

# **Entwicklung einer domänenspezifischen grafischen Sprache zur Ausführung bedingungsgesteuerter semi-automatischer Prozesse**

**Implementation of a Domain-Specific Graphical Language for the Execution  
of Condition-Driven Semi-Automated Processes**

## **Masterarbeit**

zur Erlangung des akademischen Grades Master of Science (M.Sc.)

Vorgelegt an der  
Hochschule München  
Fakultät für Informatik und Mathematik

**Autor:** Thomas Großbeck

**Studiengang:** Master Informatik (Software Engineering)

**Matrikelnummer:** 11475115

**Erstprüfer:** Prof. Dr. Thomas Kofler

**Abgabedatum:** 11.04.2024

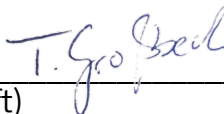
Die folgende Erklärung ist in jedes Exemplar der Masterarbeit einzubinden und jeweils persönlich zu unterschreiben.

Großbeck, Thomas (Familienname, Vorname)	München, 11.04.2024 (Ort, Datum)
12.01.1997 (Geburtsdatum)	6 / SS 20 24 (Studiengruppe / WS/SS)

### Erklärung

Gemäß § 40 Abs. 1 i. V. m. § 31 Abs. 7 RaPO

Hiermit erkläre ich, dass ich die Masterarbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

  
(Unterschrift)

# I Kurzfassung

Das Ziel der Masterarbeit ist die Entwicklung einer domänenspezifischen Sprache auf Basis eines Metamodells, das es mittels einer grafischen Repräsentation ermöglicht, Abläufe zu erfassen. Die domänenspezifische Sprache soll Domänenexperten die Möglichkeit bieten, Ablauflogiken grafisch mit Unterstützung eines Werkzeuges zu modellieren. Nach einer Einführung in die Abstraktion von Abläufen und insbesondere in das Themenfeld der domänenspezifischen Sprachen, werden die Anforderungen an das Metamodell benannt. Im Anschluss erfolgt die Entwicklung und Beschreibung des Metamodells. Darauf aufbauend wird eine Auswahl an Werkzeugen zur Abbildung des Metamodells als grafische Domain Specific Language (domänenspezifische Sprache) (DSL) evaluiert. Dabei werden Kriterien und Anforderungen an die Werkzeuge aufgestellt, um diese folgend auf ihre Vorstellung bewerten und vergleichen zu können. Nachdem das Metamodell mit Hilfe des gewählten Werkzeugs abgebildet wurde, folgt die Interpretation des Modells. Hierbei wird ein selbstgeschriebener Interpreter zur Simulation des Ablaufs angewendet, sodass die Ablauflogik ausgeführt werden kann. Abschließend werden weitere Schritte für die Entwicklung der grafischen DSL sowie sinnvolle Einsatzmöglichkeiten aufgezeigt.

## II Inhaltsverzeichnis

<b>I</b>	<b>Kurzfassung</b>	<b>I</b>
<b>II</b>	<b>Inhaltsverzeichnis</b>	<b>II</b>
<b>III</b>	<b>Abbildungsverzeichnis</b>	<b>III</b>
<b>IV</b>	<b>Tabellenverzeichnis</b>	<b>IV</b>
<b>V</b>	<b>Listing-Verzeichnis</b>	<b>IV</b>
<b>VI</b>	<b>Abkürzungsverzeichnis</b>	<b>IV</b>
<b>1</b>	<b>Einführung und Problemstellung</b>	<b>1</b>
<b>2</b>	<b>Grundlagen der Abstraktion von Abläufen</b>	<b>5</b>
2.1	Workflow und Prozess . . . . .	6
2.2	Metamodell . . . . .	7
2.3	Domain Specific Language (Domänenspezifische Sprache) . . . . .	8
<b>3</b>	<b>Anforderungen und verwandte Arbeiten</b>	<b>13</b>
3.1	Anforderungen an das Metamodell . . . . .	13
3.2	Verwandte Arbeiten zur grafischen Beschreibung von Abläufen . . . . .	14
<b>4</b>	<b>Entwicklung eines Metamodells für eine grafische DSL</b>	<b>18</b>
<b>5</b>	<b>Evaluierung von Werkzeugen zur Abbildung grafischer Modelle</b>	<b>25</b>
5.1	Kriterien zum Vergleich der Werkzeuge . . . . .	25
5.2	Analyse einer Auswahl an Werkzeugen . . . . .	28
5.2.1	MetaEdit+ . . . . .	28
5.2.2	ADOxx . . . . .	35
5.2.3	Eclipse Modeling Framework (EMF) . . . . .	39
5.2.4	Meta Programming System (MPS) . . . . .	41
5.3	Bewertung der Werkzeuge . . . . .	42
<b>6</b>	<b>Implementierung der grafischen DSL</b>	<b>50</b>
<b>7</b>	<b>Interpretation des modellierten Ablaufs</b>	<b>58</b>
<b>8</b>	<b>Fazit und Ausblick</b>	<b>67</b>
<b>9</b>	<b>Quellenverzeichnis</b>	<b>69</b>

### III Abbildungsverzeichnis

Abb. 1	Anzahl der Unternehmen, die Low- und No-Code Plattformen einsetzen [LFSS21] . . . . .	2
Abb. 2	Metamodellierung basierend auf Sprachebenen [KK02] . . . . .	15
Abb. 3	Abbildung des Metamodells auf oberster Ebene . . . . .	21
Abb. 4	Abbildung des Metamodells der <i>Page</i> . . . . .	22
Abb. 5	Abbildung des Metamodells der <i>Condition</i> . . . . .	22
Abb. 6	Abbildung des Metamodells der Expressions . . . . .	23
Abb. 7	Auswahl der zentralen Qualitätseigenschaften nach [Sta20] . . . . .	26
Abb. 8	Verbindung zwischen Objekten [WWJM19] . . . . .	28
Abb. 9	Einsatz von GOPPRR am Modellierungsbeispiel <i>Familienbaum</i> [metb] . . . . .	29
Abb. 10	Erstellung eines Graphen in MetaEdit+ . . . . .	30
Abb. 11	Erstellung eines Objekts in MetaEdit+ . . . . .	31
Abb. 12	Symbol Editor zu einem Objekt in MetaEdit+ . . . . .	32
Abb. 13	Verbindungen in MetaEdit+ . . . . .	33
Abb. 14	Beispielhaftes Modell in MetaEdit+ . . . . .	34
Abb. 15	Zusammenhänge zwischen den grundlegenden Elemente der ADOxx Bibliothek [ado] . . . . .	35
Abb. 16	Darstellung einer Klasseninstanz mit der grafischen Repräsentation aus Lst. 1 . . . . .	36
Abb. 17	Ansicht im Modelling-Toolkit . . . . .	37
Abb. 18	Informationen einer Klasse bestehend aus dem Namen und dem Attribut <i>description</i> . . . . .	37
Abb. 19	Abfrage aller Objekte der Klasse <i>Page</i> . . . . .	38
Abb. 20	Ergebnis der Abfrage nach allen Objekten der Klasse <i>Page</i> . . . . .	38
Abb. 21	Auswertung des Vergleichs von Werkzeugen zur Erstellung grafischer Modellierungseeditoren [Gra16] . . . . .	39
Abb. 22	Beispielhafter Ausschnitt aus dem Spezifikationseditor . . . . .	40
Abb. 23	Konfiguration des Knoten <i>Page</i> . . . . .	40
Abb. 24	Preisstaffelung der akademischen Lizenzen [meta] . . . . .	46
Abb. 25	Netzdiagramm zur Bewertung von MetaEdit+ . . . . .	48
Abb. 26	Netzdiagramm zur Bewertung von ADOxx . . . . .	48
Abb. 27	Arten eines Ausgangs . . . . .	52
Abb. 28	Komponenten der Modellierungssprache im Development-Toolkit . . . . .	54
Abb. 29	Beispielhaftes Modell im Modelling-Toolkit . . . . .	54
Abb. 30	Globale Variable <i>Alter</i> . . . . .	55
Abb. 31	Attribute des <i>Default-Forwarding</i> . . . . .	55
Abb. 32	Fehlgeschlagener Kardinalitätencheck . . . . .	56
Abb. 33	Konsolenausgabe der ersten Seite . . . . .	62
Abb. 34	Konsolenausgabe der zweiten Seite . . . . .	63
Abb. 35	Aktueller Standpunkt im modellierten Ablauf . . . . .	63
Abb. 36	Konsolenausgabe des nächsten Schritts . . . . .	64
Abb. 37	Standpunkt nach der Abzweigung im modellierten Ablauf . . . . .	64
Abb. 38	Auswahl der Option <i>Zurück</i> und Eingabe eines neuen Alters . . . . .	65
Abb. 39	Konsolenausgabe einer Endseite . . . . .	66
Abb. 40	Erreichtes Ende im modellierten Ablauf . . . . .	66

## IV Tabellenverzeichnis

Tab. 1	Vorteile von Modellierung nach [RBGN <sup>+</sup> 17]	6
Tab. 2	Gegenüberstellung GPL und DSL nach [Voe13]	9
Tab. 3	Zuordnung der Qualitätsanforderungen zu den Kriterien mit ihrer Metrik	27
Tab. 4	Bewertung der Erweiterbarkeit	42
Tab. 5	Bewertung der Wiederverwendbarkeit	43
Tab. 6	Bewertung der Modularität	43
Tab. 7	Bewertung der funktionalen Vollständigkeit	44
Tab. 8	Bewertung der Anpassbarkeit	44
Tab. 9	Bewertung der Bedienbarkeit	45
Tab. 10	Bewertung der Verständlichkeit/Erlernbarkeit	45
Tab. 11	Bewertung der Ausgereiftheit	46
Tab. 12	Bewertung des Betriebs	46
Tab. 13	Normalisierung einer Ordinalskala mit 3 Werten	47
Tab. 14	Normalisierung der Lizenzkosten	47

## V Listing-Verzeichnis

1	GraphRep für eine Klasse in ADOxx	36
2	Grafische Repräsentation eines Kreises in MPS	41
3	Repräsentation der Attribute einer <i>Page</i>	50
4	Repräsentation der Modellattribute	51
5	Regel für Variablenwerte	52
6	Darstellung einer Variable als Checkbox	53
7	Kardinalitäten der Klasse <i>Condition</i>	53
8	Auszug aus der XML-Repräsentation der Modellattribute	56
9	XML-Repräsentation der Instanz einer <i>Page</i>	57
10	XML-Repräsentation einer Relation	57
11	Zurodnung eines Übergangs ( <i>Transition</i> zu einem Navigationselement)	60
12	Algorithmus zur Suche der validen Startseite	61

## VI Abkürzungsverzeichnis

<b>BPMN</b>	Business Process Modelling Notation
<b>DSL</b>	Domain Specific Language (domänenspezifische Sprache)
<b>GPL</b>	General Purpose Language
<b>EMF</b>	Eclipse Modeling Framework
<b>UML</b>	Unified Modeling Language
<b>MPS</b>	Meta Programming System
<b>ALL</b>	ADOxx Library Language
<b>AQL</b>	ADOxx Query Language
<b>AST</b>	Abstract Syntax Tree
<b>XML</b>	Extensible Markup Language
<b>ADL</b>	ADOxx Development Language
<b>API</b>	Application Programming Interface

# 1 Einführung und Problemstellung

Der digitale Wandel erstreckt sich mittlerweile über viele Lebensbereiche hinweg. Vor allem in Unternehmen wird stetig an der Einführung neuer, tiefgreifender Technologien in die Geschäftsabläufe gearbeitet. Ziel ist es dabei die Produktivität und Effizienz zu verbessern, neue Einnahmequellen zu erschließen und Wertsteigerungen zu erzielen. Automatisierung für eine schnellere und billigere Produktentwicklung und -einführung ist ein Weg diese Ziele zu erreichen. [LFSS21] Die Automatisierung vieler Aufgaben erfolgt dabei durch den Einsatz von Software, sodass Workflows und Geschäftsprozesse zunehmend datengesteuert ablaufen. Die Entwicklung solcher Systeme ist eine der Herausforderungen der heutigen Zeit. [HT20]

Innerhalb der Organisationen steht der effektive und effiziente Umgang mit Geschäftsprozessen im Fokus. Die Koordination der Prozesse über die Stakeholder hinweg stellt einen der Schlüssel hierzu dar. Die Herausforderung dabei ist es eine Übereinkunft und ein gemeinsames Verständnis über die Prozesse zu erzielen. [BKK18] Zwei grundlegende Konzepte für die Kommunikation verschiedener Stakeholder mit unterschiedlichen, technischen Kenntnissen sind Abstraktion und Modellierung. Abstraktionen können dabei helfen mit der Komplexität des Softwaredesigns umzugehen und ein allgemeines Verständnis herzustellen. Präzise Modelle können dann zur Formalisierung dieser Abstraktionen genutzt werden. [HT20] Modellierungssprachen dienen somit als Schnittstelle zwischen verschiedenen Stakeholdergruppen [BKK18].

Bereits Anfang der 2000er Jahre wurde eine engere Zusammenarbeit von Domänenexperten und Softwareentwicklern angestrebt. Die Vision ist es dabei, dass die Domänenlogik durch die späteren Nutzer erstellt wird. Die Entwickler sorgen lediglich für die entsprechende Toolunterstützung und programmieren Werkzeuge zum Be- und Verarbeiten der Domänenlogik. Eine Möglichkeit hierzu ist die Bereitstellung einer domänenspezifischen Sprache DSL. Diese dient auf der einen Seite zur Optimierung des Entwicklungsvorgehen und auf der anderen Seite ermöglicht sie die Nachvollziehbarkeit und Überprüfbarkeit für Domänenexperten. Des Weiteren wird der Kommunikationsaufwand zwischen technischer und fachlicher Seite reduziert, da *die selbe Sprache gesprochen wird*. [Fow05]

Ein weiterer Faktor, der das Einbeziehen von Nicht-Programmierern in den Entwicklungsprozess begünstigt, ist der Fachkräftemangel und die damit einhergehende Entwicklerknappheit. Ganze Abteilungen müssen auf Grund von Ressourcenknappheiten auf IT-Produkte warten. Dem hohen Bedarf an Workflowautomatisierungen kann nur schwer nachgekommen werden, sodass die Bereitschaft von Geschäftsangestellten Low- und No-Code Plattformen zu nutzen steigt. Durch diese angebotenen Tools können sich die An-

gestellten zu einem gewissen Grad selber helfen und somit die Nachfrage nach professionellem IT-Personal reduzieren sowie gleichzeitig die Effektivität und Produktivität aufrechterhalten. Die Endnutzer können dementsprechend, ohne ausgeprägte technische Fähigkeiten, mit Hilfe von Shared Services, Low- oder No-Code Plattformen und Cloud-Computing Diensten eigene Anwendungen bauen. Diese können intern für einzelne Abteilungen gedacht sein oder auch für die Öffentlichkeit bereitgestellt werden. Insbesondere die geringen Einstiegshürden bei der Nutzung dieser Plattformen, da sie einfach zu bedienen sind und schnelle Änderungen ermöglichen, sorgen für die Verbreitung des sogenannten Citizen Developer. [LFSS21] Im *Gartner Information Technology Glossar* wird der Citizen Developer als Nutzer beschrieben, der nicht der IT-Abteilung angehört und mit Entwicklungs- und Laufzeitumgebungen, die vom Unternehmen genehmigt sind, neue Geschäftsanwendungen erstellt [gar].

Zu beachten ist, dass dennoch einige Herausforderungen für die Unternehmen bestehen bleiben. Dies sind unter anderem der Betrieb dieser Anwendungen, das Zusammenspiel mit der IT-Abteilung während der Entwicklung, die Datenhoheit oder auch die Sicherheit der Anwendungen, da Abhängigkeiten zu den genutzten Plattformen und Werkzeugen aufgebaut werden und diesen bei der Implementierung bezüglich Security und Datenschutz vertraut werden muss. [LFSS21]

In einer Umfrage der *University of South Florida* wurden Ende 2021 knapp 50 IT-Schaffende nach dem Einsatz von Low- oder No-Code Plattformen in ihrem Unternehmen befragt. Trotz der kleinen Stichprobengröße ist eine klare Tendenz erkennbar (siehe Abb. 1). [LFSS21]

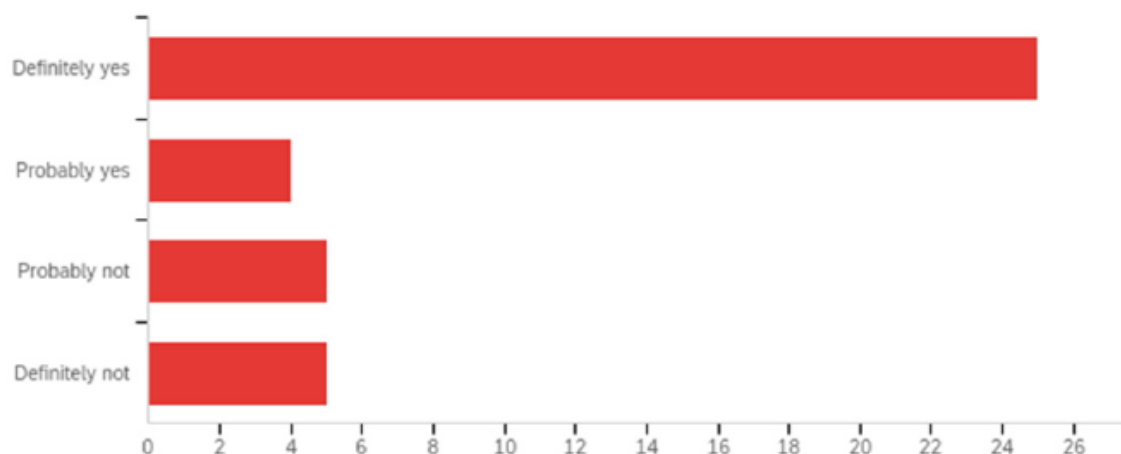


Abbildung 1: Anzahl der Unternehmen, die Low- und No-Code Plattformen einsetzen [LFSS21]



Unabhängig von der Unternehmensgröße beantworteten die meisten Befragten die Frage nach dem Einsatz von Low- oder No-Code Plattformen in ihrem Unternehmen mit ja. Das Konzept des Citizen Developers verbreitet sich somit auch in größeren Unternehmen, obwohl diese zu meist eine entsprechende IT-Abteilung für die technischen Umsetzungen besitzen. Für kleinere Unternehmen hingegen bietet sich durch den Citizen Developer eine Alternative zu den steigenden Kosten im Bereich des Outsourcing. [LFSS21]

Als einer der Treiber dieser Entwicklung stellen sich schnell wechselnde Geschäftsanforderungen heraus. Diese Volatilität der Anforderungen wird beispielsweise durch den Wunsch nach schnelleren Markteinführungszeiten und kürzeren Produktlebenszyklen notwendig. Durch steigende Abhängigkeiten zwischen den Geschäftspartnern und eine engere Integration der unterliegenden Geschäftssysteme wächst auch die Komplexität in der Entwicklung der Anwendungen. Hierfür wird immer mehr auf einen modellbasierten Ansatz gesetzt. Besonders geeignet für die Modellierung und Automatisierung der Geschäftsprozesse sind Systeme mit einer grafischen Oberfläche, da diese für alle Nutzergruppen eine niedrige Einstiegshürde bieten. [BP07]

Das Ziel dieser Arbeit ist es daher eine grafische DSL zu entwickeln, die es ermöglicht bedingungsgesteuerte Prozesse zu modellieren und auszuführen. Dabei wird sich besonders mit dem Fachgebiet der Metamodellierung sowie der Interpretation von Modellen beschäftigt.

Um die Möglichkeiten zur Abstraktion von Abläufen kennenzulernen, werden zu Beginn der Arbeit die Themengebiete Workflow und Prozess (Kapitel 2.1), Metamodellierung (Kapitel 2.2) sowie DSL (Kapitel 2.3) beleuchtet.

Im nächsten Schritt werden die Anforderungen an das Metamodell der grafischen DSL erläutert und vorgestellt (Kapitel 3.1). Die folgende Recherche nach verwandten Arbeiten ermöglicht die Einordnung inwieweit bestehende Modelle die vorgegebenen Anforderungen bereits umsetzen oder sich auch davon unterscheiden (Kapitel 3.2).

Nachdem die Modellierung und Darstellung des Metamodells abgeschlossen ist (Kapitel 4), erfolgt eine Evaluierung von Werkzeugen zur Unterstützung der Entwicklung und späteren Nutzung der DSL. Hierfür werden Kriterien zum Vergleich der Tools definiert. Anhand dieser werden eine Auswahl an Werkzeugen bewertet und verglichen (Kapitel 5).

Auf Basis des ausgewählten Unterstützungswerkzeugs findet die Implementierung der grafischen DSL statt (Kapitel 6). Anschließend wird die Interpretation des modellierten Ablaufs behandelt. Hierbei wird ein selbstgeschriebener Interpreter eingesetzt, der das grafische Modell entgegen nimmt und durch den Prozess leitet (Kapitel 7).

Zuletzt findet eine Betrachtung der Erweiterungsmöglichkeiten der grafischen DSL sowie die Beleuchtung möglicher Einsatzmöglichkeiten statt (Kapitel 8).

## 2 Grundlagen der Abstraktion von Abläufen

Die Abstraktion von Abläufen ist ein elementarer Bestandteil für das Schaffen eines einheitlichen Verständnisses. Durch eine einfache und durchgängige Syntax soll die Lücke zwischen Domänenexperten und der Technik geschlossen werden. Stakeholder ohne technische Kenntnisse haben oftmals Schwierigkeiten abstrahierte Modelle nachzuvollziehen. Sie bevorzugen angereicherte Bilder mit unterscheidbaren grafischen Symbolen. [RBGN<sup>+</sup>17]

Laut Markus Völter gibt es zwei Arten von Modellen. Deskriptive Modelle beschreiben und abstrahieren ein bestehendes System. Sie können als gemeinsame Grundlage zur Kommunikation genutzt werden. Präskriptive Modelle hingegen dienen zur Konstruktion eines Zielsystems. Sie geben strenge formale Regeln zur Vollständigkeit und Einheitlichkeit vor. [Voe13] Anders formuliert beschreibt das deskriptive Modell die Realität, jedoch ist die Realität nicht daraus konstruiert. Die Wahrheit des Modells liegt somit in der Realität. Das präskriptive Modell hingegen schreibt die Struktur und das Verhalten der Realität vor. Die Realität ist somit nach dem Modell entworfen, sodass die Wahrheit im Modell an sich liegt. [YZQ20]

Dies bedeutet, dass es sich beim späteren Metamodell (siehe Kapitel 4) um ein präskriptives Modell handelt, da die Eigenschaften der späteren DSL vorgeschrieben werden. Dagegen sind die Modelle, die mit der Modellierungssprache aus Kapitel 6 erstellt werden, deskriptiv. Sie beschreiben einen spezifischen Ablauf.

Da die Modellierung eine wichtige Rolle bei der Abstraktion der Abläufe einnimmt, werden in der folgenden Tabelle 1 eine Auswahl der Vorteile und Einsatzzwecke der Modellierung dargestellt.

<b>Zweck</b>	<b>Beschreibung</b>
Kommunikation	Einheitliches Verständnis des modellierten Systems, Gemeinsame Grundlage für Diskussionen
Design	Abbildung der neuen Systemstruktur
Verständnis	Vereinfachung von komplexen Konzepten, Übergeordnete Sicht auf das System
Entwicklungsunterstützung	Dokumentation von Architekturentscheidungen, Unterstützung und Anhaltspunkte für die Entwicklung
Visuelle Darstellung	Visualisierung der Systemstruktur und architektonischen Metadaten
Anforderungserhebung	Veranschaulichung der Anforderungen auf verschiedenen Abstraktionsebenen, Unterstützung bei der Identifizierung von fehlenden, falschen oder unnötigen Anforderungen

Tabelle 1: Vorteile von Modellierung nach [RBGN<sup>+</sup>17]

Bei der Betrachtung der Vorteile fällt bereits die enge Verbindung sowie die Wechselwirkung zwischen der konzeptionellen Arbeit und dem Programmieren auf. Im Model-Driven-Engineering beispielsweise werden in der ersten Phase des Softwareentwicklungszyklus die entsprechenden Modelle konzipiert [MAEFH18]. Bei der Modellierung und Programmierung finden ähnliche Abläufe statt. In beiden Fällen wird eine geeignete Notation mit passendem Grad an Ausdruckskraft benötigt. Unterschiede gibt es in der Abstraktionsebene, der Sicht auf das System und dem Maß der Domänenspezifität. Trotz dieser Gemeinsamkeiten betrachten die Entwickler zu meist die Programmierung als ihr Fachgebiet und überlassen die Modellierung den Domänenexperten. [Voe13]

Um von den Vorteilen der Modellierung zu profitieren, wird in einer Vielzahl von verschiedenen Anwendungsfällen auf den Einsatz von Modellen zurückgegriffen. So gibt es beispielsweise Produktmodelle, Prozessmodelle oder Organisationsmodelle. Für all diese Modelle lassen sich verschiedene Konzepte und Werkzeuge nutzen. So verknüpfen zum Beispiel Webservice-Modelle die Geschäftsmodelle mit der Informationstechnologie unter der Nutzung von standardisierten Sprachen und gemeinsamen Ontologien [BP07]. In dieser Arbeit wird sich auf die Art der Prozess- beziehungsweise Ablaufmodelle fokussiert.

