnerschaft aufzubauen, um den Umsatz des

gemeinsamen Unternehmenssektors zu steigern und das Risiko für das einzelne Unternehmen zu minimieren. Ferner kann ein Unternehmen auch Ressourcen von Drittanbietern anmieten, um die eigenen Kosten zu reduzieren (siehe: Betriebssysteme von Smartphones).

Cost Structure

Abschließend sind natürlich auch die Kosten eines Unternehmens zu betrachten. Wo liegen die größten Ausgabenpunkte eines Produktes? Dabei zu unterscheiden sind zwei verschiedene Kostenstrukturen: Einerseits die maximale Minimierung aller Kosten und andererseits die Wertsteigerung eines Produktes. Während das eine Modell darauf beruht, alle Kosten so gering wie möglich zu halten, durch zum Beispiel vollständige Automation des Services und maximales Outsourcing, beruht das andere Modell darauf, durch ausgedehnten, exklusiven Service oder sonstige Boni den Wert des Produkts zu steigern. Kostenstrukturen lassen sich in mehrere Bereiche einteilen: Zunächst gibt es die Fixkosten, die unabhängig von der Produktionsmenge anfallen. Dazu gehören die Gehälter der Mitarbeiter oder die Mietkosten für angemieteten Räume. Dann wiederum gibt es variable Kosten, die von der produzierten Produktmenge abhängig sind. Dazu gehören auch Teile für die Produktion eines Produkts: Je mehr Teile einem Zulieferer abgenommen werden, desto geringer sind auch die Kosten pro Teil. Wenn ein Unternehmen bereits etabliert ist, senken sich ebenfalls die Kosten für jedes neu entwickelte Produkt. Bestehen zum Beispiel bereits Vertriebs- und Marketingwege für ein bestehendes Produkt, senken sich auch die individuellen Kosten für jedes neue Produkt.

2.6.4 Vor- und Nachteile des Business Model Canvas

Das Business Model Canvas wurde als analoge Schablone, die ausgedruckt und beschriftet werden kann, entwickelt. Dies mag von Vorteil sein, wenn lediglich ein Team lokal an einem Business Model Canvas arbeitet, stellt jedoch einen erheblichen Nachteil dar, wenn mehrere Teams örtlich voneinander getrennt zusammenarbeiten sollen. Daher existieren inzwischen auch diverse Tools, die das verteilte Arbeiten erlauben. [6][3].
Weiterhin gibt das Business Model Canvas zwar eine Struktur für den kreativen Entwicklungsprozess vor, die eine einfachere Vermittlung des Businessplans erlaubt, schränkt jedoch gleichzeitig die Benutzer auf die neun Blöcke ein und blockiert somit auch den Entwicklungsprozess. [12] Ferner wird als Kritikpunkt herangezogen, dass die Blöcke sich teilweise überschneiden und miteinander verflochten sind, was zu Verwirrungen oder dem kompletten Auslassen von ganzen Blöcken bei Benutzern führen kann. [12] Darüber hinaus bietet das Business Model Canvas keine gute Analysemöglichkeit der Konkurrenz und eignet sich daher eher für einen innovativen Entwicklungsprozess, als zur Transformation eines bereits bestehenden Businessplans. [11] Außerdem gibt das Business Model Canvas dem Benutzer keinerlei Motivation dazu, Belege für Annahmen innerhalb eines Blockes

anzugeben, was die Überprüfbarkeit und Nachvollziehbarkeit dieser Annahmen erschwert. [12]
Abschließend bildet das Business Model Canvas jedoch durch seine einfache Gestaltung und die Vielzahl von Einstiegspunkten und Perspektiven ein gutes Hilfsmittel zur einfachen Kommunikation und Erstellung eines Businessplans. [12]
Ferner existiert mit dem *Value Proposition Canvas* eine Erweiterung des Business Model Canvas, welches die Blöcke Value Proposition und Customer Segments granularer aufteilt und in direkte Beziehung setzt. Darauf soll hier jedoch nicht weiter eingegangen werden.

Kapitel 3

Konzepte Ontologie

Die konkrete Ontologie, die verwendet wird, um die im dieser Arbeit aufgezeigte DSL zu entwickeln, stammt von der Firma Schulz Systemtechnik GmbH. Diese zählt zu den führenden Entwicklern von ganzheitlichen Automatisierungslösungen für verschiedenste Branchen. Diese umfassen die Mechanik, sowie die Elektrotechnik und Informatik. Dadurch kann die Schulz Systemtechnik GmbH die gesamte Wertschöpfungskette in eigener Hand abbilden [2].

3.1 Ontologie

Die Firma Schulz Systemtechnik GmbH plant, ihre interne Aufbauorganisation zu einer Matrixorganisation hin aufzubauen. Dabei bezeichnet die Aufbauorganisation eines Unternehmens, die hierarchische Gliederung in sogenannte Organisationseinheiten unterschiedlichen Umfangs. Aufbauorganisationen können grafisch mit der Hilfe von Organigrammen dargestellt werden. Die von den Organisationseinheiten zu erfüllenden Aufgaben werden in Funktions- und Aufgabenbeschreibungen festgehalten, wobei verschiedenen Arten der Aufbauorganisationen bestehen, wobei die von der Firma angestrebte Matrixorganisation eine spezielle Art davon darstellt. Die Matrixorganisation wird dadurch gekennzeichnet, dass die Bildung der organisatorischen Einheiten unter gleichzeitiger Anwendung von zwei Gliederungskriterien geschieht.

Die Auslegung der Organisation eines Unternehmens erfolgt nach vier Prinzipien:

1. **Zielorientierung**
   Die Organisation dient als Instrument zur Erreichung von Unternehmenszielen

2. **Kontinuität**
   Die Organisation hat für einen definierten Zeitraum bestand

3. **Koordination**
   Die Organisation wird durch Beziehungen zwischen den Elementen der Organisation (Menschen, Sachmitteln und Informationen) geregelt

4. **Arbeitsteilung**
   Die Organisation ordnet den Elementen (Menschen, Sachmitteln, und Informationen) jeweils Teilaufgaben zu

Auf dem Weg hin zu einer bestehenden Aufbauorganisation, werden mehrere Schritte durchlaufen. Die zu erledigende Aufgabe wird in der *Aufgabenanalyse* auf verschiedene Teilaufgaben aufgegliedert. In der *Aufgabensynthese* werden zweckmäßige Aufgabenkomplexe gebildet. Diese Aufgabenkomplexe werden durch eine *Stellenzusammenfassung*, dem Begriff entsprechend, zu Stellen zusammengefasst. Diese Ansammlungen von Stellen, die auch als *Abteilung* bezeichnet werden, werden in Beziehung gesetzt und bilden abschließend die Gesamtstruktur, entsprechend die bestehende Aufbauorganisation. Die Ziele, die mit einer funktionierenden Aufbauorganisation verfolgt werden, teilen sich auf drei Bereiche auf:

1. **Stabilität**
   Eigenschaft, auf ähnliche Einwirkungen (unter vergleichbaren Randbedingungen) standardisiert zu reagieren

2. **Flexibilität**
   Eigenschaft, auf veränderte Umweltbedingungen in kurzer Zeit zu reagieren

3. **Effektivität**
   Eigenschaft, begrenzt zur Verfügung stehende Ressourcen möglichst gewinnbringend zu nutzen

Mit dem Versuch, alle drei Ziele zu verwirkliche, ist es möglich, auf die Variabilität der Umwelt bestmöglich zu reagieren [38].

3.1.1 Aufbau Firmenstruktur

Um diese Struktur verstehen zu können, wird im Folgenden auf den gesamten Aufbau eingegangen, die die Firma Schulz Systemtechnik GmbH zur Verfügung gestellt hat, um ihre Unternehmensontologie abbilden zu können. Wie in Kapitel 2.4 dargelegt, kann eine Ontologie, sofern die grafisch dargestellt werden soll, als Knoten bestehen, die wiederum Daten beinhalten und mit Kanten verbunden sind, um die Knoten miteinander in Beziehungen zu setzen. Diese Art der Darstellung kann in 3.1 eingesehen werden, auf die im Folgenden näher eingegangen wird. Dabei soll auf die internen Abhängigkeiten eingegangen werden, die innerhalb der Ontologie des Unternehmens herrschen und auf Profile, die die Kernaspekte der Ontologie des Unternehmens darstellen, von der aus die Ontologie betrachtet werden

kann. Als zentrales Element sind die Projekte zu sehen, die innerhalb der Firma ablaufen und abgearbeitet werden. Dabei definiert sich ein Projekt, als eine zeitlich befristete, relativ innovative und risikobehaftete Aufgabe von erheblicher Komplexität, die aufgrund ihrer Schwierigkeit meist ein gesondertes Projektmanagement erfordert. Weiterführend bestehen unternehmensinterne, sowie -externe Auftraggeber. Interne Projekt werden durch unternehmensinterne Auftraggeber initiiert und mittels unternehmenseigener Ressourcen (Human- und Sachkapital) abgewickelt. Bei externen Projekten erfolgt die Auftragserteilung durch einen Auftraggeber. Dieser bestimmt den Projektgegenstand auf Basis eines abzuschließenden Vertrages [17]. Ein Projekt besteht nicht einfach nur für sich, sondern ist durch diverse Informationen und Detaillierungsgrade ein Bestandteil der Ontologie, und damit des Unternehmens, um die Charakteristik des Unternehmens darzulegen. Mit den Informationen können die Beziehungen verstanden werden, mit denen das Projekt in der Firmenontologie verankert ist. Das sind sind zum einen Projektart und Projektstatus, die das Profil in seiner Rolle als zentrales Element weiter spezifizieren, in eine Art des Projektes und den Status, der Auskunft darüber gibt, inwiefern sich das konkrete Projekt noch in der Planung befindet, ob es abgelehnt, oder bereits abgeschlossen wurde. Die Definition eines Projektes spricht darüber hinaus von einem benötigten Projektmanagement. Dieses wird in der betrachteten Ontologie von einem Projektleiter dargestellt. Im Unterschied zu den Mitarbeitenrn, welche sich auf der Ebene der Fachkräfte bewegen, gliedern sich die Aufgaben der Projektleiter zum größten Teil in den Bereich des Managements ein. So muss die Führungskraft Fach-, Sach- und Generalistenwissen gleichermaßen beherrschen [37]. Projektleiter, wie auch alle weiteren angestellten Mitarbeiter eines Unternehmens, lassen sich Standorten zuweisen. Sofern benötigt, können mehrere Standorte wiederum zu einer Niederlassung zusammengefasst werden. Diese Niederlassungen bieten in der betrachteten Ontologie einen weiteren zentralen Gesichtspunkt, von dem aus das gesamte Unternehmen betrachtet werden kann. Mit Bezug auf die Aufbauorganisation, besteht ein Unternehmen aus einer Gruppe vom Mitarbeitern, die Stellen besetzen und damit die gesamte Organisation ausmachen. Hierbei kann es relevant sein, von einzelnen Mitarbeitern auf ihre konkrete Funktion innerhalb des Unternehmens zu schließen. Durch die Spezifikation des Profils des Mitarbeiters, auf seine Fähigkeiten hin, können Mitarbeiter gezielt auf Projekte zugeordnet werden, da die bereits erwähnten Projekte, mit ihren Teilprojekten, ebenfalls mit den Fachbereichen verbunden sind. Hier können Ressourcenzuteilungen effizient vorgenommen werden.

Die Aufteilung eines Proejektes in Teilprojekte ist nötig, um möglichst identische Kompetenzen in einem Teilproejekt zu bündeln und effiziente Arbeitsleisutngen zu produzieren. Dabei lässt sich jeweils ein Projekt in mehrere Teilprojekte aufteilen, einem Teilprojekt ist entsprechend ein Projekt zugeteilt, dem es entstammt. Zur Zielerfüllung der Teilprojekte, werden Technologien verwendet, die in der Ontologie einen weiteren Kernaspekt darstel-

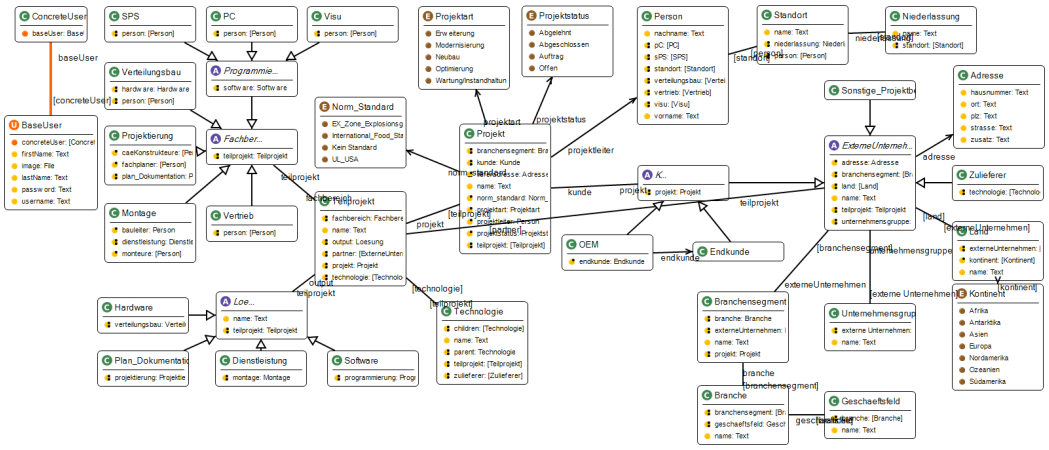


Abbildung 3.1: Die Ontologie der Firma Schulz Systemtechnik GmbH

len. Da mit den Technologien und den darin festgelegten Fähigkeiten die Teilprojekte und damit die gesamten Projekte durchgeführt werden können, bilden diese gleichzeitig die Kompetenzen des Unternehmens ab. Die allgemeine Definition dieser Kompetenzen, liegt in der Fähigkeit, in offenen, unüberschaubaren, komplexen, dynamischen und zuweilen chaotischen Situationen kreativ und selbstorganisiert zu handeln. Da in den in der Ontologie aufgeführten Kompetenzen auch das Kapital der Mitarbeiter eine große Rolle spielt, zählen auch diese Personen mitsamt ihrer Erfahrung und situativen Handlungsweisen. Technologien, über die ein Unternehmen verfügt, bilden demnach einen weiteren Aspekt, mit dem ein Unternehmen in konkrete Profile eingeteilt werden kann. Da es immer auch die bereits erwähnten Mitarbeitet sind, die die Kompetenzen hinter den Technologien ausmachen, kann die Verknüpfung zwischen den beiden Aspekten dafür sorgen, die dem Unternehmen zur Verfügung stehenden Ressourcen an Personal und Arbeitsmittel optimal aufeinander abzustimmen.

Auf der anderen Seite eines Projektes stehen die externen Unternehmen. Diese sind in einem definierten Branchensegment angesiedelt und kooperieren mit der betrachteten Ontologie über ein Teilprojekt, in dem sie mitwirken. Zur genaueren Spezifikation eines solchen externen Unternehmens, kann es in die Gruppen der Kunden, beziehungsweise der Zulieferer betrachtet werden. Zu beiden besteht jeweils eine Verbindung, wobei der Unterschied nur in der Richtung liegt, in der Daten und Informationen und ein bestimmtes Teilprojekt ausgetauscht werden.

3.1.2 Aufbau Ontologiemodell

Damit das Ziel einer automatisch generierten Anwendung anhand eines Ontologiemodells greifbarer wird, wurde die Entscheidung getroffen, einen mit DIME generierte Webapplikation, zu einem festgelegten DIME-Datenmodell, welches die Ontologie abbildet, zu erstellen. Diese soll als Prototyp dienen. Dieser erfüllt zeitgleich verschiedene Aufgaben. Die Erstellung des Prototypen sorgt dafür, dass Erfahrungen mit DIME gesammelt werden. Aus dem Prototypen heraus können *Pattern* bestimmt und extrahiert werden. Als letztes dient der Prototyp als Leitfaden für die anschließende automatisierte Generierung.

3.1.3 Erste Schritte in DIME

Zu Beginn wurden aus dem Ontologiemodell die *Rootelemente* (Elemente, die von keinen anderen Elementen direkt abhängig sind) bestimmt. Es wurden Kleingruppen gebildet, die sich jeweils mit der Umsetzung eines Rootelementes beschäftigt haben. Dabei wurde das Hauptaugenmerk auf die Einarbeitung und die direkte Umsetzung gelegt. Es ging in diesem Schritt explizit nicht darum, dass alle Gruppen möglichst ähnliche Ergebnisse haben. Dadurch, dass allen Gruppen selbst überlassen wurde, wie sie die Aufgabe lösen, gab es nach der Umsetzung der Rootelemente genau so viele verschiedene Lösungsansätze, wie es Gruppen gab. Aus diesen Lösungen wurden die besten Ideen festgehalten. Als nächstes lag der Fokus auf den Verbindungen zwischen den Rootelementen. Dieses mal wurde zwischen den Gruppen viel auf Zusammenarbeit gesetzt, um mögichst wenige Redundanzen in die Anwendung zu bringen. Nach der Fertigstellung eines ersten funktionsfähigen Prototyps wurden die Ergebnisse der Einzelgruppen und der Zusammenarbeit ausgewertet.

3.1.4 Pattern extrahieren

Bei der Auswertung der Ergebnisse wurde festgestellt, welche Umsetzungen bestimmter Probleme besonders gelungen waren. Es wurde zwischen verschiedenen Ergebnissen entschieden, wie gut diese als Vorlagen einzusetzen sind und ob diese Vorlagen für die automatische Codegenerierung ebenfalls sinnvoll einzusetzen sind. Zusätzlich wurden Anwendungsfälle besprochen, die von der Projektgruppe nicht vollständig identifiziert worden sind. Für diese Fälle wurden ebenfalls Lösungs- und Umsetzungsansätze besprochen. Anhand dieser Liste an Ansätzen und Lösungen wurden die Pattern erarbeitet, die im Kapitel 3.2 näher beschrieben werden. Diese Pattern sollten alle zu diesem Zeitpunkt auftretenden Muster und Standardfunktionen der Anwendung abdecken.

3.1.5 Umbenannt: Prototypen

Nachdem die Pattern erarbeitet wurden, wurde eine Teilgruppe der Projektgruppe damit beauftragt, einen neuen Prototypen zu erstellen. Die Hauptaufgabe des zweiten Prototyps

war die Überprüfung der Anwendbarkeit der ausgearbeiteten Pattern. Es sollte sowohl festgestellt werden, ob die erarbeiteten Pattern in der praktischen Umsetzung die besten Pattern waren, als auch ob alle Funktionen der Anwendung abdeckt wurden. Zusätzlich sollte für den Fall, dass die Pattern nicht ausreichend waren, die notwendigen Pattern sowohl erstellt als auch umgesetzt werden.
Als zweite Aufgabe stand die Überarbeitung der GUI im Mittelpunkt. Die GUI des ersten Prototyps war noch nicht auf Benutzerfreundlichkeit ausgelegt. Daher sollte in diesem Schritt eine GUI erstellt werden, bei der die Benutzerfreundlichkeit und die intuitive Benutzung für neue Anwender im Mittelpunkt stand. Aus der Neuausrichtung der GUI wurden ebenfalls GUI-Pattern extrahiert.

3.2 Pattern

Die Pattern werden in zwei Gruppen unterteilt. Eine Gruppe bilden die *Prozess Pattern* und die andere Gruppe besteht aus den *GUI Pattern* Während die Prozess Pattern die Logik der Anwendung bilden, wird in den GUI Pattern die Darstellung definiert. Im Folgenden werden diese näher erläutert.

3.2.1 Prozess Pattern

In DIME werden Prozesse definiert, um in der Anwendung Daten hinzuzufügen, zu bearbeiten oder zu löschen. In der Anwendung für die Schulz Systemtechnik GmbH werden einfache Pattern benötigt. Diese sind das Erstellen, das Bearbeiten, das Zuweisen und das Löschen, sowie das Laden von Daten, das Aufrufen von Modalansichten und das Hinzufügen von Attributen.
Neben diesen einfachen Prozessen gibt es die Komplexen, die zum einen der Hauptprozess aus dem die Anwendung gestartet wird und die jeweiligen übergeordneten Erstellungs- und Bearbeitungsprozesse eines Rootelementes sind. Diese Pattern werden im Folgenden näher erläutert.

Primitive Pattern

Die primitiven Pattern umfassen die Prozesse, die von konkreten Typen erstellt werden und keine eigenen GUIs oder Prozesse in ihrem Prozess beinhalten.

Erstellen

Wird ein neues Objekt erstellt, müssen dem Prozess die benötigten Attribute übergeben werden. In Abbildung 3.2 ist das beispielhaft im StartSIB zu sehen. Dort werden beim Erstellen einer Person der Vor- und Nachname als Text übergeben. Dann wird mit dem CreateSIB das eigentliche Objekt erzeugt, welches im EndSIB zurückgegeben wird. So ist

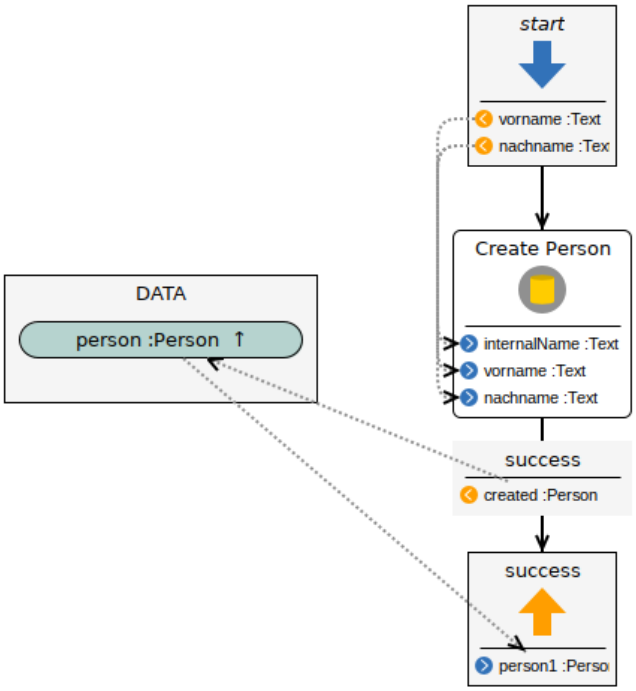


Abbildung 3.2: Beispiel für das Erstellen-Pattern: Erstellen einer Person

in dem Beispiel in Abbildung 3.2 die Person mit dem CreateSIB entstanden und wird mit dem EndSIB zurückgegeben.

Attribut bearbeiten

Das Pattern mit dem Objekte bearbeitet werden, wird immer aufgerufen, wenn Werte eines bestehenden Objektes verändert werden sollen. In Abbildung 3.3 ist das Bearbeiten einer Person als Beispiel für dieses Pattern dargestellt.
Dem StartSIB muss das alte zu bearbeitende Objekt und die neuen veränderten Attribute, in dem Beispiel **person1**, **vorname** und **nachname**, übergeben werden. Das zu bearbeitende Objekt wird in den DataContext geschrieben und die übergebenen Attribute werden in dem Objekt durch Complex-Pfeile überschrieben.

Löschen

Bei den Löschen-Pattern muss unterschieden werden, ob das Objekt in einer Liste eines anderen Typen vorkommt. Ist das nicht der Fall, kann es mit einem DeleteSIB zwischen dem Start- und EndSIB gelöscht werden. Ansonsten müssen die betreffenden Listen mit im StartSIB übergeben und mit dem RemoveFromListSIB aus der Liste gelöscht werden. Beispielhaft ist das in Abbildung 3.4 dargestellt.

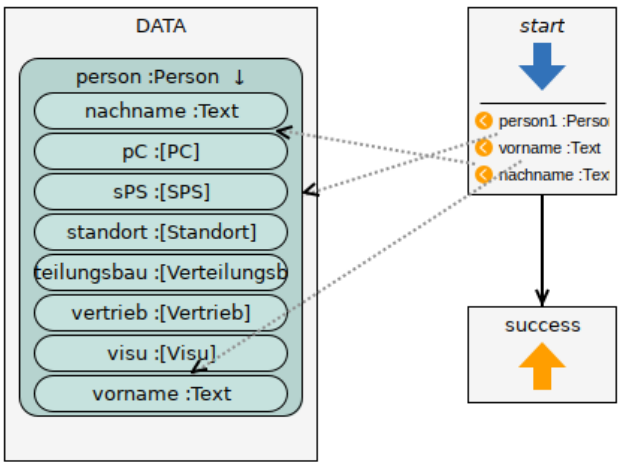


Abbildung 3.3: Beispiel für das Bearbeiten-Pattern: Bearbeiten einer Person

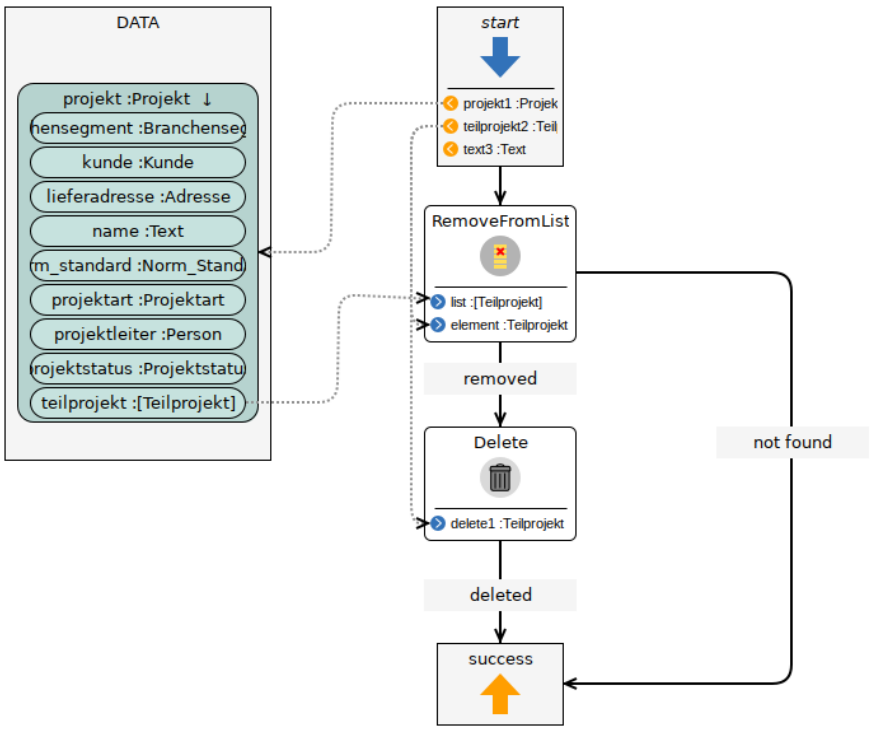


Abbildung 3.4: Beispiel für das Löschen-Pattern: Löschen eines Teilprojektes mit dem Löschen aus der Projektliste

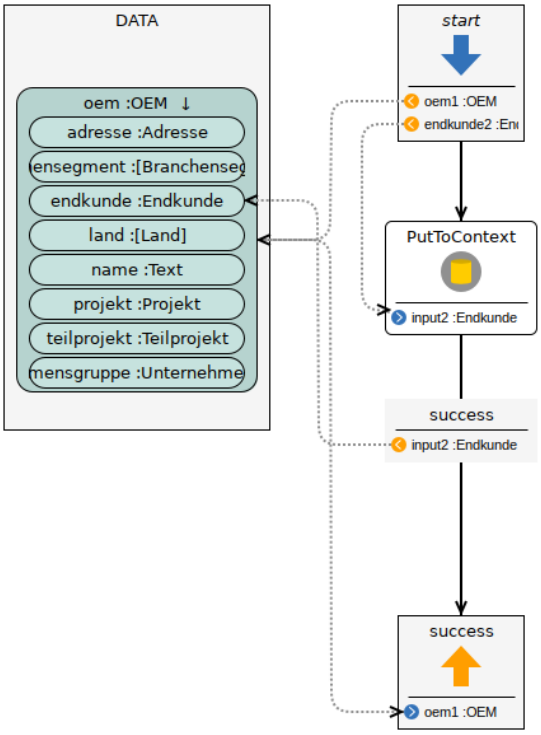


Abbildung 3.5: Beispiel für das Zuweisenpattern: Zuweisen eines Endkunden zu einem OEM

Zuweisen

Das Zuweisenpattern wird aufgerufen, wenn ein bestehendes Objekt einem anderen Objekt als Attribut hinzugefügt werden soll. Das geschieht beispielsweise bei der Zuweisung eines Endkunden zu einem OEM oder wenn ein Fachbereich einem Teilprojekt hinzugefügt wird. Dem Prozess werden das Objekt und das hinzuzufügende Objekt übergeben. Nachdem Letzteres durch ein PutToContextSIB hinzugefügt wurde, wird es im DataContext dem übergeordneten Objekt hinzugefügt. Dieses wird dann im EndSIB zurückgegeben. Der Prozess sieht dann wie in Abbildung 3.5 für das Zuweisen eines Endkunden zu einem OEM aus.

Laden

Das Laden-Pattern wird benötigt, wenn ein übergeordneter, komplexer Prozess Daten aus der Datenbank benötigt. Der Prozess besteht nach dem StartSIB aus den jeweiligen RetrieveOfTypeSIBs, die alle Objekte eines bestimmten Typen aus der Datenbank laden und endet mit der Übergabe der geholten Objekte.
Als Beispiel ist in Abbildung 3.6 dieser Ablauf zu sehen, indem zunächst alle Personen und dann alle Standorte aus der Datenbank geholt und zurückgegeben werden.

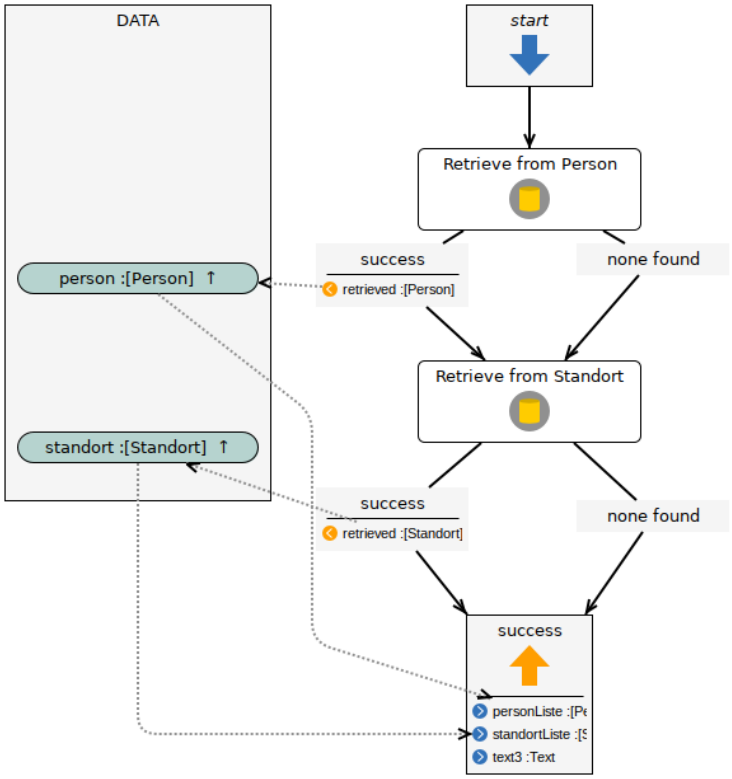


Abbildung 3.6: Beispiel für das Laden-Pattern: Laden der benötigten Daten für den übergeordneten Personenprozess

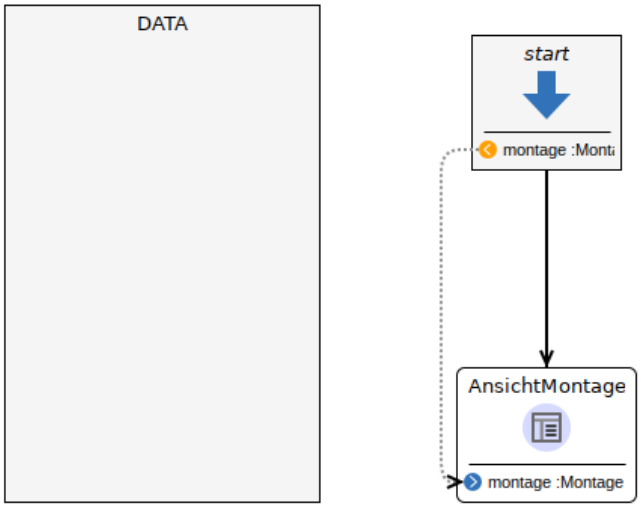


Abbildung 3.7: Beispiel für die Modalansicht

Modalansicht

Die Modalansicht wird aufgerufen, wenn in einer Ansichtstabelle ein anderes Objekt ebenfalls mit angezeigt werden soll. Aus dieser Tabelle wird der Modalprozess gestartet, der beispielsweise in Abbildung 3.7 für die Modalansicht der Montage zu sehen ist. Dieser Prozess besteht aus dem StartSIB, dem das anzuzeigende Objekt übergeben wird, und einer GUI, die die Modalansicht, eine Verwalten-GUI, aufruft (siehe Abschnitt 3.2.2).

Filtern

Das Filter-Pattern wird dann eingesetzt, wenn herausgefunden werden muss, welche Objekte als Attribut einem Objekt zugeordnet sind und welche nicht. Dadurch kann der Nutzer auswählen, welche weiteren Objekte dieser Liste hinzugefügt und gelöscht werden.
Ein Beispiel ist in Abbildung 3.8 dargestellt, in welchem über alle Standorte iteriert und für jeden Standort überprüft wird, ob dieser in der Liste des jeweiligen Personenobjektes enthalten ist. Dafür muss dem StartSIB die Liste aller Standorte und das Personen-Objekt übergeben werden, für den herausgefunden werden soll, welche Standorte diesem nicht zugewiesen wurden.
Allgemein muss also das Objekt, bei dem herausgefunden werden soll, welche Objekte von einem Attribut diesem nicht zugewiesen wurden und die Liste über alle Objekte von dem Typen des Attributen übergeben werden. Danach wird über die gesamte Liste, in dem Beispiel über die Standorte, iteriert. Mit dem ContainsSIB wird abgefragt, ob der Standort in dem übergebenen Personen-Objekt enthalten ist. Ist das der Fall wird das nächste Objekt untersucht. Ist es nicht enthalten, wird es mit dem PutComplexToContextSIB der Liste im DataContext hinzugefügt und das nächste Objekt wird untersucht.
Ist der IterateSIB am Ende der Liste, wird der Prozess über den exit-Branch und dem damit verbundenen EndSIB verlassen. Die Liste über alle nicht enthaltenen Standorte wird zurückgegeben.

Komplexe Pattern

Die komplexen Pattern sind die Pattern, die sich aus mehreren Prozessen und Pfaden zwischen diesen zusammensetzen. Die Prozesse, die von der Startseite aus in die jeweiligen Rootelemente gehen, sind beispielsweise solche.

Startprozess

Der Startprozess ist der erste Prozess, welcher nach dem Starten der Anwendung aufgerufen wird. In diesem wird die GUI mit den Rootelementen eingebettet. Diese erscheint als Navigationsleiste am oberen Rand der Anwendung.
Wie in Abbildung 3.9 dargestellt, muss nach dem StartSIB die StartGUI aufgerufen werden.

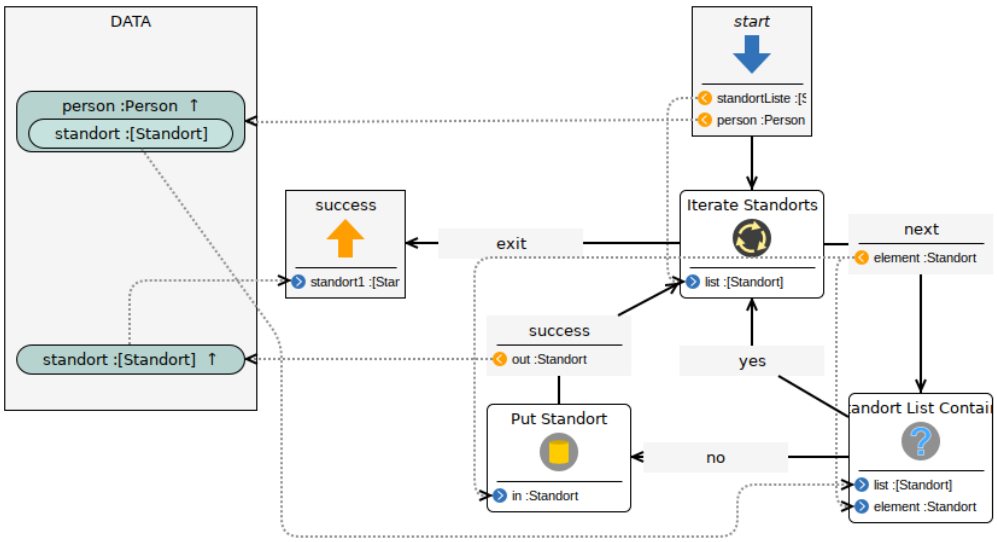


Abbildung 3.8: Beispiel für das Filter-Pattern: Alle Standorte, die nicht der Person zugeordnet wurden, werden zurückgegeben

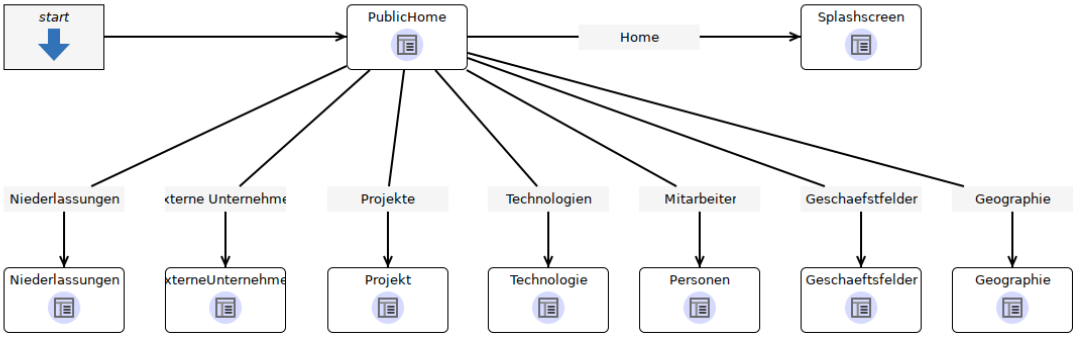


Abbildung 3.9: Beispiel für den Startprozess

Diese hat so viele Verzweigungen, wie es Rootelemente gibt und einen weiteren, um auf die Startseite (Splashscreen) wieder zurückzugelangen. Je nachdem welcher Knopf gedrückt wird, wird der jeweilige Branch und die dazugehörige GUI aufgerufen.

Rootprozess

Jedes Rootelement hat einen eigenen Hauptprozess, welcher die jeweiligen Aktionen in diesem Element verwaltet. Dieses Pattern fängt nach dem StartSIB mit dem Laden-Prozess an, welcher die benötigten Daten aus der Datenbank lädt. Danach wird die GUI für das Rootelement angezeigt und der Benutzer wählt zwischen den einzelnen Möglichkeiten aus. Das sind drei Verzweigungen: Ein neues Rootelement anlegen, ein bestehendes Rootelement bearbeiten oder löschen. Diese Branches rufen daraus die jeweiligen GUIs (beim Erstellen

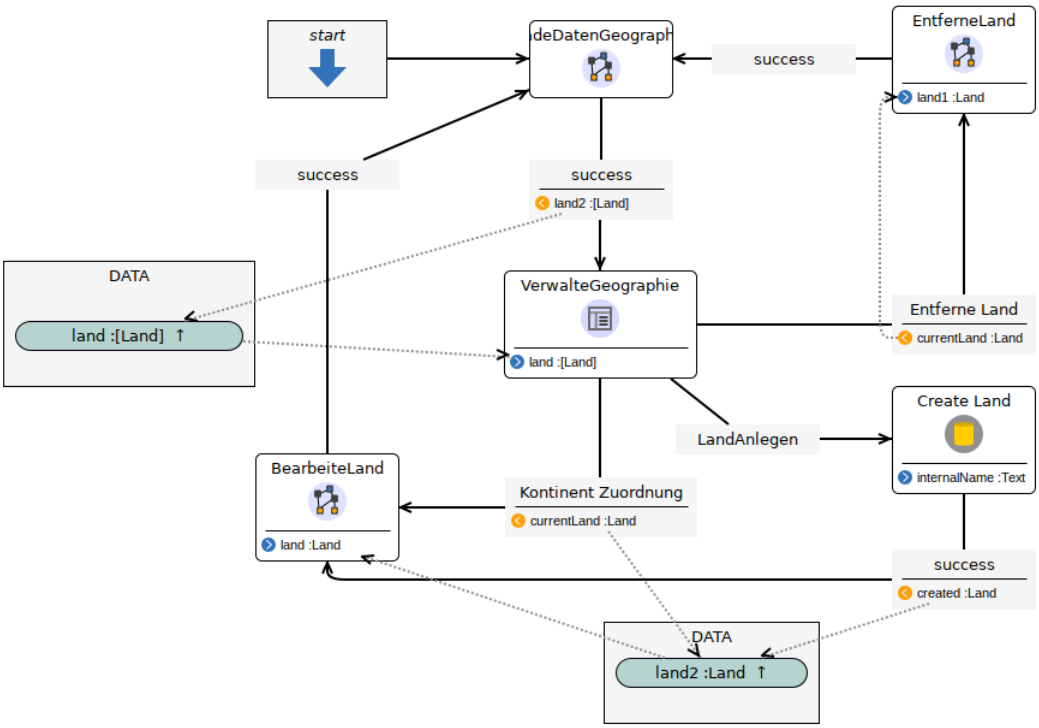


Abbildung 3.10: Beispiel für den Rootelementprozess Geographie

und Bearbeiten) und Prozesse (beim Löschen) auf.
Wird ein Rootelement neu angelegt, folgt nach dem Branch ein CreateSIB mit dem jeweiligen Element. Dadurch ist es möglich, Erstellen und Bearbeiten über dieselbe GUI zu starten. Das ist in Abbildung 3.10 dargestellt. Hier kann ein neues Land angelegt, bearbeitet und gelöscht werden. Beim Erstellen wird vorher ein leeres Objekt des jeweiligen Typs erstellt, beim Bearbeiten wird das Ausgewählte der GUI übergeben.
Ist ein Rootelement ein abstrakter Typ, muss für jeden konkreten Typen, der von diesem erbt, das Erstellen, Bearbeiten und Löschen in dem Prozess modelliert werden.

3.2.2 GUI Pattern

Zur einheitlichen Darstellung und zur möglichst intuitiven Nutzung ähnlicher Prozesse, Abläufe und Darstellungen innerhalb der Anwendung, gibt es neben den Prozess-Pattern auch GUI-Pattern. Diese Pattern erfüllen insbesondere ästhetische Aufgaben, die es dem Nutzer erleichtern sollen, die Anwendung zu benutzen.

Home-GUI

Die Home-GUI oder der sogenannte *Splashscreen* ist das erste, das der Nutzer sieht, wenn er die Anwendung startet. Die gesamte GUI ist mithilfe eines *Major/Minor Pattern* aufgeteilt. Die GUI besteht immer aus zwei Teilen: Im oberen Bereich befindet sich eine

Abbildung 3.11: Das Startmenü mit der Navigationsleiste oben

Navigationsleiste, die in jeder Situation sichtbar ist und als Major-GUI zählt. Der restliche Teil der GUI ist die Minor-GUI. Diese zeigt im Splashscreen das Logo der Schulz Systemtechnik GmbH an, ansonsten befinden sich dort interagierbaren Teile der GUI, welche die Funktionalität der Website bereitstellen. Wird ein auf eines der Elemente in der Navigationsleiste geklickt, so ersetzt die damit aufgerufene GUI die zurzeit dargestellte GUI (vor dem ersten Schritt also das Logo). Somit wird nicht klassisch zwischen Seiten gewechselt. Dadurch wird ermöglicht, dass das Grundgerüst der GUI überall gleich aussieht.

Elementpfade

Alle Elemente zeigen jederzeit an, auf welchem hierarchischem Pfad sie liegen. Das bedeutet, dass immer in einem Pfad angezeigt wird, welche Elemente in direkter Reihenfolge über dem aktuell angezeigtem Element liegen. Dies ist vergleichbar mit der Darstellung des Pfades in einem generischen Filebrowser. Jedes Element, das auf diesem Pfad angezeigt wird, ist anklickbar und bringt den Nutzer sofort in die entsprechende GUI des Elements.

(Einfache) Verwalten-GUIs

Für jedes einfache (Root)-Element wird eine *Verwalten-GUI* (vergleiche Abbildung 3.12) angelegt. Innerhalb dieser GUI wird der Name des Elements zusammen mit dem Wort „Verwalten“ als Überschrift angezeigt. Darunter werden alle Instanzen dieses Elements in einer

Tabelle angezeigt. Diese Tabelle besteht immer aus einer Spalte, die einen Namen enthält sowie einer Bearbeiten-Spalte, welche Interaktionselemente beinhaltet. Die Namensspalte ist sortierbar und durchsuchbar. Zu den Interaktionselementen gehört ein Bearbeiten-Knopf, welcher auf die „Bearbeiten-GUI“ für das entsprechende Element verweist. Der Löschenknopf löscht das Element aus der Tabelle und der Datenbank. Der Ansichtsknopf zeigt eine Detailansicht für das Element an. Zusätzlich haben manche Typen noch weitere Spalten, in denen gewisse wichtige Attribute oder andere zugehörige Typen angezeigt werden. Diese können auch einen Bearbeitenknopf besitzen, welcher direkt zur Bearbeiten-GUI für dieses Element führt.
Um einer Verwaltentabelle ein neues Element hinzuzufügen, wird der Anlegenknopf links oben benutzt. Dieser führt zu einer leeren Bearbeiten-GUI.

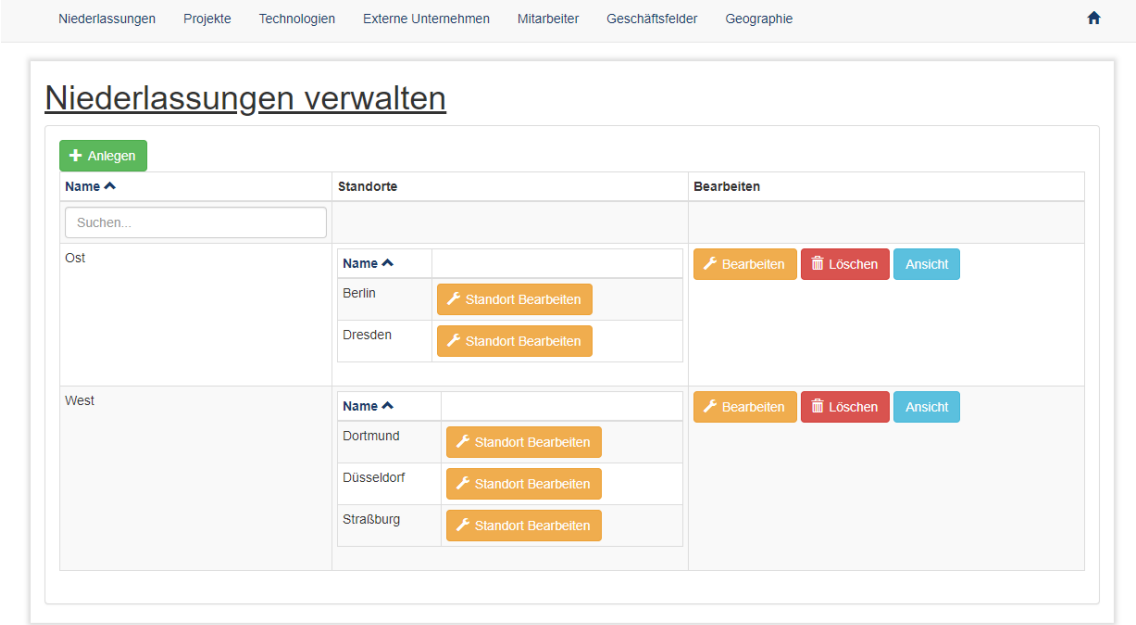


Abbildung 3.12: Beispiel für eine Verwalten-GUI

Abstrakte Verwalten-GUI

Wird eine GUI für ein abstraktes (Root)-Element erstellt, zeigt die GUI alle implementierenden Elemente an. Für jedes dieser implementierenden Elemente wird eine einfache Verwalten-GUI 3.2.2 innerhalb eines Panels erzeugt und diese daraufhin untereinander aufgelistet.

Bearbeiten-GUI

Die Bearbeiten GUI eines Elements zeigt untereinander alle Datentypen an, die beim Erstellen oder Bearbeiten geändert werden können. Für primitive Datentypen wird ein ein-

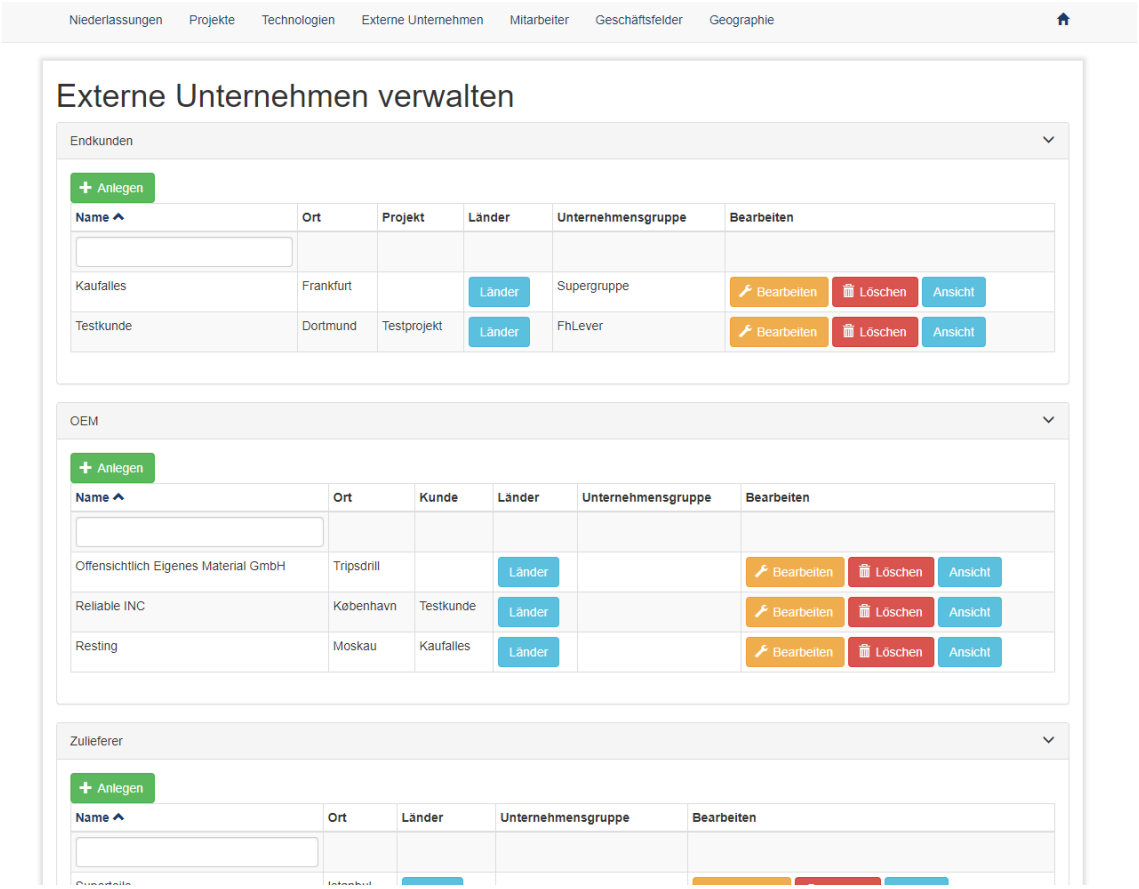


Abbildung 3.13: Beispiel für eine Verwalten-GUI eines abstrakten Typs

faches Formular angezeigt, in das die Eingaben direkt geschrieben werden können. Für komplexe Datentypen gibt es die Unterscheidung, ob es sich um einen Datentyp handelt, der nur für dieses Element angelegt und von diesem benutzt wird, oder ob es ein Datentyp ist, der aus einer Liste von Datentypen zugewiesen wird. Wird der Datentyp erstellt (wie z.B. eine Adresse zu einem Unternehmen), wird ein einfaches Formular für diesen Datentyp angezeigt. Wird der Datentyp zugewiesen, wird die Liste der auswählbaren Instanzen abhängig von ihrer Auswahlmöglichkeiten angezeigt. Soll nur eine Instanz ausgewählt werden, wird diese aus einer Liste mit einer Combobox gewählt. Handelt es sich um eine Mehrfachauswahl, dann werden entweder Checkboxen genutzt oder es wird die gesamte Liste aller entsprechenden Typen angezeigt, die dann einzeln hinzugefügt oder wieder entfernt werden können. Eine dritte Möglichkeit ist, dass ein komplexer Datentyp angelegt werden soll. Dafür wird innerhalb der Bearbeiten-GUI eine Verwalten-GUI des anzulegenden Datentyps eingefügt. Auch hier wird jede Möglichkeit, ein Attribut zu verändern, in ein eigenes Panel eingebettet.

Eine gewisse Besonderheit bilden abstrakte Datentypen, da bei ihnen Teile der Bearbeiten-GUIs wiederverwendet werden können. Dies hatte vor allem beim Erstellen der GUIs den Vorteil, das es sich erübrigte, gleiche GUIs mehrfach anzulegen. Außerdem konnten Än-

derungen deutlich schneller durchgeführt werden und die allgemeine Fehleranfälligkeiten reduziert werden.

3.3 Baumpattern/Baumtypen

Bei der Umsetzung der Ontologie von der Firma Schulz fiel auf, dass gewisse Teile hierarchisch verwaltet werden sollten. Da dieses Problem mit den bisherigen Pattern nicht lösbar war, wurde eine „Baumstruktur“ konstruiert. Diese Struktur ermöglicht es Objekte zu verwalten, die sich beliebig oft unterteilen lassen. Was diese Struktur an Funktionalität aufweisen muss, wird in Kapitel 3.3.1 erläutert. Die Pattern dieser Baumstruktur werden im Folgenden in zwei Abschnitte unterteilt. Einerseits gibt es Pattern für die Verwaltung der Baumstruktur (Kapitel 3.3.2 und 3.3.3) und andererseits Pattern für die Auswahl von Elementen, wenn man diese in einem anderen Objekt referenzieren möchte (Kapitel 3.3.4 und 3.3.5).

3.3.1 Definition / Regeln - Bäume im Kontext unserer Webapp / Ontologie

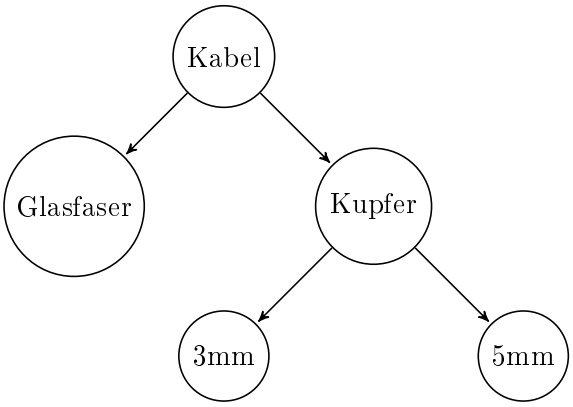


Abbildung 3.14: Beispiel einer Baumstruktur

Anhand der Abbildung 3.14 werden im Folgenden Eigenschaften erklärt, welche die Baumstruktur erhalten hat. Hierbei gilt ein Objekt als Elternknoten (parent), wenn er zugehörige Elemente in einer Unterkategorie besitzt. Diese werden Kinderknoten (children) genannt. So sind zum Beispiel „3mm“ und „5mm“ Kinder von dem Elternknoten „Kupfer“. Der Knotenpunkt auf der obersten Ebene wird auch Wurzel genannt.
Für das Verwalten der Baumstruktur sind folgende Eigenschaften nötig:

- Hinzufügen: Es muss möglich sein auf jeder Ebene ein neues Objekt einzufügen. So ist es zum Beispiel möglich, neben „Glasfaser“ und „Kupfer“ auch „Aluminium“ als Unterkategorie von „Kabel“ hinzuzufügen.
- Löschen: Wenn eine Oberkategorie aus dem Baum gelöscht wird, werden auch die Kinder gelöscht. Wenn im Beispiel „Kupfer“ entfernt wird, werden die Kinder „3mm“ und „5mm“ ebenfalls entfernt.

- Anzeigen: Die verschiedenen Ebenen der Baumstruktur sollen in der GUI erkennbar sein.

- Verwalten: In der Verwaltungsansicht eines Baumobjektes soll es einen Pfad geben, der es ermöglicht über die Baumebenen der jeweiligen Elternknoten zu navigieren. Wenn man sich in der Ansicht für den Knoten „5mm“ befindet, enthält der Pfad dementsprechend „Kupfer“ und „Kabel“.

Falls man Elemente einer Baumstruktur in einem anderen Type benutzt, um eine Beziehung zu bilden, müssen folgende Eigenschaften vorliegen:

- Auswählen bzw. Abwählen eines Objektes: Wenn man ein Objekt der Baumstruktur auswählt, so werden auch alle Kinder ausgewählt und alle Eltern bis zur Wurzel implizit auch. Die implizit ausgewählten Knoten werden dementsprechend graphisch anders dargestellt. Wählt man beispielsweise „3mm“ aus, so wird „3mm“ als ausgewählt angezeigt und „Kupfer“ und „Kabel“ als implizit ausgewählt. Wird jedoch „Kupfer“ ausgewählt, so werden „Kupfer“, „3mm“ und „5mm“ als ausgewählt angezeigt und nur „Kabel“ als implizit ausgewählt.

- Anzeigen: Die verschiedenen Ebenen der Baumstruktur sollen in der GUI erkennbar sein.

Aufgrund der verschachtelten neuen Struktur wäre es mit den bisherigen Pattern nicht möglich gewesen, diese Eigenschaften zu erfüllen. Daher wurden komplett neue GUI- und Process-Pattern erstellt, die sich selber beinhalten und als Input die Objekte der Unterkategorie erhalten. Mit solchen rekursiven Ansätzen ist es möglich, die Eigenschaften als Pattern abzubilden.

3.3.2 Schematische Darstellung der Baumverwaltung

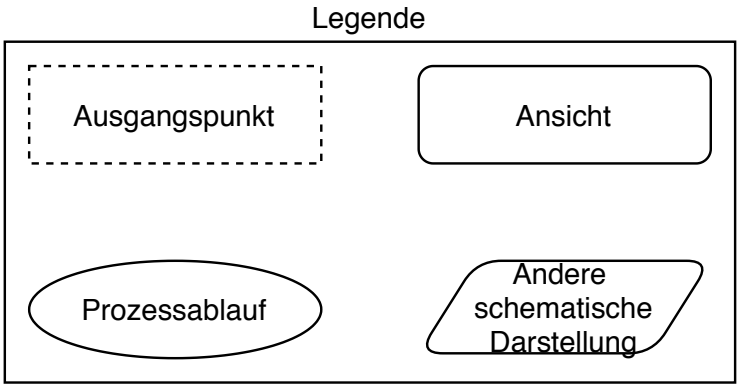


Abbildung 3.15: Legende für schematische Darstellungen

Durch die eingangs bereits erwähnten rekursiven Strukturen von als Baum gehandhabten Typen sind viele der in 3.2 behandelten Vorgehensweisen von Grund auf neu zu entwickeln. Im Folgenden wird daher die Konzeption der Verwaltung von Baumobjekten anhand schematischer Darstellungen näher betrachtet. Dabei werden folgende Konventionen (Abbildung 3.15) genutzt:

- Elliptisch umrandete Knoten beschreiben Prozess-Abläufe.
- Rechteckig umrandete Knoten beschreiben Ansichten.
- Gestrichelt rechteckig umrandete Knoten beschreiben unterschiedliche Ausgangspunkte.
- Mit einem Parallelogramm umrandete Knoten beschreiben Verweise auf eine schematische Darstellung.

Baumübersicht

Wie auch bei anderen konkreten Typen der Ontologie bedarf es verschiedener Darstellungen von Informationen zu einem Objekt bzw. zu einer Menge von Objekten eines Typs. Die Baumübersicht (siehe Abb. 3.16) ist hierbei das Pendant zur zuvor beschriebenen Verwalten-GUI und soll eine Zusammenfassung aller Elemente eines Baumtyps bieten. Auch hier gibt es dabei unterschiedliche Ausgangspunkte unter denen die Übersicht angefordert werden kann:

- Der Baumtyp ist ein Root-Typ, entsprechend müssen alle Elemente, die sich in der Datenbank befinden, dargestellt werden.
- Der Baumtyp kann innerhalb eines anderen Typen verwaltet werden, entsprechend sind alle zugewiesenen Elemente darzustellen.

Im Gegensatz zur Verwalten-GUI herkömmlicher Ontologie-Typen ist eine Listendarstellung der Objekte eines Baumtyps wenig sinnvoll, denn die Struktur aus Eltern und Kindern wäre dadurch weder erkennbar noch verwaltbar. Da sowohl die Rückgabe einer Anfrage an die Datenbank als Liste erfolgt, als auch die Beziehung zwischen Dime-Typen in diesem Fall nur als Liste dargestellt werden kann, ist hier also eine Vorverarbeitung der Daten notwendig.
Um schlussendlich eine Darstellung der Baumstruktur erreichen zu können, wird die Eigenschaft ausgenutzt, dass die Wurzelknoten eines Baums keinerlei Elternknoten haben. Aus der erhaltenen Liste von Objekten werden entsprechend zunächst die obersten Objekte des Baumes mittels des in 3.3.3 näher beleuchteten Prozesses *GetRootNodesProcess* herausgefiltert. Diese dienen im Folgenden als neue Eingabe für die Baumübersicht und werden der Übersichtlichkeit halber zunächst ohne deren Kinder dargestellt. Bei Bedarf kann jedoch

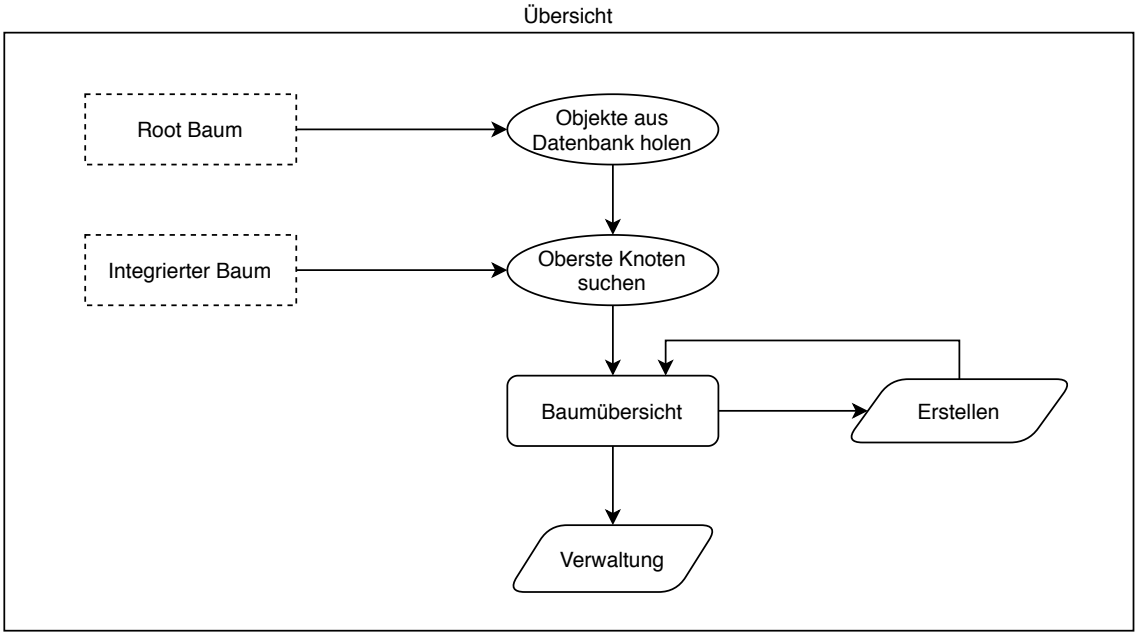


Abbildung 3.16: Schematische Darstellung der Baumübersicht

jeder Wurzelknoten "ausgeklappt" werden, wodurch alle direkten Kinder angezeigt werden können.
Da hier rekursiv immer wieder die selbe GUI mit den jeweiligen Kind-Objekten ineinander verwendet wird (siehe Abb. 3.17), kann dieses Vorgehen bis zu einer beliebigen Tiefe wiederholt werden, sodass im vollständig ausgeklappten Zustand alle Elemente sichtbar sind. So wird in der laufenden Web-Applikation eine Darstellung des Baums, wie sie beispielsweise in Abb. 3.18 zu sehen ist, möglich.
Jedes Element des Baums kann von der Baumübersicht aus verwaltet werden (Abb. 3.20). Darüber hinaus können weitere Wurzelknoten erstellt werden, was nachfolgend näher erläutert wird.

Baumerstellung

Der Prozessablauf der Baumerstellung (Abbildung 3.19) wird auf zwei verschiedene Arten benutzt. Zum einen wird er benötigt, um ein neues Wurzelelement anzulegen. Zum anderen wird er auch gebraucht, um ein neues Kind innerhalb eines bereits bestehenden Baumelements zu erzeugen. Insbesondere das Vorhandensein bzw. das Fehlen eines Elter unterscheidet die beiden Fälle. Beim Aufrufen des Prozesses wird entweder das richtige Elternelement übergeben oder gar keins. Dadurch kann unterschieden werden, in welchem der beiden Fälle man sich befindet.

Der Prozess beginnt damit, ein Baumelement zu erstellen und in die Datenbank zu schreiben. Falls der Nutzer sich im Verlauf der Erstellung dazu entschließt, doch kein neues

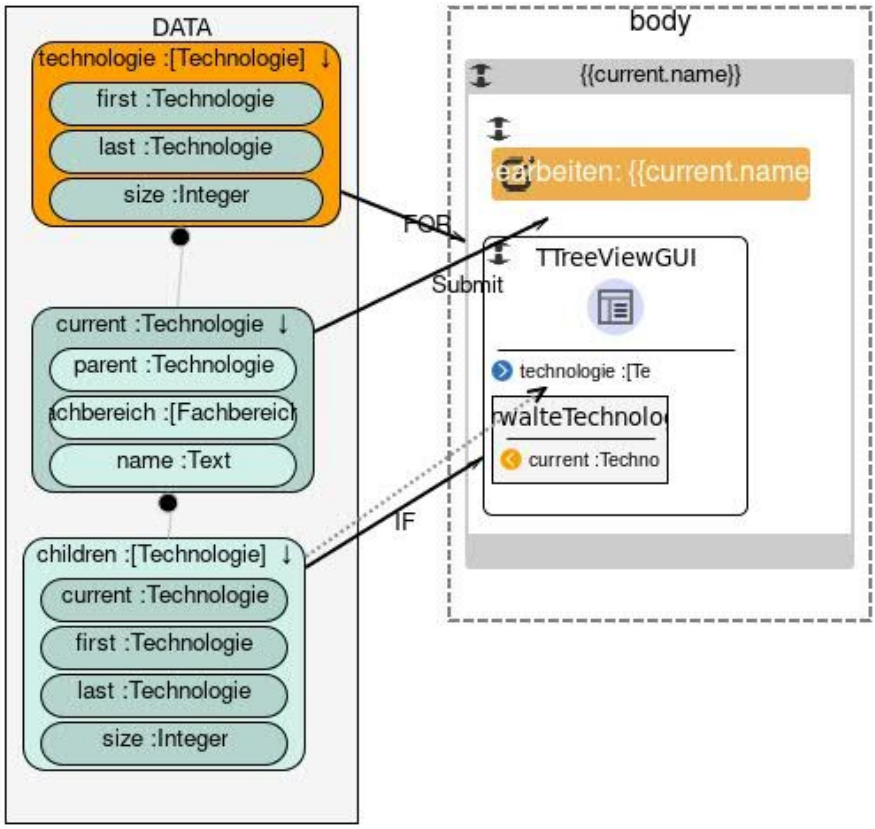


Abbildung 3.17: TreeViewGUI: Ruft sich selbst mit den Kindern wieder auf

Element anlegen zu wollen, wird das neu erstelle Objekt wieder gelöscht. Als nächstes landet der Nutzer in der Erstellen-GUI. Dort hat er die Möglichkeit, die einzelnen Attribute des Baumelements zu bearbeiten. Falls der Nutzer dann das neue Baumelement speichern möchte, geht der Prozessablauf weiter.

Falls keine neue Wurzel, sondern ein neues Kind, angelegt wurde, werden nun die ensprechenden Referenzen gesetzt. Als letztes werden verschiedene Objektreferenzen von dem neuen Kind und seinem Elter überprüft und möglicherweise geändert. Dies wird nur für die Typen durchgeführt, die in der Lage sind, einzelne Baumelemente auszuwählen; also diese, für die Prozesse der Baumauswahl (Kapitel 3.3.4) generiert werden. Die Baumauswahl arbeitet nach dem Grundsatz, dass ein Baumelement in einem anderen Typ enthalten ist, genau dann, wenn auch alle seine Kinder in diesem Typ enthalten sind. Wird nun das erste Kind für ein Baumelement erstellt und bekäme nicht direkt alle Referenzen seines Elter zugewiesen, wäre diese Ordnung verletzt. Genauso muss ein Baumelement seine Referenz auf andere Objekte verlieren, wenn es ein neues Kind erhält, welches diese Referenzen nicht hat.