## 2.1 Workflow und Prozess

Bei der Betrachtung von Abläufen wird oft von Workflows oder Prozessen gesprochen. Im Folgenden werden die beiden Begriffe erläutert und dargestellt inwieweit sie zusammenhängen.

Ein Workflow beschreibt die notwendigen Aktivitäten zum Abschluss einer Aufgabe. Die

Reihenfolge der benötigten Schritte ist dabei vorgegeben und kann beliebig wiederholt werden. [Bab24]

Prozesse charakterisieren sich durch Zustandsänderungen. Es wird zwischen einem kontinuierlichen Prozess mit stetigem Fluss und einem diskreten Prozess, der die Ausführung von weiteren Aktivitäten enthält und durch diese unterbrochen werden kann, unterschieden. Oft wird im Unternehmensumfeld von Geschäftsprozessen gesprochen. Diese sind eine Spezialisierung eines Prozesses und haben einen Bezug zur Organisation. [Wag23]

Ein Prozess beinhaltet alle Elemente für das Erreichen eines großen, organisatorischen Ziels. Workflows hingegen sind feingranularer und spielen sich auf kleinerer Ebene ab. Sie können als verbundene Einheiten Teil eines Prozesses sein. Ein Prozess kann auch ohne einen Workflow bestehen, wohingegen ein Workflow immer Teil eines größeren Prozesses ist. [Bab24]

Für die Beschreibung von Prozessen gibt es mit der BPMN eine eigene Sprache, die es ermöglicht Prozesse ausführlich zu modellieren. Für den in dieser Arbeit betrachteten prototypischen Anwendungsfall ist diese jedoch zu umfangreich.

## 2.2 Metamodell

Die Entwicklung eines Metamodells ist ein elementarer Bestandteil dieser Arbeit. Metamodelle spielen eine wichtige Rolle bei der konzeptionellen Modellierung. Sie beschreiben das Abstraktionslevel der Modelle und beziehen sich auf die präzise Syntax einer Modellierungssprache, in der das Modell abgebildet wird [MAEFH18]. Es handelt sich um eine Repräsentation für eine Klasse von Modellen. Ein Metamodell ist ein präskriptives Modell und ist allgemein ausgedrückt das Modell, das die Modelle vorschreibt [YZQ20].

Des Weiteren enthält es die verfügbaren Konzepte und ihre validen Kombinationen [BKK18]. Ein Modell ist also eine Instanz eines Metamodells und gehört zu ihm. Jedes Metamodell ist zudem ein Modell. Durch diese Zusammenhänge lassen sich die verschiedenen Abstraktionsstufen abbilden. [Voe13] Brian Henderson-Sellers stellt in seinem Buch zur Mathematik der Modellierung eine zweigeteilte Sichtweise auf die Rolle des Metamodells dar. Auf der einen Seite ist für ihn die Modellierungssprache das Metamodell. Somit definiert das Metamodell die Sprache. Auf der anderen Seite kann die Sprache auch als die Summe aller Modelle, die konform zum Metamodell sind, betrachtet werden. In diesem Fall ist das Metamodell ein Modell der Sprache. [HS12]

Unabhängig davon, ob es sich um ein textbasiertes oder grafisches Metamodell handelt, wird es durch eine Syntax, Semantik und Notation dargestellt. Die Syntax beschreibt die Regeln und Elemente zur Erstellung von Modellen und wird durch eine Grammatik festge-

legt. Die Semantik legt die Bedeutung der Modellierungssprache fest. Die Notation bildet die Visualisierung der Sprache ab. So bietet ein Metamodell beispielsweise Basiskonzepte wie Klassen, Attribute, Modelltypen, Beziehungen und Skripte. [BP07] Die Notation mit ihren Symbolen zur Visualisierung beeinflusst nicht den Zustand des Modells während der Modellierung. [KK02]

Designentscheidungen im Metamodell beeinflussen Nutzen, Fähigkeiten und Ausdruckskraft der Modellierungssprache und der erzeugten Modelle [BKK18]. Die Änderungen im Metamodell werden an die darauf basierenden Modelle delegiert, um diese konsistent zu halten [BP07].

### 2.3 Domain Specific Language (Domänenspezifische Sprache)

Ein Ansatz die steigende Komplexität im Softwareentwicklungsprozess zu handhaben, ist es das Softwaresystem aus Sicht der Domäne zu betrachten. Eine DSL gibt die Notation zur Modellierung des Domänenwissens vor. Sie ist eine begrenzte Form einer Programmiersprache, entworfen für eine bestimmte Klasse von Problemen [Fow05]. Die Repräsentation der DSL bezieht sich dabei nicht auf die Logik oder Daten des Systems, sondern die Elemente der DSL sind die typischen Konzepte und Beziehungen der bestimmten Domäne. Business Process Modelling Notation (BPMN) beispielsweise ist eine DSL zur Modellierung von Geschäftsprozessen. [MAEFH18]

Außerdem zielt die Verwendung einer DSL auch auf die anfangs erwähnte Idee ab, die Zusammenarbeit zwischen Domänenexperten und Entwicklern zu verbessern. Der Fokus auf domänenspezifische Aspekte ermöglicht es die Sprache genauer auf die Problemstellung zuzuschneiden. Die Modellierer können damit das Domänenwissen so intuitiv wie möglich spezifizieren, sodass durch eine Erhöhung des Abstraktionslevels des Modells die Nähe zur Problemdomäne hergestellt wird, jedoch die Distanz zu Implementierungsdetails gewahrt wird. Die Aufgabe der Modellierer besteht darin die Domänenkonzepte und -beziehungen in das Modell einzubringen. [YZQ20]

Eine DSL besteht aus den folgenden vier Hauptbestandteilen. Die konkrete Syntax definiert die Notation, mit welcher der Nutzer das Programm oder Modell erstellt. Dies kann grafisch oder textbasiert sein. Unter der abstrakten Syntax wird die Struktur, welche die semantisch relevanten Informationen abbildet, verstanden. Bei dieser Struktur kann es sich zum Beispiel um einen Baum oder Graphen handeln. Die statische Semantik beinhaltet die Regeln und Bedingungen, die eingehalten werden müssen, um strukturell korrekt zu sein. Die Bedeutung eines Programms, wenn es ausgeführt wird, betrachtet die Ausführungssemantik. [Voe13] Im Anwendungsfall dieser Arbeit handelt es sich dabei um einen Interpreter, der den modellierten Ablauf simuliert. Eine DSL muss jedoch nicht

ausführbar sein. Es kann sich auch lediglich um eine Spezifikation, Definition oder Beschreibung handeln. Als Alternative zur Interpretation der DSL kann auch ein Generator geschrieben werden. Dieser kompiliert die gegebene Repräsentation beispielsweise in eine andere bekannte Programmiersprache. [MHS05]

Im Gegensatz zu einer General Purpose Language (GPL), durch welche jedes Programm flexibel ausgedrückt werden kann, bleiben trotz der möglichen Codegenerierung die Domänenbelange im Mittelpunkt. In der nachfolgenden Tabelle 2 sind die Unterschiede bei der Programmierung mit einer GPL und DSL gegenübergestellt. [Voe13]

	<b>GPL</b>	<b>DSL</b>
Domäne	groß und komplex	klein und klar definiert
Größe der Sprache	groß	klein
Turing-Vollständigkeit	immer	meistens nicht
Benutzerdefinierte Abstraktionen	anspruchsvoll	begrenzt
Ausführung	über GPL	nativ
Lebensdauer	Jahre bis Jahrzehnte	Monate bis Jahre (je nach Kontext)
Entworfen von	Guru oder Komitee	ein paar Softwareingenieuren und Domänenexperten
Benutzer-Community	groß, anonym und weit verbreitet	klein, zugänglich und lokal
Evolution	langsam, oft standardisiert	schnelllebig
Veraltete/inkompatible Veränderungen	fast unmöglich	möglich

Tabelle 2: Gegenüberstellung GPL und DSL nach [Voe13]

Anhand der Tabelle lassen sich bereits einige Vorteile in der Nutzung einer DSL ableiten. Durch die Komprimiertheit und Effizienz einer DSL entsteht wenig Overhead, sodass eine Steigerung der Produktivität bei der Implementierung mit der DSL erzielt wird. Dies könnte bei der Entwicklung mittels einer GPL auch durch die Nutzung einer Bibliothek oder eines Frameworks erreicht werden. Durch den Fokus auf den Domänenkontext ist die domänenspezifische Sprache semantisch reichhaltig und es ergeben sich Vorteile in der Validierung und Verifizierung. So lassen sich Analysen anhand aussagekräftiger Fehlermeldungen einfacher implementieren und Domänenexperten können manuelle Reviews durchführen. Ein weiterer Vorteil ist die, einführend bereits erwähnte, geförderte Kommunikation zwischen Entwicklern und Domänenexperten. Durch die Einbindung im Entwicklungsprozess lassen sich Unterschiede im Verständnis oder verschiedene Sichtweisen frühzeitig ausräumen. Für eine produktive Werkzeugunterstützung werden oftmals Tools an die DSL angepasst oder eigens erstellt. Des Weiteren wird durch die Begrenzung

der Sprache auf eine Domäne eine bessere Qualität des Codes gefördert. Es entstehen weniger Bugs, die Architektur kann passender geschnitten werden und auch die Wartbarkeit ist höher. Diese Limitierung, die lediglich die Erstellung Domänen-korrektur Programme erlaubt, wird auch als *correct-by-construction* bezeichnet. Zuletzt kann durch den Einsatz einer DSL eine Plattformunabhängigkeit erreicht werden. Die Sprache abstrahiert dabei die unterliegende Technologieplattform und die Ausführung erfolgt unabhängig von der Zielplattform. [Voe13]

Nachteile bei Anwendung einer DSL sind unter anderem der Aufwand in der Entwicklung der Sprache. Hinzu kommen die benötigten Fähigkeiten zur Sprachentwicklung, die über standardmäßige Softwareentwicklungskenntnisse hinaus gehen. Außerdem geht die gewonnene Wiederverwendbarkeit mit einem gestiegenen Aufwand für die Wartung und Weiterentwicklung einher, um eine veraltete Sprache zu vermeiden. Auch der beschriebene Vorteil der zugeschnittenen Werkzeuge hat eine negative Seite. Durch die Spezialisierung entsteht eine Abhängigkeit zum Tool, das die DSL unterstützen muss. Ein weiterer großer Punkt sind kulturelle Herausforderungen. Besonders Sätze wie “Entwickler wollen programmieren und nicht modellieren“ oder “Domänenexperten sind keine Programmierer“ stellen eine veraltete Sichtweise dar und können am Anfang Hindernisse für das Team sein. Vor allem der Prozess für die Zusammenarbeit und Interaktion zwischen Sprachentwicklern, Sprachnutzern und Domänenexperten muss geklärt sein. Auf der anderen Seite muss auf eine ausgewogene Nutzung von DSLs geachtet werden. Die Kompaktheit und Spezialisierung einer möglichen DSL können schnell zu einer Neuentwicklung verleiten. Dies führt letztendlich zu einer schwer handhabbaren Menge an DSLs. Es sollte zu Beginn immer erst eine Prüfung auf bestehende Lösungen und die Tauglichkeit einer DSL erfolgen. [Voe13]

Bei einer DSL kann zwischen einer internen oder externen DSL unterschieden werden. Bei einer externen DSL handelt es sich um eine neue Sprache, wohingegen eine interne DSL auf einer bestehenden Programmiersprache aufbaut und in diese eingebettet wird. Weitere Unterschiede zwischen externer und interner DSL werden im Folgenden aufgeschlüsselt.

**Externe DSL** Durch die Entwicklung einer separaten Sprache ergibt sich eine große Freiheit in der Gestaltungsform und Abbildung der Funktionalitäten. Die einzige Grenze ist die Sicherstellung einer späteren Übersetzung oder Interpretation. Der Nachteil ist hierbei, dass ein Interpreter entwickelt werden muss. Dieser ermöglicht es jedoch die DSL zur Laufzeit zu evaluieren, sodass Änderungen im Gegensatz zu kompilierten Sprachen direkt nachvollzogen werden können. Bei der Erstellung einer eigenen Sprache ergibt sich außerdem die Herausforderung eine gute Abstraktion zu finden. Bei der Vielfalt an Sprachen und Notationen ist die Einordnung in den ent-



sprechenden Kontext und die Einbindung geeigneter Symbole für eine eindeutige semantische Interpretation wichtig. Des Weiteren muss eine passende Werkzeugunterstützung sichergestellt werden. [Fow05]

**Interne DSL** Hier drehen sich die Vor- und Nachteile im Vergleich zur externen DSL um. Es gibt keine symbolische Barriere in der Notation, da die Sprache gleich bleibt und wiederverwendet wird. Dafür besteht die Abhängigkeit zur Basissprache. Der Entwickler ist in ihrer Struktur gefangen. [Fow05] Nicht-Entwickler hingegen können durch die Basisfunktionen der Entwicklungssprache verwirrt werden, sodass auf Grund der Programmiersprache die bekannte Kommunikationsbarriere zwischen Entwicklern und Stakeholdern besteht. [Lä18]

Bei der Entscheidung für die Nutzung einer DSL handelt es sich unter anderem um die Abwägung zwischen den Vorteilen einer DSL und den Kosten der Entwicklung eigener Unterstützungswerkzeuge für die Bearbeitung der Problemstellung. Interne DSLs helfen bei der Reduzierung der Toolkosten. Dafür besteht die Abhängigkeit zur gewählten Programmiersprache. Externe DSLs hingegen bieten das meiste Potenzial, um alle Vorteile zu nutzen. Hier steckt jedoch Aufwand im Design der Sprache. Außerdem müssen Überlegungen und Evaluierungen zu Werkzeugen zur Entwicklungsunterstützung angestellt werden. [Fow05]

Ein wesentlicher Bestandteil bei der Entwicklung eines Systems ist das Testen des Codes. Dies stellt bei der Verwendung einer DSL eine besondere Herausforderung dar. Es können zwar beispielsweise Tests für den Interpreter geschrieben werden, allerdings lässt sich nur schwer überprüfen, ob sich mit der gegebenen Syntax alle notwendigen Funktionalitäten abbilden lassen oder ob nicht sogar falsche Ausdrücke möglich sind und diese nicht erkannt werden. Mit manuellen Unittests kann kontrolliert werden, ob sich das Programm wie erwartet verhält. Für eine formale Überprüfung zusätzlich zu den manuellen Tests der Semantik kann beispielsweise die DSL erweitert werden, um Testfälle abzubilden. Hierbei besteht allerdings die Gefahr, dass die Tests, geschrieben mit einer fehlerhaften DSL, ebenfalls fehlerhaft sind. Außerdem muss auf eine angemessene Anzahl an Testfällen geachtet werden. Da jeder Test ein spezielles Ausführungsszenario beschreibt, sind für eine aussagekräftige Abdeckung viele Tests notwendig. Besser ist hier eine Verifizierung des ganzen Programms. Dafür eignen sich insbesondere die Ansätze des *Model Checking* und der *Abstract Execution*. [Voe13]

Wichtig für den Erfolg einer DSL ist, dass die Nutzer durch die Nutzung Vorteile haben. Erfolgsfaktoren können die Fehlerbehandlung, Werkzeugunterstützung, statische Analysen oder auch Simulatoren sein. Vor allem die Nutzung einer grafischen Sprache bietet

den Vorteil einer zugänglicheren Syntax im Vergleich zu einer Textsprache. Durch die somit geschaffene Nachvollziehbarkeit wird die Erlernbarkeit sowie die Überprüfbarkeit für Domänenexperten erleichtert. [Tom17]

Bei der Entwicklung der grafischen DSL im weiteren Verlauf der Arbeit wird zu Beginn die abstrakte Syntax definiert. Dabei handelt es sich um das Schema der abstrakten Repräsentation der Sprache. Im zweiten Schritt erfolgt die Recherche nach einem geeigneten Werkzeug bzw. Editor, um die abstrakte Repräsentation zu manipulieren. Zuletzt wird der Interpreter definiert und entwickelt. Anders als bei einem Generator, der die abstrakte Repräsentation in eine ausführbare Repräsentation übersetzt, führt der Interpreter das Programm direkt aus und evaluiert es. Mit dem Interpreter wird somit die Semantik der DSL definiert. [Fow05]

### 3 Anforderungen und verwandte Arbeiten

In diesem Kapitel werden die vorgegebenen Anforderungen an das Metamodell erläutert. Da bedingungsgesteuerte Prozesse mehrere Schritte mit verzweigten Abläufen enthalten können, werden nun zwei Elemente eingeführt. Bei einer *Page* handelt es sich um einen Schritt oder Zustand im Ablauf. Nach einem Zustandsübergang stellt eine *Page* die neue Landeseite dar. Das zweite Element wird im Folgenden *Condition* genannt. Diese übernimmt die bedingungsgesteuerte Navigation und leitet anhand einer vorgegebenen Regel zur passenden *Page* weiter.

Nachdem im ersten Schritt die Zusammenhänge zwischen *Pages* und *Conditions* sowie die damit einhergehenden Anforderungen und Regeln für das Metamodell beschrieben werden, erfolgt im zweiten Unterkapitel die Betrachtung von verwandten Arbeiten.

#### 3.1 Anforderungen an das Metamodell

Die nachstehende Auflistung enthält die Anforderungen und Regeln, die den Ablauf der bedingungsgesteuerten Prozesse bestimmen.

1. Ein Ablauf besteht aus mehreren Pages.
2. Einzelne Pages können aus mehreren Komponenten bestehen. Diese werden innerhalb des ersten Entwurfs dieser Arbeit jedoch nicht weiter behandelt und spezifiziert. Zukünftig könnte es sich dabei um Elemente zur Beschreibung einer grafischen Nutzerschnittstelle handeln.
3. Jeder Ablauf besteht aus einem Start und einem oder mehreren Enden.
4. Innerhalb einer Page können globale Variablen gelesen und gesetzt werden.
5. Jede Page hat Eingänge und Ausgänge.
  - (a) Die Start-Page muss keinen Eingang haben.
  - (b) Die End-Pages können auf einen Ausgang verzichten.
  - (c) Jede andere Page navigiert mindestens zu einer weiteren Page. Sie besitzt somit einen Default-Ausgang.
6. Ein Ausgang kann genau zu einer weiteren Page oder Condition führen.
7. Eine Page kann lokale Variablen enthalten.
8. Alle Ausgänge einer Page sind eindeutig benannt

9. Die Navigation von einer Page auf die nächste Page kann von einer Condition abhängen.
10. Eine Condition hängt von einer globalen Variable ab.
11. Eine Condition kann mehrere Eingänge und Ausgänge haben.
12. Die Ausgänge einer Condition sind an Bedingungen geknüpft. Auch hier gibt es immer einen Default-Weg.
13. Alle Ausgänge einer Condition sind eindeutig benannt.

Die nachfolgenden Punkte stellen weitere Randbedingungen, welche die Arbeit mit dem Metamodell betreffen, dar.

1. Das Metamodell ist als DSL mit grafischer Repräsentation modellierbar.
2. Die Ausführung von Modellen des Metamodells erfolgt immer als Interpretation
3. Die Interpretation des modellierten Ablaufs ermöglicht es die einzelnen Ausgänge anzusteuern und in den Pages globale Variablen zu setzen.
4. Eine Kompilierung des Metamodells soll nicht stattfinden.

Um in der Arbeit den Fokus auf der technischen Umsetzung zu belassen, wird auf die Einführung einer spezifischen Domäne verzichtet. Es handelt sich um einen generischen Ablauf. Die vorgegebenen Anforderungen müssen bei einer späteren Erweiterung um eine bestimmte Domäne lediglich geschärft und um den Domänenkontext erweitert werden, sodass ein Großteil der Implementierung übernommen werden kann.

## **3.2 Verwandte Arbeiten zur grafischen Beschreibung von Abläufen**

Hier werden veröffentlichte Paper oder Artikel betrachtet, die sich mit der Modellierung einer domänenspezifischen Sprache, der Unterstützung des Vorgehens durch geeignete Tools und Werkzeuge sowie den Auswirkungen befassen. Des Weiteren werden dann Arbeiten gesucht, in denen sich Teile der Anforderungen an die Beschreibung der Abläufe wiederfinden. Daraus ergeben sich einige elementare Anforderungen an die Plattformen zur Unterstützung der Modellierung. Diese resultieren aus der Anpassung an die Problemstellung und der damit einhergehenden freien Definition des Metamodells als Formalisierung der Modellierung. Dies sind unter anderem Flexibilität, Anpassbarkeit und Offenheit. Des Weiteren schreiben Karagiannis und Kühn in ihrer Veröffentlichung zu Metamodellierungsplattformen der Modellierungshierarchie eine wichtige Rolle zu. Durch das

Metamodell als Modell einer Modellierungssprache wird eine Hierarchie an Sprachen beziehungsweise Metasprachen gebildet. Die Beschreibung des Metamodells geschieht somit beispielsweise mit einer Metamodellierungssprache. Der Zusammenhang zwischen einer Instanz eines Modells, welches mit einer Modellierungssprache basierend auf einem Metamodell erstellt wurde, ist in Abbildung 2 dargestellt. Hier wird auch der Zusammenhang zwischen den verschiedenen Sprachebenen eingeordnet. [KK02]

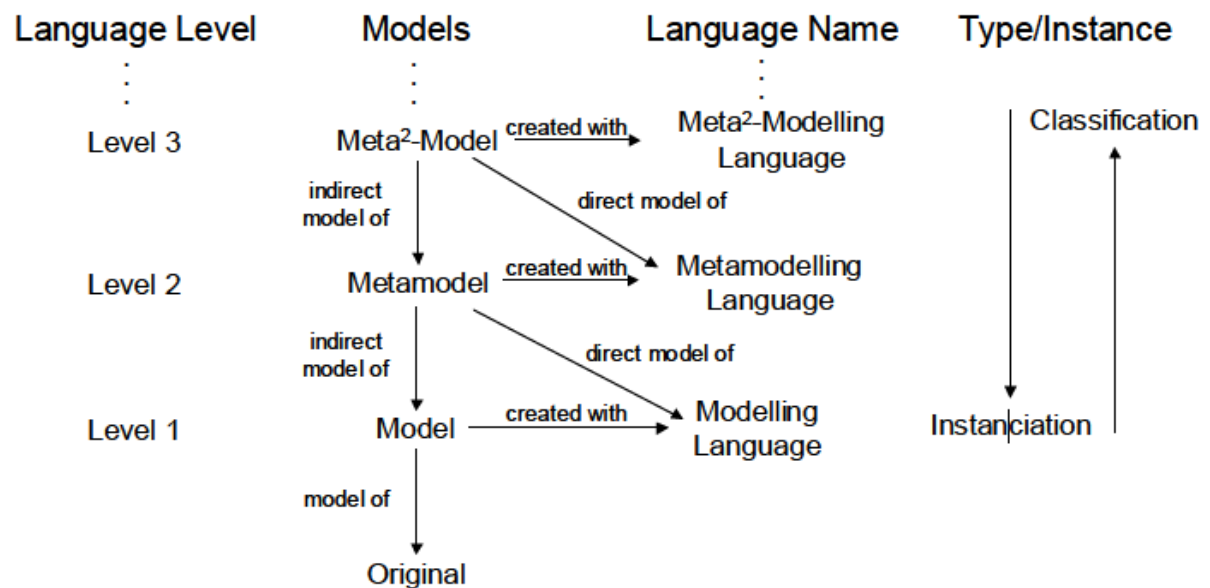


Abbildung 2: Metamodellierung basierend auf Sprachebenen [KK02]

Dabei ist es wichtig die passende Anzahl an Sprachebenen für den richtigen Grad an Abstraktion zu finden. Als etablierte Produkte für die Modellierung nennen Karagiannis und Kühn in ihrem Paper von 2002 ADONIS und MetaEdit+ . [KK02]

Da es sich bei dem Anwendungsfall in dieser Arbeit um einen sehr generischen Ablauf handelt, kann auf eine Aufgliederung in mehrere Ebenen verzichtet werden. Bei einer weiteren Spezifizierung der Anforderungen kann in Zukunft jedoch auch die Erweiterung des Sprachmodells in Form einer zusätzlichen Ebene sinnvoll sein. Das hier entwickelte Metamodell würde dann zum Meta-Metamodell werden.