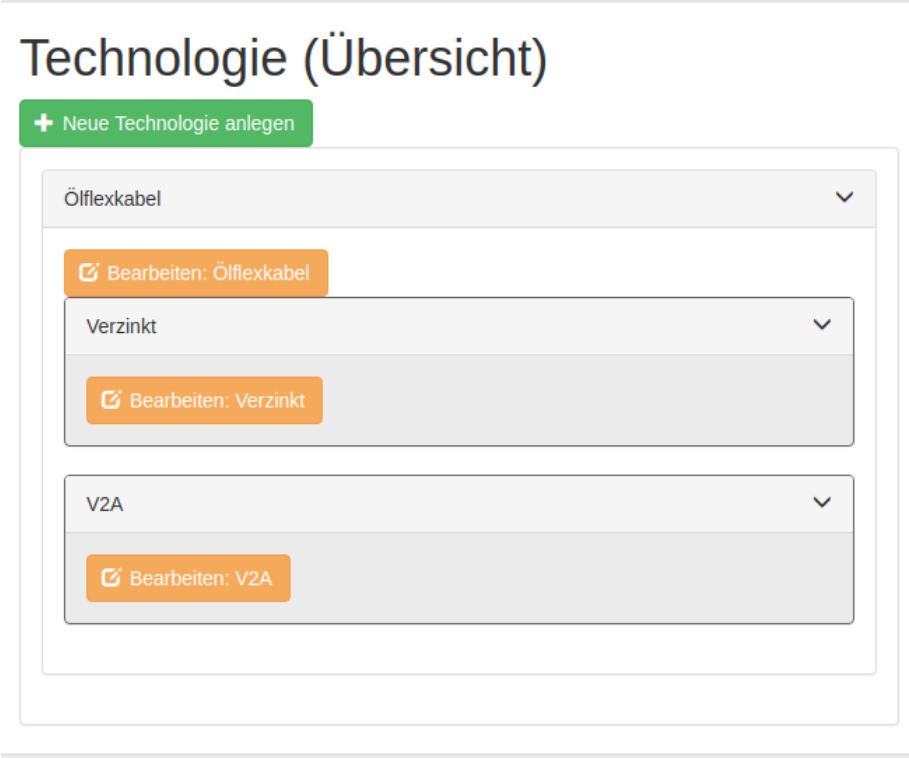


Abbildung 3.18: Baumübersicht in der laufenden Web Applikation

Baumverwaltung

Die Baumverwaltung bietet die Möglichkeit, einzelne Baumobjekte zu verwalten (s. Abb. 3.20):

- Dem Objekt können neue direkte Kinder hinzufügt werden.
- Die Bearbeiten-Ansicht des Objekts kann aufgerufen werden.
- Das Objekt kann gelöscht werden.
- Die Verwaltungs-Ansicht aller Eltern und Kinder kann aufgerufen werden.
- Die Baumübersicht kann aufgerufen werden.

Wie in Abb. 3.22 zu erkennen ist, setzt sich die Verwaltungs-Ansicht aus verschiedenen Komponenten zusammen.
An oberster Stelle ist die *PathViewGUI* zu dem verwalteten Baumobjekt positioniert. Diese bietet die Möglichkeit zur Baumübersicht, sowie zur Verwaltungs-Ansicht für jedes Elternobjekt zu gelangen. Dazu müssen zunächst sämtliche Eltern des Objekts gefunden werden, was durch einen Prozess *GetAllParentsProcess* erreicht wird. Da durch den Aufruf dieses Prozesses die Eltern in der falschen Reihenfolge zurückgegeben werden, wird die erhaltene Liste zusätzlich durch einen Prozess *InvertListProcess*, der in Abschnitt 3.3.3 im Detail

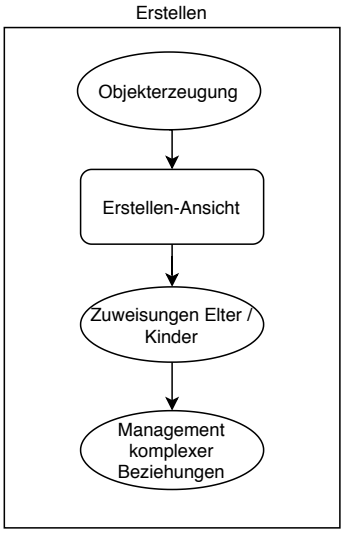


Abbildung 3.19: Schematische Darstellung der Baumerstellung

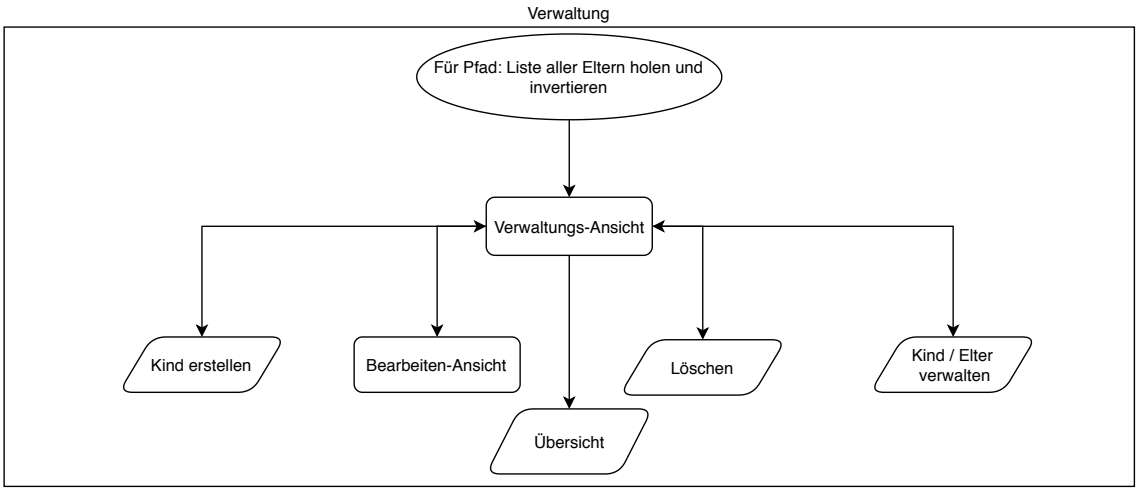


Abbildung 3.20: Schematische Darstellung der Baumveraltung

erklärt wird, invertiert. Durch diese Daten kann so ein Pfad aus Buttons erzeugt werden, die jeweils zur Verwaltungs-Ansicht des entsprechenden Objekts führen (Abb. 3.21).

Nach einer Überschrift folgt die bereits aus der Baumübersicht bekannte TreeViewGUI, die hier nun eine Baumdarstellung der Kinder ermöglicht. So kann von der Verwaltungs-Ansicht eines jeden Baumelements jedes darunter liegende Kind auf einfache Weise verwaltet werden. Wie zuvor beschrieben wurde, können hier außerdem weitere direkte Unterelemente des aktuell verwalteten Objekts erzeugt werden (siehe Abb. 3.19).

Nach der Anzeige der Unterelemente ist eine Ansicht aufzufinden, die mit der Detail-Ansicht anderer Typen vergleichbar ist und sowohl primitive als auch komplexe Beziehungen anschaulich darstellt. Beides kann über die Bearbeiten-Ansicht, die durch einen

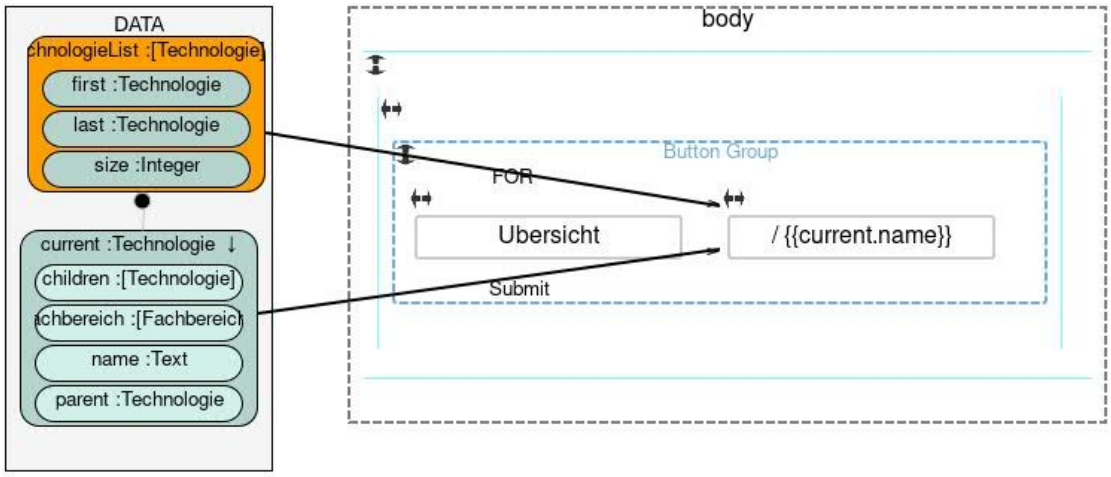


Abbildung 3.21: Pfad-Ansicht für ein Baumobjekt

entsprechenden Button auf der Verwaltungs-Ansicht erreichbar ist, nach den Vorgaben in der modellierten Ontologie angepasst werden.

Schlussendlich folgt die Option, den Löschprozess des Objekts anzustoßen, die im Folgenden separat beschrieben wird.

Löschen eines Baums

Wenn ein Baumelement gelöscht (Abbildung 3.23) werden soll, bedeutet das, dass auch alle seine Kinder gelöscht werden müssen. Dafür ist eine rekursive Vorgehensweise von Nöten, wobei die Rekursion immer auf alle Kinder aller bisher betroffenen Baumelemente angewandt werden muss.
Eine genauere Betrachtung dieser Prozessstruktur kann man in Kapitel 3.3.3 finden.

3.3.3 Pattern der Baumverwaltung

In diesem Abschnitt wird eine Auswahl an Pattern der Baumverwaltung im Detail betrachtet.

GetRoodNodesProcess

Der GetRootNodesProcess (siehe Abb. 3.24) ist, wie bereits im vorherigen Abschnitt beschrieben wurde, dafür zuständig, die Wurzelknoten innerhalb einer Liste an Baumobjekten zu finden und zurückzugeben. Dementsprechend erhält der Prozess als Eingabe eine Liste, über deren Elemente im Folgenden iteriert wird. Da die Wurzel einer Baumstruktur die Eigenschaft hat, keinen Elternknoten zu besitzen, wird jedes Objekt mittels eines weiteren Prozesses *HasParentProcess* auf diese Eigenschaft geprüft. Aufgrund der bidirektionalen

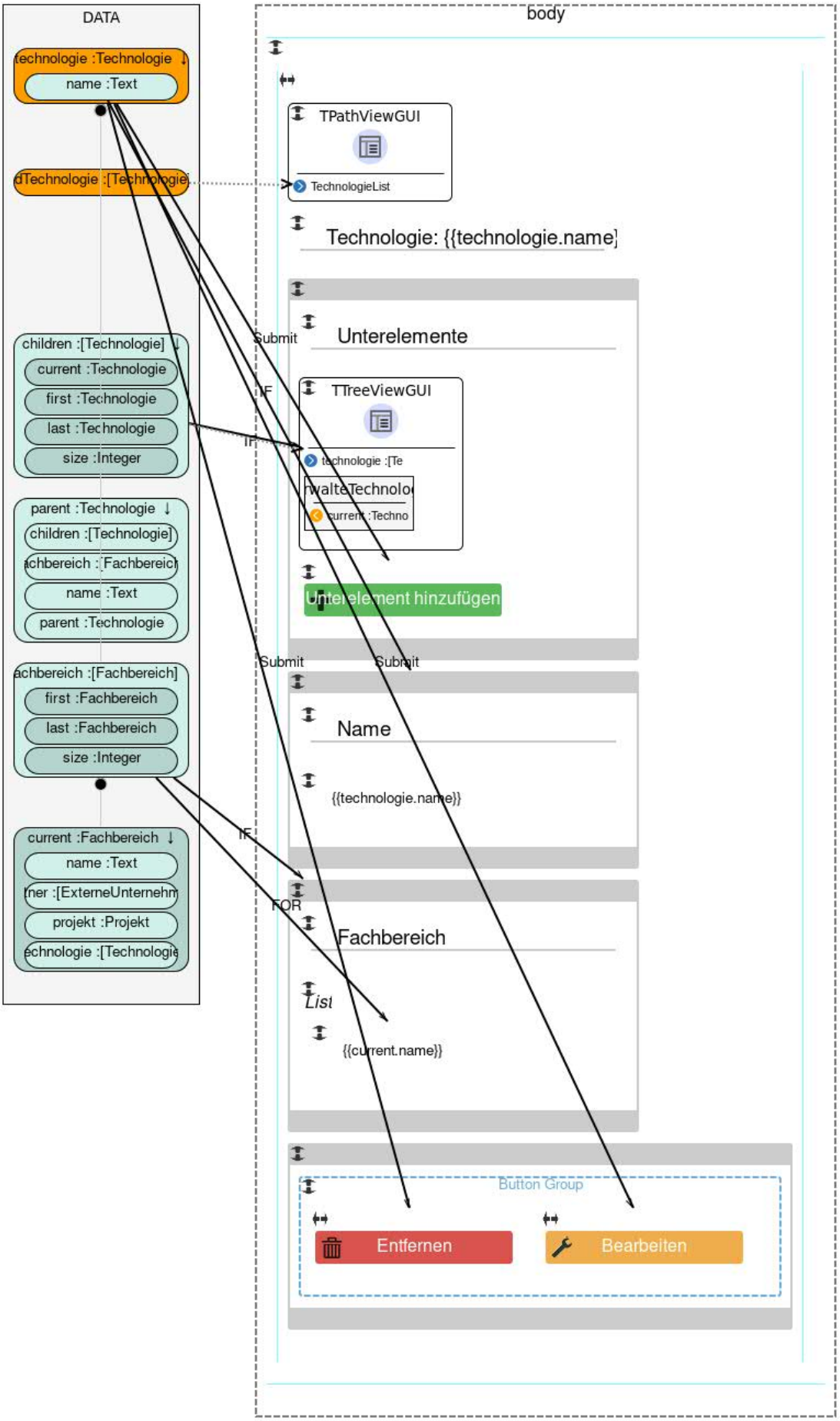


Abbildung 3.22: Verwaltungs-Ansicht für ein Baumobjekt

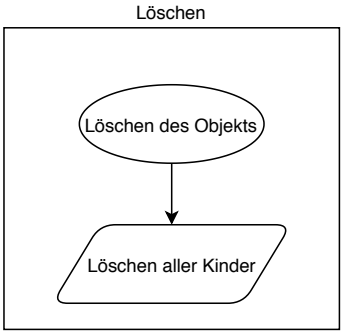


Abbildung 3.23: Schematische Darstellung des Löschens eines Baumknotens

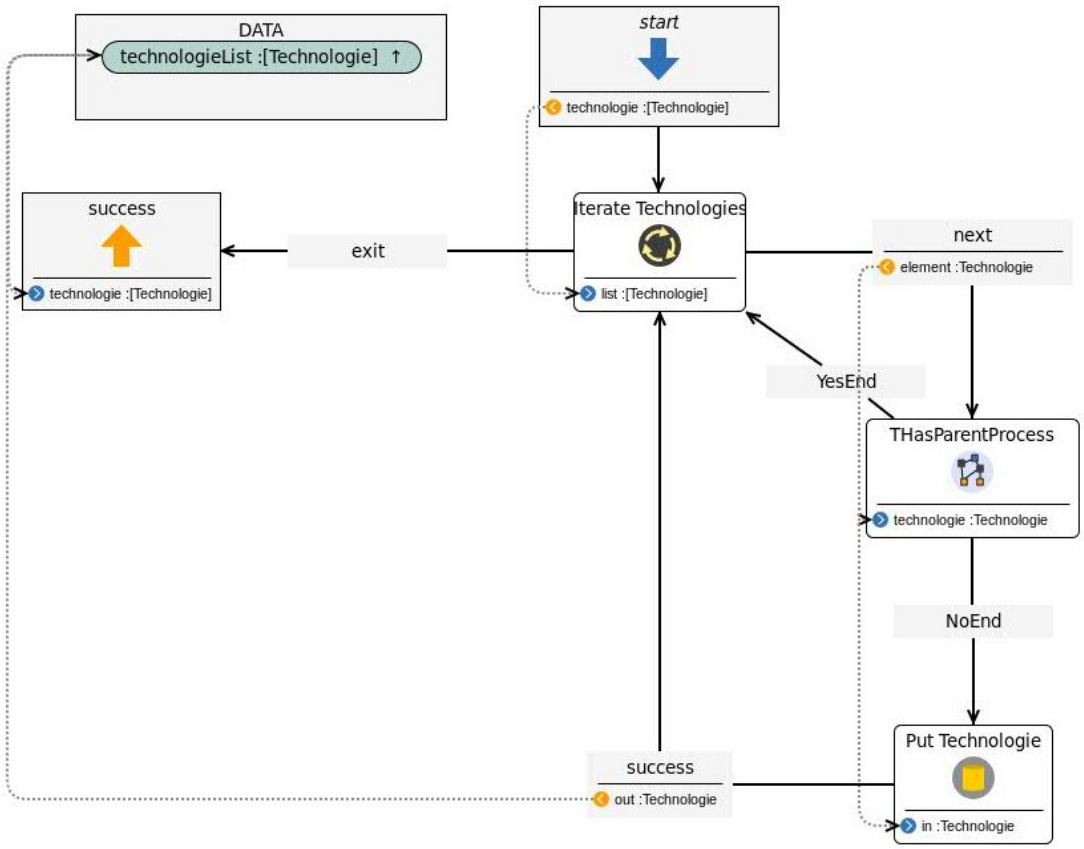


Abbildung 3.24: Der GetRootNodesProcess

Beziehung zwischen Baumobjekten kann dieser Test auf einfache Weise durchgeführt werden.

Hat ein Baumobjekt einen Elternknoten (YesEnd), so kann ohne weitere Schritte mit dem nächsten Element fortgefahren werden. Wird andernfalls kein Elternknoten gefunden (NoEnd), so wird das Objekt einer Listen-Variable hinzugefügt. Da im Kontext der Ontologie-Strukturen mehrere Wurzelknoten existieren können, wird die Liste in der beschriebenen Art und Weise vollständig durchlaufen. Ist dieser Schritt abgeschlossen, wird die Listen-Variable mit den Wurzelknoten zurückgegeben.

RemoveProcess und RemoveChildrenProcess

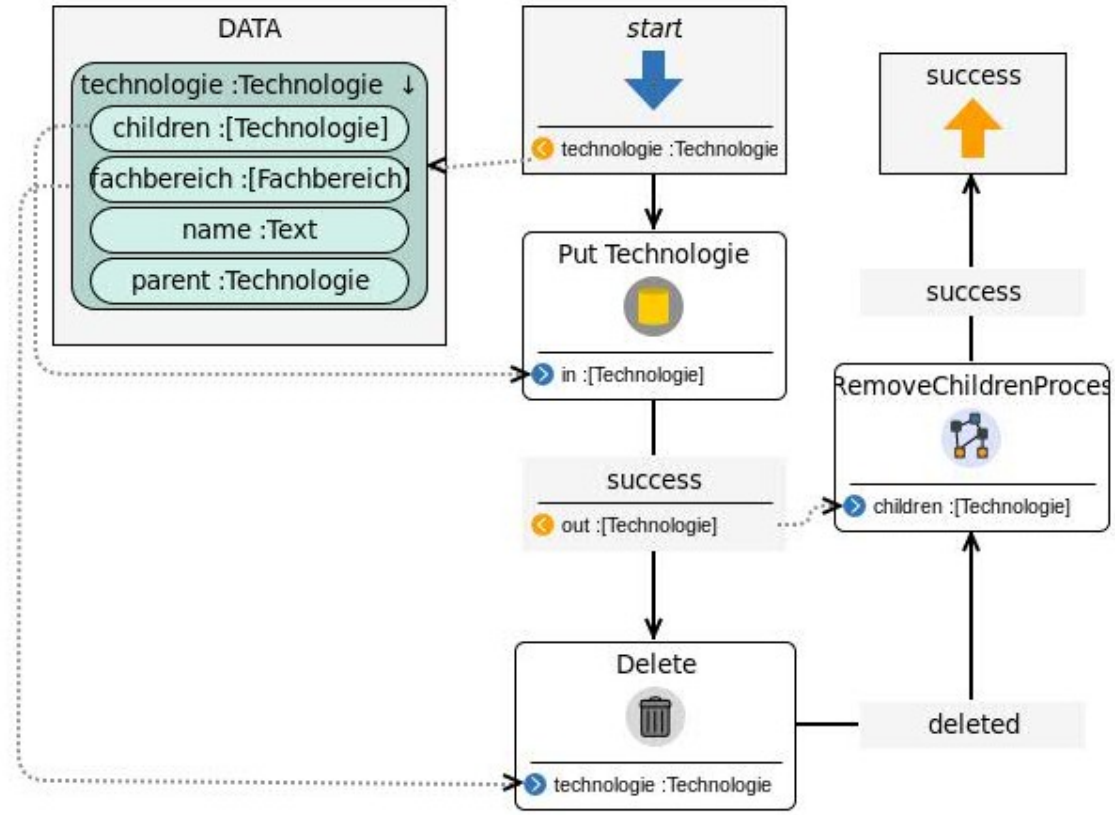


Abbildung 3.25: Der RemoveProcess

Der *RemoveProcess* (Abb. 3.25) führt die korrekte Löschung eines Baumelements durch. Zu beachten ist dabei die Vorgabe, dass nicht nur das Baumobjekt selber, sondern der gesamte Teilbaum mit sämtlichen Kindern gelöscht werden muss. Andernfalls würden die direkten Kinder des Objekts ungewollt zu neuen Wurzelknoten, da sie in der Datenbank verbleiben, aber keinen Elternknoten mehr besitzen.

Um dies zu erreichen wird zunächst eine Variable mit dem zu löschenden Element initialisiert. Durch einen PutComplexToContextSIB kann außerdem die Referenz auf die Liste der Kinder zwischengespeichert werden. Erst daraufhin wird das Element aus der Datenbank entfernt.

Im nächsten Schritt wird der Löschprozess *RemoveChildrenProcess* für die zuvor gemerkten Kinder angestoßen. Dieser ist in Abbildung 3.26 dargestellt. Wie dort zu erkennen ist, wird über alle Elemente der Liste an Kindern iteriert und für jedes Objekt jeweils der zuvor beschriebene RemoveProcess aufgerufen. Der RemoveProcess und der RemoveChildrenProcess rufen sich hier also so lange gegenseitig auf, bis beim aktuell zu löschenden Objekt keine Kinder mehr existieren. Durch die beschriebenen rekursiven Aufrufe kann der gesamte Teilbaum unabhängig von der Tiefe aus der Datenbank entfernt werden.

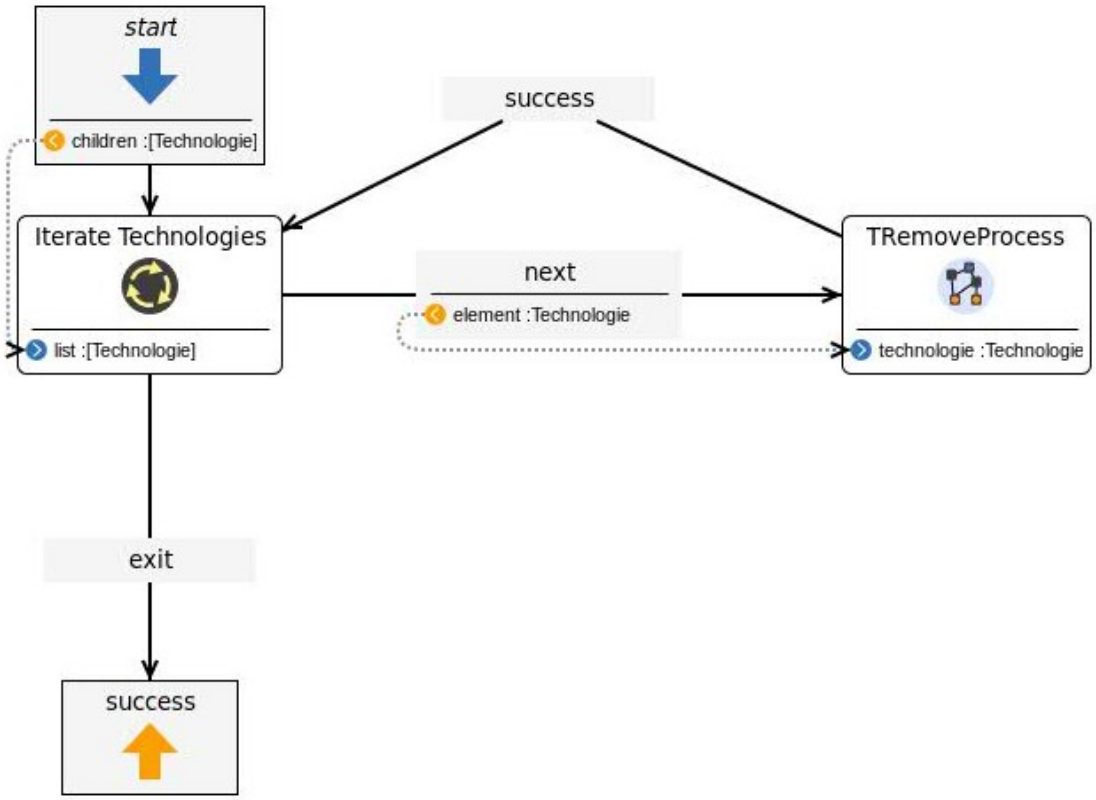


Abbildung 3.26: Der RemoveChildrenProcess

InvertListProcess

Der InvertListProcess (Abb. 3.27) invertiert eine Liste zur Datenvorverarbeitung für die Pfaddarstellung. Dazu wird die Eingabeliste in einer Variablen gespeichert und darüber iteriert, bis das letzte Element erreicht wird. Während dieses Vorgangs wird jeweils das aktuell betrachtete Element einzeln in einer zusätzlichen Variablen hinterlegt, sodass nach Abschluss dieses Vorgangs das letzte Element der Liste in der entsprechenden Variable zur Verfügung steht.

Für eine weitere Listen-Variable, die zur Speicherung der invertierten Liste angelegt ist, wird nun geprüft, ob das aktuelle letzte Element bereits vorhanden ist. Dies wird erst zu Ende des Prozesses relevant. Da die Elemente noch nicht vorhanden sind, bis alle Objekte der Eingabeliste übertragen wurden, wird hier beim *no*-Branch fortgefahren. Das Element wird nun der invertierten Liste hinzugefügt und aus der Eingabeliste entfernt, sodass bei Wiederholung dieses Vorhangs nun ein neues letztes Element zur Verfügung steht.

Sind sämtliche Objekte aus der Eingabeliste entfernt worden, wird das Iterieren sofort beendet. In der Variable für das letzte Element ist hier allerdings noch das aus der letzten Iteration vermerkte Objekt referenziert, sodass bei der Prüfung des ContainsSIBs dieses Mal mit dem *yes*-Branch fortgefahren wird. Da an diesem Punkt die gesamte Eingabeliste abgearbeitet wurde, kann nun die invertierte Liste zurückgegeben werden.

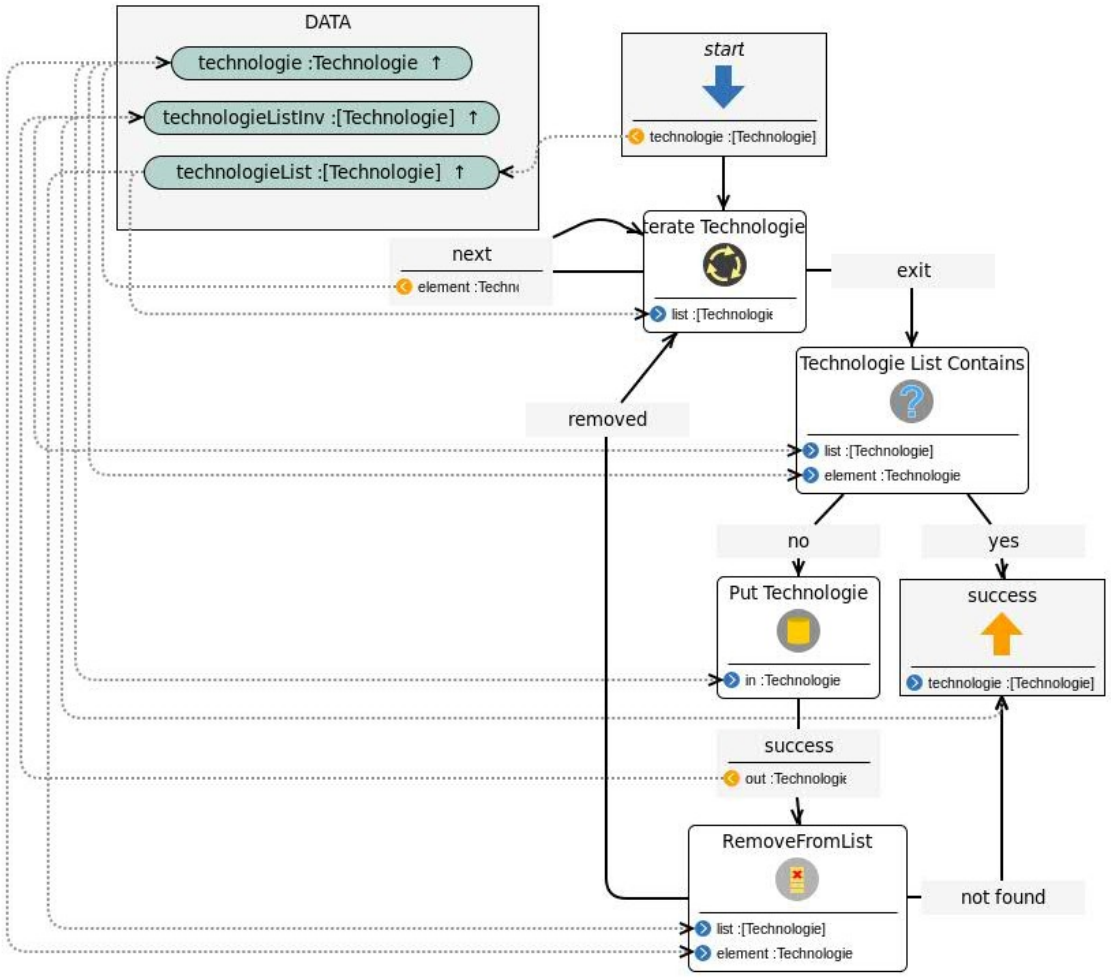


Abbildung 3.27: Der InvertListProcess

Dieses Vorgehen zur Invertierung kann entsprechend nur funktionieren, wenn kein Objekt mehrfach in der Eingabeliste vorhanden ist. Aufgrund der Umsetzung der Baumstruktur ist dies im Kontext der Ontologie Web-Applikation aber zu keinem Zeitpunkt der Fall.

3.3.4 Schematische Darstellung der Baumauswahl

Das Erstellen und Verwalten von Bäumen unterscheidet sich stark von den vergleichbaren Aufgaben der bereits vorgestellten anderen Typen. Das gleiche gilt auch für das Zuweisen von Bäumen. Dieses Unterkapitel beschäftigt sich mit der Funktionsweise der Modelle der Baumauswahl. Für eine einfache Einführung der Nutzung der Baumauswahl bietet sich der User-Guide an.

Im Folgenden werden einzelne Baumeinträge als „Baumelement“ bezeichnet. Der Typ, für den Baumelemente ausgewählt werden, wird „Besitzertyp“ genannt.

Jedes Mal, wenn ein Typ einen Baumtyp enthält, bei dem es möglich sein soll, Bäume auszuwählen, wird in der Bearbeiten-GUI des Besitzertypen ein Prozess eingebunden, der die komplette Funktionalität der Baumauswahl auf mehrere inneren Prozessen verteilt enthält. Diese Prozesse werden in diesem Kapitel in unterschiedliche Ebenen aufgeteilt und dann einzeln genauer betrachtet. Dabei besteht die erste Ebene aus der einfachen Grundstruktur der benutzten Prozesse. Die zweite Ebene beschreibt, welche abstrakten Aktionen in den einzelnen Schritten der ersten Ebene durchgeführt werden. Auf der dritten Ebene werden manche der komplexeren Teile von Ebene 2 noch genauer betrachtet.

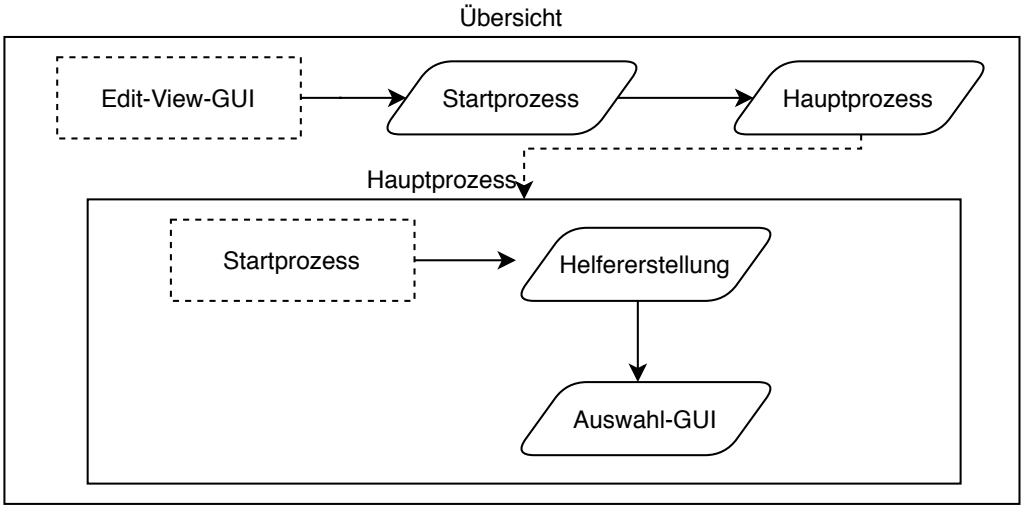


Abbildung 3.28: Schematische Darstellung des gesamten Prozessablaufs

Ebene 1

Auf der ersten Ebene werden nur zwei Prozesse betrachtet (siehe Abbildung 3.28). Dazu gehört zum einen der Startprozess und zum anderen der Hauptprozess. Der Startprozess ist ein Prozess-SIB, welcher in der Bearbeiten-GUI des Besitzertypen eingebunden wird. Von dort wird dann der Hauptprozess aufgerufen.
Der Hauptprozess enthält die relevante Funktionalität der Baumauswahl. Insbesondere enthält er die GUI, welche die Nutzerinteraktion ermöglicht. Bevor die GUI aufgerufen werden kann, müssen allerdings sogenannte Helferobjekte erstellt werden. Diese können als Wrapperobjekt für Baumobjekte verstanden werden, um die Baumobjekte mit Zusatzinformationen anzureichern.
Während die Baumverwaltung in den verschiedenen Iterationen der Webapp keine nennenswerten Änderungen seit der ersten Version erfahren hat, wurde die Baumverwaltung ab einem Zeitpunkt komplett überarbeitet. Der Grund dafür war die rekursive Implementierung der Bäume. Ab einer gewissen Größe, die schon bei relativ wenigen Elementen erreicht wurde, konnte der Nutzer bemerken, wie sich die gespeicherte Baumstruktur in der Webapp langsam nacheinander aufbaute. Dies lag unter anderem daran, dass sich

Baumelemente Informationen ihrer Eltern mithilfe einer Kette aus Prozess- und GUI-SIBs holen mussten, welche bei zunehmender Tiefe deutlich größer wurde.

Der Ansatz für eine Lösung dieses Problems bestand nun darin, zu verhindern, dass Baumelemente bestimmte Informationen über komplizierte Wege erhalten müssen. Dies wurde durch eine Speicherung dieser Informationen direkt in den Baumelementen erreicht. Es wurde also ein Tradeoff zwischen Laufzeit und Speicherplatz vorgenommen.

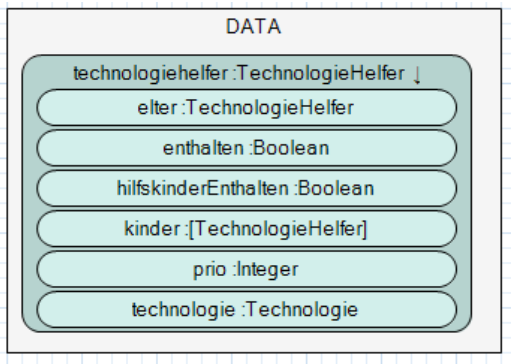


Abbildung 3.29: Helferobjekt im Datenkontext

Die bereits angesprochenen Wrapperobjekte sind die Umsetzung dieser Speicherung von zusätzlichen Informationen. Der Grund für einen neuen Typen war, dass das Datenmodell dafür vorgesehen ist, dass jeder Typ nur genau die Daten enthält, die in der Webapp auch bearbeitet und ausgelesen werden können und sollen; damit sind Hilfsdaten nicht in Typen vorgesehen. Da die Wrapperobjekte ausschließlich im Hintergrund existieren und nach Benutzung auch wieder gelöscht werden, eignen sie sich zum Speichern der Hilfsdaten. Grundsätzlich bietet DIME die Benutzung von sogenannten Transient-Objekten an, die sich wie normale Objekte verhalten, aber nie richtig in der Datenbank persistiert werden. Die Nutzung dieser wäre ideal für die Wrapperobjekte gewesen, allerdings ist es aufgrund bestimmter Funktionalitäten , die die GUI für die Baumauswahl erfüllen muss, nicht möglich gewesen, Transient-Objekte zu benutzen.
Wie man Abbildung 3.29 entnehmen kann, haben Baumhelfer die für Bäume typischen Attribute „elter“ und „kinder“, wobei letzteres eine Liste ist. Außerdem besitzen sie eine Referenz auf das Baumobjekt, zu dem sie gehören. Schließlich enthalten sie auch die drei Attribute, welche die Handhabung der Bäume verbessern. Die beiden Booleanattribute „enthalten“ und „hilfskinderEnthalten“ ermöglichen der GUI, sowohl sehr leicht als auch sehr schnell herauszufinden, ob ein bestimmtes Baumelement in dem anderen Typen bereits enthalten ist oder nicht, woraufhin dann das jeweils richtige angezeigt wird. Das führt dazu, dass nur ein nicht enthaltenes Element hinzugefügt oder ein bereits enhaltenes Element entfernt werden kann. Außerdem wird der Prozess des Entfernens ähnlich vereinfacht und

beschleunigt. Das Integerattribut „prio“ erlaubt es, eine einmalige Sortierung vorzunehmen, anstatt diese bei jedem Schritt neu wiederholen zu müssen.

Ebene 2

Die bereits angesprochene Struktur von Ebene 1 wird hier weiter verfeinert und etwas detaillierter beschrieben.

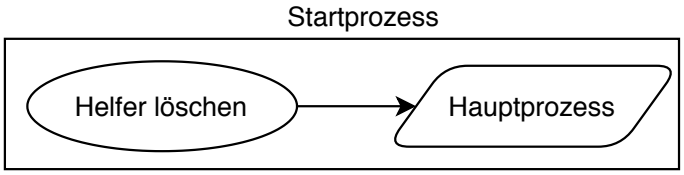


Abbildung 3.30: Schematische Darstellung des Startprozesses

Der Anfangsprozess (Abbildung 3.30) existiert, um die Einbindung der Baumauswahl in der Generierung zu generalisieren. Zusätzlich hat er die Funktion, bereits angelegte Helferobjekte zu löschen. Der Grund für das Löschen am Anfang der Benutzung der Baumauswahl ist, dass ein unangemessenes Verlassen der Baumauswahl eventuell nicht richtig abgefangen werden kann, wodurch verhindert werden würde, dass die Helferobjekte korrekt gelöscht werden. Da dies nach mehrfacher Durchführung dazu führen könnte, dass die Datenbank große Mengen an Helferobjekten enthält, wird stattdessen in Kauf genommen, dass die Datenbank fast immer eine eingeschränkte Menge von Helferobjekten enthält, was durch die vorher erwähnte Löschung am Anfang erreicht wird.

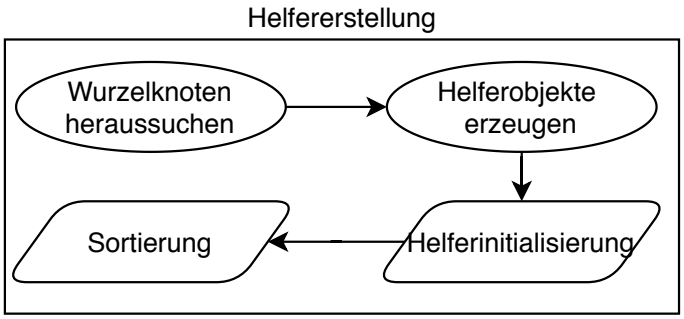


Abbildung 3.31: Schematische Darstellung der Helfererstellung

Ein wichtiger Aspekt der Baumauswahl ist die korrekte Erstellung der Helfer (Abbildung 3.31). Während in den früheren Iterationen komplexe rekursive Prozessstrukturen notwendig waren, um eine korrekte Verknüpfung zwischen Baumtypen und anderen Typen herzustellen, reicht jetzt eine simplere und iterative Lösung aus. Zuerst müssen aus allen passenden Baumtypen die Wurzeln gefiltert werden, da im weiteren Verlauf der Helfererstellung sowie bei der Nutzung der GUI mit Wurzellisten der Bäume gearbeitet wird. Als

nächstes wird für jeden Baum aus der Wurzelliste ein neuer Helferbaum erstellt, welcher isomorph zu den Ausgangsbäumen ist. Jedes Helferbaumelement besitzt eine Referenz auf das passende Baumelement. Ab jetzt wird nur noch mit der Wurzelliste von Helferbäumen gearbeitet.
In den letzten Schritten werden die Helferobjekte initialisiert und eine Sortierung auf ihnen durchgeführt. Diese Vorgänge werden in Ebene 3 nochmal genauer betrachtet.

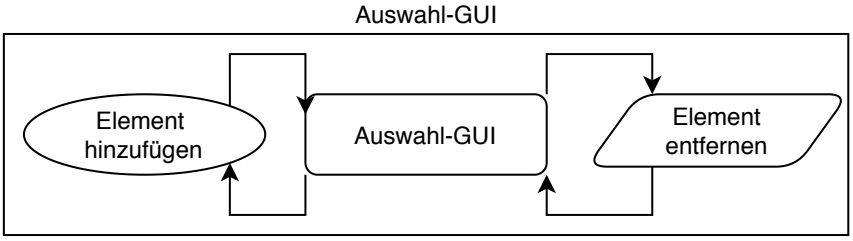


Abbildung 3.32: Schematische Darstellung der Auswahl-GUI

Wie im User-Guide beschrieben wurde, bietet die GUI (Abbildung 3.32) für die Baumauswahl vor allem zwei Aktionen für den Nutzer: Das Aus- und Abwählen von Baumelementen. Dabei kann beim Auswählen zwischen implizitem und normalen Auswählen unterschieden werden, was abhängig vom Auswahlzustand der Kinder ist. Funktionell ist das Auswählen allerdings in beiden Situationen identisch. Da beim Abwählen gewisse Randfälle abgefangen werden und die Aktion zu den Kindern und Eltern propagiert werden muss, ist das Abwählen in mehrere etwas komplexere Unterprozesse aufgeteilt worden. Deshalb wird es auch genauer in Ebene 3 vorgestellt.

Ebene 3

In diesem Abschnitt werden die erwähnten Prozessteile nochmal genauer vorgestellt und untersucht.

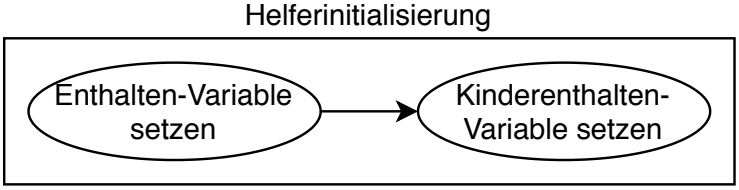


Abbildung 3.33: Schematische Darstellung der Helferinitialisierung

Es gibt zwei Setter-Prozesse, die die beiden enthalten-Booleanvariablen für jedes Helferobjekt festlegen (Abbildung 3.33). Zuerst wird bei jedem Helferobjekt überpüft, ob der aktuelle Typ in dem zugehörigen Baumelement enthalten ist. Falls dies der Fall sein soll-

te, wird „enthalten“ auf „true“ gesetzt. Nachdem dies für alle Helferobjekte durchgeführt wurde, wird für alle Helferobjekte die Booleanvariable „hilfskinderEnthalten“ auf „true“ gesetzt, wenn ein bestimmtes Hilfsbaumelement Kinder hat, bei denen eine der beiden Booleanvariablen den Wert „true“ hat.

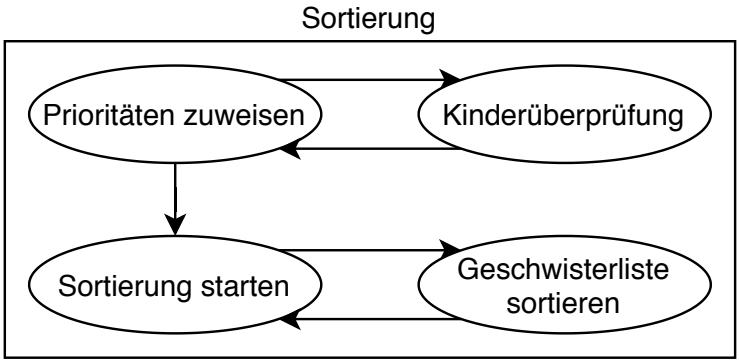


Abbildung 3.34: Schematische Darstellung der Sortierung

In der Baumauswahl werden alle angezeigten Bäume nach bestimmten Kriterien sortiert (Abbildung 3.34). Dabei sollen relevante Baumeinträge weiter oben stehen als andere. Für die Webapp wurde relevant so definiert, dass ein Baumelement, welches besonders oft in anderen Objekten des selben Auswahltyps ausgewählt wurde, weiter oben in der Auswahl erscheint. Das gilt auch für Baumelemente, deren Kinder besonders oft ausgewählt wurden. Die Sortierung wird dabei auf jeder Ebene eines Baumes vorgenommen. Das bedeutet, dass sowohl die Wurzelliste als auch die Kinderlisten von jedem Baumhelfer sortiert werden. Somit sind alle Geschwister untereinander korrekt sortiert.
Der erste Schritt der Sortierung besteht darin, jedem Helferobjekt eine Priorität zuzuweisen, nach der dieses sortiert und dann angezeigt werden soll. Diese Priorität wird in der Integervariable „prio“ gespeichert. Nachdem jedes Helferobjekt seine Priorität erhalten hat, wird die Sortierung für jedes Element der Wurzelliste angestoßen. Innerhalb eines Sortierungsvorgang wird die Sortierung auch rekursiv für alle Kinder gestartet.

Beim Abwählen (Abbildung 3.35) von einem Baumelement aus dem Auswahltyp können verschiedene Dinge passieren. Zuerst wird das abgewählte Baumelement aus dem Auswahltypen entfernt. Zusätzlich werden auch alle Kinder abgewählt. Zum Schluss wird überprüft. ob das abgewählte Element einen Elter hat. Ist dies der Fall, wird dieser entweder komplett abgewählt oder erhält den Status „implizit enthalten“, falls er noch andere Kinder hat, die enthalten sind. Dieser Vorgang wird ebenfalls für alle darüberliegenden Eltern wiederholt.

3.3.5 Pattern der Baumauswahl

In diesem Teil werden ein paar ausgewählte Prozesse genauer vorgestellt.

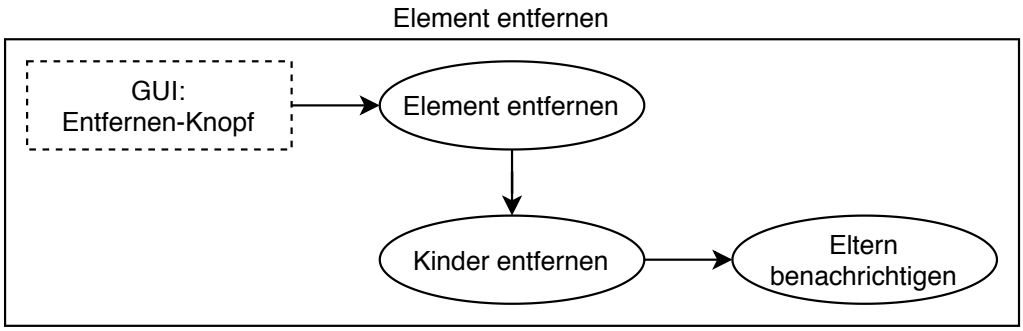


Abbildung 3.35: Schematische Darstellung der Baumelemententfernung

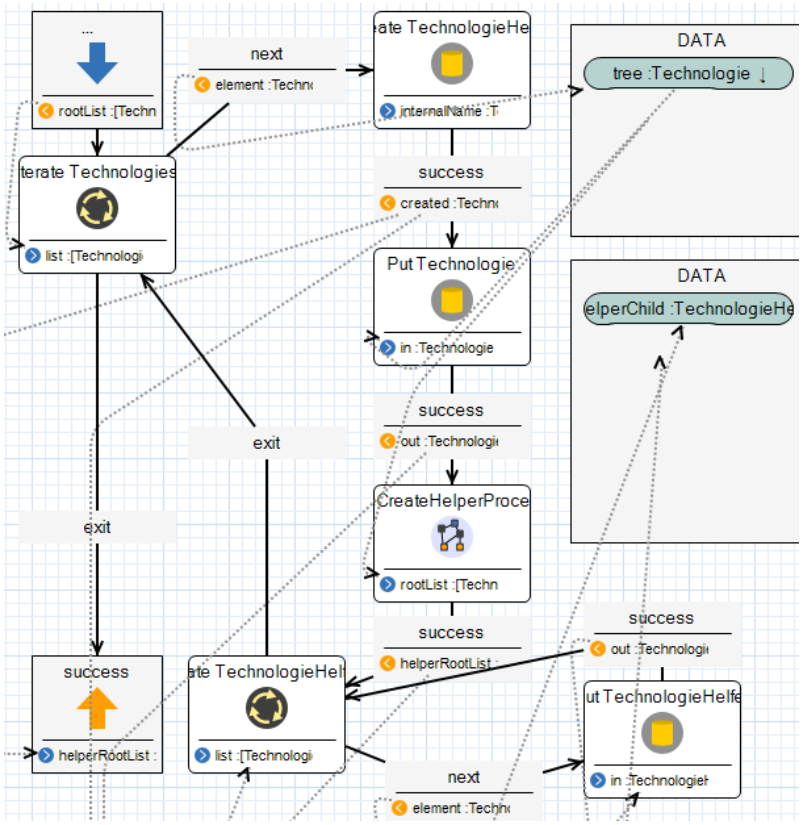


Abbildung 3.36: Der „CreateHelper“-Prozess

Der CreateHelper-Prozess

Im „*CreateHelper*“-Prozess (Abbildung 3.36) werden die Helferelemente erstellt. Als Eingabe erhält der Prozess die Wurzelliste der Baumelemente. Danach wird Folgendes für jede Wurzel durchgeführt: Es wird ein neues Helferobjekt erstellt und darin das aktuelle Baumelement gespeichert. Danach wird der „CreateHelper“-Prozess erneut aufgerufen, wobei die Eingabe diesmal nicht die Wurzelliste ist, sondern die Kinder des aktuellen Baumelements. Diese Art von Rekursion ist typisch für viele Baumprozesse. Nachdem in jedem Schritt das Ende der Rekursion erreicht wurde, wird der Helfer des aktuellen Baumelements mit

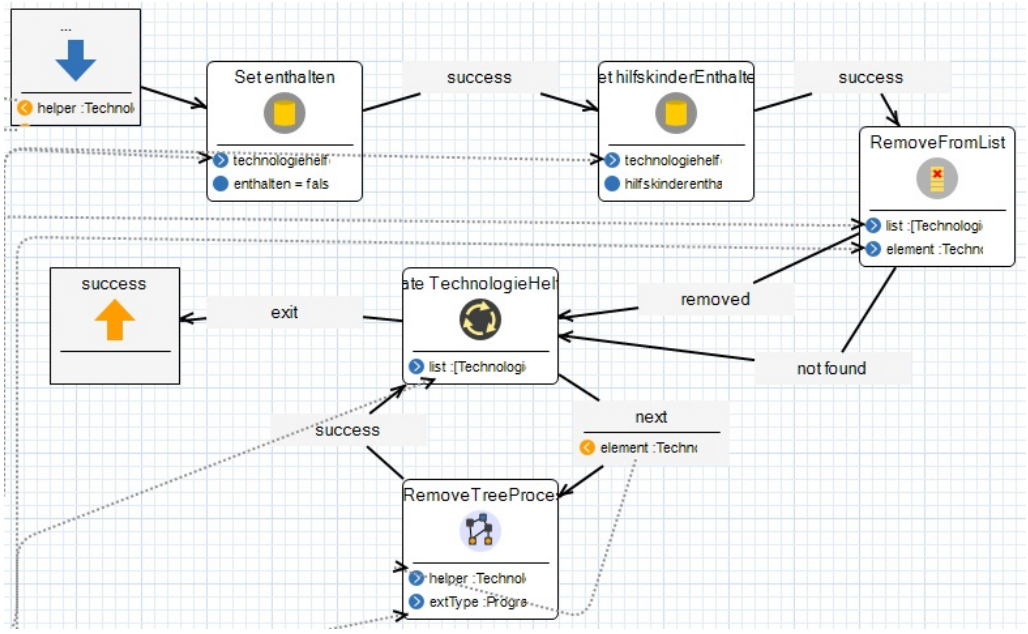


Abbildung 3.37: Der „RemoveTree“-Prozess

den Helfern seiner Kinder verknüpft, sodass der Helferbaum am Ende die gleiche Struktur hat wie der Ausgangsbaum. Die Rekursion erfolgt im Schritt davor, damit alle Element bereits vorhanden sind, um miteiander verknüpft werden zu können. Hiernach wird der nächste Baum aus der Eingabeliste abgearbeitet. Nachdem dies für alle Elemente durchgeführt wurde, ist der Prozess abgeschlossen und es wird eine Liste von Helferelementen zurückgegeben, die alle keinen Elter haben; dieser wird normalerweise nach der Rekursion hinzugefügt. Die Ausnahme hiervon ist der letzte Schritt, bei dem eine Wurzelliste von Helferelementen zurückgegeben wird. Diese Ausgabe enthält jedes einzelne Baumelement, dass sich in der Datenbank befunden hat, da die Eingabe alle Wurzeln enthält und durch die Rekursion allen Kinder ein Helferobjekt zugewiesen worden ist.

Der RemoveTree-Prozess

Im „*RemoveTree*“-Prozess (Abbildung 3.37) wird der ausgewählte Baum sowie alle seine Kinder aus dem Besitzertyp entfernt. Dafür werden zuerst in seinem Helferobjekt alle Enthalten-Variablen auf „false“ gesetzt. Danach wird das Baumelement aus der Liste des Besitzertyps entfernt. Zum Schluss werden die genannten Aktionen rekursiv für alle Kinder des Helferbaums ausgeführt, wodurch erreicht wird, dass alle Kinder auch aus dem Besitzertyp gelöscht werden.

Der SortSiblings-Prozess

Im „*SortSiblings*“-Prozess (Abbildung 3.38) wird eine Liste von Baumhelfern, die alle einen gemeinsamen Elter haben, nach ihrer Integervariable „prio“ absteigend sortiert. Dafür wird

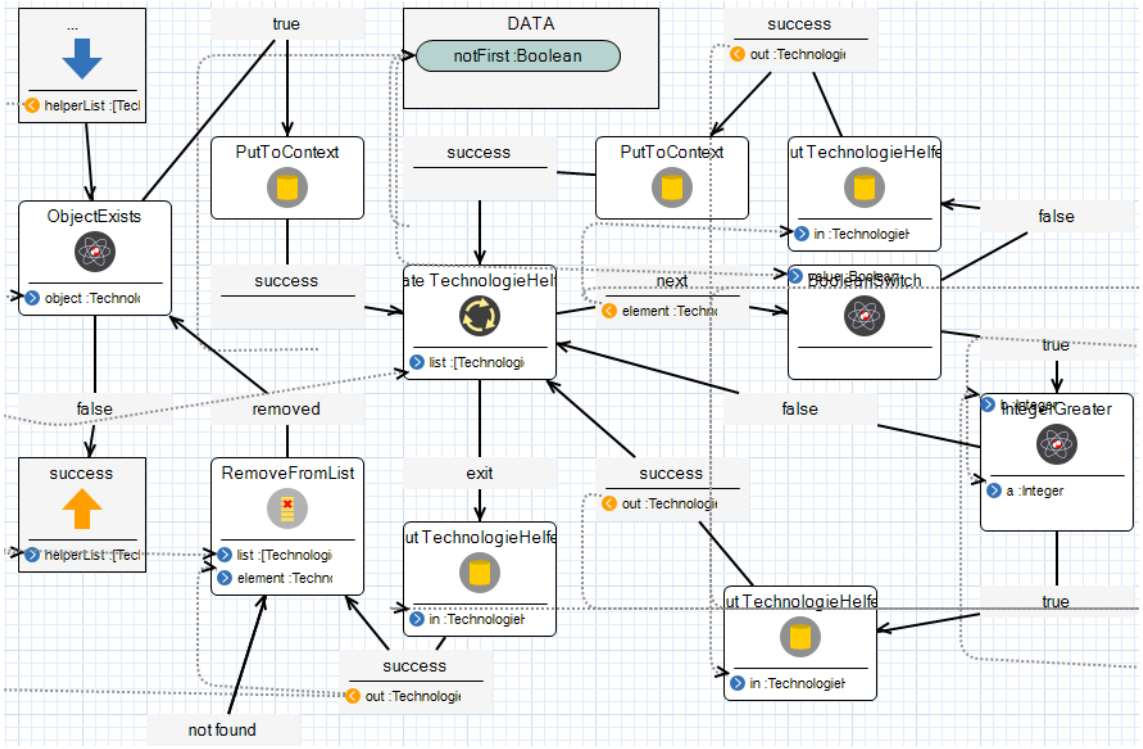


Abbildung 3.38: Der „SortSiblings“-Prozess

zuerst solange das erste Element aus der Eingabeliste entfernt, wie die Eingabeliste nicht leer ist. Da es zu Fehlern führen kann, über eine Liste zu iterieren, die während der Iteration verändert wird, wurde dies stattdessen mit einem ObjectExists-SIB realisiert, der das erste Element der Eingabeliste überprüft. Nachdem also sichergestellt wurde, dass die Liste nicht leer ist, kann über sie normal iteriert werden, um das Maximum zu finden. Bei jedem Element, dass so überprüft wird, kann einer von zwei Fällen eintreten. Der erste Fall tritt ein, wenn das aktuelle Element das erste der aktuellen Iteration ist. Dann wird es ohne weitere Überprüfungen zum Maximum gemacht. Bei jedem anderen Element tritt der zweite Fall ein: Das aktuelle Element wird dem mit Maximum verglichen, und letzteres dann mit dem aktuellen Element ersetzt, wenn dieses eine höhere Priorität hat. Diese Fallunterscheidung wird durch eine prozessinterne Booleanvariable realisiert, die vor jedem Beginn einer ersten Iteration auf „true“ und nach speichern des ersten Elements im Maximum auf „false “ gesetzt wird. Nach jeder Iteration wird das Maximum aus der Liste gelöscht und der Ausgabeliste hinzugefügt.

Die Auswahl-GUI

Als nächstes wird die Auswahl-GUI (Abbildung 3.39), mit der der Benutzer bei der Baumauswahl interagiert, vorgestellt. Die GUI besteht aus drei fast identischen Teilen. Diese Teile sind zwar identisch aufgebaut, unterscheiden sich aber sowohl in der Art, wie der

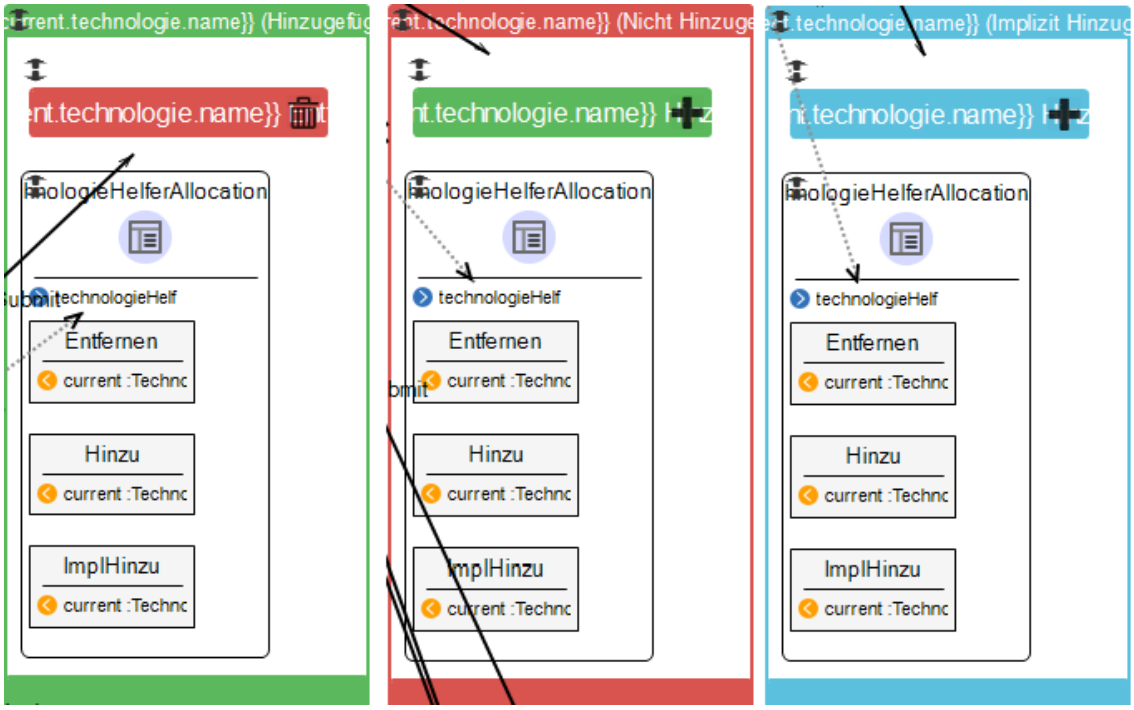


Abbildung 3.39: Die Auswahl-GUI

Nutzer sie wahrnimmt, als auch, wie das Programm auf eine Nutzereingabe reagiert. Die Auswahl von einer der drei GUI-Optionen hängt hierbei von dem aktuellen Wert der beiden „Enthalten“-Booleanvariablen ab. Da es nie vorkommen kann, dass ein Baumelement nicht ausgewählt ist und gleichzeitig Kinder davon ausgewählt sind, sind alle vier Möglichkeiten abgedeckt, die durch die beiden Booleanvariablen gegeben sein können.
Ein Teil der GUI besteht aus einem farbigen Panel, einem farbigen Knopf und einem Prozess, welcher die gleiche GUI für die Kinder des Baumelements anzeigt. Bei den drei Varianten unterscheiden sich jeweils die Farben sowie die Texte der Buttons, außerdem wird hier entschieden, ob man das Element hinzufügen oder abwählen kann, was durch einen Icon auf dem Button in Form eines Mülleimers oder eines Plus weiter verdeutlicht wird.

3.4 Transformation

Die im vorigen Abschnitt erarbeiteten Pattern zeigen, dass sich die Prozess- und GUI-Modelle für die verschiedenen Elemente des Ontologie-Modells sehr stark ähneln. Im Wesentlichen ist erkennbar, dass sich zwar die Elementtypen und deren Eigenschaften ändern, der grundsätzliche Aufbau der Modelle aber stets identisch bleibt. Gleiches gilt auch für die nachträglich entwickelten Pattern der Canvas-DSL (siehe Kapitel 5). Durch diese Erkenntnis soll im weiteren Verlauf ermöglicht werden, aus den jeweiligen Modellen automatisch eine funktionsfähige DIME-Applikation zu erzeugen, die sich an dem zuvor beschriebenen Prototypen für die Ontologie der Firma Schulz Systemtechnik GmbH, bzw. an dem in 5.1.2 vorgestellten Prototypen eines Canvas orientiert. Dazu ist eine entsprechende Transformation vonnöten, deren Konzeption im Folgenden näher besprochen wird.

3.4.1 Modell-zu-Modell Transformation in CINCO

Zur Erstellung einer DIME-Applikation ist das Anlegen mehrerer Modelle verschiedenen Typs notwendig: Neben dem Datenmodell sowie den Prozess- und GUI-Modellen wird alles in einem DAD-Modell zusammengeführt.
Auf ähnliche Weise werden auch das Ontologie-Modell und Canvas-Modelle in einem *OntologyTool*-Modell, welches in 4.1.5 näher beschrieben wird, zu einem Projekt vereint.
Bei der Erzeugung einer DIME-Applikation, ausgehend von einem solchen OntologyTool-Modell, kann hier also grundsätzlich von einer *Modell-zu-Modell Transformation* (M2M) gesprochen werden. Den Paradigmen des OTA und XMDD folgend, sollen keine Anpassungen am Zielmodell der M2M mehr notwendig sein. Voraussetzung dafür ist, dass bereits im Ausgangsmodell alle Informationen enthalten sind, die etwaige Konfigurationen des Endprodukts benötigen. Eine Umsetzung durch die entsprechend entwickelten DSLs wird in den Kapiteln 4 und 6 erläutert.

Da es sich bei DIME um ein CINCO-Produkt handelt, sind folglich alle dort enthaltenen Modelltypen CINCO-Modelltypen. Die Code-basierte Erzeugung eines Modells kann in CINCO mittels einer automatisch erzeugten *Factory* erreicht werden. Demzufolge steht für jeden zu erstellenden DIME-Modelltyp eine eigene Factory zur Verfügung. In Abbildung 3.40 wird das Vorgehen dargestellt:

- Das OntologyTool-Modell wird dem Transformator übergeben.
- Der Transformator erzeugt unter Nutzung der Modell-spezifischen Informationen mithilfe der Factories die entsprechenden Modelle für die DIME-Applikation.
- Dazu werden einige statische Inhalte benötigt, die bereits durch DIME zur Verfügung gestellt werden und somit in das Endprodukt kopiert werden können.

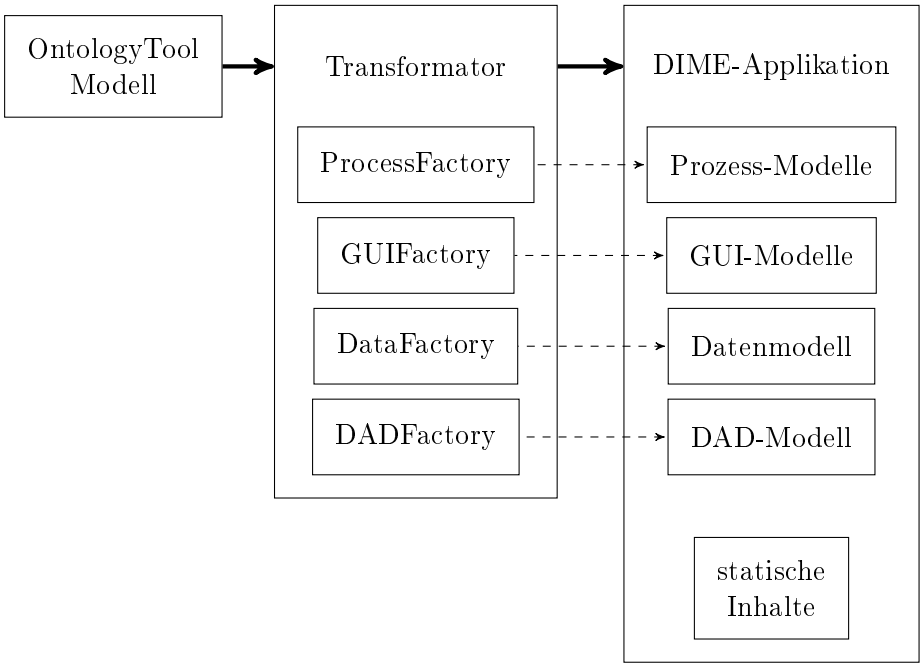


Abbildung 3.40: Erzeugung einer DIME-Applikation mithilfe von Factories

Die Implementierung eines Modell-Generators mithilfe der Factory-API verhält sich sehr ähnlich zur manuellen Erstellung eines Modells in einem CINCO-Produkt. Jeder Container ist dabei in der Lage, die Modellelemente, welche dieser enthalten kann, sowie auch Kanten durch einen Methodenaufruf zu erzeugen. Bei der Erstellung, Bearbeitung und dem Löschen von Modellelementen werden außerdem die implementierten Hooks ausgelöst, die auch im CINCO-Produkt unter manueller Bearbeitung ausgeführt werden. So geschieht etwa auch bei einer Implementierung die Erzeugung der Branches eines in einem Modell erzeugten SIBs vollkommen ohne weiteren Aufwand.

3.4.2 Modell-Generatoren

Wie bereits im vorherigen Abschnitt deutlich wurde, ist zur M2M von einem OntologyTool-Modell zu einer DIME-Applikation die Implementierung verschiedener Generatoren notwendig. Diese sind innerhalb einer einheitlichen Hierarchie organisiert, wie sie in Abbildung 3.41 dargestellt ist. So erben alle Generatoren von der abstrakten Klasse *CincoModelGenerator*. Diese ist zunächst unabhängig von einem konkreten Modelltyp und kann somit für beliebige CINCO-Modelle verwendet werden. Der CincoModelGenerator stellt die Implementierung von nachfolgenden Methoden in den Unterklassen sicher:

- Eine Methode *getNewModel*, die eine neue Instanz des konkretisierten Modelltyps zurückgibt.
- Eine Methode *init*, in der die Initialisierung eines Modells durchgeführt wird.

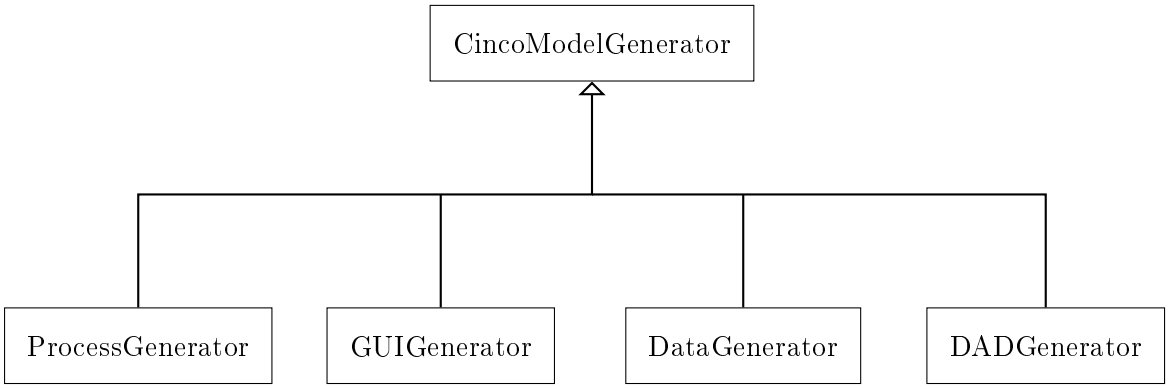


Abbildung 3.41: Vererbungshierarchie der Modellgeneratoren

- Eine Methode *execute*, in der die vollständige Generierung eines Modells durchgeführt wird.

- Eine Methode *getName*, die einen Namen für den implementierenden Generator zurückgibt. Dieser Name muss projektweit einzigartig sein.

Die hier angedeuteten Phasen der Initialisierung bzw. Generierung werden in Abschnitt 3.4.4 genauer betrachtet, können aber nun für jeden Generator mittels der vom CincoModelGenerator zur Verfügung gestellten Methoden *create* bzw. *generate* ausgeführt werden. Die abstrakten Unterklassen *ProcessGenerator*, *GUIGenerator*, *DataGenerator* und *DADGenerator* sind nun jeweils für die Generierung der entsprechenden DIME-Modelltypen konkretisiert. Mittels Implementierung der Methode *getNewModel* kann bereits hier über die jeweilige Factory eine neue Instanz des dazugehörigen Modelltyps zur Verfügung gestellt werden, sodass erbende Generatoren sich nicht um die Erstellung bemühen müssen.
Die letztendlichen Modell-Generatoren implementieren nun die abstrakten Generatoren ihres jeweiligen Modelltyps, wobei im Wesentlichen gesagt werden kann, dass pro Modellart bzw. Pattern ein Generator zur Verfügung steht und für eine typspezifische Umsetzung sorgt. Darüber hinaus sind für das Datenmodell, das DAD-Modell sowie für einige andere Modelle, welche die grundsätzliche Funktionalität der DIME-Applikation gewährleisten, weitere Generatoren notwendig. Eine Auflistung dieser sowie weitere Informationen zu deren Implementierung folgen in 4.2 sowie in Kapitel 6.

Hilfsmethoden

Die verschiedenen abstrakten Modell-Generatoren bieten teils einige Hilfsmethoden, welche die Implementierung erleichtern sollen. So kann etwa in Prozess- und GUI-Modellen das Code-basierte „Herausziehen“ eines komplexen Attributs und das Wiederfinden der dadurch neu entstehenden komplexen Variable mittels eines einfachen Aufrufs der Funktion *toComplexVariable* geschehen. Des Weiteren empfiehlt sich bei der Generierung von

Prozess-Modellen erst am Ende eine Positionierung der einzelnen Modellelemente vorzunehmen, was im ProcessGenerator automatisiert geschieht. Um dies zu ermöglichen, ist nach der Erstellung eines neuen Elements auf selbigem die Methode *layout* aufzurufen.
Um in der fertiggestellten Web-Applikation stets Informationen zu einem Objekt anzeigen zu können, bietet die in Kapitel 4 vorgestellte DSL die Möglichkeit zur Angabe eines *Identifiers*, der mittels sogenannter *Expressions* in doppelten geschweiften Klammern Objektspezifische Informationen enthalten kann. Diese können auch in GUI-Modellen von DIME verwendet werden, müssen aber auf eine Variable in einem Datenkontext bezogen werden. Dafür steht die Methode *convertIdentifier* zur Verfügung, die etwa zu einer Variable *current* und einem Typ *Person* mit den Attributen *vorname* und *nachname* die folgende Konvertierung des Identifiers vornehmen würde:

$$\{\{vorname\}\}\,,\ \{\{nachname\}\} \Longrightarrow \{\{current.vorname\}\}\,,\ \{\{current.nachname\}\}$$

Namensfindung für Modelle

Um nach Fertigstellung der Transformation eine funktionsfähige Applikation bereitstellen und Modelle während der Generierung eindeutig identifizieren zu können, ist es notwendig, eine geeignete Namensgebung für jedes Modell zu finden. Grundsätzlich lassen sich die zu generierenden Modelle in vier Kategorien einteilen:

1. Modelle, die nicht *einem* speziellen Typen zugeordnet werden können, z. B. das Datenmodell oder das DAD-Modell.

2. Modelle, die einmal pro Typ generiert werden und somit einem speziellen Typen zugeordnet werden können, z. B. ein GUI-Modell für eine Ansicht zum Bearbeiten.

3. Modelle, die mehrmals pro Typ für verschiedene Attribute generiert werden und somit einem speziellen Typen und einem konkreten Attribut zugeordnet werden können, z. B. Prozesse zum Entfernen von Elementen aus verschiedenen Listen-Attributen eines Typs.

4. Modelle, die mehrmals pro Typ generiert werden und dabei von einer anderen Information abhängig sind und somit dem Typen und der Information zugeordnet werden können, z. B. eine Detailansicht, die das eine Mal als Tabelle und das andere Mal als Liste zur Verfügung stehen soll.

Demgegenüber stehen folgende Voraussetzungen:

- Die Namen der Generatoren sind wie oben gefordert wurde einzigartig.

- Der Name eines Typs ist innerhalb eines Datenmodells einzigartig.

- Der Name eines Attributs ist innerhalb eines Typs einzigartig.

- Zusätzliche Informationen lassen sich eindeutig benennen.

Unter den o.g. Bedingungen lässt sich somit für jedes Modell ein eindeutiger Name definieren. Für Modelle, die unabhängig von einem konkreten Typen sind, kann so entweder der Name des Generators oder ihr „traditioneller“ Name verwendet werden. So heißt etwa ein Datenmodell in DIME standardmäßig *app.data* und das DAD-Modell *app.dad*. Allen anderen Modelle können durch eine Konkatenation von den Namen des Generators und ihren Abhängigkeiten eindeutige Identifizierungen zugeordnet werden. So ergeben sich die Namen der Kategorien 2-4 wie folgt:

2. «Typname»«Generatorname»
3. «Typname»«Generatorname»«Attributname»
4. «Typname»«Generatorname»«Informationsname»

Da es im Laufe des Projektes Problematiken wegen zu langen Dateinamen gegeben hat, werden die Namen aller Typen für die Namensfindung soweit gekürzt, dass die Zuordnung trotzdem noch eindeutig erfolgen kann.

3.4.3 ModelProvider

Als zentrale Datenstruktur und Zugriffsquelle für generierte Modelle dient während der Transformation der *ModelProvider*. Die Notwendigkeit einer solchen Datenstruktur ergibt sich dadurch, dass in einigen Modellen andere generierte Modelle referenziert werden müssen. Die Speicherung erfolgt innerhalb des ModelProviders durch ein Mapping vom eindeutigen Namen des Modells auf die Modell-Instanz. Mittels einer umfangreichen Ansammlung an Zugriffsmethoden kann dann der einfache Abruf auch Typ-, Attribut- oder Informationsabhängiger Modelle ermöglicht werden.
Seit dem Hinzukommen der Canvas-DSL wurde hier eine Trennung der ModelProvider-Struktur vorgenommen, sodass zu jeder DSL jeweils eine eigene Implementierung des ModelProviders zur Verfügung steht. So kann sichergestellt werden, dass für die einzelnen Bereiche nur die jeweils notwendigen Zugriffsmethoden angeboten werden, wodurch insgesamt die Übersichtlichkeit gesteigert wird.

Die Implementierung sämtlicher Generatoren erfolgt mit der Programmiersprache *Xtend* [8]. Die auf Java basierende Sprache bietet eine Vielzahl an verkürzten Schreibweisen und weitere hilfreiche Werkzeuge, die ausführlich in der umfangreichen Dokumentation [7] beleuchtet werden. Namensgebendes Merkmal von Xtend sind die sogenannten *Extensions*, durch die es ermöglicht wird, Methoden anderer Klassen ohne den expliziten Aufruf auf einem entsprechenden Objekt zur Verfügung gestellt zu bekommen. Als eine solche Extension wird auch der ModelProvider den Generatoren übergeben, sodass der Zugriff auf

sämtliche generierten Modelle durch einfache Ausdrücke möglich ist. Um beispielsweise an die Übersichtsansicht *OverviewGUI* einer Typ-Variablen *type* zu gelangen, genügt hier der Aufruf

```
type.overviewGUI
```

statt in Java:

```
provider.getOverviewGUI(type);
```

3.4.4 Transformationsphasen

Um eine strukturierte und korrekte Transformation gewährleisten zu können, wurde diese in verschiedene Phasen aufgeteilt. Gesteuert wird der Ablauf durch eine zentrale Organisationsstruktur.
Die für die einzelnen Schritte notwendigen Modell-Generatoren werden dabei von *GeneratorFactories* bereitgestellt, die jeweils für einen Teilbereich der Transformation zur Verfügung stehen:

- *OntologyToolGeneratorFactory*: stellt alle Generatoren zur Verfügung, die wichtige Rahmenmodelle erzeugen, wie etwa das Datenmodell oder das DAD-Modell.
- *OntologyGeneratorFactory*: stellt alle Generatoren zur Verfügung, die Modelle für die Ontologie-Verwaltung erzeugen.
- *CanvasGeneratorFactory*: stellt alle Generatoren zur Verfügung, die Modelle für Canvas-Funktionalitäten erzeugen.

Die in Abbildung 3.42 dargestellten Phasen werden nachfolgend genauer betrachtet.

Projekterzeugung und Generierung des Datenmodells

Die erste Transformationsphase schafft die ersten notwendigen Voraussetzungen zur eigentlichen Generierung des vollständigen Endproduktes. So wird zunächst ein Projekt und dessen Ordnerstruktur in der laufenden Eclipse-Instanz erstellt und die statischen Inhalte hinzugefügt. Des Weiteren wird ein ModelProvider zur Verfügung gestellt.
Daraufhin wird bereits das Datenmodell der DIME-Applikation generiert, was in mehreren Schritten erfolgt:

1. Zunächst wird für alle Elemente aus dem Ontologie-Modell ein entsprechendes Element im Datenmodell angelegt. Dabei werden DerivedAttributes vorerst nicht berücksichtigt.
2. Für jedes DerivedAttribute werden Prozesse initialisiert (siehe unten) und als sogenannte *Extension-Attribute* in das Datenmodell eingefügt. Extension-Attribute sind

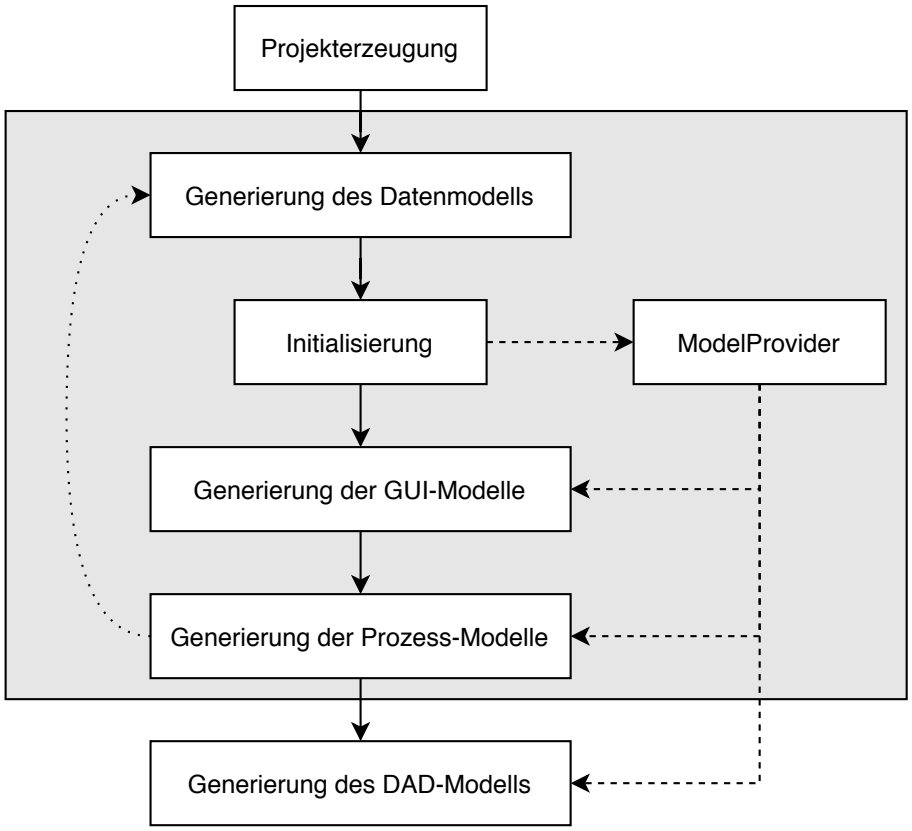


Abbildung 3.42: Phasen der Transformation

ein besonderer Attribut-Typ in DIME, bei dem der aktuelle Wert durch einen Prozess bestimmt werden kann.

Die Notwendigkeit dieser Vorgehensweise ergibt sich dadurch, dass bereits die Initialisierung der Prozesse für die Extension-Attribute die Typen des Ontologie-Modells und deren Attribute benötigt. Für alle weiteren Transformationsschritte ist nun das vollständige Datenmodell inklusive der Extension-Attribute Voraussetzung. Die bereits angesprochene Initialisierung wird nun im Folgenden näher beleuchtet.

Initialisierung

Die Initialisierung der Generatoren und Modelle beinhaltet im Wesentlichen die folgenden Schritte:

- Die benötigten Generatoren werden erstellt und der ModelProvider wird ihnen übergeben.
- Alle Modelle, die im Endprodukt enthalten sind, werden mit all ihren Inputs und Outputs erzeugt.

Dadurch, dass hier bereits sämtliche Modelle angelegt werden, wird gewährleistet, dass bei der eigentlichen Generierung bereits alle Modelle existieren, die etwa als Prime-Reference in anderen Modellen benötigt werden. Außerdem können wegen der bereits vorhandenen Inputs und Outputs alle notwendigen Datenflusskanten gesetzt werden. Beispiele für eine

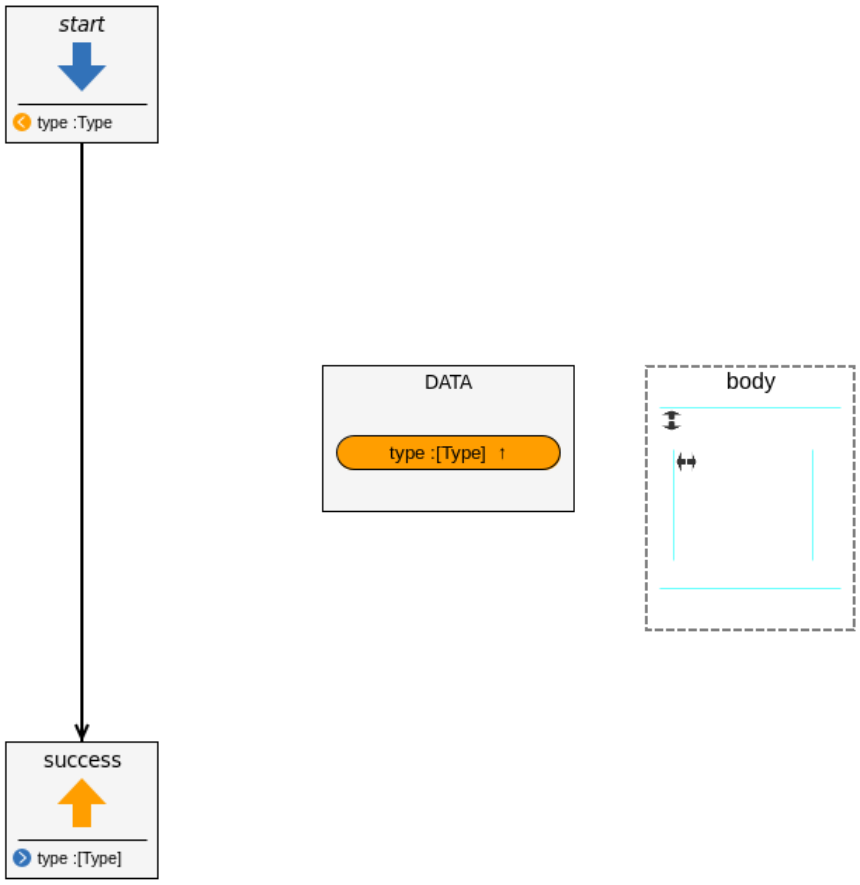


Abbildung 3.43: Prozess-Modell (links) und GUI-Modell (rechts) nach der Initialisierung

derartige Modell-Initialisierung sind in Abbildung 3.43 dargestellt. Prozess-Modelle enthalten dementsprechend nach der Initialisierung

- den StartSIB mit allen Outputs,
- sofern vorhanden, sämtliche EndSIBs mit all ihren Inputs,
- ggf. Kontrollflusskanten vom StartSIB zum EndSIB.

Die im letzten Punkt genannten Kontrollflusskanten waren zu Anfang des Projektes zur Sicherung der Korrektheit des Modells notwendig. Mittlerweile wurden allerdings Anpassungen in DIME getroffen, sodass diese während der Generierung entfallen können. Im Allgemeinen müssen diese im nächsten Schritt der Transformation ohnehin wieder entfernt werden.

GUI-Modelle enthalten nach ihrer Initialisierung

- die bei der Modellerzeugung automatisch angelegten Modellelemente (Datenkontext, Template für den Body),
- alle Inputs im bereits vorhandenen Datenkontext.

Während der Modell-Initialisierung fügt jeder Generator das eigene Modell zur ModelProvider-Datenstruktur hinzu, sodass daraufhin auch alle anderen Generatoren Zugriff darauf haben. Nach der erfolgreichen Initialisierung kann die eigentliche Generierung der Modelle erfolgen.

Generierung der GUI- und Prozess-Modelle

Die finale Generierung teilt sich letztlich in zwei Phasen auf, während derer nun allen Generatoren sämtliche initialisierten Modelle durch den Zugriff auf den ModelProvider zur Verfügung stehen. Im ersten Schritt werden hier sämtliche GUI-Modelle erzeugt, denn es ist dringend erforderlich, dass diese vor den Prozess-Modellen vervollständigt werden. Dies ist durch die Tatsache begründet, dass beim Einfügen von GUI-SIBs in Prozess-Modelle erst dann alle Branches per Hook erzeugt werden, wenn im entsprechenden GUI-Modell alle Buttons, LoadSubmit-Kanten, etc. vorhanden sind.
Ist die erste Phase der Generierung beendet, werden nachfolgend alle Prozesse vollständig generiert. Wie bereits zuvor erwähnt, sind an dieser Stelle unter Umständen Anpassungen notwendig, die eventuell während der Initialisierung getätigte Erzeugungen von Kanten rückgängig machen.

Generierung der Canvas-Modelle

In dem Fall, dass im zu transformierenden OntologyTool-Modell Canvas Modelle referenziert wurden, müssen - ausgenommen von der Projekterzeugung - sämtliche genannten Phasen wiederholt werden. Obwohl es technisch unproblematisch wäre, alle Modelle bereits in der ersten Iteration zu erzeugen, wurde die Notwendigkeit einer zweiten Iteration vor allem durch das späte Hinzukommen der Canvas-Modell-Transformation bedingt. Das liegt im Wesentlichen daran, dass dafür eine Erweiterung des Datenmodells notwendig ist, die in 6.4 erklärt wird. Die zahlreichen bereits implementierten Generatoren der anderen Bereiche gehen allerdings von einem unbearbeiteten Datenmodell aus, weshalb der Einfachheit halber auf diese Lösung zurückgegriffen wurde.

Abschluss der Generierung

Im letzten Schritt erfolgt die Generierung des DAD-Modells, welches das DIME-Projekt schlussendlich komplettiert. Damit ist der Transformationsprozess von einem OntologyTool-

Modell zu einer funktionsfähigen DIME-Applikation abgeschlossen.

Der wesentliche Vorteil der o. g. Aufteilung der Transformation in die verschiedenen Phasen ergibt sich zusammenfassend also vor allem dadurch, dass zwar die Reihenfolge der einzelnen Phasen nicht geändert werden darf, die Reihenfolge der Generierung innerhalb der Phasen aber zu keiner Zeit von Relevanz ist. Dies ermöglicht auch das parallele Ausführen von Generatoren innerhalb einer Transformationsphase. Insgesamt wird so die Komplexität der Transformation gegenüber einer von der Reihenfolge abhängigen Lösung erheblich reduziert.
Des Weiteren ist diese Vorgehensweise nicht beschränkt auf das hier beschriebene Projekt und kann auch als Konzept zur Generierung anderer DIME-Applikationen verwendet werden.

Kapitel 4

Implementierung Ontologie

4.1 DSL

Bei einer DSL handelt es sich, wie in 2.1.3 und 2.2 beschrieben, um eine Sprache für einen spezifischen Anwendungsfall. Der Anwendungsfall dieser Projektgruppe ist die Erzeugung von Firmenontologien. Dementsprechend muss eine DSL erstellt werden, mit der solche Ontologien erzeugt werden können. In 3.1 wurden Anforderungen an eine konkrete Ontologie am Beispiel von Schulz gestellt. Diese ursprünglichen Anforderungen und die Ontologie sollen als Grundlage für die Abstraktion der DSL dienen. Die Hauptanforderung dieser DSL ist, dass die mit Schulz zusammen erarbeitete Ontologie mit ihr erzeugbar ist.
Um eine möglichst einfache und intuitive Nutzung der DSL zu ermöglichen, ergeben sich folgende weitere Anforderungen:

- Anlehnung der DSL an eine bekannte grafische Modellierungssprache
- Hilfestellung beim Erzeugen einer Ontologie in Form von Fehlermeldungen bei falscher Benutzung
- Deskriptive Bezeichnungen für neue oder spezifische Elemente der DSL
- Minimalisierung der DSL; Standardfälle voraussetzen und Anpassungsmöglichkeiten reduzieren, um Benutzbarkeit zu vereinfachen

Zusätzlich soll die DSL eine externe Dokumentation und Anleitung zur Benutzung erhalten. Dieser User-Guide ist aber keine Anforderung an die DSL an sich.

4.1.1 Basis der DSL

Die gewünschte DSL wird mit CINCO erstellt. Die DSL wird allerdings nicht von Grund auf entwickelt. Als Grundlage für die DSL haben wir uns für die DIME-Datenmodell-DSL entschieden. Dieses wurde ebenfalls in CINCO erstellt und ist somit sowohl mit CINCO

als auch mit DIME kompatibel.
Die Nutzung von CINCO mit der DIME-Datenmodell-DSL hat dabei einige entscheidende Vorteile. Der größte Vorteil, ist wie bereits angemerkt, die Interoperabilität zwischen beiden Anwendungen. Zusätzlich orientiert sich die DIME-Datenmodell-DSL an der Objektorientierung nach JAVA-Implementierung. Es ist dementsprechend ähnlich zu UML und lässt sich mit geringem Vorwissen in seinen Grundlagen verstehen. Des Weiteren ist das Datenmodell frei erweiterbar. Daher ist es möglich, unsere gesamten zusätzlichen Anforderungen in diesem Datenmodell umzusetzen.
Die Basis unserer DSL besteht aus den aus der Objektorientierung bekannten Klassen: Abstrakten Klassen, Attributen, Vererbungshierarchien, sowie der Polymorphie und Assoziationen. Grundsätzlich bietet die DIME-Datenmodell-DSL bereits den Großteil der Funktionalität, um eine Firmenontologie erstellen zu können (siehe Abbildung 3.1). Allerdings wurden bei der Umsetzung dieser Ontologie in eine Anwendung mit DIME Annahmen von unserer Seite getroffen, andere Datentypen per Hand implementiert oder neue Abhängigkeiten zwischen Objekten genutzt. Damit diese von uns umgesetzten Änderungen im weiteren Schritt automatisch generiert werden können, bedarf es der Formalisierung eben jener in unserer eigenen DSL.

4.1.2 Implementierung der Features in der DSL

Im Folgenden werden einige der größten und wichtigsten Features thematisiert, die Einzug in die neue DSL erhalten haben.

Bäume (nach 3.3)

Die Baumstruktur ist in vielen Anwendungsfällen nützlich. In der Schulz-Ontologie gibt es die Anforderung, dass Technologien Untertechnologien beinhalten können. Diese Untertechnologien können ebenfalls Untertechnologien haben. Die damit beschriebene Datenstruktur für Technologien ist ein Baum. Damit die Besonderheit der Baumstruktur für den Nutzer deutlich wird und gleichzeitig in der Transformation eindeutig wird, gibt es den Baum als Datentyp in unserer DSL. Bäume erhalten implizit einen Elterntyp und eine Liste von Kindertypen des selben Typs wie der Baum.

Derived Attribute

Derived Attributes sind Attribute, die nicht wirklich Teil eines Typs sind, sondern transitiv über eine Assoziationskette abgeleitet (engl. derived) werden. Derived Attributes sind also Attribute eines anderen Typs, zu dem ein Typ eine Assoziationsverbindung hat. Ein Anwendungsfall ist zum Beispiel, dass der Abteilungsleiter immer in den Projekten seiner Abteilung als Verantwortlicher vermerkt werden soll. Ein Derived Attribute in einem

Projekt, das über die Abteilung, die dem Projekt zugewiesen wird, automatisch den Abteilungsleiter findet, sorgt für eine Automatisierung von voneinander abhängigen Information und verringert fehlerhafte Zuweisungen. Da das Derived Attribute abgeleitet wird, ist es in diesem Beispiel unmöglich eine falsche Person als Abteilungsleiter im Projekt zu referenzieren.

Aggregate Association

Aggregationen sind besondere Assoziationen. Typen, die andere Typen aggregieren, erstellen und entfernen die aggregierten Typen, wenn sie selbst erstellt oder entfernt werden. Die Existenz aggregierter Typen ist an einen anderen Typen gebunden. Aggregationen sind sehr hilfreich, wenn es Typen gibt, die nur Sinn in einem anderen Kontext ergeben. Die Adresse eines Standortes ist ein solcher Fall. Wenn ein neuer Standort angelegt wird, wird für ihn auch eine neue Adresse angelegt. Wird der Standort entfernt, ergibt es wenig Sinn, die Adresse weiterhin als Objekt in der Anwendung zu behalten. Die Aggregate Association stellt sicher, dass dieser Zusammenhang zwischen zwei Typen immer bewahrt wird.

Type Properties

Damit die Anwendung vollständig generiert werden kann, sind viele Annahmen getroffen worden. Ein Großteil dieser Annahmen oder Auswahlmöglichkeiten werden in Type Properties als einstellbare Optionen umgesetzt. Type Properties sind Einstellungsmöglichkeiten, die abhängig vom jeweiligen Typ, Attribut oder Assoziation sind. So haben konkrete Typen, abstrakte Typen und Bäume unterschiedliche Type Properties. Komplexe und primitive Attribute haben ebenfalls unterschiedliche Type Properties. Für alle Arten von Typen, kann eingestellt werden, ob diese Typen Roottypes sind, welche Anzeigenamen sie erhalten sollen und an welcher Stelle sie in der Menüleiste platziert werden sollen. Bei komplexen Attributen kann eingestellt werden, auf welche Art und Weise sie angelegt oder ausgewählt werden, ob es sich um eine Liste handelt, ob die Belegung des Attributs vorausgesetzt ist, ob es einzigartig sein soll, und welcher Struktur es angezeigt werden soll. Bei den möglichen Einstellungen ist Wert darauf gelegt worden, dass möglichst wenig vom Nutzer eingestellt werden kann und möglichst viel impliziert wird. Nur die Funktionalität der resultierenden Anwendung kann angepasst werden, das Design ist festgelegt und lässt sich vom Nutzer nicht über die DSL verändern.

4.1.3 Custom Checks und Appearances

Um dem Nutzer Rückmeldung darüber zu geben, ob die von ihm erstellte Ontologie korrekt im Sinne der Sprache ist, oder auch um den Nutzer Hilfestellung bei der Erstellung zu geben, gibt es *Checks* (Überprüfungen) und *Appearances* (Aussehen). Die Checks prüfen

die Ontologie auf ihre Korrektheit und geben eine Fehlermeldung mit detaillierter Fehlermeldung und Hilfestellung zurück. Appearances verändern das Aussehen der Ontologie, um Zusammenhänge besser sichtbar zu machen. Im Folgenden sind einige der wichtigeren neuen Checks und Appearances und ihre Funktionalität aufgelistet.

rootsExistCheck

Die Ontologie benötigt mindestens einen Root-Typ, da diese die Einstiegspunkte für die Anwendung sind.

isCreateableCheck

Jeder Typ in der Ontologie muss erstellbar sein. Das bedeutet, dass er entweder ein Root-Typ sein muss, oder über eine Kette von erstellbaren Typen erstellbar sein muss.

rootAttributeNotCreateableCheck

Ein Root-Typ ist immer über das Root-Menü der Anwendung erstellbar und darf nicht von anderen Typen aggregiert oder erstellt werden.

abstractImplementedCheck

Jeder abstrakte Typ muss in der resultierenden Anwendung auch als Objekt erstellbar sein können. Deswegen müssen abstrakte Typ am Ende ihrer Vererbungshierarchie in einem konkreten Typen enden. Diese Anforderung gilt für jeden Vererbungspfad, der über diesen abstrakten Typ verläuft.

multipleInheritanceCheck

Typen dürfen höchstens von einem anderen Typen erben.

aggregateMustBeCreateableCheck

Aggregierte Typen müssen in den Type Properties zum Anlegen oder Auswählen auf Createable (also erstellbar) gesetzt werden. Die *Aggregate-Association* dient als Ausgangspunkt für die Aggregation; damit keine falsche Auswahlkombination getroffen wird, also eine Aggregation angelegt wird aber gleichzeitig das Objekt ausgewählt und nicht erstellt werden soll, erinnert der Check daran, die Auswahl auf Createable zu setzen.

derivedPathCheck

Der Pfad eines Derived Attributes zu seinem korrespondierenden Typen wird über einen Moustache-Ausdruck angegeben. Ein Moustache-Ausdruck ist eine Zeichenkette über die Typen und deren Attribute. Eine erweiterte Beschreibung findet sich im User-Guide im

Anhang. Der Check versucht, den Pfad durchzulaufen und den entsprechenden Typ zu erreichen. Falls der Durchlauf entweder nicht zum gewünschten Ziel führt, oder der Ausdruck syntaktische Fehler enthält, wird dem Nutzer eine Rückmeldung gegeben.

Zusätzliche Checks

Zusätzlich gibt es noch weitere kleine Checks, die überprüfen, dass keine reservierten Zeichenketten in Namen vorkommen, dass Abhängigkeiten zwischen Typen richtig sind, oder dass Bäume korrekt benutzt werden.

Root Type Appearance

Root Types werden in grauer Farbe hinterlegt, um sie von den anderen Typen abzuheben.

Tree Types

Bäume haben ihr eigenes Symbol bekommen, damit sie zu der Optik der anderen Typen passen.

4.1.4 Beispielhafte Umsetzung einer Firmenontologie in der Ontologie-DSL

Mithilfe der so geschaffenen Ontologie-DSL, kann die Schulz-Ontologie nachgebaut werden. Zusätzlich dazu werden durch die neu hinzugefügten Funktionalitäten Verhältnisse deutlicher ausgedrückt und angelegt, als es vorher mit der DIME-Datenmodell-DSL möglich gewesen ist. Im Vergleich zur ersten Schulz-Ontologie in Kapitel 3.1 sind im Folgenden die großen und nennenswerten Änderungen aufgelistet. Eine beispielhafte Anleitung, wie eine Ontologie von Grund auf erstellt werden kann, ist im User-Guide im Anhang zu finden.
Die Abbildung 4.1 zeigt die vergleichbare Ontologie. Die Root-Typen sind grau hervorgehoben und haben alle einen individuellen Index erhalten. Technologien und Branchen wurden mit den richtigen Baumtypen umgesetzt. Von den Root-Typen aus sind alle anderen referenzierten Typen zumindest in einem Root-Typ Createable. Für alle komplexen Attribute wurde ein Moustache-Ausdruck angegeben, auf welche Art und Weise sie angezeigt werden sollen. Der Adresstyp wird immer aggregiert. Für jeden Typ und jedes Attribut wurden die *Type Properties* so eingestellt, wie sie gewünscht sind. Einige komplexe Listenattribute werden bei der Auswahl als Tabelle angezeigt, andere als eine Liste mit Checkboxen. Primitive Attribute wie zum Beispiel der Name eines Standortes sind Unique; ihr Name kann also unter allen Standorten nur ein mal vergeben werden. Bei bidirektionalen Assoziationen gibt es Typen, die in eine Richtung den anderen Typen anlegen und auf der Rückrichtung ohne Auswahlmöglichkeiten nur angezeigt werden. Es gibt ebenfalls solche Assoziationen, bei denen auf beiden Seiten der Assoziation Auswahlen getroffen werden können.

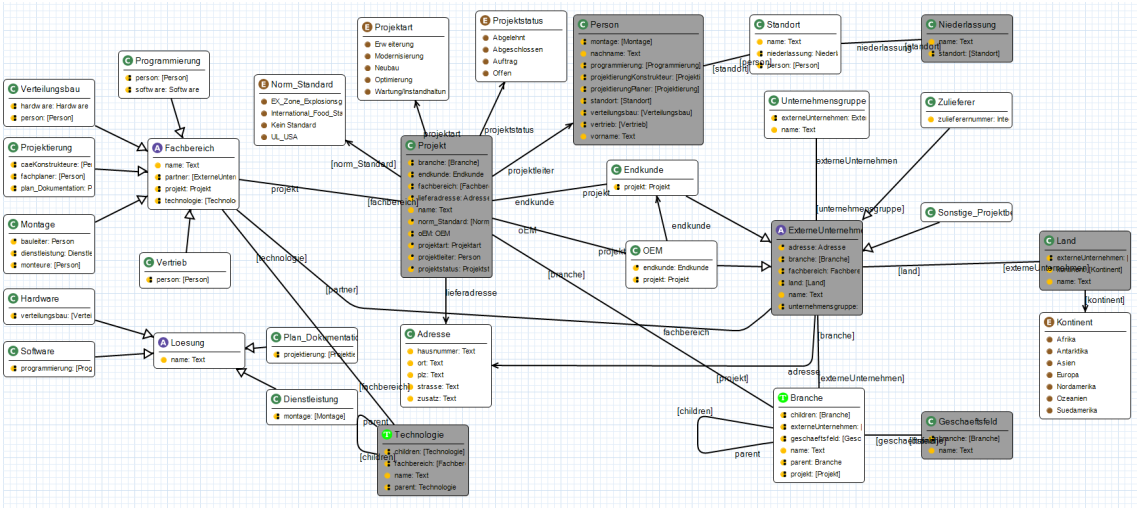


Abbildung 4.1: Die Ontologie der Firma Schulz Systemtechnik GmbH mit der Ontologie-DSL

Mit dieser Ontologie-DSL können jetzt Firmenontologien erzeugt werden. Wenn sich an alle Vorgaben gehalten wird, und am Ende keine Fehler mehr angezeigt werden, dann kann die resultierende Ontologie mithilfe der nachfolgend beschriebenen Transformatoren automatisch in eine Anwendung transformiert werden.

4.1.5 OntologyTool-DSL

Um die hier beschriebene Ontologie-DSL schlussendlich mit der in Kapitel 6 vorgestellten Canvas-DSL zusammenführen zu können, wurde die *OntologyTool*-DSL entwickelt.

Abbildung 4.2: Ein OntologyTool-Modell

Im Wesentlich stellt die OntologyTool-DSL nur zwei unterschiedliche Arten von Knoten zur Verfügung (Abb. 4.2):

- *Ontology*-Knoten: dient als Prime-Reference für ein Ontology-Modell.
- *Canvas*-Knoten: dienen als Prime-Reference für Canvas-Modelle.

Der Nutzer kann so durch ein OntologyTool-Modell definieren, welches Ontology-Modell, bzw. welche Canvas-Modelle im Endprodukt enthalten sein sollen. Grundsätzlich müssen dafür folgende Voraussetzungen erfüllt sein:

- Es muss genau ein Ontology-Modell enthalten sein.

- Es können beliebig viele Canvas-Modelle enthalten sein.
- Jedes der Canvas-Modelle darf ausschließlich Elemente aus dem referenzierten Ontology-Modell enthalten.

Diese Voraussetzungen werden durch entsprechende Checks überprüft.

OntologyTool-Modelle können somit wie auch DAD-Modelle in DIME als Einstiegspunkt der Generierung dienen und ermöglichen die Transformation mehrerer Modelle in ein zusammenfassendes Projekt.

4.2 Transformator

Der nachfolgende Abschnitt beschäftigt sich mit den Modell-Generatoren, die zur Umsetzung des in Kapitel 3.4 beschriebenen Konzepts zur Transformation implementiert wurden. Nach Betrachtung der Datenmodell-Generierung folgt im weiteren Verlauf zunächst eine Übersicht über die Prozess- und GUI-Generatoren, nach der schließlich die Implementierung an einigen Beispielen genauer betrachtet wird.

4.2.1 DataModelGenerator

Der *DataModelGenerator* übernimmt die Generierung des Datenmodells. Als direkte Vorlage dient hier das Ontologie-Modell, welches nahezu kopiert werden kann. Da die Ontologie-DSL auf der Datenmodell-DSL basiert und diese grundsätzlich nur mit Informationen anreichert, besteht die Erzeugung des Datenmodells im Wesentlichen aus dem Auslassen aller hinzugekommenen Eigenschaften. Insgesamt werden folgende Punkte ohne Abänderungen übernommen:

- Typen und Attribute (bis auf DerivedAttributes) sowie deren Namen
- Kanten und Listeneigenschaften
- Positionen von allen Knoten,
- weitere in DIME verfügbare Eigenschaften von Typen und Attributen,
- Typart (ConcreteType, AbstractType, EnumType, UserType, primitive Attribute),
- Sichtbarkeit von Kanten.

Abgesehen davon wird für jedes DerivedAttribute in der DIME-Applikation ein Prozess erzeugt, der im DataModelGenerator als *Extension-Attribute*, einem speziellen Attribut-Typ in DIME, bei dem der Wert durch einen Prozess ermittelt wird, an die entsprechenden Stellen in das Datenmodell eingepflegt.

Außerdem wird der *ConcreteTreeType*, der gesondert behandelt werden muss, da dieser nicht in der DIME-DSL vorhanden ist, zu einem ConcreteType konvertiert und die Attribute *children* und *parent*, sowie der entsprechende Helfertyp (siehe 3.3.4) erstellt.
Ansonsten fließen alle zusätzlichen Informationen, die im Ontologie-Modell enthalten sind, ausschließlich in die anderen generierten Modelle.

Für die Generierung der Canvas-Modelle bietet der DataModelGenerator eine Methode zur Erweiterung des Datenmodells. Dies wird in 6.4 weiter ausgeführt.

4.2.2 Übersicht Prozess- und GUI-Generatoren

- *HomeProcess*
  Der Prozess verknüpft die HomeGUI, die RootGUI und die SplashScreenGUI. Er bildet damit den Start der Anwendung.

- *CRUDProcessGenerator*
  Der CrudProcessGenerator beinhaltet die OverviewGUI eines Typs. Er leitet die zugehörigen SIBs zum Erstellen, Bearbeiten und Löschen weiter. Dieser Generator wird in Abschnitt 4.2.3 näher betrachtet.

- *AbstractCRUDProcessGenerator*
  Analog zum CRUDProcessGenerator beinhaltet der AbstractCRUDProcessGenerator die OverviewGUI des Obertypen und alle OverviewGUIs der erbenden Typen. Die ausgehenden Bearbeiten-Kanten führen zu separaten EndSIBs, während die Lösch-Kanten mit einer einzigen EndSIB verbunden werden.

- *EditViewProcessGenerator*
  Dieser Prozess enthält die EditViewGUI. Sofern bereits ein Objekt existiert, werden die zugehörigen Daten in die GUI geladen.

- *ToListCRUDProcess*
  In diesem Prozess wird der CRUDProcess aufgerufen. Da der CRUDProcess nur Listen als Eingabe akzeptiert, werden hier einzelne Objekt-Elemente in eine Liste umgewandelt, bevor sie dem CRUDProcess übergeben werden.

- *RetrieveProcess*
  Der RetrieveProcess verknüpft den CRUDProcess mit dem EditViewProcess. Die Bearbeiten-Ausgänge des CRUDProcess' werden zum EditViewProcess geleitet.

- *AbstractRetrieveProcess*
  Der AbstractRetrieveProcess leistet die selbe Funktionalität wie der RetrieveProcess für abstrakte Typen.

- *SubTypeProcess*
  Der SubTypeProcess liefert für einen übergebenen Typen seine Subtypen zurück.

- *ModalProcessGenerator*
  Der Prozess ruft die InnerDetailViewGUI auf.

- *ContainingListGenerator*
  Dieser Prozess wird für das modale Löschen benötigt. Es wird eine Liste von Objekten zurückgegeben, in denen das zu löschende Objekt referenziert wird.

- *ModalDeleteProcess*
  Der ModalDeleteProcess setzt das modale Löschen um. Hierbei werden alle referenzierten Objekte des zu löschenden Elements mit entfernt.

- *UniqueCheckProcess*
  In diesem Prozess wird geprüft, ob bereits ein Objekt mit demselben Variablenwert existiert.

- *AddSingleToListGenerator*
  Dieser Prozess fügt ein einzelnes Element einer Liste hinzu.

GUIs:

- *OverviewGenerator*
  Der OverviewGenerator erzeugt eine Verwaltungs-GUI für alle Objekte eines angegebenen Typs. Dieser wird in Abschnitt 4.2.4 vorgestellt.

- *AbstractOverviewGenerator*
  Der AbstractOverviewGenerator erzeugt eine Verwaltungs-GUI für alle Objekte eines angegebenen vererbenden Typs. Diese besteht aus den in Abschnitt 4.2.4 vorgestellten GUIs für jeden Subtypen.

- *InnerDetailViewGUIGenerator*
  Der InnerDetailViewGenerator erstellt für komplexe Typen eine Detailansicht, die einen Überblick über die Attribute ermöglicht. Die Generierung wird in Abschnitt 4.2.5 präsentiert.

- *EditViewGenerator*
  Der EditViewGenerator erzeugt eine Bearbeitungs-Übersicht für den angegebenen Typen und seine Attribute. Auf die Funktionsweise wird genauer in Abschnitt 4.2.6 eingegangen.

- *ModalDeleteViewGenerator*
  Ein Popup-Fenster, das darauf hinweist, dass referenzierte Objekte mitgelöscht werden und bittet, den Löschvorgang zu bestätigen. Die referenzierten Objekte werden angezeigt.

- *HomeGUI*
  Die HomeGUI ist die Startseite der Anwendung. Sie enthält die Navigationsleiste und einen Placeholder für den Splashscreen.

- *RootGUI*
  Die RootGUI enthält die RetrieveProcesse der Root-Elemente.

- *SplashScreenGUI*
  Die SplashScreenGUI beinhaltet den Splashscreen, sofern ein Splashscreen ausgewählt wurde.

4.2.3 CrudProcessGenerator

In diesem Abschnitt wird detaillierter auf die Implementierung des CrudProcessGenerator eingegangen, in dem die jeweiligen SIBs der OverviewGUI zum Erstellen, Bearbeiten und Löschen an den RetrieveProcess weitergeleitet werden. Abbildung 4.3 zeigt einen generierten CrudProcess für einen Typ *Person*. Der Prozess beginnt mit der StartSIB. Diese verfügt über eine Liste von Personen als Attribut, sowie über zwei primitive Input-Ports *hasHeadline* und *hasHeadlineNormal* vom Datentyp Boolean. Diese sind später für die OverviewGUI wichtig, um die Ausrichtung und Art der Überschrift zu bestimmen. Nach der StartSIB folgt eine OverviewGUI für *Person*. Hierbei handelt es sich um eine Übersicht über alle Objekte des Typs, die hier verwaltet werden können. Der Input der OverviewGUI sind die Attribute der StartSIB. Von der OverviewGUI aus werden die weiteren Bearbeitungsmöglichkeiten von *Person* (*Löschen, Bearbeiten* und *Erstellen*) weitergeleitet. Der Branch *modalDeleteSuccess* wird zum Löschen benötigt und wird über die EndSIB *DS* dem RetrieveProzess zum modalen Löschen übergeben. Für die beiden Optionen *Erstellen* und *Bearbeiten* wird eine gemeinsame EndSIB *PES* verwendet. Dieser Ausgang wird im RetrieveProcess mit der EditView verknüpft. Beim *Bearbeiten*, das über den Branch *EditP* von der OverviewGUI aus angestoßen wird, ist bereits ein Objekt vom Typ *Person* vorhanden, das bearbeitet werden soll. Es wird an die EndSIB übergeben. Beim *Erstellen*, das über den Branch *CreateP* angeregt wird, muss zunächst ein leeres Objekt erstellt werden, welches von der EditView verwendet werden kann. Hierzu wird eine CreateSIB eingebunden, die dann mit der EndSIB *PES* verbunden wird.
Am oberen linken Ende der Abbildung befindet sich außerdem ein Datenkontext, durch den das jeweilige Objekt in der Datenbank persistiert wird.

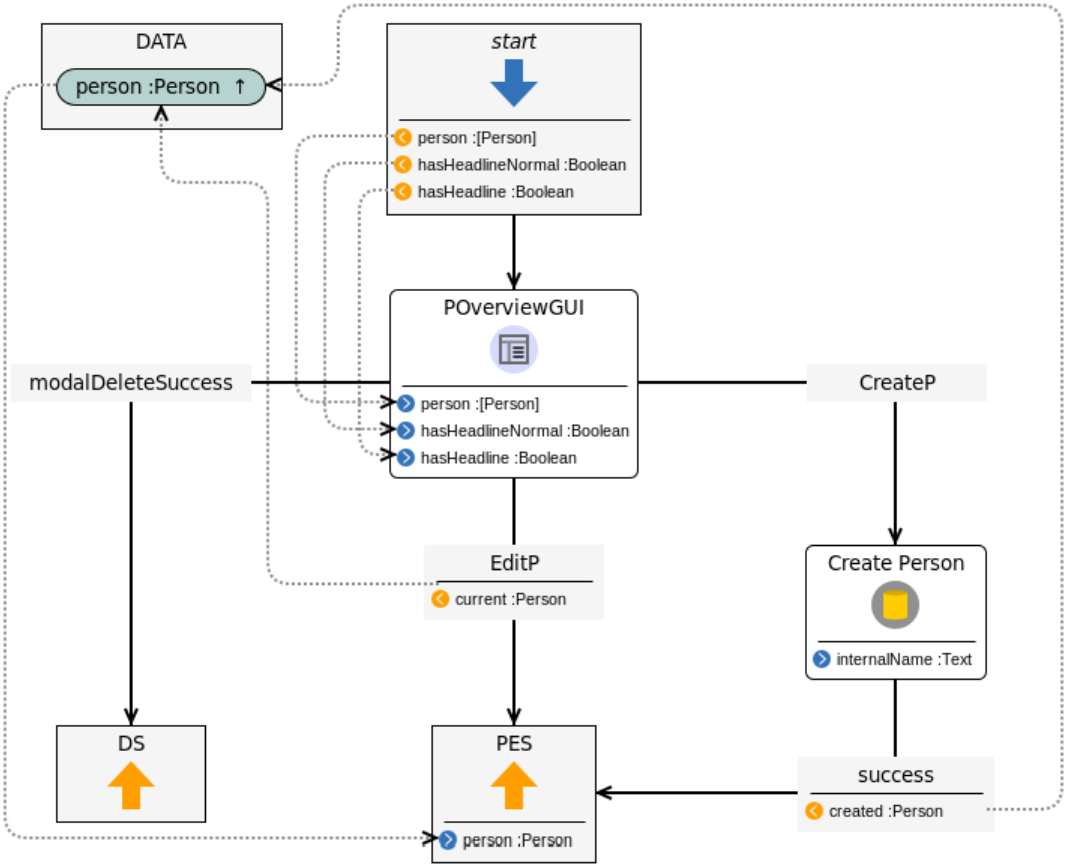


Abbildung 4.3: Durch den CrudProcessGenerator generierter Prozess für einen Typ *Person*

Die Generator-Klasse besteht aus einem Konstruktor, in dem der *Dime*-Typ übergeben wird. Zusätzlich kann ein komplexes Attribut des Typs übergeben werden. Dies ist nötig, da zum Bearbeiten von komplexen Attributen jeweils ein eigener Generator erstellt wird. Neben dem Konstruktor besteht die Klasse aus drei überschriebenen Methoden, die der ProcessGenerator zur Verfügung stellt.

In der *init*-Methode wird die StartSIB mit ihren primitiven Attributen und ihrem komplexen Attribut vorinitialisiert. Zusätzlich werden eine EndSIB zum Bearbeiten und eine EndSIB zum Löschen erstellt.

Die *getName*-Methode liefert einen einzigartigen Namen für den Generator zurück, um eindeutige Modellnamen erstellen zu können.

In der *execute*-Methode werden die Hauptfunktionalitäten implementiert. Es werden zunächst alle Komponenten (in Abbildung 4.3 zu sehen) erzeugt. Sie werden durch die Erstellung der jeweiligen Kanten miteinander verbunden. Über die Kontrollflusskanten können die jeweiligen Branches gefunden werden. Die Input- und Output-Ports werden anschließend ebenfalls miteinander verbunden.

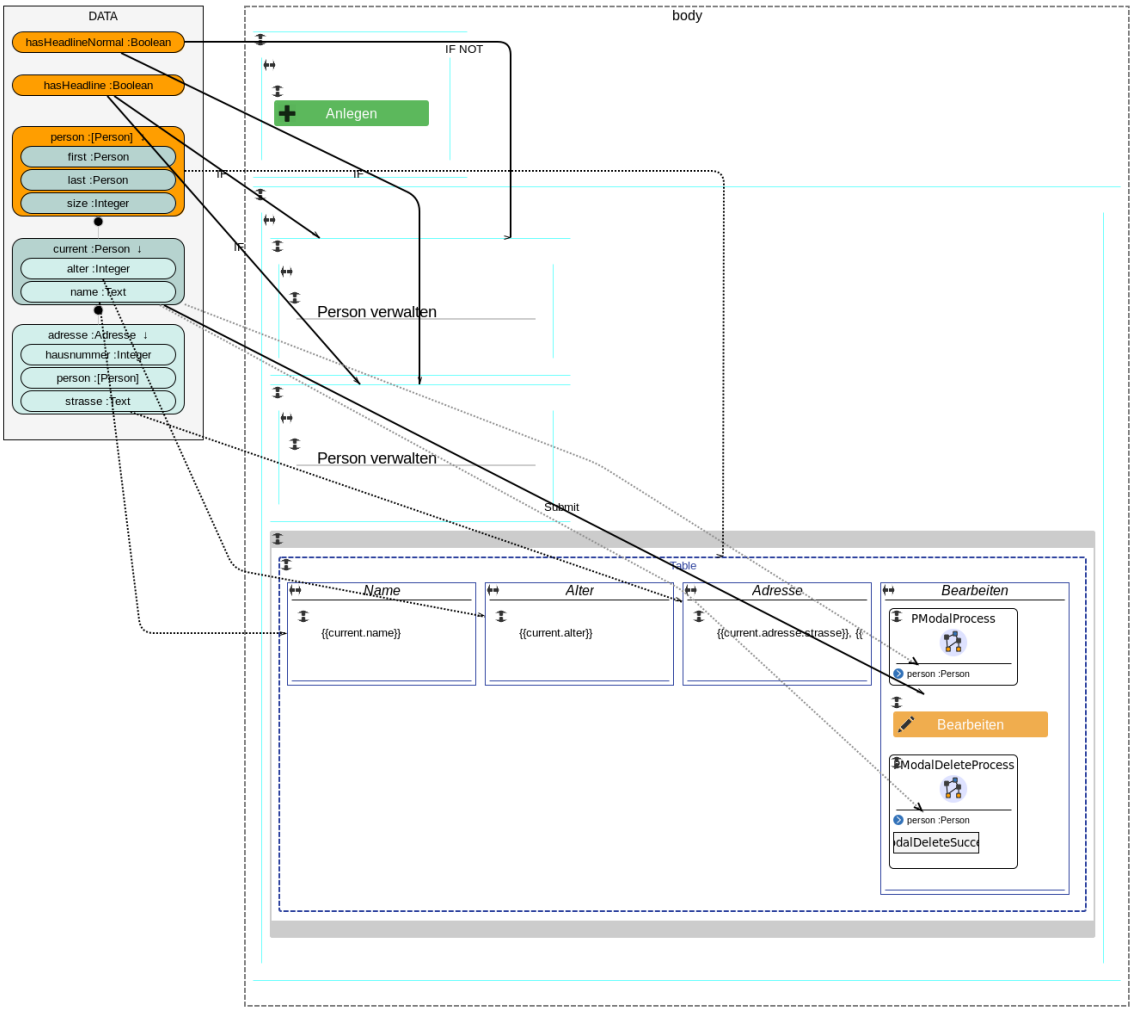


Abbildung 4.4: Verwaltungs-GUI für den Typ *Person*. Dieser besitzt das primitive Attribut *name* und ein komplexes Attribut *Adresse*

4.2.4 OverviewGenerator

Der OverviewGenerator erstellt eine GUI zur Verwaltung der Objekte eines als Parameter übergebenen Typs (siehe Abbildung 4.4).

Als Eingabe erhält die GUI eine Liste von Objekten des angegebenen Typs. Diese sollen in einer Tabelle dargestellt werden.

Um dies umzusetzen, wird die Liste durch eine TableLoad-Kante mit der Tabelle verbunden. Für jedes Attribut des Typs wird ein neuer Tabelleneintrag in die Tabelle eingefügt.

Verschiedene Attributtypen werden auf unterschiedliche Art eingetragen:

- Für primitive Attribute wird der Wert der Variable direkt in den Tabelleneintrag geschrieben.

- Komplexe Attribute besitzen einen vorher festgelegten Identifier. Dieser enthält eine beliebige Anzahl von Attributen des referenzierten Objekts, die in die Tabelle eingetragen werden sollen. Das genaue Vorgehen wird in Kapitel 3.4 beschrieben.

- In den Tabelleneintrag von komplexen Listenattributen wird eine weitere Tabelle gesetzt. In dieser werden alle Objekte der Liste eingetragen. Auch hier geschieht dies über den Identifier des referenzierten Typs.

Alle Varianten haben gemeinsam, dass sie mit einer Iteratorvariable *current* verbunden sind. Durch diese wird über die komplette Liste iteriert und es werden alle Objekte in die Tabelle eingetragen.
Außerdem kann in der DSL eingestellt werden, welche Attribute nicht sichtbar sein sollen. Diese werden dann entsprechend in der Tabelle nicht angezeigt.
In den letzten Tabelleneintrag werden die Buttons zum Bearbeiten, Löschen und dem Anzeigen der Detailansicht der Objekte gelegt. Diesen wird mit einer LoadSubmit-Kante das zu bearbeitende bzw. löschende Objekt übergeben. Ein Knopf zum Erstellen von Objekten wird oberhalb der Tabelle angebracht.
Außerdem gibt es noch die Möglichkeit über die boolean Variablen *hasHeadline* und *hasHeadlineNormal* anzugeben, ob eine Überschrift für die GUI angezeigt werden soll bzw. in welcher Größe sie angezeigt werden soll. Dies wird bei der Erstellung einer *AbstractOverview* genutzt.

4.2.5 DetailViewGenerator

Die DetailView soll ein Objekt mit seinen Attributen genauer darstellen. Dafür werden die primitiven und komplexen Daten des Objektes aufgelistet. Eine solche GUI soll nur generiert werden, wenn im Typ des Objektes der *hasDetailView*-Parameter gesetzt ist. Die weiteren Parameter dieses Typs haben keine Auswirkung auf die Generierung der GUI.
Der Ablauf der Generierung folgt einem simplen Muster:

1. *Initialisierung*: Alle komplexen Attribute des Typs werden dem Datenkontext als Inputs hinzugefügt und anschließend ausgeklappt.

2. *Attributdarstellung*: Für jedes Attribut wird der Seite ein Abschnitt hinzugefügt, welcher dieses genauer darstellt. Dies geschieht nicht, wenn das Attribut leer ist.

3. *Finalisierung*: Der GUI wird oben eine Überschrift hinzugefügt.

Die einzelnen Schritte werden im folgenden Abschnitt genauer erläutert. Außerdem wird die Implementierung der Abläufe vorgestellt.

Listing 4.1: Codefragment zur Generierung der Darstellung von primitiven Attributen

```
//Primitive Attribute Anzeigen
for(pAttr : data.primitiveAttributes){
   if(!pAttr.attribute.isIsList){
      //Inhalt
      val pan = body.rows.head.cols.head.newPanel(0,0)
      pan.newText(0,0) => [
         content.head.rawContent = "{{" + typeName.toFirstLower + "." +
             pAttr.attribute.name + "}}"
      ]
      //Headline
      pan.newHeadline(0,0) => [
         content.head.rawContent = pAttr.attribute.name
         size = HeadlineSize.MEDIUM
      ]
      //IF Kante
      pAttr.newIF(pan)
   }
}
```

Generierungsschritte

Die Initialisierung der View geschieht, indem dem Datenkontext zuerst der Typ hinzugefügt wird, der visualisiert werden soll. Anschließend wird dieser ausgeklappt. Alle komplexen Attribute werden aus dem Type extrahiert und an einem eigenen Platz im Datenkontext platziert. Der Zugriff auf den DIME-Typen und den zugehörigen Typen in der DSL wird somit gewährleistet.
Der zentrale Schritt der Generierung von DetailViews ist die Darstellung der Attribute. Dabei werden sowohl direkte Attribute des Typs, als auch geerbte Attribute eingebunden. Dies läuft iterativ ab, und jedes Attribut wird einzeln verarbeitet. Dabei wird ein Abschnitt der View mit der Darstellung des Attributs ergänzt. Dieser wird abhängig von den Eigenschaften des Attributs unterschiedlich konfiguriert:

1. *Primitive Attribute*: Die primitiven Attribute werden dargestellt, indem eine Überschrift erstellt wird, die die Typbezeichnung des Attributs enthält. Anschließend wird der Wert des Attributs in Textform hinzugefügt. Die Implementierung dieser Funktion ist in dem Codefragment 4.1 dargestellt.

2. *Komplexe Attribute*: Komplexe Attribute werden abhängig von dem *completeDetailViewInformation*-Parameter unterschiedlich dargestellt. Wenn der Wert dieses Parameters auf *false* gesetzt ist, wird das Attribut, ähnlich zu primitiven Attributen, nur mit seinem Namen und Wert dargestellt. Andernfalls wird eine genauere Darstellungsweise generiert. Diese besteht aus allen Subattributen des jeweiligen Attributs, die entsprechend ihrem Typ visualisiert werden. Die Subattribute werden jedoch nur

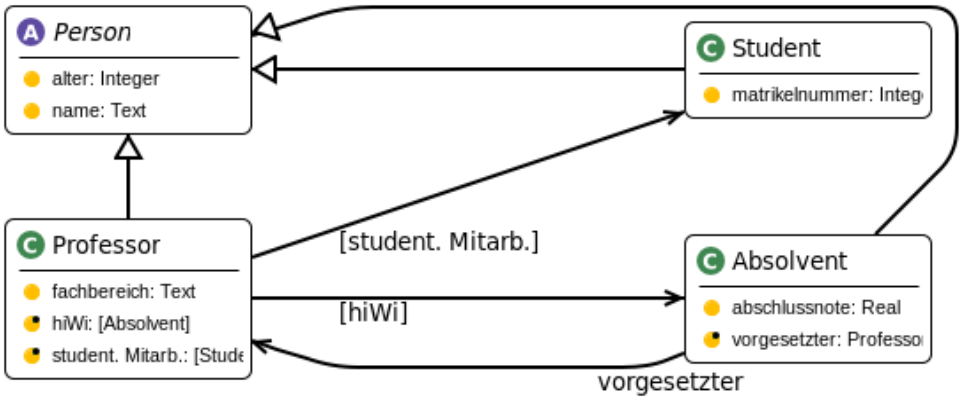


Abbildung 4.5: Beispielhaftes Datenmodell mit komplexen und primitiven Attributen

mit ihrem Identifier dargestellt, da sonst unbegrenzt lange Attributsketten entstehen könnten.

3. *Attributslisten*: Der *view*-Parameter bestimmt, ob eine Attributsliste in Form einer Liste oder in Form einer Tabelle dargestellt werden soll. Die Listendarstellung enthält alle Identifier der einzelnen Attribute. Dafür wird das Attribut im Datenkontext durch eine FOR-Schleife mit einer Listing-Komponente verknüpft. Die Tabellendarstellung enthält weiterführende Informationen, obwohl der Grundaufbau analog ist (das Attribut wird durch eine FOR-Kante mit einer Table-Komponente verknüpft). Innerhalb dieser Tabelle wird für jedes Subattribut, das im Typen der Elemente der Attributsliste enthalten ist und dargestellt werden soll, eine Spalte hinzugefügt. Ein Zugriff auf den Inhalt dieses Subattributs ist nun nach dem Muster

$$\{\{current.subattributname\}\}$$

möglich. Auf diese Weise kann eine Darstellung der Werte erfolgen.

Abschließend wird in dem Darstellungsschritt noch eine IF-Kante von den Attributen zu den zugehörigen Abschnitten hinzugefügt, die den Effekt hat, dass Abschnitte ausgeblendet werden, wenn die betreffenden Attribute leer sind.
Die Finalisierung der View erfolgt, indem der Seite oben eine Überschrift hinzugefügt wird. Dabei werden Name und Identifier des darzustellenden Typs kombiniert. Eine beispielhaft aus dem Datenmodell in Abbildung 4.5 nach dem oben beschriebenen Muster generierte GUI ist in Abbildung 4.6 dargestellt.

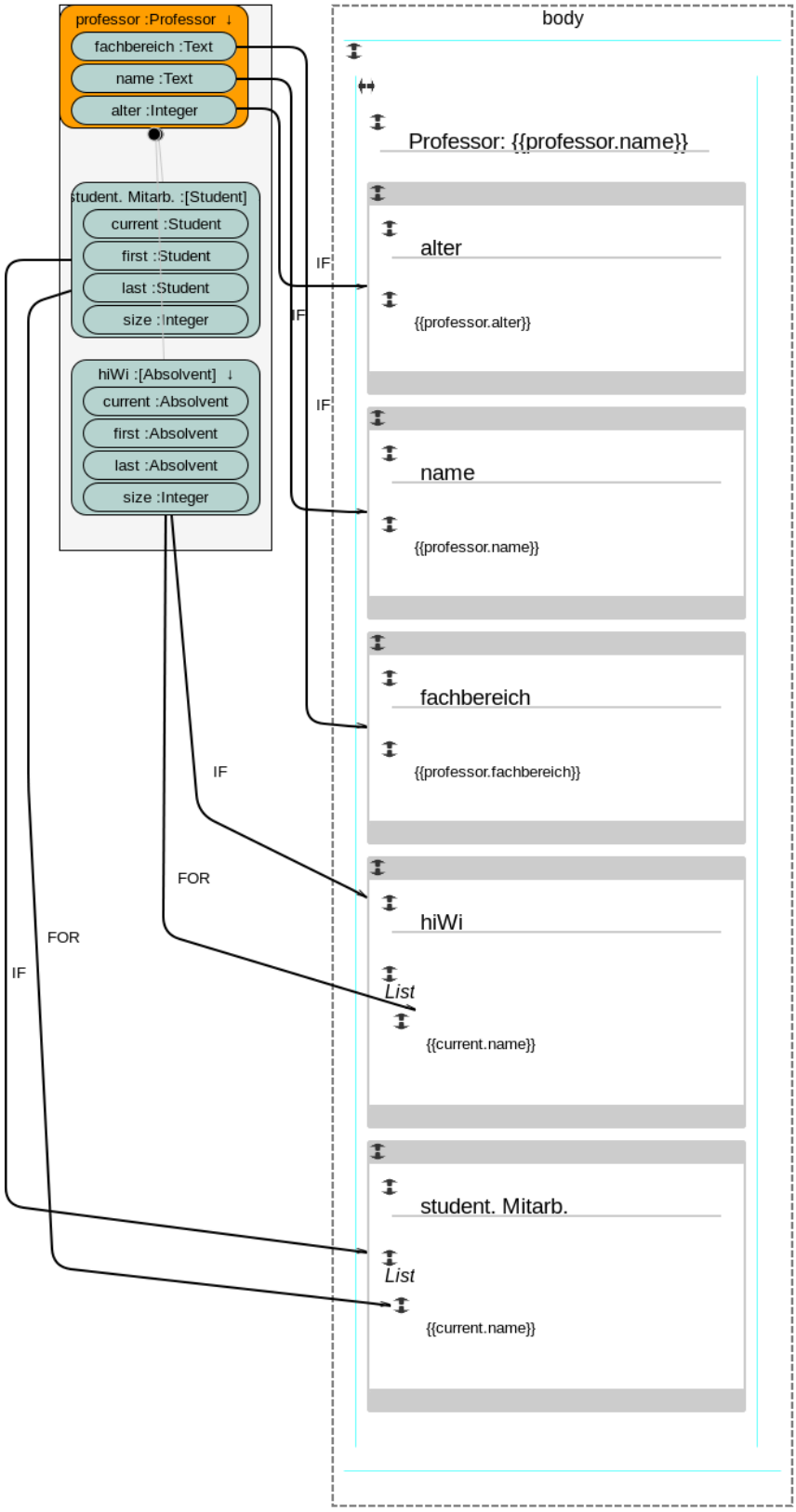


Abbildung 4.6: Die aus dem Datenmodell in Abbildung 4.5 für den Typ *Professor* generierte Detailansicht

4.2.6 EditViewGenerator

Der *EditViewGenerator* stellt eine *EditView-GUI* bereit, in der die Attribute eines Typs bearbeitet oder erstellt werden können.

Besitzt der Typ primitive Attribute, wird für jedes ein Textfeld zum Bearbeiten angelegt und eventuell bereits vorhandene Werte werden beim Aufrufen der GUI in die Textfelder geladen. Falls der Parameter *isRequired* für ein primitives Attribut gesetzt ist, wird dieses farblich hervorgehoben.

Für komplexe Attribute stellt die Ontologie-DSL einen *createEditType*-Parameter bereit, der vier verschiedene Auswahlmöglichkeiten bietet. Diese Auswahlmöglichkeiten bestehen aus *Createable*, *Selectable*, *Viewable* und *Hidden*.

Wird der createEditType auf Createable gesetzt, bindet der EditViewGenerator eine *Overview-GUI* ein, die bereits für diesen Typ erstellte Instanzen des Attributtyps in einer Tabelle auflistet. Weiterhin bietet diese Overview-GUI, mittels eines *Erstellen*-Buttons, die Möglichkeit, eine Instanz des Attributtyps anzulegen. Bei einem Klick auf diesen Button wird der Benutzer auf die EditView-GUI des Attributtyps weitergeleitet. Nachdem der Benutzer die Erstellung beendet hat, wird er wieder auf die EditView-GUI des Haupttyps zurückgeleitet und die erstellte Instanz wird der Tabelle hinzugefügt. Handelt es sich bei dem Attributtyp jedoch um einen Baumtyp, wird hingegen eine *Tree-GUI* eingebunden, in der ein neuer Baum des Attributtyps direkt angelegt werden kann.

Bei Auswahl von Selectable wird, abhängig davon ob der Parameter *tableInsteadOfCheckbox* gesetzt ist und es sich bei dem Attribut um eine Liste handelt, eine Zuweisung von Instanzen zu dem Haupttyp mittels verschiedener GUI-Elemente ermöglicht:

Handelt es sich bei dem Attribut um keine Liste, wird eine *Combobox* bereitgestellt, in der genau eine bestehende Instanz des Attributtyps dem Haupttyp zugewiesen werden kann. Ist das Attribut jedoch eine Liste, bietet die GUI, falls der tableInsteadOfCheckbox-Parameter gesetzt ist, eine Tabelle der bestehenden Instanzen des Attributtyps zur Zuweisung zum Haupttyp an, in der die Attributwerte des Attributtyps aufgelistet werden. Wenn der tableInsteadOfCheckbox-Parameter hingegen nicht gesetzt ist, wird eine Zuweisung mittels *Checkbox* angeboten, die ähnlich der Combobox funktioniert, jedoch eine Mehrfachauswahl ermöglicht.

Alternativ bindet der EditView-Generator jedoch eine *AllocateGUI* ein, wenn es sich bei dem Attributtyp um einen Baumtyp handelt, in der sowohl bereits existierende einzelne Baumblätter, als auch ganze Unterbäume, dem Haupttyp zugewiesen werden können.

Ist der createEditType auf Viewable gesetzt, werden die Attributwerte des Attributs textuell als Liste angezeigt, die lediglich betrachtet, jedoch nicht verändert werden kann. Hidden blendet das Attribut gänzlich aus, es kann weder betrachtet, noch bearbeitet werden.

In der Initialisierungsphase werden im Datenkontext der GUI komplexe Variablen für den angegebenen Haupttyp, sowie, falls nötig, für seine komplexen Attribute, erzeugt.

Nun folgt in der Generierungsphase die Erzeugung der eigentlichen Bearbeitungs-Ansicht. Dafür wird in der GUI eine Form erzeugt und für jedes komplexe Attribut, sowie für die primitiven Attribute des Haupttyps, ein Panel eingefügt. In diese Panel wird dann die im createEditType angegebene Bearbeitungsmöglichkeit erzeugt. Am unteren Ende der GUI gibt es abschließend zwei Buttons. Einen "Speichern"-Button, um das Bearbeitete zu sichern und einen "Zurück"-Button, um das Bearbeiten abzubrechen.
Gesetzt den Fall, dass bei mindestens einem der primitiven Attribute des Haupttyps einer der Parameter *isRequired* oder *isUnique* gesetzt ist, findet abschließend noch eine Validierung statt:
Ist für ein Attribut der Parameter isRequired gesetzt und im entsprechenden Textfeld befindet sich noch keine Eingabe, wird ein Klick auf "Speichern" untersagt und der Button wird ausgegraut, bis eine Eingabe erfolgt ist. Wenn hingegen der Parameter isUnique gesetzt ist, wird, falls der Wert im jeweiligen Textfeld bereits in irgendeiner anderen Instanz des Haupttyps existiert, eine Fehlermeldung eingeblendet und der Benutzer kann die GUI solange nicht verlassen, bis der Fehler behoben wurde.