Ein praxisbezogenes Beispiel bietet der Artikel zur Einführung der domänenspezifischen Modellierung bei der Firma *Polar*, einem der führenden Hersteller von Sportuhren zur Messung der Herzfrequenz. Dort konnte durch die Erhöhung der Abstraktion mittels der Spezifizierung der Sprache und ihrer Regeln basierend auf der Domäne die Entwicklungsproduktivität im Bereich der Embedded-Geräte um fast 750% gesteigert werden. Hierzu wurde auf Basis der Modelle ein vollständig funktionaler Produktionscode generiert. Als

besonders hilfreich hat sich dabei der Einsatz der MetaEdit+ Workbench erwiesen. Mit der Erweiterung konnte im Gegensatz zum klassischen Modeler neben der Modellierungssprache auch der Codegenerator entwickelt werden. [KTK09]

Die damit erreichten Ziele bei *Polar*, wie zum Beispiel die Produktivitätssteigerung, ein höherer Grad an Automatisierung und Anpassbarkeit oder auch die einfachere Einführung für neue Entwickler, lassen sich auch mit der domänenspezifischen Sprache in dieser Arbeit erreichen. Im letzten Schritt der hier dargelegten Ausarbeitung wird lediglich auf die Generierung von Code, wie im Beispiel von *Polar*, verzichtet und auf die direkte Interpretation des Modells gesetzt (siehe Kapitel 7).

Einen weiteren Anhaltspunkt für die Implementierung einer domänenspezifischen Modellierungssprache bietet das Paper vom *Institute for Software-Integrated Systems*. Darin werden unter anderem verschiedene Tools zur Unterstützung von Modellierungsprozessen betrachtet. Für das bereits bekannte MetaEdit+ wird beispielsweise die Möglichkeit zum Import und Export einer XML-Datei hervorgehoben. Außerdem wird das Eclipse Modeling Framework (EMF) erwähnt, welches im Kosmos von Eclipse viele Funktionalitäten über die Modellierung hinaus anbietet. So kann zum Beispiel anhand einer Dateirepräsentation des Modells eine Java-API generiert werden. [MBL<sup>+</sup>11]

Abschließend zur Betrachtung der Arbeiten mit ähnlichen Zielen können einige zur Unterstützung geeignete Anwendungen festgehalten werden, die in der späteren Werkzeuganalyse genauer analysiert werden. Bezüglich der Modellierungssprache wurden nach der Recherche über Google Scholar, dem OPAC Bibliothekssystem der Hochschule und weiteren Seiten zur Veröffentlichung wissenschaftlicher Arbeiten keine Ausarbeitung mit ähnlichen Anforderungen zur Ablaufbeschreibung gefunden. Auf Grund der fehlenden fachlichen Ähnlichkeit in den begutachteten Werken sowie der geringen Anzahl an praktischen Ausarbeitungen wird im Folgenden eine eigene DSL implementiert.

Zur Unterstützung dieses Beschlusses wird nachfolgend auf eine kleine Auswahl der Entscheidungspattern zum Einsatz von eigenen DSLs aus dem Paper *When and How to Develop Domain-Specific Languages* eingegangen, sodass die Wiederverwendung, Erweiterung oder Spezialisierung einer bestehenden Sprache endgültig ausgeschlossen werden kann. [MHS05]

**Notation** Bekannte Notationen wie UML oder BPMN sind zu umfangreich und werden dem spezifischen Charakter, der hier vorgegebenen Anforderungen nicht gerecht.

**Automatisierung** Ein gewichtiges Argument bei der Entscheidung für die Nutzung einer DSL ist die Möglichkeit zur automatischen Ausführung wiederkehrender Abläufe. Dies trifft den Kern der beschriebenen Anforderungen und kann durch die Interpre-

tation des Modells erreicht werden.

**Trennung vom Code** Im Gegensatz zu oben aufgeführten Beispielen soll in dieser Arbeit das Modell nicht als Grundlage zur Codegenerierung dienen. Dadurch entsteht eine Trennung zwischen dem Domänenwissen und dem Code des Interpreters.

## 4 Entwicklung eines Metamodells für eine grafische DSL

Nachdem in den vorherigen Kapiteln die verschiedenen Umsetzungsarten bezüglich eines Einsatzes von DSLs beschrieben und durchleuchtet wurden, erfolgt nun im ersten Schritt der praktischen Ausarbeitung die Implementierung des Metamodells.

Bei der Entwicklung eines eigenen Metamodells zur Abbildung einer grafischen DSL gibt es einige Punkte zu beachten. Die Akzeptanz des Metamodells beispielsweise wird von verschiedenen Aspekten beeinflusst. Es muss sich für die geplante Anwendergruppe als nützlich erweisen, sodass der Arbeitsablauf und die Leistung verbessert werden kann. Ein weiterer Faktor ist die Benutzerfreundlichkeit. Das Modell sollte unter geringem Aufwand eingesetzt werden können und somit direkt den Nutzen steigern. Des Weiteren ist auf die Einführung innerhalb der Organisation oder des Teams zu achten. Hier ist eine ausreichende Unterstützung durch das Unternehmen, wie zum Beispiel Schulungsangebote, hilfreich. Ebenfalls wichtig für eine hohe Akzeptanz ist der Reifegrad des Modells und der Sprache. Alle benötigten Fähigkeiten für den geplanten Aufgabenbereich sollten vorhanden sein und der Status eines Prototypen sollte bereits überschritten sein. Zuletzt ist eine entsprechende Wirksamkeit elementar. Die Nutzung des Metamodells sollte zu Vorteilen in der Durchführung der Arbeit und einer steigenden Effektivität führen. [MAEFH18]

Für die Entwicklung einer eigenen DSL haben unter anderem Gabor Karsai und Holger Krahn in ihrem Paper *Design Guidelines for Domain Specific Languages* grundlegende Richtlinien für textuelle und grafische DSLs beschrieben. Im Anschluss werden einige Leitlinien wiedergegeben. Analog zur Ausarbeitung des Papers sind diese in verschiedene Kategorien, wie unter anderem Zweck der Sprache und Sprachinhalt, untergliedert. [KKP<sup>+</sup>14]

### Zweck der Sprache

**Sprachgebrauch** Es soll frühzeitig der Sprachgebrauch identifiziert und festgelegt werden, welche Aufgaben die Sprache erfüllen soll.

**Konsistenz** Die Sprache soll kohärent und konsistent gestaltet sein. Alle Eigenschaften der Sprache sollen dem speziellen Einsatzzweck dienen.

### Umsetzung der Sprache

**Textuell vs. Grafisch** Die Entscheidung zwischen einer textuellen und grafischen DSL beeinträchtigt auch die mögliche Unterstützung durch Frameworks und Werkzeuge. Für diese Arbeit wurde sich bereits auf eine grafische DSL fest-



gelegt, sodass beispielsweise die Vorteile einer besseren Verständlichkeit und eines schnelleren Überblicks genutzt werden.

**Sprachkombination und Wiederverwendung** Aus bestehenden Sprachen können Teile übernommen und somit zur neuen Sprache kombiniert werden. Auch das Wiederverwenden von bekannten Konzepten oder Typsystemen kann die Entwicklung der neuen Sprache erleichtern.

### Sprachinhalt

**Notwendige Domänenkonzepte** Angepasst an den Zweck der Sprache sollten nur die notwendigen Domänenkonzepte umgesetzt werden. Ein ausuferndes Set an Funktionalitäten kann der späteren Akzeptanz schaden.

**Einfachheit** Anknüpfend an die Wiederverwendung und Kombination bekannter Konzepte soll alles so einfach wie möglich gehalten werden.

**Vermeidung von Verallgemeinerungen** Die DSL soll auf ihre spezielle Absicht zugeschnitten sein und braucht nicht alle Facetten auf eine allgemeine Basis herunterbrechen.

**Limitierung der Sprachelemente** Desto mehr Elemente die Sprache enthält, umso schwieriger wird es für den Nutzer die Sprache zielgerichtet anzuwenden.

**Vermeidung von Redundanzen** Die mehrfache Umsetzung von Konzepten auf verschiedene Weisen hat ebenso einen negativen Einfluss auf die Nutzbarkeit und Eindeutigkeit der Sprache.

### Sprachsyntax

**Übernahme von Notationen** Wie bei den Sprachkonzepten soll auch bei der Notation der Sprache auf bereits bestehende und bekannte Notationen zurückgegriffen werden. Die Übernahme einer Syntax, die bereits Domänenexperten bekannt ist, kann die Akzeptanz der DSL positiv beeinflussen.

**Kommentarfunktion** Die Möglichkeit dem Modell Kommentare hinzuzufügen erleichtert den Domänenexperten die Dokumentation sowie den Austausch über verschiedenen Modelle.

Insbesondere bei der Entwicklung einer grafischen DSL lassen sich drei Herausforderungen festhalten. [RBGN<sup>+</sup>17]

1. Komplexitätsmechanismus: Laien werden durch die Komplexität der visuellen Komponenten herausgefordert. Ihnen fehlt oft die Fähigkeit das Gesamte in kleinen

Bausteinen zu erfassen. Daher sollte das Modell die Gruppierung von Elementen zu kleineren Modulen ermöglichen und abbilden können.

2. Variation von visuellen Variablen: Visuelle Variablen können bei der Unterscheidung von verschiedenen Elementen im Modell helfen. Dabei sollten jedoch Elemente und Variablen vermieden werden, die nur in einer Kombination aus Form, Farbe, Größe oder Textur unterscheidbar sind.
3. Variation der Anzahl der Sprachelemente: Bei der Erstellung der visuellen Variablen muss darauf geachtet werden, dass mit jeder Variable die Komplexität der Modellierungssprache erhöht wird. Als Richtlinie kann beispielsweise die nach George A. Miller magische Zahl  $7 \pm 2$  genutzt werden [Mil56]. Demnach kann der Anwender nur zwischen fünf und neun Elementen sinnvoll verarbeiten, sodass die Komplexität und der Umfang begrenzt werden. Weitere Optionen innerhalb der Modellierungssprache bieten die Verkettungen der bekannten Symbole.

Die aufgeführten Punkte gilt es nun bei der Überführung der Anforderungen (siehe Kapitel 3.1) in das neue Metamodell zu beachten.

Zu Beginn der Modellierung der Anforderungen aus dem Kapitel 3.1 wird auf oberster Ebene mit der Abbildung der *Page* und der *Condition* gestartet. Da es sich um einen Ablauf mit der Navigation durch verschiedene Schritte handelt, werden die beiden Elemente *Page* und *Condition* als Navigationselemente zusammengefasst. Durch die Anforderungen 4 und 10 aus Kapitel 3.1 ist festgelegt, dass in beiden Navigationselementen der Zugriff auf globale Variablen benötigt wird. Aus diesem Grund wird direkt auf der höchsten Ebene auch der globale Speicher (*Memory*) eingeführt (siehe Abb. 3).

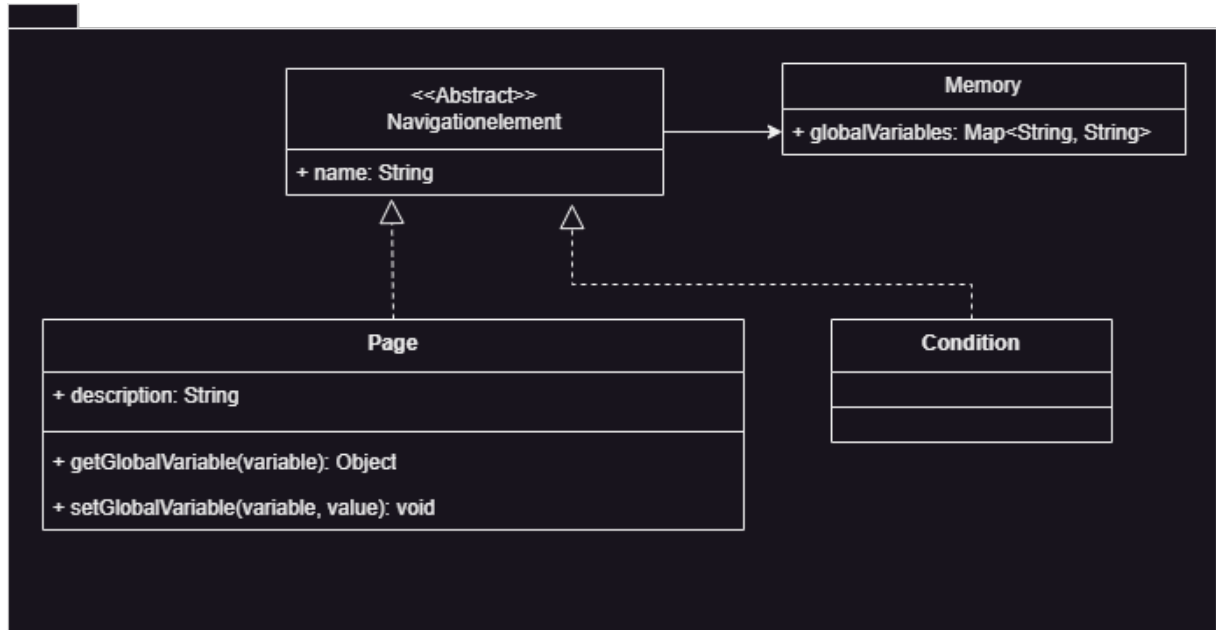
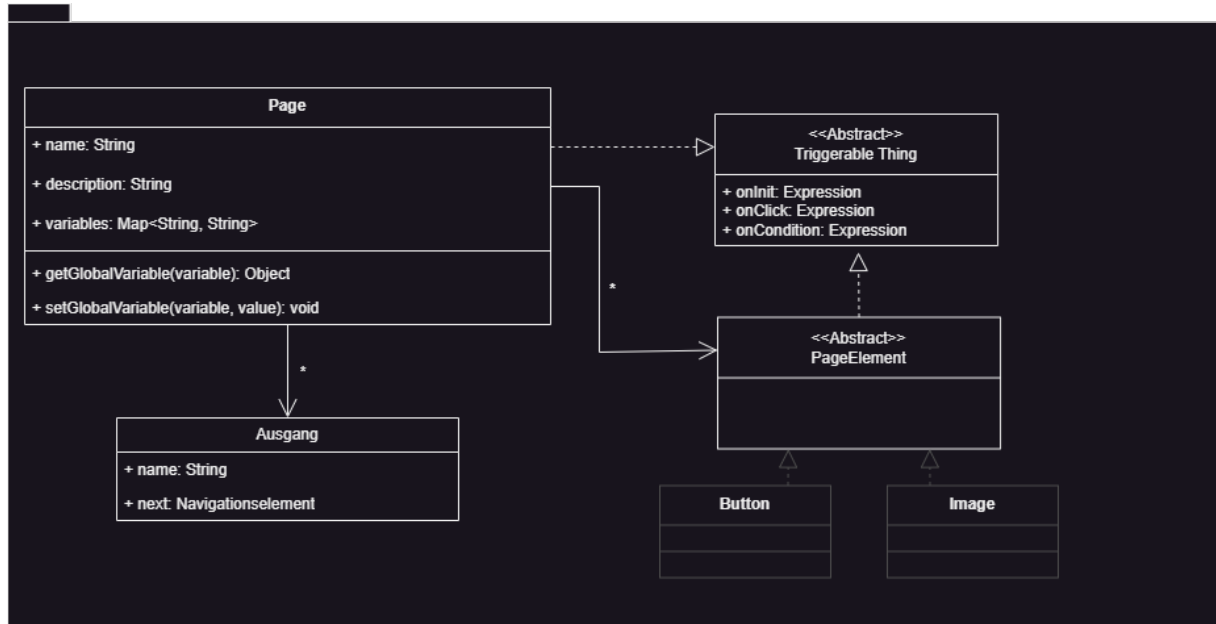
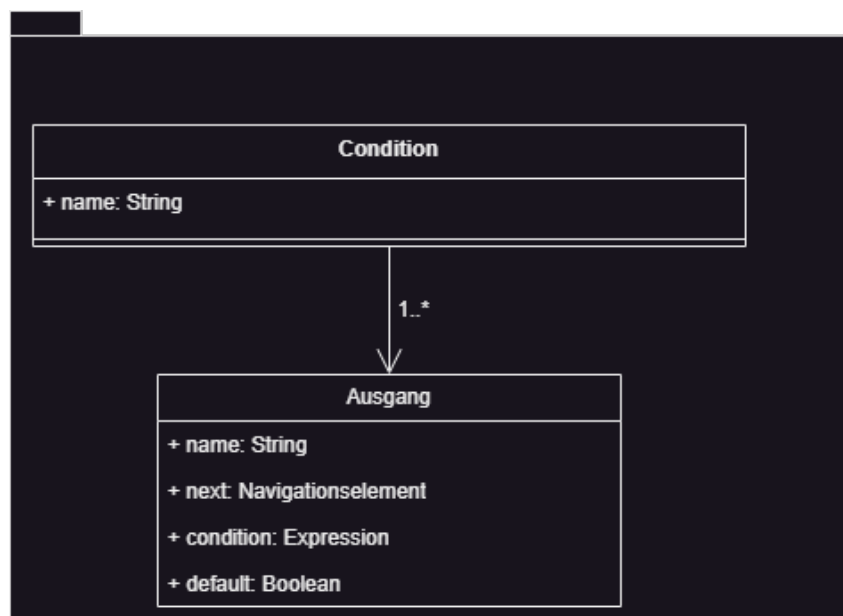


Abbildung 3: Abbildung des Metamodells auf oberster Ebene

Im weiteren Verlauf werden nun die beiden Navigationselemente *Page* und *Condition* genauer spezifiziert. Wie in Anforderung 2 erläutert, kann eine Page aus einer beliebigen Anzahl an Komponenten bestehen. Diese können beispielsweise Elemente einer grafischen Darstellung für den Nutzer sein. Sie sind nachfolgend als *PageElement* bezeichnet. Die Seitenelemente können dann wiederum zur Steuerung des Ablaufs genutzt werden. Dies kann von einer Bedingung abhängen, aber auch bereits bei der Initialisierung der Seite oder erst beim Bedienen eines Elements ausgeführt werden. Diese drei Optionen zum Auslösen der Fortsetzung des Ablaufs sind unter dem Objekt *Triggerable Things* gesammelt. Die Verknüpfung der Navigationselemente erfolgt über den Ausgang. Für das spezielle Szenario einer Seite muss nicht unbedingt ein Ausgang angegeben sein. Besitzt eine Seite keinen Ausgang handelt es sich um das Ende des Ablaufs. Die Startseite kann als die Seite identifiziert werden, auf die durch keinen Ausgang gezeigt wird. Die Eigenschaften einer *Page* sind in Abbildung 4 dargestellt.

Abbildung 4: Abbildung des Metamodells der *Page*

Im Gegensatz zu einer Seite ist die *Condition* kein Element, mit dem interagiert wird. Es dient lediglich zur Regelung des Ablaufs und wird genutzt, um Abzweigungen abzubilden. Eine *Condition* kann nie das Ende eines modellierten Ablaufs sein. Daher gibt es immer mindestens einen Ausgang. Gibt es mehrere Ausgänge, so hängt das Ansteuern des richtigen Ausgangs von der Bedingung, die dem Ausgang zugeordnet ist, ab. Zusätzlich wird nach Anforderung 12 immer ein Ausgang als Default-Ausgang gekennzeichnet. (siehe Abb. 5).

Abbildung 5: Abbildung des Metamodells der *Condition*

Sowohl innerhalb einer *Page* beim *Triggerable Thing* als auch bei der *Condition* im Ausgang gibt es Variablen, die den Ablauf bestimmen und als Expression abgebildet werden. Expressions können genutzt werden, um arithmetische Ausdrücke darzustellen. Diese könnten dann als binärer Baum abgebildet werden. Dabei sind die Blätter die Variablen und die Knoten die Operatoren. [CT91]

Für das hier erstellte Metamodell schaut der Teil der Expressions wie folgt aus (siehe Abb. 6).

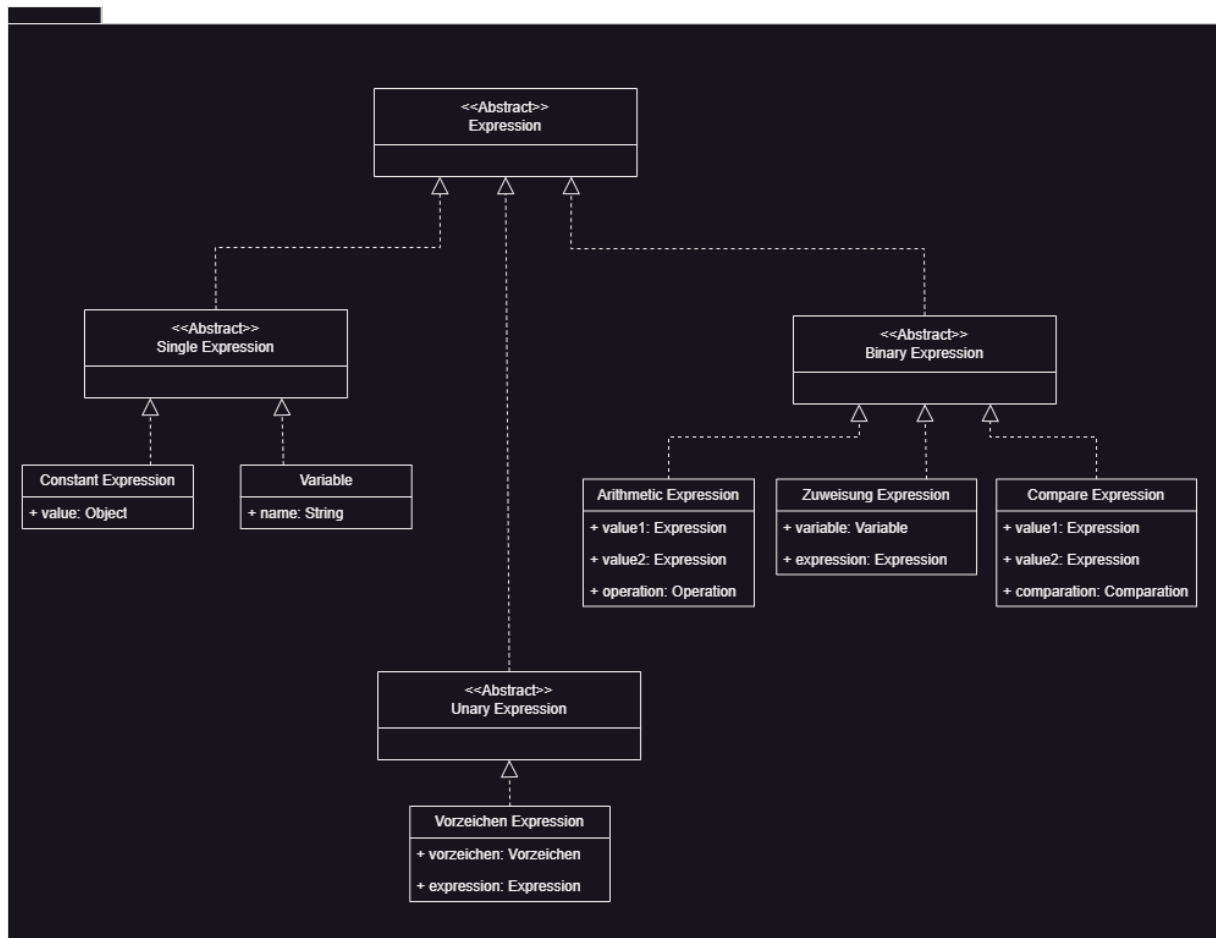


Abbildung 6: Abbildung des Metamodells der Expressions

Mit der Erstellung des Metamodells wurden nun die Regeln und Elemente, die für die zukünftige Erstellung von darauf basierenden Modellen zu beachten sind, definiert. Dabei wurde darauf geachtet die zu Beginn des Kapitels beschriebenen Richtlinien einzuhalten. So wurde sich bei der Darstellung der einzelnen Komponenten an bekannte Notationen der UML gehalten. Die Gruppierung in die einzelnen Übersichten sorgt für eine verständliche

und deutliche Übersicht auf die verschiedenen Komponenten. Des Weiteren wurde beachtet, dass das Metamodell lediglich die vorgegebenen Anforderungen umsetzt. Hierfür wurde immer wieder auf die entsprechenden Anforderungen aus dem vorherigen Teil der Arbeit referenziert.

Weitere Punkte der hier aufgeführten Entwicklungshinweise werden nochmals bei der Implementierung der Modellierungssprache in Kapitel 6 aufgegriffen.

## 5 Evaluierung von Werkzeugen zur Abbildung grafischer Modelle

Bevor anhand des erstellten Metamodells die DSL abgeleitet werden kann, werden nun im Folgenden einige Werkzeuge, die als Support-Tool für die Implementierung der Modellierungssprache in Frage kommen, analysiert. Zum Vergleich der betrachteten Werkzeuge werden diese nach einigen Kriterien, die dabei helfen eine Aussage über die Qualität und Abdeckung der Anforderungen durch die jeweilige Applikation zu treffen, bewertet. In der abschließenden Bewertung und Auswertung lassen sich dann die am besten abschneidenden Werkzeuge benennen.

### 5.1 Kriterien zum Vergleich der Werkzeuge

Bevor die Werkzeuge vorgestellt und anschließend verglichen werden, erfolgt in diesem Teil die Erarbeitung der Kriterien anhand derer die Tools später bewertet werden.

In der Literatur finden einige relevante Anforderungen an Werkzeuge oder Plattformen für die Metamodellierung Erwähnung. Darunter fallen allgemeine und ersichtliche Punkte wie die Unterstützung beim Entwurf und der Entwicklung des Geschäftsmodells oder auch die Evaluierung der genutzten Ressourcen [BP07]. Die Ermöglichung der effektiven Arbeit mit dem Modell wird im Paper zur domänenspezifischen Modellierung bei *Polar* noch weiter spezifiziert. Genannt werden hierbei unter anderem die Wiederverwendbarkeit von Teilen des Modells, die Umstrukturierung und das Ersetzen von Modellelementen, die Organisation mehrerer Modelle sowie die Bereitstellung gängiger Modellierungssprachen. Ebenso wird auf den Punkt zur Evaluierung der genutzten Ressourcen eingegangen. Dabei wird gefordert, dass eine Liste der verfügbaren Elemente im Tool angezeigt werden sollte. [KTK09] Die Möglichkeit zur Wiederverwendung beinhaltet beispielsweise die Referenzierung auf existierende Modelle, die Einbindung bestehender Komponenten oder die Kombination verschiedener Instanzen des gleichen Modellierungsobjekts. [PK02]

Weitere relevante Kriterien nennt David Granada in seinem Toolvergleich zu Editoren für die grafische Modellierung. Ein Beispiel ist die Lizenzierung der Anwendung. Dabei wird die Frage gestellt, ob es sich um ein kommerzielles oder quelloffenes Produkt handelt. Ein weiterer Punkt ist die Nutzbarkeit. Hier werden zur Bewertung der Grad der Dokumentation sowie die Anzahl vorhandener Beispiele herangezogen. Hinzu kommt die Unterscheidung zwischen abstrakter und konkreter Syntax. Die abstrakte Syntax behandelt die Definition des Metamodells. Die konkrete Syntax hingegen definiert die grafische Notation. Es ist darauf zu achten, dass die abstrakte Syntax nicht von der konkreten Syntax abhängt. [Gra16]

Viele der aufgeführten Anforderungen lassen sich auch in den Qualitätsmerkmalen nach *DIN/ISO-25010* finden. Diese werden im Folgenden als Grundlage für die Kriterien zur Bewertung der Werkzeuge herangezogen. Dazu gehören unter anderem Funktionale Eignung, Zuverlässigkeit, Wartbarkeit oder Übertragbarkeit. Diese Überbegriffe lassen sich dabei noch in weitere hierarchische Eigenschaften verfeinern. [Sta20] Die Anforderungen und Kriterien zur Einordnung der Werkzeuge werden diesen Merkmalen zugeordnet.

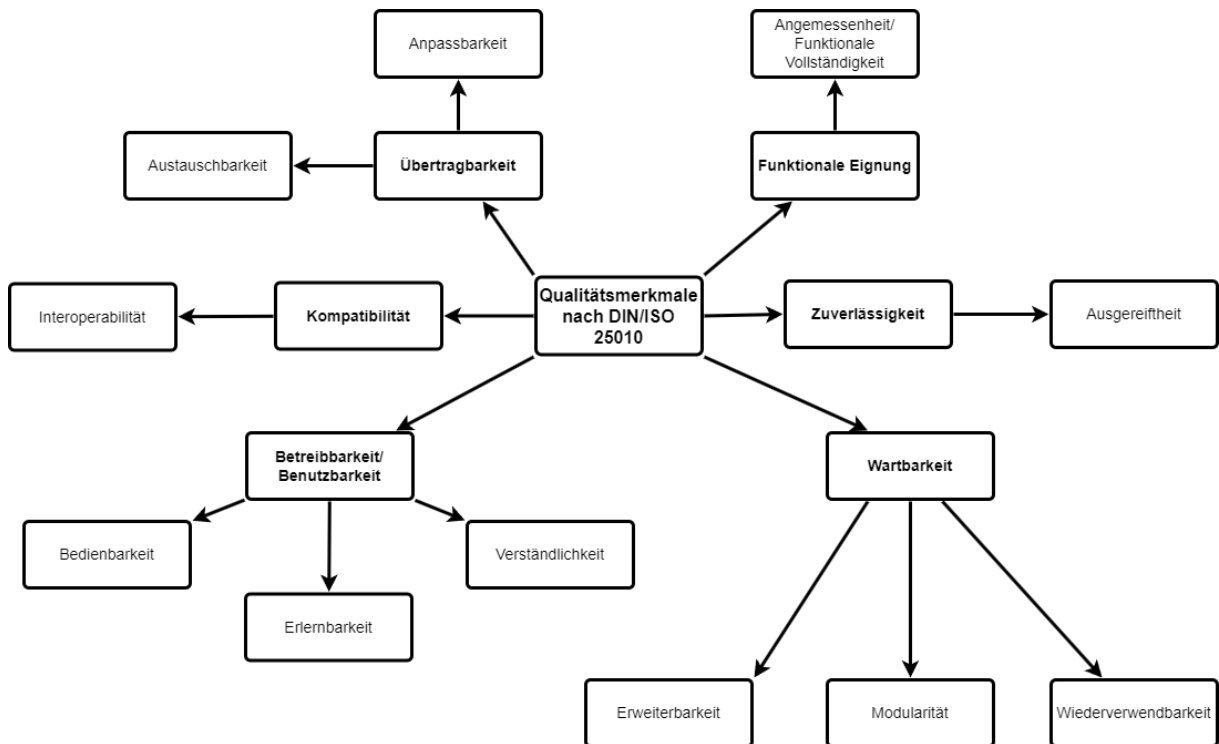


Abbildung 7: Auswahl der zentralen Qualitätseigenschaften nach [Sta20]

In der vorstehenden Abbildung 7 ist ein Auszug der verfeinerten Qualitätsanforderungen mit ihren zugehörigen Qualitätsmerkmalen dargestellt. Auf eine genauere Betrachtung der in anderen Anwendungsfällen wichtigen Qualitätsmerkmale Effizienz und Sicherheit wird in dieser Arbeit verzichtet. Die Kriterien werden diesen, für die Bewertung relevanten Anforderungen, mit einer passenden Metrik zugeordnet (siehe Tab. 3), um die Messbarkeit und spätere Vergleichbarkeit der Kriterien zu gewährleisten. Als Metrik kommen sowohl eine Nominalskala als auch eine Ordinalskala zum Einsatz. Bei der Nominalskala handelt es sich für die Kriterien lediglich um die Unterscheidung zwischen erfüllt und nicht erfüllt. Die Ordinalskala ermöglicht es hingegen verschiedenen Abstufungen auszudrücken, sodass gewonnene Erkenntnisse genauer eingeordnet werden können. Zuletzt gibt es auch Kriterien, die in absoluten Werten ausgedrückt werden können. Hierzu gehören beispielsweise die Lizenzkosten. Diese sind in keinem Qualitätsmerkmal der Norm enthalten, werden



jedoch den Anforderungen für den späteren Betrieb zugeordnet.

Anforderung	Kriterium	Metrik
Erweiterbarkeit	Aufwand beim Hinzufügen neuer Komponenten zu einem Modell	Ordinalskala (gering, mittel, hoch)
Wiederverwendbarkeit	Wiederverwendung von bestehenden Komponenten	Ordinalskala (sehr gut, gut, gering)
Wiederverwendbarkeit	Referenzierung auf bestehende Modelle	Ordinalskala (sehr gut, gut, gering)
Modularität	Gruppierung von Modellen in einzelne Module/Komponenten	Ordinalskala (sehr gut, gut, gering)
Funktionale Vollständigkeit	Import- und Export-Funktion	Nominalskala (gegeben, nicht gegeben)
Funktionale Vollständigkeit	Grafische Darstellung von Abläufen	Nominalskala (gegeben, nicht gegeben)
Funktionale Vollständigkeit	Evaluierung der genutzten Ressourcen im Modell	Nominalskala (sehr gut, gut, gering)
Funktionale Vollständigkeit	Unterscheidung zwischen abstrakter und konkreter Syntax	Nominalskala (gegeben, nicht gegeben)
Anpassbarkeit	Aufwand beim Anpassen des Metamodells oder der DSL	Ordinalskala (gering, mittel, hoch)
Anpassbarkeit	Aufwand bei der Umstrukturierung von Modellelementen	Ordinalskala (gering, mittel, hoch)
Bedienbarkeit	Intuitive Bedienung	Ordinalskala (sehr gut, gut, gering)
Bedienbarkeit	Organisation von mehreren Modellen	Ordinalskala (sehr gut, gut, gering)
Verständlichkeit/ Erlernbarkeit	Dokumentation	Ordinalskala (ausführlich, vorhanden, nicht vorhanden)
Verständlichkeit/ Erlernbarkeit	Support durch den Anbieter	Ordinalskala (sehr gut, gut, gering)
Ausgereiftheit	Beständigkeit und Etablierung des Anbieters auf dem Markt	Ordinalskala (sehr gut, gut, gering)
Betrieb	Lizenzkosten	absoluter Wert

Tabelle 3: Zuordnung der Qualitätsanforderungen zu den Kriterien mit ihrer Metrik

## 5.2 Analyse einer Auswahl an Werkzeugen

Bei der Auswahl der Kandidaten liegt der Fokus besonders auf den Werkzeugen, die bereits bei der Recherche verwandter Arbeiten erwähnt wurden. Diese sind somit bereits erprobt und können in die engere Auswahl genommen werden. Die vier Tools MetaEdit+ , ADOxx, EMF und MPS werden nun vorgestellt. Dabei werden ihre grundlegenden Funktionalitäten erläutert und das Vorgehen für eine erste beispielhafte Nutzung skizziert.

### 5.2.1 MetaEdit+

MetaEdit+ ist eine Anwendung zur Implementierung von Modellierungssprachen sowie der anschließenden Modellierung. Durch den Fokus auf das Erstellen von Sprachen können die Vorteile der Nutzung einer DSL gut genutzt werden. So lässt sich zum Beispiel die Produktivität steigern, weil das Endprodukt direkt auf den erstellten Spezifikationen basiert. MetaEdit+ verspricht des Weiteren hohe Flexibilität und eine einfache Reaktion auf Änderungen, da die Anpassungen lediglich auf Domänenebene erfolgen und nicht die Codebasis betroffen ist. Geworben wird auch mit der bereits bekannten Einung des Entwicklungsteams. Die Expertise kann breit verteilt werden, sodass auf eine Trennung zwischen Domäne und Code verzichtet werden kann. [metb]

In MetaEdit+ wird nach dem *GOPRR* Modellierungsansatz gearbeitet. Dies steht für *Graph*, *Object*, *Point/Port*, *Property*, *Role* und *Relationship*. Ein Graph ist dabei die übergeordnete Sammlung von Objekten und Beziehungen. Objekte bilden die grundlegenden Elemente des Graphen. Die Verbindungen zwischen den Objekten werden durch Beziehungen (*Relationship*) beschrieben. Die Rollen wiederum zeigen die Verbindung zwischen einem Objekt und einer Beziehung an und ein Punkt ist der Port eines Objekts, der mit der Beziehung verbunden ist. Dieses Zusammenspiel zwischen Rollen, Ports und Beziehungen bei der Verknüpfung von Objekten ist in Abb. 8 zu sehen. [WWJM19]

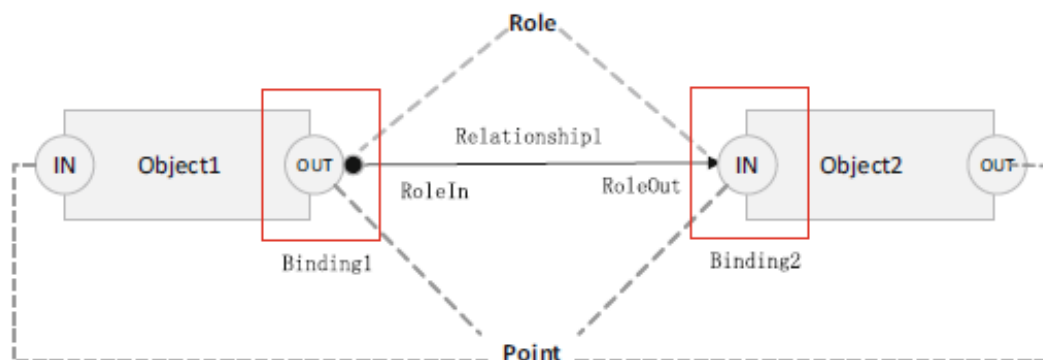


Abbildung 8: Verbindung zwischen Objekten [WWJM19]

Der letzte Typ der Eigenschaften (*Property*) bezeichnet Attribute zur Beschreibung der restlichen Typen [WWJM19]. In der Dokumentation von MetaEdit+ gibt es zum Einsatz von *GOPRR* ein Modellierungsbeispiel anhand eines Familienbaums (siehe Abb. 9). [metb]

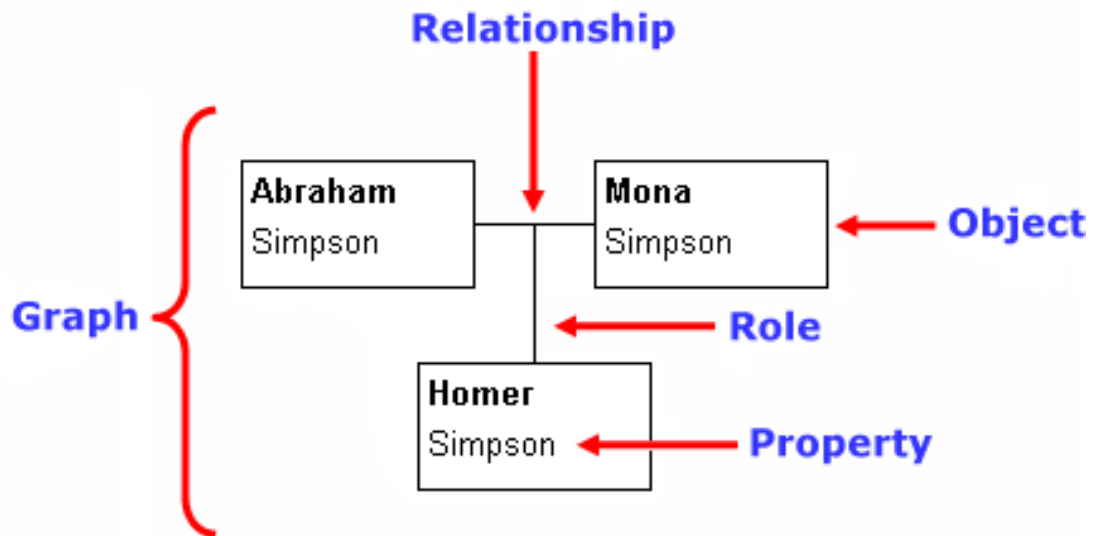


Abbildung 9: Einsatz von GOPRR am Modellierungsbeispiel *Familienbaum* [metb]

Bei der Erstellung einer Modellierungssprache über die Oberfläche wird im ersten Schritt ein Graph angelegt. Diesem können bereits übergeordnete Eigenschaften, wie zum Beispiel der Name des Modells, zugewiesen werden. Unter dem Reiter *Types* erfolgt dann der Aufbau der Objekte, Beziehungen und Rollen (siehe Abb. 10).

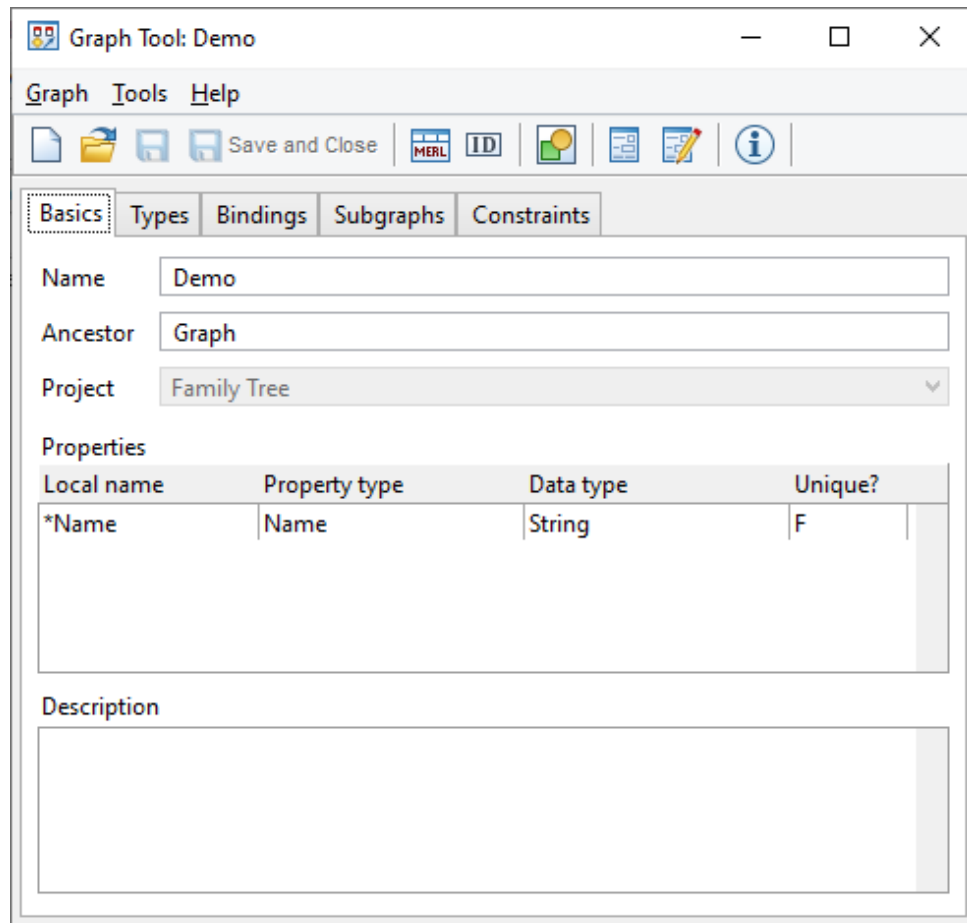


Abbildung 10: Erstellung eines Graphen in MetaEdit+

In Abbildung 11 ist die Maske zur Erstellung eines Objektes abgebildet. In diesem beispielhaften Fall wird die *Page* mit ihren Attributen Namen und Beschreibung (*description*) erstellt.

The screenshot shows a dialog box titled "Object Tool: Page". It has a menu bar with "Object", "Tools", and "Help". Below the menu bar is a toolbar with icons for file operations, an "ID" button, and an information icon. The main area contains the following fields and sections:

- Name:** A text field containing "Page".
- Ancestor:** A dropdown menu showing "Object".
- Project:** A dropdown menu showing "Family Tree".
- Properties:** A table with the following data:

Local name	Property type	Data type	Unique?
*description	description	String	F
Name	Name	String	F

Below the table is a section labeled "Description" with a large empty text area.

Abbildung 11: Erstellung eines Objekts in MetaEdit+

Des Weiteren kann für das Objekt ein Symbol hinterlegt werden. Hierzu kann bei der Auswahl des Objekts der *Symbol Editor*, in dem der Style sowie ein Bereich für spätere Verknüpfungen festgelegt wird, aufgerufen werden. In Abbildung 12 ist dies für das Symbol zum Objekt *Page* zu sehen. Die Auswahl ist dabei auf ein Rechteck gefallen. Dieses enthält im Zentrum den Wert des Attributs *Name*.

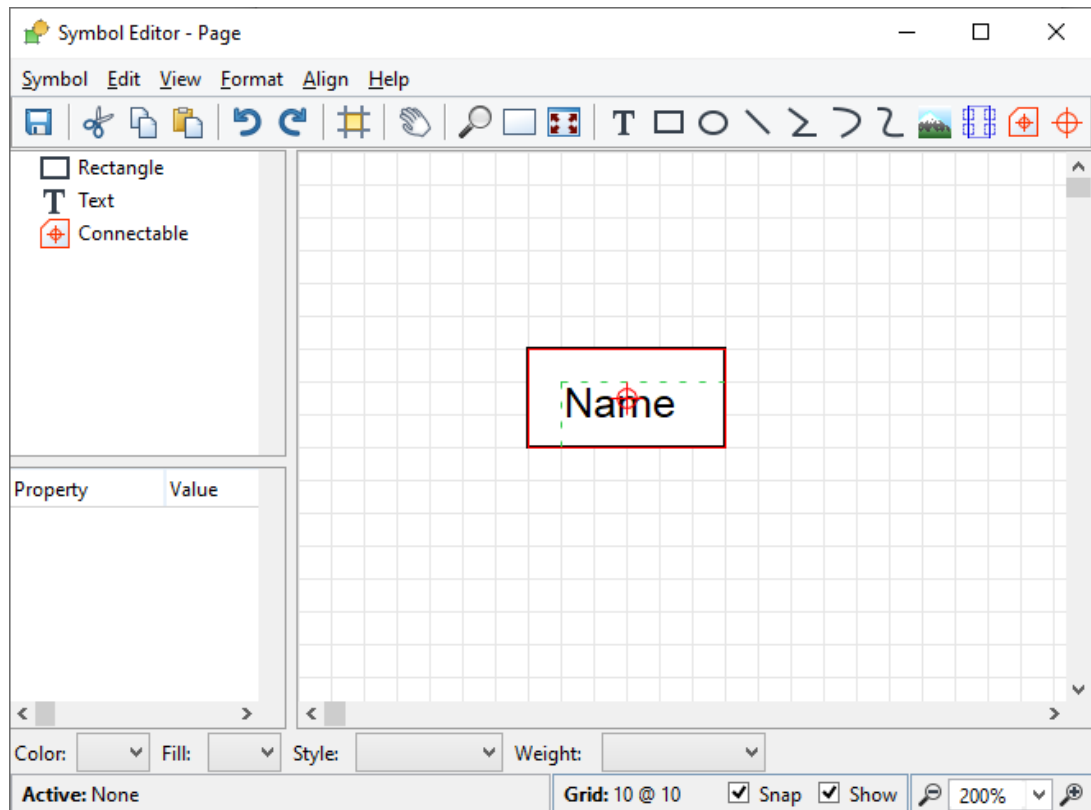


Abbildung 12: Symbol Editor zu einem Objekt in MetaEdit+

Für die Verbindungen im Graphen gibt es den Tab *Bindings*. Dort werden die Regeln definiert inwieweit die Instanzen der Elemente miteinander verbunden werden können. Jede Verbindung besteht dabei aus einer Beziehung, zwei oder mehreren Rollen sowie zu jeder Rolle ein oder mehrere Objekte. Für die Rollen kann zusätzlich eine Kardinalität angegeben werden. Diese bestimmt die mögliche Anzahl der Beziehungen für dieses Objekt (siehe Abb. 13). [metb]

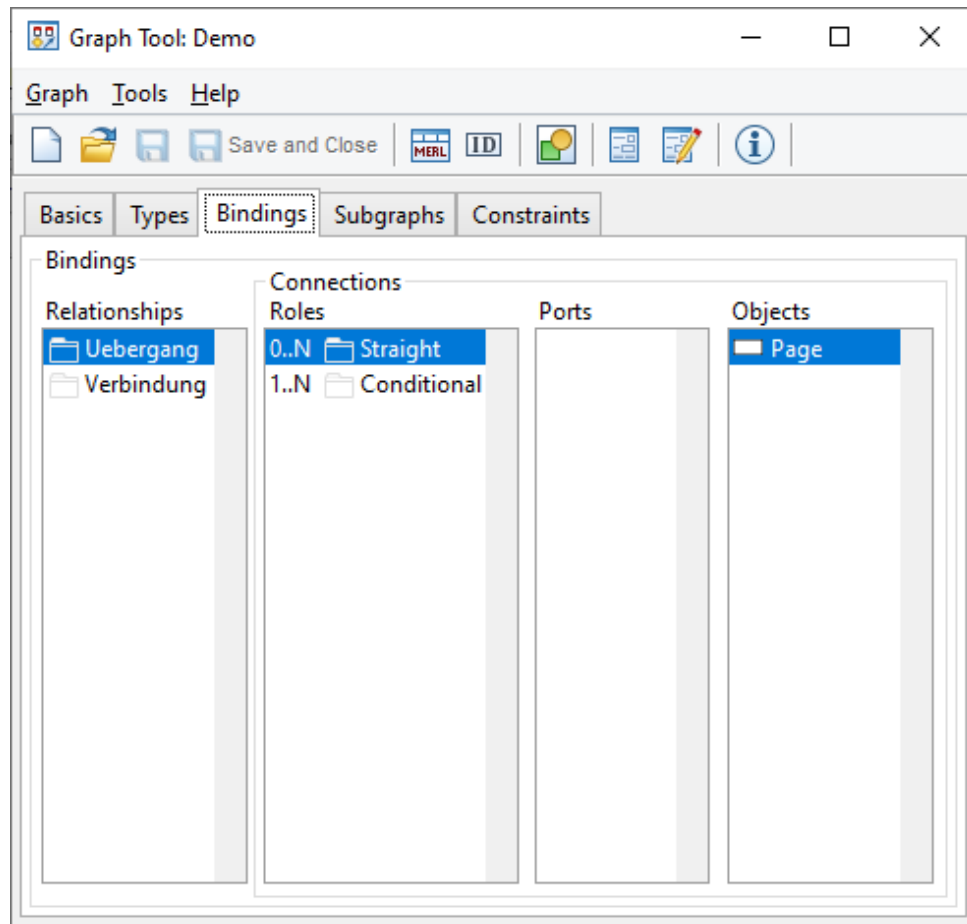


Abbildung 13: Verbindungen in MetaEdit+

Bei der Erstellung eines Modells kann dann auf die angelegten Elemente zugegriffen werden. Diese stehen in einer Menüleiste zur Auswahl und können in die freie Fläche gezogen werden (siehe Abb. 14).