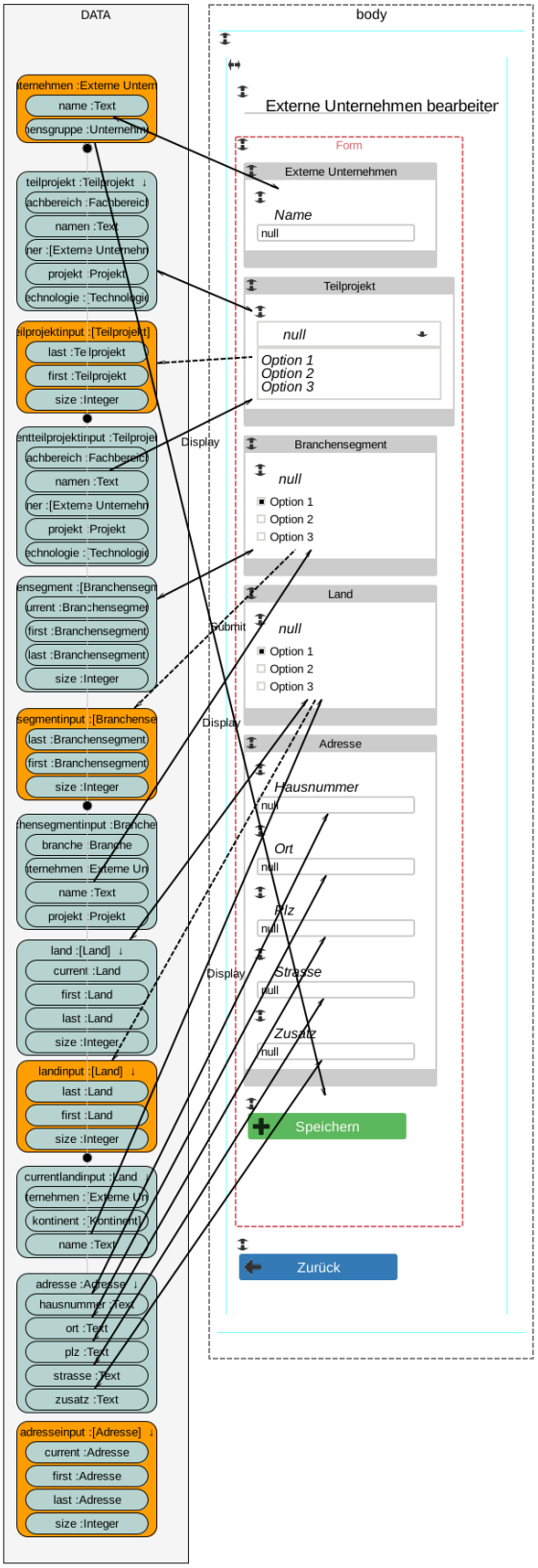


Abbildung 4.7: Eine Beispielgenerierung des EditViewGenerators anhand des Typs "Externe Unternehmen" der Schulz-Ontologie

4.2.7 Baumstrukturen

Baumverwalten - GUI

Damit die komplexen Vorgänge der Baumhandhabung bewerkstelligt werden können, bestehen die Baumverwaltung und die Baumauswahl aus sehr vielen einzelnen Prozessen. Diese werden im Folgenden aufgelistet und kurz beschrieben.

- *TreeGUIGenerator*
  Dieser Generator erzeugt eine Verwaltungsansicht über den kompletten Baum. Dafür wird die entsprechende ParentViewGUI eingebunden. Ein Button ermöglicht es, neue Objekte der obersten Baumschicht anzulegen. In Kapitel 3.3.2 wird dies genauer beschrieben.

- *TreeEditViewGenerator*
  Erzeugt eine GUI zum Anlegen bzw. Bearbeiten von Baumobjekten.

- *TreeViewGUIGenerator*
  Übersicht über das aktuelle Baumelement. Die erzeugte GUI beinhaltet sich selbst, um rekusiv alle darunter liegenden Baumschichten anzeigen zu können. Durch einen Button lässt sich das Baumelement bearbeiten. Auch diese GUI wird in Kapitel 3.3.2 genauer beschrieben.

- *ParentViewGUIGenerator*
  Übersicht über das aktuelle Baumelement. Die erzeugte GUI beinhaltet die TreeViewGUI, um rekusiv alle darunter liegenden Baumschichten anzeigen zu können. Durch einen Button lässt sich das Baumelement bearbeiten.

- *PathViewGUIGenerator*
  Baut aus einer Liste von Bäumen den Pfad, hierfür wird für jedes Element ein Button generiert.

- *TreeOverviewGUIGenerator*
  Erzeugt eine Verwaltungsansicht für ein Baumobjekt, in der alle dazugehörigen Attribute aufgeführt werden. Mithilfe von Buttons kann man das Objekt löschen, bearbeiten oder dem Objekt neue Kinder hinzufügen. Für das Anzeigen des Pfades wird die entsprechende PathViewGUI eingebunden, für das Anzeigen des Baumes die TreeViewGUI.

Baumverwalten - Process

- *CRUDTreeProcessGenerator*
  Generiert den Verwaltungsprozess eines Baumes. Der generierte Prozess holt sich mithilfe des zugehörigen GetAllParentsProcess und InvertListTreeProcess die obersten

Knoten des Baumes. Diese werden für die TreeOverviewGUI benötigt. Die Branches des TreeOverViewGUI führen zu den Prozessen: CreateProcess, EditViewProcess, RetrieveProcess und RemoveProcess. Das die Verwaltung rekusiv ist, wird ebenfalls wieder der CRUDTreeProcess benötigt.

- *CreateProcessGenerator*
  Ist für die Erstellung eines neuen Baumelements zuständig und fängt die Nutzereingaben ab, falls das Erstellen abgebrochen wird.

- *GetAllParentsProcess*
  Benutzt sich selbst und den GetParentProcess, um rekusiv alle Eltern bis zur Wurzel eines Baumobjektes zurückgeben zu können.

- *GetParentProcessGenerator*
  Liefert den Elternknoten eines Kindes.

- *GetRootNodesProcessGenerator*
  Überprüft, ob eine Liste von Baumobjekten Elternobjekte besitzen, um dadurch die Wurzelelemente des Baumes zu erhalten. Hierfür wird der HasParentProcess eingebunden. Auf diesen Prozess wird in Kapitel 3.3.3 genauer eingegangen.

- *HasParentProcessGenerator*
  Überprüft mithilfe des GetParentProcess, ob ein Elternknoten existiert.

- *InvertListTreeProcessGenerator*
  Invertiert eine Liste von Baumobjekten. Wird auch in Kapitel 3.3.3 nochmal vorgestellt.

- *IsFirstChildProcessGenerator*
  Sorgt bei der Erstellung eines neuen Baumelements dafür, dass seine Eltern in einem validen Zustand bleiben.

- *RemoveChildrenProcessGenerator und RemoveProcessGenerator*
  Rufen sich gegenseitig auf, um von einem Baumobjekt rekusiv alle Kinder zu entfernen.

- *RemoveParentsRefsProcessGenerator*
  Löscht ein komplexes Attribut aus einem Baumobjekt.

Baumauswahl - GUI

- *AllocationGUIGenerator*
  Erzeugt die GUI, welche die gesamte Baumauswahl handhabt. Dazu gehört das an- und abwählen von Baumelementen sowie das navigieren durch den Baum.

Baumauswahl - Process

- *AddTreeGenerator*
  Sorgt dafür, dass ein Baumelement in der Datenbank mit dem Besitzertyp verknüpft wird und setzt die Attribute der Helferobjekte

- *AllocateTreesToTreeowner*
  Der Einstiegsprozess für die Baumauswahl. Stößt das Löschen von alten Helfern an.

- *AssignPriorityProcess*
  Weißt jedem Helferobjekt ein Priorität für die danach folgende Suche zu.

- *CheckIfParentsIncludedGenerator*
  Wird nach dem Abwählen eines Baumelements angestoßen, um die Baumauswahl in einem validen Zustand zu halten.

- *CreateHelperGenerator*
  Erzeugt die Helferobjekte und weißt ihnen alle Baumelemente zu. Dies wird in Kapitel 3.3.5 genauer betrachtet.

- *DeleteHelperProcessGenerator*
  Löscht alle Helferobjekte aus der Datenbank.

- *GetHelperParentsProcessGenerator*
  Sucht die Eltern eines Baumelementes bei einer Auswählaktion heraus.

- *HelperHasChildsProcessGenerator*
  Überprüft bei der Prioritätsvergabe, ob die Kinder eines Baumelementes ausgewählt wurden.

- *RemoveTreeGenerator*
  Führt die Helfer- und Datenbankmanipulation beim Abwählen für das abgewählte Baumelement durch. Hierauf wird in Kapitel 3.3.5 nochmal eingegangen.

- *RemoveTreeParentsGenerator*
  Führt die Helfer- und Datenbankmanipulation beim Abwählen für die Eltern des abgewählten Baumelements durch.

- *SetChildsIncludedGenerator*
  Füllt die Helferobjekte mit Informationen über bereits ausgewählte Kinder der Baumelemente.

- *SetHelperProcessGenerator*
  Koordiniert das Anlegen, Initialiseren und Sortieren der Helferobjekte.

- *SetIncludedProcessGenerator*
  Füllt die Helferobjekte mit Informationen über bereits ausgewählte Baumelemente.

- *SortSiblingsGenerator*
  Sortiert eine Liste von Baumelementen mit dem gleichen Elter. Dieser komplizierte Prozess wird in Kapitel 3.3.5 detailliert vorgestellt.

- *StartSortingGenerator*
  Koordiniert die Sortierung der Baumelemente.

- *TreeAllocationProcess*
  Koordiniert die Handhabung der Nutzereingaben.

Kapitel 5

Konzepte Canvas

5.1 Canvas

In Kapitel 2.6 wird erläutert, was genau ein Canvas darstellt und wie dieser in der Umgebung einer Firma, bzw. einer Organisation eingesetzt werden kann. Dieses Kapitel soll nun darauf eingehen, wie die Ideen, die solch einen Canvas ausmachen, umgesetzt werden, um daraus einen in Form einer Webapp ausfüllbaren Canvas zu gestalten.

5.1.1 Aufbau Canvasmodell

Um eine Webapp aufbauen zu können, ist es notwendig, dass Pattern bekannt sind, die generierbar sind, um daraus eine solche Anwendung erstellen zu können. Um sich ein Bild von diesen Pattern zu machen, wurde vor dem Implementieren von Generatoren, eine Webapp gebaut, die ein konkretes Beispiel eines Canvas abbildet. Dieser Canvas bildet einen Prototypen, der ohne vorherige Generierung auskommt, der allerdings schon alle Funktionalitäten beinhaltet, die auch in der später generierten Anwendung vorhanden sein sollen. Wie auch schon bei dem Prototypen, der für die Ontologie erstellt wurde, dient dieser des Canvas dafür, Pattern zu bestimmen und zu extrahieren. Diese Pattern werden im Zuge der später implementierten Generatoren nachgebildet, bzw. durch die Generatoren erstellt. Somit bieten die Pattern einen Leitfaden, an dem sich die Erstellung der Generatoren orientiert. Zu jedem Zeitpunkt der Erstellung der Generatoren, kann das Ergebnis eines Generators mit dem Zielbild des durch den Generator gebildeten Patterns abgeglichen werden. Somit kann sichergestellt werden, dass die generierte Anwendung den Funktionsumfang des Prototypen abbildet und die gleichen Schritte durchläuft, um Funktionen umzusetzen.

5.1.2 Prototyp

Der modellierte Prototyp eines Canvas, ist in Abbildung 5.1 dargestellt. Dieser beinhaltet alle Funktionen, die im Laufe der Zeit erarbeitet wurden. Zu erkennen ist, dass dieser sich an dem Business Model Canvas nach Osterwalder orientiert, wie er bereits in Kapitel 2.6.2

erläutert wird. Die 9 Felder des Business Model Canvas sind mit ihren ursprünglichen Bezeichnungen vorhanden und auch so angeordnet, dass die Reihen und Spalten erkennbar sind. Darüber hinaus wurden sich Gedanken dazu gemacht, welche Funktionen eine Canvas die Anwendung erfüllen soll. Dabei geht es vor allem um Funktionen, die ein auf Papier ausgefüllter Canvas Modell nicht bieten kann.
So wurde sich darauf geeinigt, dass der hier vorgestellte Canvas eine *User Guidance* enthalten soll. Das bedeutet, dass der Benutzer der Anwendung durch seine Eingaben durch die Ausfüllung des Canvas begleitet wird. Das Ziel dahinter ist, dass der Anwender auch mit vergleichsweise wenig Erfahrung auf dem Gebiet eines Canvas, in die Lage versetzt wird, die richtigen Eingaben in den entsprechenden Feldern zu tätigen. An dem Beispiel in 5.1 lässt sich das so erklären: In dem Feld *ValueProposition* können Projekte eingetragen werden, in dem Feld *CustomerSegments* entsprechend Endkunden. Wird der Canvas jetzt in der Reihenfolge ausgefüllt, dass zuerst ein Projekt angewählt wird, so können daraufhin nur noch diejenigen Endkunden in das dafür vorgesehene Feld eingetragen werden, die zu dem vorher ausgewählten Projekt gehören. So verhält es sich bei jeder beliebigen Reihenfolge, die von einem Anwender beim Ausfüllen das Canvas genutzt wird.

Die Pattern, die dazu verwendet werden, diesen Prototypen zu erzeugen, werden im Folgenden einzeln aufgeführt.

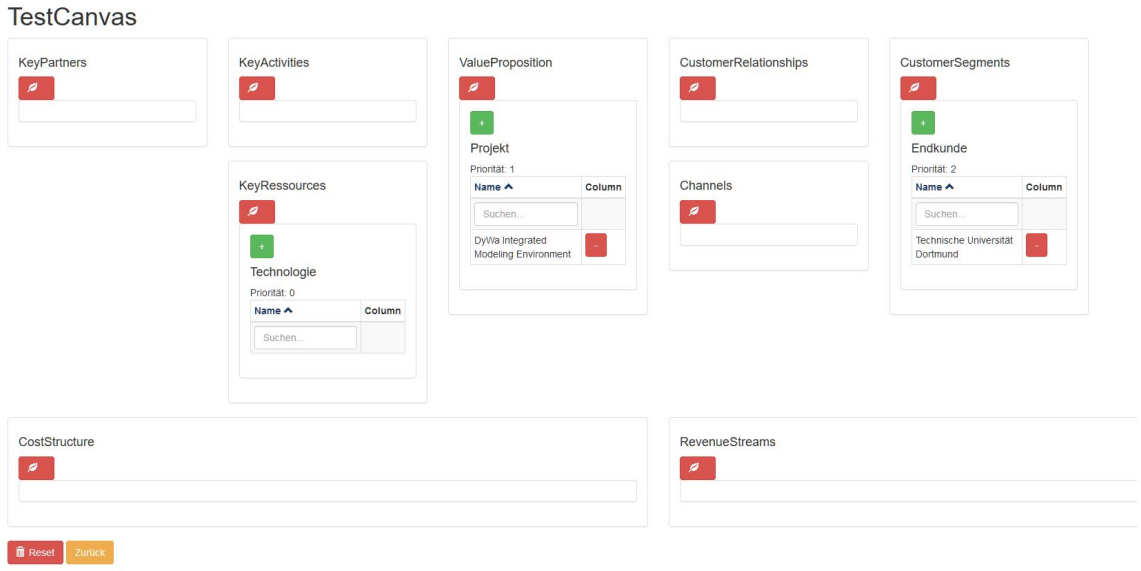


Abbildung 5.1: Canvas Prototyp mit vollem Funktionsumfang

5.2 Pattern

5.2.1 Prozess Pattern (Elementtypen)

Der Canvas bedient sich einer Vielzahl von Pattern. Elemente werden im Folgenden definiert als beliebige Eintragung in der Ontologie, die zu einem Elementtyp, wie bspw. Projekten, Endkunden oder Technologien, gehören. Alle, in diesem Unterkapitel erläuterten Pattern, müssen für jeden, in der Ontologie genutzten Elementtyp angelegt werden. Diese Elementtypen werden in dem Prozessnamen als "XX" und "YY" symbolisiert.
Das *Management* Pattern ist der Hauptprozess der Beeinflussungen im Canvas. Er wird nach dem Hinzufügen und Löschen von Elementen aus der jeweiligen Auswahl gestartet, um die neuen gültigen und damit zur Auswahl stehenden Elemente der anderen Elementtypen einzuschränken oder zu erweitern. In den zugehörigen Prozessen *Add_ Interact* und *Sub_ Interact* werden dazu Prioritätswerte vergeben und verwaltet, sowie die neue Auswahl an zur Verfügung stehenden Elementen in den Canvas eingepflegt.
In den primitiven Pattern finden sich die dazu genutzten Hilfsprozesse.

primitive Pattern

Aktualisierung: *XX_ aktualisieren*
In den Prozessen im Canvas werden die verwendbar Listen neu erzeugt. So kann die verwendbar Liste Elemente enthalten, die bereits ausgewählt wurden und gleichzeitig können Elemente für eine Auswahl ungültig werden, aber gleichzeitig in der Auswahlliste vorhanden sein. Der Aktualisierungsprozess nimmt eine Neuverteilung zwischen den beiden Listen vor. Als Eingabe bekommt der Prozess den Canvas übertragen. Zu diesem Zeitpunkt enthält die jeweilige Auswahlliste die veralteten Auswahlen und die verwendbar Liste alle gültigen Elemente. Der Prozess iteriert, wie in Abbildung 5.2 zu sehen, über alle Elemente der Auswahlliste des jeweiligen Elementtyps und überprüft für jedes Element, ob es in der verwendbar Liste vorhanden ist. Ist das Element in der verwendbar Liste vorhanden, so ist es gültig und wird aus der verwendbar Liste entfernt. Im anderen Fall ist es nicht mehr gültig und muss aus der Auswahlliste gelöscht werden.

Vereinigung zweier Listen: *XX_ AddList*
Dieser Prozess vereinigt zwei Listen desselben Elementtyps, ohne Dopplungen zuzulassen. Hierzu werden, wie in Abbildung 5.3 zu sehen, die Elemente der zweiten Liste der ersten hinzugefügt, wenn sie nicht bereits in dieser enthalten sind.

Schnittmenge zweier Listen: *XX_ AddListIntersection*
Dieser Prozess bildet die Schnittmenge der beiden übergebenen Listen. Wie in Abbildung 5.4 zu sehen, werden diese einer neuen Liste nur zugewiesen, wenn sie in beiden Listen vorhanden sind.

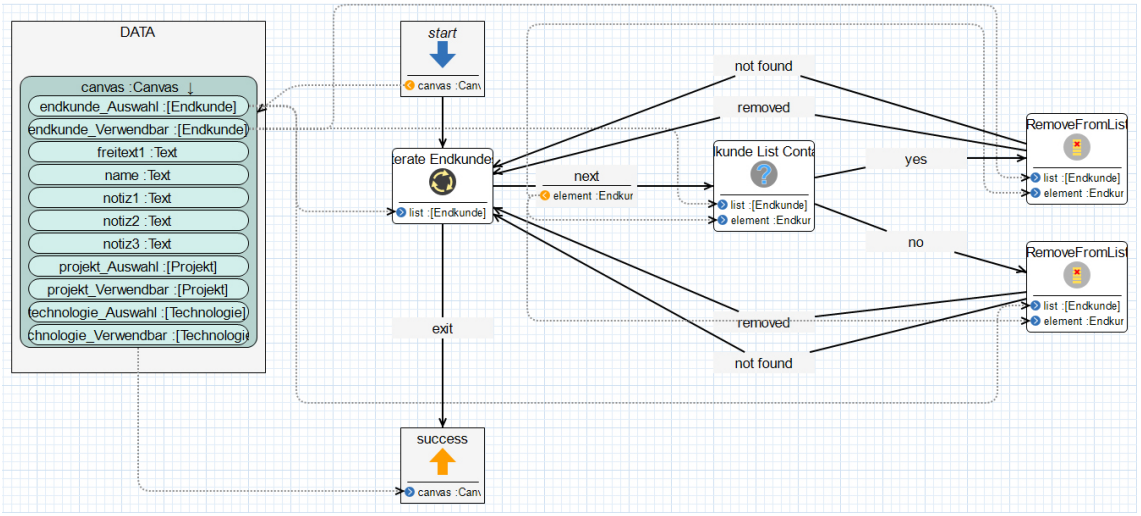


Abbildung 5.2: Canvas Aktualisieren Pattern (Endkunde) ANMERKUNG: Die Verbindungen im Bild werden noch bearbeitet

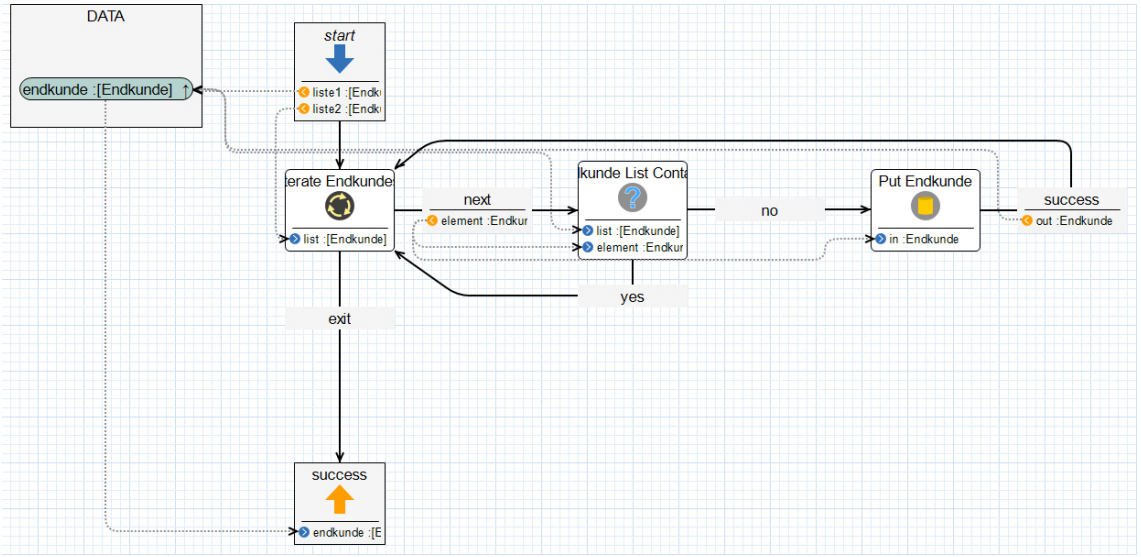


Abbildung 5.3: Canvas Vereinigung zweier Listen (Endkunde) ANMERKUNG: Die Verbindungen im Bild werden noch bearbeitet

Modalansichten Hinzufügen: *XX_ AddModal*

Dieser Prozess führt, analog zu den Modalansichts-Pattern der Ontologie, die Modalansichts-GUI zur Auswahl der hinzuzufügenden Elemente aus. Nach der Auswahl wird das ausgewählte Element, wie in Abbildung 5.5, der jeweiligen Auswahlliste hinzugefügt und aus der jeweiligen verwendbar Liste entfernt.

Duplikate entfernen: *XX_ remove_ dublicates*

Dieser Prozess entfernt Duplikate aus der jeweiligen Liste. Der Prozess iteriert, wie in Abbildung 5.6 zu sehen, über die übergebene Liste und sortiert die jeweiligen Elemente nur ein, wenn sie nicht bereits in der neuen Liste enthalten sind.

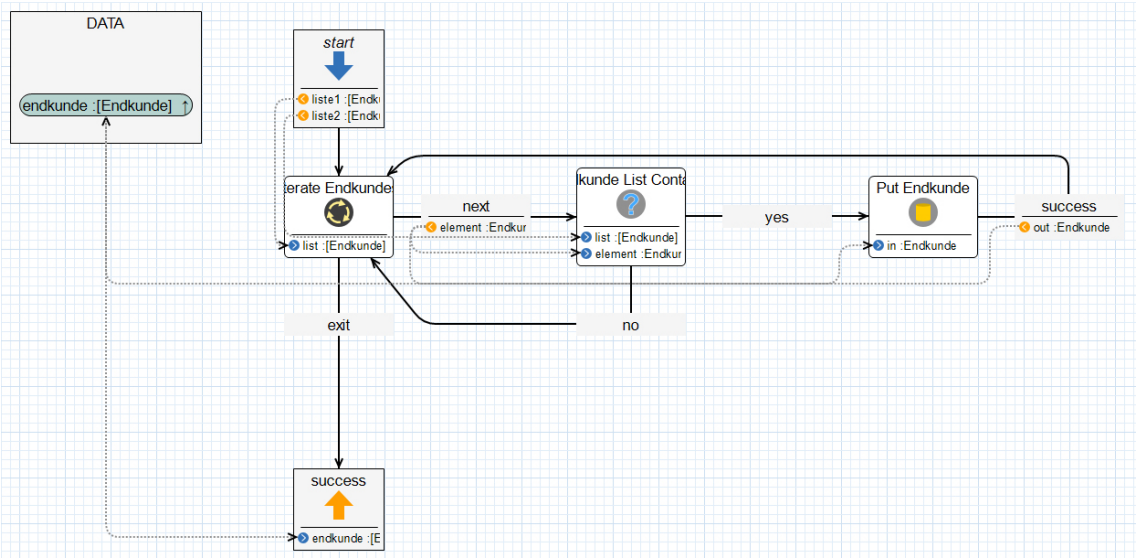


Abbildung 5.4: Canvas Schnittmenge zweier Listen (Endkunde) ANMERKUNG: Die Verbindungen im Bild werden noch bearbeitet

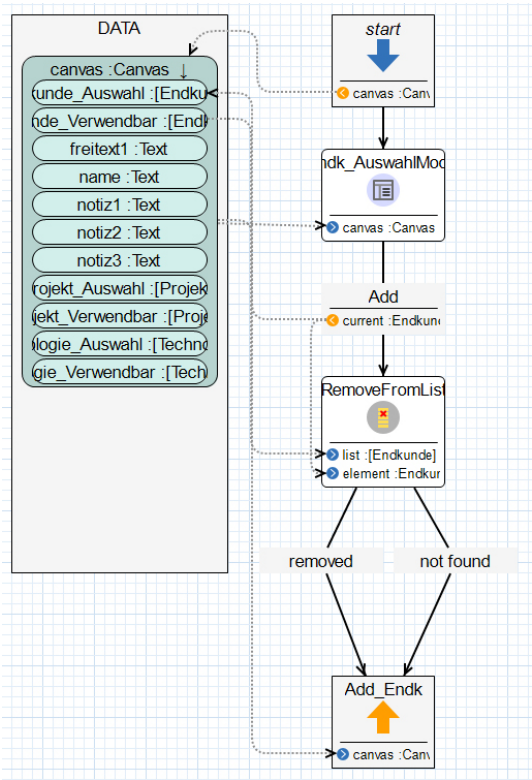


Abbildung 5.5: Canvas Modalansicht Hinzufügen (Endkunde) ANMERKUNG: Die Verbindungen im Bild werden noch bearbeitet

Sub Prozess: *XX_ SubProcess*

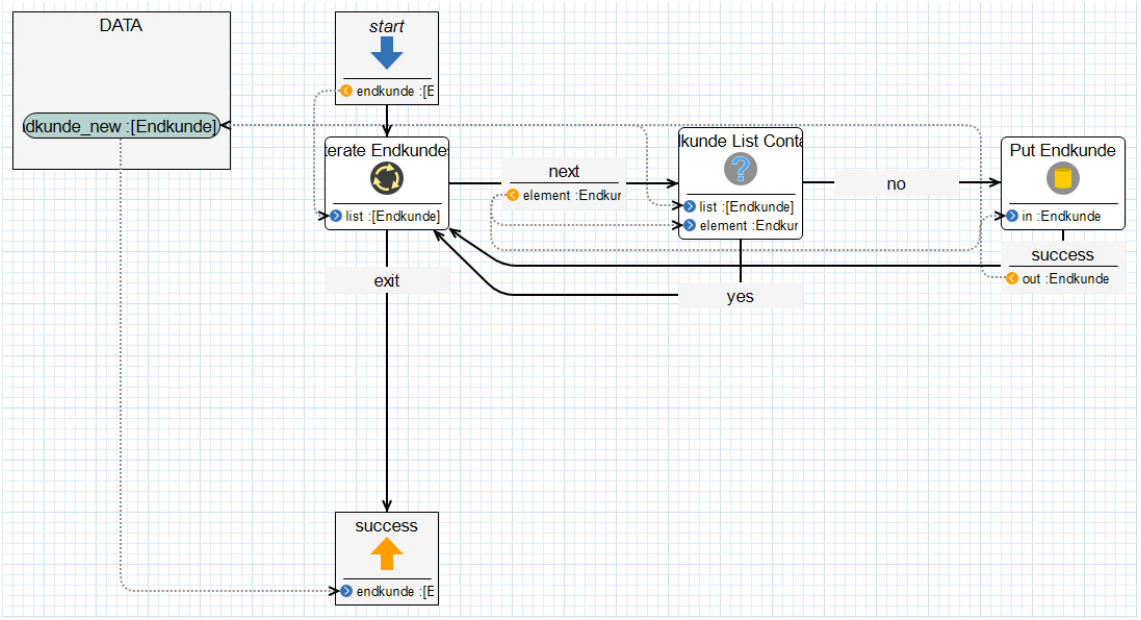


Abbildung 5.6: Canvas Dublikate entfernen (Endkunde) ANMERKUNG: Die Verbindungen im Bild werden noch bearbeitet

Da in der GUI keine Modalansicht vorhanden ist, übernimmt dieser Prozess die Nachbearbeitung nach der Auswahl des zu löschenden Elements, analog zu dem Prozess der Modalansicht für das Hinzufügen von Elementen zu der Auswahl.

Such-Prozesse zweier Elementtypen: *XX_Search_YY*

Die Such-Prozesse geben die Elemente zurück, zu denen das übergebene Element eine direkte Verbindung hat. Wie bspw. in Abbildung 5.7 zu sehen ist, wird durch den Suchprozess vom Endkunden zum Projekt direkt die Liste an Projekten zurückgegeben, mit denen der Endkunde in Verbindung steht.

Komplexe Pattern

Suche verbundener Elementtypen: *XX_getAll*

Der *getAll* Prozess nutzt die primitiven Such-Prozesse, um alle Elemente, mit denen das übergebene Element in Verbindung steht, zu finden und zurück zu geben. Im Fall der beispielhaften Umsetzung (siehe Abbildung 5.8) steht der Endkunde direkt mit Projekten und indirekt mit Technologien in Verbindung. Durch den primitiven Suchprozess wird die Liste an Projekten ermittelt. Für jedes dieser Projekte werden die in Verbindung stehenden Teilprojekte zuerst über eine primitive Suche ermittelt und im Folgenden zu jedem der gefundenen Teilprojekte über eine weitere primitive Suche die in Verbindung stehenden Technologien. Diese Technologien werden über den Vereinigungsprozess disjunkt in einer neuen Liste gespeichert.

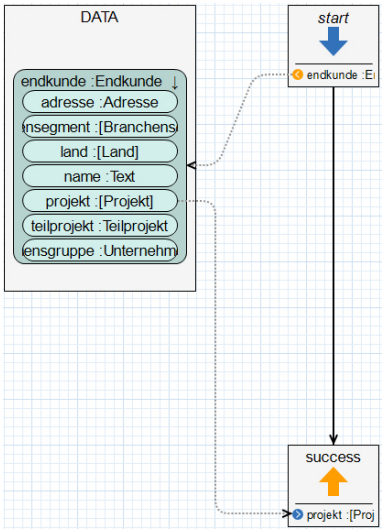


Abbildung 5.7: Canvas Suche verbundene Projekte zu dem Endkunden

Management: *XX_ Management*

Der *Management* Prozess ist der zentrale Prozess im Canvas, der die Beeinflussung nach dem Hinzufügen oder Entfernen eines einzelnen Elementes auf die Listen der anderen Elementtypen umsetzt. Im Folgenden bezeichnet Aktion das Hinzufügen oder Entfernen eines einzelnen Elementes. Zu Beginn des Prozesses wurde die jeweilige Aktion bereits durchgeführt und die Prioritäten darauf basierend angepasst. Das betreffende Element wurde bereits verschoben, die Listen der anderen Elemente aber noch nicht angepasst. Ziel ist es, jeweils eine neue verwendbar Liste zu den anderen Elementtypen zu erzeugen, die die Elemente beinhalten, die nach der durchgeführten Aktion noch gültig für eine Auswahl sind, ohne dabei die bereits getätigten Auswahlen des entsprechenden Elementtyps zu betrachten. Dies wird im Prozess durch erneutes Auslesen und Befüllen von jeweils einer neuen verwendbar Liste umgesetzt. Die verschiedenen zu betrachtenen Fälle (siehe Abbildung 5.9) werden zeilenweise nach vergebenem Prioritätswert unterschieden. Für den Fall, dass der Prioritätswert 1 ist, wurde die höchste Priorität vergeben und es werden die neuen Listen aus dem *getAll* Prozess gewonnen. Im Anschluss müssen die weiteren Prioritäten betrachtet werden, um die weiteren Einschränkungen umzusetzen, falls weitere Auswahlen in den anderen Elementtypen bereits vorgenommen wurden. Dies geschieht durch einen rekursiven Aufruf des *Management* Prozesses für die entsprechenden Elementtypen. Von dieser Überprüfung ausgenommen ist die niedrigste Priorität. Für alle weiteren Fälle wird ebenfalls zuerst der jeweilige *getAll* Prozess aufgerufen, die Ergebnisse aber nicht für Elementtypen übernommen, die einen niedrigeren Prioritätswert haben, ausgenommen der 0. Da in dieser Beispielentwicklung nur drei Prioritäten existieren, muss an dieser Stelle keine weitere Überprüfung stattfinden. Sollten weitere Prioritäten existieren, muss analog zum ersten Fall im Nachhinein die weitere Beeinflussung überprüft werden.

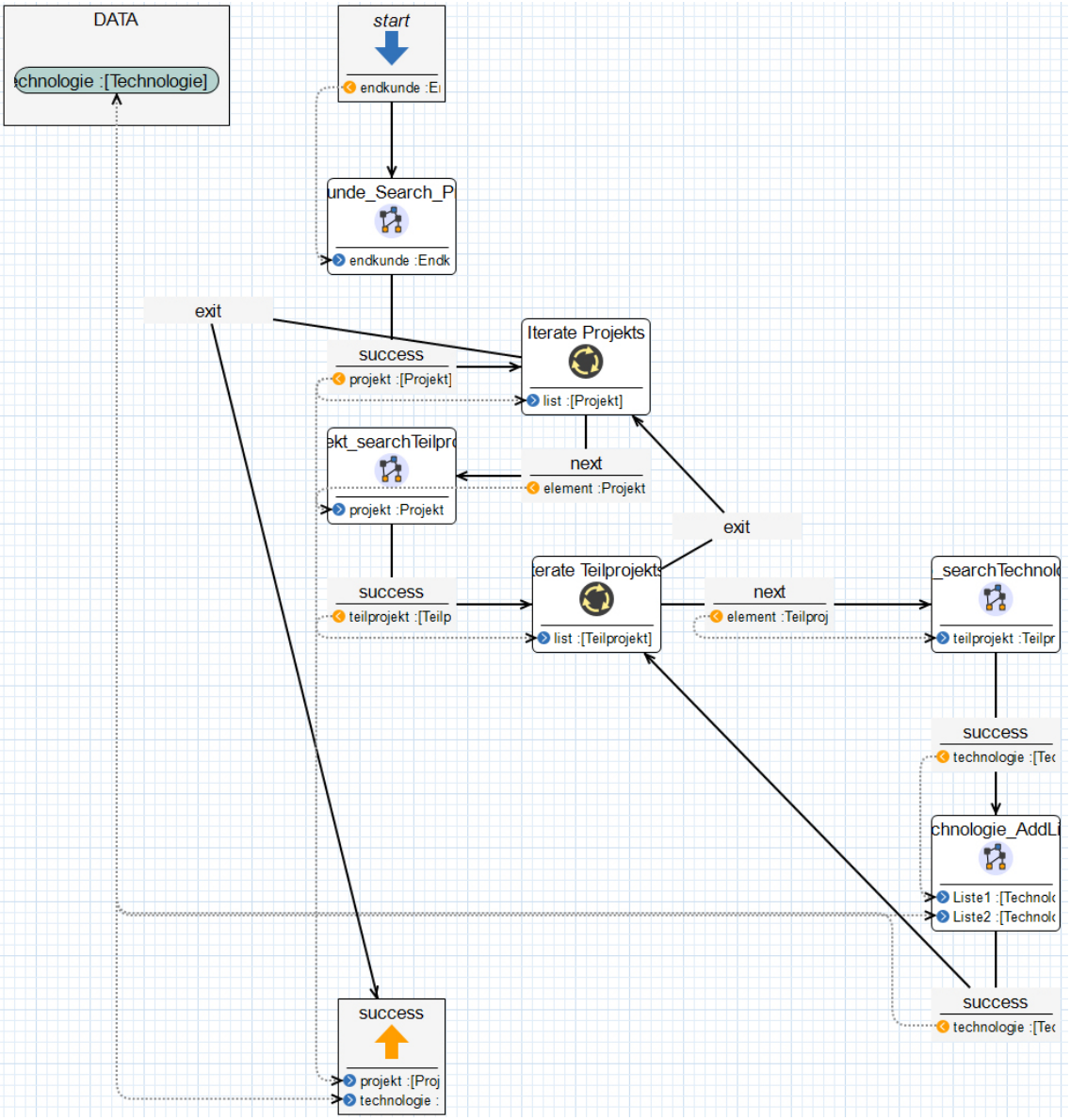


Abbildung 5.8: Canvas Suche alle verbundenen Elemente zu dem Endkunden

Um den Fall zu unterscheiden, dass keine gültigen Elemente mehr in der verwendbar Liste sind und dem Fall, dass die verwendbar Liste nie beschrieben wurde, wurde zu jeder Liste ein Boolean eingeführt mit der Endung "write". Dieser wird nach jedem Überschreiben der Liste auf Wahr gesetzt, was auch den Fall einer leeren Liste abdeckt.

Beeinflussung nach dem Hinzufügen: *XX_Add_Interact*

Dieser Prozess dient der Verwaltung der Prioritätswerte nach dem Hinzufügen eines Elementes zu der jeweiligen Auswahl. Zur Verwaltung der Prioritäten müssen nach dem Hinzufügen eines Elements zur Auswahl zwei Fälle unterschieden werden. Ist der Prioritätswert nicht 0, so wurde eine passende Priorität bereits vergeben, bzw. vorher ein Element des jew.

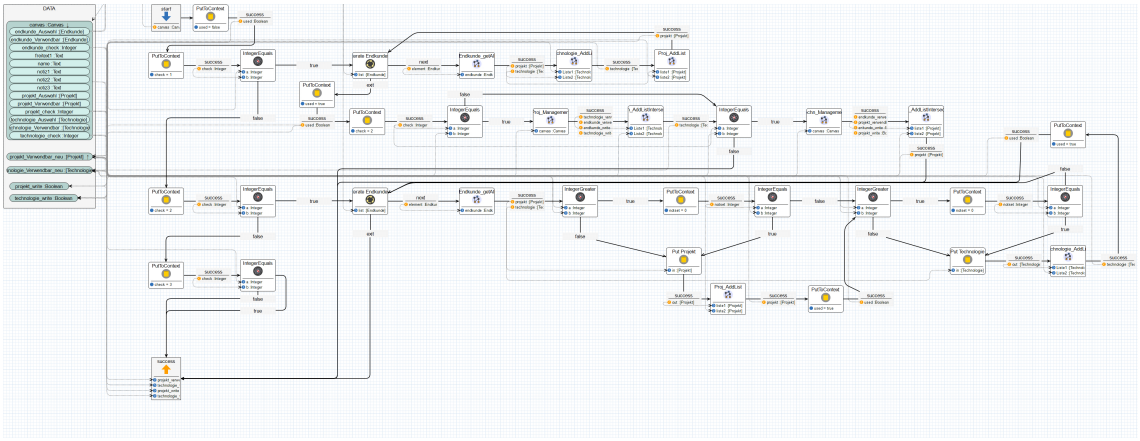


Abbildung 5.9: Canvas Management Prozess (Endkunde)

Elementtyps zur Auswahl hinzugefügt und es muss keine Priorität vergeben werden. Falls der Prioritätswert 0 ist, muss die Priorität neu vergeben werden. Hierzu wird zuerst der maximale, bereits vergebene Prioritätswert im Canvas gesucht. Der neue Prioritätswert ist dann der um eins erhöhte, bisherige Maximalwert. Wie in Abbildung 5.10 zu sehen, geschieht dies im ersten Schritt des Prozesses.

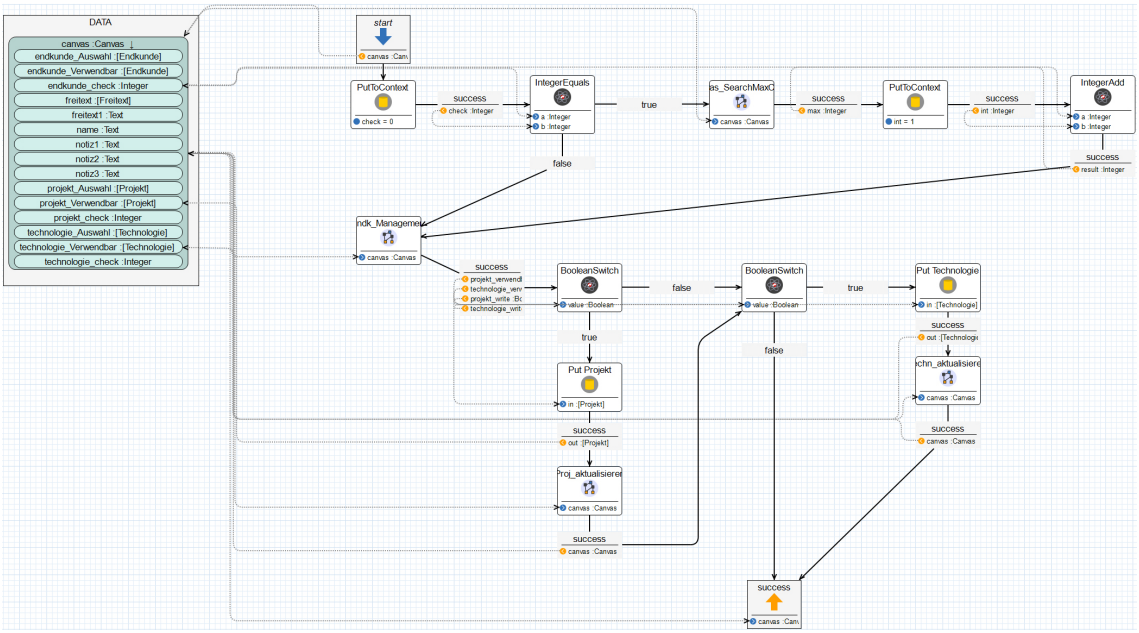


Abbildung 5.10: Canvas Beeinflussung nach dem Hinzufügen eines Endkunden

Nachdem der Prioritätswert gesetzt wurde werden mithilfe des *Management* Prozesses die neuen gültigen Elemente für jeden weiteren Elementtyp berechnet und in den verwendbar Listen zurück gegeben. Wurden diese im Laufe des Prozesses erzeugt, werden sie in den Canvas geschrieben. Mit des jeweiligen *aktualisieren* Prozesses wird dann die Verteilung der verwendbar- und Auswahllisten aktualisiert.

Beeinflussung nach dem Entfernen: *XX_Sub_Interact*

Dieser Prozess dient der Verwaltung der Prioritätswerte nach dem Entfernen eines Elementes aus der jeweiligen Auswahl.

Analog zu dem *interact* Prozess nach dem Hinzufügen werden zuerst die Prioritätswerte überprüft und angepasst. Für den Fall, dass nach dem Entfernen die Auswahlliste nicht leer ist, müssen die Prioritätswerte nicht angepasst werden und es werden, analog zu *Add_Interact*, mithilfe des entsprechenden *Management* Prozesses die Listen der anderen Elementtypen aktualisiert. Ist die Auswahlliste leer, werden die Prioritäten angepasst. Die Prioritäten Vergabe ist in Abbildung 5.11 dargestellt. Der Prioritätswert des

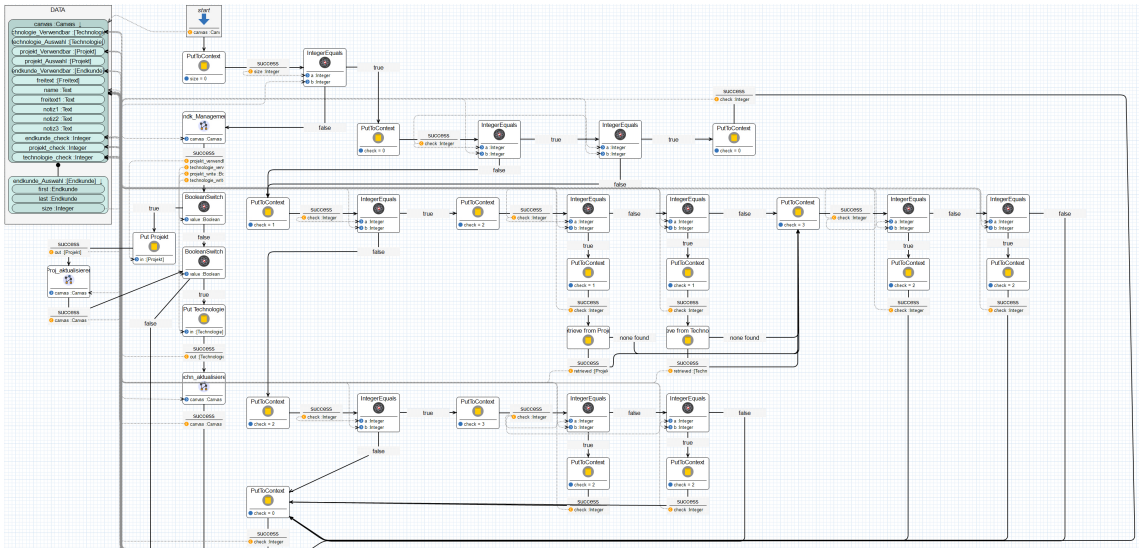


Abbildung 5.11: Canvas *XX_Sub_Interact* Prioritätenvergabe (Endkunde)

Elementtyps, der gelöscht wurde, wird wieder auf 0 gesetzt. Die Anderen Prioritätswerte rücken zueinander auf, ohne dass die Rangfolge geändert wird. Für den Fall, dass alle anderen Prioritätswerte 0 sind, kann der Wert einfach geändert werden. Für alle weiteren Fälle wird zuerst überprüft welchen Prioritätswert der Elementtyp des gelöschten Elementes hat. Alle höheren Prioritätswerte werden dann um 1 verringert. Für den Fall, dass ein Prioritätswert auf 1 verringert wurde und damit die höchste neue Priorität hat, wird die Ontologie ausgelesen, da nun alle Einträge verwendbar sind. Im Anschluss dazu wird der Prioritätswert des Elementtyps des gelöschten Elements auf 0 gesetzt. Im nächsten Schritt (siehe Abbildung 5.12) muss für den Elementtyp, der den neuen Prioritätswert 1 hat, noch einmal der *Management* Prozess aufgerufen werden um die neuen Prioritäten auf alle Listen zu übertragen, die dann analog zu *Add_Interact* in den Canvas geschrieben und die Listen aktualisiert werden. Ist keine Priorität 1 vergeben, wird lediglich die Ontologie ausgelesen, da in diesem Fall alle Prioritäten 0 sind.

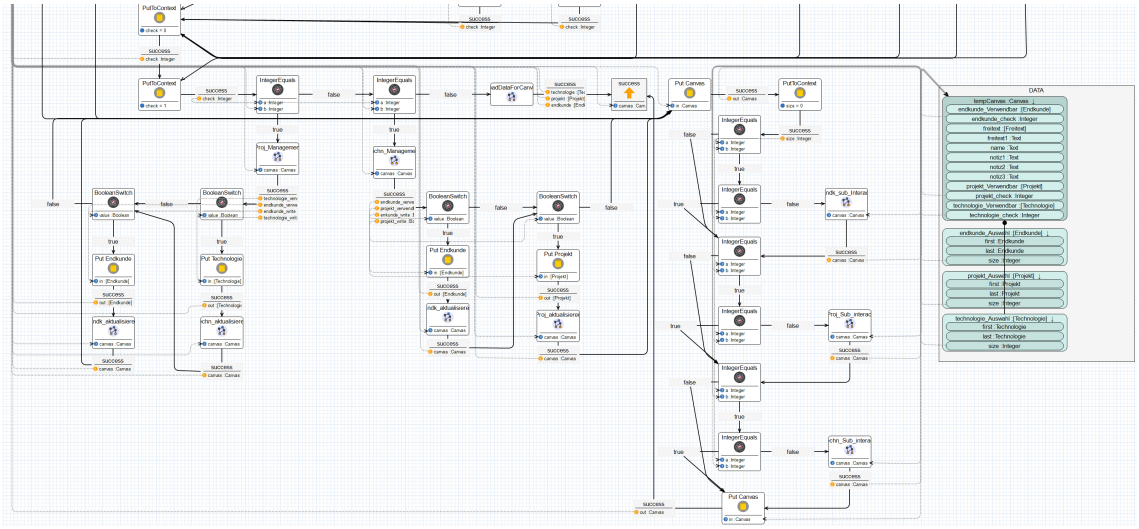


Abbildung 5.12: Canvas *XX_Sub_Interact* Teil 2 (Endkunde)

Im letzten Schritt wird für jeden Elementtyp mit leerer Auswahlliste nochmals der jeweilige *Sub_Interact* Prozess aufgerufen. Dies ist nötig, da ansonsten die Prioritäten bei indirektem Löschen durch den *Management* Prozess nicht korrekt vergeben werden.

5.2.2 Prozess Pattern (Canvas)

primitive Pattern

Prioritäten Korrektur: *Canvas_PrioCorrectCheck*

Dieser Prozess ist ein Hilfsprozess für den *Update* Prozess. Im Updateprozess können Prioritäten durch indirektes Löschen fehlerhaft werden. In diesem Fall rücken sie nicht auf, sodass bspw. trotz eines nicht vergebenen Prioritätswerts von 2 ein Prioritätswert von 3 existiert. Am Ende wird aus diesem Grund dieser Prozess aufgerufen, um alle Prioritäten zu korrigieren.

In Abbildung 5.13 ist der erste Teil dieses Prozesses zu sehen. Es werden drei boolsche Hilfsvariablen für die Überprüfung benötigt, die im ersten Schritt auf False gesetzt werden. Im nächsten Schritt (rot) werden alle Prioritäten auf 0 gesetzt, dessen Elementtypen eine leere Auswahlliste haben. Im nächsten Schritt (blau) wird die Hilfsvariable "one" auf True gesetzt, sobald ein Elementtyp den Prioritätswert 1 hat und die weiteren Überprüfungen abgebrochen. Der nachfolgende Schritt hat eine Fallunterscheidung. Es werden nacheinander, wie im vorherigen Schritt, die Elementtypen überprüft. Jeder dieser Fälle wird analog zueinander bearbeitet. Liegt ein Prioritätswert von 2 vor, wird überprüft, ob ein Prioritätswert von 1 existiert. Ist dies nicht der Fall, wird der aktuelle Prioritätswert auf 1 gesetzt und die Hilfsvariable "one" auf True gesetzt und mit der Überprüfung fortgefahren. Im anderen Fall wird die Hilfsvariable "two" auf True gesetzt und die weiteren Überprüfungen abgebrochen, da keine Lücke in den Prioritätswerten entstanden ist.

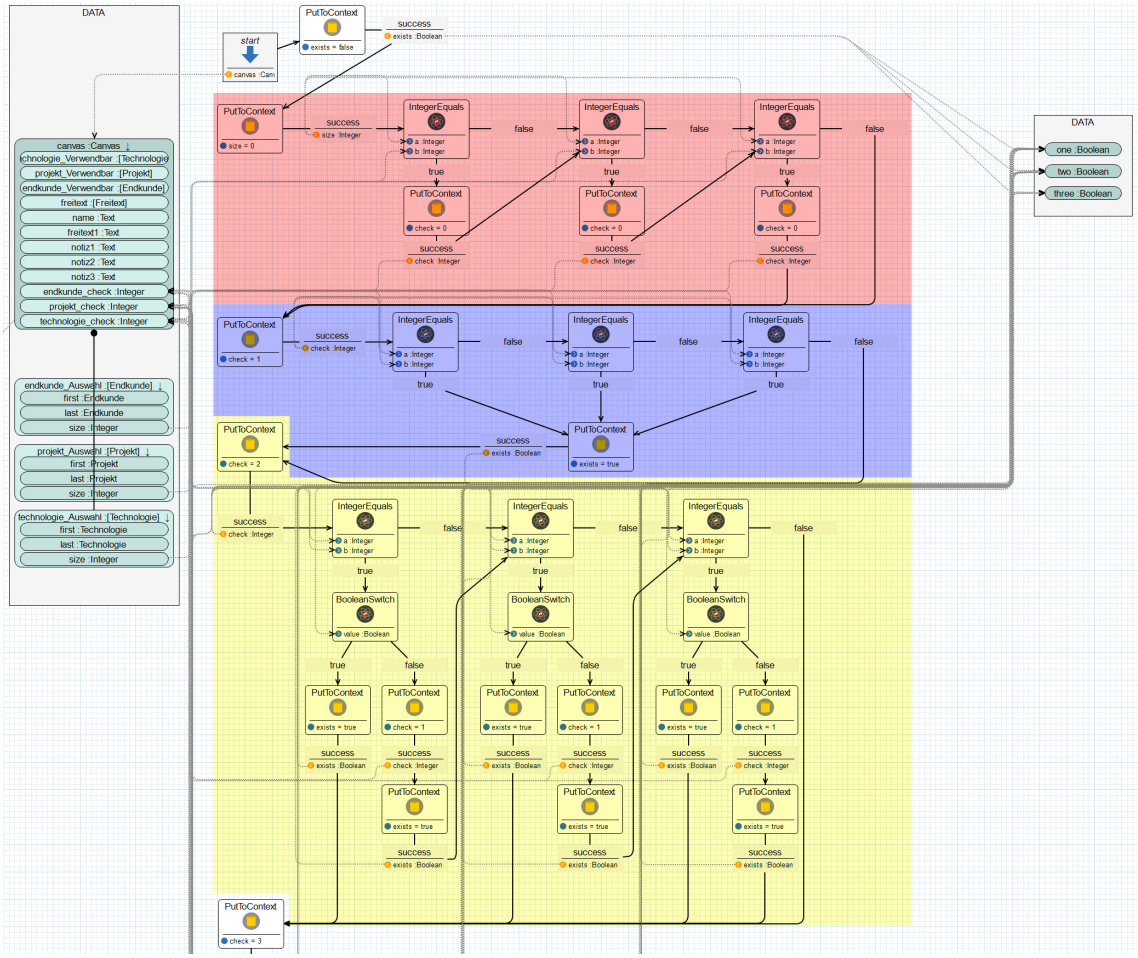


Abbildung 5.13: Canvas Prioritäten Korrektur Teil 1 ANMERKUNG: Die Verbindungen im Bild werden noch bearbeitet

Der nächste Schritt ist in Abbildung 5.14 zu erkennen. Auch in diesem Schritt werden die Elementtypen nacheinander auf ihren Prioritätswert geprüft. Existiert kein Prioritätswert von 3 oder wird ein Prioritätswert von 3 gefunden und existiert ein Prioritätswert von 2, wird der Prozess beendet und der korrigierte Canvas zurückgegeben. Wie auch in den vorherigen Fällen sind die Fälle analog zueinander. Existiert ein Prioritätswert von 3, aber kein Prioritätswert von 2 wird dieser korrigiert. Er wird auf 2 gesetzt, wenn der Prioritätswert 1 vorhanden ist, sonst auf 1. In beiden Fällen werden die jeweiligen Hilfsvariablen mit angepasst und mit der Überprüfung weiter fort gefahren.

Reset: *Canvas_Reset*

Dieser Prozess ist für das Zurücksetzen des jeweiligen Canvas gedacht. Wie in Abbildung 5.15 zu sehen ist, wird der Reihe nach über die Auswahllisten aller Elementtypen iteriert und die Elemente dabei aus den Listen gelöscht. Danach werden alle Prioritätswerte auf 0 gesetzt, die einzelnen Notitzen mit Null ersetzt und zum Schluss über alle Freitexte iteriert, wobei jedes Element aus der Liste entfernt wird.

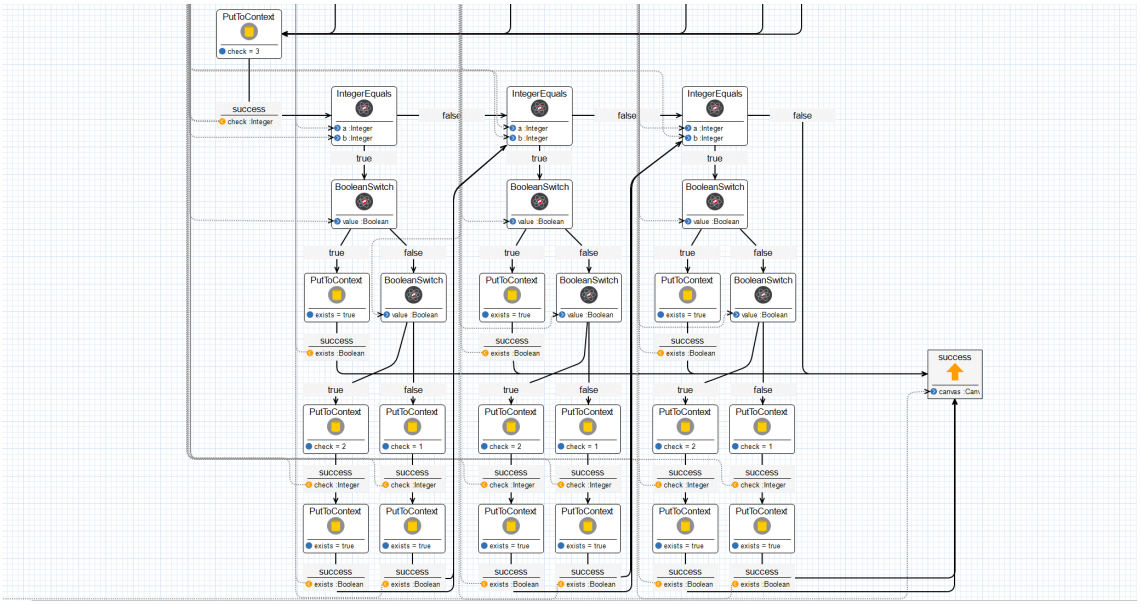


Abbildung 5.14: Canvas Prioritäten Korrektur Teil 2 ANMERKUNG: Die Verbindungen im Bild werden noch bearbeitet

Die verwendbar Listen müssen dabei nicht verwaltet werden, da diese im Fall, dass alle Prioritätswerte 0 sind, durch den Updateprozess durch die Ontologiedaten ersetzt werden.

Suche maximale Priorität: *Canvas_SearchMaxCheck*
Dieser Prozess sucht den höchsten vergebenen Prioritätswert der Elementtypen im Canvas. Hierzu wird, wie in Abbildung 5.16 zu sehen, zuerst der maximale Wert auf 0 gesetzt. Im ersten Schritt wird dieser überschrieben, falls der erste Prioritätswert bereits gesetzt wurde und damit größer als 0 ist. In den nachfolgenden Schritten werden die Prioritätswerte der anderen Elementtypen überprüft und der Maximalwert überschrieben, falls der neue Prioitätswert größer ist als der derzeitige Maximalwert.

Modalansicht Freitext: CanvasFreitextModal
Analog zu den Modalansichten in der Ontologie öffnet der Prozess im ersten Schritt die modale GUI zur Ansicht der Freitexte. Wie in Abbildung 5.17 zu erkennen ist, finden hier die Pattern aus der Ontologie für Erstellen, Löschen und Bearbeiten ihre Anwendung.

Modalansicht Notiz: CanvasNotizModal
Analog zu den Modalansichten in der Ontologie öffnet der Prozess im ersten Schritt die modale GUI zur Ansicht der Notizen. Da Notizen nur über einzelne Textvariablen gespeichert und dargestellt werden, entspricht dieser Prozess dem Prozess für die Modalansichten der Freitexte ohne eine Listenverwaltung.

Canvases laden: *LadeDatenCanvas*

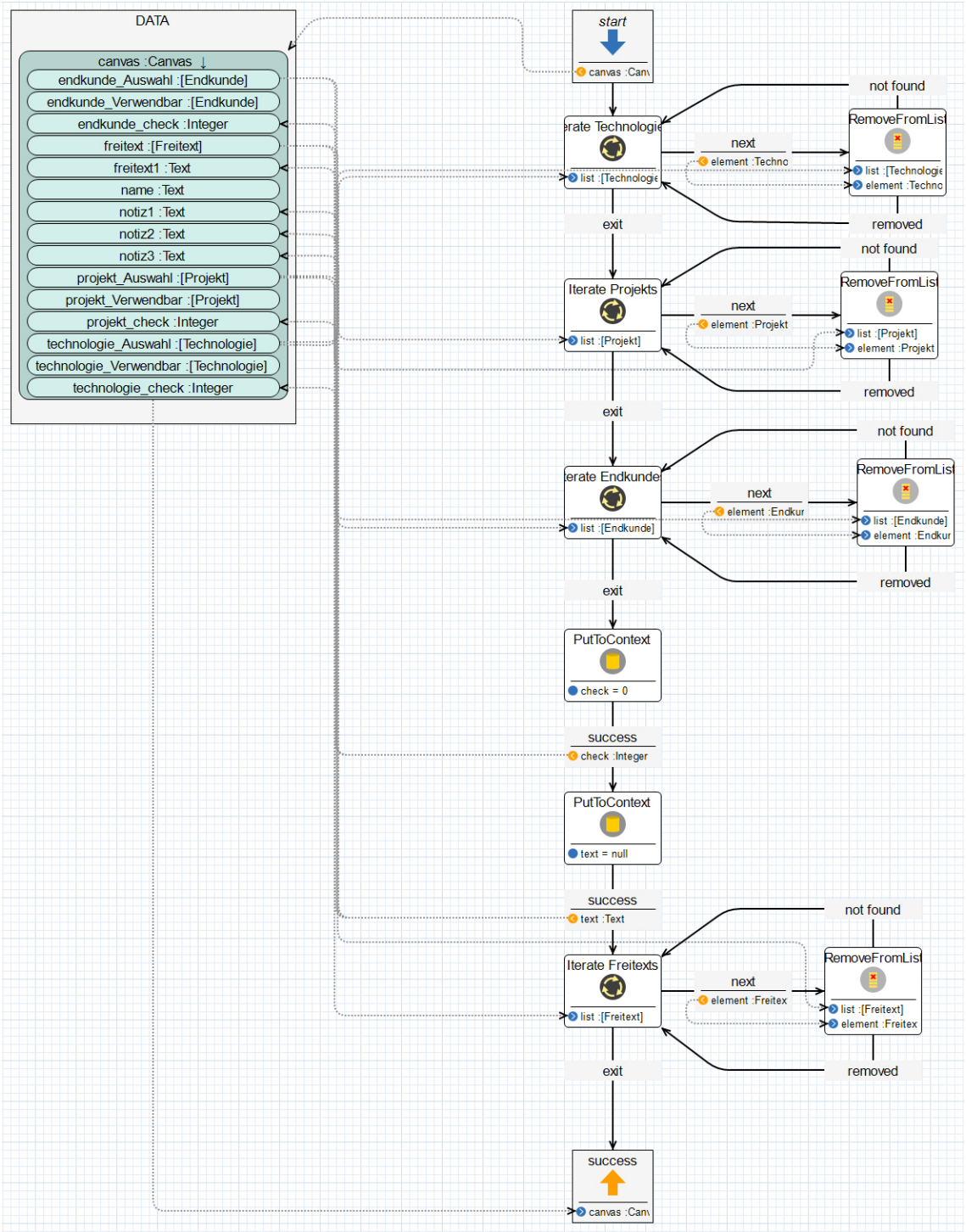


Abbildung 5.15: Canvas Reset ANMERKUNG: Die Verbindungen im Bild werden noch bearbeitet

Dieser Prozess läd die bereits angelegten Canvases über ein RetrieveSIB und gibt sie als Liste zurück.

Canvas löschen: *EntferneCanvas*

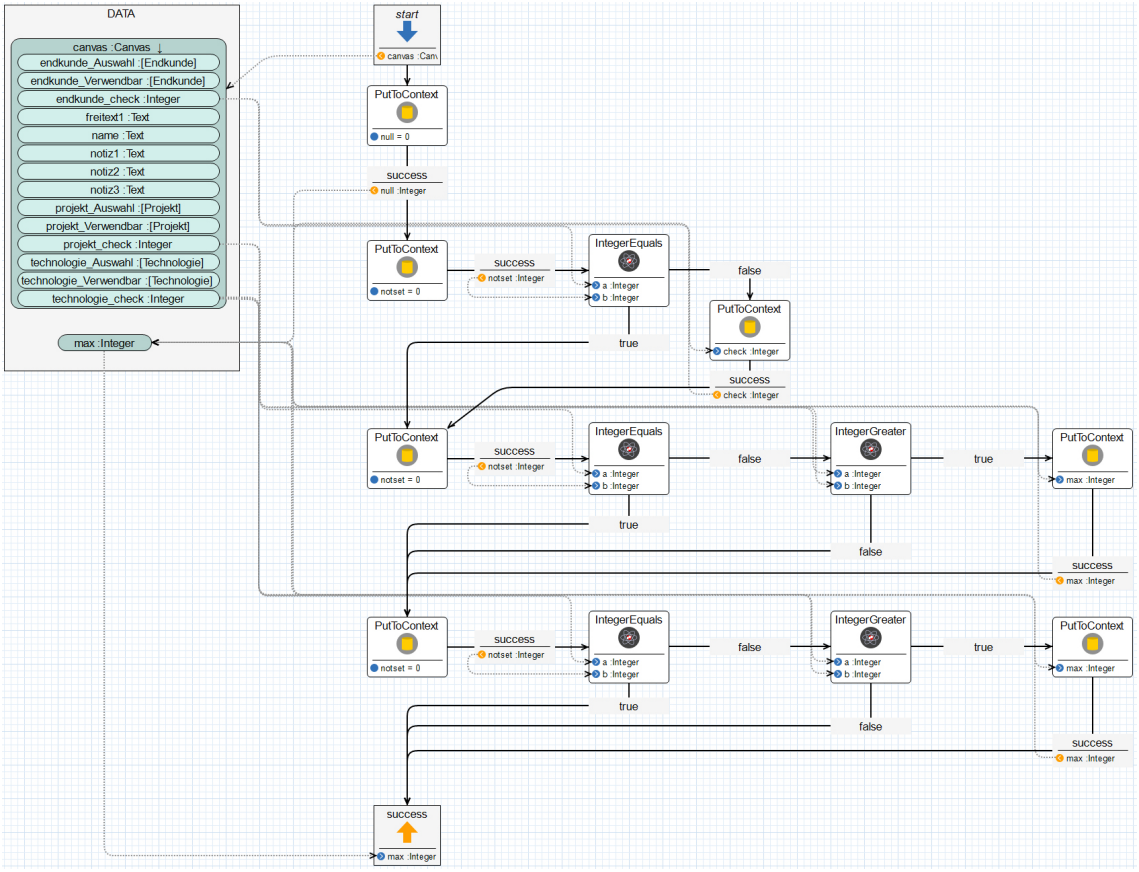


Abbildung 5.16: Canvas Suche maximale Priorität

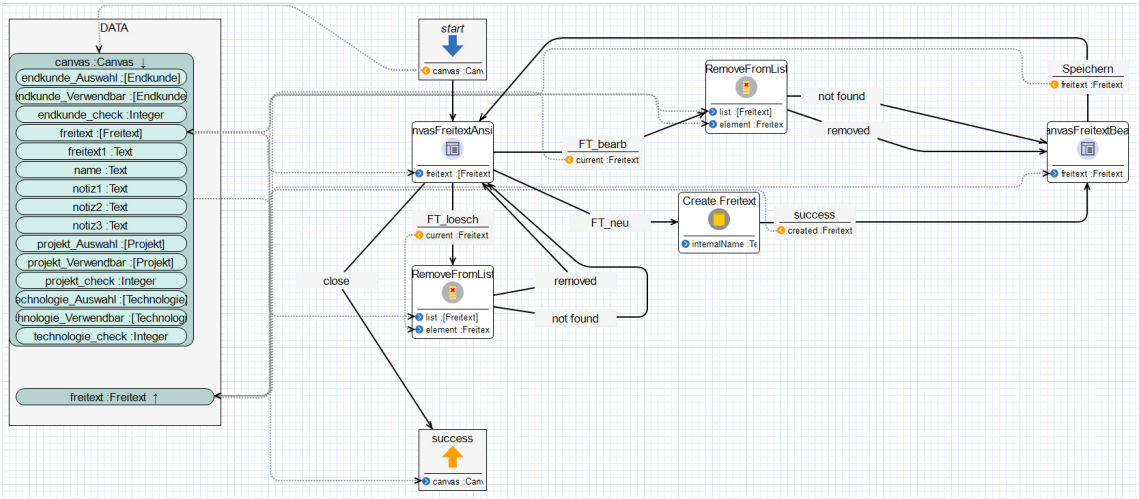


Abbildung 5.17: Canvas Modalansicht Freitext ANMERKUNG: Die Verbindungen im Bild werden noch bearbeitet

Dieser Prozess entfernt, wie in Abbildung 5.18 zu sehen ist, zuerst das übergebene Canvas Element aus der ebenfalls übergebenen Liste und löscht es danach.

Daten für den Canvas laden: *LoadDataForCanvas*

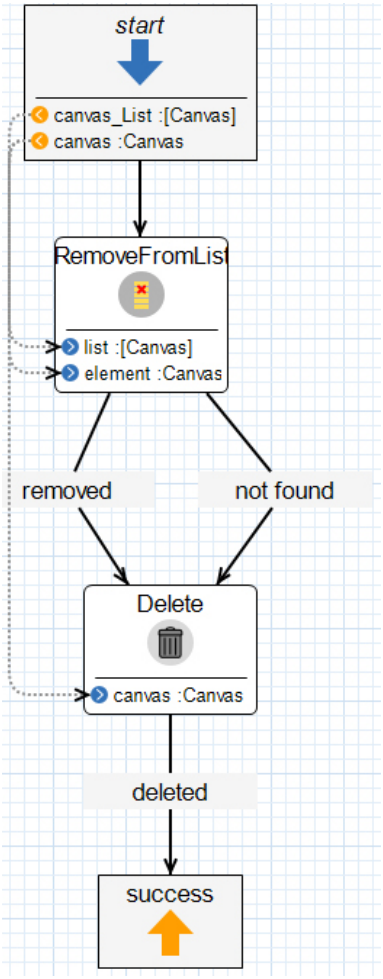


Abbildung 5.18: Canvas löschen

Bei diesem Prozess handelt es sich um einen einfachen Hilfsprozess, der die Daten für einen neuen Canvas aus der Ontologie läd und zurückgibt. Hierbei wird nacheinander ein RetrieveSIB für jeden Elementtypen ausgeführt.