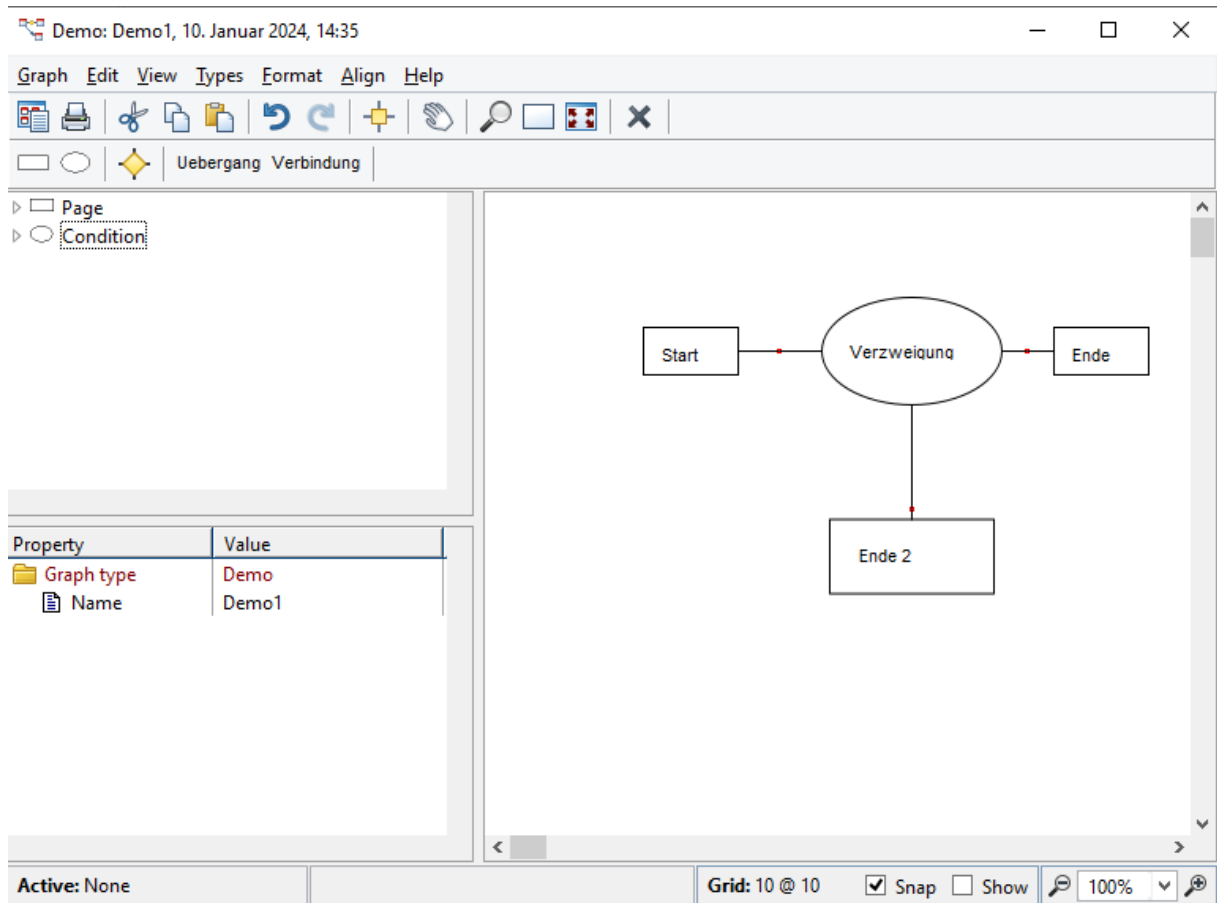


Abbildung 14: Beispielhaftes Modell in MetaEdit+



### 5.2.2 ADOxx

ADONIS ist ein klassisches Tool des Prozessmanagements und zur Bearbeitung von Abläufen geeignet. Es basiert auf der Metamodellierungsplattform ADOxx. Diese Plattform kann als eigenes Modellierungstool für die Implementierung einer domänenspezifischen Modellierungssprache genutzt werden. Hierfür stehen ein Modelling- und Development-Toolkit für die Erstellung eines eigenen Modellierungsbaukasten zur Verfügung. Das Development-Toolkit dient als grafische Oberfläche und unterstützt somit den Konfigurationsansatz der verschiedenen Plattformkomponenten, sodass auf den komplexeren Ansatz mittels Programmierung unter dem Einsatz der ADOxx Library Language (ALL) verzichtet werden kann. [ado]

Die ADOxx Bibliothek enthält mit Modelltypen, Klassen, Attributen und Relationen vier grundlegende Elemente. Modelltypen bilden eine Sammlung an Klassen und Relationen eines Metamodells. Klassen wiederum dienen als Template zur Erstellung eines Objekts dieser Klasse. Die Eigenschaften eines Modells, Objekts oder einer Relation werden Attribut genannt. Jedes Attribut hat einen Typen und einen Wert. Für die Erstellung von Beziehungen zwischen den Objekten werden Relationen genutzt. Eine Relation wird zwischen zwei Klassen definiert und ist immer eine gerichtete Verbindung. Die Zusammenhänge zwischen den beschriebenen Definitionen sind ebenfalls in der nachstehenden Abbildung veranschaulicht (siehe Abb. 15). [ado]

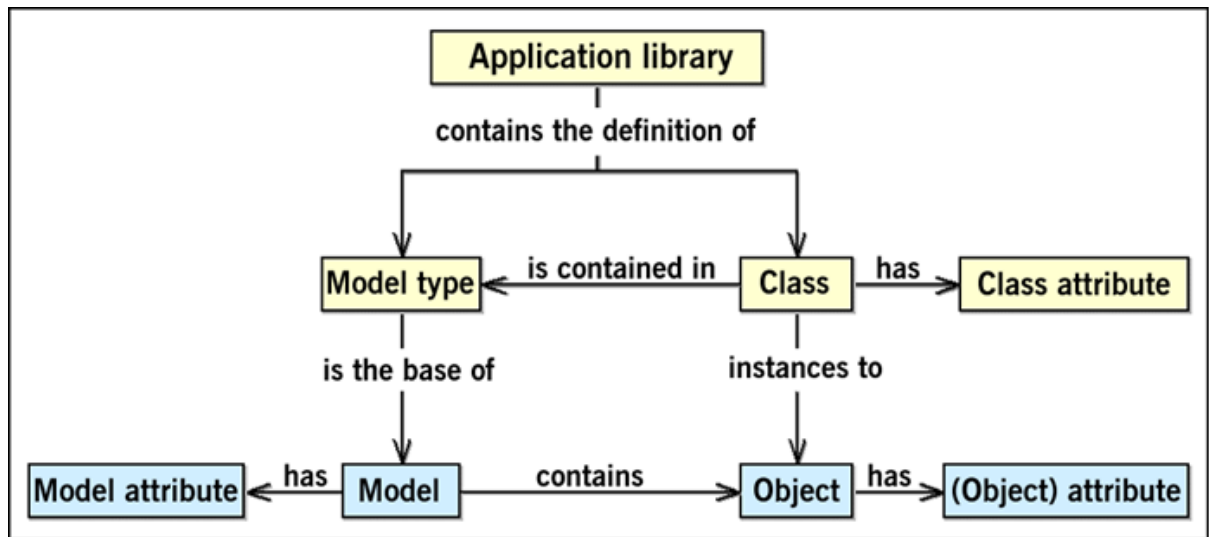


Abbildung 15: Zusammenhänge zwischen den grundlegenden Elementen der ADOxx Bibliothek [ado]

Bei der Erstellung der Klassen gilt es die drei Arten von Klassen zu beachten. So gibt es bereits vordefinierte abstrakte Klassen, die sich von ADOxx Metamodellklassen ableiten. Sie ermöglichen es die ADOxx Funktionalitäten zu nutzen. Selbstdefinierte abstrakte Klassen können für die Strukturierung des Metamodells genutzt werden. Sie erben entweder von der Rootklasse oder einer anderen Klasse des Metamodells und enthalten beispielsweise Attribute, die sie an Unterklassen vererben. Die dritte Art sind konkrete Klassen, die bestimmte Modellierungsklassen beschreiben, welche für die Modellierungssprache genutzt werden und damit die Realisierung des Metamodells ermöglichen. Sie erben die Semantik und Attribute ihrer zugehörigen (vordefinierten) abstrakten Klasse. [ado]

Neben möglichen Attributen wird innerhalb einer konkreten Klasse auch die grafische Darstellung angegeben. Dies geschieht in ADOxx mit *GraphRep* als grafische Notationssprache. Dabei können unter anderem Form, Größe, Position, Layer und Style angegeben werden.

Listing 1: GraphRep für eine Klasse in ADOxx

---

```

1  GRAPHREP
2  PEN w:0.05cm
3  RECTANGLE x:-2cm y:-.5cm w:4cm h:1cm
4  ATTR "Name" x:0cm y:0cm w:c:3.5cm h:c:1cm line-break:rigorous

```

---

Im hier dargestellten Beispiel handelt es sich um ein Rechteck, das den Wert des Attributs *Name* enthält (siehe Lst. 1). Eine Instanz dieser Klasse würde dann wie folgt aussehen (siehe Abb. 16).

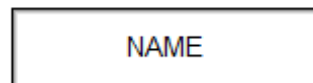


Abbildung 16: Darstellung einer Klasseninstanz mit der grafischen Repräsentation aus Lst. 1

Für die Definition der Relationen werden sogenannte Relationsklassen definiert. Diese enthalten neben einem Namen immer die Klasse, von der die Verbindung weggeht, sowie die Klasse, zu der die Relation zeigt. Der Relation kann dann analog zur Klasse eine passende grafische Darstellung mitgegeben werden.

Nachdem einige beispielhafte Klassen und Relationen im Development-Toolkit angelegt wurden, kann darauf im Modelling-Toolkit zugegriffen und mit der Modellierung gestartet werden. Am Rand sind die verfügbaren Elemente abgebildet. Sie können in die Arbeits-

fläche gezogen und dort ausgerichtet sowie miteinander verbunden werden (siehe Abb. 17).

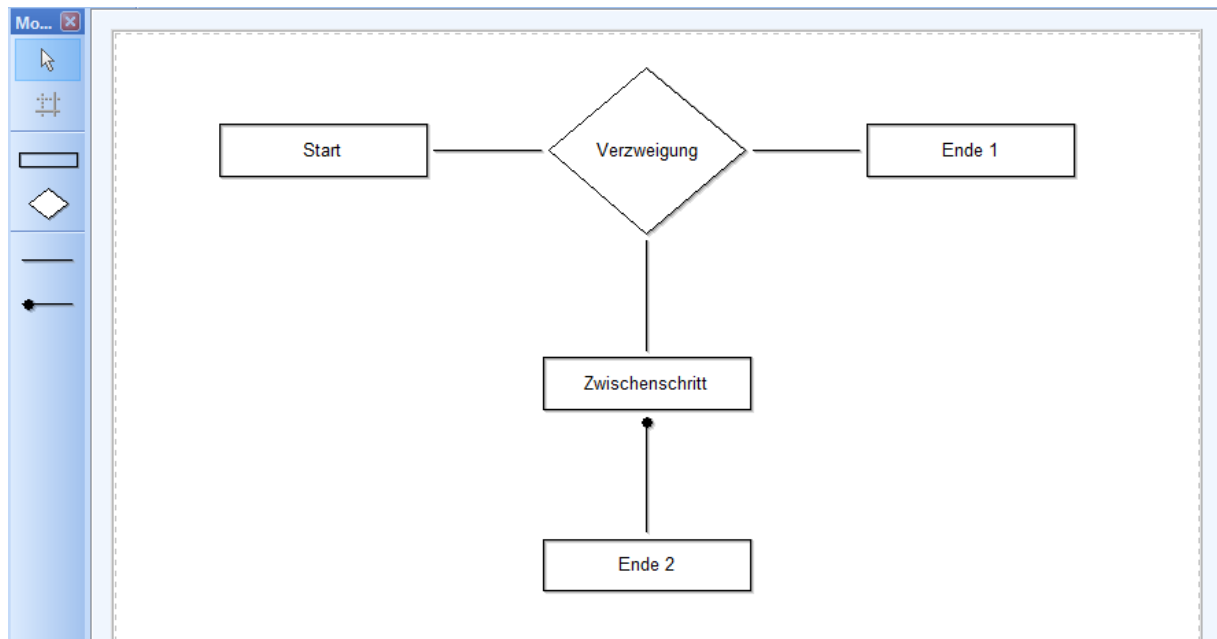


Abbildung 17: Ansicht im Modelling-Toolkit

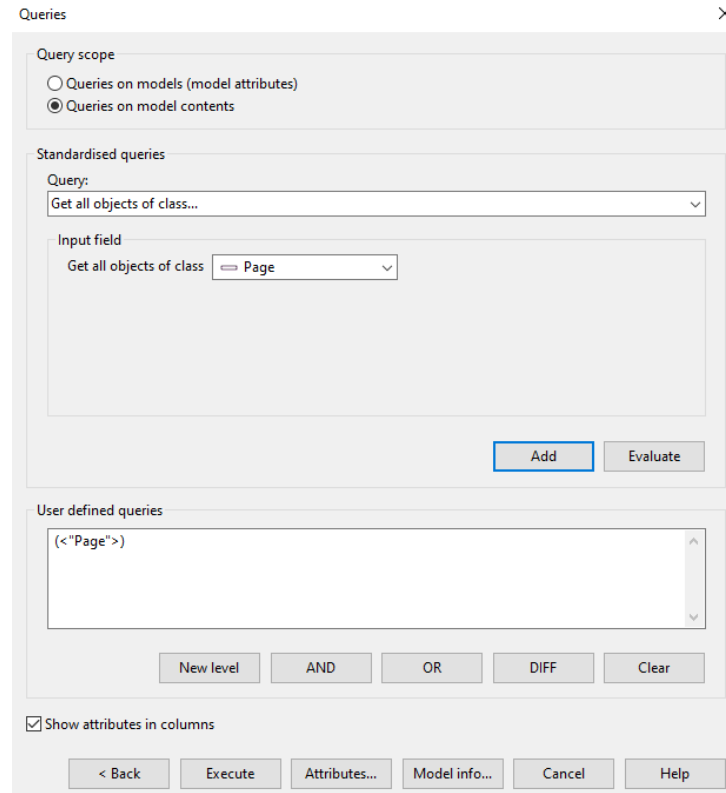
Durch den Doppelklick auf eine Element lässt sich die Detailansicht öffnen. Hier lassen sich unter anderem die Werte für klassenspezifische Attribute einstellen (siehe Abb. 18).

The screenshot shows the 'Start (Page)' detail view. It contains two input fields:

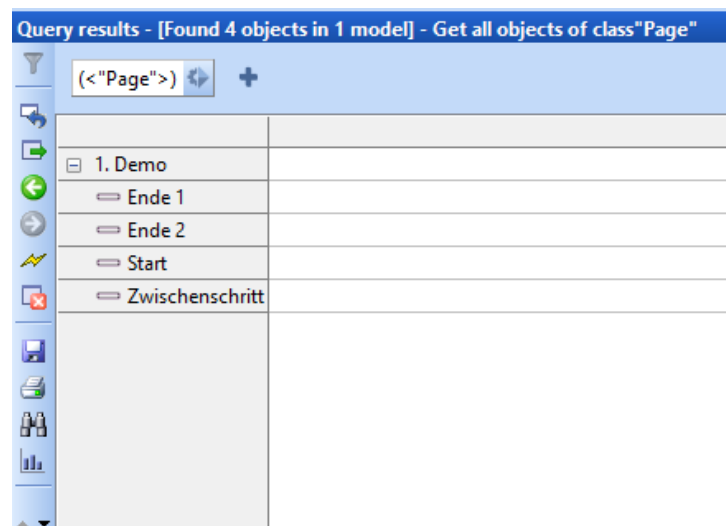
- A field labeled "Name:" with the text "Start" entered.
- A field labeled "description:" with the text "Hier geht es los" entered.

Abbildung 18: Informationen einer Klasse bestehend aus dem Namen und dem Attribut *description*

ADOxx bietet im Modelling-Toolkit auch vordefinierte Evaluierungsanfragen an. Diese können genutzt werden, um Auswertungen über das Modell zu erstellen. Zusätzlich können die Abfragen auch mit der ADOxx Query Language (AQL) selbstständig erweitert werden. In der Anwendung kann hierzu beispielsweise eine der vorgegebenen Queries ausgewählt werden. Im hier abgebildeten Fall werden alle Objekte einer bestimmten Klasse abgefragt (siehe Abb. 19).

Abbildung 19: Abfrage aller Objekte der Klasse *Page*

Als Ergebnis erscheinen die vier modellierten Objekte der gesuchten Klasse (siehe Abb. 20). Vor allem bei größeren Modellen mit einer Vielzahl an Elementen kann diese Möglichkeit für Auswertungen auf dem Modell helfen einen Überblick zu behalten oder auch das Modell zu verifizieren.

Abbildung 20: Ergebnis der Abfrage nach allen Objekten der Klasse *Page*

### 5.2.3 Eclipse Modeling Framework (EMF)

Beim Modellierungsframework von Eclipse liegt der Fokus hauptsächlich auf der Codegenerierung anhand von Modellspezifikationen. Begünstigt durch den Start von Eclipse als Java-Entwicklungsumgebung setzt ein Großteil von EMF auf die Modellierung mittels Code. Dies steht im Gegensatz zur Anforderung eine grafische Modellierungssprache zu entwickeln. Des Weiteren stellte David Granada in seinem Vergleich von Werkzeugen zur Erstellung grafischer Modellierungseditoren fest, dass die meisten Erweiterungen oder Frameworks von Eclipse vor allem in der breiten Abdeckung aller Kriterien nicht mit MetaEdit+ mithalten können. Dabei wurden die Tools unter anderem nach Umfang, Automatisierung, Nutzbarkeit und methodische Grundlage bewertet. Vor allem bei Letzterem konnten nur wenige mithalten. Hierbei wurde beurteilt inwieweit die bereitgestellten Funktionalitäten von wissenschaftlichen Theorien und Methoden abgeleitet sind. Die Ergebnisse der zitierten Auswertung sind in der nachfolgenden Übersicht abgebildet. [Gra16]

<i>Tools</i>	<i>Scope</i>	<i>Frame- work</i>	<i>Abs. syntax</i>	<i>Conc. syntax</i>	<i>Syntax distinct.</i>	<i>Editing</i>	<i>Models</i>	<i>Auto- mation</i>	<i>Usabi- lity</i>	<i>Meth. basis</i>
<b>Diagen</b>	OS (GPL)	Eclipse	Ecore/ UML	DiaMeta Design	No	✓✓	✓✓	✓✓	✓✓✓	-
<b>Eugenia</b>	OS (EPL)	Eclipse	Ecore	EOL	Yes	✓✓	✓✓✓	✓✓✓	✓✓✓	-
<b>GMF</b>	OS (EPL)	Eclipse	Ecore	Draw2D	Yes	✓✓✓	✓✓✓	✓✓	✓✓	-
<b>Graphiti</b>	OS (BSD)	Eclipse	Ecore/ Java	Draw2D	Yes	✓✓✓	✓✓✓	✓✓✓	✓✓	-
<b>MetaEdit+</b>	Com	Own	GOP- PRR	Internal API	Yes	✓✓✓	✓✓	✓✓✓	✓✓✓	✓
<b>Obeo Designer</b>	Com	Eclipse	Ecore	Odesign	Yes	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓
<b>Sirius</b>	OS	Eclipse	Ecore	Odesign	Yes	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓
<b>Tiger</b>	OS (GPL)	Eclipse	AGG	Shape- Fig	No	✓	✓✓	✓	✓✓✓	-
<b>% of compliance of the qualitative criteria:</b>						<b>83.3%</b>	<b>87.5%</b>	<b>83.3%</b>	<b>91.6%</b>	<b>12.5%</b>
Legend (for weightable fields): None (-), Poor (✓), Good (✓✓), Excellent (✓✓✓)										

Abbildung 21: Auswertung des Vergleichs von Werkzeugen zur Erstellung grafischer Modellierungseditoren [Gra16]

Da der Obeo Designer sowie Sirius als die zwei Ausnahmen in allen Kategorien mithalten können und sogar bei der Anwendung von bekannten Modellen leicht besser abschneiden, wird im Folgenden überprüft, ob es eine Eignung bezüglich der Fähigkeit zur grafischen

Modellierung gibt. Dies wird anhand des Obeo Designers überprüft, da er auf dem Sirius Eclipse Projekt aufbaut und somit einen Großteil der Funktionalitäten vereint.

Ähnlich zu den bereits vorgestellten Werkzeugen können über die Oberfläche der Anwendung Elemente erstellt werden. Im dafür vorgesehenen Spezifikationseditor werden die einzelnen Elemente der Modellierungssprache angelegt. Jedem Knoten und jeder Relation wird hierbei direkt ein Symbol zugeordnet. Für die Symbole können Styleattribute wie Farbe, Form und Größe angegeben werden. Ein Beispiel des Spezifikationseditors ist in Abbildung 22 zu sehen.

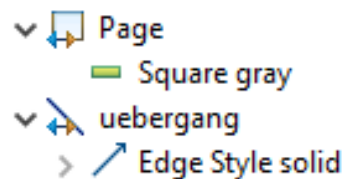


Abbildung 22: Beispielhafter Ausschnitt aus dem Spezifikationseditor

Im weiteren Vorgehen treten nun immer mehr Unterschiede zu den zwei bisher betrachteten Tools auf. So muss zu jedem Knoten eine Domänenklasse verlinkt werden (siehe Abb. 23).

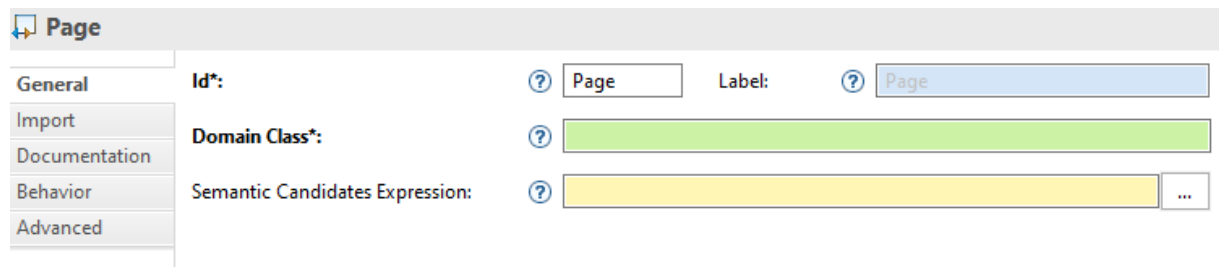


Abbildung 23: Konfiguration des Knoten *Page*

Diese kann dann weitere Konfigurationen sowie Attribute und Funktionalitäten für den Knoten enthalten. An dieser Stelle schließt sich der Kreis zur bereits angedeuteten Codenähe. Im Vergleich zu ADOxx und MetaEdit+ ist die Erstellung einer grafischen Modellierungssprache mit dem Obeo Designer nicht vollumfänglich ohne Programmierkenntnisse möglich. Es werden zusätzlich zu der Konfiguration über die Benutzeroberfläche auch Codefragmente benötigt.

### 5.2.4 Meta Programming System (MPS)

Als Alternative zu EMF wurde noch das Äquivalent von JetBrains MPS betrachtet. Erwähnung findet es unter anderem in den Artikeln von Martin Fowler zu DSLs. [Fow05] Ähnlich zu den Werkzeugen aus dem Eclipse Umfeld basieren viele Konzepte in MPS auf der objektorientierten Programmierung, sodass ein Großteil der Modellierung auf Code beruht. Dabei besteht auch die Möglichkeit grafische Elemente zu erstellen. Hierfür muss zu Beginn das Konzept eines Canvas definiert werden. Dies dient als oberster Knoten aller grafischen Repräsentationen. Um auf dem Canvas Formen zu zeichnen, wird als nächstes ein abstraktes Konzept, *Shape* genannt, benötigt. Von diesem können dann die konkreten Formen abgeleitet werden. Für einen Kreis könnte dies beispielsweise wie folgt ausschauen (siehe Lst. 2).