Komplexe Pattern

Canvas erzeugen: *CreateNewCanvas*
Wie in dem *Erzeugen* Pattern der Ontologie wird der Canvas über ein CreateSIB erzeugt. Hierzu müssen keine Daten übergeben werden. Wie in Abbildung 5.19 zu sehen ist, werden diese Daten im nächsten Schritt über den Prozess *LoadDataForCanvas* aus der Ontologie geladen. Zum Schluss werden die Daten in dem Canvasobjekt gespeichert, alle Prioritätswerte auf 0 gesetzt und das Canvasobjekt zurückgegeben.

Vorauswahl: *CanvasVorauswahl*
Mit diesem Prozess wird die Vorauswahl bereit gestellt. Damit ist das Anlegen, Benennen und Löschen der verschiedenen Canvases gemeint. Wie in Abbildung 5.20 zu sehen, werden

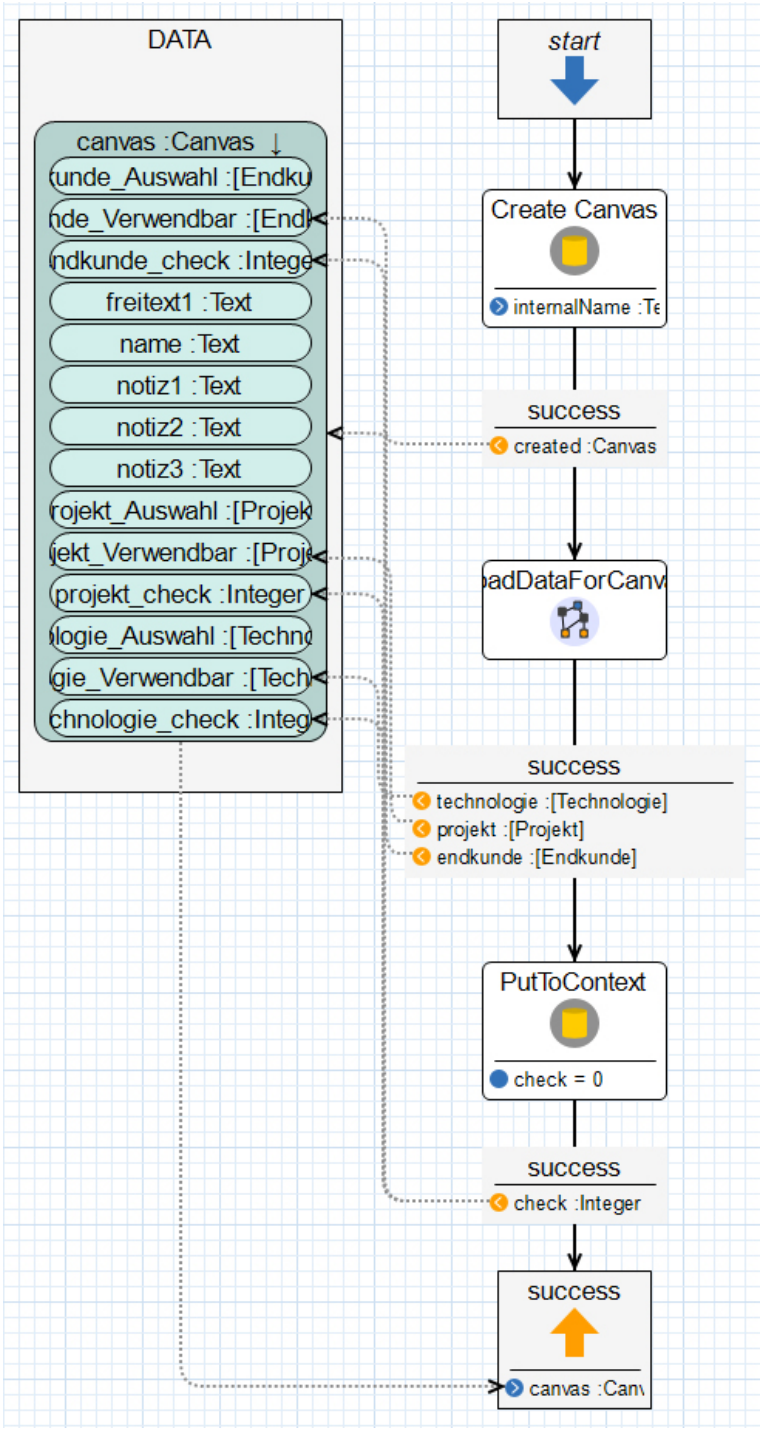


Abbildung 5.19: Canvas erzeugen ANMERKUNG: Die Verbindungen im Bild werden noch bearbeitet

zuerst die bereits erzeugten Canvases über den Prozess *LadeDatenCanvas* geladen, in den Datenkontext übergeben und die Canvas Verwalten GUI geöffnet. Die *Canvas Bearbeiten* GUI wird nach einem Klick auf den Button Bearbeiten oder Anlegen ausgeführt. Im letzten Fall wird vorher ein Canvas über den Prozess *CreateNewCanvas* erzeugt und mit

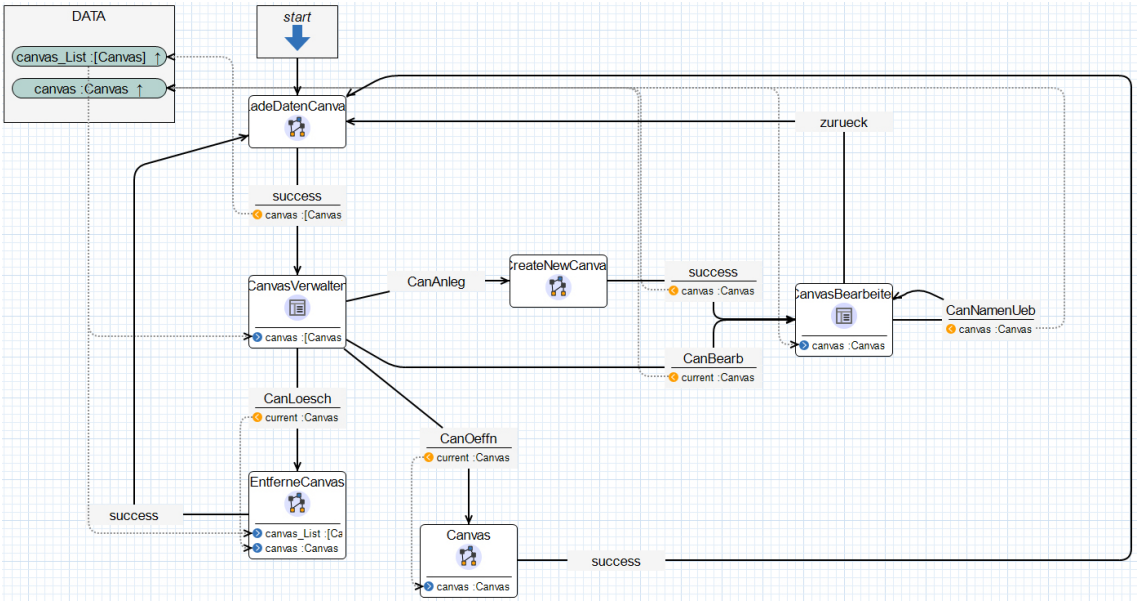


Abbildung 5.20: Canvas Vorauswahl

den Ontologiedaten befüllt. Nach dem Betätigen des Löschen Buttons löscht der Prozess *EntferneCanvas* den jeweiligen Canvas aus der Liste der Canvases und mit einem Klick auf Öffnen wird der Hauptprozess zur Verwaltung des jeweiligen Canvas ausgeführt.

Canvas Hauptprozess: *Canvas*

Dieser Prozess steuert die Ausführung der verschiedenen Prozesse zur Verwaltung des Canvas. Im ersten Schritt wird der *Canvas_ Update* Prozess ausgeführt, der den Canvas auf die aktuellen Ontologiedaten aktualisiert. Nach jedem Schritt in diesem Prozess wird der Canvas im Datenkontext zwischengespeichert und von dort an die benötigten Prozesse weitergeleitet.

In Abbildung 5.21 lassen sich zwei gefärbte Bereiche erkennen. Diese beinhalten die Prozesse, die durch das Hinzufügen (rot) und das Entfernen (blau) aufgerufen werden. Nach dem Klick auf den Hinzufügen Button der Modalansicht wird das jeweilige Element in den Canvas Hinzugefügt und der jeweilige *XX_ Add_ Interact* Prozess aufgerufen, der die Prioritäten aktualisiert und anschließend die Beeinflussung im Canvas umsetzt und zurückgibt. Analog dazu ist das Vorgehen nach dem Entfernen eines Elements aus der Auswahl. Im Vorhinein wird an dieser Stelle der Prozess *XX_ SubProcess* ausgeführt, um die Daten aus der Auswahlliste zu löschen und der verwendbar Liste hinzuzufügen. Am Ende jedes Prozesspfades wird der *Canvas_ Update* Prozess ausgeführt, um immer den aktuellen Ontologiestand in der Canvasverwaltung zu haben.

Das Schließen der jeweiligen Modalansichten für Freitexte und Notizen endet in dem *success* Pfad, wobei der Canvas im Datenkontext aktualisiert wird und danach der *Canvas_ Update* Prozess ausgeführt wird. Ein Freitext wird nach dem Ontologiepattern zum Löschen von

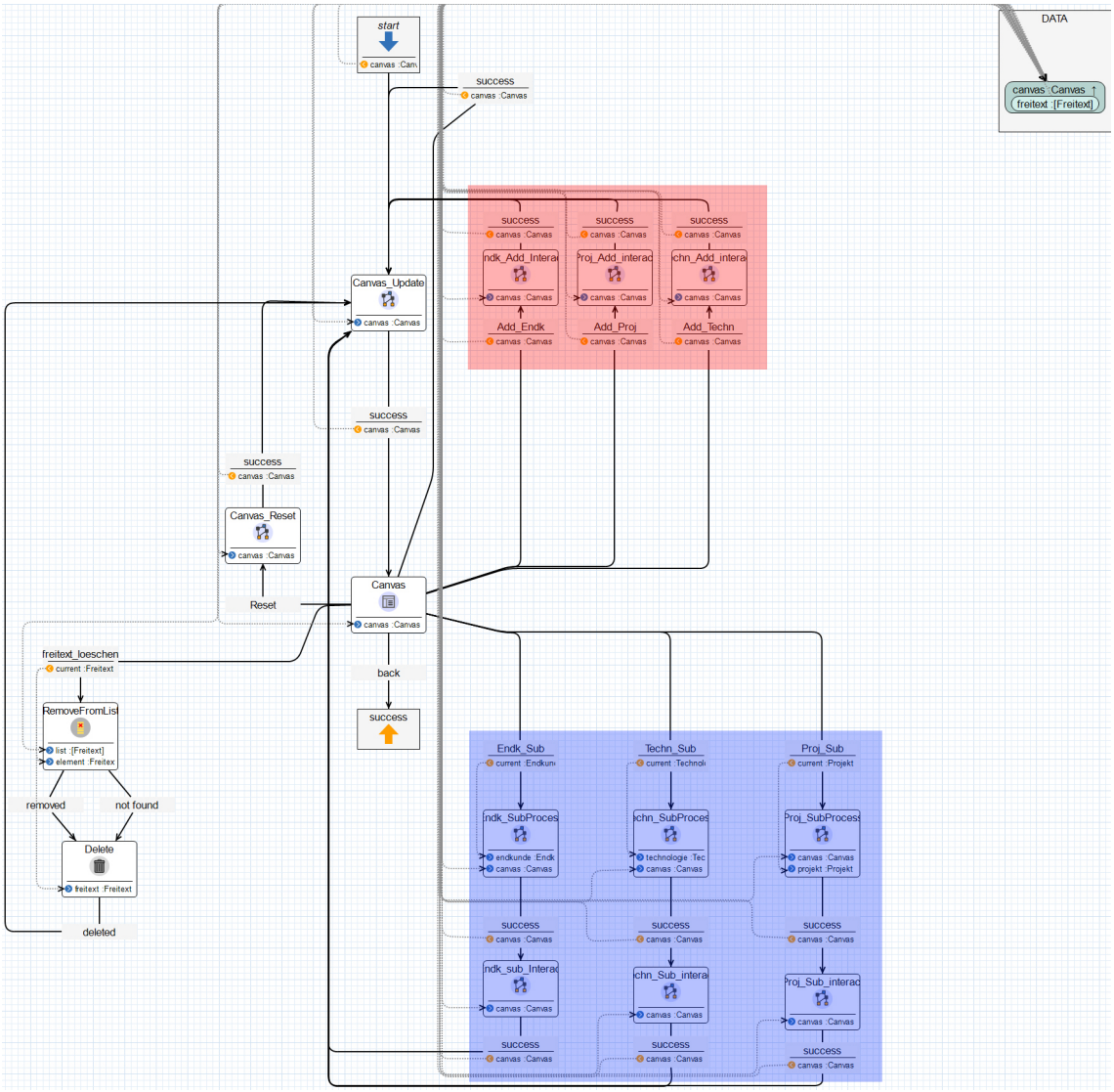


Abbildung 5.21: Canvas Hauptprozess

Listen entfernt und ein Klick auf den Reset Button führt den Prozess *Canvas_ Reset* aus, um den Canvas zurückzusetzen.

Canvas Update: *Canvas_ Update*

Der *Update* Prozess ließt die Elemente aus der Ontologie aus und aktualisiert den Canvas unter Beachtung der Prioritäten und bereits getätigten Auswahlen. Hierzu werden nacheinander alle Prioritätswerte der verschiedenen Elementtypen überprüft und darauf basierend die Elemente aus der Ontologie neu verteilt. Der Prozess ähnelt einer rein iterativen Version der *Management* Prozesse. Abbildung 5.22 zeigt den ersten Teil des Prozesses. Zu Beginn (gelb) werden die aktuellen Daten aus der Ontologie ausgelesen. Für den Fall, dass alle Prioritätswerte 0 sind (blau) werden die ausgelesenen Daten direkt übernommen. Die Überprüfung für einen Prioritätswert von 1 ist in Abbildung 5.22 rot markiert. Es wird für

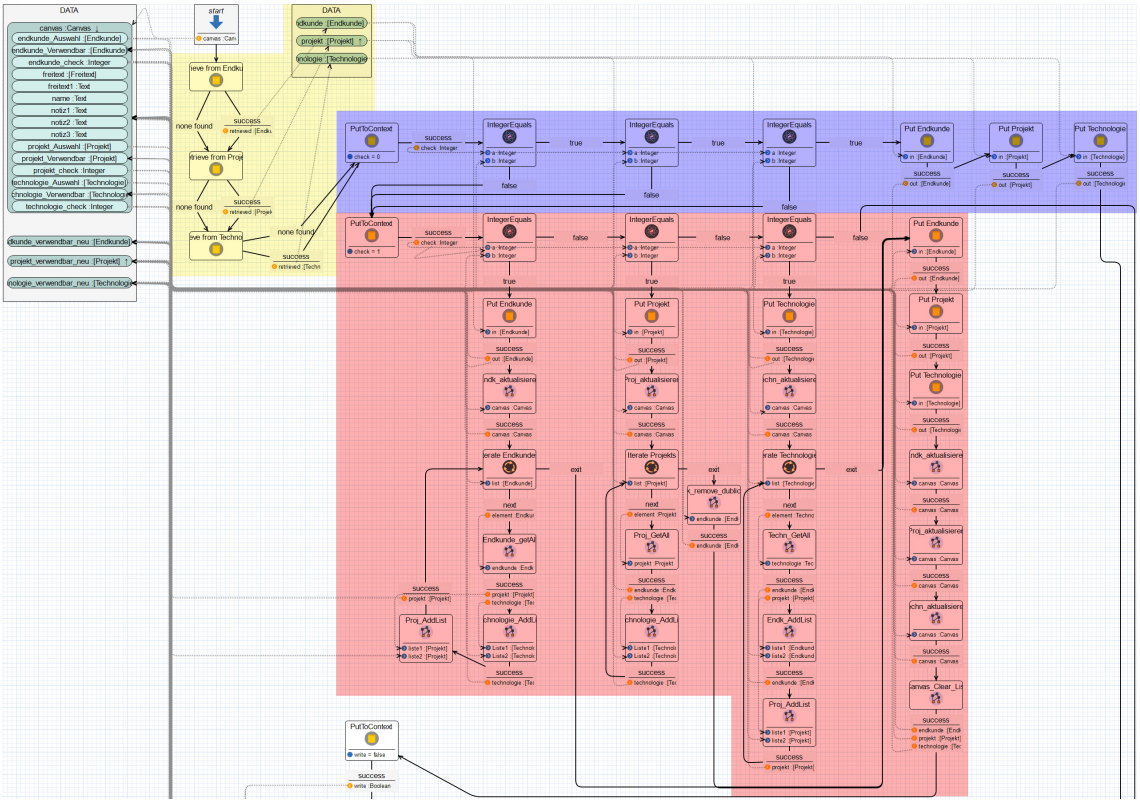


Abbildung 5.22: Canvas Updateprozess erster Teil

jeden Elementtyp überprüft, ob dieser die Priorität 1 zugewiesen bekommen hat. Nachdem ein Elementtyp mit dieser Zuweisung gefunden wurde werden die weiteren Fälle nicht mehr betrachtet, da Prioritäten immer nur einzeln vergeben werden.

Abbilung 5.23 zeigt einen dieser Fälle. Zu Beginn werden die neu ausgelesenen Daten in die entsprechende verwendbar Liste des Canvas und in eine neue verwendbar Liste übernommen und die entsprechende Auswahlliste aktualisiert. In diesem Fall sind dies die Listen der Projekte. Analog zu den *Management* Prozessen werden dann basierend auf den Projektlisten die neuen verwendbar Listen für alle weiteren Elementtypen gebildet. Für den Fall, dass von dem Prozess *getAll* einzelne Elemente und keine Liste zurückgegeben wird, müssen im Nachhinein doppelte Elemente entferne werden. Die so erzeugten neuen verwendbar Listen werden danach in den Canvas übernommen, die Listen aktualisiert und zum Schluss die temporären, neuen Listen gelöscht, da diese in den weiteren Überprüfungen wieder genutzt werden.

Abbildung 5.24 zeigt den Rest des Prozesses. Nachdem dir Boolschen Variablen, die analog zu den Boolschen Werten der *Management* Prozesse genutzt werden, auf False gesetzt wurden, wird analog zum ersten Fall jeder Elementtyp auf den Prioritätswert 2 überprüft und nach der Behandlung eines einzelnen die weiteren Überprüfungen abgebrochen. In der Nachbehandlung werden dann nur die Elementtypen aktualisiert, die vorher im Prozess beschrieben wurden.

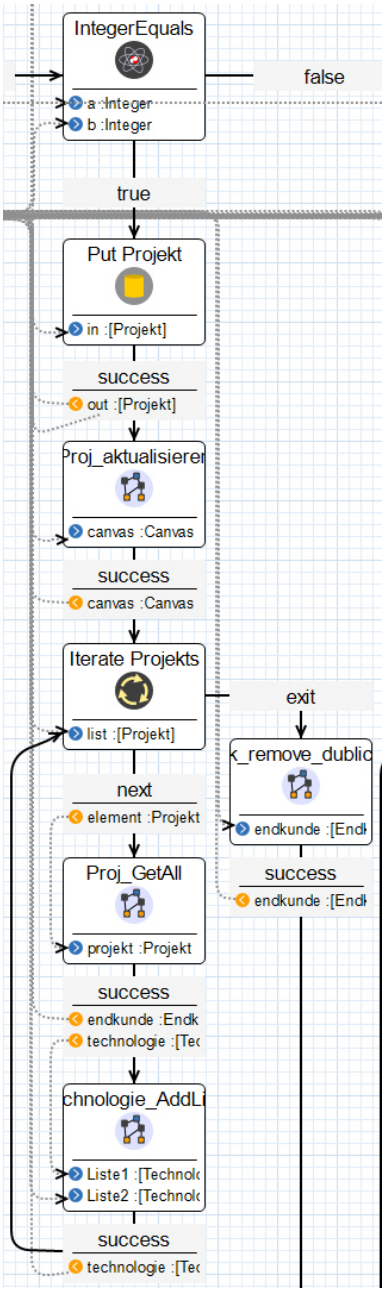


Abbildung 5.23: Canvas Updateprozess: Ausschnitt Fall Priorität = 1

Abbildung 5.25 zeigt den Fall, dass Projekte den Prioritätswert 2 haben. Wie auch im ersten Beispiel in Abbildung 5.23 wird über alle Projekte in der Auswahl iteriert und per *getAll* alle damit verknüpften Elemente geladen. Für jeden dieser Elementtypen wird dann überprüft, ob er einen höheren Prioritätswert oder den Prioritätswert 0 hat. In diesen Fällen wird analog zum ersten Beispiel die jeweilige verwendbar Liste in den Canvas übernommen. Ist der Prioritätswert kleiner, so ist die Priorität höher und die verwendbar Liste darf nicht eingeschränkt werden. Im letzten Schritt werden doppelt gesetzte Elemente entfernt.

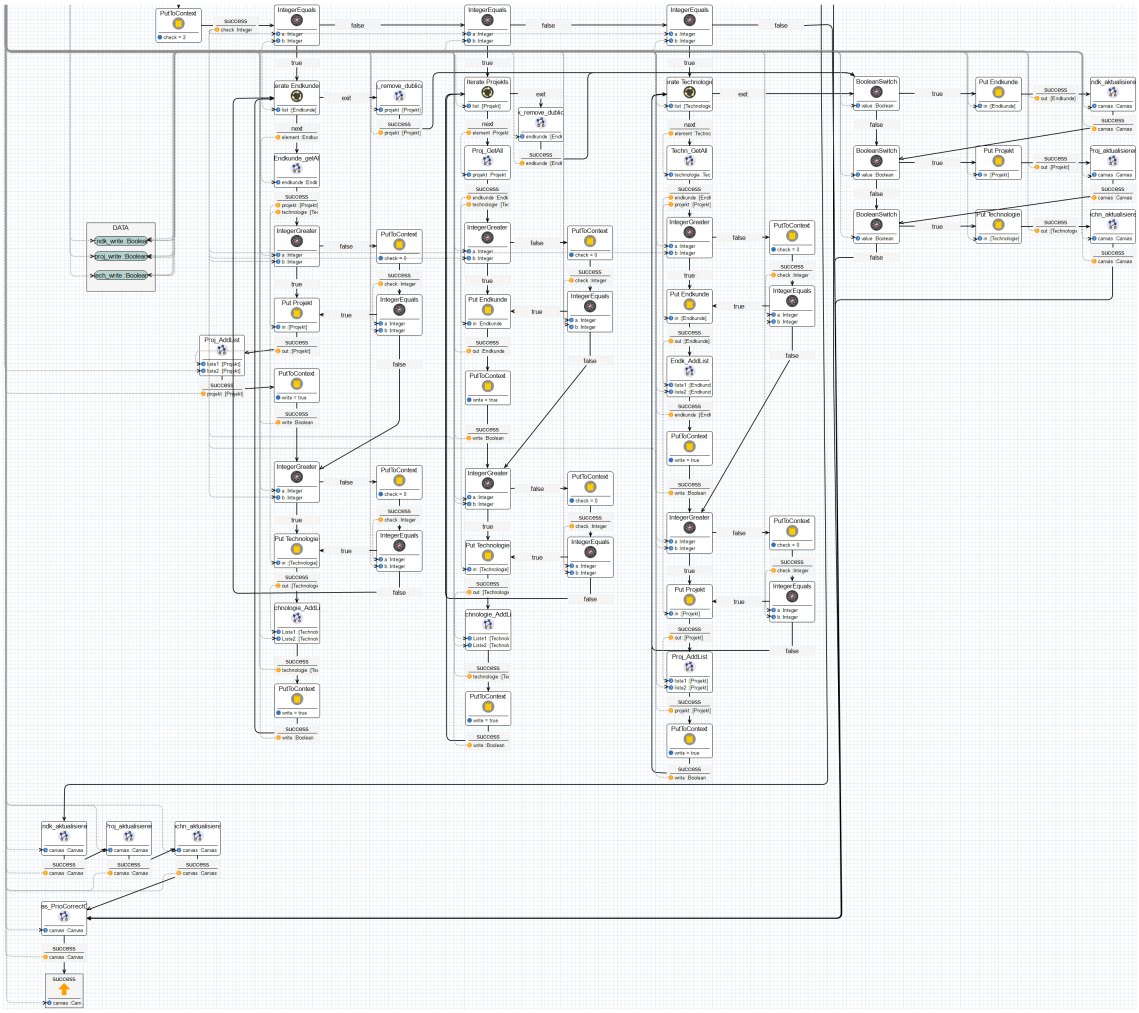


Abbildung 5.24: Canvas Updateprozess zweiter Teil

Für jede weitere Priorität, außer der letzten muss dieses Pattern weitergeführt werden. Im Fall dieses Beispiels gäbe es nurnoch den Prioritätswert 3, der nichts weiter einschränken kann.

Da im Laufe des Algorithmus durch indirektes Löschen die Prioritätswerte nicht bearbeitet werden, werden im letzten Schritt über den Prozess *PrioCorrect* die Prioritätswerte korrigiert.

5.2.3 GUI Pattern

Vorauswahl der Canvases

Die Vorauswahl entspricht der einfachen Verwalten GUI aus der Ontologie. Canvases können erzeugt, benannt und gelöscht werden. Die Tabelle zur Darstellung kann durchsucht werden. Mit einem Klick auf den Öffnen Button öffnet sich die Canvas GUI.

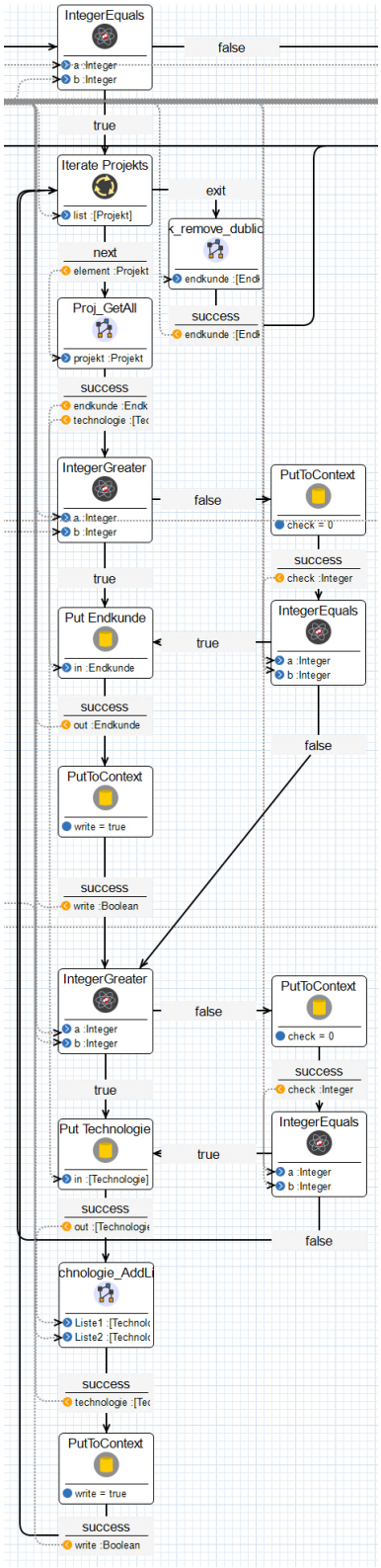


Abbildung 5.25: Canvas Updateprozess: Ausschnitt Fall Priorität = 2

Canvas GUI

Der Canvas wird anhand des Beispielcanvas in Abbildung 5.26 erläutert, der dem Business Model Canvas nach Alexander Osterwalder (2005) nachempfunden wurde. In diesem Beispielcanvas wurden Endkunden, Projekt und Technologie als auswählbare Elemente hinzugefügt. Außerdem befindet sich ein Freitext in dem Feld "Customer Segment". Der

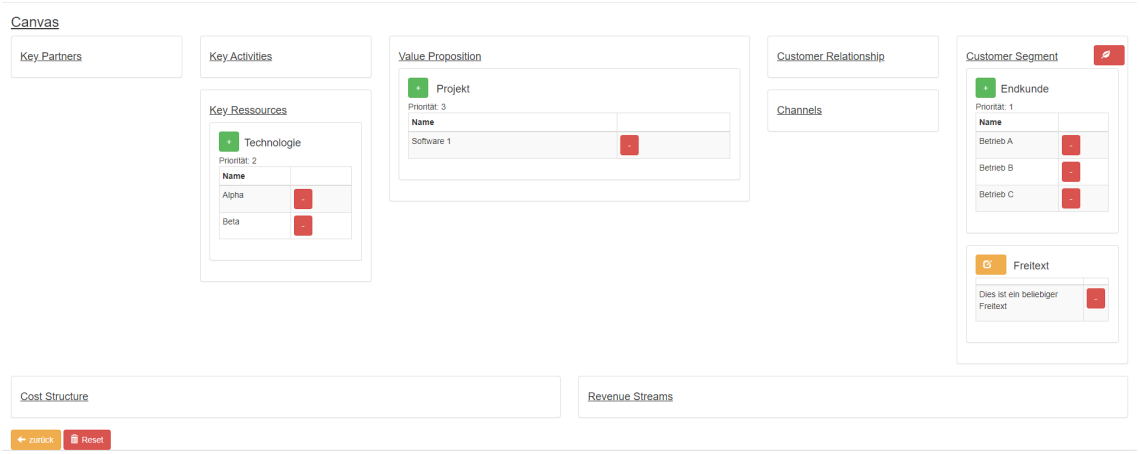


Abbildung 5.26: Canvas GUI ANMERKUNG: Bild wird noch ausgetauscht gegen eines ausgetauscht mit befüllten Daten

Canvas zeigt die Übersicht über alle bereits ausgewählten Daten in den jeweiligen Tabellen an. Die vergebene Priorität wird über der Tabelle angezeigt. Mit einem Klick auf den "-" Button wird das ausgewählte Element aus der Auswahl entfernt. Ein Klick auf den "+" Button öffnet die Modalansicht zum Hinzufügen von Elementen des jeweiligen Elementtyps. Freitexte werden ebenfalls als Feld im Canvas angezeigt mit einer darunter befindlichen Listendarstellung der eingespeicherten Freitexte. Mit einem Klick auf die in Abbildung 5.27 zu sehenden Buttons öffnet sich die Modalansicht für die Notiz Ansicht oder Freitext bearbeiten Ansicht.

Modalansichten

Element Hinzufügen In der Modalansicht zum Hinzufügen von Elementen zu der Auswahl werden die Elemente in einer durchsuchbaren Tabelle mit ihrem Namen dargestellt. Abbildung 5.28 zeigt dies beispielhaft anhand von Endkunden. Mit einem Klick auf den Button "+" lässt sich das gewünschte Element zur Auswahl hinzufügen und die Modalansicht beendet sich automatisch.

Freitext bearbeiten Modalansicht

Notiz Modalansicht

Abbildung 5.27: Freitext und Notiz Buttons

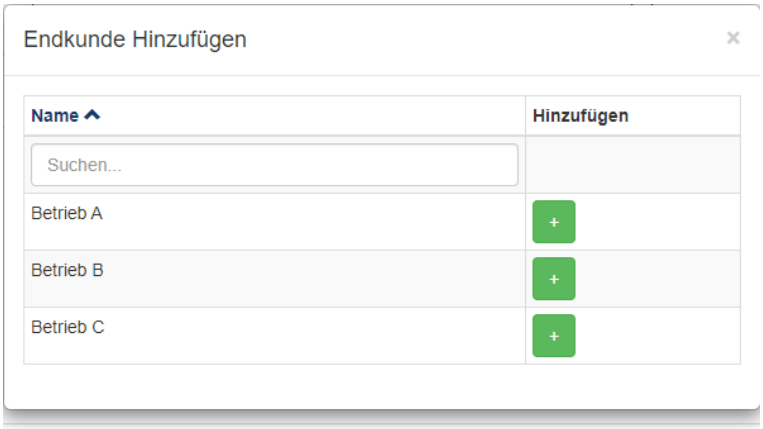


Abbildung 5.28: Modalansicht Element Hinzufügen

Freitext GUI Die Freitext bearbeiten Ansicht stellt, wie in Abbildung 5.29 zu sehen, alle bereits erzeugten Freitexte in einer nicht durchsuchbaren Tabelle dar. Diese können einzeln über den Button "-" gelöscht werden. Die Ansicht muss über den Schließen Button beendet werden, damit der Canvas direkt aktualisiert wird. Bei einem Schließen der Ansicht über das X oben rechts werden Änderungen erst bei der nächsten Aktion im Canvas sichtbar. Durch den Bearbeiten Button gelangt man in eine neue Ansicht mit der sich ein neuer Freitext anlegen lässt. Ein Klick auf Bearbeiten öffnet die gleiche Ansicht und zeigt

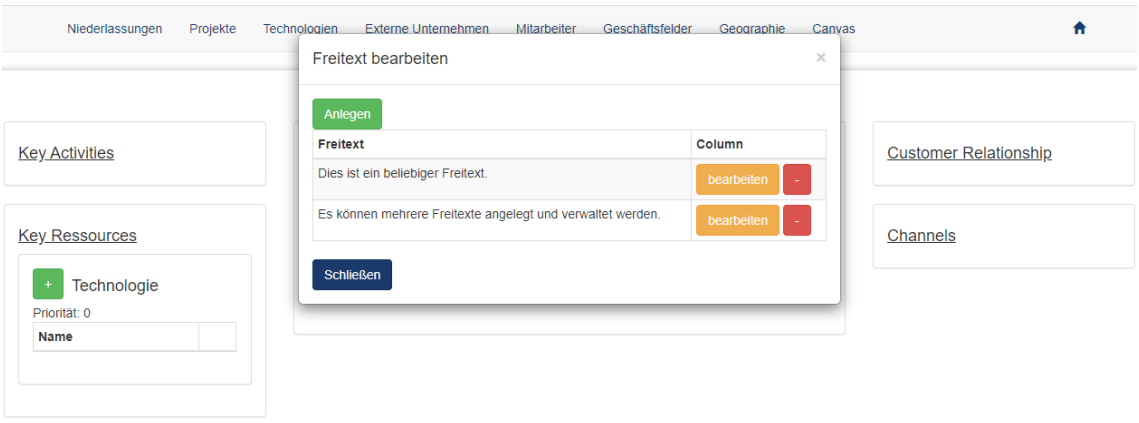


Abbildung 5.29: Modalansicht Freitext bearbeiten

direkt den zu bearbeitenden Text an. In dieser Ansicht kann ein beliebiger neuer Freitext eingeben werden. Zeilenumbrüche sind möglich und werden in den Tabellen ebenso angezeigt. Der Speichern Button übernimmt die Änderungen in den Canvas.

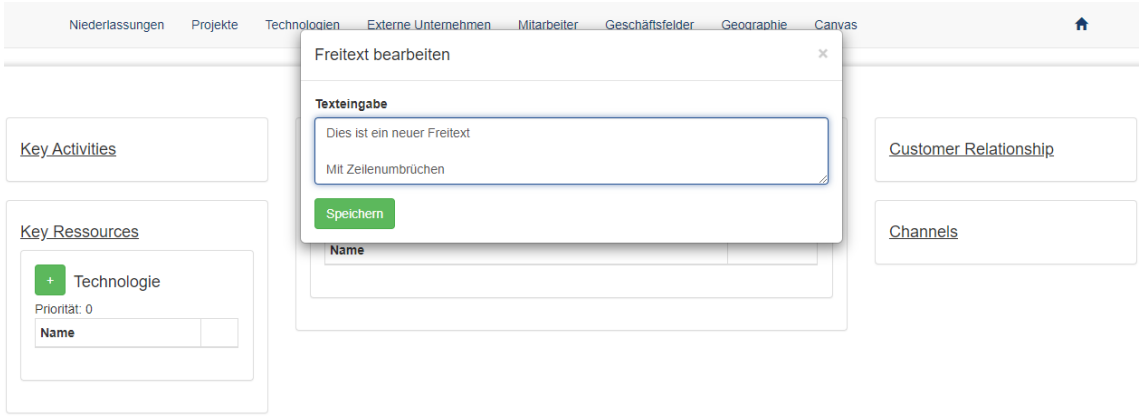


Abbildung 5.30: Modalansicht Freitext erstellen

Notiz GUI Die Notiz Modalansicht ist analog zu der Freitext Modalansicht mit dem Unterschied, dass nur eine Notiz angezeigt oder erzeugt werden kann. In Abbildung 5.31 ist eine beispielhafte Notiz erzeugt worden. Über einen Klick auf den gelben Button öffnet sich die gleiche Bearbeiten Ansicht wie bei der Erstellung oder Bearbeitung von Freitexten. Wurde eine Notiz dem Canvas hinzugefügt ändert sich die Farbe des Notiz Buttons, wie

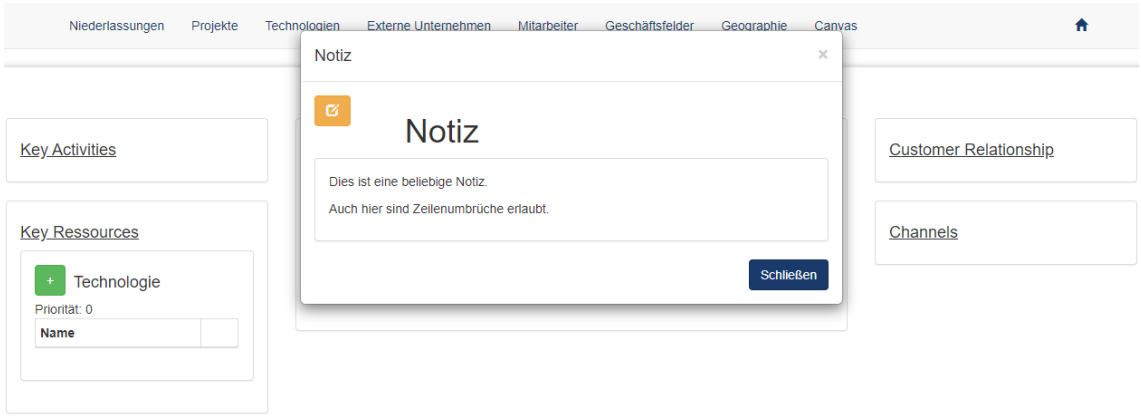


Abbildung 5.31: Modalansicht Notiz Ansicht

in Abbildung 5.32 zu sehen ist.

keine Notiz Notiz vorhanden

Abbildung 5.32: Farbe der Notizbuttons

Kapitel 6

Implementierung Canvas

Im Folgenden wird die Implementierung des Canvas beschrieben. Dazu wird zuerst die DSL mit den Bestandteilen und Checks, die überprüfen, ob der Canvas richtig angelegt wurde, erklärt. Danach wird ein Beispielcanvas beschrieben. Im Anschluss wird die Implementierung der Generation und wichtige Generatoren detailliert erläutert.

6.1 DSL

In Kapitel 4.1 wird bereits erwähnt, inwiefern die Projektgruppe sich eine DSL zu Nutzen macht, um Firmenontologien abzubilden. Solch eine DSL soll nun auch dazu dienen, dass ein individueller Canvas erstellt und mit den Inhalten der zuvor erstellen Ontologie befüllt werden kann.

6.1.1 Basis der Canvas-DSL

In Anlehnung an 4.1 wird auch die für den Canvas entwickelte DSL innerhalb von CINCO erstellt. Dabei handelt es sich bei dieser DSL weniger um eine Ansammlung klassischer UML-Elemente, als vielmehr um das tabellarische Anordnen von Elementen, die mit Inhalten aus einer Ontologie befüllt werden können.

6.1.2 Implementierung der Features in der DSL

Hier werden die Features angesprochen, aus denen die DSL des Canvas zusammengesetzt ist.

Rows und Cols

Rows und *Cols* bieten gemäß ihrer wörtlichen Übersetzung mit den Reihen und Spalten das Grundgerüst eines Canvas. Diese können beliebig in einander verschachtelt werden. Durch dieses Verschachteln ergeben sich Felder, in die sich *Fields* einfügen lassen. Bevor

ein Field in einen Canvas eingefügt werden kann, ist daher mindestens eine Reihe oder eine Spalte notwendig.

Fields

Fields können innerhalb von Reihen und Spalten eingefügt werden. Sie sind in der Lage, *Contents* aus einer Ontologie-DSL aufzunehmen. Darüber hinaus können auch *Notizen* und *Freitexte* in die Fields eingebracht werden.

Freitexte

Freitexte können, genauso wie die Contents der Ontologie, mit in Fields aufgenommen werden. Im Gegensatz zu Contents, können Freitexte nicht mit einem Path versehen werden. In der Webapp wirken die Freitexte ähnlich wie Contents, sitzen demnach mittig innerhalb der Felder der Webapp.

Notizen

Notizen können ebenfalls in die Fields des Canvas eingefügt werden. Wie schon die Freitexte, können sie weder Start- noch Endpunkt eines Path darstellen. Notizen müssen von Freitexten unterschieden werden, weil sie nach der Generierung in der Webapp anders dargestellt werden und eine andere Position bekommen.

Prime Reference zu Contents

Per Prime Reference zu den Contents aus der Ontologie-DSL ist es möglich, diese Inhalte in die Fields der Canvas-DSL zu ziehen. Diese Contents können mit einem *Path* mit anderen Contents in anderen Fields verbunden werden, um zwischen diesen Contents eine Beeinflussung in der später generierten Webapp zu erzeugen.

Path und Suchpfad

Mit der Hilfe eines Path können zwei Contents miteinander Verbunden werde, um eine Beeinflussung zwischen diesen beiden Elementen zu ermöglichen. Um eine korrekte Beeinflussung zu erzielen, muss zu einem Path ein Suchpfad hinzugefügt werden. In diesem wird definiert, über welche Wege die Ontologie durchlaufen werden muss, um von dem Startcontent des Path zu dem Zielcontent zu gelangen, Also welche Attribute dort mittels Assoziationen in Verbindung stehen. Beispielhaft ist dies in Abbildung 6.2 dargestellt. Ein solcher Suchpfad besitzt dabei immer den folgenden Aufbau: .{{$startcontent$}}.{{$zwischenattribut_n$}}.{{$zielcontent$}}.. Dabei können für .{{$zwischenattribut_n$}}. beliebig viele Zwischenattribute zwischen dem Startcontent und Zielcontent stehen, abhängig davon, wie viele benötigt werden, um den entsprechenden Path korrekt abzubilden.

6.1.3 Custom Checks und Appearances

Ähnlich zu der Ontologie, gibt es auch in der Canvas-DSL Checks, die den Aufbau der DSL überwachen und Rückmeldungen darüber geben, ob diese korrekt erstellt ist. Der Nutzer hat so einen Überblick darüber, ob er den erstellten Canvas in der aktuellen Form in eine Webapp umwandeln kann, oder ob gewisse Bestandteile noch überarbeitet werden müssen. Im Folgenden wird ein Überblick darüber gegeben, welche Checks die DSL beinhaltet, um diese Form der Prüfung vorzunehmen:

EdgeWithoutPath

Wird ein Pfad zwischen zwei Feldern im Canvas gezogen, um dort eine Beeinflussung herzustellen, so muss dort vom Nutzer auch ein Suchpfad innerhalb des Path hinterlegt werden. Ohne einen Suchpfad kann nicht aufgelöst werden, zwischen welchen Attributen die Verbindung zwischen den beiden miteinander verbundenen Contents besteht. Wird keiner dieser Suchpfade zu einen gezogenen Path eingegeben, so wirft dieser Check einen Error.
In der Umsetzung wird für diesen Check zunächst in der DSL nach allen Path-Werten gesucht. Jeder davon enthält eine Liste, bestehend aus Strings, die die einzelnen Werte für einen Path abbildet. Damit der Check ein positives Ergebnis liefert, muss in dieser Liste mindestens ein Element enthalten sein.

FieldIsEmpty

Werden Felder in der DSL hinzugefügt, haben diese den Sinn, die Typen aus der im Projekt befindlichen Ontologie aufzunehmen. Ein Feld, in das kein Typ eingefügt wird, kann auch in der generierten Webapp nicht weiter befüllt werden und erfüllt somit keine sinnvolle Aufgaben. Allerdings stört es den Ablauf der Generierung auch nicht. Ein leeres Feld erzeugt durch diesen Check daher nur eine Warnung und keinen Error.
In der Umsetzung wird für diesen Check zunächst in der DSL nach allen Field-Elementen gesucht. Aus einem Field können dann sowohl die konkreten Elemente jeweils aus einer Liste ausgelesen werden, die aus einer Ontology in den Canvas übertragen wurden, als auch die im Field vorkommenden Freitexte und Notizen, sofern diese verwendet werden. Dieser Check liefert ein positives Ergebnis, wenn mindestens eine dieser Listen nicht leer ist und das Field damit ein beliebiges Element enthält.

FieldNamesEmpty

Ein Feld wird durch seinen Namen identifizierbar eingestellt. Innerhalb der Cinco Properties kann einem Feld ein Name gegeben werden. Dieser Name wird auch in der generierten Webapp als Bezeichner für das Feld verwendet. Dieser Name darf nicht leer gelassen wer-

den. Passiert das trotzdem, wird dies durch diesen Check als Fehler angezeigt.
In seiner Umsetzung sucht dieser Check zunächst nach allen in der DSL vorkommenden Fields. Von jedem Field wird der ausgelesen, sollte kein leeres Name-Attribut gefunden werden, wird von dem Check ein positives Ergebnis ausgegeben.

FieldsDifferentNames

Die Namen, die den Feldern gegeben werden, sollen ein Feld eindeutig identifizieren. Dafür dürfen in einem Canvas keine zwei oder mehr Felder mit dem gleichen Namen vorliegen. Geschieht das trotzdem, wird durch diesen Check ein Fehler angezeigt.
Der Check durchläuft der Reihe nach alle in der DSL vorhandenen Fields. Die Namen der Fields werden ausgelesen und mit allen anderen Namen der sonstigen vorhandenen Fields vergleichen. Wird für kein Field ein weiteres mit identischem Namen gefunden, so liefert der Check ein positives Ergebnis.

MultipleEdgeSameDirection

Wird zwischen mehren Feldern des Canvas ein Pfad gezogen, so muss dieser eindeutig sein. Es dürfen keine zwei oder mehr Pfade in identische Richtung gezogen werden. Es können aber zu einem Pfad mehrere Pfadwerte hinzugefügt werden. Der Versuch, mehrere identische Pfade hinzuzufügen, sorgt dafür, dass dieser Check einen Fehler anzeigt.
Für diesen Check wird nacheinander jedes Field, bzw. jeder Type innerhalb eines Fields durchlaufen. Relevant sind diese Types, die zwei oder mehr eingehende Kanten haben. Von jedem Type mit mehreren eingehenden Kanten werden diejenigen Types ausgelesen, von denen die betreffenden Kanten ausgehen. Handelt es sich bei diesen um jeweils voneinander verschiedene Types, sind alle Kanten gültig und der Check liefert ein positives Ergebnis.

MultipleFreetextWithSameContent

Freitexte können im Canvas in jedes vorhandene Feld eingefügt werden. Der Inhalt von einem Freitext kann beliebig erweitert werden. Werden in einem Field mehrere Freitexte hinzugefügt, die den identischen Inhalt haben, wird durch diesen Check eine Warnung angezeigt. Die Ausführung der Webapp würde dadurch nicht beeinflusst werden.
Für diesen Check werden für jeweils ein im Canvas vorhandenes Field alle Freitexte ausgelesen. Jeder dieser Freitexte wird in seinem Inhalt mit allen anderen im diesem Field befindlichen Freitexten vergleichen. Finden sich in einem Field keine zwei oder mehr Freitexte mit identischem Inhalt, so liefert dieser Check ein positives Ergebnis.

PathHasValidSemantik

Die Pathwerte, die zu einem Pfad hinzugefügt werden, orientieren sich in ihrem Inhalt an den Typen, die in der zugehörigen Ontologie vorhanden sind. Durch diesen Check wird

geprüft, ob alle in einem Path verwendeten Namen für Typen auch einem Typ aus der Ontologie entsprechen. Entsprechend werden auch einfache Rechtschreibfehler durch diesen Check gefunden und darauf hingewiesen.
Die Durchführung dieses Checks teilt sich in mehrere Abschnitte auf. Zunächst werden für jeden Path alle Pathwerte ausgelesen. Jeder Pathwert besteht zu dem Zeitpunkt sowohl aus den für den Check relevanten Namen von Attributen der zugehörigen Ontologie, als auch aus . und { } Zeichen. In der Abarbeitung des Checks wird der aktuell betrachtete zunächst um die nicht benötigten Symbole reduziert. Als Ergebnis liegt zu dem Zeitpunkt eine Liste mit String-Werten vor, die möglichen Namen von Attributen entsprechen. Im einem weiteren Schritt werden diese Werte nacheinander mit der Gesamtmenge aller in der Ontologie vorhandenen Attributnamen abgeglichen. Ist jeder der Namen aus allen Path-Werten in der Ontologie vorhanden, so liefert der Check ein positives Ergebnis.

PathHasValidSyntax

Die Pathwerte, die zu einem Pfad hinzugefügt werden, orientieren sich an einer geforderten Syntax. Diese besteht auch einer Anordnung von den Namen der Typen, aus der zugehörigen Ontologie, dazu sowohl Punkten als auch Klammern. Nur wenn diese in der korrekten Anordnung verwendet werden, kann ein Pfad in seiner weiteren Verwendung korrekt weiter verarbeitet werden. Ist dies nicht der Fall, wird durch diesen Check ein Fehler angezeigt.
In der Ausführung dieses Checks werden zunächst für jeden vorhandenen Path alle Pathwerte ausgelesen. Um die korrekte Syntax zu prüfen, wird jeder der Pathwerte danach durch getrennte Methoden zunächst auf das korrekte Vorkommen der Punkte, sowie auf das korrekte Vorkommen der Klammern überprüft. Für den Check der Punkte wird der Pathwert für jeden Char durchlaufen. Dieser muss zunächst einmal zu Beginn, sowie zum Ende, einen Punkt beinhalten. Für alle anderen Positionen gilt, dass ein Punkt immer direkt hinter einer geschlossenen Klammer, sowie vor einer geöffneten Klammer steht. Ist das für die komplette Länge des Pathwertes der Fall, wird dieser noch auf das korrekte Vorhandensein der Klammern überprüft. Ähnlich zu den Punkten wird der Pathwert dafür Charweise durchlaufen. Über die komplette Länge hinweg, werden sowohl die geöffneten, als auch die geschlossenen Klammern gezählt. Diese Zahlen müssen als ein Kriterium am Ende der Überprüfung übereinstimmen. Wird an einer Position ein Buchstabe erkannt, so muss die Bedingung gelten, dass sich diese als ein Teil eines Attributnamens nur zwischen jeweils zwei offenen und geschlossenen Klammern befinden dürfen. Sind beide Bedingungen, sowohl für die Punkte, als auch für die Klammern, erfüllt, so liefert der Check ein positives Ergebnis.

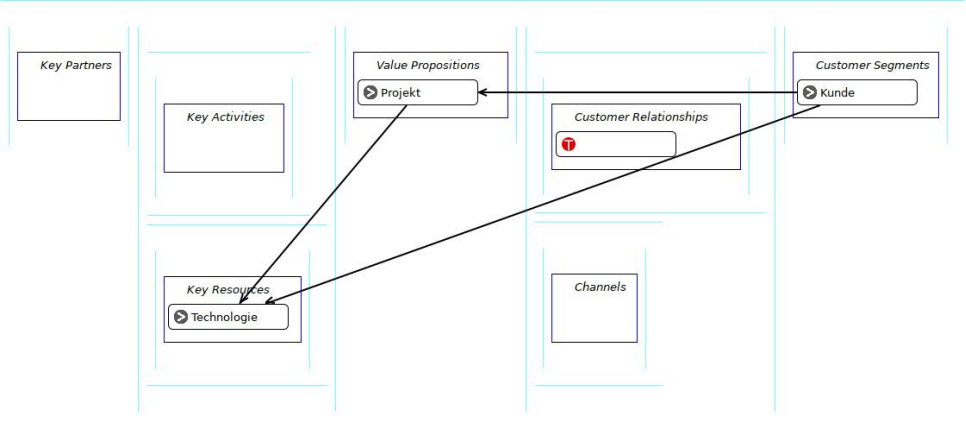


Abbildung 6.1: Beispielcanvas

ValidTransparencyValue

Wird ein Feld oder ein darin enthaltenes Element mit einem Farbwert belegt, so kann neben der Farbe auch die Transparenz der Farbe ausgewählt werden. Der Wert, der die Transparenz regelt, muss in einem bestimmten Bereich liegen. Dadurch, dass dieser Wert in einem Freitextfeld eingegeben wird, ist es theoretisch auch möglich, einen Wert einzugeben, der keinem gültigen Wert für eine Transparenz zuzuordnen ist. In diesem Fall wird der Check einen Fehler anzeigen, dass der Wert zu korrigieren ist.
In der Ausführung, werden für diesen Check alle Fields, sowie die sich darin befindlichen Types der Ontologie, durchlaufen. Der Wert für die eingetragene Transparenz kann dabei als Double-Wert aus jedem der Elemente ausgelesen werden. Dieser kann dann mit den Randbedingungen von 0.0 und 1.0 abgeglichen werden. Befindet sich jeder in der DSL verwendete Wert für eine Transparenz innerhalb dieser Grenzwerte, so liefert der Check ein positives Ergebnis.

6.1.4 Beispielhafte Umsetzung eines Canvases in der Canvas-DSL

Ein Beispielcanvas ist in Abbildung 6.1 zu sehen. In diesem wurde ein Canvas durch das Verschachteln von Rows und Columns zu einem Canvas mit neun Feldern erstellt. Es handelt sich um den Canvas nach dem Modell von Osterwalder. Dementsprechend beeinhalten die neun Felder ein Field mit dem dafür vorgesehenen Namen.
In dem Beispiel sollen Projekte in die Spalte „Value Proposition“ eingefügt werden. Daher muss aus der Ontologie der Content „Projekt“ in das Feld hineingezogen werden. Außerdem soll im Feld „Key Resources“ die Technologien ausgewählt und im Feld „Customer Segment“ die Kunden hinzugefügt werden. Alle drei Contents sind mit Pfaden verbunden, die angeben, wie man vom Startcontent über die Attribute zum Zielcontent gelangt. Beispielhaft ist die Beschriftung des Paths in 6.2 zu sehen. Außerdem befindet sich in dem Field „Customer Relationships“ ein Freitext.

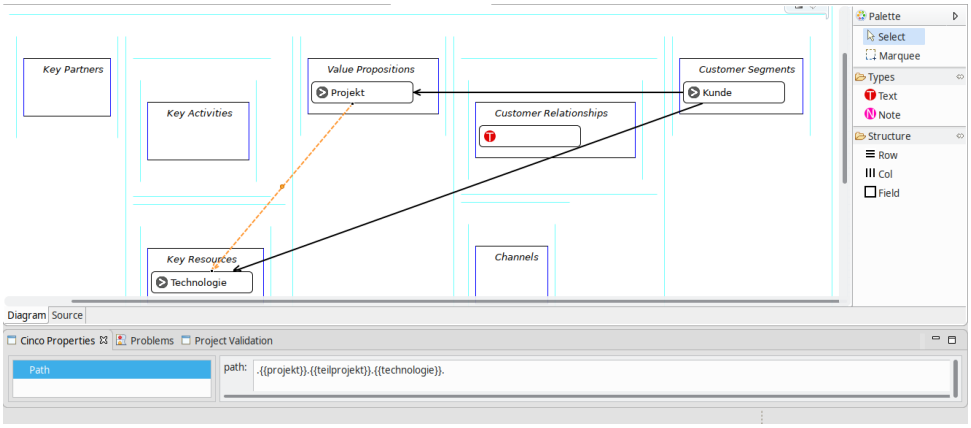


Abbildung 6.2: Beispielcanvas: Unten wird der Pfad mit dem Inhalt `.{{projekt}}.{{teilprojekt}}.{{technologie}}.` zwischen „Projekt“ und „Technologie“ angezeigt.

Der Canvas ist als `strict` eingestellt, was bedeutet, dass die Beeinflussung der Objekte stattfindet.

6.2 Transformator

Im Folgenden wird die Implementierung der Generatoren erläutert. Dafür wird zunächst auf die CanvasGeneratorFactory eingegangen. Danach wird der Generator für das Datenmodell und zuletzt die GUI- und Prozess-Generatoren dargestellt. Dabei werden die komplexen Generatoren genauer erklärt.

6.3 CanvasGeneratorFactory

Die CanvasGeneratorFactory erstellt die einzelnen Modelle für konkrete Canvase. Dazu werden die Generatoren in Gruppen zusammengefasst. Es gibt Generatoren, die unabhängig vom Canvas erzeugt werden, wie beispielsweise der EditNoteGUIGenerator. Diese Generatoregruppe wird in der Methode `createCanvasIndependentGenerators` erstellt. Die Methode `createTypeGenerators` kreiert die Prozesse, die für jeden Content in einem Field benötigt werden. Das sind beispielsweise die `AddModalViewGUI`, der `SubProcess`, der `AddInteractProcess` oder der `ManagementProcess`. Dafür müssen über die Methode `findAllContents` die angegebenen Contents aus dem Canvas extrahiert werden.
Im Anschluss daran werden Modelle, die nur einmal pro Canvas benötigt werden, generiert. Das sind beispielsweise der `LoadDataForCanvasProcess`, der `CanvasUpdateAllProcess`, der `CanvasProcess` oder der `CanvasPreChoiceProcess`.
Für alle Felder in einem Canvas werden danach die Notizprozesse in der Methode `createTypeGeneratorsNote` erzeugt.
Um die Prozesse für die Suche zu generieren, werden zuerst die Pfade analysiert. Danach

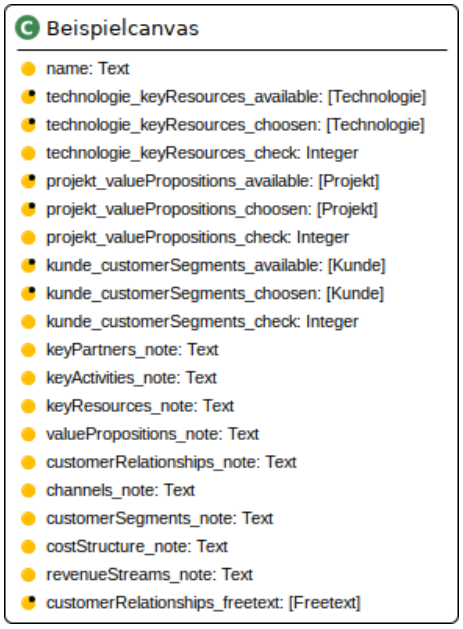


Abbildung 6.3: Beispielcanvas-Objekt in dem Data-Modell

werden mit der Methode `searchStartEndPath` Tripel gebildet. Der Methode werden alle verwendeten Contents übergeben und für jeden die Pfade ausgelesen. Das Tripel bildet dabei der Start-, der Endtyp und der Pfad als String.
Auf dieser Grundlage werden kleinere Tripel mit der Methode `getTripleFromCanvas` gebildet. Ein Tripel ist auf einem Pfad der Typ des Startattributs und der Typ des darauf folgende Attributs. Zusätzlich wird dem Paar der Attributname als String übergeben. Beispielsweise ergeben sich für den Pfad „.{{projekt}}.{{teilprojekt}}.{{technologie}}“die Tripel (Projekt, Teilprojekt, „teilprojekt“) und (Teilprojekt, Technologie, „technologie“). Mit diesen Tripeln kann der `SearchProcessGenerator` die benötigten Prozesse generieren. Danach werden die Listenprozesse `AddListProcess` und `AddListIntersectProcess` für jeden Typ über die Methode `createTypeGeneratorsUnique` gebildet und über die Methode `createTypeGeneratorsPathComplete` die GetAll-Prozesse generiert.

6.4 DataModelGenerator

Der DataModelGenerator übernimmt das generierte Datenmodell der Ontologie. Für den Canvas müssen dort einige konkrete Typen hinzugefügt werden. Für jeden Canvas, der generiert werden soll, wird ein eigener `ConcreteType` erstellt. Dieser enthält pro verwendeten Content zwei Listenattribute. Die erste Liste speichert die ausgewählten Objekte, die Zweite die noch verfügbaren Elemente. Außerdem wird ein primitives Attribut für die Priorität pro Content angelegt. Für das in Kapitel 6.1.4 beschriebene Beispiel ist der ConcreteType BeispielCanvas in Abbildung 6.3 zu sehen.
Um gleiche Attributnamen zu verhindern, wird der Name des Attributs auf

`[Contenttype]_[Fieldname]` gesetzt. Für eine eindeutige Identifikation wird diesem Namen `_available` für die Verfügbarliste, `_chosen` für die Auswahlliste und `_check` für die Priorität angehängt.
Pro erstelltem Field müssen die Notizen abgespeichert werden können. Dazu wird ein `[Fieldname]_note` Attribut vom Typ `Text` als Liste erzeugt.
Außerdem wird ein `Freetext` als `ConcreteType` erzeugt. Dieser enthält das Attribut `text` vom Typ Text. Es wird für die Freitexte verwendet, die im Canvas angelegt werden können. Wird ein solches in einem Field verwendet, wird in dem passenden Canvas-Typ ein Attribut mit dem Namen `[Fieldname]_freetext` erzeugt das vom Typ `Freetext` ist.
Der Canvastyp bekommt zusätzlich das Attribut `name`, damit beim Anlegen eines Canvas dieser angezeigt wird.
Mit einem Objekt von diesem Typen kann ein Canvas abgespeichert und angezeigt werden.

6.5 Übersicht Prozess- und GUI-Generatoren

Im Folgenden werden alle Generatoren mit der jeweiligen Funktion aufgeteilt in Prozess- und GUI-Generatoren erklärt. Dabei wird genauer auf die `CanvasGUI-`, `CanvasProcess-`, `InteractAddProcess-`, `SubInteractProcess-`, `ManagementProcess-`, `UpdateProcess-`, `SearchProcess-` und `CanvasCorrectPriorityProcessGenerator` eingegangen.

6.5.1 Prozess-Generatoren

Es wurden folgende Prozess-Generatoren erstellt:

- *CanvasPreChoiceProcessGenerator*
  Der durch den `CanvasPreChoiceProcessGenerator` erstellte Prozess ist der erste, der aufgerufen wird. Dieser verwaltet die Canvasübersicht in der alle Canvase geladen, angezeigt, bearbeitet, erstellt und gelöscht werden können.

- *LoadCanvasProcessGenerator*
  Der daraus resultierende Prozess lädt alle erstellten Canvase eines bestimmten Canvastyps aus der Datenbank.

- *CanvasCreateProcessGenerator*
  Mit dem Prozess des `CanvasCreateProcessGenerator` wird ein neuer Canvas erstellt und alle Objekte, die in diesem verwendet werden können, in das neu erstellte Canvasobjekt geladen.

- *CanvasUpdateProcessGenerator*
  In dem `CanvasUpdateProcess` werden alle Objekte aus der Datenbank für den Canvas geladen, in die Auswahlliste geschrieben und mit dem `UpdateDimeTypeProcess`

aktualisiert werden, sodass die Auswahl- und Verfügbarliste richtig verteilte Objekte enthält.

- *RemoveCanvasGenerator*
  Ein ausgewählter Canvas wird aus der Liste der vorhandenen Canvase und aus der Datenbank gelöscht.

- *LoadDataForCanvasProcessGenerator*
  Dieser Generator erstellt einen Prozess, der alle Objekte aus der Datenbank lädt, die im Canvas verwendet werden können.

- *CanvasProcessGenerator*
  Dieser Prozess verwaltet den geöffneten Canvas. Alle Funktionalitäten, die im Canvas vorgenommen werden, werden hier verwaltet. Näheres wird in Kapitel 6.5.4 erklärt.

- *CallAddModalViewProcessGenerator*
  Mit diesem Prozess, der für jeden Content erzeugt wird, wird die modale Hinzufügen-Ansicht verwaltet. Dieser ruft die GUI-Ansicht auf und löscht das ausgewählte Objekt aus der Verfügbarliste.

- *InteractAddProcessGenerator*
  Der `InteractAddProcessGenerator` wird ausführlicher in Kapitel 6.5.5 erklärt. Dieser Prozess wird für jeden Content erzeugt und es wird die Verfügbarliste und Priorität aktualisiert. Nach dem Hinzufügen werden die noch verfügbaren Objekte der anderen Contents angepasst.

- *SubInteractProcessGenerator*
  Analog zum InteractAddProcessGenerator werden die Verfügbarlisten der anderen Contents nach dem Entfernen eines Objekts aus dem Canvas aktualisiert.

- *SubProcessGenerator*
  Nach dem Löschen eines Objektes aus dem Canvas wird das Element aus der Auswahlliste entfernt und der Verfügbarliste hinzugefügt.

- *CanvasSearchMaxProcessGenerator*
  Dieser Prozess geht alle Contents durch und gibt das Maximum der vergebenen Prioritäten zurück.

- *ManagementProcessGenerator*
  Der `ManagementProcessGenerator` generiert den Prozess für jeden Content aus der Canvas-DSL. Für jedes konkrete Objekt des Contents werden alle verfügbaren Objekte anderer Contents entlang des Suchpfades zurückgegeben. Genauer wir der Prozess in Kapitel 6.5.7 erklärt.

- *CanvasUpdateAllProcessGenerator*
  Der CanvasUpdateAll-Prozess aktualisiert die Verfügbarliste. Entlang der Prioritäten werden die Listen gefiltert und Objekte, die zu der Auswahl nicht passen, entfernt. In Kapitel 6.5.8 wird das näher erläutert.

- *UpdateDimeTypeProcessGenerator*
  Im `UpdateDimeTypeProcess` muss für jedes konkrete Objekt die Verfügbar- und Auswahlliste richtig gesetzt werden. Sobald ein Objekt in der Auswahlliste enthalten ist, wird es aus der Verfügbarliste gelöscht. Analog verhält es sich mit der Verfügbarliste, aus der Objekte gelöscht werden, wenn sie in der Auswahlliste enthalten sind. So entstehen zwei disjunkte Mengen.

- *SearchProcessGenerator*
  Der `SearchProcessGenerator` wird für jedes Paar generiert über das mit einem Attribut gesucht wird. Das Zielobjekt, welches zurückgegeben wird, ist das Objekt, welches vom Startobjekt über das angegebene Attribut gefunden wurde. Ausführlicher wird das in 6.5.10 erläutert.

- *GetAllProcessGenerator*
  Der `GetAllProcess` wird für alle Contents erzeugt und sucht pro Attribut über den passenden SearchProcess entlang des angegebenen Pfades. Der Typ am Ende des Pfades wird zurückgegeben. Für alle Pfade, die mit dem Content verbunden sind, werden die passenden Objekte gesucht und zurückgegeben.

- *CanvasCorrectPriorityProcessGenerator*
  Der `CanvasCorrectPriorityProcess` wird einmal generiert und passt nach der Auswahl die Prioritäten richtig an. Näheres ist in Kapitel 6.5.9 beschrieben.

- *CanvasResetProcessGenerator*
  Der `CanvasResetProcessGenerator` erzeugt den Resetprozess, der in einem Canvas die Auswahlliste löscht und die enthaltenen Objekte in die Verfügbarliste einfügt. Außerdem werden die Prioritäten auf null gesetzt.

- *CanvasClearListProcessGenerator*
  Mit dem `CanvasClearListProcess` wird die Auswahlliste gelöscht. Dieser Prozess wird bei dem CanvasResetProcess verwendet.

- *FreeTextModalViewProcessGenerator*
  Der `FreeTextModalViewProcess` wird einmal generiert. Der Prozess verwaltet übergebene Freitexte. Somit können in der aufgerufenen `CanvasFreeTextViewGUI` Texte erstellt, bearbeitet oder gelöscht werden.

- *NoteModalProcessGenerator*
  Der `NoteModalProcessGenerator` verhält sich ähnlich wie der `FreeTextModalViewProcessGenerator`. Zu jedem Field wird ein Notizfeld erstellt, welches mit der darin enthaltenen Modalgui `NoteViewGUI` verwaltet wird. Die Notiz kann erstellt, bearbeitet oder gelöscht werden.

- *AddListIntersectionProcessGenerator*
  Der Generator erzeugt für jeden verwendeten Typen den `AddListIntersectionProcess`. Dieser bildet bei zwei gegebenen Listen den Schnitt der Menge.

- *AddListProcessGenerator*
  Der für jeden verwendeten Typen generierte Prozess vereint zwei Listen.

6.5.2 GUI-Generatoren

Es wurden folgende GUI-Generatoren entwickelt:

- *CanvasAdministrationGUIGenerator*
  Der Generator erstellt die `CanvasAdministrationGUI`. Diese GUI verwaltet die erstellten Canvase. So können Neue erstellt und Bestehende bearbeitet oder gelöscht werden. Außerdem kann ein Canvas geöffnet werden.

- *EditCanvasGUIGenerator*
  Die `EditCanvasGUI` wird aufgerufen, wenn ein Canvas bearbeitet wird. Es kann der Name des Canvas verändert werden.

- *CanvasGUIGenerator*
  Der `CanvasGUIGenerator` erzeugt aus dem erstellten Canvas die GUI des Canvas. In dieser Ansicht können Objekte ausgewählt, gelöscht, Notizen oder Freitexte erstellt oder der Canvas zurückgesetzt werden. Ausführlicher wird der Generator in Kapitel 6.5.3 erklärt.

- *AddModalViewGenerator*
  Der Generator erstellt für jeden Content die `AddModalView`. Diese wird aufgerufen, wenn im Canvas ein Objekt hinzugefügt werden soll. Alle Elemente der Verfügbarliste werden angezeigt und der Benutzer kann Objekte dem Canvas hinzufügen.

- *CanvasFreeTextViewGUIGenerator*
  Die `CanvasFreeTextViewGUI` zeigt alle erstellten Freitexte in einer Tabelle an. Die Freitexte können gelöscht oder bearbeitet werden. Außerdem gibt es einen Erstellen-Button, über den ein neuer Freitext erstellt werden kann.

- *FreeTextEditGUIGenerator*
  Mit der `FreeTextEditGUI` wird ein vorhandener Freitext angezeigt und kann in einem Feld verändert werden.

- *NoteViewGUIGenerator*
  Die `NoteViewGUI` ist ähnlich zu der `CanvasFreeTextViewGUI`. Es können Notizen angezeigt, bearbeitet oder gelöscht werden.

- *EditNoteGUIGenerator*
  Wird eine Notiz bearbeitet, öffnet sich die `EditNoteGUI`. In dieser ist ein Feld, in dem der Inhalt der Notiz angezeigt wird und verändert werden kann. Mit dem Speichern-Button wird der neue Inhalt übernommen. Bei dem Löschen-Button wird der Inhalt auf eine leere Zeichenkette gesetzt.

6.5.3 CanvasGUIGenerator

Der `CanvasGUIGenerator` generiert für jeden Canvas eine GUI, in der der geöffnete Canvas zusehen ist. In Abbildung 6.4 ist der fertige GUI-Prozess für den Beispielcanvas dargestellt. Um diesen Canvas so zu generieren, werden zunächst alle Rows im vordefinierten Canvas durchgegangen. Rekursiv werden die Spalten und Zeilen bis zu einem Field durchsucht. Für jede Spalte wird in dem generierten Prozess eine Col und für jede Zeile eine Row eingefügt. Ist ein Field gefunden, so wird das Field-Pattern, welches in Abbildung 6.5 dargestellt ist, generiert. Dieses besteht aus einem Panel, in dem sich eine Row für den Kopf und ein Panel darunter für die Contents befindet. In der Kopfzeile wird, sofern `includeFieldnameAsHeader` in der DSL gesetzt wurde, der Fieldname als Headline eingefügt. Bei einer Farbauswahl wird die Farbe ebenfalls gesetzt. Daneben sind zwei Prozesse eingefügt, die die Notizen darstellen. Beide Prozesse beeinhalten den `CanvasNoteModalProcess`-Prozess. Abhängig von vorhandenen Notizen wird der linke oder rechte Prozess dargestellt. Dieser setzt das Notiz-Symbol entweder auf die Farbe rot, wenn keine Notiz vorhanden ist oder auf grün, wenn eine Notiz enthalten ist.
In dem Panel, das pro Content erstellt wird, befindet sich eine Row, die den `AddModalViewProcess` als Modalansicht einfügt und rechts daneben den Namen des Contents enthält. Unter dieser Zeile befindet sich die Darstellung der aktuellen Priorität mit einem Text. Daran schließt sich die Tabelle an, in der alle ausgewählten Contents dargestellt werden. In dieser werden die Anzeige des Identifiers und eine zusätzliche Spalte mit einem Button generiert. Der Button löscht ausgewählte Objekte aus der Auswahlliste.
Bei Freitexten verhält es sich analog.
Abhängig von der Anzahl der Spalten, die benutzt werden, wird die Spaltenweite auf `Full` (Spaltenanzahl gleich eins), `Half` (Spaltenanzahl gleich zwei), `Quarter` (Spaltenanzahl gleich vier), `Sixth` (Spaltenanzahl gleich sechs) oder `Twelfth` (sonst) gesetzt.

Abbildung 6.4: GUI des Beispielcanvas als DIME-GUI

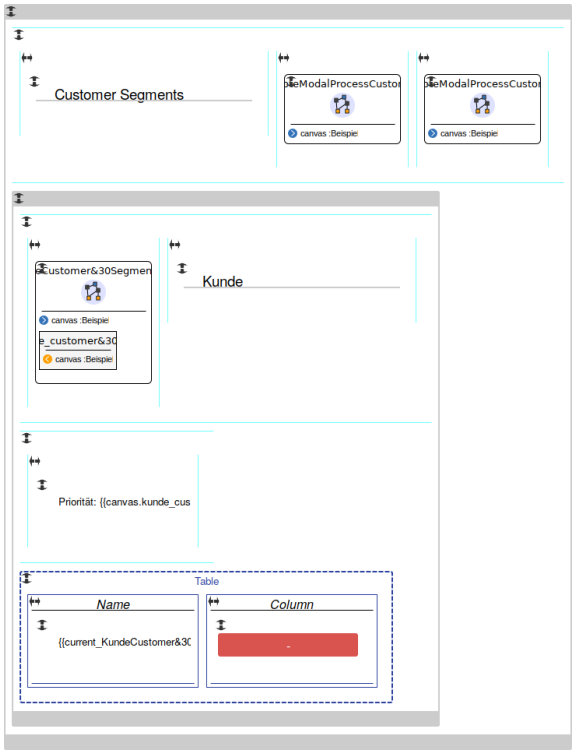


Abbildung 6.5: Feld „Customer Segment“ des Beispielcanvas als Beispiel für das Field-Pattern

Zusätzlich zu den Fields müssen in dem Canvas der Reset-Button und der Lösch-Button erstellt werden. Diese werden links unten eingefügt.

6.5.4 CanvasProcessGenerator

Der `CanvasProcessGenerator` generiert den CanvasProcess. Dieser verwaltet alle Funktionen, die in der GUI vorgenommen werden. Der Generator lässt sich in zwei Teile unterteilen. Zunächst wird die Startphase generiert, in der alle Objekte geladen und die Prioritäten gesetzt werden. Danach wird die GUI aufgerufen, aus der das Zurücksetzen des Canvas möglich ist. Der zweite Teil besteht aus dem Generieren der Content-Funktionen, die aus dem Hinzufügen und Entfernen von Objekten besteht.
Zu der Startphase zählt das Laden des Canvases mit dem `CanvasUpdateAllProcess` und dem Aufrufen der CanvasGUI. Der Reset wird nach dem Drücken des Reset-Buttons mit dem `CanvasResetProcess`-Prozess und dem `LoadDataForCanvasProcess` abgeschlossen. Durch den letzten Prozess werden die Auswahl- und Verfügbarlisten neu gesetzt.
Pro Content wird das Hinzufügen und Löschen generiert. Bei dem Hinzufügen wird nach dem entsprechenden Aktivieren des Buttons der `InteractAddProcess`-Prozess aufgerufen. Beim Löschen wird der passende `Sub-` und `SubInteractProcess`-Prozess generiert und verbunden.
Für Freitexte verhält es sich analog.

6.5.5 InteractAddProcessGenerator

Beim `InteractAddProcessGenerator` muss zunächst eine If-Abfrage erstellt werden, die prüft, ob die vergebene Priorität null oder größer als null ist. Ist sie null, wurde noch kein Element von diesem Content hinzugefügt. Dementsprechend muss eine neue Prioritätenzahl für den Content mit dem `CanavsSearchMaxProcess` gefunden, um eins erhöht und für den Content gesetzt werden. Danach wird der `ManagementProcess` aufgerufen.
Er wird außerdem aufgerufen, wenn die If-Abfrage auf null `false` ergibt. Das bedeutet, dass schon eine Priorität vergeben ist und diese nicht neu gesetzt werden muss.
Der `ManagementProcess` gibt pro beeinflussten anderen Content eine boolsche Variable zurück, die angibt, ob die Liste neu gesetzt werden muss. Außerdem wird eine Liste zurückgegeben, die die neue Verfügbarliste darstellt. Somit muss für jedes Paar, dass zurückgegeben wird die boolsche Variable überprüft werden und falls diese `true` ist, mit einem PutToContextSIB gesetzt werden. Danach wird der Typ mit dem `UpdateDimeTypeProcess` aktualisiert.

6.5.6 SubInteractProcessGenerator

Der `SubInteractProcess` ist in mehrere Teile unterteilbar. Zunächst gibt es den Teil, wenn die Größe der Auswahlliste des aufgerufenen Contents auf null ist. Wenn die anderen Contents ebenfalls den Wert null haben, muss die Priorität für den gelöschten Content null sein.
Wenn ein anderer Content eine Priorität größer null hat, muss mit `PutToContextSIBs` und `IntegerEqualsSIBs` die Priorität des aufgerufenen Contents abgefragt werden. Diese kann zwischen eins und der Anzahl der definierten Contents aus der DSL liegen. Ist die passende Priorität gefunden, muss überprüft werden, ob die anderen Contents eine Priorität größer oder gleich der gefundenen Priorität haben. Sollte dies der Fall sein, muss die Priorität um eins erniedrigt werden, da der Content, der den `SubInteractProcess` aufgerufen hat, nicht mehr in der Prioritätenliste aufgeführt wird, weil seine Listengröße gleich null ist. Falls die neue Priorität der anderen Contents eins wird, werden alle Objekte, die in der Datenbank sind, in die Verfügbarliste geladen. Andernfalls wird die Priorität des aufgerufenen Contents auf null gesetzt und der nächste Teil des Prozesses beginnt.
Für jeden Content, der eine höhere Priorität als die des aufgerufenen Contents hat, muss die Auswahl angepasst werden. Dafür werden mit dem `ManagementProcess` die passenden Listen erstellt und im `SubInteractProcess` gesetzt.
Ist die Anfangsliste nicht gleich null, sondern größer, werden mit dem `ManagementProcess` die Listen der anderen Contents neu gesetzt.

6.5.7 ManagementProcessGenerator

Der `ManagementProcess` für einen bestimmten Content überprüft, ob der Content die Priorität eins bis Anzahl der verwendeten Contents hat. Daraus entstehen Reihen, die jede dieser Zahlen mit einem `IntegerEqualsSIB` vergleichen. In einer Reihe wird bei einem `true` über jedes konkrete Element des Contents iteriert und mit dem `GetAllProcess` alle Objekte, die mit diesem assoziiert werden, gefunden. Daraus ergeben sich die neuen Verfügbarlisten. Im Anschluss daran wird für jede Priorität, die höher ist als die des Contents, mit dem passenden `ManagementProcess` die Verfügbarliste der Elemente bestimmt, die von dem Content beeinflusst werden.

6.5.8 UpdateAllProcessGenerator

Der `UpdateAllProcess` wird einmal im `CanvasProcess` benutzt. Er setzt alle Verfügbar- und Auswahllisten richtig und aktualisiert die Prioritäten.
Zunächst werden alle Objekte der verwendeten Contents aus der Datenbank geladen und zwischengespeichert. Von dort werden alle Verfügbarlisten anhand der Auswahlliste neu berechnet. Um das zu erreichen werden alle Prioritäten auf die Werte null bis zur Anzahl der verwendeten Contents geprüft.
Wenn alle Prioritäten den Wert null haben, ist kein Objekt ausgewählt. Die zuvor geladenen Objekte werden in die Verfügbarliste geschrieben und der Prozess ist beendet.
Haben nicht alle Contents den Prioritätenwert null, muss je nach gesetzter Priorität die Verfügbarliste neu berechnet werden. Dafür werden alle Prioritäten von eins bis zu der Zahl vor der Anzahl der verwendeten Contents abgefragt. Ist die Priorität beispielsweise für einen Content gleich eins, werden alle Elemente aus der Datenbank in die Verfügbarliste mit einem `PutToContextSIB` geladen. Danach wird der entsprechende Content mit dem `DimeTypeUpdateProcessSIB` aktualisiert, sodass die Auswahl- und Verfügbarliste richtig gesetzt sind. Für jedes Element aus der gesetzten Auswahlliste müssen die beeinflussten Objekte berechnet werden. Das geschieht mit einer Schleife, die für jedes Objekt den `GetAllProcess` aufruft und die resultierenden Ausgaben in einer Liste mit dem `AddListProcess` speichert. So entsteht, nachdem die Schleife beendet wurde, eine Liste mit allen Elementen von einem Content, die durch die aktuelle Auswahl beeinflusst wurden.
Sind alle Contents auf eins abgefragt, werden die nun entstandenen Listen in die Verfügbarliste des Canvas kopiert und die einzelnen Contents mit dem `DimeTypeUpdateProcessSIB` aktualisiert. Die zwischengespeicherten Objekte werden mit dem `CanvasClearListProcess` geleert, sodass keine Objekte enthalten sind. Zuletzt werden die `write`-Variablen auf `false` gesetzt. Nun kann die nächste Priorität, in dem Beispiel zwei, abgefragt werden, welches nach einem ähnlichen Schema abläuft. Die Schleife wird erweitert, sodass ein Content, der die Priorität zwei hat, nur die Listen aus dem `GetAllProcess` in die Verfügbarliste schreibt, wenn diese Contents eine Priorität größer zwei haben. Falls die Priorität gleich null ist,

wird die Liste ebenfalls angepasst. Dies wird durch einen `IntegerEqualsSIB` abgefragt werden. Das ist in Abbildung 6.6 schematisch dargestellt.
Zum Schluss wird für jede write-Variable überprüft, ob die Verfügbarliste neu beschrieben werden muss. Dies geschieht mit einem
`BooleanSwitchSIB`, einem `PutToContextSIB` und dem `DimeTypeUpdateProcess` und ist in Abbildung 6.7 abgebildet.

Bevor der Prozess beendet wird, müssen die Prioritäten richtig gesetzt werden. Das geschieht durch den `CanvasCorrectPriorityProcess`.

6.5.9 CanvasCorrectPriorityProcessGenerator

Um Prioritäten nach veränderten Auswahl- oder Verfügbarlisten richtig zu setzen, wird der `CanvasCorrectPriorityProcess` aufgerufen. Dieser startet zunächst damit, dass boolsche Hilfsprioritäten auf `false` gesetzt werden. Anhand dieser wird entschieden, ob ein Content die Priorität besitzt.
Dann wird für jede Zahl von null bis zur Anzahl der gesetzten Contents eine Zeile erzeugt, die alle verwendeten Contents auf die Zahl abfragt. Bei der Abfrage auf null wird die Auswahllistengröße abgefragt, bei den anderen Zahlen der jeweilige Contentcheck.
Enthält die Auswahlliste kein Element, wird die Priorität mit einem `PutToContext` auf null gesetzt. Die anderen Werte überprüfen, ob eine Zahl in den vergebenen Prioritäten vorhanden ist. Beispielsweise bei der Zeile mit dem `PutToContext` zwei wird überprüft, ob eine Priorität den Wert Zwei hat. Sollte dies der Fall sein, muss eine weitere Priorität vorhanden sein, die den Wert eins hat. Das wird im Anschluss durch die passende boolsche Hilfsvariable abgefragt. Ist das der Fall wird die boolsche Hilfsvariable `Zwei` auf `true` gesetzt, ansonsten muss die gefundene Priorität herabgesetzt und auf eins gesetzt werden. Die boolsche Hilfsvariable `Eins` wird dann auf `true` gesetzt.
Analog passiert das für alle weiteren Zeilen, bis keine Zahl mehr überprüft werden muss. Danach wird der Prozess beendet.

6.5.10 SearchProcessGenerator

Der `SearchProcessGenerator` wird für jedes Paar aufgerufen, dass über einen angegebenen Suchpfad zu erreichen ist. Aus dem in Abschnitt 6.1.4 dargestellten Beispiel gibt es den Pfad `.{{projekt}}.{{teilprojekt}}.{{technologie}}.`. Daraus ergeben sich die Paare `(projekt, teilprojekt)` und `(teilprojekt, technologie)`. Da der Start des Paths den Typ `Projekt` hat, wird für das erste Paar vom Projekt das Attribut `teilprojekt` gesucht und der Typ, in diesem Fall Teilprojekt, zurückgegeben. Das zweite Paar startet dann mit dem Typen `Teilprojekt`, sucht das Attribut `technologie` und gibt dessen Typ zurück.
Der Prozess sucht für das erste Paar alle Teilprojekte, die in einem bestimmten Projekt

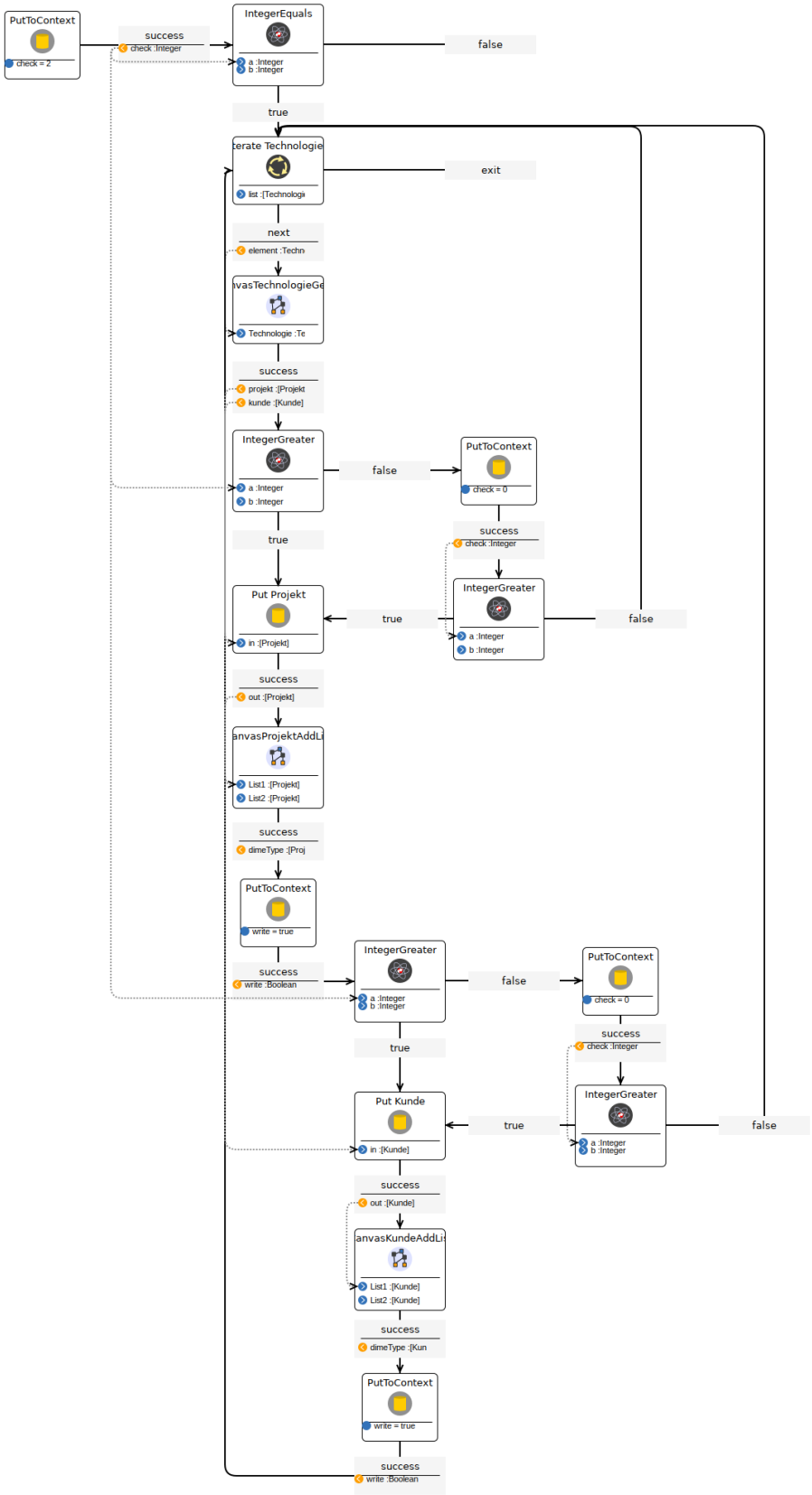


Abbildung 6.6: Aktualisieren der Auswahlliste anderer Contents bei Priorität zwei

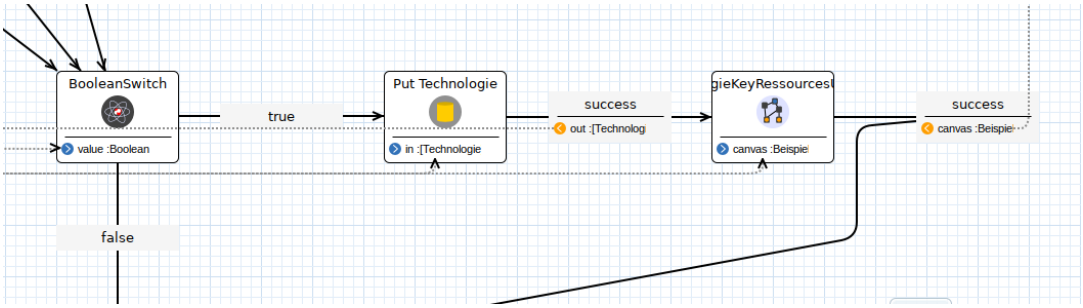


Abbildung 6.7: Schreiben der Verfügbarliste, wenn die Hilfsvariablen gesetzt sind

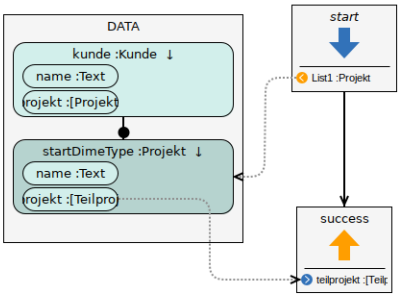


Abbildung 6.8: Suche nach dem Teilprojekt über ein Projekt

verwendet werden. Das ist in Abbildung 6.8 abgebildet. Ist auf einem Suchpfad ein abstrakter Typ angegeben, werden alle konkreten Typen des Objektes zurückgegeben. Wenn der Suchpfad danach mit einem Attribut weitergeführt wird, so müssen aus den zuvor gesammelten konkreten Typen die Attribute durchgesucht und auf den Namen verglichen werden. Gibt es ein solches Attribut, ist der Rückgabetyp gefunden. Gibt es kein Attribut mit dem Namen, muss das Attribut eine Vererbungsinstanz höher liegen. Dann wird dort weitergesucht.

Kapitel 7

Ergebnisse

In diesem abschließenden Kapitel sollen die Ergebnisse und Erkenntnisse der Projektgruppe zusammengefasst werden. Diese bestehen aus den Erfahrungen aus der Zusammenarbeit mit dem Unternehmen, sowie den technischen Resultaten, die erzielt wurden. Zum Ende wird ein Ausblick auf eventuelle zukünftige Themen und Fragestellungen gegeben.

7.1 Zusammenarbeit mit der Schulz Systemtechnik GmbH

Ein zentraler Teil der Projektgruppe war die Zusammenarbeit mit der Firma Schulz Systemtechnik GmbH, die in 3.1 schon kurz vorgestellt wurde. Der Kontakt fand hauptsächlich digital statt, jedoch gab es jeweils einen Besuch an den Standorten der Projektgruppe und in der Firmenzentrale der Schulzsystemtechnik GmbH. Der Besuch beim Unternehmen war für die Gruppe der Startpunkt, da hier Anforderungen und Rahmenbedingungen für das Projekt festgelegt wurden. Der Besuch eines Mitarbeiters des Unternehmens in Dortmund diente später dazu, gebündelt Feedback zum Zwischenstand einzuholen.

Ergebnisse des ersten Iterationsschritts

Vor dem Besuch der Firma Schulz hat sich die Projektgruppe mit verschiedenen Tools beschäftigt, mit denen eine Firmenontologie dargestellt werden kann. Diese Tools wurden zu Beginn der ersten Phase einigen Mitarbeitern der Firma vorgestellt. Aus diesem Grund sind einige Personen der Projektgruppe zu dem Firmenhauptsitz von Schulz in Visbek gefahren. Vor Ort wurde dann beschlossen, das Tool DIME (siehe Kap. 2.5) zu nutzen, um die Firmen-Ontologie in einem DIME Datenmodell darzustellen und diese dann mit DIME als Webapplikation zu modellieren. Im Anschluss wurde zusammen mit den Mitarbeitern eine erste Version eines Ontologie-Datenmodells(vgl. Abbildung 3.1) erstellt. Anhand dieses Datenmodells wurde in DIME ein Prototyp einer Webapplikation modelliert, die in Abbildung 3.11 zu sehen ist. Darüber hinaus wurde eine DSL entwickelt, mit der allgemein Ontologien beschrieben werden können, dies ist in Kapitel 4.1 beschrieben. Diese DSL ist

die Basis, um aus einer Firmenontologie eine Webapplikation zu generieren, die dann eine Firmenontologie komplett beinhaltet. Dies geschieht über eine Modell-zu-Modell Transformation, genauer beschrieben wird dies in Abschnitt 4.2. Bei der Modellierung der Website wurden Modellierungs-Pattern herausgestellt, die immer wieder auftauchten. Diese Pattern sind die Grundlage für die Transformation gewesen.

Ergebnisse der weiteren Iterationsschritte

Im Verlauf des Projektes gab es kontinuierlichen Kontakt zu der Firma Schulz. Nach einiger Bearbeitungszeit wurde ein Prototyp der Webapplikation fertig gestellt und konnte einem Mitarbeiter der Firma präsentiert werden. Dieser konnte den Prototypen der Website bewerten und eine erste Version der DSL betrachten. Die DSL stand zu diesem Zeitpunkt allerdings nur in einem frühen Entwicklungsstadium zur Verfügung. Es gab bei diesem Austausch hilfreiches Feedback für die Projektgruppe. So wurden einige Änderungen an der Umsetzung gewünscht, wie beispielsweise, dass Branchen auch in einer Baumstruktur verwaltet werden sollen, ähnlich zu den Technologien. Außerdem sollte es auch einige Änderungen am Datenmodell selbst geben, wie z. B. die Einführung eines Niederlassungsleiters und eines Geschäftsführers für Standorte. Die Änderungen wurden von der Projektgruppe in die Ontologie-DSL und in das daraus generierte DIME Produkt eingearbeitet.
Bei der Präsentation der ersten Version der DSL stellte sich schnell heraus, dass für die Nutzung der DSL ein gewisses Expertenwissen in Bezug auf die Hintergründe und die Bedienung des DSL-Editors vorrausgesetzt wird. Als Konsequenz daraus wurde versucht, die DSL für Endnutzer zu vereinfachen. Die darauffolgenden Feedbackrunden fanden alle nur noch Online statt. Sobald mithilfe der Transformation und der DSL die ersten generierten Prototypen erstellt werden konnten, wurden noch Feedbackwünsche und Änderungen an der Ontologie-DSL selbst vorgenommen. Auch zu der relativ neu integrierten Canvas-DSL wurde schon erstes Feedback eingeholt.

Erkenntnisse aus der Zusammenarbeit

An erster Stelle war für die Teilnehmer der Projektgruppe die Mitarbeit an so einem kooperativen Projekt zusammen mit einem großen Unternehmen eine neue Erfahrung. Ein zentraler Lernfaktor war hierbei auch die Strukturierung der Arbeit innerhalb des Teams. Außerdem wurde das allgemeine Wissen der Projektgruppenteilnehmer zu Themen wie DSLs, Modellierung und Firmenontologien erweitert.
Die Zusammenarbeit mit der Schulz Systemtechnik GmbH war stets ein zentraler Teil der Projektgruppe. Durch diese Kooperation wurde den Teilnehmern der Projektgruppe ein tieferer Einblick in ein größeres Unternehmen gewährt. Der relevanteste Aspekt dabei war der Aufbau und die Organisation der Firma.
Die Kooperation hat den Teilnehmern verdeutlicht, dass regelmäßige Kommunikation und

Austausch mit dem Partner oberste Priorität haben müssen, damit gemeinsam dasselbe Ziel verfolgt wird. So hat bspw. der anfängliche Besuch in der Unternehmenszentrale in Visbek überhaupt erst eine näherungsweise Modellierung der Firmenstruktur möglich gemacht.
Das Feedback von dem Partnerunternehmen hat außerdem Defizite in der Benutzerfreundlichkeit der entwickelten Elemente aufgedeckt. Somit hilft die Kommunikation mit einem Kontakt aus der Zielgruppe dabei, erstellte Lösungen endbenutzerfreundlich zu gestalten und zu beschreiben. Hierfür wurde ein User Guide erstellt. Ziel hierbei war es, dass Mitarbeiter der Firma Schulz oder einer beliebigen anderen Firma ihre eigene Firmenontologie in der DSL abbilden und eine Webapplikation damit generieren können, ohne sich vorher tiefergehendes Wissen über Programmierung oder die Modellierung in DIME aneignen zu müssen.

7.2 Überblick

Hier werden zusammenfassend die Ergebnisse präsentiert, die in den letzten zwei Semestern von der Projektgruppe erbracht wurden. Diese bestehen im Wesentlichen aus dem Modell der Firma Schulz Systemtechnik GmbH und der daraus generierten Webapplikation, der entwickelten DSL für Ontologien und die Erstellung von einem zugehörigen Canvas, und den Transformatoren, die aus konstruierten DSL-Modellen in mehreren Schritten nutzbare Software generieren.

Firmen-Ontologie als DSL-Datenmodell

Die Ontologie liegt als fertiges vollständiges Modell in der DSL vor (siehe Kapitel 4.1) und bietet die Grundlage für die generierte Webapplikation. Dieses Datenmodell entspricht nach dem Feedback des Mitarbeiters der Schulz Systemtechnik GmbH, dem aktuell gewünschten Stand. Mögliche Änderungen können jedoch direkt in der DSL vorgenommen werden und würden dann im Ablauf der Transformation umgesetzt werden.

Ontologie-DSL

Die DSL für die Firmenontologien ist vollständig vorhanden. Sie wurde auf die wesentlichen Punkte reduziert, um sie zu vereinfachen und nutzerfreundlicher zu machen. Mit dieser DSL soll es für jedes Unternehmen möglich sein, ihre Ontologie abzubilden und anhand dieser eine Webapplikation zu generieren. Darüber hinaus wurde die Ontologie der Firma vollständig in die DSL überführt(siehe 4.1) und anhand dieser die Transformation getestet. Die DSL erlaubt also die Beschreibung von einer große Menge an Firmenontologien, was die Transformation ermöglicht. Dieser Vorgang wird unterstützt durch Checks, die dem Nutzer Feedback geben, wenn die erstellte Ontologie noch nicht mit der DSL konform ist.

Canvas-DSL

Mit der Canvas-DSL kann ein eigener Canvas auf der Grundlage eines Ontologiemodelles erstellt werden. Es können eigene Canvas-Felder erstellt werden und für jedes dieser Felder eigene Abhängigkeiten definiert werden. Diese Felder werden mit Datentypen aus der Ontologie befüllt. Außerdem ermöglicht die Canvas-DSL Freitextbausteine, welche in der generierten Webapplikation als Notizfelder dargestellt werden. Die definierten Abhängigkeiten sorgen für einen besseren Überblick, genauer beschrieben wird dies in 6.1.

Transformatoren und Generierte Webapplikations

Die Erzeugung einer Webapplikation ist mithilfe der Modell-zu-Modell Transformation möglich. Es gibt eine gute Übersicht über die Funktionsweise der Transformatoren (siehe Kapitel 4.2). Es können sowohl Canvas- als auch Ontologiemodelle aus der DSL in DIME-Modelle transformiert werden. Die aus DIME generierten Webapplikations sind funktionsfähig und ermöglichen eine Umsetzung der in der DSL beschriebenen Vorgänge. Durch die Verlagerung des Transformationsprozesses auf mehrere Threads wurde das Laufzeit und Rechenleistungsproblem größtenteils behoben.

7.3 Zusammenfassung und Ausblick

Die Ergebnisse der Projektgruppe ermöglichen es nun mithilfe einer DSL Webapplikationen zu generieren, die eine spezifische Firmenontologie abbilden. Darüber hinaus ist es möglich, zu der Firmenontologie einen passenden Business Übersichtsplan in Form eines Canvas zu entwerfen. Beide finden sich in einer firmeneigenen Webapplikation wieder. Diese wird anhand der DSL und einer Modell-zu-Modell Transformation automatisiert generiert. Die Möglichkeiten, welche dieses Tool bietet, sind vielfältig und stellen einen Mehrwert für verschiedene Nutzergruppen dar. Jegliche Firma oder Organsiation kann mithilfe der DSL ihre eigene Ontologie abbilden.
Automatisiert wird daraus dann eine auf diese Beschreibung angepasste Webapplikation generiert, welche mit Daten befüllt werden kann und einen Übersicht über interne und externe Prozesse sowie Ressourcen bietet. Darüber hinaus kann mithilfe der Canvas DSL ein Canvas beschrieben werden. Dieser wird auch in die resultierende Webbapplikation integriert. Die Software bietet eine erste funktionale Vollständigkeit, die es ermöglicht, Webapplikationen für Unternehmen wie die Schulz Systemtechnik GmbH zu generieren. Der genaue Umfang der Funktionalität ist in den entsprechenden Kapiteln beschrieben. Zudem existiert ein Benutzerhandbuch, welches eine Anleitung zu den bestehenden Funktionalitäten beinhaltet.
Im Folgenden werden einige Betrachtungspunkte diskutiert, um die Grenzen und Mög-

lichkeiten des fertigen Projekts zu analysieren. Darüber werden denkbare neue Features evaluiert, die in der momentanen Version noch nicht implementiert sind.

Technische DSL Die Beschreibungssprache für die Ontologie und den Canvas sind beide einem objektrelationalen Modell nachempfunden. Es wurde bei der Entwicklung zwar darauf geachtet, dass die DSLs möglichst benutzerfreundlich und intuitiv zu bedienen sind, dies ist aber aufgrund der Nähe zu DIME-Datenmodellen nur begrenzt möglich gewesen. So dürfte die Benutzung der DSL Menschen mit IT-Hintergrund leichter fallen. Wem Begriffe wie abstrakte Typen, Baumstruktur oder Vererbung nicht geläufig sind, dem dürfte die Benutzung der DSL, auch mit dem Nutzerhandbuch, anfänglich recht schwer fallen.
Dies eröffnet die Frage, ob in der DSL weiter abstrahiert werden soll, um die Nutzerfreundlichkeit zu erhöhen oder ob diese dann zu ungenau und die eindeutige Transformation damit zu schwierig wird. In dieser Hinsicht muss ein Trade-Off zwischen Abstraktion und Informationsgehalt abgewogen werden. Dies wurde bei der Entwicklung der Canvas- bzw. Ontologie-DSL berücksichtigt. Das Ergebnis ist eine DSL, in der zwar noch einige IT-spezifische Begriffe eine Rolle spielen, sie aber dennoch eine gute Abstraktion zu der direkten Modellierung in DIME bietet. Auch wenn IT-Kenntnissen bei der Umsetzung der Ontologie und dem Canvas in der DSL sehr hilfreich sind, so ist dies kein disqualifizierendes Merkmal, da jedes größere Unternehmen heutzutage in gewissem Umfang IT-Kompetenzen haben muss.

Vereinfachung des gesamten Prozesses In der aktuellen Version des Projektes ist der Vorgang aufgrund der vielen Zwischenschritte von der DSL zu der generierten und ausführbaren Webapplikation noch recht lang. Dies ist zwar aus Sicht des Endnutzers nicht optimal, da sich aber sowohl das Projekt als auch DIME noch in der Entwicklung befinden, ist eine Automatiesierung des gesamten Prozesses noch nicht vorgesehen. Dies wäre aber in naher Zukunft eine sichere Optimierungsmöglichkeit, da der Endbenutzer sich dann lediglich mit der DSL auseinandersetzen muss und nicht mit dem Generierungsprozess und dem Ausführen der Webapplikation. Dies sind Schritte, die Automatisierungspotenzial bieten.
Ein weiterer Punkt ist die Laufzeit und die Rechenleistung, die es braucht, um eine DSL nach DIME zu transformieren. Dieser Ressourcenbedarf wurde zwar schon etwas optimiert, sodass die komplette Ontologie von Schulz in aktzeptabler Zeit generiert und als Webanwendung zu Verfügung gestellt werden konnte. Werden die Ontologien aber noch größer als diese, könnte es zu Laufzeiten außerhalb eines praktikablen Rahmens kommen. Es sollte auf jeden Fall davon abgesehen werden, diesen Vorgang auf einem schwachen Rechner laufen zu lassen. Daher bleibt die Optimierung des Transformationsprozesses ein wichtiger Punkt, an dem in der Zukunft angeknüpft werden sollte.

Feedback Zum aktuellen Zeitpunkt wurde bisher lediglich durch einen Vertreter der Schulz Systemtechnik GmbH und durch die Betreuer der Projektgruppe Feedback eingeholt. Es ist ratsam, von Seiten der Schulz Systemtechnik GmbH weiteres Feedback zu den beiden entwickelten DSLs zu bekommen, da vor allem die Canvas-DSL noch jung ist und erst wenig Feedback erhalten hat. Dies muss, wenn das Projekt fortgesetzt werden sollte, mit hoher Priorität erweitert werden. Jeder Feedbackzyklus ermöglicht neue Anpassungsmöglichkeiten, die später auch zu einer größeren Kundenzufriedenheit führen.
Da die resultierende Webapplikation in mehreren Iterationsschritten zusammen mit der Schulz Systemtechnik GmbH erarbeitet wurde, ist es möglich, dass die DSL sehr spezifisch auf die Bedürfnisse dieser Firma angepasst ist. Es wäre gut, diese auch von anderen Parteien testen zu lassen. Hierbei sollte beachtet werden, dass die verschiedenen Tester aus unterschiedlich IT-affinen Bereichen stammen, um die Nutzerfreundlichkeit der DSL zu evaluieren. Unter Umständen muss die DSL auf neu auftauchende Probleme und Fragestellungen in anderen Firmen erweitert werden. Denkbar sind hier beispielsweise neue Datentypen oder eine Möglichkeit, die Darstellungsweise der resultierenden Webapplikation genauer zu beschreiben. Dadurch, dass auch andere Ontologien als nur die von Schulz Systemtechnik GmbH umgesetzt würden, könnten möglicherweise auch neue Fehler entdeckt werden, die beim Testen bisher noch nicht aufgetreten sind.

Erweiterungen Applikationen können generell meist weiter entwickelt und um neue Features ergänzt werden. Solche Erweiterungen werden oft nach einer Feedbackrunde mit einem potentiellen Kunden deutlich. Einige denkbare Features wurden am Ende des letzten Abschnittes schon genannt. Eine weitere mögliche Ergänzung ist die Einführung eines Accountsystems mit verschiedenen Rollen. Dieses wurde von der Projektgruppe anfänglich geplant, aufgrund des zeitlichen Rahmens jedoch verworfen. Unterschiedliche Nutzer der Webapplikation könnten verschiedene Zugriffsrechte in dieser haben. Mit einem solchen System könnten Projektleiter beispielsweise ihre Mitarbeiter organisieren und Aufgaben verteilen. Momentan kann jeder Nutzer alles in der Webapplikation bearbeiten und löschen; dies ist für kleine Firmen zwar kein großes Problem, aber sobald eine große Firma ihre Ontologie als Webapplikation verwalten will, ist ein Accountsystem unabdingbar.

Anhang A

Dokumentation

In diesem Anhang wurde auf den folgenden Seiten die aktuelle Version des Nutzerleitfadens angefügt. Dieser soll für neue Nutzer als Dokumentation eine Anleitung für die Benutzung der entwickelten Tools bieten.

User Guide

PG 631: Responsive and Knowledge-Driven Business Modeling
Lehrstuhl XIV - Software Engineering
Lehrstuhl V - Programmiersysteme
TU Dortmund

30. September 2020

Inhaltsverzeichnis

1 Domänenspezifische Sprache **3**
- 1.1 Typen . . . 3
  - 1.1.1 Enum . . . 3
  - 1.1.2 ConcreteType . . . 3
  - 1.1.3 AbstractType . . . 3
  - 1.1.4 ConcreteTreeType . . . 4
- 1.2 Attribute . . . 4
- 1.3 Typparameter . . . 4
  - 1.3.1 AbstractType/ConcreteType . . . 4
- 1.4 Attributparameter . . . 5
  - 1.4.1 PrimitiveAttribute . . . 5
  - 1.4.2 ComplexAttribute . . . 5
  - 1.4.3 DerivedAttribute . . . 6
- 1.5 Kanten . . . 6
  - 1.5.1 Association . . . 6
  - 1.5.2 Bidirectional Association . . . 6
  - 1.5.3 Aggregate Association . . . 7
  - 1.5.4 Inheritance . . . 7
- 1.6 Checks . . . 7
- 1.7 Beispiel . . . 7

2 WebApp **9**
- 2.1 Startseite . . . 9
- 2.2 Übersicht . . . 10
  - 2.2.1 Übersicht für konkrete Typen . . . 10
  - 2.2.2 Übersicht für vererbende Typen . . . 14
- 2.3 Bearbeiten . . . 17
- 2.4 Detailansicht . . . 22
  - 2.4.1 Primitive Attribute/Enums . . . 22
  - 2.4.2 AbstractTypes . . . 22
  - 2.4.3 ConcreteTypes . . . 22
- 2.5 Bäume . . . 23
  - 2.5.1 Was ist eine Baumstruktur und warum wird sie genutzt? . . . 23
  - 2.5.2 Beispiele in der Webapp . . . 23
  - 2.5.3 Die Übersicht . . . 24
  - 2.5.4 Die Anlegen-/Bearbeiten-Übersicht . . . 24
  - 2.5.5 Das Fenster innerhalb der Technologie . . . 25
  - 2.5.6 Die Navigationsleise . . . 26
  - 2.5.7 Zuweisung von Baumtypen . . . 26

3 Canvas **27**
- 3.1 Domänspezifische Sprache . . . 28
- 3.2 WebApp . . . 28
  - 3.2.1 Übersicht Canvases . . . 28
  - 3.2.2 Beispielcanvas . . . 30
  - 3.2.3 Symbole . . . 30

3.2.4 Die Beeinflussungen im Canvas 34

4 Canvas DSL **35**
4.1 Canvas anlegen und Übersicht 36
4.1.1 Palette 36
4.1.2 Import von Ontologytypen 38
4.1.3 Freitexte und Notizen 38
4.1.4 Cinco Properties 38
4.2 Path 40
4.2.1 Path erstellen 40
4.2.2 Path beschreiben 40
4.3 Validierung eines Canvas 41

1 Domänenspezifische Sprache

Bevor eine WebApp generiert werden kann, muss eine Ontologie erstellt werden. Die Ontologie besteht aus Typen, die Objekte der Realität repräsentieren, und Beziehungen zwischen diesen Typen. Die Struktur dieser Daten wird durch die domänenspezifische Sprache (DSL) festgelegt.

1.1 Typen

Die Typen der DSL sind AbstractType, ConcreteType, ConcreteTreeType und Enum. Die Bedeutung der einzelnen Typen ist folgend aufgelistet.

1.1.1 Enum

Enums (kurz für Enumerations) werden ähnlich der gängigen Verwendung in der Informatik verwendet, um Aufzählungen anzulegen. Diese Aufzählungen sind statisch und können in der WebApp nicht mehr verändert werden.
Beispiel: Eine Auflistung von Kontinenten oder Ländern.

1.1.2 ConcreteType

Dieser Typ repräsentiert die klassichen Objekte, mit denen In der Webapp gearbeitet wird. ConcreteTypes sind die einfachste Art, Elemente der echten Welt abzubilden. ConcreteTypes (sowie Abstract- und ConcreteTreeTypes) können Attribute beinhalten. ConcreteTypes können alle Eigenschaften und Attribute von anderen Concrete- oder AbstractTypes erben.
Beispiel: ConcreteType Standort mit Attributen Name, Adresse und einer Liste von Mitarbeitern.

1.1.3 AbstractType

Ein AbstractType ist ein Typ, der nicht konkret existiert. Der Typ kann als Blaupause verstanden werden, der das Grundgerüst für andere ConcreteTypes legt. Er eignet sich als Strukturelement innerhalb einer Klassenhierarchie. Wie ConcreteTypes kann er auch Attribute und Verweise auf andere Typen enthalten. Ein AbstractType kann allerdings

innerhalb der Anwendung nie als Objekt angelegt werden. Es muss mindestens einen ConcreteType geben, der den AbstractType beerbt.
Beispiel: AbstractType Mitarbeiter mit Attributen Name, Standort. Zusätzlich zwei ConcreteTypes Angestellter und Abteilungsleiter, die von Mitarbeiter erben und somit automatisch Namen und Standort haben. Der Abteilungsleiter Typ wird um eine Liste von Angestellten erweitert, die in seiner Abteilung arbeiten.

1.1.4 ConcreteTreeType

Dieser Typ ist eine besondere Form des ConcreteTypes. Der ConcreteTreeType enthält Typverweise auf seinen eigenen Typen und stellt somit eine rekursive Datenstruktur dar. Dem Namen nach, bildet der ConcreteTreeType eine Baumdatenstruktur. Bäume erhalten immer automatisch einen "parent"(also einen Verweis auf das Element eine Hierarchie höher) und eine Liste von "children"(der Elemente in der direkten Hierarchie nach unten) , die den Tree selbstreferenzieren.
Beispiel: ConcreteTreeType Branche mit Attribut Name. Jede Branche hat also einen Namen, und kann gleichzeitig weitere Unterbranchen besitzen. Diese Unterbranchen sind wiederum vom selben ConcreteTreeType Branche, die wiederum eigene Unterbranchen haben können und zusätzlich wissen, welche ihre Elternbranche ist.

1.2 Attribute

Attribute repräsentieren die Daten, die ein Typ beinhalten und darstellen soll. Attribute können primitiv (Text, Zahlen, Boolean,..) oder komplex (andere Typen) oder Enums sein. Enums verfügen über eigene EnumLiteral Attribute. Zusätzlich gibt es ein DerivedAttribute, das genutzt wird, um indirekte (transitive) Verbindungen zu anderen Typen darzustellen. Der Pfad wird über das Attribut 'pathToReference' spezifiziert (siehe Abschnitt 1.4.3).

1.3 Typparameter

Die verschiedenen Typen enthalten Parameter, die im folgenden erklärt werden. Die Typparameter dienen zur Feinsteuerung der oben genannten Typen.

1.3.1 AbstractType/ConcreteType

Abstrakte und konkrete Typen enthalten folgende Parameter:

- name: Name des Typs (z.B. Mitarbeiter, Projekt, Kunde)
- defaultSort: Auswahl eines PrimitiveAttribute des Typs, über den eine Tabelle des Typs standardmäßig sortiert wird (z.B. Nachname bei Mitarbeiter -¿ die Mitarbeiter Tabelle wird standardmäßig über die Nachnamen sortiert)
- identifier: Eingabe eines Strings, der eine eindeutige Identifizierung des Objektes in dem Typen ermöglicht. Zur Identifizierung werden die Attribute des Typs genutzt. Zum angeben des Strings soll ein Mustache-Ausdruck genutzt werden (z.B.

"{{nachname}} {{vorname}} ({{standort.name}})"liefert "Mustermann Max (Musterstadt)"für einen Mitarbeiter). Standardmäßig ist bereits ein Default-Identifier gesetzt.

- isRoot: Markiert ein Typ als Rootelement. Rootelemente werden auf der Startseite angezeigt. Sie bilden die Einstiegspunkte in der WebApp.
- rootIndex: Gibt die Sortierung unter den Rootelementen auf der Startseite an.
- pluralName: Wird als Titel aus der Startseite angezeigt. Ein ConcreteType (Root) person mit pluralName Mitarbeiter wird in der WebApp als Mitarbeiter angezeigt.
- hasDetailView: Gibt an, ob es in der entsprechenden Verwalten-GUI einen Ansichts-Knopf geben soll. Die Ansicht zeigt alle Informationen über ein Objekt an, die für die Detailview ausgewählt wurden (siehe 'completeDetailViewInformation' in Abschnitt 1.4.2).
- isUndeletable: Gibt an, ob ein Objekt unlöschbar ist.

1.4 Attributparameter

Die Attributparamter werden in diesem Abschnitt erklärt.

1.4.1 PrimitiveAttribute

Primitive Attribute enthalten folgende Parameter:

- dataType: Primitiver Datentyp (Text, Integer, Boolean, usw.).
- name: Name des Attributs.
- isList: Gibt an, ob das Attribut eine Liste ist. (Das Attribut ist entweder genau ein Element, oder eine Liste von Elementen gleichen Datentyps)
- regularExpression: Es kann ein regulärer Ausdruck angegeben werden, der überprüft, ob die Eingabe für dieses Attribut valide ist (z.B. E-Mail Adresse).
- isRequired: Wenn ein neues Objekt erstellt wird, muss dieses Attribut einen Wert erhalten. Es darf nicht leer bleiben. isUnique: Attribut muss eindeutig sein. (Es darf für dieses Attribut keine zwei Elemente mit selben Inhalt geben)

1.4.2 ComplexAttribute

Komplexe Attribute enthalten zusätzlich folgende Parameter:

- view: Gibt die Art der Darstellung von Listenobjekten an (Liste oder Tabelle).
- completeDetailViewInformation: Bei Aktivierung, wird dieses Attribut in der Detailansicht mit allen Attributen des Typs aufgelistet, andernfalls wird nur der Identifier angezeigt.
- showInTable: Attribut wird in der Übersicht angezeigt.

- optIdentifier: Überschreibt den Identifier (siehe Identifier in Abschnitt 1.3.1).
- createEditType: Für primäre und komplexe Attribute gibt dieser Parameter an, wie ein Attribut beim Erstellen oder Bearbeiten dargestellt wird:
  - Creatable: Ein Attribut wird innerhalb des Typs angelegt.
  - Selectable: Ein Attribut wird aus einer Liste von vorhandenen Elementen ausgewählt.
  - Viewable: Das Attribut wird beim Erstellen und Bearbeiten nicht editiert. Es wird nur angezeigt.
  - Hidden: Das Attribut ist versteckt. Es wird weder angezeigt noch ist es editierbar.
- TableInsteadOfCheckbox: Ist die Auswahl bei createEditType auf Selectable gestellt, kann mit dieser Einstellung ausgesucht werden, ob die Auswahl für eine Liste des ComplexAttribute über eine Tabelle oder eine CheckBox Auswahl geschehen soll. (Keine Wirkung bei nicht Listen, Creatable, Viewable oder Hidden)

1.4.3 DerivedAttribute

Derived Attribute enthalten zusätzlich folgende Parameter:
PathToReference: Ein Mustache-Ausdruck, der den Pfad zum referenzierten Attribut anhand der Attributsnamen nachvollzieht (z.B. projekt.nummer In dem Beispiel wird über das Attribut projekt, das Attribut nummer im Objekt Projekt referenziert).
dataType: Der Datentyp, der hier derived wird. (Muss deckungsgleich mit dem Datentyp des Endpunktes des ReferenzPfades sein)

1.5 Kanten

In diesem Abschnitt werden die unterschiedlichen KantenTypen erklärt, die zwischen den Typen bestehen können. Kanten werden gezogen, indem über den Typen gehovert wird und auf den Pfeil oben rechts neben dem Typnamen geklickt wird. Die Kante kann dann zu einem beliebigen Typen gezogen werden.

1.5.1 Association

Sie beschreibt die einfachste Beziehung zwischen zwei Typen. Der Typ (Ursprung), von dem aus die Kante gezogen wird, referenziert den Typ (Ziel), auf den gezeigt wird. Im Ursprung wird ein ComplexAttribute mit Datentyp des Ziels erstellt.

1.5.2 Bidirectional Association

Bei dieser Assoziation referenzieren die Typen sich gegenseitig. Sowohl im Ursprung als auch im Ziel wird ein ComplexAttribute mit dem Datentyp des jeweils anderen Typ erstellt.

1.5.3 Aggregate Association

Der Ursprung aggregiert das Ziel. Der Ursprung erhält wie in der Association ein ComplexAttribute des Ziels. Die Besonderheit der Aggregate Association besteht darin, dass in diesem ComplexAttribute der createEditType auf Createable gestellt sein. Zusätzlich werden die Elemente des Ziels gelöscht, wenn das Ursprungselement gelöscht wird.
Wenn also ein Standort Typ, der eine Adresse aggregiert, gelöscht wird, wird die entsprechende Adresse ebenfalls gelöscht. Aggregierte Elemente existieren nur solange das Ursprungselement existiert.

1.5.4 Inheritance

Der Typ, von dem aus die Kante gezogen wird, erbt von den Typen, auf den gezeigt wird. Der Ursprung erhält alle Attribute des Ziels.

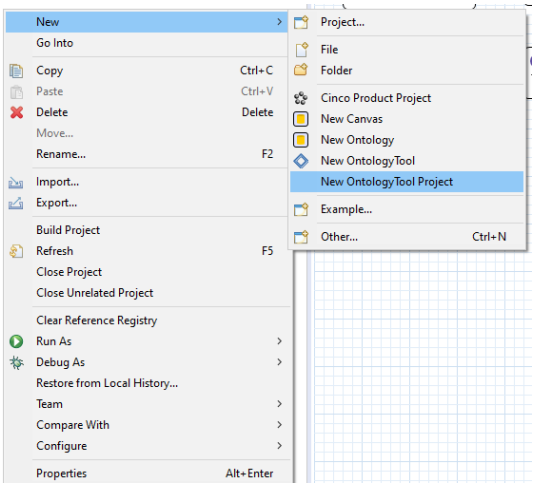
1.6 Checks

Die DSL ist mit verschiedenen Checks ausgestattet, die beim Speichern der Ontologie ausgeführt werden. Die Checks weisen auf Fehler oder unzulässige Einstellungen innerhalb der Ontologie hin. Die Ontologie darf keine Fehlermeldungen mehr beinhalten, bevor sie weiter benutzt wird, da sonst die richtige Funktionalität nicht gewährleistet ist.

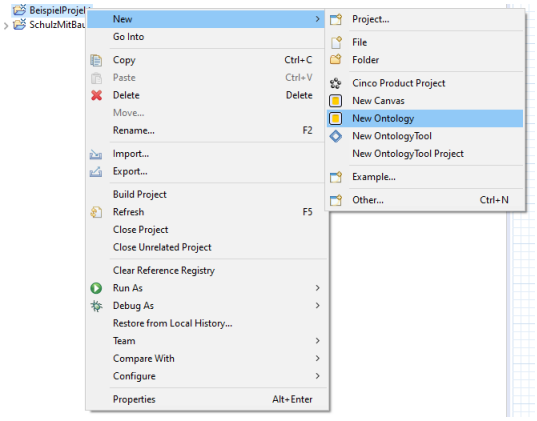
1.7 Beispiel

Es soll als Beispiel eine kleine Ontologie mit Standort und Angestellten erstellt werden. Als erstes muss ein OntologyToolProject angelegt werden. Dazu wird im geöffneten Cinco Product mit Rechtsklick in der Projektspalte auf New, New OntologyTool Project geklickt. Das Projekt benötigt dann einen Namen. Anschließend wird per Rechtsklick auf das Projekt mit New, New Ontology eine neue Ontologie angelegt. Mit Doppelklick auf die Ontologie öffnet sich die Bearbeitungsansicht. Als erstes wird ein ConcreteType per Drag and Drop in die Bearbeitung gezogen. Dem Type wird über die Type Properties am unteren Bildschirm der Name Standort gegeben. Eventuell muss der Standort Type noch mal angeklickt werden, damit sich die Properties richtig öffnen. Anschließend wird ein Primitive Attribute per Drag and Drog auf den Standort gezogen. Mit einem Klick auf das Attribut, kann wie beim Standort bereits dem Attribut ein Namen gegeben werden. Anschließend wird beim Standort in den type Properties der Identifier auf {{name}} gesetzt, damit der Standort immer mit seinem Namen angezeigt wird. Als Default Sort wird ebenfalls der Name angegeben. In den Root Properties wird dann is Root angeklick, der Root Index 1 vergeben und der Plural Name auf Standorte gesetzt.

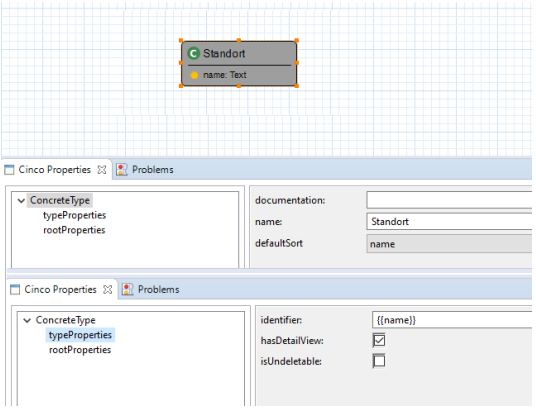
Als zweites wird ein ConcreteType Adresse mit 5 Attributen angelegt (Straße, Hausnummer, PLZ, Ort, Zusatz). Mit Klick auf Standort, Klick auf den Pfeil im Pop up und dann per Drag and Drop auf die Adresse, kann eine Verbindung zwischen Standort und Adresse erstellt werden. Als Art der Verbindung wird hier Aggregate Association gewählt. Damit wird die Adresse immer im Standort angelegt, wenn sie gebraucht wird. Um die Standorte jetzt mit Angestellten zu füllen, wir ein Abstract Type Angesteller mit Name, Vorname angelegt, der zusätzlich auch ein Root Type mit Index 2 und Plural Name Angestellte sein soll. Der Angestellte erhält ebenfalls eine Aggregate Association auf den Type Adresse. Zusätzlich zwei ConcreteTypes Mitarbeiter und Abteilungsleiter



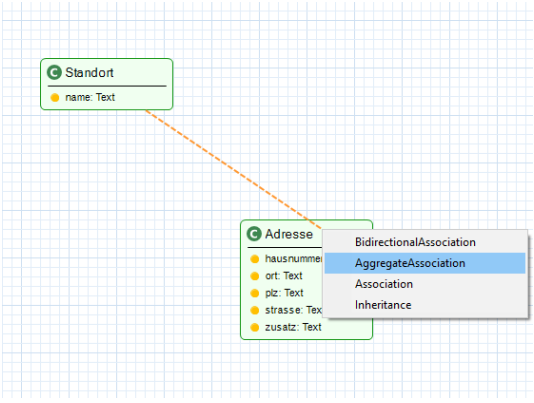
(a) Ontology Tool Project Anlegen



(b) Ontology Anlegen



(a) Standort Anlegen



(b) Adresse Anlegen

angestellt. Beide ConcreteTypes erhalten die Verbindung Inheritance ausgehend von Mitarbeiter/Abteilungsleiter auf den Abstract Type Angestellter. Als letztes werden noch einige Bidirectional Associations gezogen. Zwischen Standort und Angestellte (Dabei muss in Standort das ComplexAttribute über die complex properties auf is List gestellt und bei createEditType auf Viewable gestellt werden, in Angestellte muss der createEditType auf Selectable gestellt werden. Das bedeutet, dass ein Standort eine Liste von Angestellten erhält, die aus den Mitarbeitern besteht, bei denen der entsprechende Standort ausgewählt wurde. Jeder Angestellter ist nur einem Standort zugewiesen, da das komplexe Attribut Standort in Angestellter keine Liste ist.) und zwischen Abteilungsleiter und Mitarbeiter (hier ebenfalls in Abteilungsleiter die Mitarbeiter zu einer Liste machen, allerdings hier den createEditType auf Selectable, in Mitarbeiter den Abteilungsleiter Viewable machen. Somit kann einem Abteilungsleiter die entsprechenden Mitarbeiter zugewiesen werden.) Falls alles richtig gemacht worden ist, sollte die Ontologie so aussehen wie in dem Bei-

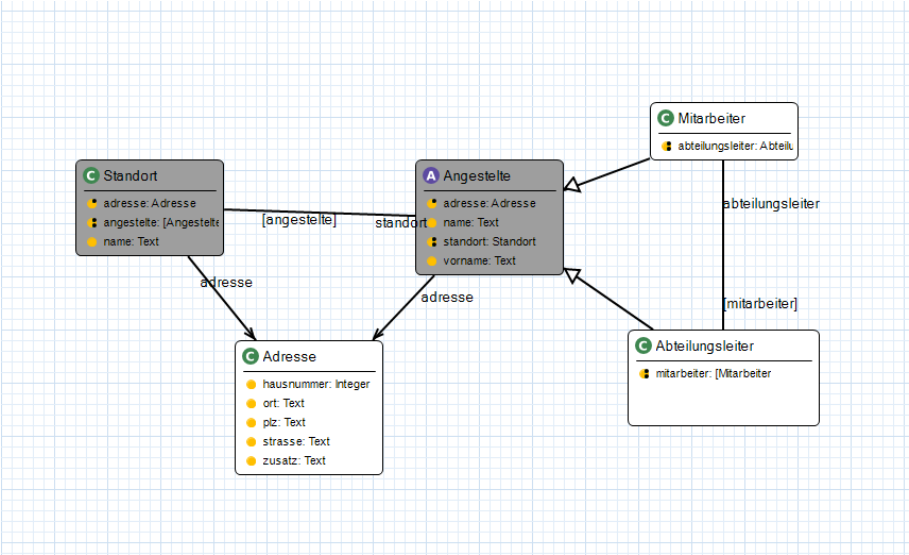


Abbildung 3: Fertige Ontologie

spiel, und nach dem Speichern kein Fehler angezeigt werden. Ansonsten kann über die Fehlermeldungen herausgefunden werden, was genau nicht richtig gemacht worden ist.

2 WebApp

In diesem Abschnitt wird die Bedienung der WebApp erläutert. Die zugehörigen DSL-Optionen lassen sich in Kapitel 1 nachlesen.

2.1 Startseite

Abbildung 4 zeigt die Ontologie mit der die nachfolgende Startseite erklärt wird. Abbildung 5 zeigt die zugehörige Startseite der Webapp. In der Ontologie wurden die beiden Typen *Person* und *Projekt* in der Checkbox *isRoot* unter dem Reiter *rootProperties* markiert. Dies ist sichtbar durch den grauen Farbton. Der Typ *Standort* ist kein Root-Element und somit weiß unterlegt. Die Startseite enthält eine Navigationsleiste, die die Root-Elemente der Anwendung enthält. In diesem Fall zeigt die Leiste die beiden Typen

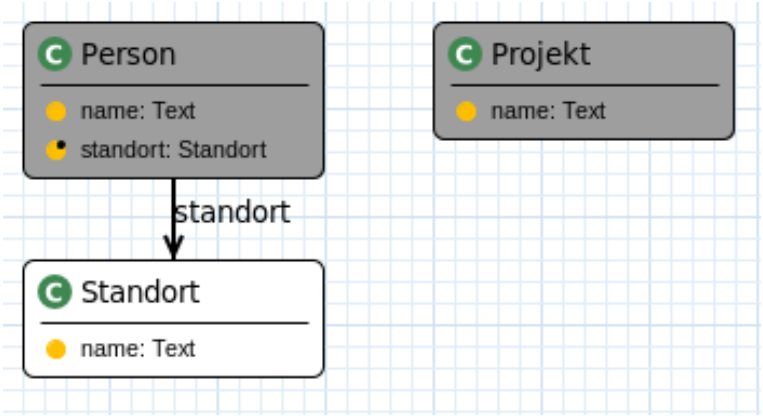


Abbildung 4: Ontologie mit der Roots erstellt wurden

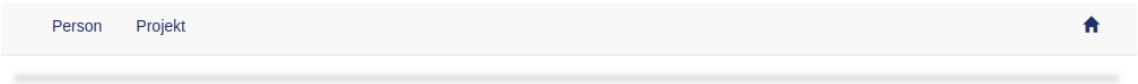


Abbildung 5: Navigationsleiste

Person und *Projekt*. Per Klick gelangt der User zu den jeweiligen Übersichten. Am oberen rechten Ende befindet sich ein Home-Button, über den man zur Startseite zurückkehrt. Die Navigationsleiste wird auf jeder Seite angezeigt.
Unter der Navigationsleiste kann ein Bild eingebunden werden. In Abbildung 5 ist kein Bild eingebunden, daher zeigt die Startseite nur einen grauen Rahmen unter der Navigationsleiste. Abbildung 6 zeigt eine Startseite, bei der ein Bild eingebunden wurde. Das Bild befindet sich unter der Navigationsleiste.

2.2 Übersicht

In diesem Abschnitt werden die Funktionalitäten der Übersichts-GUI beschrieben.