Listing 2: Grafische Repräsentation eines Kreises in MPS

---

```

1  concept Circle extends Shape
2      implements <none>
3
4      instance can be root: false
5      alias: circle
6      short description: <without further specification>
7
8      properties:
9          x: integer
10         y: integer
11         radius: integer
12
13     children:
14         << ... >>
15
16     references:
17         << ... >>

```

---

Das neue Konzept *Circle* leitet von *Shape* ab und enthält im ersten Entwurf Werte für die Positionierung im Canvas (x, y) sowie die Größe des Radius. [mps]

Die Nutzung der definierten Objekte erfolgt, wie die Definition der Ablauflogik, textbasiert. Es gibt nicht die klassischen Funktionalitäten eines Zeichen- oder Modellierungswerkzeugs. MPS setzt dabei auf einen projektiven Editor, mit dem direkt auf dem Abstract Syntax Tree (AST) des Modells gearbeitet wird. [mps]

### 5.3 Bewertung der Werkzeuge

Nachdem mit MetaEdit+ , ADOxx, EMF und MPS vier Werkzeuge mit ihren Kernfunktionalitäten vorgestellt wurden, fällt bereits auf, dass sich die beiden letztgenannten deutlich von MetaEdit+ und ADOxx unterscheiden. Die Anwendungen für EMF und MPS basieren auf Entwicklungsumgebungen und legen damit bei der Nutzung einen großen Wert auf die Programmierung. Dies steht im Gegensatz zum Ziel beim Einsatz einer DSL die Diskrepanz zwischen Anforderungsbeschreibung und Implementierung zu verringern. Die grafischen Benutzeroberflächen von MetaEdit+ und ADOxx stellen hierbei eine deutlich geringere Einstiegshürde für die Umsetzung und den Einsatz der Modellierungssprache dar.

Da somit schon zwei der vier Kandidaten ausgeschlossen werden können, wird die detaillierte Bewertung im nächsten Abschnitt lediglich für MetaEdit+ und ADOxx dargelegt. Dabei werden die einzelnen Qualitätsanforderungen aus der Tabelle 3 aufgegriffen und für die beiden Werkzeuge die Erfüllung der zugeordneten Kriterien mittels der Metriken eingeordnet.

#### Erweiterbarkeit

In beiden Werkzeugen bietet die grafische Oberfläche einen umfangreichen Modellierungseeditor, der es ermöglicht weitere Komponenten zu einem Modell hinzuzufügen. Somit können bestehende Modelle mit geringem Aufwand erweitert werden (siehe Tab. 4).

Kriterium	MetaEdit+	ADOxx
Aufwand beim Hinzufügen neuer Komponenten zu einem Modell	gering	gering

Tabelle 4: Bewertung der Erweiterbarkeit

#### Wiederverwendbarkeit

Bei der Wiederverwendung von bestehenden Komponenten gibt es in beiden Tools keine Unterschiede. Es können sowohl bereits modellierte Teile in ein neues Modell übernommen werden und auch bei der Kombination verschiedener Instanzen des gleichen Objekts innerhalb eines Modells gibt es keine Einschränkungen.

Die Referenzierung auf bestehende Modelle oder auch über verschiedene Modelle hinweg ist ebenfalls möglich. Im Gegensatz zur einfachen Wiederverwendung von Komponenten sind hierzu jedoch tiefere Kenntnisse erforderlich. In ADOxx beispielsweise ist dafür der Einsatz des Attributs *InterRef* vorgesehen. Auf Grund der fortgeschrittenen Nutzung der Werkzeuge wird dieser Punkt in beiden Fällen mit gut bewertet (siehe Tab. 5).



Kriterium	MetaEdit+	ADOxx
Wiederverwendung von bestehenden Komponenten	sehr gut	sehr gut
Referenzierung auf bestehende Modelle	gut	gut

Tabelle 5: Bewertung der Wiederverwendbarkeit

### Modularität

Ein modularer Aufbau dient als Grundlage für eine Kapselung von zusammengehörigen Funktionalitäten und erlaubt dem Nutzer eine einfache Orientierung innerhalb des Modells. Aufbauend auf den Erkenntnissen der Wiederverwendbarkeit stehen sich die zwei Werkzeuge auch bei der Umsetzung der Modularität in nichts nach. Hilfreich ist dabei der *GOPRR* Ansatz von MetaEdit+ . ADOxx ist mit dem Einsatz von Modelltypen, Klassen, Attributen und Relationen ähnlich aufgestellt, sodass eine Gruppierung auf verschiedenen Ebenen möglich ist (siehe Tab. 6).

Kriterium	MetaEdit+	ADOxx
Gruppierung von Modellen in einzelne Module/Komponenten	sehr gut	sehr gut

Tabelle 6: Bewertung der Modularität

### Funktionale Vollständigkeit

Ein Teil der Kriterien, die unter dem Punkt funktionale Vollständigkeit behandelt werden, ergeben sich aus den Anforderungen aus 3.1. Wichtig ist die Import- und Export-Funktion, damit das exportierte Modell später interpretiert werden kann. In beiden Werkzeugen ist ein Import und Export im Extensible Markup Language (XML) Format möglich. In ADOxx kann der Export auch als ADOxx Development Language (ADL) Datei erfolgen, um sich weiterhin im ADONIS/ADOxx Kontext zu bewegen.

Die grafische Darstellung von Abläufen als eine der Grundvoraussetzungen ist gegeben. Wie eingangs festgestellt ist dies sowohl für die Entwicklung der DSL als auch bei der späteren Nutzung der Modellierungssprache elementar und hat somit für den Ausschluss von EMF und MPS gesorgt.

Bei der Evaluierung der genutzten Ressourcen gibt es erstmals Unterschiede in der Ausprägung zwischen MetaEdit+ und ADOxx. In erstgenannter Anwendung gibt es zu jedem Modell eine tabellarische Übersicht der modellierten Komponenten und ihrer Attribute. Diese Zusammenfassung als Tabelle bietet auch ADOxx. Darüber hinaus besteht bei ADOxx jedoch auch die Möglichkeit mittels der eigenen Abfragesprache AQL selbstdefinierte Evaluationen durchzuführen.

Als letzter Punkt wird die Unterscheidung zwischen abstrakter und konkreter Syntax betrachtet. Dabei steht im Fokus, dass die Definitionen des Metamodells nicht von der grafischen Notation, der konkreten Syntax, abhängen. Dies ist in beiden Werkzeugen der Fall. Wie in 5.2 beschrieben, können den Objekten Symbole zugewiesen werden ohne ihre Funktion zu beeinflussen (siehe Tab. 7).

Kriterium	MetaEdit+	ADOxx
Import- und Export-Funktion	gegeben	gegeben
Grafische Darstellung von Abläufen	gegeben	gegeben
Evaluierung der genutzten Ressourcen im Modell	gut	sehr gut
Unterscheidung zwischen abstrakter und konkreter Syntax	gegeben	gegeben

Tabelle 7: Bewertung der funktionalen Vollständigkeit

### Anpassbarkeit

In beiden Anwendungen ist eine hohe Anpassbarkeit gegeben. Muss beispielsweise ein Objekt um ein Attribut erweitert werden, so kann das neue Attribut in allen bereits bestehenden Modellen genutzt werden. Das Anpassen des Metamodells beziehungsweise der Sprache ist genauso wie die Umstrukturierung von Modellelementen mit einem geringen Aufwand möglich (siehe Tab. 4).

Kriterium	MetaEdit+	ADOxx
Aufwand beim Anpassen des Metamodells oder der DSL	gering	gering
Aufwand bei der Umstrukturierung von Modellelementen	gering	gering

Tabelle 8: Bewertung der Anpassbarkeit

### Bedienbarkeit

Dadurch, dass beide Anwendungen sowohl die Implementierung einer eigenen Modellierungssprache als auch das anschließende Erstellen von Modellen mit dieser Sprache ermöglichen, ist es im ersten Moment für unerfahrene Nutzer schwierig zwischen den beiden Handlungsfeldern zu unterscheiden. ADOxx versucht mit der Trennung in Modellierung- und Development-Toolkit etwas Klarheit herzustellen. Im Gegensatz zu MetaEdit+ wird damit jedoch ein Wechsel zwischen den Anwendungen notwendig. Losgelöst von dieser Problematik erfolgt die Modellierung in beiden Fällen jedoch intuitiv und nach bekannten Mustern.

Dies führt auch dazu, dass die Organisation von mehreren Modellen ohne Einschränkungen möglich ist. Die Modelle werden im Dateieditor angezeigt und können nach belieben auf verschiedenen Ebenen gruppiert werden (siehe Tab. 9).

Kriterium	MetaEdit+	ADOxx
Intuitive Bedienung	gut	gut
Organisation von mehreren Modellen	sehr gut	sehr gut

Tabelle 9: Bewertung der Bedienbarkeit

### Verständlichkeit/Erlernbarkeit

Beide Anbieter stellen eine ausführliche Dokumentation zu ihrer Anwendung bereit. In dieser werden die einzelnen Aspekte des Werkzeugs sowie die zugrunde liegenden Konzepte erläutert. Förderlich ist auch die Bereitstellung von Lernvideos und Praxisbeispielen, in denen exemplarische Umsetzungen Schritt für Schritt erläutert werden. Ebenso hervorzuheben ist das Forum von MetaEdit+ und auf der anderen Seite die Sammlung häufig gestellter Fragen von ADOxx. Hier lassen sich viele Informationen zu Detailfragen finden (siehe Tab. 10).

Kriterium	MetaEdit+	ADOxx
Dokumentation	ausführlich	ausführlich
Support durch den Anbieter	sehr gut	sehr gut

Tabelle 10: Bewertung der Verständlichkeit/Erlernbarkeit

### Ausgereiftheit

MetaEdit+ als Produkt des Unternehmens MetaCase wurde bereits in der ersten Version 1995 als beste Software zur Anwendungsentwicklung ausgezeichnet. MetaCase agiert bereits seit 1991 und ist mittlerweile eine der führenden Anbieter für Umgebungen zur domänenspezifischen Modellierung. [meta]

ADOxx wird von OMILAB betreut. Dabei handelt es sich um eine globale Gruppe, die sich mit der Unterstützung für konzeptionelle Modellierung beschäftigt. [omi] Des Weiteren setzt, wie bereits unter 3.2 erwähnt, die *BOC Group* als Anbieter der Modellierungsplattform ADONIS auf ADOxx als unterliegende Entwicklungsplattform und hält ihr Produkt damit seit über 20 Jahren auf dem Markt (siehe Tab. 11).

Kriterium	MetaEdit+	ADOxx
Beständigkeit und Etablierung des Anbieters auf dem Markt	sehr gut	sehr gut

Tabelle 11: Bewertung der Ausgereiftheit

## Betrieb

Für die Nutzung von MetaEdit+ ist nach einer kostenlosen Evaluierungsphase eine Lizenz notwendig. Diese kann in verschiedenen Modellen erworben werden. Für die Modellierung mit MetaEdit+ startet dies bei 250€ im Monat. Für die erweiterte Nutzung aller APIs erhöht sich der monatliche Betrag auf 295€. Der Preis für die vollumfängliche Nutzung der MetaEdit+ Workbench ist nur auf Anfrage erhältlich. Für die akademische Nutzung gibt es extra Lizenzen. Eine Lizenz kostet dann beispielsweise nur einmalig 150€ (siehe Abb. 24). [meta]

Number of licenses	Price (EUR)
1	150 €
10	500 €
20	1 000 €
50	2 000 €
>50	contact us

Abbildung 24: Preisstaffelung der akademischen Lizenzen [meta]

Da ADOxx von OMILAB als Open Source Plattform bereitgestellt wird, fallen keine zusätzlichen Lizenzkosten an (siehe Tab. 12). [omi]

Kriterium	MetaEdit+	ADOxx
Lizenzkosten	ab 150€	Open Source

Tabelle 12: Bewertung des Betriebs

Auf die Gegenüberstellung der beiden Anwendungen folgt nun die Auswertung der Ergebnisse. Ziel ist es die gesammelten Erkenntnisse anschaulich vergleichen und einordnen zu können. Für eine visuelle Darstellung eignet sich besonders gut ein Netzdiagramm. Die Achsen stellen dabei die neun betrachteten Qualitätsanforderungen dar. Die Achsen der einzelnen Qualitätskategorien bilden dabei den Gesamt-Erfüllungsgrad in Prozent ab. Um diesen Prozentwert zu erlangen, müssen die Ergebnisse der Evaluation in eine normalisierte Form, also Werte zwischen 0 und 1, gebracht werden. Im Fall einer dreistufigen Ordinalskala kann dies folgendermaßen aussehen.

<b>Ergebnisse</b>	<b>Normalisierung</b>
sehr gut	1
gut	0.5
gering	0

Tabelle 13: Normalisierung einer Ordinalskala mit 3 Werten

Für den absoluten Wert der Lizenzkosten kann die Normalisierung durch eine Abstufung der Ergebnisse erreicht werden. Eine Einordnung kann beispielsweise wie folgt ablaufen (siehe Tab. 14).

<b>Ergebnisse</b>	<b>Normalisierung</b>
0€	1
$\leq 100\text{€}$	0.8
$\leq 250\text{€}$	0.6
$\leq 500\text{€}$	0.4
$\leq 1000\text{€}$	0.2
$>1000\text{€}$	0

Tabelle 14: Normalisierung der Lizenzkosten

Die daraus resultierenden Diagramme (siehe Abb. 25 und Abb. 26) bieten einen Überblick über das Abschneiden der beiden Tools. Je näher die Werte der einzelnen Kategorien an 100% liegen, desto besser wurden die Anforderungen erfüllt.

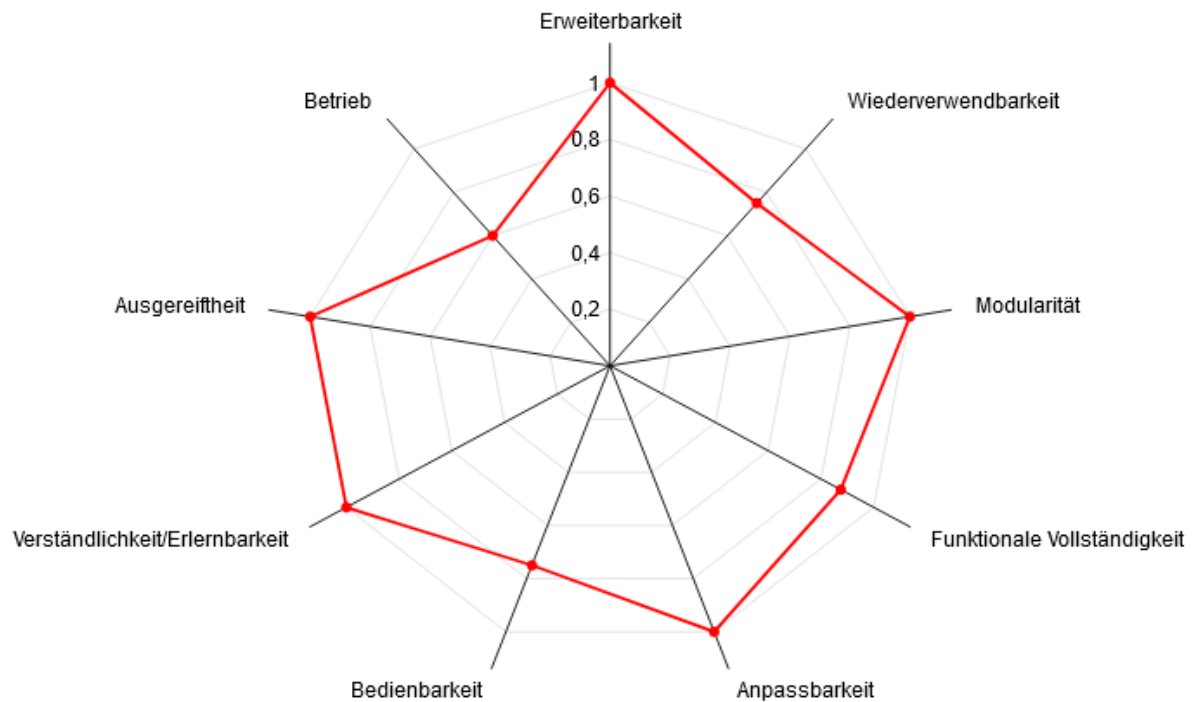


Abbildung 25: Netzdiagramm zur Bewertung von MetaEdit+

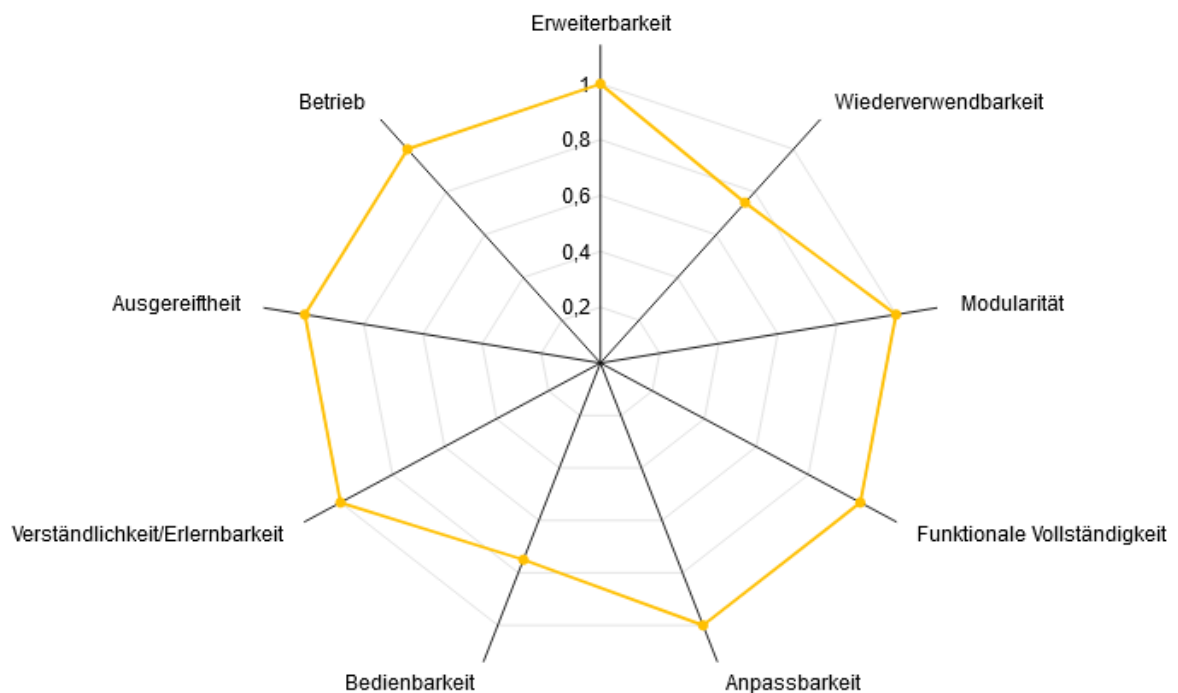


Abbildung 26: Netzdiagramm zur Bewertung von ADOxx

Abschließend ist zu erkennen, dass MetaEdit+ und ADOxx für die meisten Anforderungen eine ähnlich gute Abdeckung erreichen konnten. Lediglich in zwei Kategorien liegt ADOxx

vor MetaEdit+ . Im Falle der funktionalen Vollständigkeit konnte sich ADOxx im Bereich der Evaluierung von Modellen durch die AQL von MetaEdit+ leicht absetzen. Deutlicher ist der Unterschied im Bereich der Betriebskosten. Durch das Lizenzmodell ist es für MetaEdit+ nicht möglich mit ADOxx als Open Source Werkzeug mitzuhalten.

## 6 Implementierung der grafischen DSL

Da im vorherigen Kapitel 5.3 bei der Bewertung von Werkzeugen zur Unterstützung des Prozesses der Sprachentwicklung und anschließenden Modellierung ADOxx am Besten abgeschnitten hat, dient dieses Tool nun als Basis für die Implementierung der grafischen DSL. Die Beschreibung der Umsetzung mit ihren Besonderheiten erfolgt in diesem Abschnitt der Arbeit. Dabei wird auf die bereits beschriebenen Grundlagen von ADOxx (siehe Kap. 5.2.2) aufgebaut.

Als Klassen werden die beiden Navigationselemente *Page* und *Condition* angelegt. Die *Page* wird als Rechteck dargestellt. Die grafische Repräsentation einer *Condition* erfolgt als Raute und orientiert sich somit an der Notation bekannter Modellierungssprachen für Objekte, die eine Entscheidung beziehungsweise Verzweigung des Pfades abbilden. Für die lokalen Variablen einer Seite wird innerhalb der Klasse eine Variable *variables* vom Typ *Record* angelegt. Ein *Record* ist eine Tabelle zur Sammlung von Schlüssel-Wert-Paaren. Somit können bei der späteren Modellierung Variablen angelegt werden, indem die Tabelle durch ein Wertepaar bestehend aus Variablennamen und Variablenwert erweitert wird. Im Interpreter werden die Navigationselemente später durch ihren Namen identifiziert. Das zugehörige Attribut wird daher als Pflichtfeld definiert. Dies geschieht durch das Schlüsselwort *mandatory* (siehe Lst. 3).

Listing 3: Repräsentation der Attribute einer *Page*

---

```

1  NOTEBOOK
2
3  #—————
4  CHAPTER "General"
5  ATTR "Name" mandatory
6  ATTR "description"
7  ATTR "variables"

```

---

Der globale Speicher, im Metamodell als *Memory* abgebildet (siehe Abb. 3), muss auf Modellebene angelegt werden, sodass alle Klassen darauf zugreifen können. Hierfür gibt es in ADOxx sogenannte Modellattribute. Standardmäßig stehen hier Felder für den Autor, den Status oder die letzte Bearbeitung des Modells zur Verfügung. Das Hinzufügen eines neuen Modellattributs erfolgt in der vordefinierten abstrakten Klasse `__ModelTypeMetaData__`. Die Repräsentation des neuen Attributs erfolgt in einem eigenen *Chapter* (siehe Lst. 4). Dies sorgt dafür, dass im Modelling-Toolkit bei der Bearbeitung der Modellattribute für den globalen Speicher ein eigener Reiter angelegt wird.



Listing 4: Repräsentation der Modellattribute

---

```

1  NOTEBOOK
2
3  #-----
4  CHAPTER "General"
5  ATTR "Name" write-protected
6
7  #-----
8  CHAPTER "Memory"
9  ATTR "memory"
10
11 #-----
12 CHAPTER "Model properties"
13 ATTR "Model type"
14 ATTR "State"
15 ATTR "Reviewed by"
16 ATTR "Reviewed on"
17
18 #-----
19 CHAPTER "Model information"
20 ATTR "Author" write-protected
21 ATTR "Creation date" write-protected
22 ATTR "Last user" write-protected
23 ATTR "Date last changed" write-protected
24 ATTR "Number of objects and relations" write-protected

```

---

Für die Darstellung des Ablaufs spielen die Übergänge zwischen den Navigationselementen eine wichtige Rolle. Diese Beziehungen werden über Relationen abgebildet. Im Metamodell ist dies durch die Existenz der Ausgänge dargelegt (siehe Abb. 4 und Abb. 5). Für die Übernahme in die Modellierungssprache muss das Konzept des Ausgangs noch weiter heruntergebrochen werden. So besteht in ADOxx eine Relation immer aus Quell- und Zielklasse. Für den in dieser Arbeit behandelten Anwendungsfall ergeben sich somit drei Kombinationen.

- **Transition** *Page* zu *Page*
- **Conditional Transition** *Page* zu *Condition*
- **Forwarding** *Condition* zu *Page*

*Transition*, *Conditional Transition* und *Forwarding* sind die Spezialisierungen eines Ausgangs (siehe Abb. 27).

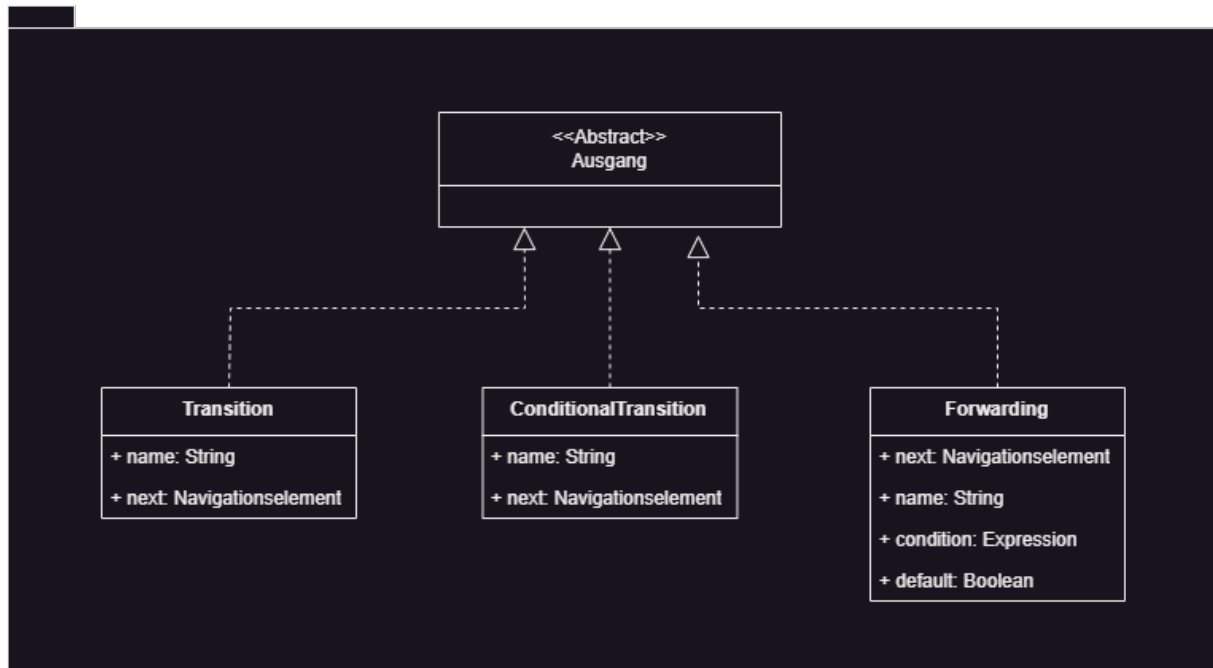


Abbildung 27: Arten eines Ausgangs

Eine *Transition* stellt den Übergang von einer Seite zur nächsten Seite ohne weitere Bedingungen dar. Die *Conditional Transition* verlässt eine *Page* und führt zu einer *Condition*. Hier ist ebenso keine weitere Logik erforderlich. Komplexer hingegen ist die dritte Übergangsart *Forwarding*. Ausgehend von einer *Condition* wird dieser Relation die Bedingung des Ausgangs zugeordnet (siehe Anforderung 12 aus Kapitel 3.1). Außerdem gibt der Wert *default* an, ob es sich hierbei um den Standardweg handelt. Da es in ADOxx keinen Datentyp für Wahrheitswerte gibt, wird auf den Zahlenwert zurückgegriffen. Dabei wird 0 als falsch und 1 als wahr interpretiert. Damit lediglich diese beiden Werte verwendet werden, erfolgt für diese Variable die Eingrenzung über folgende Regel (siehe Lst. 5).