2.2.1 Übersicht für konkrete Typen

Die Übersicht enthält die Elemente des Typen in tabellarischer Form. Abbildung 7 zeigt eine Ontologie mit den Typen *Person*, *Fähigkeit* und *Hobby*. Abbildung 8 zeigt die zugehörige Übersichtsseite von *Person*. Die Tabelle zeigt alle primitiven Attribute des Typen und die komplexen Attribute, die in der Ontologie mit *showInTable* markiert wurden. *Hobby* wurde in der Ontologie im Typen *Person* unter viewOptions nicht mit *showInTable* markiert. Die Tabelle beinhaltet das Attribut somit nicht. Komplexe Attribute werden standardmäßig mit ihrem Identifier in der Tabelle angezeigt. Bei Fähigkeit ist der Identifier *id*. In Person wurde für Fähigkeit unter dem Reiter *viewOptions* der *optIdentifier* *bezeichnung* gesetzt. Deswegen wird in Abbildung 8 die Bezeichnung der Fähigkeit angezeigt, da bei Setzung eines optIdentifiers dieser vor dem normalen Identifier Vorrang hat. Die Inhalte von Fähigkeit werden in der Tabelle als Liste angezeigt, da in der Ontologie Fähigkeit als Liste gesetzt wurde.
In Abbildung 8 ist defaultmäßig nach *Name* sortiert, da in der Ontologie bei *Person* *defaultSort* auf *name* gesetzt wurde. Es ist möglich bei Verwendung der Webapp nach

Abbildung 6: Startseite

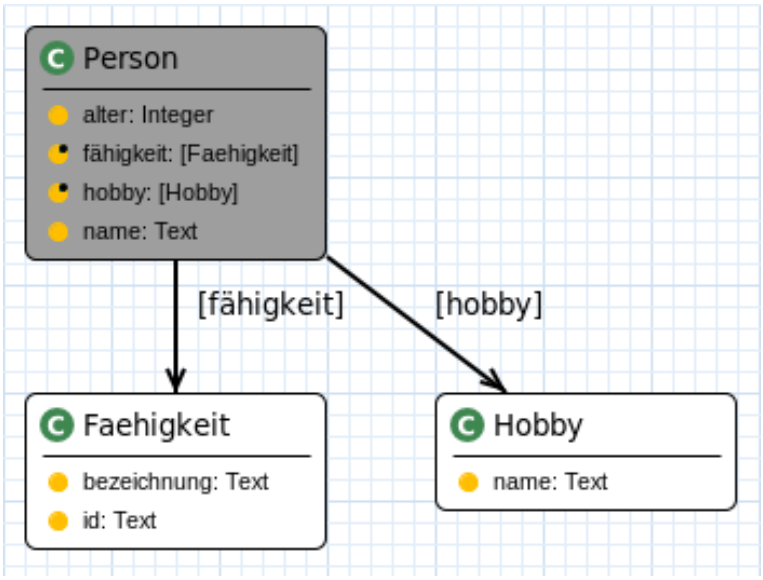


Abbildung 7: Ontologie mit der die Übersicht erstellt wurde

anderen primitiven Attributen zu sortieren. Dies geschieht durch Klick auf den jeweiligen Attributnamen, nach dem sortiert werden soll. Abbildung 9 zeigt die Übersicht, nachdem auf *Alter* geklickt wurde, um nach dem Alter zu sortieren. Bei mehrmaligen Klick wechselt die Sortierung zwischen aufsteigend und absteigend.
Für die primitiven Attribute gibt es Suchfelder. Für Strings wird nach enthaltenen Teilwörtern gesucht. In Abbildung 10 wurde beispielsweise nach *Erika* gesucht. Bei Zahlen kann der User einen Suchbereich (*from, to*) eingeben. Abbildung 11 zeigt eine Suche nach Perso-

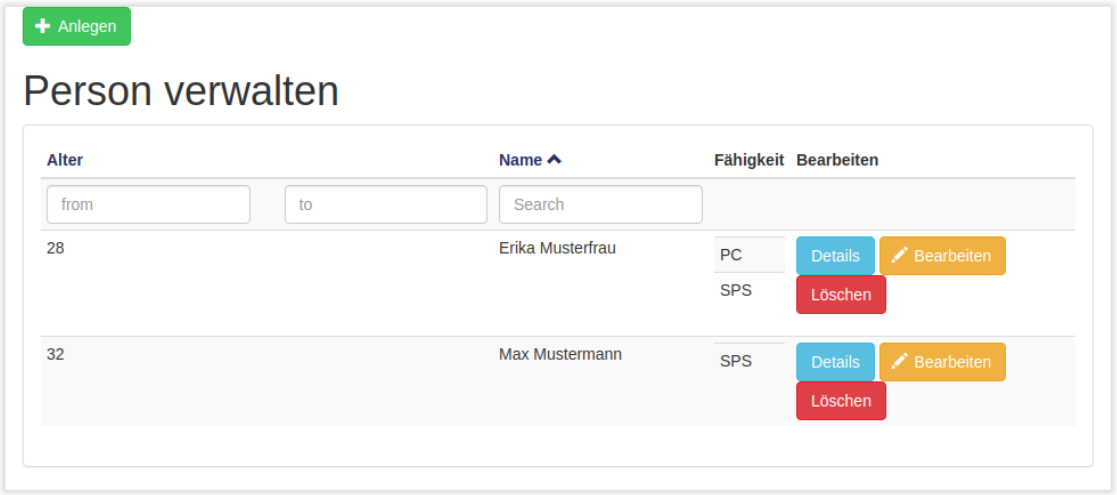


Abbildung 8: Übersicht

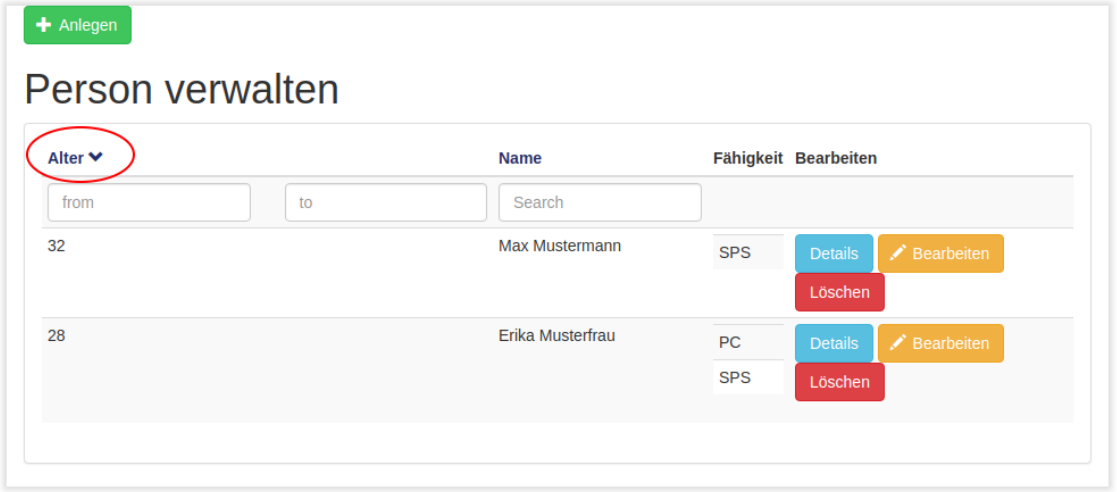


Abbildung 9: Sortierung nach 'Alter'

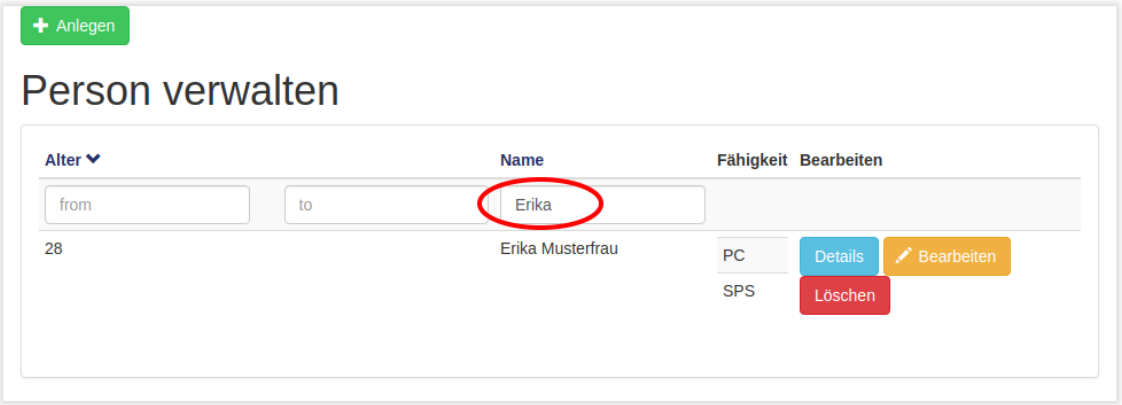


Abbildung 10: Suche nach 'Erika'

nen, deren Alter mindestens 30 ist. Abbildung 12 zeigt eine Suche nach Personen, deren Alter höchstens 30 ist. Die Übersicht aus Abbildung 8 zeigt außerdem die Buttons *Bear-*

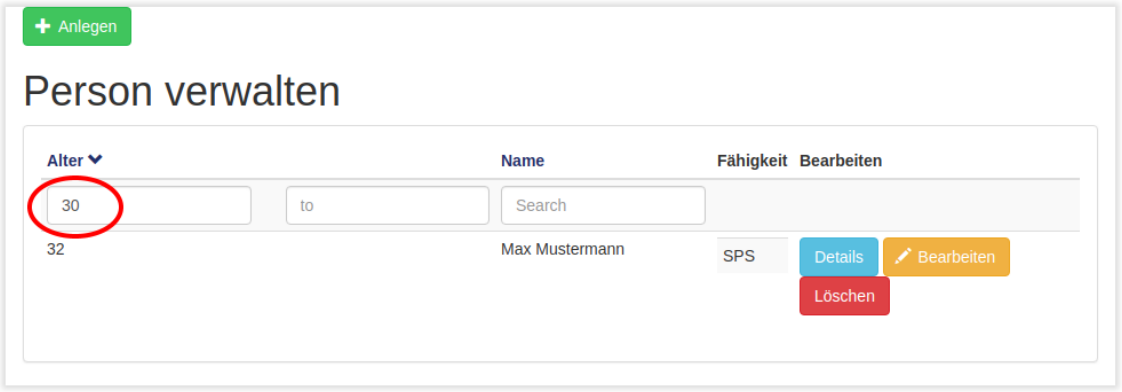


Abbildung 11: Suche Personen, die mindestens 30 sind

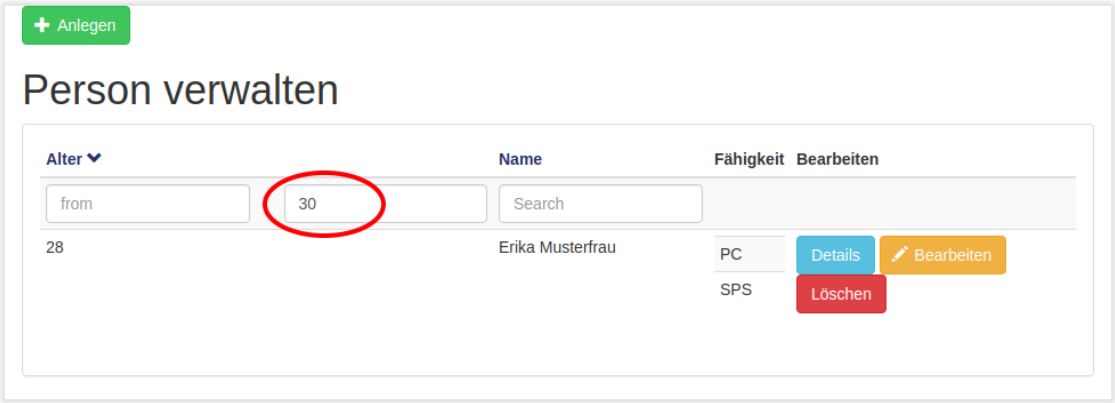


Abbildung 12: Suche Personen, die höchstens 30 sind

beiten, *Löschen* und *Details*. Der erste der drei Buttons führt zu einer Bearbeiten-GUI, in der die Werte gesetzt werden können. Der Lösch-Button ist nur vorhanden sofern die Option *isUndeletable* deaktiviert ist. Über den Button *Details*, wird eine Detailansicht geöffnet, in der detaillierte Informationen des Objekts angezeigt werden. Die Detailansicht ist nur verfügbar, sofern die Option *hasDetailView* aktiviert wurde. Abbildung 13

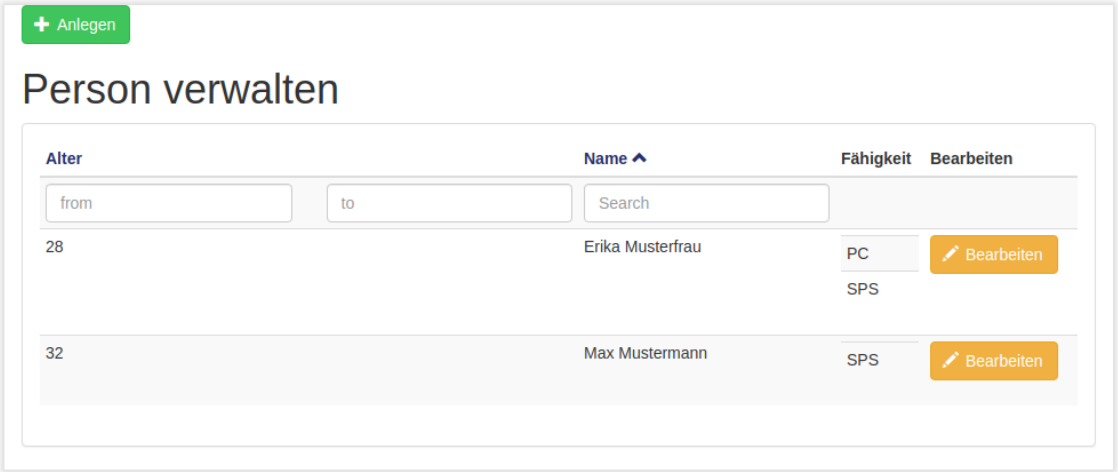


Abbildung 13: Übersicht ohne die Buttons *Löschen* und *Details*

zeigt eine Übersicht, in der der Typ *Person* als *isUndeletable* und nicht als *hasDetailView* unter *typeProperties* markiert ist. Über der Tabelle befindet sich ein Button *Anlegen*, der ebenfalls die Bearbeiten-GUI öffnet, sodass ein neues Element erstellt werden kann. Ab-

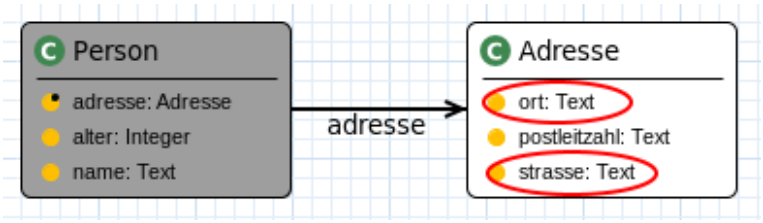


Abbildung 14: Adresse hat zwei Identifier

bildung 14 zeigt eine Beispielontologie, in der Adresse über zwei Identifier (*postleitzahl* und *strasse*) verfügt. Abbildung 15 zeigt die daraus entstehende Übersicht. Unter Adresse werden sowohl die Postleitzahl als auch die Straße angezeigt. In der Ontologie aus Abbil-

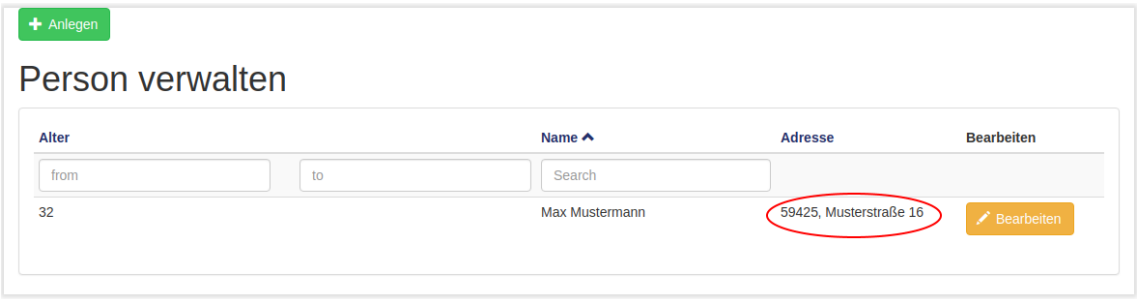


Abbildung 15: Übersicht mit doppeltem Identifier

dung 16 befindet sich im Typen Person das derived Attribut *ort*, das auf den Typen Ort referenziert. Die zugehörige generierte Übersicht wird in Abbildung 17 gezeigt. Obwohl

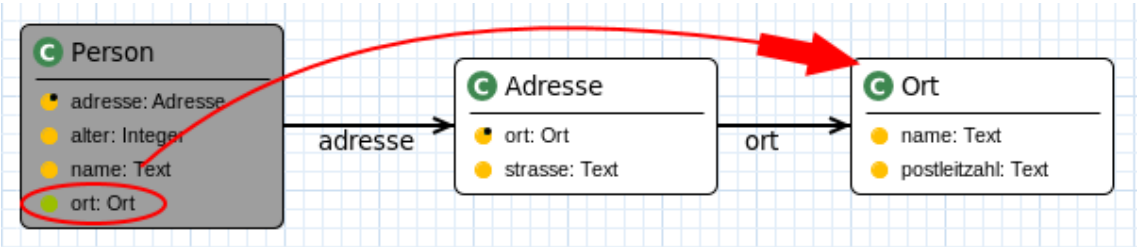


Abbildung 16: Derived Attribut

keine direkte Verbindung in der Ontologie besteht, enthält die Übersicht ebenfalls das derived Attribut *ort*.

2.2.2 Übersicht für vererbende Typen

Diese Übersicht enthält die einzelnen Übersicht-GUIs all ihrer erbenden Subtypen und ihre eigene Übersicht, sofern der Typ konkret ist (abstrakte Typen haben keine eigene Tabellenansicht). Die einzelnen Übersicht-GUIs entsprechen der Beschreibung aus Abschnitt 2.2.1. Sie befinden sich jeweils in einem Panel, das einklappbar ist. Dies geschieht durch Klick auf den Tabellentitel. Abbildung 18 zeigt eine Ontologie, in der die zwei

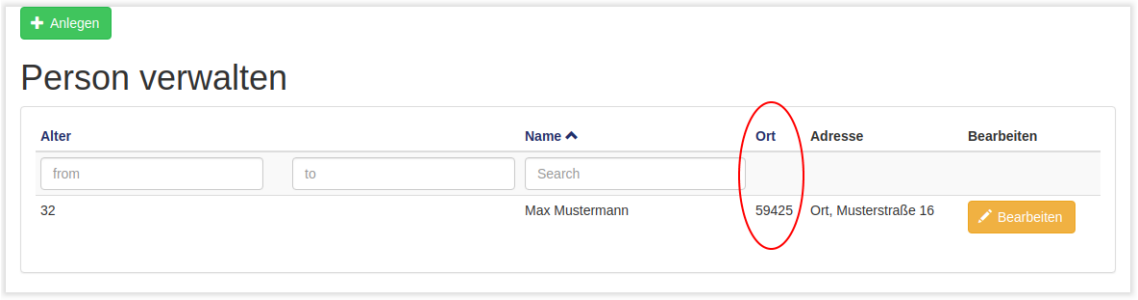


Abbildung 17: Derived Attribut in der Übersicht

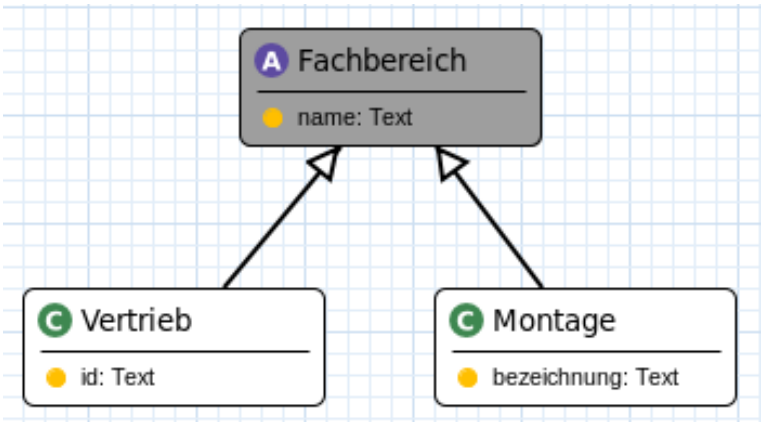


Abbildung 18: Zwei konkrete Typen erben von einem abstrakten Typen

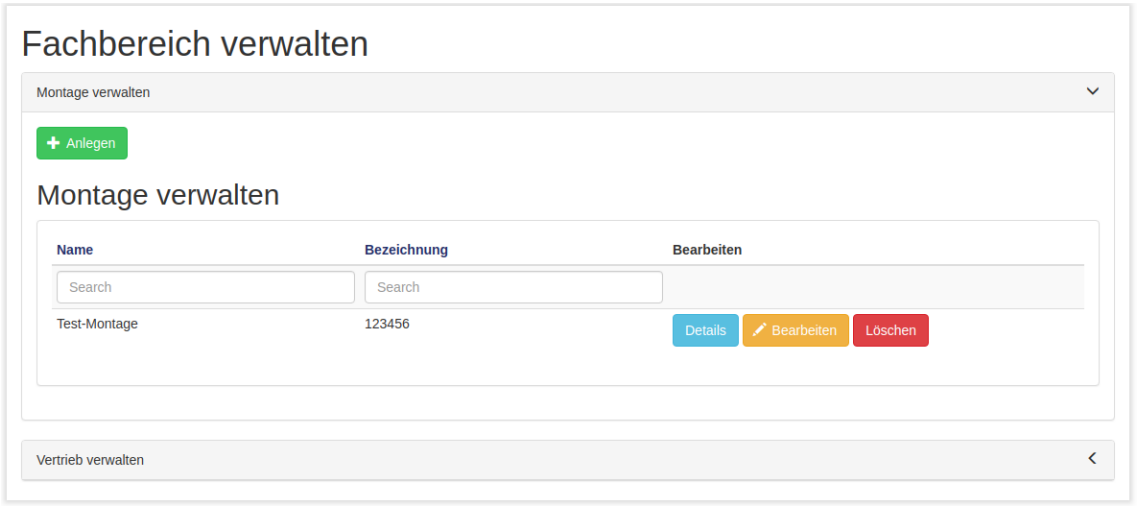


Abbildung 19: Übersicht für zwei konkrete Typen, die von einem abstrakten Typen erben

konkreten Typen *Vertrieb* und *Montage* von dem abstrakten Typen *Fachbereich* erben. Abbildung 19 zeigt die zugehörige Übersicht, die den Titel der abstrakten Übersicht und die Übersichten aller erbenden konkreten Typen in einklappbaren Panels enthält. In diesem Beispiel ist die Übersicht von *Vertrieb* aktuell eingeklappt und die Übersicht von *Montage* ausgeklappt. Die Panels können über das Klicken auf die graue Leiste ein- und ausgeklappt werden. Die erbenden Typen enthalten neben den eigenen Attributen, auch die geerbten Attribute (*name*) des Obertypen.

Die nächste Abbildung 20 zeigt eine Ontologie, in der der abstrakte Typ *Programmierung* von dem abstrakten Typen *Fachbereich* erbt. Zusätzlich erbt der konkrete Typ *SPS* von dem abstrakten Typen *Programmierung*. Abbildung 21 zeigt die zugehörige Übersicht. Bei einer Kette von abstrakten Vererbungen wird jeweils der Titel des höchsten Elements angezeigt. Die konkrete Übersicht von *SPS* ist wieder in einem Panel enthalten. Abbildung

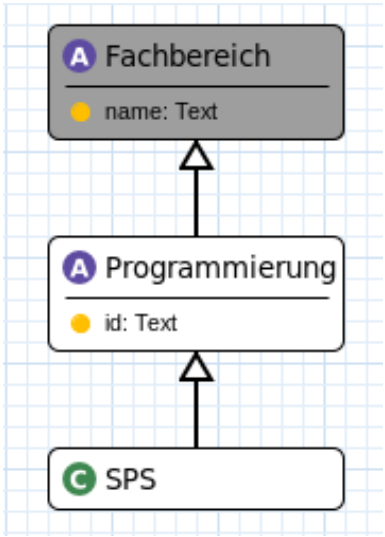


Abbildung 20: Vererbung zwischen zwei abstrakten Typen und einem konkreten Typen

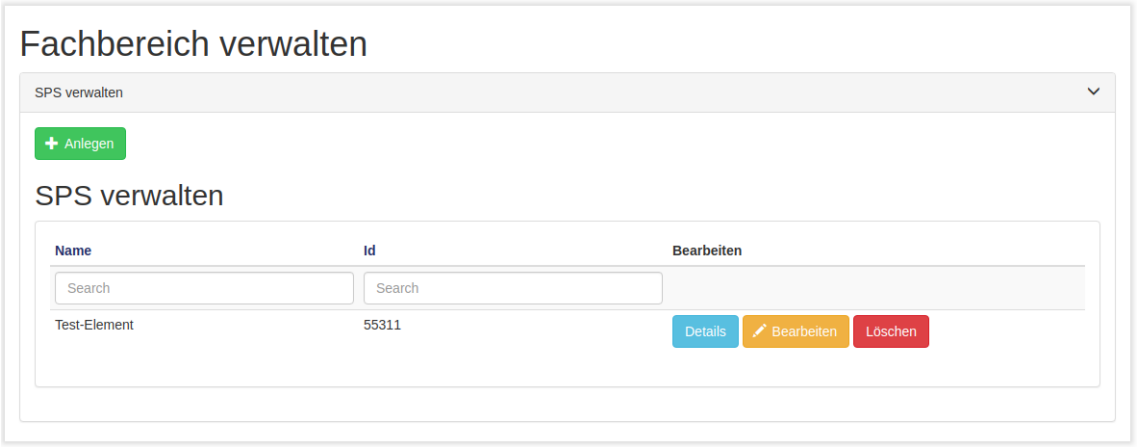


Abbildung 21: Übersicht für die Vererbung zwischen zwei abstrakten Typen und einem konkreten Typen

22 und 23 erzeugen beide eine Übersicht, die den Titel der höchsten Instanz *Fachbereich* trägt. Da Fachbereich ein vererbender konkreter Typ ist, enthält die Übersicht auch die eigene Übersicht für *Fachbereich*. Zusätzlich ist die konkrete Übersicht des erbenden Subtypen *SPS* enthalten (Abbildung 24).
Der abstrakte Typ *Programmierung* wird, wie auch bei Abbildung 20, nicht angezeigt.

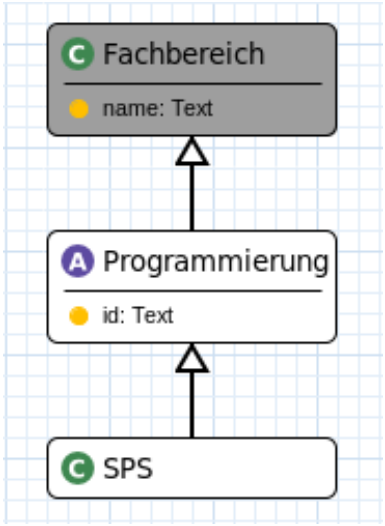


Abbildung 22: Vererbungshierarchie mit zwei konkreten und einem abstrakten Typen

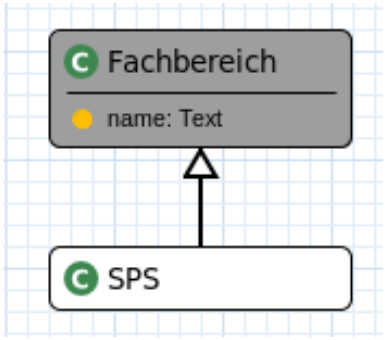


Abbildung 23: Vererbungshierarchie mit zwei konkreten Typen

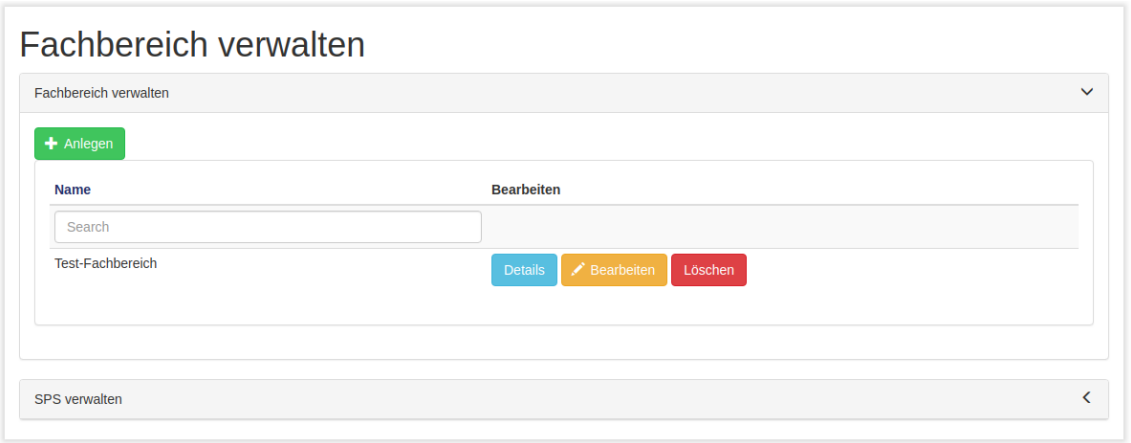


Abbildung 24: Übersicht für konkrete vererbende Typen

2.3 Bearbeiten

In der Bearbeiten-GUI können alle Attribute gesetzt werden. Die Attribute sind dazu übereinander aufgelistet. Abbildung 25 zeigt die Beispielontologie anhand der die verschiedenen Optionen der Bearbeiten-GUI erklärt werden. Der Typ *Person* enthält die

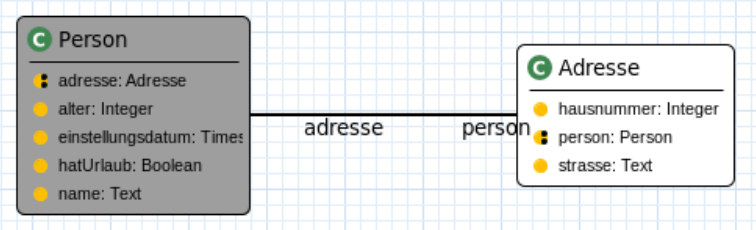


Abbildung 25: Ontologie 1 zum Bearbeiten

primitiven Attribute *name*, *alter*, *hatUrlaub* und *einstellungsdatum*, die die verschiedenen primitiven Datentypen repräsentieren. Abbildung 26 zeigt wie primitive Attribute in der

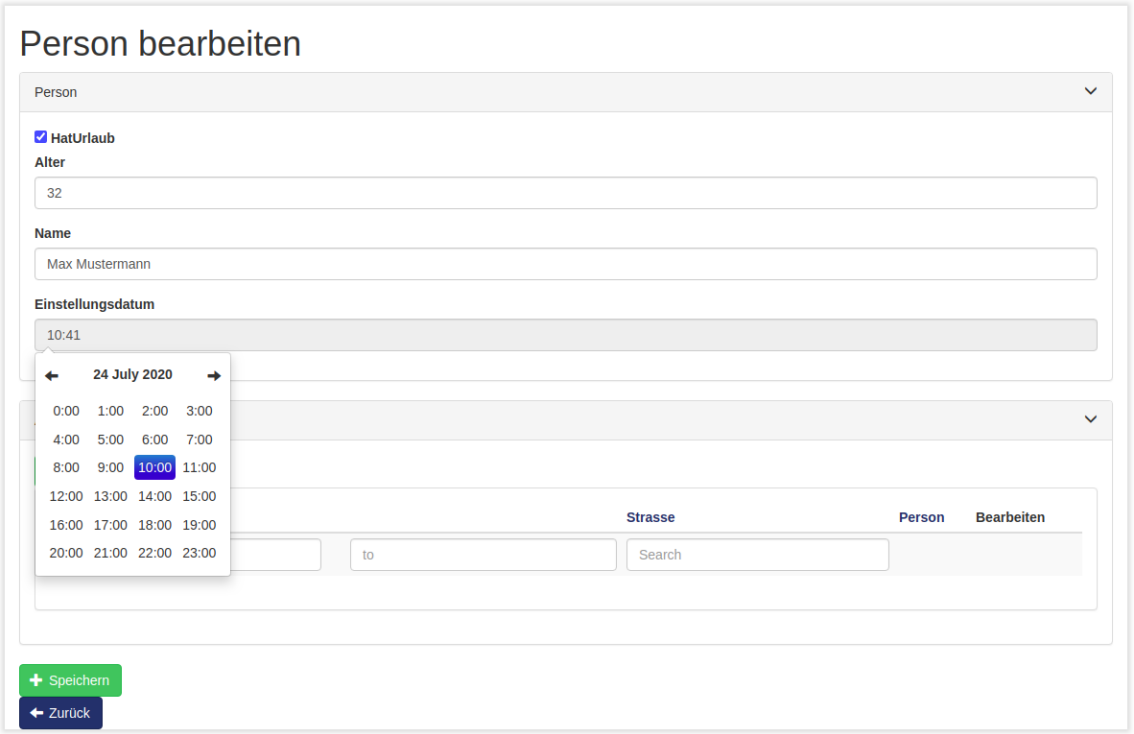


Abbildung 26: Primitive Attribute bearbeiten

Bearbeiten-GUI erstellt und bearbeitet werden können. Das Attribut *name* ist ein Text und das Attribut *alter* ist ein Integer. Der Wert des Booleans *hatUrlaub* kann über eine Checkbox gewählt werden. Der Wert des TimeStamps *einstellungsdatum* kann über das Auswahlmenü bestimmt werden. Abbildung 27 zeigt die Überprüfung der gewählten Optionen *isUnique* bei *alter* und *isRequired* bei *name*. Mit *isRequired* markierte Felder werden blau unterlegt. Dies deutet darauf hin, dass dieses Feld ausgefüllt werden muss. Für *isUnique* findet beim Speichern eine Überprüfung statt, ob es bereits ein Objekt gibt, das den selben Wert beinhaltet. Sollte es bereits so ein Objekt geben, wird die rot unterlegte Nachricht aus Abbildung 27 angezeigt.
In der Ontologie aus Abbildung 25 enthält der Typ *Person* das komplexe Attribut *adresse*. Dieses Attribut hat vier verschiedene Anzeigemöglichkeiten. Der *createEditType* kann auf *creatable*, *viewable*, *hidden* oder *selectable* gesetzt werden. *Selectable* hat zusätzlich die Option *tableInsteadOfCheckBox*. Bei Auswahl wird eine Tabelle zum bearbeiten genutzt, sonst wird eine Checkbox angezeigt. Bei Auswahl von *creatable* wird eine Übersichts-

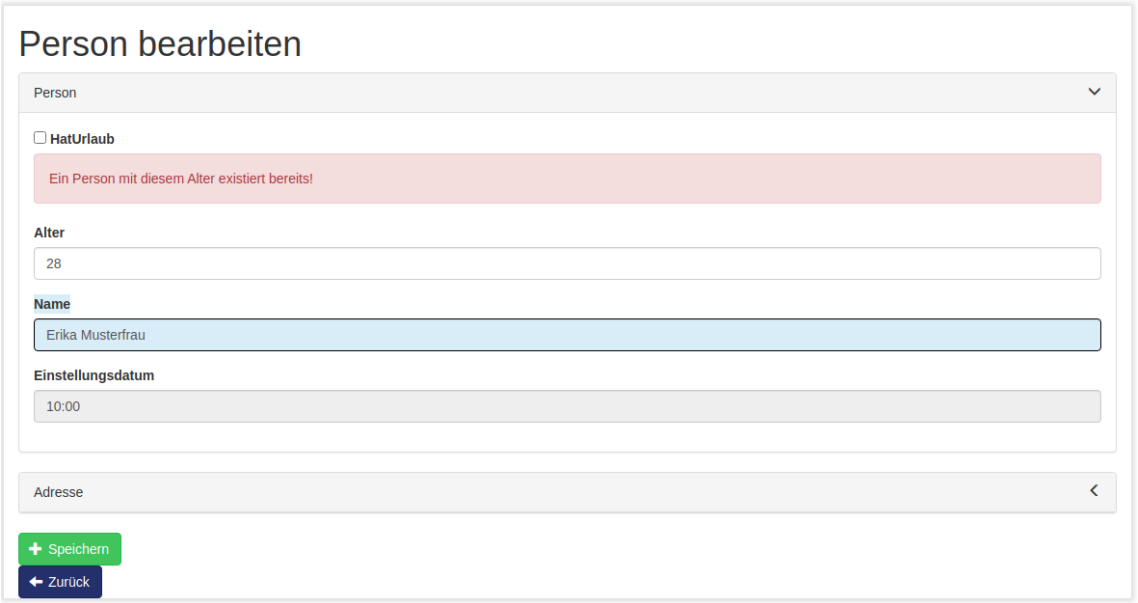


Abbildung 27: *isUnique* und *isRequired* Überprüfung

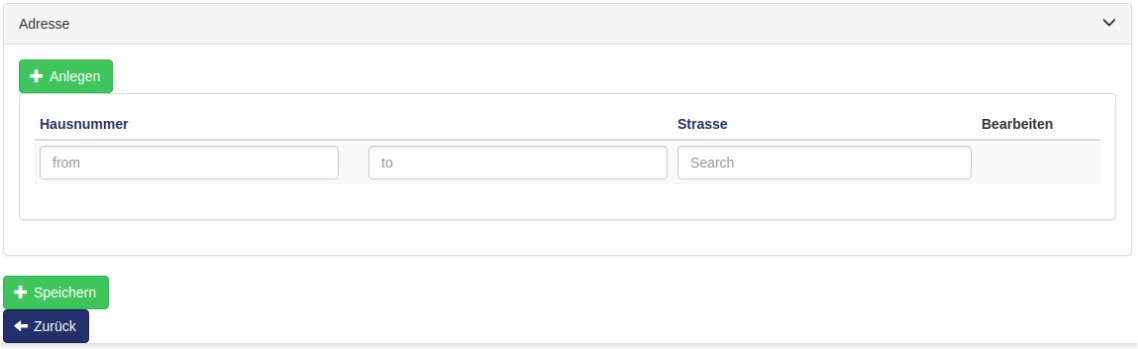


Abbildung 28: Anzeige der Bearbeiten-GUI bei Auswahl von *Creatable*

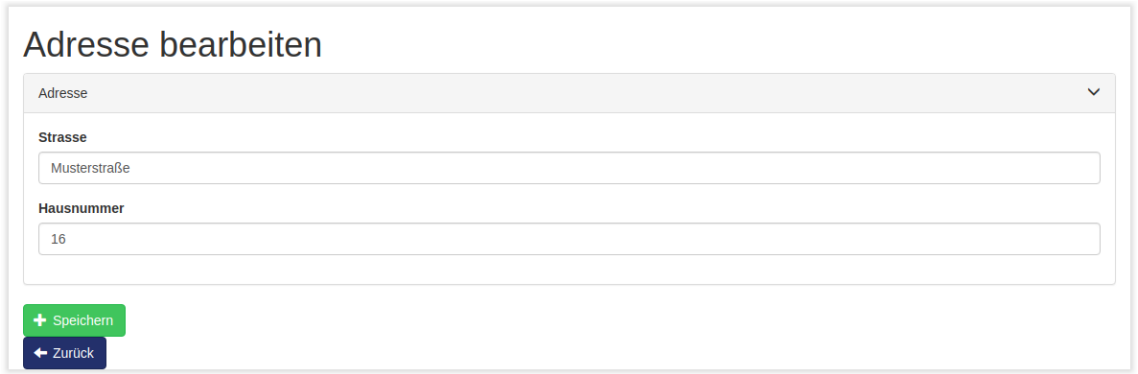


Abbildung 29: Anzeige der Bearbeiten-GUI von *Adresse* in *Person*

GUI des zu erstellenden Objekts im Bearbeiten-Menü eingebunden. Abbildung 28 zeigt die eingebundene Übersicht von *Adresse*. Bei Klick auf den Knopf *Anlegen* öffnet sich die Bearbeiten-GUI von *Adresse* (siehe Abbildung 29). In dieser kann eine Adresse erstellt werden, die der zugehörigen Person zugewiesen wird. Bei Klick auf *Speichern* wird der Vorgang beendet. Abbildung 30 zeigt die zugewiesene Adresse. In diesem Fall ist *adresse*

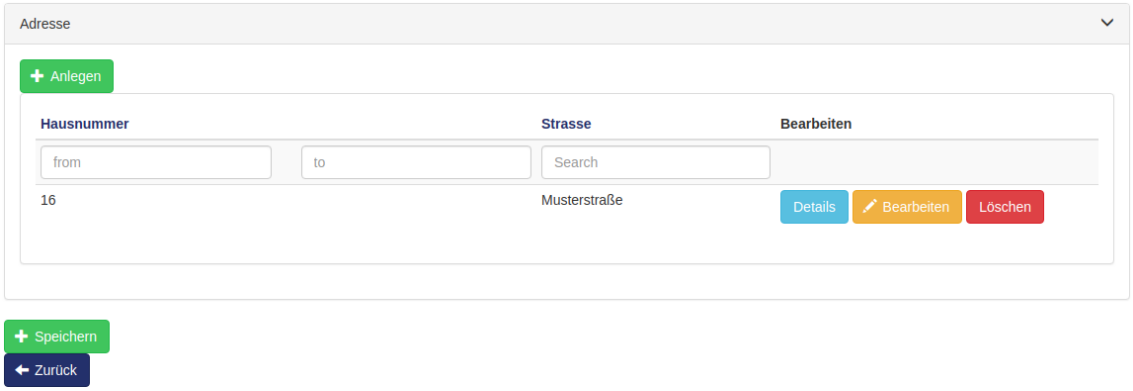


Abbildung 30: Anzeige der Bearbeiten-GUI nach Erstellung eines Objekts durch *Creatable*

im Typ *Person* keine Liste, daher kann nur ein Objekt kontingentiert werden. Über einen erneuten Klick auf *Anlegen* wird das vorher erstellte Adressen-Objekt überschrieben. Bei Listen würden mehrere Adressen-Objekte zu *Person* hinzugefügt werden.
In der Ontologie aus Abbildung 25 ist der *createEditType* von *person* im Typ *Adresse* auf *viewable* gesetzt. Abbildung 31 zeigt das Bearbeiten-Menü von Adresse. Da Person als

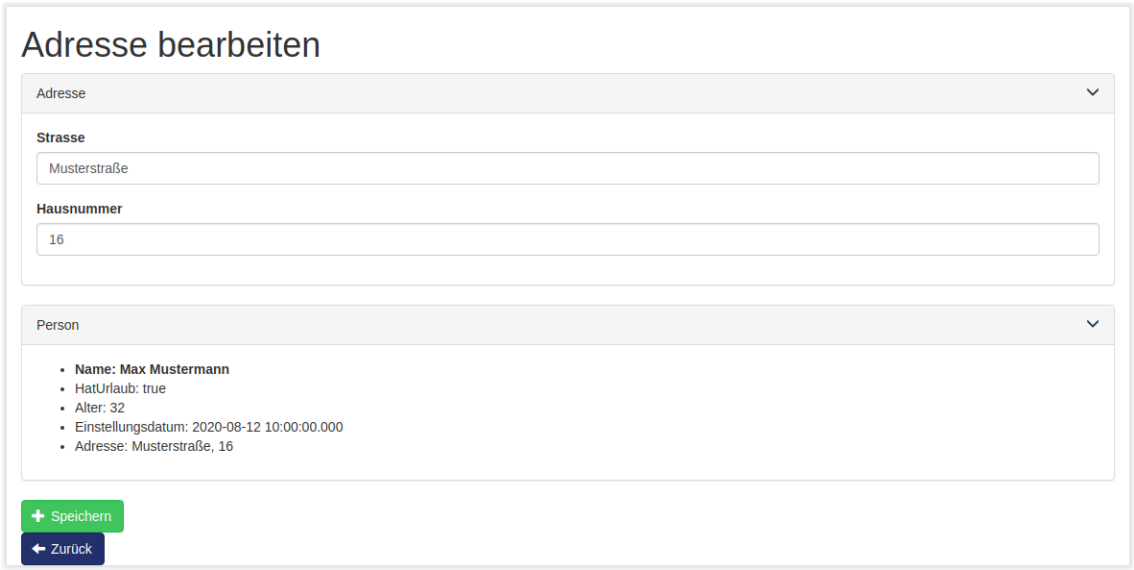


Abbildung 31: Anzeige der Bearbeiten-GUI bei Auswahl von *Viewable*

viewable gesetzt wurde, werden die zugewiesenen Personen angezeigt.
Attribute deren *createEditType* auf *hidden* gesetzt ist, werden gar nicht angezeigt, obwohl die zugehörige Referenz existiert. Falls der *createEditType* von *person* im Typ *Adresse* auf *hidden* gesetzt wird, entspricht die zugehörige Bearbeiten-GUI der GUI aus Abbildung 29.
Für die Veranschaulichung von *selectable* wird eine zweite Ontologie verwendet, in der das Attribut *person* im Typ *Adresse* eine Liste ist (siehe Abbildung 25). Im ersten Fall wird *tableInsteadOfCheckbox* nicht markiert, sodass in der Bearbeiten-GUI eine Checkbox für das Attribut *person* zur Verfügung steht. Abbildung 33 zeigt die Bearbeiten-GUI von *Adresse*. Personen werden durch eine Checkbox ausgewählt. Bei Nicht-Listen Attributen

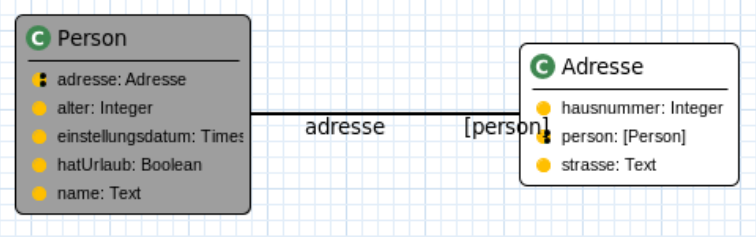


Abbildung 32: Ontologie 2 zum Bearbeiten

Abbildung 33: Anzeige der Bearbeiten-GUI bei Auswahl von *Selectable* als Checkbox

kann nur eine Person markiert werden. Abbildung 34 zeigt die Bearbeiten-GUI mit *se-*

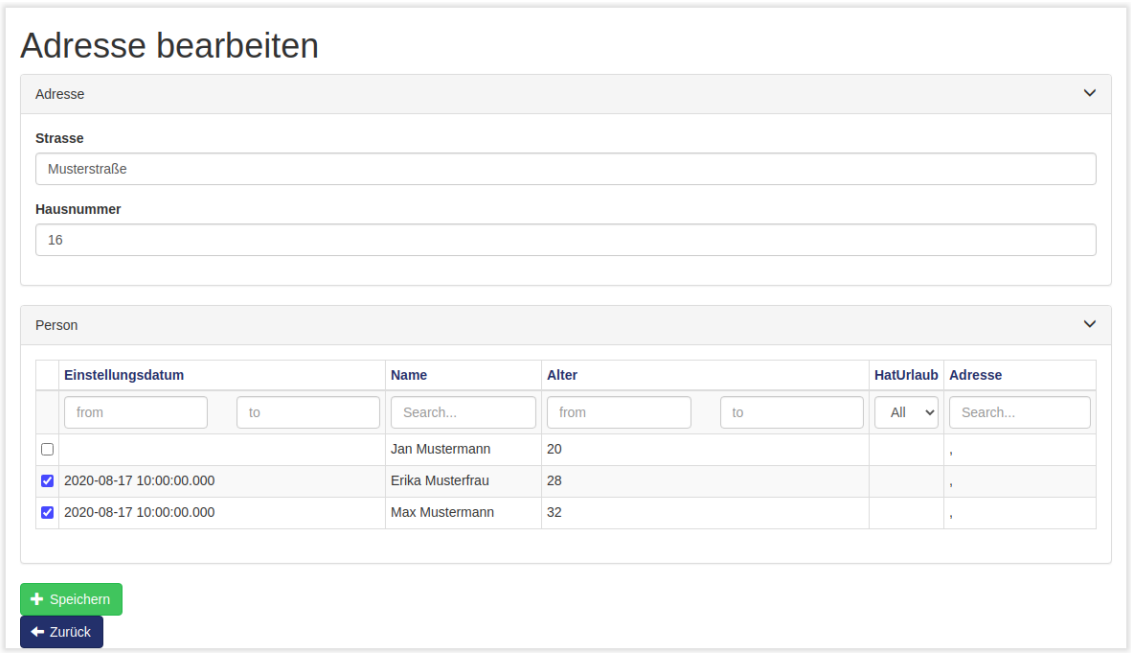


Abbildung 34: Anzeige der Bearbeiten-GUI bei Auswahl von *Selectable* als Tabelle

lectable als Tabelle. Es werden alle Informationen der Objekte angezeigt. Zusätzlich kann nach Objekten gesucht werden. Am linken Rand können die Elemente markiert werden. Bei Nicht-Listen kann wieder nur ein Element markiert werden.
Am Ende jeder Bearbeiten-GUI befindet sich ein Zurück-Button, über den die vorherige GUI erreicht wird.

2.4 Detailansicht

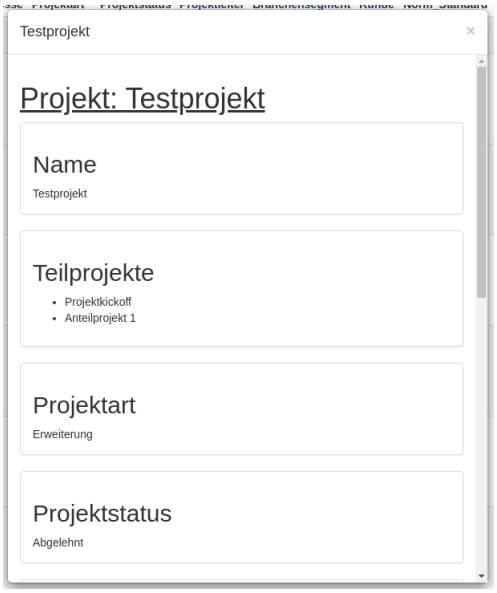


Abbildung 35: Beispielhafte Detailansicht (oberer Teil)

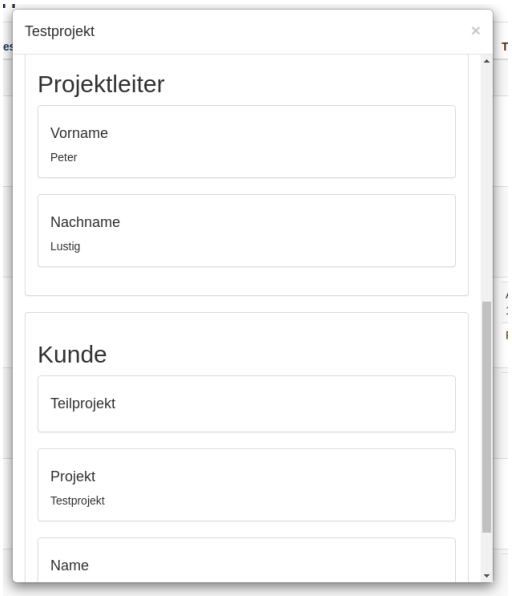


Abbildung 36: Beispielhafte Detailansicht (unterer Teil)

Die Detailansicht ist ein Pop-Up-Fenster, welches eine Übersicht über die Attribute eines Objektes bietet. Die Darstellung dieser Attribute unterscheidet sich, abhängig von deren Typ und den Attributparametern, die gesetzt wurden. Die Erklärung der verschiedenen Visualisierungsarten erfolgt anhand von Abbildungen 35 und 36.

2.4.1 Primitive Attribute/Enums

Die Darstellung der primitiven Attribute ist festgelegt, und besteht immer aus dem Attributnamen und dem Attributwert. In Abbildung 35 ist das Attribut *Name* ein Beispiel hierfür. Eine Anpassung der Darstellung ist nicht möglich. Analog dazu werden Enums mit ihrem Bezeichner aufgeführt (siehe *Projektstatus* in Abbildung 35).

2.4.2 AbstractTypes

Da AbstractTypes nur als Strukturelemente vorgesehen sind, werden diese nicht dargestellt. Dies ist unabhängig von den gesetzten Attributparametern.

2.4.3 ConcreteTypes

Die Darstellung von ConcreteTypes erfolgt abhängig von den gewählten Attributparametern. Falls *completeDetailViewInformation* auf *false* gesetzt wurde, wird dieses At-

tribut ähnlich einem primitiven Attribut nur als einfache Auflistung der/des Identifiers präsentiert. Eine ausführliche Darstellung erfolgt andernfalls. Diese ausführliche Darstellung liegt in Abbildung 36 für die Attribute *Projektleiter* und *Kunde* vor.
Falls das Attribut als Liste angelegt wurde, kann durch das *view* Parameter noch die Struktur der Auflistung festgelegt werden. Zur Auswahl stehen hier die Listendarstellung (wobei die einzelnen Elemente auf ihre Identifier reduziert werden), sowie eine Tabellendarstellung. Eine Listendarstellung ist in Abbildung 35 für das Attribut *Teilprojekte* gezeigt.
Die einzelnen (Unter-)Attribute des Typen des Attributs werden als kurze Version aufgelistet. Dies bedeutet, dass ComplexAttributes (auf der zweiten Ebene) nur mit ihrem Identifier aufgeführt werden, da sonst eine Auflistungskette von theoretisch unbegrenzter Länge entstehen könnte, was der Benutzerfreundlichkeit entgegenstehen würde. Ein Beispiel hierfür ist das Attribut *Projekt* vom Attribut *Kunde*, welches zurück auf das ursprüngliche Projekt, dessen Detailansicht generiert wurde, verweist (Abbildung 36).

2.5 Bäume

2.5.1 Was ist eine Baumstruktur und warum wird sie genutzt?

Die in Kapitel 1.1.4 eingeführten Concrete-Tree-Types führen in der Webapp zu einem Konstrukt, welches vergleichbar mit einer sogenannten Baumstruktur ist. Eine Baumstruktur kann dazu genutzt werden, verschiedene Elemente oder Themen zusammenzufassen, bei denen mehrere Elemente zu genau einem anderen gehören. Somit lassen sich also hierarchische Strukturen, wie beispielsweise einen Technologiebaum oder ein Versionsbaum, darstellen und verwalten. Insbesondere bei Versionen ist es einfach, sich deren Abhängigkeiten vorzustellen: Eine Version „V1.0“ oder „V2.0“ ist erstmal unabhängig von anderen Version. Dagegen hängt die Version „V1.1“ von „V1.0“ ab, und die Versionen „V1.1.0“ und „V1.1.1“ beide von „V1.1“.

Damit dies in der Webapp umgesetzt werden kann, ist es möglich, sehr große Baumstrukturen mit beliebigen Ebenen anzulegen. Hält man sich beim Erstellen an die Grundidee der Bäumen, sind alle Einträge auch jederzeit zusammen mit anderen ähnlichen Einträgen gruppiert, was durch den graphischen Aufbau der Webapp unterstützt wird.

2.5.2 Beispiele in der Webapp

In der fertigen Webapp gibt es zwei Stellen, an denen die Vorteilen der Baumstrukturen genutzt werden: Die Technologien, welche in der Rootelement-Leiste gefunden werden können, sowie die Branchen, welche man innerhalb eines Geschäftsfelds bearbeiten kann.

Baumstrukturen eignen sich gut zur Darstellung von Technologien, da Technologien so aufeinander aufbauen, dass man häufig eine andere Technologie als Grundlage für eine neue benötigt. Weiterhin ist es auf diese Weise einfach, verschiedene Ausprägungen von einer bestimmten Technologie zu unterscheiden. Ein Beispiel dafür sind die unterschiedlichen Reifendicken aus der Figur 38. Im folgenden werden vor allem Technologien als Beispiel für die Benutzung der Baumstrukturen in der Webapp verwendet.

Ähnliches gilt aber auch für die Branchen. Eine Branche kann in mehrere kleine Teilbranchen aufgeteilt werden, die dann aber alle auch noch zu ihrer Oberbranche gehören. Dies lässt sich perfekt mit einer Baumstruktur abbilden. Beispielsweise kann man die Branche „Automobil“ in Fördertechnik oder Antriebstechnik unterteilen (vgl. Figur 37).

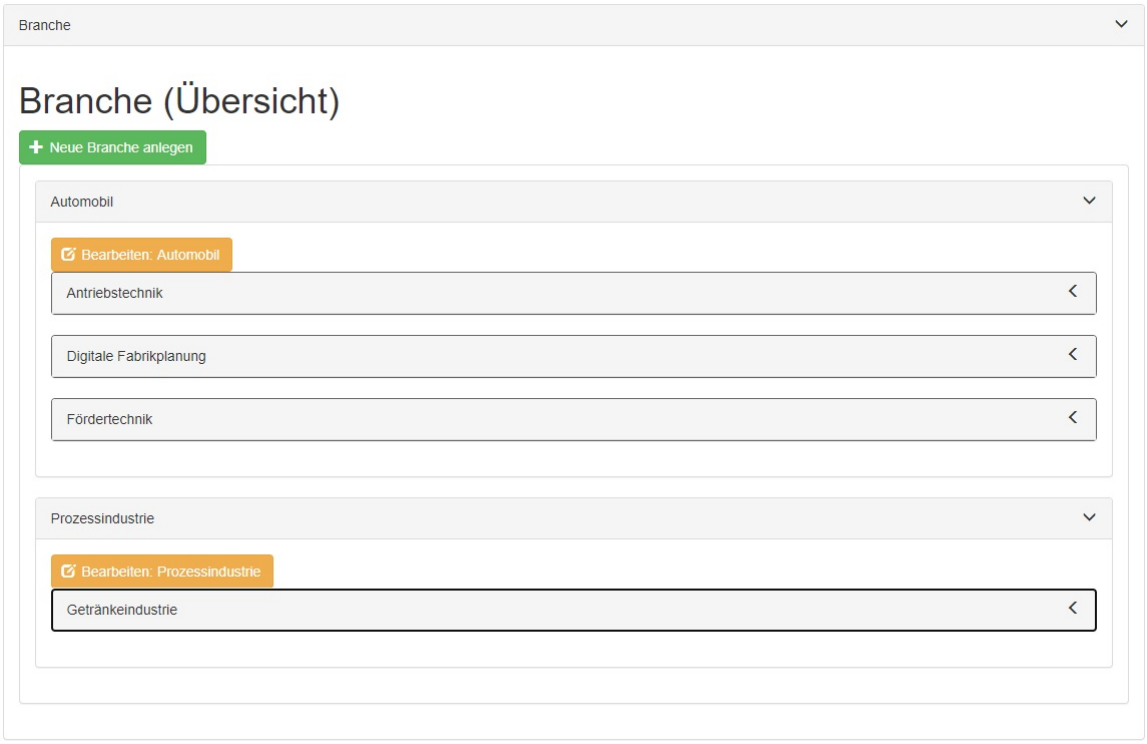


Abbildung 37: Baum für die Branchen

2.5.3 Die Übersicht

Wenn man eine neue Technologie anlegen möchte, wird man sich zuerst in der Übersichts-GUI für die Technologien wiederfinden (Figur 38). Diese ist mit dem Namen „Technologie (Übersicht)“ gekennzeichnet. In dieser GUI sind drei verschiedene Aktionen möglich.

Mit dem grünen Knopf „Neue Technologie anlegen“ wird eine komplett neue Technologie erstellt (siehe Kapitel 2.5.4) . Diese Technologie ist erstmal nicht mit anderen Technologie verknüpft und zählt damit als Oberelement. Deswegen sollte diese Funktion nur dann genutzt werden, wenn man auch eine solche Obertechnologie erhalten möchte.

Dadrunter befinden sich alle anderen bisher angelegten Obertechnologien. Somit kann es möglich sein, dass dieser Bereich leer ist. Ist er das nicht, hat man die Möglichkeit, die einzelnen Technologien ein- oder auszuklappen, um entsprechend weniger oder mehr der darunter liegenden Technologien sehen zu können. Dafür muss die Box, die den Namen der Technologie enthält und am rechten Rand einen Pfeil nach unten hat, angeklickt werden.

Wenn eine Technologie ausgeklappt ist, befindet sich unter ihr ein orangener „Bearbeiten“-Knopf. Mit diesem kann man die entsprechende Technologie bearbeiten, indem man die Maske erreicht, in der es einem ermöglicht wird, bereits erstellte Technologien nachträglich zu ändern (siehe dazu Kapitel 2.5.5).

2.5.4 Die Anlegen-/Bearbeiten-Übersicht

Nachdem man eine neue Technologie anlegen möchte oder bei einer bestehenden die bereits eingetragenen Daten verändern möchte, wird einem die Anlegen- oder die Bearbeitenübersicht angezeigt. Beide Übersichten bieten exakt die gleiche Funktionalität: Für jedes Attribut, welches die Technologie besitzt, hat man die Möglichkeit, einen neuen

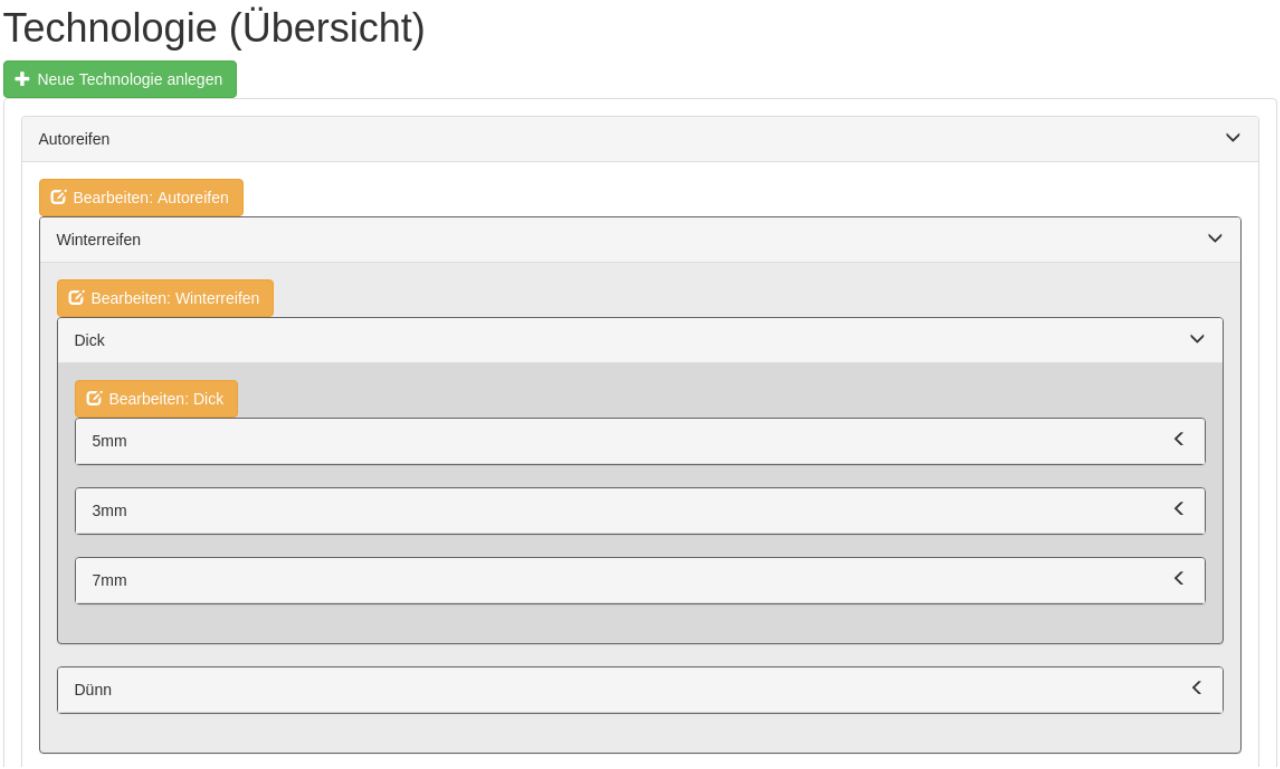


Abbildung 38: Vorschau des Technologiebaums

Wert festzulegen. In unserem Beispiel hat eine Technologie lediglich einen Namen (vgl. Figur 40); allgemein kann diese GUI aber auch deutlich umfangreicher werden.

Möchte man keine Änderungen durchführen, bricht man seine Aktion mit dem „Zurück“-Knopf ab. Ansonsten werden die Änderungen beim Klick auf den „Ändern“-Knopf übernommen.

2.5.5 Das Fenster innerhalb der Technologie

Nachdem man in der Übersichts-GUI eine neue Technologie erfolgreich angelegt hat oder dort eine bestehende bearbeiten möchte, wird man zu der Bearbeiten-GUI weitergeleitet (Figur 39). Alle Inhalte dieser GUI beziehen sich auf die neu angelegte bzw. die ausgewählte Technologie (die aktuelle Technologie). Diese GUI lässt sich in 4 Teile einteilen.

Ganz oben befindet sich die Navigationsleiste (siehe Kapitel 2.5.6) sowie die Überschrift.

Als nächstes werden die Unterelemente angezeigt. Dies sind alle Technologien, die von der momentan aktuellen Technologie abhängen. Durch diese kann man wie durch die Übersichts-GUI navigieren.

Danach kommt eine kurze Auflistung der Attribute, die in der aktuellen Technologie gespeichert sind.

Außerdem gibt es noch drei Knöpfe, die unterschiedliche Funktionen erfüllen. Mit dem „Unterelement hinzufügen“-Knopf kann man der aktuellen Technologie eine neue Untertechnologie hinzufügen. Dies ist vergleichbar mit dem Erstellen aus der Übersichts-GUI. Mit dem „Entfernen“-Knopf wird die aktuelle Technologie sowie alle ihre Einträge und Untertechnologien gelöscht. Der „Bearbeiten“-Knopf führt zu der GUI aus Kapitel 2.5.4 und ermöglicht die Änderung der Attribute der aktuellen Technologie.

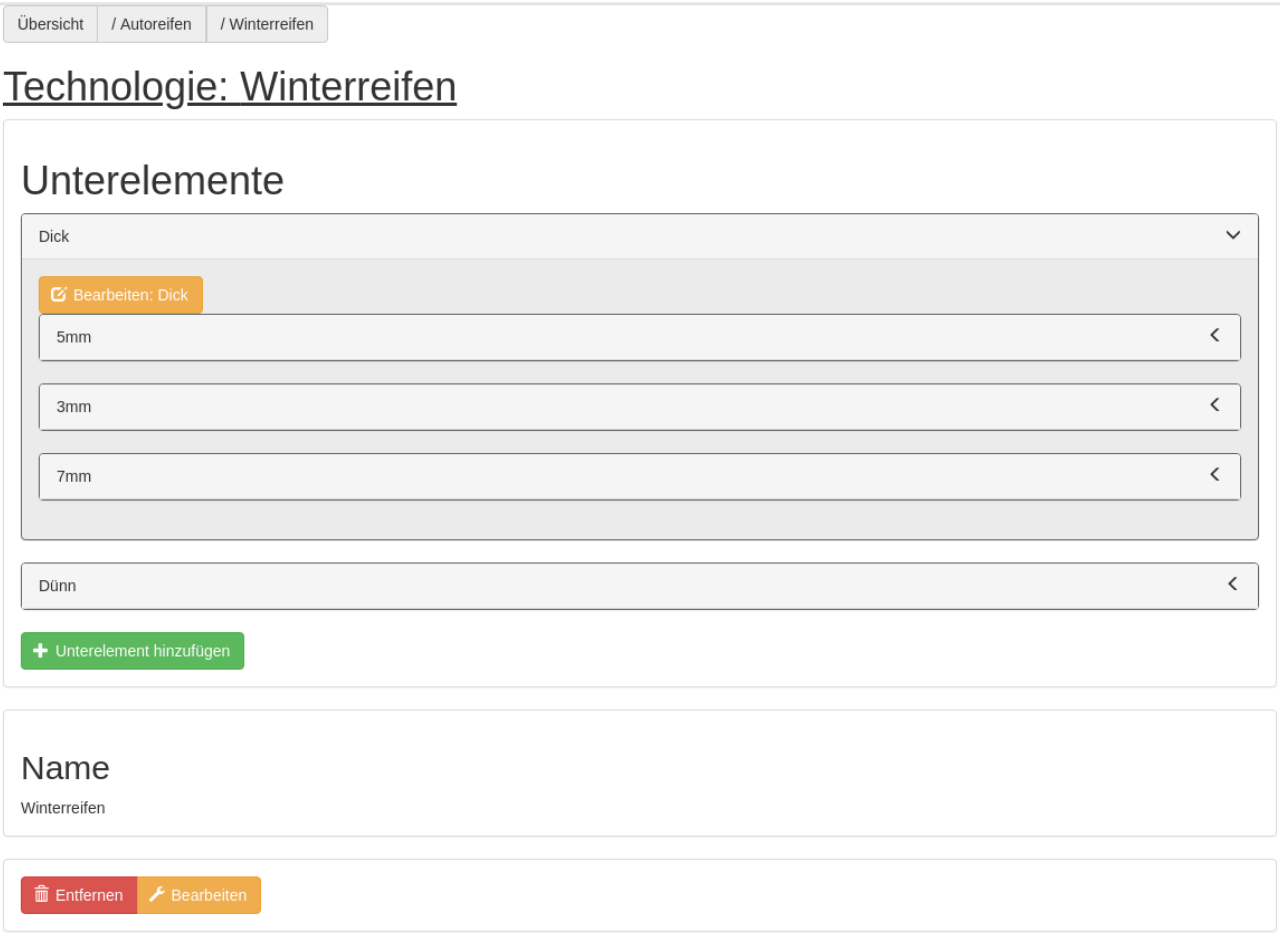


Abbildung 39: Vorschau der Bearbeiten-GUI

2.5.6 Die Navigationsleise

Die Navigationsleiste (obere Teil von Figur 39) bietet eine Orientierungshilfe, an welcher Stelle in der Baumstruktur man sich als Nutzer momentan befindet. Zusätzlich ermöglicht sie eine beschleunigte Navigation durch eben diese, da man mit einem Klick auf „Autoreifen“ direkt in die Technologieansicht für die Autoreifen springen kann.

2.5.7 Zuweisung von Baumtypen

In den vorigen Abschnitten wurde erklärt, wie man Baumstrukturen anlegt und verwaltet. Jetzt wird die Benutzung von Baumstrukturen in anderen Typen, also die Zuweiung zu diesen, vorgestellt.

In unserem Beispiel sollen einem Teilprojekt verschiedene Technologien zugewiesen werden. Dafür navigiert man zu dem Teilprojekt, welches bearbeitet werden soll. Da man Technologien zu Teilprojekten zuweisen kann, findet man in der Bearbeiten-GUI des Teilprojektes das Baumauswahlmenü (Figur 41). Dieses kann ähnlich zu der Übersichts-GUI für Bäume benutzt werden.

Bei jedem einzelnen Technologieeintrag wird durch Farbe und Text deutlich gemacht, wie sein aktueller Status ist. Am Anfang (Figur 41) ist noch nichts hinzugefügt worden. Deshalb ist jede Technologie rot eingefärbt und mit dem Text „Nicht hinzugefügt“ versehen. Zusätzlich bietet jede Technologie die Möglichkeit an, durch Drücken des ent-

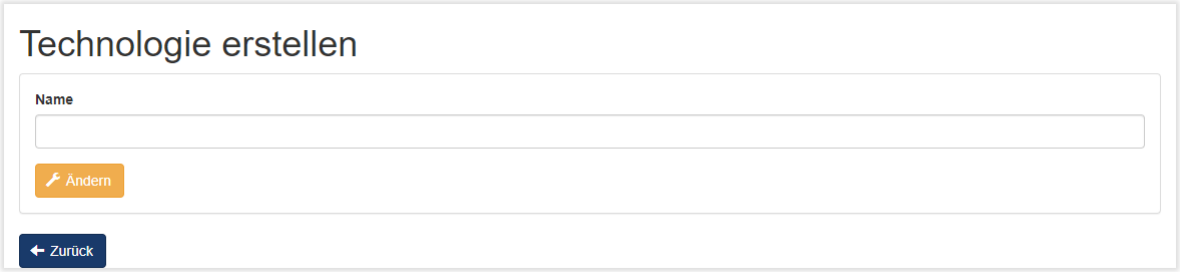


Abbildung 40: Die GUI zum Erstellen einer neuen Technologie

sprechenden Knopfs hinzugefügt zu werden.

Im Beispiel (Figur 42) wurde die Technologie „Profil: Dick“ hinzugefügt. Dadurch wurde sie grün eingefärbt und mit dem Text „Hinzugefügt“ versehen. Da die Technologien „5mm“ und „7mm“ eine Untertechnologie von „Profil: Dick“ sind, wurden beide ebenso hinzugefügt. Die Technologie „Winterreifen“ dagegen ist blau eingefärbt und mit dem Zusatz „Implizit hinzugefügt“ versehen. Dies signalisiert, dass diese Technologie zwar nicht direkt ausgewählt wurde, aber ihre Untertechnologien ab jetzt in dem Teilprojekt enthalten sind.

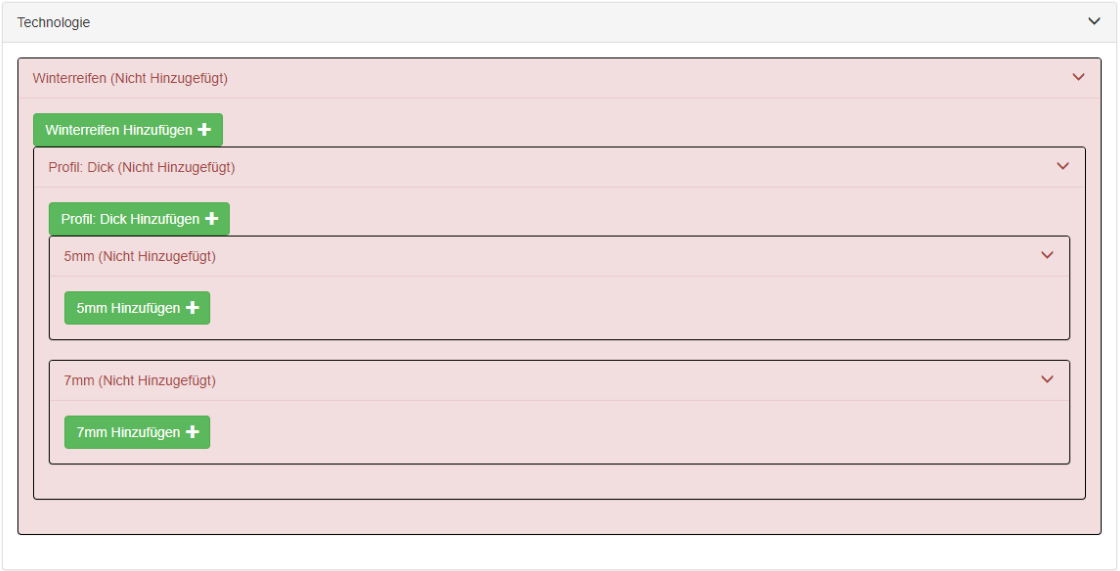


Abbildung 41: Die Baumauswahl

3 Canvas

Neben der Ontologie können Canvases erstellt werden. Dafür muss in der domänspezifischen Sprache mindestens ein Canvas ausgewählt werden, sodass dieser in der WebApp verwendet werden kann. Im Folgenden werden die Einstellungsmöglichkeiten der domänspezifischen Sprache und die Bedienung des daraus resultierenden Canvas in der WebApp erläutert.