Listing 5: Regel für Variablenwerte

```

1  DOMAIN
2  message:" Only attribute values 0 and 1 are allowed."
3  INTERVAL
4  lowerbound:0
5  upperbound:1
  
```

Eine weitere Verbesserung lässt sich innerhalb der Repräsentation der Attribute für die Relation *Forwarding* erzielen. So wird hierfür das Attribut *default* mit dem Kontrolltyp

*check* angegeben, sodass dieses Feld bei der Modellierung als Checkbox dargestellt wird (siehe Lst. 6).

Listing 6: Darstellung einer Variable als Checkbox

---

```

1  NOTEBOOK
2
3  #_____
4  CHAPTER "General"
5  ATTR "name"
6  ATTR "condition"
7  ATTR "default" ctrltype:check

```

---

Die letzte Besonderheit, die mit dieser Relation einhergeht, bezieht sich auf die Kardinalitäten der Ausgänge. Für die beiden Relationen, die von einer *Page* ausgehen, gibt es keine Einschränkungen, da ein valides Modell sowohl Seiten mit mehreren als auch mit keinen Ausgängen enthalten kann. Von einer *Condition* muss es allerdings immer mindestens zu einer Seite weitergehen, da die Objekte der Klasse *Condition* nicht das Ende eines Ablaufs sein dürfen. Außerdem benötigt eine *Condition* immer mindestens einen Eingang, da sie nicht den Start eines Ablaufs darstellen kann. Hierfür können in ADOxx für die *Condition* die Kardinalitäten der Klasse durch nachfolgende Definition eingeschränkt werden. Dabei wird festgelegt, dass jedes Objekt der Klasse *Condition* mindestens eine ausgehende Relation *Forwarding* sowie eine eingehende Relation *ConditionalTransition* besitzen muss (siehe Lst. 7).

Listing 7: Kardinalitäten der Klasse *Condition*

---

```

1  CARDINALITIES
2  RELATION "Forwarding"
3  min-outgoing:1
4  RELATION "ConditionalTransition"
5  min-incoming:1

```

---

In der Klassenhierarchie im Development-Toolkit ist zu erkennen, dass die definierte Modellierungssprache sich im Rahmen der magischen Zahl nach Miller bewegt und aus fünf Elementen besteht [Mil56]. Den zwei konkreten Klassen für die Navigationselemente sowie den drei Relationen (siehe Abb. 28).

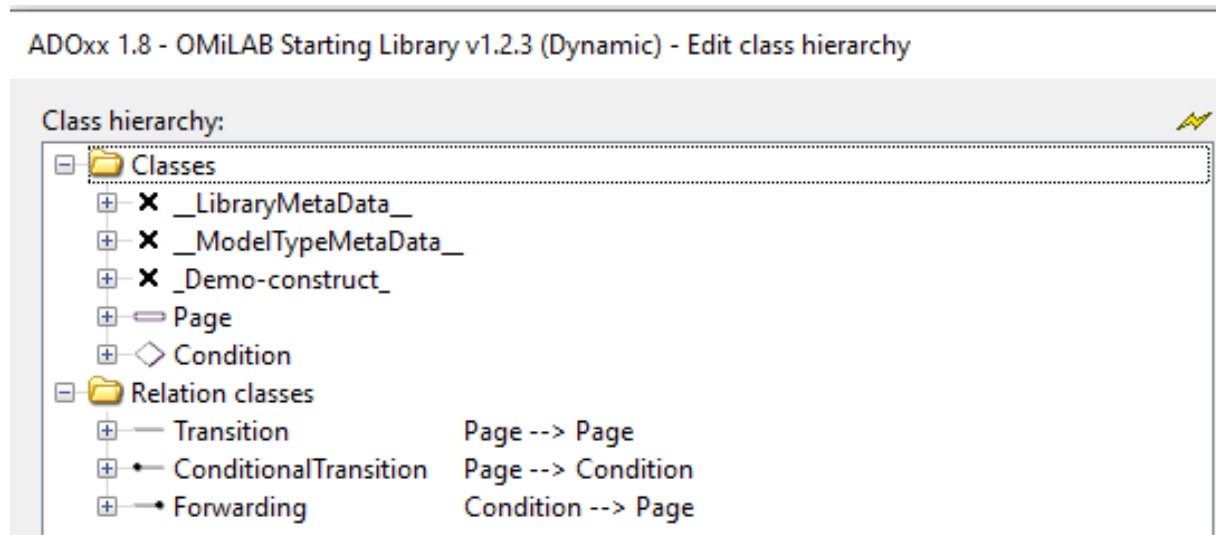


Abbildung 28: Komponenten der Modellierungssprache im Development-Toolkit

Nachdem die Implementierung der Modellierungssprache, bestehend aus der Notation, den einzelnen Symbolen, und ihrer Bedeutung, der Semantik, abgeschlossen ist, wird nun die beispielhafte Nutzung der entwickelten DSL demonstriert.

Der Einsatz der erstellten Modellierungssprache erfolgt im Modelling-Toolkit von ADOxx. Hierzu wird ein Modell vom Typ der neuen Sprache angelegt. Am Rand des Modellierungseitors stehen die fünf gerade erstellten Sprachelemente zur Verfügung (siehe rote Markierung in Abb. 29).

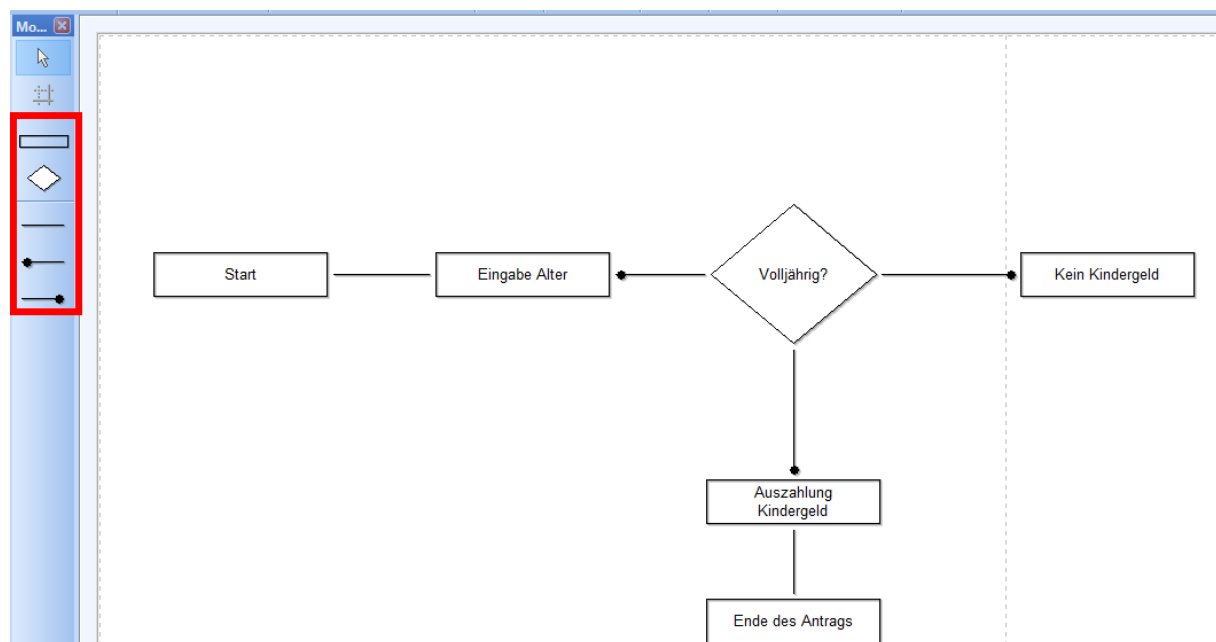


Abbildung 29: Beispielhaftes Modell im Modelling-Toolkit

Als Beispiel wurde hier ein Ablauf mit einer *Condition* als Verzweigung modelliert. Da den Abläufen durch die Vorgaben des Metamodells keine feste Domäne zugeordnet ist, wurde sich fachlich für die rudimentäre Darstellung eines Kindergeldantrags entschieden. Der weitere Verlauf ab der *Condition* hängt vom angegebenen Alter innerhalb des Antrags ab. Hierfür wird eine globale Variable, die im Laufe des Prozesses gesetzt werden muss, angelegt (siehe Abb. 30).

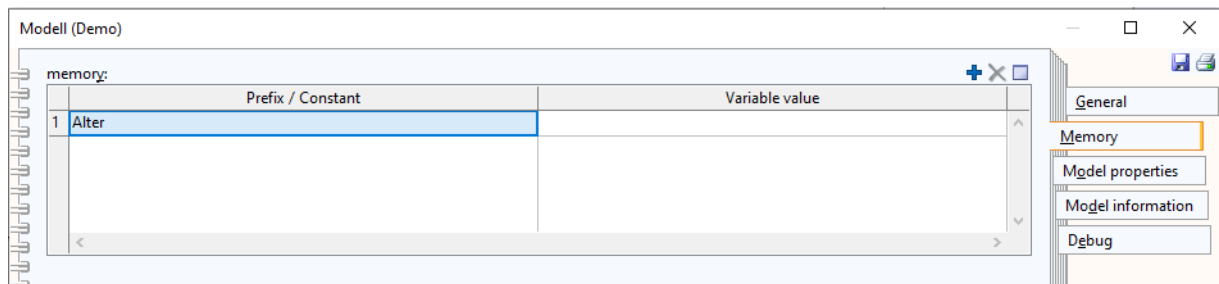


Abbildung 30: Globale Variable *Alter*

Sollte das angegebene Alter auf die Volljährig schließen lassen, also  $\geq 18$  sein, wird kein Kindergeld ausgezahlt. Die Attribute für dieses *Forwarding*, als einen der beiden Ausgänge der *Condition*, sind in Abbildung 31 zu sehen.

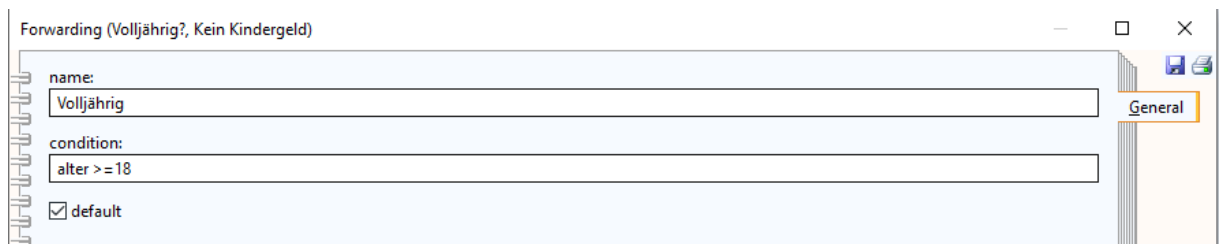


Abbildung 31: Attribute des *Default-Forwarding*

Ob das hier beschriebene Modell (siehe Abb. 29) nach den oben definierten Kardinalitäten valide ist, kann über das Menü mit dem Kardinalitätencheck überprüft werden. In diesem Fall würden dabei keine Fehler auftreten. Anders verhält es sich bei dem Minimalbeispiel aus Abbildung 32. Da hier die *Condition* lediglich einen Eingang, jedoch nicht mindestens einen Ausgang besitzt, schlägt der Kardinalitätencheck fehl. In der Fehlermeldung ist zu erkennen, dass eine ausgehende Relation vom Typ *Forwarding* vorausgesetzt wird.

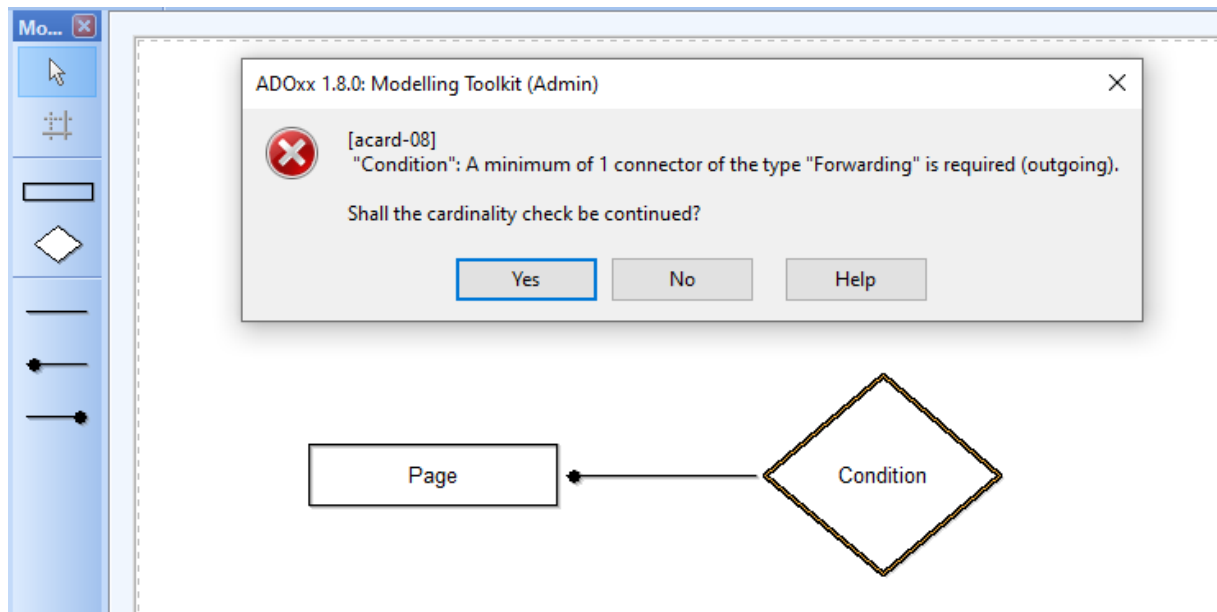


Abbildung 32: Fehlgeschlagener Kardinalitätencheck

Um im nächsten Schritt das Modell interpretieren zu können, ist ein Export aus dem Modelling-Toolkit notwendig. Hierzu wird der XML-Export von ADOxx genutzt. Die daraus entstehende XML-Datei enthält Tags für die Modellattribute sowie die modellierten Objekte und ihren Relationen.

Neben den allgemeinen Modellattributen, wie den Autor oder das Erstellungsdatum des Modells, ist hier auch der globale Speicher, als *Memory* benannt, zu finden. In Listing 8 ist zu erkennen, dass eine globale Variable namens *Alter* angelegt wurde. Ihr ist bisher kein Wert zugeordnet.

Listing 8: Auszug aus der XML-Repräsentation der Modellattribute

```

1  <MODELATTRIBUTES>
2    <ATTRIBUTE name="Author" type="STRING">Admin</ATTRIBUTE>
3    ...
4    <RECORD name="memory">
5      <ROW id="row.28217" number="1">
6        <ATTRIBUTE name="Prefix / Constant" type="STRING">Alter</
          ATTRIBUTE>
7        <ATTRIBUTE name="Variable value" type="STRING"></ATTRIBUTE>
8      </ROW>
9    </RECORD>
10 </MODELATTRIBUTES>

```

Die modellierten Objekte als Instanzen der definierten Klassen werden einzeln in der exportierten Datei aufgeführt. Die XML-Repräsentation einer *Page* aus dem modellierten

Ablauf aus Abbildung 29 ist in Listing 9 zu sehen. Unter den Attributen befindet sich beispielsweise die Positionierung in der grafischen Darstellung. Zusätzlich werden auch die spezifischen Attribute der Modellierungssprache, wie die Beschreibung (*description*) der Seite und ihre Variablen, abgebildet. Da für diese Seite keine lokalen Variablen definiert wurden, ist der *Record*, im Gegensatz zur obenstehenden Darstellung des globalen Speichers, leer.

Listing 9: XML-Repräsentation der Instanz einer *Page*


---

```

1  <INSTANCE id="obj.27827" class="Page" name="Auszahlung Kindergeld">
2    <ATTRIBUTE name="Position" type="STRING">NODE x:16cm y:10.75cm
      index:3</ATTRIBUTE>
3    <ATTRIBUTE name="External tool coupling" type="STRING"></ATTRIBUTE>
4    <ATTRIBUTE name="object-id" type="EXPRESSION">EXPR val:27827</
      ATTRIBUTE>
5    <ATTRIBUTE name="description" type="LONGSTRING">Es wird Kindergeld
      ausbezahlt.</ATTRIBUTE>
6    <RECORD name="variables"></RECORD>
7  </INSTANCE>

```

---

Zuletzt werden die Relationen dargelegt. Diese sind mit dem Tag **CONNECTOR** gekennzeichnet (siehe Lst. 10). Das Attribut *Name* gibt dabei die Art des Ausgangs an. Im hier vorliegenden Fall handelt es sich um ein *Forwarding*. Mit den Tags **FROM** und **TO** werden die Ausgangs- und Zielinstanz benannt. Des Weiteren ist angegeben, ob es sich um den Default-Ausgang handelt und welche Bedingung mit dieser Weiterleitung verknüpft ist. **&lt;** steht dabei für *less than* also kleiner als (<).

Listing 10: XML-Repräsentation einer Relation

---

```

1  <CONNECTOR id="con.27844" class="Forwarding">
2    <FROM instance="Volljaehrig?" class="Condition"></FROM>
3    <TO instance="Auszahlung Kindergeld" class="Page"></TO>
4    <ATTRIBUTE name="Positions" type="STRING">EDGE 0 index:7</ATTRIBUTE>
5    <ATTRIBUTE name="name" type="STRING">Minderjaehrig</ATTRIBUTE>
6    <ATTRIBUTE name="default" type="INTEGER">0</ATTRIBUTE>
7    <ATTRIBUTE name="condition" type="LONGSTRING">alter&lt;18</ATTRIBUTE
      >
8  </CONNECTOR>

```

---

Diese XML-Repräsentation dient in Kapitel 7 als Eingabe für den Interpreter.

## 7 Interpretation des modellierten Ablaufs

Abschließend ist es nun das Ziel den modellierten Ablauf zu interpretieren. Bei einem Interpreter handelt es sich um ein Metaprogramm, das ein gegebenes maschinenlesbares Programm ausführt und evaluiert. Er kann Argumente entgegen nehmen und Zugriff auf die Systemumgebung, wie zum Beispiel das Dateisystem, haben. Das Ergebnis kann von einem einfachen Wert bis hinzu einem wiederum vereinfachten maschinenlesbaren Programm reichen. In dem Anwendungsfall dieser Arbeit führt der Interpreter Elemente aus und erzeugt somit das Ein- und Ausgabeverhalten des modellierten Ablaufs. [Lä18]

Statt der Interpretation einer DSL gibt es noch andere Möglichkeiten zur weiteren Verarbeitung. Dies ist unter anderem die Kompilierung. Dabei wird aus der DSL eine Anwendung generiert. Eine andere Alternative ist die Nutzung eines Präprozessors. Dieser übersetzt Teile der DSL in Konstrukte einer Basissprache, in die diese Übersetzungen dann integriert werden. Außerdem kann eine DSL durch die Definition neuer Datentypen und Operatoren in eine GPL eingebettet werden. Dies kann beispielsweise bei der Entwicklung von Anwendungsbibliotheken zutreffen. [MHS05]

Allgemein bieten die Interpretation und Kompilierung jedoch den Vorteil, dass die Syntax der DSL nahe an den bekannten Notationen der Domänenexperten gehalten werden kann, da eine weitere Implementierung durch die Entwickler notwendig ist. Hier eröffnet sich ebenso die Möglichkeit eine angemessene und aussagekräftige Fehlerbehandlung zu implementieren. Allerdings geht mit diesen Vorteilen auch der größte Nachteil einher. Zusätzlich zur Definition und Erstellung der Domänensprache gibt es weiteren Entwicklungsaufwand. [MHS05]

Der Interpreter ist also ein Programm, welches auf der Zielplattform läuft, das DSL-Programm lädt und nach ihm handelt. Im Gegensatz dazu transformiert der Compiler die DSL in ein Artefakt, wobei oft Code einer GPL generiert wird. Dieses Artefakt wird dann auf der Zielplattform ausgeführt. [Voe13]

Dies führt dazu, dass bei der Nutzung eines Interpreters eine höhere Kontrolle über die Ausführungsumgebung besteht. Außerdem gestaltet sich die Erweiterung eines Interpreters im Vergleich zu einem Compiler/Codegenerator leichter, da dies während der Laufzeit möglich ist. [MHS05]

Da der Interpreter das DSL-Programm schrittweise durchgeht und die semantischen Aktionen ausführt, die mit den Elementen verknüpft sind, gilt es bei der Implementierung des Interpreters drei Punkte zu beachten. Zu Beginn wird das Modell eingelesen. Im Anschluss werden die erkannten Objekte semantisch eingeordnet, sodass im letzten Schritt die



Aktionen durchgeführt werden, die zur Ausführungssemantik der Sprache passen. [Voe13]

Die Entwicklung des Interpreters erfolgt mit der Programmiersprache Java. Der Prototyp wird als Konsolenanwendung zur Verfügung gestellt, sodass über eine Nutzereingabe durch die einzelne Seiten navigiert werden kann.

Für das Einlesen, dem sogenannten Parsen, der XML-Datei wird der von Java standardmäßig zur Verfügung gestellte `DocumentBuilder` eingesetzt. Dieser erstellt eine Baumstruktur, woraufhin auf die einzelnen XML-Tags zugegriffen werden kann. Das exportierte Modell muss dabei im Verzeichnis des Interpreters liegen. Der Dateiname wird als Parameter bei der Ausführung an das Programm übergeben. Als zweiter Schritt werden aus dem geparsen XML-Dokument die verschiedenen Elemente der Modellierungssprache extrahiert und als Java-Objekte angelegt.

Begonnen wird mit der Erstellung des globalen Speichers. Dessen Variablen sind, wie oben beschrieben, unter den Modellattributen zu finden. Das Dokument wird somit nach dem Tag `MODELATTRIBUTES` durchsucht. Bei der Iteration über die erhaltenen Attribute werden dann aus den Zeilen des *Records* die Schlüssel-Werte-Paare zu einer `Map` hinzugefügt. Diese stellt den *Memory* dar.

Analog zu diesem Vorgehen werden die Objekte für die Navigationselemente und Relationen angelegt. Die Relationen werden dabei direkt mit ihrem Ziel verknüpft. Für den Abschluss der Erstellung für die Verbindungen zwischen den Instanzen der Navigationselemente, werden den *Pages* und *Conditions* ihre Ausgänge zugeordnet. Hierfür wird ein weiteres mal über das XML-Dokument iteriert. Dabei wird für jeden `CONNECTOR` Tag das Navigationselement gesucht, auf dessen Instanz im Tag `FROM` verwiesen wird. Dieser Seite oder Bedingung wird anschließend die Relation zugewiesen, die auf den nächsten Schritt zeigt. Dieses Vorgehen des Interpreters ist auszugsweise in Listing 11 zu erkennen.