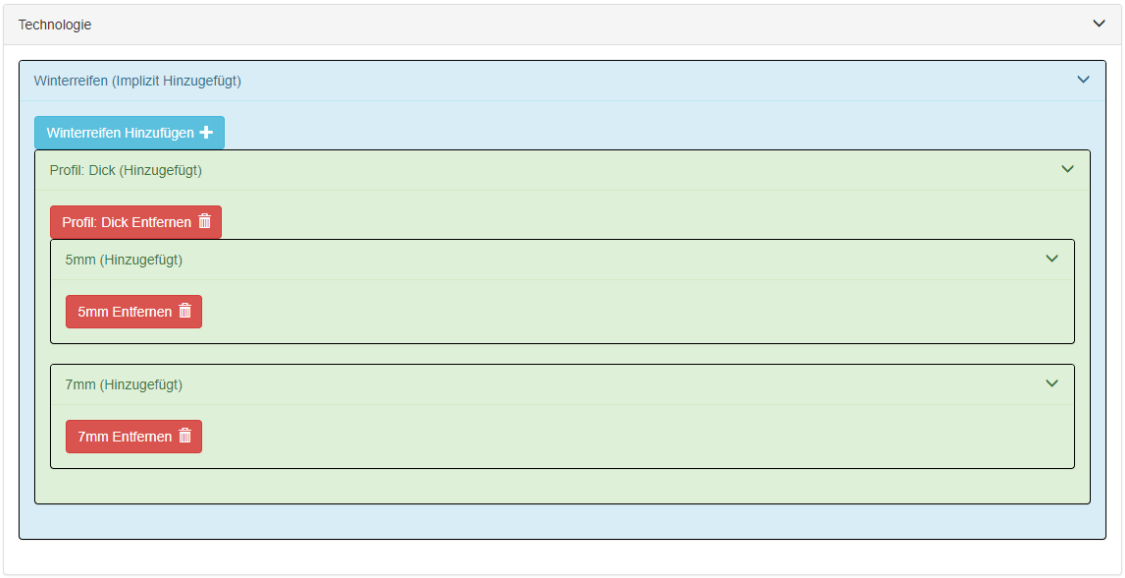


Abbildung 42: Nach der Auswahl von „Profil: Dick“

3.1 Domänspezifische Sprache

3.2 WebApp

Mit einem Klick auf den Tab „Canvases“ in der oberen Menüleiste wird der Canvasteil der WebApp geöffnet.
Im Folgenden Abschnitt wird der Canvas in der WebApp erläutert. Dabei wird zunächst auf die Übersichtsdarstellung der Canvases eingegangen. Danach wird ein Beispielcanvas erläutert, wie er mit Hilfe der domänspezifischen Sprache erstellt sein kann. Zum Schluss werden die verschiedenen Symbole in einem geöffneten Canvas erklärt und die Art der Auswahleinschränkung beim Hinzufügen oder Löschen von Elementen erläutert.

3.2.1 Übersicht Canvases

Wird auf „Canvases“ in der oberen Menüleiste geklickt, wird die Canvasübersicht aufgerufen. In dieser sind alle Canvases, die in der DSL spezifiziert wurden, untereinander aufgelistet. In Abbildung 43 ist ein Canvas mit dem Namen „BeispielCanvas“ generiert worden.
In der Tabelle in der zweiten Zeile ist ein Suchfeld. Hier kann ein Name eines Canvas eingegeben werden, um danach zu suchen.

Abbildung 43: Canvasübersicht ohne erstellten Canvas

Erstellen

Durch einen Klick auf den grünen, links oben befindlichen Button „Anlegen“ wird ein neuer Canvas angelegt. Dabei öffnet sich eine neue Ansicht, die in Abbildung 44 zu sehen ist. Hier muss ein Name in das Feld unter „Name“ geschrieben werden und durch einen Klick auf „Namen übernehmen“ bestätigt werden. Durch den blauen Button mit der Beschriftung „Zurück“ gelangt der Benutzer auf die Übersicht, die in Abbildung 45 dargestellt ist, zurück.

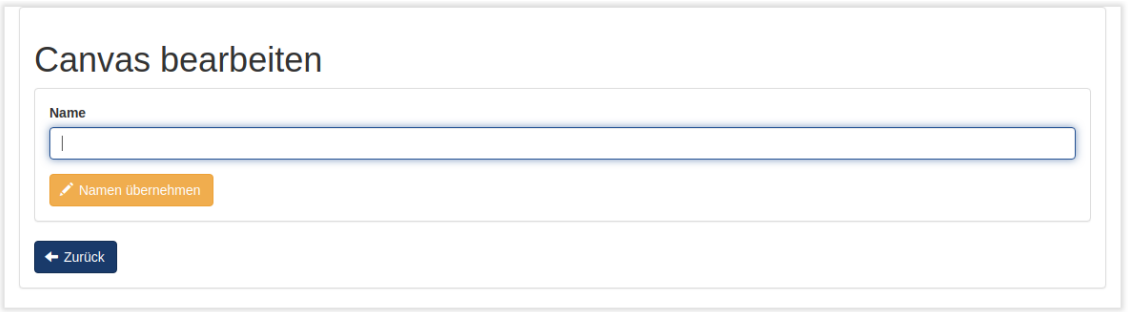


Abbildung 44: Erstellen eines Canvas

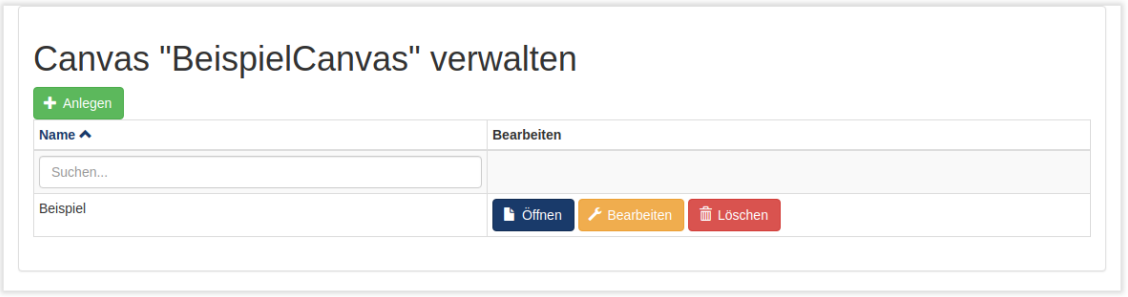


Abbildung 45: Übersicht mit einem Canvas mit dem Namen „Beispiel“

Löschen

Ein Canvas kann durch den roten Button „Löschen“ gelöscht werden. Dabei bleiben die Daten in der Ontologie bestehen.

Bearbeiten

Das Bearbeiten eines Canvas ist möglich durch das Drücken des Buttons „Bearbeiten“. Der Benutzer gelangt in das Fenster, welches in Abbildung 44 zu sehen ist, um dort den Namen zu verändern.

Öffnen

Durch das Drücken des Buttons „Öffnen“ wird ein Canvas geöffnet. Als Beispiel ist das in Abbildung 46 zu sehen.

3.2.2 Beispielcanvas

Der Canvas wird anhand des Beispielcanvas in Abbildung 46 erläutert, der dem Business Model Canvas nach Alexander Osterwalder (2005) nachempfunden wurde. In diesem Beispielcanvas (siehe Abbildung 46) wurden Endkunden, Projekt und Technologie als auswählbare Elemente hinzugefügt. Außerdem befindet sich ein Freitext in dem Feld „Customer Segment“.
Alle Daten, die in dem jeweiligen Canvas auswählbar sind, stammen aus der vorher angelegten Ontologie und verändern diese nicht.

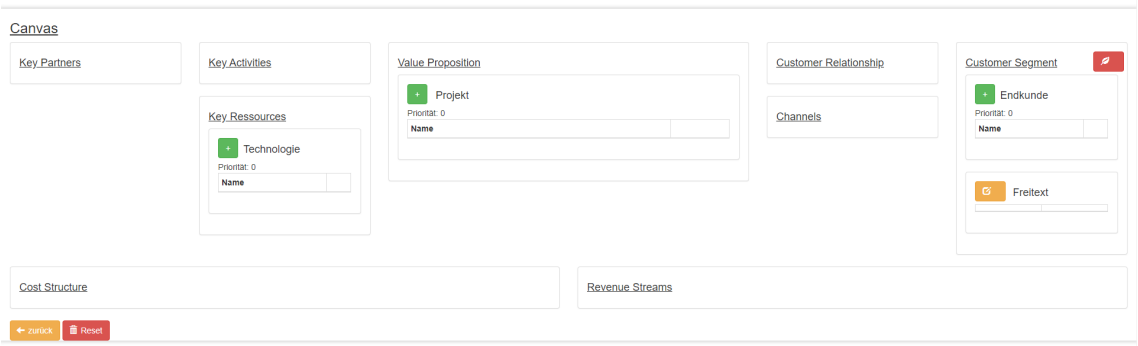


Abbildung 46: Der Beispiel Canvas

3.2.3 Symbole

In Abbildung 46 ist ein Beispielcanvas direkt nach der Erstellung zu sehen. Es fallen diverse Symbole auf, die im Folgenden näher erläutert werden.
Der Canvas aktualisiert sich nach jeder Aktion im Canvas automatisch, sodass Änderungen in der Ontologie sich direkt auf die möglichen und getätigten Auswahlen, basierend auf den vergebenen Prioritäten auswirken.

Freitext

Freitexte werden als Element im Canvas, wie in Abbildung 47 zu sehen ist, angezeigt. Das Freitextelement besteht aus beliebig vielen Freitexten, die als Listen angezeigt werden können. Die einzelnen Freitexte können Zeilenumbrüche darstellen.

Das Bearbeitungsmenü öffnet sich mit einem Klick auf den Freitext Button (siehe Abbildung 48).

In der Bearbeitungsansicht (siehe Abbildung 49) lässt sich über den Button ”Anlegen” ein neuer Freitext anlegen und über den Button ”Bearbeiten” und ”-” den jeweiligen Freitext bearbeiten bzw. löschen. ”Anlegen” und ”Bearbeiten” öffnet dabei das gleiche Untermenü (siehe Abbildung 50) mit einem Textfeld in das der neue Freitext eingetragen, bzw. der alte Freitext bearbeitet werden kann.

Die Eingabe kann über mehrere Zeilen erfolgen und Umbrüche enthalten. Über den Button ”Speichern” wird der Freitext gespeichert.

Um die Anzeige im Freitextelement direkt zu aktualisieren wird das Pop-up über den Button ”Schließen” beendet.

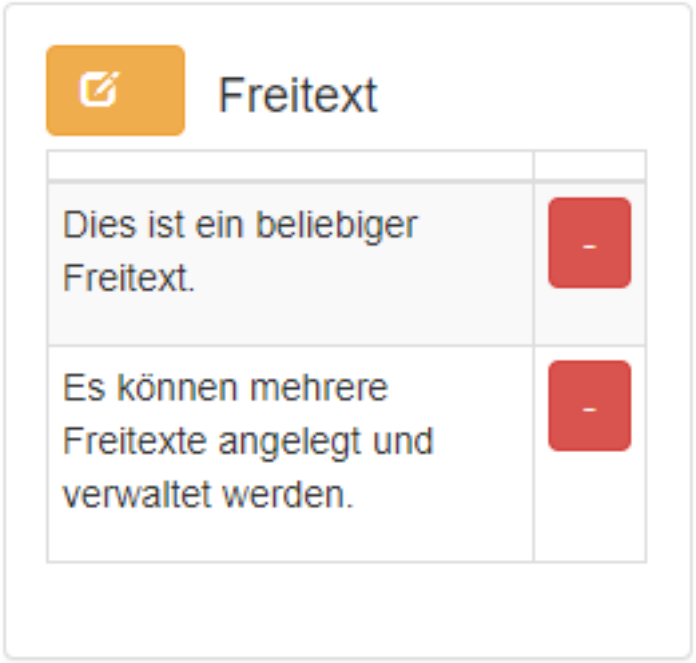


Abbildung 47: Freitextelement im Canvas

Freitext Notiz

Abbildung 48: Canvas Bedienelemente für Freitexte und Notizen

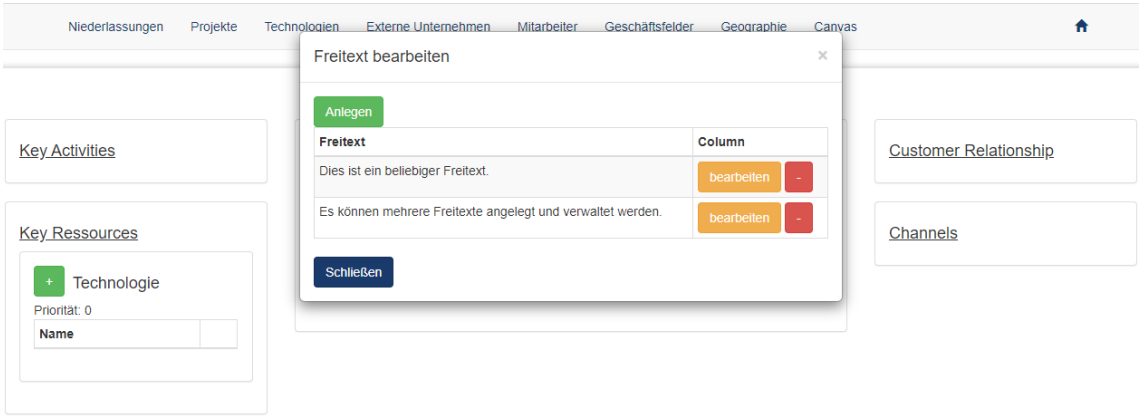


Abbildung 49: Freitext bearbeiten Ansicht

Notizen

Notizen werden im Canvas nicht direkt angezeigt. Sie können aufgerufen werden mit einem Klick auf den Notiz Button (siehe Abbildung 48) und öffnen sich als Pop-up. Wie in Abbildung 51 zu sehen verändert der Button seine Farbe und zeigt damit an ob eine Notiz vorhanden ist (grün) oder nicht (rot).

Nach einem Klick auf den Notiz Button öffnet sich die Notiz Ansicht (siehe Abbildung 52) in einem Pop-up. Anders als bei Freitext Elementen bestehen Notizen nur aus einzel-

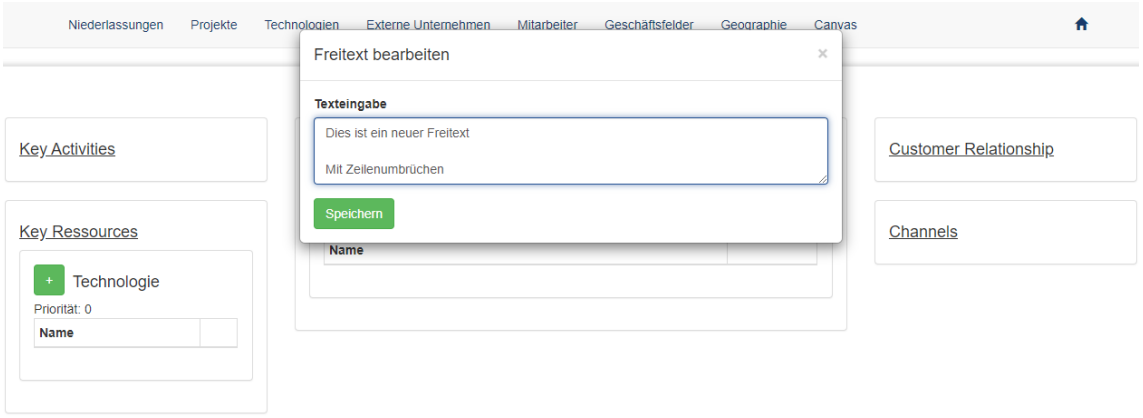


Abbildung 50: Ein neuer Freitext

nicht vorhanden vorhanden

Abbildung 51: Notiz Button Farbe

nen Einträgen, nicht aus Listen. Mit einem Klick auf den Bearbeiten Button wird eine

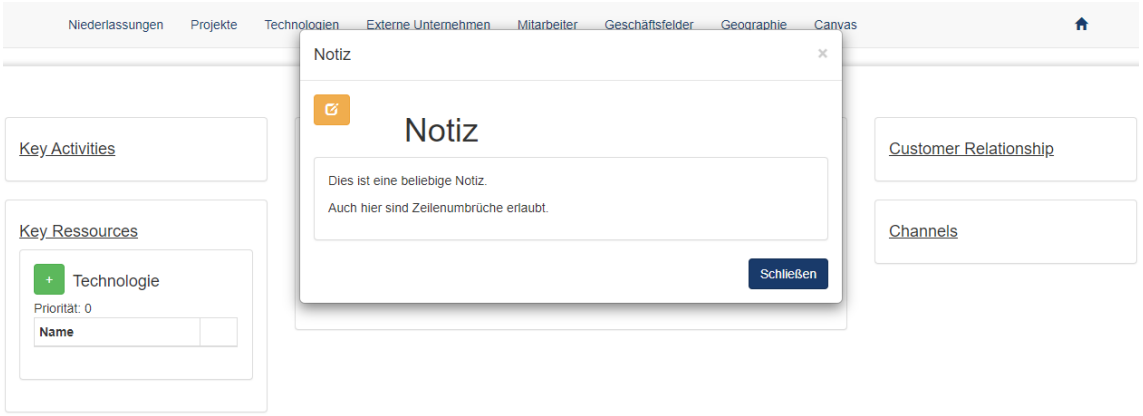


Abbildung 52: Notiz Ansicht

neue Notiz erzeugt, bzw. die alte Notiz zum Bearbeiten angezeigt und kann analog zu der Bearbeiten Ansicht der Freitexte bearbeitet werden. Mit einem Klick auf "Schließen" wird das Pop-up geschlossen und der Canvas aktualisiert.

Einträge hinzufügen

Für jedes Element aus der Ontologie lassen sich im Canvas Auswahlen tätigen. Über einen Klick auf das Hinzufügen Symbol "+"öffnet sich ein Pop-up mit einer Liste aller zur Auswahl stehenden Einträge (siehe Abbildung 53). Ein weiterer Klick auf den "+" Button in der Liste fügt den jeweiligen Eintrag der Auswahl hinzu.

Ausgewählte Einträge werden in der Tabelle unten in dem jeweiligen Element angezeigt. Abbildung 54 zeigt die Auswahl des Projekts mit dem Namen Alpha. Bereits ausgewählte Einträge lassen sich nicht noch einmal auswählen. Bei dieser Auswahl wird

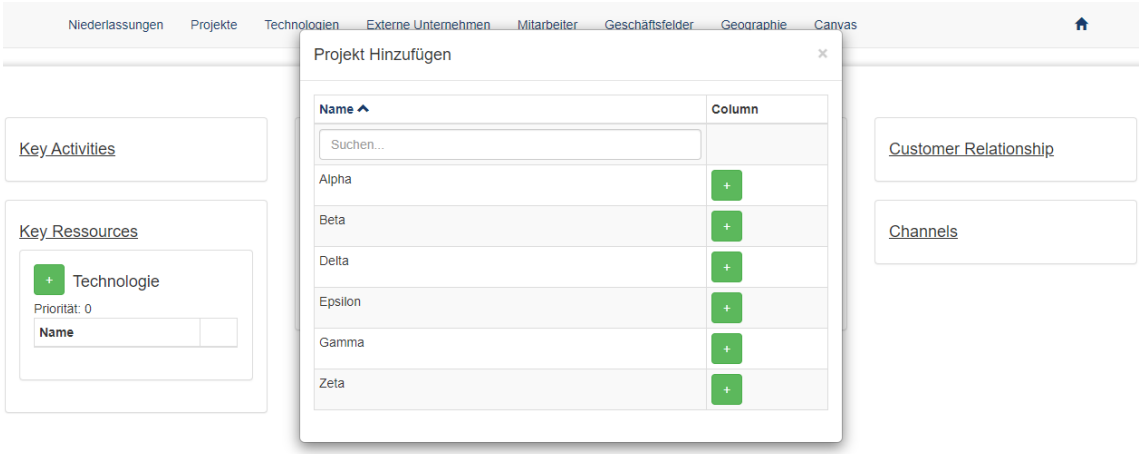


Abbildung 53: Auswahlmenü am Beispiel Projekt

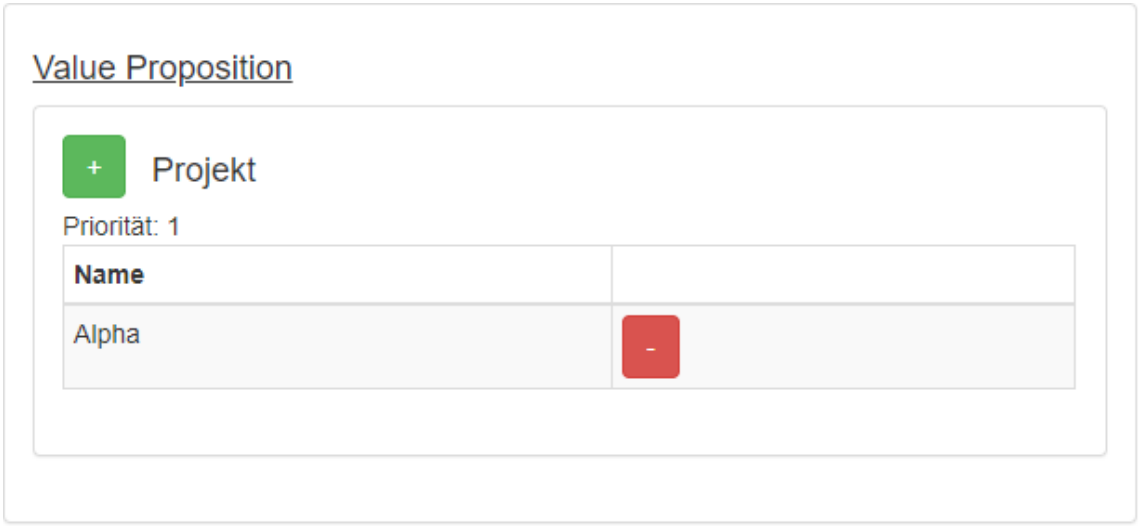


Abbildung 54: Anzeige der Auswahlen im Canvas

eine Priorität vergeben, deren Bedeutung in Kapitel 3.2.4 beschrieben wird.

Elemente löschen

Ausgewählten Einträge können über den Button "-" in Abbildung 54 aus der Auswahl gelöscht werden.

Reset-Button

Eine weitere Möglichkeit den Canvas zu löschen bietet der "Reset" Button. In dem Fall wird der Canvas auf einen leeren Canvas zurück gesetzt. D.h., alle Prioritäten werden auf 0 gesetzt und alle Auswahlen, Freitexte und Notizen gelöscht.

Zurück-Button

Mit dem Button in Abbildung 46, der links unten mit der Überschrift "Zurück" zu sehen ist, wird die Canvas Übersicht aufgerufen.

3.2.4 Die Beeinflussungen im Canvas

Eine Auswahl im Canvas beeinflusst die bisherigen Auswahlen und die Liste der auswählbaren Einträge der anderen Elemente im Canvas, basierend auf einer Prioritätenvergabe.

Ohne getätigte Auswahl bekommt jedes Element den Prioritätswert 0 zugeteilt. Dies ist die niedrigste Priorität bei der das jeweilige Element von jedem anderen Element mit höherer Priorität beeinflusst wird.

Abgesehen von dem Prioritätswert 0 ist ein kleinerer Prioritätswert als höhere Priorität zu interpretieren. Durch weitere Auswahlen werden die Prioritäten nach Reihenfolge der getätigten Auswahlen vergeben, sodass keine zwei Elemente die gleiche Priorität zugewiesen bekommen können, mit Ausnahme von dem Prioritätswert 0. Jeder ausgewählte Eintrag eines Elements nimmt dann auf alle anderen Elemente mit höheren Prioritätswerten und auf die Elemente mit Priorität 0 Einfluss.

In der DSL wurde im Vorhinein festgelegt welches Element durch ein anderes erreichbar ist. Alle von einem Element erreichbaren Elemente werden von dem jeweiligen Element beeinflusst. An dem Beispielcanvas sind alle Elemente untereinander erreichbar. Beispielsweise sind Endkunden eine Liste an Projekten zugewiesen. Projekte beinhalten Teilprojekte zu denen Technologien zugewiesen werden können. So lassen sich in diesem Beispiel alle Elemente gegenseitig erreichen.

Abbildung 55 zeigt die jeweiligen, beispielhaft befüllten Auswahllisten. Durch die farbi-

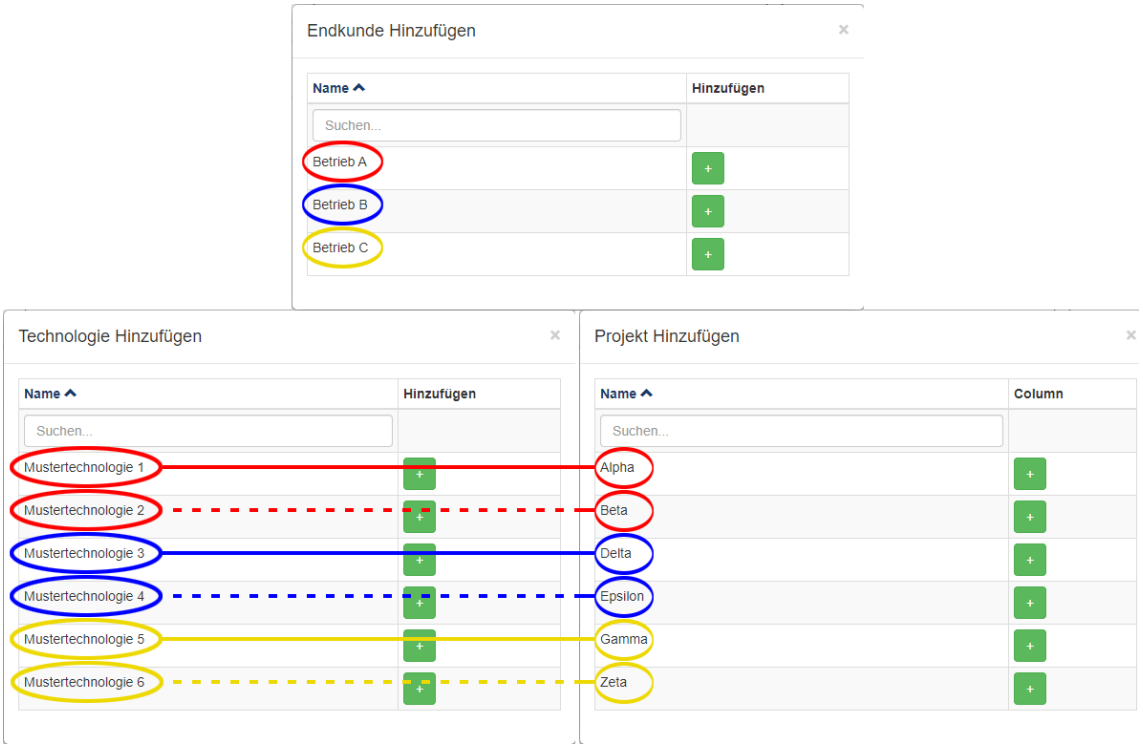


Abbildung 55: Zusammenhang der Beispieldaten

gen Markierungen werden die Verknüpfungen in der Ontologie grafisch dargestellt, welche hier bidirektional zu interpretieren sind. Die Kreise markieren die Zugehörigkeit, basierend auf den Endkunden. So ist bspw. "Betrieb A" mit Projekt "Alpha" und "Beta" verknüpft. Die Linien markieren die Zugehörigkeit von Technologien zu den Projekten. Bspw. besitzt Projekt "Alpha" ein Teilprojekt, dass "Mustertechnologie 1" zugewiesen

bekommen hat.

Für eine Beeinflussung wird diese Erreichbarkeit für jeden ausgewählten Eintrag des jeweiligen Elements überprüft. Alle nicht erreichbaren Einträge werden bei der Beeinflussung dann aus den jeweiligen Listen der beeinflussten Elemente gelöscht. Beginnend mit einem leeren Canvas würde die Erstauswahl von Endkunde "Betrieb A" bspw. dazu führen, dass in den weiteren Schritten bei Projekten nur noch Projekt "Alpha" und "Beta" sowie bei den Technologien nur noch "Mustertechnologie 1" sowie "Mustertechnologie 2" auswählbar sind. Endkunde bekommt dabei den Prioritätswert 1 zugewiesen. Ein weiterer Endkunde würde die Auswahlmöglichkeit von Projekten und Technologien um die jeweils verknüpften Einträge erweitern. Wird nun bspw. Projekt "Alpha" als weitere Auswahl hinzugefügt, bekommt Projekt den Prioritätswert 2 zugeteilt und verringert die Auswahl für Technologien auf "Mustertechnologie 1". Würde nun ein weiterer Endkunde hinzugefügt werden, würde dieser nur die Auswahlmöglichkeiten für die Projekte erweitern, für Technologien würde weiterhin nur "Mustertechnologie 1" zur Auswahl stehen.

Wird ein Eintrag manuell aus der Auswahl gelöscht, so wird auch die Beeinflussung aktualisiert, was sich unter Umständen auf die bereits getätigten Auswahlen auswirkt. Getätigte Auswahlen werden dann gelöscht, wenn ein Eintrag aus einem Element mit höherer Priorität gelöscht wird und die übrige Auswahl den entsprechenden Eintrag nicht verknüpft. Vorausgesetzt es wird nicht der letzte Endkunde aus der Auswahl entfernt sorgt bspw. ein Entfernen von dem Endkunden "Betrieb A" dazu, dass die Projekte "Alpha" und "Beta" sowie die Technologien "Mustertechnologie 1" sowie "Mustertechnologie 2" aus den Auswahllisten und getätigten Auswahlen gelöscht werden. Eine Ausnahme dazu bildet das Löschen des letzten Eintrags aus der Auswahl. In diesem Fall werden die getätigten Auswahlen der anderen Elemente nicht gelöscht, sondern die Prioritätswerte aller Elemente angepasst. Das Element mit der nun leeren Auswahl bekommt die Priorität 0. Die anderen Elemente rücken in ihrer Priorität auf, sodass die Ordnung untereinander eingehalten wird. Dabei wird auch die Beeinflussungen aktualisiert. D.h., das Element, aus dem soeben der letzte Eintrag aus der Auswahl gelöscht wurde, hat nun die geringste Priorität und so eine verringerte Liste möglicher Auswahlen. Vorausgesetzt Endkunde hat den Prioritätswert 1, Projekt den Prioritätswert 2 und Technologie den Prioritätswert 3. Dann würde bspw. das Löschen von "Betrieb A" als letzte Auswahl der Endkunden dazu führen, dass keine weitere Auswahl in Projekten und Technologien gelöscht wird. Stattdessen bekommt Endkunde den Prioritätswert 0, Projekt den Prioritätswert 1 und Technologie den Prioritätswert 2. Für eine erneute Auswahl steht nun nur noch "Betrieb A" in den Endkunden zur Verfügung, da vorweg nur Elemente ausgewählt werden konnten, die mit diesem Eintrag in Verbindung stehen. Eine weitere Auswahl von Projekten und Technologien würde in diesem Fall die Auswahlmöglichkeiten wieder erweitern.

4 Canvas DSL

Im folgenden Abschnitt wird die DSL erläutert, die zur Erstellung eines Canvas genutzt wird. Eine Gesamtübersicht über die Oberfläche ist in Abbildung 56 gegeben. In den nächsten Abschnitten wird ausgehend von dieser Übersicht, die Schrittweise Erstellung eines Canvas erläutert.

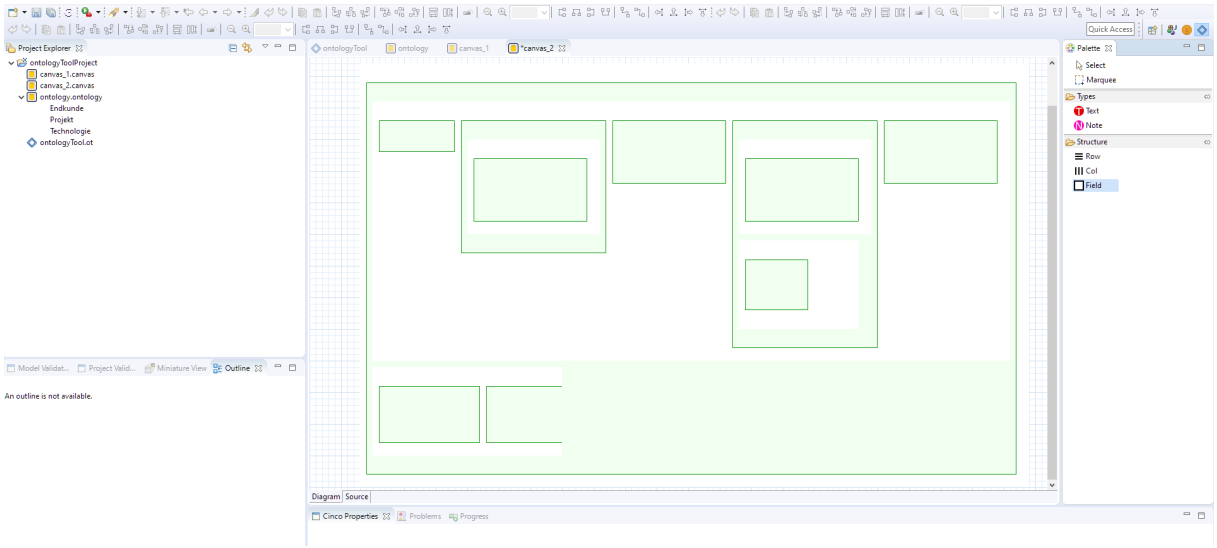


Abbildung 56: Die GUI der Gesamtübersicht über einen Canvas

4.1 Canvas anlegen und Übersicht

Das Anlegen eines neuen Canvas geschieht innerhalb des Project Explorers, dargestellt in Abbildung 57, in dem auch das OnotologyTool, sowie die Ontologie angelegt werden. Mit einem Rechtsklick auf das angelegte Projekt, lässt sich unter **New; New Canvas**, ein neuer Canvas für das ausgewählte Projekt anlegen. Erkennbar ist ein erstellter Canvas an der Dateiendung *.canvas*. Beim Öffnen eines angelegten Canvas, gelangt man in die in Abbildung 56 dargestellte Ansicht. Auf der linken Seite des Bildschirm bleibt der Project Explorer zu erkennen. Das mittlere Fenster, das im oberen Reiter den Namen des Canvas trägt, zeigt die Oberfläche, in welcher der Canvas erstellt wird. Die rechte Seite zeigt Werkzeuge, um den Canvas zu gestalten.

Der Grundaufbau eines jeden Canvas besteht aus den drei Elementen:

- Row
- Col
- Field

Row bezieht sich auf die im Canvas vorhandenen Zeilen, Col auf die vorhandenen Spalten und Field auf die Felder, die sich in den Zeilen und Spalten befinden. In dem unter Abbildung 56 aufgeführten Beispiel, ist die Anordnung der Zeilen und Spalten an den Business Mode Canvas nach Osterwalder angelehnt. Diese Anordnung entspricht der Standardanordnung, die beim Erstellen eines neuen Canvas vorzufinden ist. Felder sind nach dem Erstelle eines Canvas noch nicht vorhanden. Diese müssen noch aktiv hinzugefügt werden.

4.1.1 Palette

In der rechten Seite der Benutzeroberfläche, ist die in Abbildung 57 abgebildete Palette zu finden. Sollte diese nicht direkt ersichtlich sein, gibt es am oberen rechten Bildrand den so genannten *Quick Access*, der ebenfalls in Abbildung 57 abgebildet ist. Das Eintippen

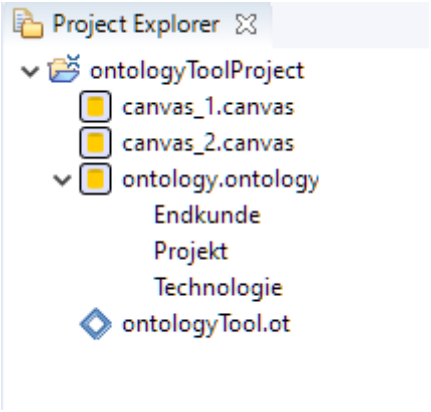


Abbildung 57: Der Project Explorer der DSL, mit Beispielprojekt, bestehend aus einem OntologyTool, einer Ontology und 2 Canvases

des Namens einer Ansicht, macht es möglich, diese auszuwählen und per Klick auf die Gewünschte zu öffnen. Die Palette lässt sich also durch die Eingabe des Begriffes *Palette* erreichen. Das Menü der Palette dient zum Hinzufügen der strukturellen Elemente eines Canvas. Sollen zu dem zu Beginn bestehenden Konstrukt aus Rows und Cols weitere hinzugefügt werden, können diese auf dem Auswahlmenü der Palette der Drag and Drop an die gewünschte Stelle innerhalb des Canvas gezogen werden. Anderes als Rows und Cols, sind zu Beginn noch keine Fields in dem Canvas vorhanden. Diese können ebenfalls per Drag and Drop innerhalb der Rows and Cols platziert werden. Ebenfalls ist es so möglich, bereits bestehende Elemente innerhalb der Oberfläche zu verschieben. Die darüber hinaus in der Palette vorhandenen Auswahlelemente *Text* und *Note*, werden in Kapitel 4.1.3 noch einmal näher beschrieben.

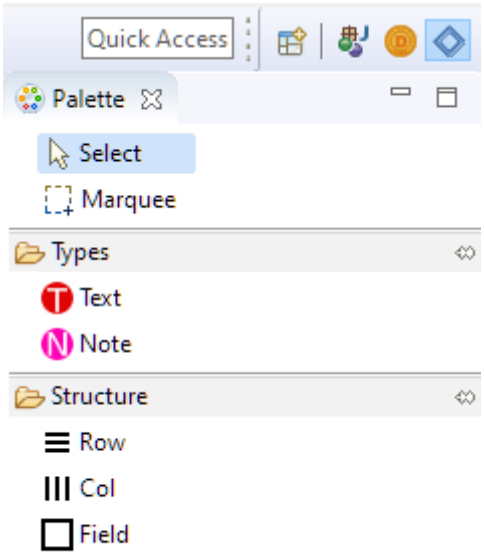


Abbildung 58: Das Auswahlmenü der DSl, zum Hinzufügen von Freitexten, Notizen, sowie weiterer Rows, Cols und Fields

4.1.2 Import von Ontologytypen

Sollen Typen aus einer Ontologie in einen Canvas eingebracht werden, so müssen sowohl die Ontologie, als auch der Canvas im gleichen Projekt liegen. In Abbildung 57 ist dies dadurch gegeben, dass die Ontologie *ontology.ontology*, sowie die beiden Canvas *canvas_ 1.canvas* und *canvas_ 2.canvas* in dem Projekt *ontologyToolProject* vorhanden sind. Mit dem schwarzen Pfeil neben der Ontology, lässt dich diese Aufklappen und eine Übersicht über die in der Ontology vorhandenen Typen einsehen. Diese Typen können in einen der geöffneten Canvas per Drag and Drop übertragen werden. Dafür benötigt der Canvas mindestens ein Field, da Typen nur innerhalb konkreter Fields eingefügt werden können.

4.1.3 Freitexte und Notizen

Zusätzlich zu den bereits bekannten Elementen zum Hinzufügen von Reihen, Spalten und Feldern, besitzt die Palette noch die Elemente *Text*, sowie *Note*. Diese können, vergleichbar mit den Typen aus einer vorhandenen Ontologie, ebenfalls in im Canvas vorhandene Felder eingefügt werden. Das geschieht, wie aus anderen Elementen bekannt, per Drag and Drop.

- Text: Ein Freitext ist durch einen roten Kreis mit weißem *T* gekennzeichnet. Der Inhalt eines Freitextes, kann über die zugehörigen *Cinco Properties*, unter dem Punkt *name*, bestimmt werden. Wird aus der DSL heraus die zu dem Canvas zugehörige Webapp generiert, so verhält sich ein vorhandener Freitext in einem Feld ähnlich zu einem Typ der Ontologie. Der Freitext erscheint also mittig innerhalb des Feldes. Der Unterschied zu einen Typ der Ontologie besteht darin, dass Freitexte nicht mit Pathwerten belegt werden können, diese können also weder Start- noch der Endpunkt eines Path sein.

- Note: Eine Notiz ist durch einen pinken Kreis mit weißem *N* gekennzeichnet. Der Inhalt einer Notiz, kann über die zugehörigen *Cinco Properties*, unter dem Punkt *name*, bestimmt werden. Notizen verhalten sich in der generierten Webapp nicht wie Typen innerhalb von Feldern. Notizen bieten die Möglichkeit, ergänzende Informationen zu einem Feld hinzuzufügen, die keinerlei Auswirkungen auf die Funktionsweise des Canvas haben. Notizen, die bereits in der DSL in den Canvas eingebracht werden, bleiben in der generierten Webapp an den entsprechenden Feldern bestehen. Auch können dort weitere Notizen hinzugefügt, oder bestehende Notizen aus dem Canvas entfernt werden.

In Abbildung 59 ist ein Feld dargestellt, in dem sich sowohl ein Typ aus einer Ontologie befindet, als auch jeweils ein Freitext und eine Notiz.

4.1.4 Cinco Properties

Zu jedem angewählten Objekt innerhalb des Canvas, gibt es im unteren Bereich des Oberfläche die *Cinco Properties*. Alternativ sind diese über den Quick Access zu erreichen. In Abbildung 60, sind diese am Beispiel für ein ausgewähltes Feld dargestellt. Für ein angewähltes Objekt, sind hier diverse Einstellungen vorzunehmen, die im Folgenden einzeln aufgeführt sind:

- modelName: Bezeichnet den Namen des Canvas, dieser wird initial beim Erstellen eines neuen Canvas im OntologyToolProjekt festgelegt und kann danach hier geändert werden.

Abbildung 59: Ein Feld in einem Canvas, in dem sich neben dem Typ Technologie aus der Ontologie, auch ein Freitext und eine Notiz befinden

- myColor: Ermöglicht das Wählen einer Farbe, in der Felder oder in Feldern befindliche Typen aus einer Ontologie dargestellt werden.
- fieldName: Bezeichnet den Namen eines Feldes.
- transparancyValue: Bezeichnet die Intensität der Farbe, in der ein Feld oder ein Type einer Ontologie dargestellt wird. Lässt Werte zwischen 0.0 und 1.0 zu. Ein Wert von 0.0 entspricht dabei einer vollständigen Transparenz. Ein Feld, dass mit diesem Transparenzwert belegt wird, ist demnach nicht mehr sichtbar. Der Wert von 1.0 sorgt für gar keine Transparenz der Farbe, das Feld und die Farbe sind in voller Deckkraft im Canvas abgebildet.
- path: Bezeichnet den Pfad, durch den die Beeinflussung verschiedener Ontologie Typen in verschiedenen Feldern des Canvas geschehen soll.

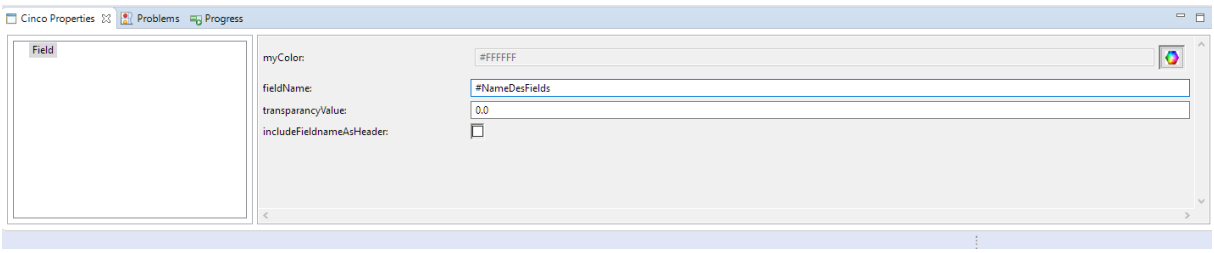


Abbildung 60: Cinco Porperties am Beispiel eines ausgewählten Feldes mit definiertem Namen, der Farbe Weiß und dem Wert der Transparenz der Farbe von 0.0

4.2 Path

Mit Hilfe eines Path, können Typen aus unterschiedlichen Feldern miteinander verbunden werden. Das Ziel dahinter ist es, dass in der später generierten Webapp eine Beeinflussung der in den Feldern zur Verfügung stehenden Werte stattfindet, ausgehend von den Attributen der Typen, die aus der Ontologie in das entsprechende Feld des Canvas eingesetzt werden. Eine weiterführende Erklärung dieser Art der Beeinflussung, wie sie in der Webapp vorzufinden ist, ist bereits in Kapitel 3.2.4 gegeben.

4.2.1 Path erstellen

In Abbildung 61 ist ein Path von *Endkunde* zu *Projekt* gezogen. Um einen solchen Path zu erstellen, die Maus über den Type halten, von wo aus der Path seinen Ursprung haben soll. In dem abgebildeten Beispiel ist dies *Endkunde*. Daraufhin erscheint rechts neben dem Cursor ein schwarzer Pfeil. Das Klicken auf diesen Pfeil und das Ziehen der gedrückten linken Maustaste auf den Type, an dem der Path enden soll, erstellt diesen. In diesem Beispiel wird der Pfeil auf *Projekt* gezogen.

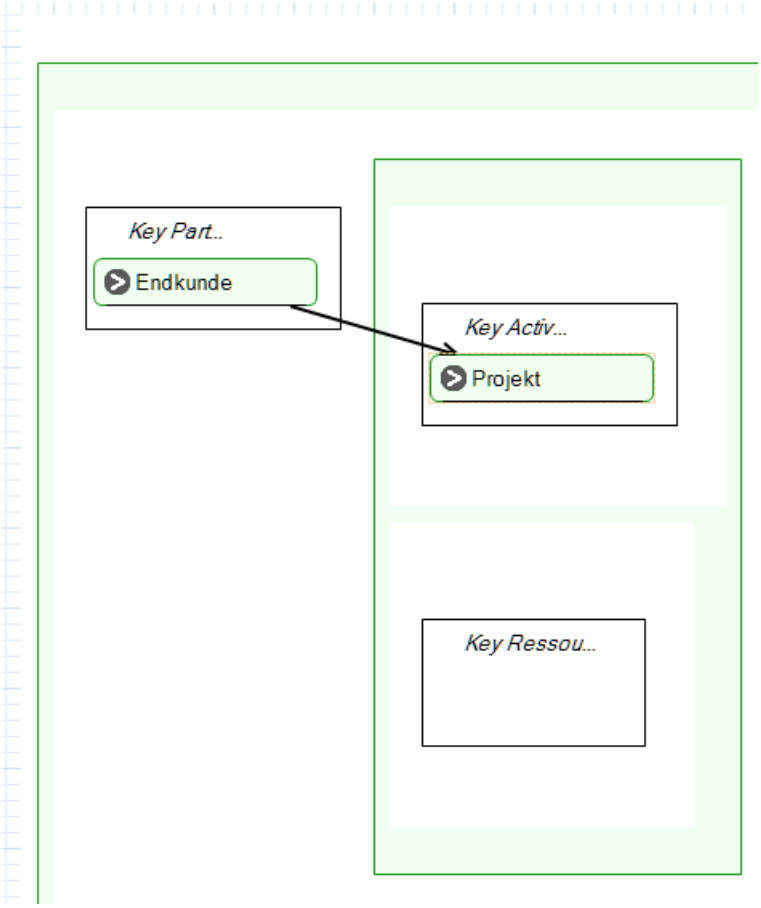


Abbildung 61

4.2.2 Path beschreiben

In den Feldern des Canvas können per Definition nur Typen aus der Ontologie untergebracht werden. Wird ein Pfad zwischen zwei dieser Typen gezogen, muss zusätzlich

sichergestellt werden, welche Attribute der Typen bekannt sein müssen, um eine Verbindung zwischen den beiden Typen herzustellen. Diese Verbindung muss von Hand, über die Eingabe eines Pathwertes erfolgen. Ein solcher Wert ist in Abbildung 62 dargestellt. Das zugrunde liegende Beispiel der dazugehörigen Ontologie, ist in Abbildung 63 abgebildet. Dort besitzt der Typ *Projekt*, das Attribut *endkunde*. Auf der anderen Seite besitzt der Typ *Endkunde* das Attribut *projekt*. Genau dieses Verhältnis wird durch den Path dargestellt. Bildlich gesprochen, wird der Weg beschrieben, der durch die Assoziationen durchlaufen wird, um von einem Typen zu dem anderen zu gelangen, die jeweils durch den gezogenen Path miteinander verbunden sind. Befinden sich innerhalb der Ontologie Typen zwischen dem Start- und dem Zieltypen, so kann der Pathwert nach dem gleichen Schema beliebig erweitert werden:.{{*Startattribut*}}.{{*Zwischenattribut*1}}.{{*Zwischenattribut*2}}.{{*Endattribut*}}. usw., die Anzahl der Zwischenattribute innerhalb des Pathwertes, ist dabei beliebig zu erweitern und hängt nur von dem Aufbau der verwendeten Ontologie ab.

Das Erstellen eines neuen Pathwertes, geschieht über einen Rechtsklick in das Feld *path*, welches auf der rechten Seite der Abbildung 62 dargestellt ist. Nach einem Klick auf *New Path*, kann in das neu erscheinende Fenster der Pathwert eingetragen und bestätigt werden. Soll ein Wert einmal zu einem späteren Zeitpunkt verändert werden, ist dies möglich, ein Rechtsklick auf einen Wert, lässt diesen unter *Edit...* bearbeiten.

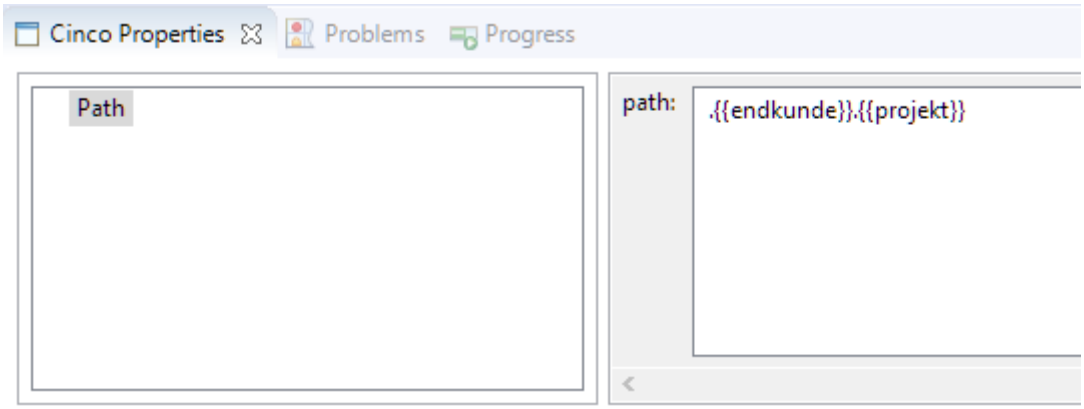


Abbildung 62: Pfadangabe, um eine Beeinflussung der Projekte, ausgehend von den Endkunden stattfinden zu lassen. Die Projekte sind innerhalb des Typen Endkunde durch das Attribut projekt gekennzeichnet

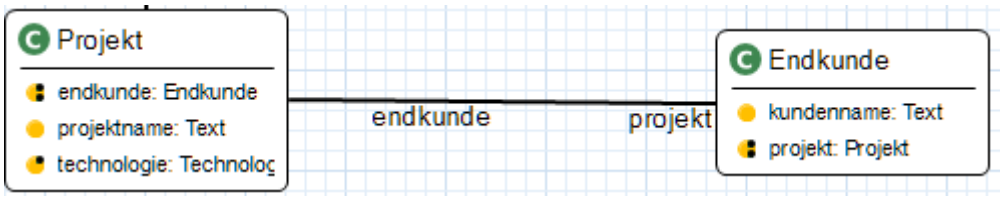


Abbildung 63: Ein Ausschnitt aus einer Ontologie, der die Typen Projekt und Endkunde beinhaltet

4.3 Validierung eines Canvas

Die Validierung eines ausgefüllten Canvas dient dazu, dem Nutzer Informationen darüber zu geben, ob der Canvas korrekt ausgefüllt ist. Dazu stehen im unteren linken Bereich des

Fensters 2 Bereiche zur Verfügung. Die *Model Validation*, sowie die *Project Validation*. Die Model Validation gibt nur Aufschluss über das aktuell geöffnete Modell. Also entweder den aktuellen Canvas, oder auch die Ontologie. Mit der Ansicht der Project Validation, lässt sich eine Übersicht über die Ergebnisse aller Checks des Projektes abrufen. Die Checks laufen ständig im Hintergrund, während der Canvas erstellt, bzw. ausgefüllt wird. Somit besteht ein ständiger Überblick darüber, auf die aktuelle Version eines Canvas bereits in eine funktionierende Webapp überführt werden kann.

Um einen besseren Überblick über das Ergebnis der durchgeführten Checks zu erhalten, werden die Meldungen in drei verschiedene Kategorien unterteilt:

- Check: Die durch diesen Check abgedeckte Funktion des Canvas wurde korrekt ausgeführt, hier braucht nichts weiter beachtet zu werden.

- Warnung: Warnungen weisen darauf hin, dass für eine vollständige Funktionalität des Canvas noch weitere Einstellungen vorgenommen werden können. Allerdings müssen diese nicht zwangsläufig in einen erfolgreich verlaufenden Check umgewandelt werden. Auch wenn ein Canvas Warnungen enthält, kann dieser in eine funktionierende Webapp umgewandelt werden. Ein Beispiel für einen solchen Fall ist ein Feld, in dem Typ der Ontologie vorhanden ist. Bleibt dieses Feld beim Generieren der Webapp leer, hat es keine Auswirkungen auf die Funktionsweise, die Warnung soll hier nur verhindern, dass Felder vergessen werden mit Inhalten zu füllen.

- Fehler: Solange Fehler im Canvas vorhanden sind, sollte dieser nicht in eine Webapp umgewandelt werden. Fehler können dafür sorgen, dass die Generierung schief läuft, somit keine Webapp zu erhalten ist. Der Canvas sollte daher an den fehlerbehafteten Stellen so abgeändert werden, dass die entsprechenden Fehlerpunkte zu einer Warnung, bzw. einem erfolgreichen Test umgewandelt werden.

Die Meldungen werden wie in Abbildung 64 dargestellt durch drei verschiedene Symbole grafisch identifizierbar gestaltet. Der grüne Kreis mit weißem Haken steht für einen erfolgreich verlaufenen Check, die gelbe Raute mit schwarzem Ausrufezeichen für eine Warnung. Ein Fehler wird durch das rote Sechseck mit dem weißen Kreuz dargestellt.

Soll eine Warnung oder ein Fehler korrigiert werden, um diese entweder in eine Warnung oder einen erfolgreichen Check umzuwandeln, lassen sich die Meldungen erweitern, um mehr Informationen über sie zu erhalten. Neben jeder Warnung, sowie neben jedem angezeigten Fehler, gibt es einen nach rechts in Richtung der Meldung zeigenden Pfeil (Vgl. Abbildung 64). Ein Linksklick auf diesen Pfeil zeigt an, wo genau im Canvas dieser Meldung erzeugt wurde. Beispielsweise wird der Name eines Feldes oder eines Pathwertes angegeben, sollte der betreffende Check dort eine Warnung oder einen Fehler ausfindig machen.

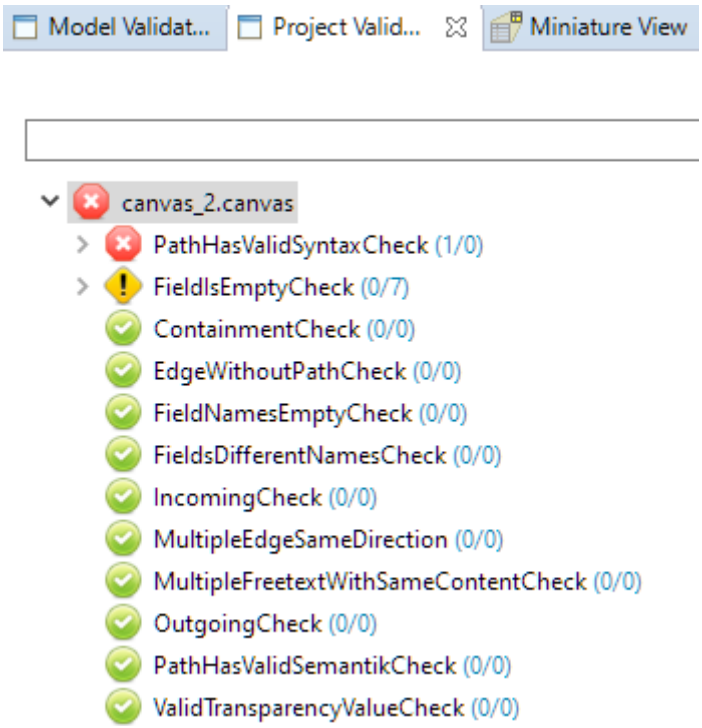


Abbildung 64: Eine Besipielhafte Validierung eines Canvas, die sowohl korrekte Checks, als auch Warnungen und Fehler Meldungen enthält

Guide/UserGuide

Abbildungsverzeichnis

1.1 Konzept: Living Business Model Canvas 2

2.1 Anforderung an das Metamodell [10] 7
2.2 Klassendiagramm Backofen 14
2.3 HTML Als Text und als Webseite 17
2.4 Von der Idee zur DSL mit MetaEdit+ 19
2.5 Marama Tools: Meta-Model (links), Shape Designer (mitte), View Definer (rechts) 20
2.6 Das HTML-Dokument aus Abbildung 2.2.1 als grafische Darstellung 22
2.7 Beispielhaftes Datenmodell mit den verschiedenen Datentypen 37
2.8 Ein Prozessmodell mit Datenflüssen und dem Prozessfluss von *start* zu *success* 38
2.9 Das Business Model Canvas 41

3.1 Die Ontologie der Firma Schulz Systemtechnik GmbH 50
3.2 Beispiel für das Erstellen-Pattern: Erstellen einer Person 53
3.3 Beispiel für das Bearbeiten-Pattern: Bearbeiten einer Person 54
3.4 Beispiel für das Löschen-Pattern: Löschen eines Teilprojektes mit dem Löschen aus der Projektliste 54
3.5 Beispiel für das Zuweisenpattern: Zuweisen eines Endkunden zu einem OEM 55
3.6 Beispiel für das Laden-Pattern: Laden der benötigten Daten für den übergeordneten Personenprozess 56
3.7 Beispiel für die Modalansicht 56
3.8 Beispiel für das Filter-Pattern: Alle Standorte, die nicht der Person zugeordnet wurden, werden zurückgegeben 58
3.9 Beispiel für den Startprozess 58
3.10 Beispiel für den Rootelementprozess Geographie 59
3.11 Das Startmenü mit der Navigationsleiste oben 60
3.12 Beispiel für eine Verwalten-GUI 61
3.13 Beispiel für eine Verwalten-GUI eines abstrakten Typs 62
3.14 Beispiel einer Baumstruktur 64

3.15 Legende für schematische Darstellungen 65
3.16 Schematische Darstellung der Baumübersicht 67
3.17 TreeViewGUI: Ruft sich selbst mit den Kindern wieder auf 68
3.18 Baumübersicht in der laufenden Web Applikation 69
3.19 Schematische Darstellung der Baumerstellung 70
3.20 Schematische Darstellung der Baumveraltung 70
3.21 Pfad-Ansicht für ein Baumobjekt 71
3.22 Verwaltungs-Ansicht für ein Baumobjekt 72
3.23 Schematische Darstellung des Löschens eines Baumknotens 73
3.24 Der GetRootNodesProcess 73
3.25 Der RemoveProcess 74
3.26 Der RemoveChildrenProcess 75
3.27 Der InvertListProcess 76
3.28 Schematische Darstellung des gesamten Prozessablaufs 77
3.29 Helferobjekt im Datenkontext 78
3.30 Schematische Darstellung des Startprozesses 79
3.31 Schematische Darstellung der Helfererstellung 79
3.32 Schematische Darstellung der Auswahl-GUI 80
3.33 Schematische Darstellung der Helferinitialisierung 80
3.34 Schematische Darstellung der Sortierung 81
3.35 Schematische Darstellung der Baumelemententfernung 82
3.36 Der „CreateHelper“-Prozess 82
3.37 Der „RemoveTree“-Prozess 83
3.38 Der „SortSiblings“-Prozess 84
3.39 Die Auswahl-GUI 85
3.40 Erzeugung einer DIME-Applikation mithilfe von Factories 87
3.41 Vererbungshierarchie der Modellgeneratoren 88
3.42 Phasen der Transformation 92
3.43 Prozess-Modell (links) und GUI-Modell (rechts) nach der Initialisierung . . 93

4.1 Die Ontologie der Firma Schulz Systemtechnik GmbH mit der Ontologie-DSL 102
4.2 Ein OntologyTool-Modell 102
4.3 Durch den CrudProcessGenerator generierter Prozess für einen Typ *Person* 107
4.4 Verwaltungs-GUI für den Typ *Person*. Dieser besitzt das primitive Attribut *name* und ein komplexes Attribut *Adresse* 108
4.5 Beispielhaftes Datenmodell mit komplexen und primitiven Attributen 111
4.6 Die aus dem Datenmodell in Abbildung 4.5 für den Typ *Professor* generierte Detailansicht 112

4.7 Eine Beispielgenerierung des EditViewGenerators anhand des Typs "Externe Unternehmen" der Schulz-Ontologie 115

5.1 Canvas Prototyp mit vollem Funktionsumfang 122
5.2 Canvas Aktualisieren Pattern (Endkunde) ANMERKUNG: Die Verbindungen im Bild werden noch bearbeitet 124
5.3 Canvas Vereinigung zweier Listen (Endkunde) ANMERKUNG: Die Verbindungen im Bild werden noch bearbeitet 124
5.4 Canvas Schnittmenge zweier Listen (Endkunde) ANMERKUNG: Die Verbindungen im Bild werden noch bearbeitet 125
5.5 Canvas Modalansicht Hinzufügen (Endkunde) ANMERKUNG: Die Verbindungen im Bild werden noch bearbeitet 125
5.6 Canvas Dublikate entfernen (Endkunde) ANMERKUNG: Die Verbindungen im Bild werden noch bearbeitet 126
5.7 Canvas Suche verbundene Projekte zu dem Endkunden 127
5.8 Canvas Suche alle verbundenen Elemente zu dem Endkunden 128
5.9 Canvas Management Prozess (Endkunde) 129
5.10 Canvas Beeinflussung nach dem Hinzufügen eines Endkunden 129
5.11 Canvas *XX_ Sub_ Interact* Prioritätenvergabe (Endkunde) 130
5.12 Canvas *XX_ Sub_ Interact* Teil 2 (Endkunde) 131
5.13 Canvas Prioritäten Korrektur Teil 1 ANMERKUNG: Die Verbindungen im Bild werden noch bearbeitet 132
5.14 Canvas Prioritäten Korrektur Teil 2 ANMERKUNG: Die Verbindungen im Bild werden noch bearbeitet 133
5.15 Canvas Reset ANMERKUNG: Die Verbindungen im Bild werden noch bearbeitet 134
5.16 Canvas Suche maximale Priorität 135
5.17 Canvas Modalansicht Freitext ANMERKUNG: Die Verbindungen im Bild werden noch bearbeitet 135
5.18 Canvas löschen 136
5.19 Canvas erzeugen ANMERKUNG: Die Verbindungen im Bild werden noch bearbeitet 137
5.20 Canvas Vorauswahl 138
5.21 Canvas Hauptprozess 139
5.22 Canvas Updateprozess erster Teil 140
5.23 Canvas Updateprozess: Ausschnitt Fall Priorität = 1 141
5.24 Canvas Updateprozess zweiter Teil 142
5.25 Canvas Updateprozess: Ausschnitt Fall Priorität = 2 143

5.26 Canvas GUI ANMERKUNG: Bild wird noch ausgetauscht gegen eines ausgetauscht mit befüllten Daten 144
5.27 Freitext und Notiz Buttons 144
5.28 Modalansicht Element Hinzufügen 145
5.29 Modalansicht Freitext bearbeiten 145
5.30 Modalansicht Freitext erstellen 146
5.31 Modalansicht Notiz Ansicht 146
5.32 Farbe der Notizbuttons 146

6.1 Beispielcanvas 152
6.2 Beispielcanvas: Unten wird der Pfad mit dem Inhalt `.{{projekt}}.{{teilprojekt}}.{{technologie}}` zwischen „Projekt“ und „Technologie“ angezeigt. 153
6.3 Beispielcanvas-Objekt in dem Data-Modell 154
6.4 GUI des Beispielcanvas als DIME-GUI 160
6.5 Feld „Customer Segment“ des Beispielcanvas als Beispiel für das Field-Pattern 161
6.6 Aktualisieren der Auswahlliste anderer Contents bei Priorität zwei 165
6.7 Schreiben der Verfügbarliste, wenn die Hilfsvariablen gesetzt sind 166
6.8 Suche nach dem Teilprojekt über ein Projekt 166

Literaturverzeichnis

[1] *CINCO Wiki.*
`http://gitlab.com/scce/cinco/wikis/home`.
Zuletzt zugegriffen: 12.09.2019.

[2] *Firmenprofil der Schulz Systemtechnik GmbH.*
`https://www.schulz.st/page/schulz-systemtechnik`.
Zuletzt zugegriffen: 02.03.2020.

[3] *Glidr Website.* `https://www.glidr.io/`.
Zuletzt zugegriffen: 24.09.2020.

[4] *HTML Bausteine – HTML lernen mit Blockly.* `http://htmlbausteine.zgtm.de/`.
Zuletzt zugegriffen: 02.03.2020.

[5] *MetaCase Website.* `https://www.metacase.com/`.
Zuletzt zugegriffen: 02.03.2020.

[6] *Strategyzer Website.* `https://www.strategyzer.com/`.
Zuletzt zugegriffen: 24.09.2020.

[7] *Xtend Dokumentation.*
`https://www.https://www.eclipse.org/xtend/documentation/index.html`.
Zuletzt zugegriffen: 27.02.2020.

[8] *Xtend Website.* `https://www.eclipse.org/xtend/index.html`.
Zuletzt zugegriffen: 27.02.2020.

[9] Bosselmann, Steve, Markus Frohme, Dawid Kopetzki, Michael Lybecait, Stefan Naujokat, Johannes Neubauer, Dominic Wirkner, Philip Zweihoff und Bernhard Steffen: *DIME: A Programming-Less Modeling Environment for Web Applications.* In: Margaria, Tiziana und Bernhard Steffen (Herausgeber): *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, Seiten 809–832, Cham, 2016. Springer International Publishing.

[10] Brambilla, Marco, Jordi Cabot und Manuel Wimmer: *Model-Driven Software Engineering in Practice: Second Edition*. Morgan & Claypool Publishers, 2017.

[11] Ching, Hong: *CRITICISMS, VARIATIONS AND EXPERIENCES WITH BUSINESS MODEL CANVAS*. 06 2014.

[12] Coes, D. H.: *Critically assessing the strengths and limitations of the Business Model Canvas*. 2014.

[13] Corcho, Oscar, Mariano Fernández-López und Asunción Gómez-Pérez: *Methodologies, tools and languages for building ontologies. Where is their meeting point?* Data & Knowledge Engineering, 46(1):41–64, 2003.

[14] Fowler, Martin: *Domain Specific Languages*. Addison-Wesley Professional, 1st Auflage, 2010.

[15] Gruber, Thomas R.: *Toward principles for the design of ontologies used for knowledge sharing?* International Journal of Human-Computer Studies, 43(5-6):907–928, 1995.

[16] Grundy, John, John Hosking, Jun Huh und Karen Li: *Marama: An eclipse meta-toolset for generating multi-view environments*. Seiten 819–822, 2008.

[17] Kai-Ingo Voigt, Gerhard Schewe: *Projekt*. Gabler Wirtschaftslexikon, 2018.

[18] Kecher, Christoph: *UML 2 das umfassende Handbuch ; [inkl. CD-ROM mit UML-Tools und DIN-A2-Poster mit allen Diagrammen]*. Galileo Press, Bonn, 2011.

[19] Margaria, Tiziana und Bernhard Steffen: *Service-Orientation: Conquering Complexity with XMDD*, Seiten 217–236. Springer London, London, 2012.

[20] McAffer, Jeff, Jean-Michel Lemieux und Chris Aniszczyk: *Eclipse rich client platform*. Addison-Wesley Professional, 2010.

[21] *CINCO Meta-Plugins*.
`https://gitlab.com/scce/cinco/wikis/Meta-plugins`.
Zuletzt zugegriffen: 12.09.2019.

[22] Naujokat, S.: *Cinco User?s Manual*, 2016.

[23] Naujokat, S., M. Lybecait, D. Kopetzki und B. Steffen: *CINCO: A Simplicity-Driven Approach to Full Generation of Domain-Specic Graphical Modeling Tools*, 2017.

[24] Naujokat, Stefan, Michael Lybecait, Dawid Kopetzki und Bernhard Steffen: *CINCO: a simplicity-driven approach to full generation of domain-specific graphical modeling tools*. International Journal on Software Tools for Technology Transfer, 20(3):327–354, 2018.

[25] Neubauer, Johannes, Markus Frohme, Bernhard Steffen und Tiziana Margaria: *Prototype-Driven Development of Web Applications with DyWA*. In: Margaria, Tiziana und Bernhard Steffen (Herausgeber): *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, Seiten 56–72, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[26] Osterwalder, A., Y. Pigneur und T. Clark: *Business Model Generation: A Handbook for Visionaries, Game Changers, and Challengers*. Strategyzer series. Wiley, 2010.

[27] Osterwalder, Alexander und Yves Pigneur: *Business model generation: a handbook for visionaries, game changers, and challengers*. John Wiley & Sons, 2010.

[28] *CINCO Meta-Graph-Language*.
`https://gitlab.com/scce/cinco/wikis/Meta-Graph-Language`.
Zuletzt zugegriffen: 12.09.2019.

[29] Smolander, Kari, Kalle Lyytinen, Veli-Pekka Tahvanainen und Pentt Marttiin: *MetaEdit— A flexible graphical environment for methodology modelling*. In: Andersen, Rudolf, Janis A. Bubenko und Arne Sølvberg (Herausgeber): *Advanced Information Systems Engineerin*, Seiten 168–193, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[30] Stefan Münz, Wolfgang Nefzger: *HTML Handbuch*. Franzis-Verlag, 2005.

[31] Steffen, B. und S. Naujokat: *Archimedean Points: The Essence for Mastering Change*, 2016.

[32] Steffen, Bernhard, Tiziana Margaria, Ralf Nagel, Sven Jörges und Christian Kubczak: *Model-Driven Development with the jABC*. In: Bin, Eyal, Avi Ziv und Shmuel Ur (Herausgeber): *Hardware and Software, Verification and Testing*, Seiten 92–108, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[33] Tiziana, Margaria und Bernhard Steffen: *Chapter I Business Process Modelling in the jABC: The One-Thing-Approach*. Handbook of Research on Business Process Modeling, Seiten 1–26, 2009.

[34] Uschold, Mike und Michael Gruninger: *Ontologies: principles, methods and applications*. The Knowledge Engineering Review, 11(2):93–136, 1996.

[35] Voelter, Markus: *DSL Engineering*. 2013.

[36] Voelter, Markus, Bernd Kolb, Klaus Birken, Federico Tomassetti, Patrick Alff, Laurent Wiart, Andreas Wortmann und Arne Nordmann:

Using language workbenches and domain-specific languages for safety-critical software development. Software & Systems Modeling, 18(4):2507–2530, 2019.

[37] Volkert: *Neu in der Geschäftsführung.* Springer, 2014.

[38] Wiendahl, Hans-Peter, Jürgen Reichardt und Peter Nyhuis: *Handbuch Fabrikplanung.* Carl Hanser Verlag GmbH & Co. KG, 2014.