Listing 11: Zurodnung eines Übergangs (*Transition* zu einem Navigationselement)

---

```

1  public static List<Navigationelement>
    linkTransitionToNavigationelement(
2      Document document, List<Navigationelement> navigationElements,
3      List<Transition> transitions) {
4      for (Node node : iterable(document.getElementsByTagName(
        NodeConstants.CONNECTOR))) {
5          if (node.getNodeType() == Node.ELEMENT_NODE) {
6              Element element = (Element) node;
7              for (Navigationelement navigationelement : navigationElements) {
8                  if (navigationelement.getName().equals(((Element) element
9                      .getElementsByTagName(NodeConstants.FROM).item(0))
10                     .getAttribute(AttributeConstants.INSTANCE))) {
11                      for (Transition transition : transitions) {
12                          if (transition.getName().equals(getAttribute(element,
13                              AttributeConstants.NAME))) {
14                              navigationelement.getTransition().add(transition);
15                          }
16                      }
17                  }
18              }
19          }
20      return navigationElements;
21  }

```

---

Nachdem alle relevanten Elemente für die Abbildung und Durchführung des Ablaufs erstellt wurden, wird im letzten Schritt die Ausführungslogik implementiert. Zum Start muss die erste Seite identifiziert werden. Da die Ablauffolge durch Ausgänge definiert ist, die Startseite sich jedoch durch den fehlenden Eingang auszeichnet, wird das Programm hierfür vom Ende aus durchlaufen. Ein Ende ist eine Seite, die keinen weiteren Übergang hat. Von dieser Endseite wird die Vorgängerseite gesucht. Dies wird für die gefundene Seite solange wiederholt bis kein Vorgänger mehr gefunden wird. Sollten bei diesem Prozedere mehrere mögliche Startseiten erkannt werden, wird eine **Exception** geworfen, da es einen eindeutigen Start für den Ablauf geben muss (sieht Lst. 12).

Listing 12: Algorithmus zur Suche der validen Startseite

---

```

1  public static Page findStart(List<Navigationelement>
    navigationelements) throws Exception {
2      Set<Page> startPages = new HashSet<>();
3      Set<Page> endPages = new HashSet<>();
4      for (Navigationelement navigationelement : navigationelements) {
5          if (navigationelement.getTransition().isEmpty()) {
6              endPages.add((Page) navigationelement);
7          }
8      }
9      for (Page page : endPages) {
10         Navigationelement start = page;
11         while (findPredecessor(navigationelements, start) != null) {
12             start = findPredecessor(navigationelements, start);
13         }
14         startPages.add((Page) start);
15     }
16     if (startPages.size() != 1) {
17         throw new Exception("Keine gueltige Anzahl an Startseite. Es muss
            eine spezifische Startseite existieren.");
18     }
19     return (Page) startPages.toArray()[0];
20 }

```

---

Der Name und die Beschreibung zur gefundenen Startseite werden nun auf der Konsole ausgegeben. Des Weiteren stehen dem Nutzer Auswahloptionen zur Verfügung, um mögliche globale oder lokale Variablen zu bearbeiten. Zusätzlich kann der Nutzer den Ablauf fortsetzen beziehungsweise abschließen, falls es sich bereits um eine Endseite handelt. Sobald der Ablauf über die Startseite hinaus fortgeschritten ist, gibt es immer die Option auf die vorherige Seite zurück zu navigieren.

Bei der Fortsetzung des Ablaufs wird stets nach der nächsten Seite gesucht. Falls der aktuellen Seite nur ein Übergang zu einer weiteren Seite zugeordnet ist, kann diese direkt angesteuert werden. Sollte der weitere Verlauf jedoch an einer *Condition* hängen, muss die folgende Abzweigung validiert werden, um dem richtigen Pfad zu folgen. Hierfür werden die Bedingungen aus allen Ausgängen der *Condition* ausgewertet. Die möglichen Expressions wurden dabei auf die booleschen Operatoren ( $=$ ,  $\neq$ ,  $<$ ,  $\leq$ ,  $>$ ,  $\geq$ ) beschränkt. Für die Evaluierung der linken und rechten Seite wird die Java-Bibliothek *Exp4j* eingesetzt, um die globalen Variablen, von denen eine *Condition* abhängen kann, durch ihre Werte zu ersetzen und möglicherweise weitere mathematische Operationen abzubilden. So ist beispielsweise  $\text{Alter} + 1 < 18$ , soweit der globalen Variable *Alter* ein Wert

zugeordnet ist, eine valide Bedingung einer Relation *Forwarding*.

Wird der Interpreter nun mit dem Modell aus der Abbildung 14 ausgeführt, erhält der Nutzer zu Beginn folgende Ansicht (siehe Abb. 33).

```
START
Bitte Name und Vorname angeben
-----
Optionen:
1: globale Variable Alter ändern. Aktueller Wert:
2: lokale Variable Vorname ändern. Aktueller Wert:
3: lokale Variable Name ändern. Aktueller Wert:
4: Weiter
```

Abbildung 33: Konsolenausgabe der ersten Seite

Über die Eingabe der Zahlen eins bis vier kann der Nutzer die beschriebenen Aktionen ansteuern. Es ist zu erkennen, dass es für das Modell eine globale Variable namens *Alter* gibt. Diese Seite enthält zusätzlich zwei lokale Variablen *Vorname* und *Name*. Nachdem für diese drei Variablen optional Werte angegeben wurden, kann über die Eingabe der Zahl 4 zur nächsten Seite navigiert werden. In Abbildung 34 ist zu sehen, dass der Wert der globalen Variable übernommen wurde. Da es sich nicht mehr um die Startseite handelt, gibt es jetzt zusätzlich zu *Weiter* auch die Option *Zurück*. Diese Seite besitzt keine weiteren lokalen Variablen.

```

START
Bitte Name und Vorname angeben
-----
Optionen:
1: globale Variable Alter ändern. Aktueller Wert: 13
2: lokale Variable Vorname ändern. Aktueller Wert: Thomas
3: lokale Variable Name ändern. Aktueller Wert: Großbeck
4: Weiter
4
EINGABE ALTER
Bitte geben Sie Ihr Alter an.
-----
Optionen:
1: globale Variable Alter ändern. Aktueller Wert: 13
2: Zurück
3: Weiter

```

Abbildung 34: Konsolenausgabe der zweiten Seite

Im zugehörigen Modell befindet sich der Ablauf im Moment auf der in Abbildung 35 markierten Seite.

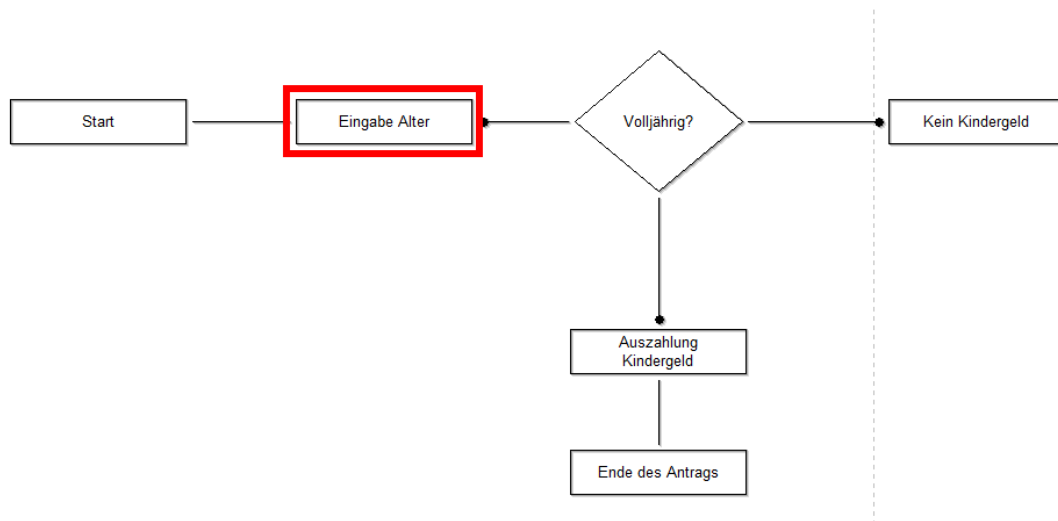


Abbildung 35: Aktueller Standpunkt im modellierten Ablauf

Trifft der Ablauf auf eine *Condition*, so ist dies nicht für den Nutzer zu sehen. Im Hintergrund werden die Bedingungen evaluiert, sodass auf die richtige Seite weitergeleitet werden kann. Bei der Fortsetzung der Navigation landet der Ablauf im Zweig der Kindergeldauszahlung, da das angegeben Alter mit 13 unter 18 liegt (siehe Abb. 36 und Abb.

37).

```
AUSZAHLUNG KINDERGELD

-----
Optionen:
1: globale Variable Alter ändern. Aktueller Wert: 13
2: Zurück
3: Weiter
```

Abbildung 36: Konsolenausgabe des nächsten Schritts

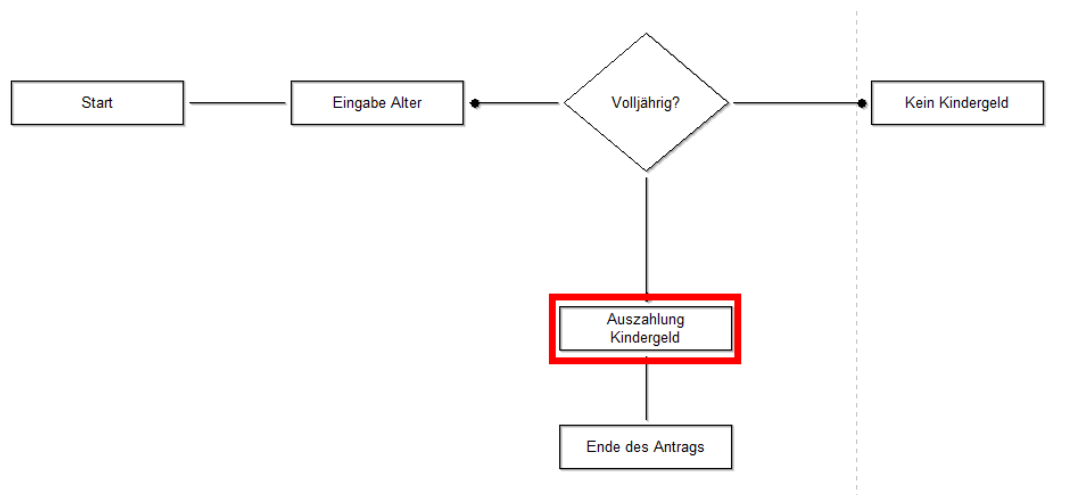


Abbildung 37: Standpunkt nach der Abzweigung im modellierten Ablauf

Wird nun die Option *Zurück* ausgewählt, so wird der vergangene Pfad über die *Condition* hinweg wieder zurückgegangen. Der Ablauf steht damit wieder am zweiten Schritt und es ist möglich ein neues Alter einzugeben (siehe Abb. 38).

```
AUSZAHLUNG KINDERGELD

-----
Optionen:
1: globale Variable Alter ändern. Aktueller Wert: 13
2: Zurück
3: Weiter
2
EINGABE ALTER
Bitte geben Sie Ihr Alter an.
-----
Optionen:
1: globale Variable Alter ändern. Aktueller Wert: 13
2: Zurück
3: Weiter
1
Geben Sie den neuen Wert für die globale Variable alter ein:
27
EINGABE ALTER
Bitte geben Sie Ihr Alter an.
-----
Optionen:
1: globale Variable Alter ändern. Aktueller Wert: 27
2: Zurück
3: Weiter
```

Abbildung 38: Auswahl der Option *Zurück* und Eingabe eines neuen Alters

Der neue Wert für die globale Variable *Alter* sorgt bei der nächsten Validierung der *Condition* dafür, dass der andere Ausgang angesteuert wird. Der Interpreter gelangt auf eine Endseite und statt der Option *Weiter* wird die Beendigung des Programms für den Nutzer zur Auswahl dargestellt (siehe Abb. 39).

```
KEIN KINDERGELD
Die Person ist bereits volljährig. Daher wird kein Kindergeld ausgezahlt.
-----
Optionen:
1: globale Variable Alter ändern. Aktueller Wert: 27
2: Zurück
3: Programm beenden
```

Abbildung 39: Konsolenausgabe einer Endseite

Da in Abbildung 39 zu erkennen ist, dass es zu keiner Auszahlung des Kindergelds kam, hat der Ablauf folgende Endseite im Modell erreicht (siehe Abb. 40).

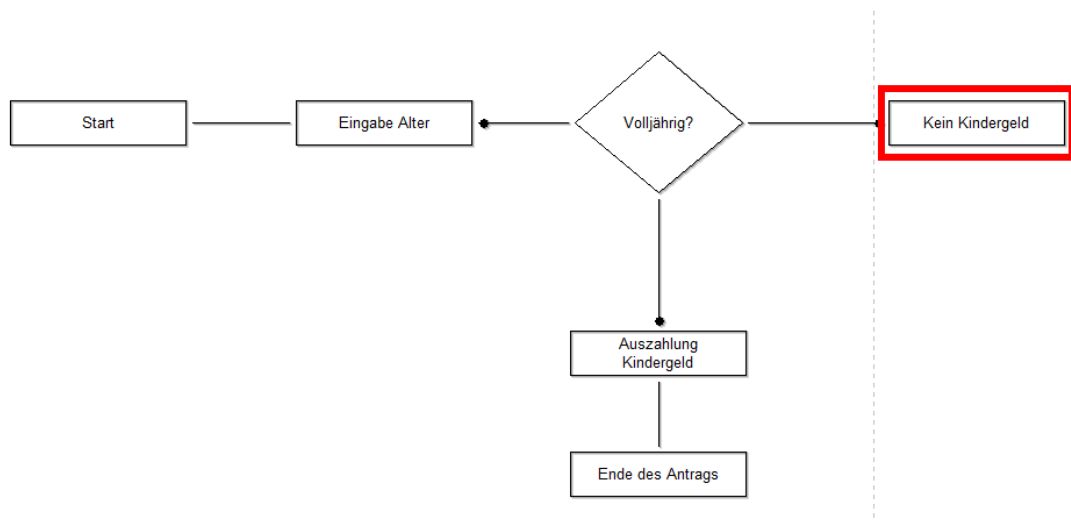


Abbildung 40: Erreichtes Ende im modellierten Ablauf

Mit Hilfe des entwickelten Interpreters ist es also möglich einen modellierten Ablauf zu simulieren. Hierzu nimmt er als Eingabe die XML-Repräsentation eines Modells, das auf Basis der eigenen Modellierungssprache erstellt wurde, und leitet durch die einzelnen Schritte. Dabei wird der aktuellen Zustand auf der Konsole ausgegeben. Über diese hat der Nutzer auch die Möglichkeit durch den Ablauf zu navigieren und Variablen zu setzen.



## 8 Fazit und Ausblick

Durch die Ausarbeitungen in dieser Arbeit konnte gezeigt werden, dass die Erfassung von Abläufen mit Hilfe einer domänenspezifischen Sprache, die eine grafische Repräsentation besitzt, möglich ist. Weitergehend konnte gezeigt werden, dass durch dieses Vorgehen einige Vorteile der Metamodellierung und Interpretation von Modellen genutzt werden können.

So wurden bei der Implementierung einige vordefinierte Ablauflogiken in die Definition der neuen Modellierungssprache aufgenommen, sodass für die Abläufe verschiedener Anwendungen nicht wiederkehrend neue Programme entwickelt werden müssen. Dies stellt einen großen Pluspunkt bei der Handhabung von immer wieder wechselnden Anforderungen dar.

Ebenso trägt die ausgearbeitete DSL mit dem zugrundeliegenden Metamodell zu einer sauberen Abstraktion und Modellierung der verschiedensten Prozesse und Abläufe bei. Dies erleichtert sowohl die Arbeit eines einzelnen Nutzers als auch die Zusammenarbeit innerhalb der Nutzergruppen. Des Weiteren wird durch die Interpretation der Modelle die automatische Ausführung der modellierten Abläufe unterstützt und somit der Automatisierungsgrad gesteigert.

Da das erstellte Metamodell bisher lediglich prototypische Funktionalitäten enthält, gibt es noch viele Möglichkeiten für Erweiterungen und Verbesserungen. Durch seine Allgemeingültigkeit für die Modellierung vieler Abläufe besteht keine große Hürde zukünftig darauf aufzubauen. So kann das Metamodell problemlos um weitere Elemente erweitert und spezialisiert werden. Hierbei gilt jedoch zu beachten das im Moment schlanke Modell nicht zu überfrachten oder gar ganze Teile bekannter Modellierungssprachen zu übernehmen oder nach zu bauen. In diesen Fällen ist von der Nutzung einer eigenen DSL abzuraten.

Eine weitere Alternative für die Erweiterung bietet sich über die bereits mehrmals erwähnte Einführung von Elementen für eine grafische Nutzerschnittstelle. Dieser Schritt würde einen enormen Mehrwert für einen Großteil der Nutzergruppe bieten. Die prototypische Ausführung des bisherigen Interpreters als Konsolenprogramm lässt bei der Bedienungs-freundlichkeit noch Spielraum nach oben. So werden die Nutzereingaben bisher auch nicht gespeichert, sondern gehen mit der Beendigung des Programms wieder verloren.

Ein letzter Punkt der den Ausbau der DSL betrifft, ist das Thema rund um das Testen einer domänenspezifischen Sprache. Dies wurde angesichts des Schwerpunkts auf den Modellierungsprozess nicht weiter behandelt. Die Ansätze des *Model Checking* und der

*Abstract Execution* aus Kapitel 2.3 bieten hier jedoch spannende Einstiegspunkte.

Für die Unterstützung der Modellierung durch ein Werkzeug konnte mit ADOxx ein Tool gefunden werden, dass fast alle Anforderungen zufriedenstellend erfüllt. Obwohl die meisten der untersuchten Werkzeuge bereits lange auf dem Markt sind, erfüllen sie immer noch einen Großteil der Anforderungen an aktuelle Modellierungssoftware und sind weiterhin zeitgemäß.

Durch das dargelegte Vorgehen mit der initialen Sammlung von Anforderungen an die Abbildung von Abläufen, gefolgt von der Implementierung des Metamodells, bis hin zur Entwicklung der grafischen DSL, konnte eine Modellierungssprache erarbeitet werden, die mittels zugehörigen Interpreter ausgeführt werden kann. Aus diesem Grund ist es abschließend möglich eigene Prozesse grafisch zu modellieren und ihren Ablauf bedingungsgesteuert zu simulieren.

Als Anwendungsfall für die Zukunft könnte zum Beispiel die Patientenaufnahme im Krankenhaus oder beim Arzt in Frage kommen. Hier geht es um einen sich wiederholenden Ablauf, der sich auf Grundlage der Eigenschaften und des Krankheitsbilds des Patienten unterschiedlich abzweigen kann. Nach der Modellierung dieses Prozess könnte der Patient mittels der Interpretation des Modells durch den Ablauf geleitet werden. Somit kann die Patientenaufnahme beschleunigt werden.

## 9 Quellenverzeichnis

- [ado] *ADOxx Documentation.* <https://www.adoxx.org/live/introduction-to-adoxx>. – Zuletzt abgerufen am 15.02.2024
- [Bab24] BABB, Benjamin: Workflow vs. Process: What Are the Key Differences? (2024). <https://www.pipefy.com/blog/workflow-vs-process/>. – Zuletzt abgerufen am 14.02.2024
- [BKK18] BUCHMANN, R. ; KARAGIANNIS, D. ; KIRIKOVA, M.: *The Practice of Enterprise Modeling*. Springer, 2018 <https://doi.org/10.1007/978-3-030-02302-7>. – ISBN 978-3-030-02301-0
- [BP07] BRUMAR, Bogdan ; POPA, Emil M.: Advanced techniques for metamodeling. (2007), 01
- [CT91] COHEN, Robert F. ; TAMASSIA, Roberto: Dynamic expression trees and their applications. In: *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, Society for Industrial and Applied Mathematics, 1991. – ISBN 0897913760
- [Fow05] FOWLER, Martin: Language Workbenches: The Killer-App for Domain Specific Languages? (2005). <https://martinfowler.com/articles/languageWorkbench.html>. – Zuletzt abgerufen am 13.07.2023
- [gar] *Citizen Developer.* <https://www.gartner.com/en/information-technology/glossary/citizen-developer>. – Zuletzt abgerufen am 04.04.2024
- [Gra16] GRANADA, David: Comparing tools to build graphical modeling editors. (2016). <https://modeling-languages.com/comparing-tools-build-graphical-modeling-editors/>. – Zuletzt abgerufen am 08.02.2024
- [HS12] HENDERSON-SELLERS, Brian: *On the Mathematics of Modelling, Meta-modelling, Ontologies and Modelling Languages*. Springer, 2012 <https://doi.org/10.1007/978-3-642-29825-7>. – ISBN 978-3-642-29824-0
- [HT20] HECKEL, Reiko ; TAENTZER, Gabriele: *Graph Transformation for Software Engineers*. Springer, 2020 <https://doi.org/10.1007/978-3-030-43916-3>. – ISBN 978-3-030-43915-6
- [KK02] KARAGIANNIS, Dimitris ; KÜHN, Harald: Metamodeling Platforms, 2002. – ISBN 978-3-540-44137-3

- [KKP<sup>+</sup>14] KARSAI, Gabor ; KRAHN, Holger ; PINKERNELL, Claas ; RUMPE, Bernhard ; SCHINDLER, Martin ; VÖLKEL, Steven: *Design Guidelines for Domain Specific Languages*. 2014
- [KTK09] KÄRNÄ, Juha ; TOLVANEN, Juha-Pekka ; KELLY, Steven: Evaluating the use of domain-specific modeling in practice. In: *The 9th OOPSLA Workshop on Domain-Specific Modeling* (2009)
- [LFSS21] LEBENS, M. ; FINNEGAN, R. ; SORSEN, S. ; SHAH, J.: Rise of the citizen developer / Muma Business Review. Version:2021. <https://doi.org/10.28945/4885>. 2021. – Forschungsbericht
- [Lä18] LÄMMEL, Ralf: *Software Languages*. Springer, 2018 <https://doi.org/10.1007/978-3-319-90800-7>. – ISBN 978-3-319-90798-7
- [MAEFH18] MEZHUYEV, Vitaliy ; AL-EMRAN, Mostafa ; FATEHAH, Murni ; HONG, Ng C.: Factors Affecting the Metamodelling Acceptance: A Case Study From Software Development Companies in Malaysia. In: *IEEE Access* (2018). <http://dx.doi.org/10.1109/ACCESS.2018.2867559>. – DOI 10.1109/ACCESS.2018.2867559
- [MBL<sup>+</sup>11] MAGYARI, Endre ; BAKAY, Arpad ; LANG, Andras ; PAKA, Tamas ; VIZHANYO, Attila ; AGARWAL, Aditya ; KARSAI, Gabor: UDM: An Infrastructure for Implementing Domain-Specific Modeling. (2011)
- [meta] *MetaCase*. <https://www.metacase.com/>. – Zuletzt abgerufen am 04.03.2024
- [metb] *MetaEdit+ Evaluation Tutorial*. <https://www.metacase.com/support/55/manuals/evaltut/et.html>. – Zuletzt abgerufen am 08.02.2024
- [MHS05] MERNIK, Marjan ; HEERING, Jan ; SLOANE, Anthony M.: When and How to Develop Domain-Specific Languages. (2005). <http://dx.doi.org/10.1145/1118890.1118892>. – DOI 10.1145/1118890.1118892
- [Mil56] MILLER, G.A.: The magical number seven, plus or minus two: Some limits on our capacity for processing information. (1956). <https://doi.org/10.1037/h0043158>
- [mps] *MPS Tutorial - Graphical shape*. <https://www.jetbrains.com/help/mps/shapes-an-introductory-mps-tutorial.html#graphicalshape>. – Zuletzt abgerufen am 21.02.2024
- [omi] *OMILAB*. <https://www.omilab.org/>. – Zuletzt abgerufen am 04.03.2024

- [PK02] POHJONEN, Risto ; KELLY, Steven: Domain-specific modeling. In: *Dr. Dobb's Journal* (2002)
- [RBGN<sup>+</sup>17] REINHARTZ-BERGER, Iris ; GULDEN, Jens ; NURCAN, Selmin ; GUEDRIA, Wided ; BERA, Palash: *Enterprise, Business-Process and Information Systems Modeling*. Springer, 2017 <https://doi.org/10.1007/978-3-319-59466-8>. – ISBN 978-3-319-59465-1
- [Sta20] STARKE, Gernot: *Effektive Softwarearchitekturen*. 9. Auflage. Carl Hanser Verlag, 2020. – ISBN 978-3-446-46376-9
- [Tom17] TOMASSETTI, Federico: *The complete guide to (external) Domain Specific Languages*. <https://tomassetti.me/domain-specific-languages/>, 2017. – zuletzt aufgerufen am 17.08.2023
- [Voe13] VOELTER, Markus: *DSL Engineering*. CreateSpace Independent Publishing Platform, 2013 <https://doi.org/10.1007/978-3-319-59466-8>. – ISBN 978-1-481-21858-0
- [Wag23] WAGNER, Gerd: DPMN: A Discrete Process Modeling Language. (2023). <https://modeling-languages.com/dpmn-a-discrete-process-modeling-language/>. – Zuletzt abgerufen am 14.02.2024
- [WWJM19] In: WANG, Hongwei ; WANG, Guoxin ; JINZHI, Lu ; MA, Changfeng: *Ontology Supporting Model-Based Systems Engineering Based on a GOPRRR Approach*. 2019. – ISBN 978-3-030-16180-4
- [YZQ20] YONGLIN, Lei ; ZHI, Zhu ; QUN, Li: An ontological metamodeling framework for semantic simulation model engineering. In: *Journal of Systems Engineering and Electronics* (2020). <http://dx.doi.org/10.23919/JSEE.2020.000032>. – DOI 10.23919/JSEE.2020.000032