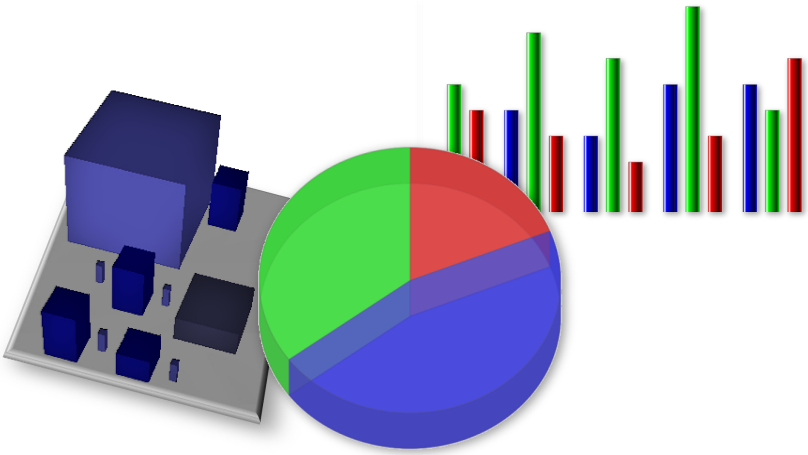
CARL VON OSSIETZKY universität OLDENBURG

Fakultät II - Informatik, Wirtschafts- und Rechtswissenschaften
Department Informatik

Bachelorstudiengang Informatik
Bachelorarbeit

Model Driven Software Visualization Services

Marie-Christin Ostendorp



Gutachter:
Prof. Dr. Andreas Winter
Jan Jelschen, M. Sc.

16. Oktober 2012

Abstract

Deutsche Version

Softwarevisualisierung verfolgt unterschiedliche Ziele: Sie kann beispielsweise zum Verstehen bestehender Softwaresysteme in einem Reengineering-Prozess entscheidend beitragen und damit die Produktivität erhöhen. Darüber hinaus kann sie Schwachstellen leichter identifizierbar machen oder als Kommunikationsgrundlage dienen.
Aus diesen unterschiedlichen Zielen folgt, dass es sehr unterschiedliche Anforderungen an die Art der Visualisierung und den dargestellten Inhalt gibt. Aktuell gibt es allerdings fast keine Personalisierungsmöglichkeiten bestehender Visualisierungsarten.
Aus diesem Grund wurde in der Bachelorarbeit der ELVIZ-Ansatz entwickelt. Bei diesem können Visualisierungsart und -inhalt unabhängig voneinander spezifiziert werden. In einer modellgetriebenen Arbeitsweise wird dann automatisch eine Transformation generiert, die die Ausgangsdaten - welche Fakten über ein zu visualisierendes System beinhalten - in Daten überführt, die zur gewünschten Visualisierungsart konform sind. Diese Daten können abschließend in einem Renderer genutzt werden, um die reale Grafik zu erzeugen. Insgesamt können so beliebige Visualisierungen erstellt werden, die exakt passend zum Ziel der Visualisierung ausgewählt oder neu definiert werden können.

English Version

Software Visualization has different aims: It can raise the efficiency in a reengineering process by easing the understanding of an existing system. Furthermore software visualization can help to identify weak points in a certain software system or it can be used as a basis to ease communication in a team.
These different aims lead to different requirements in regard to the kind of the graphical representation and in regard to the content of this representation.
Currently there is almost no possibilty to personalize the graphical representation and the desired content.
That was the reason why the ELVIZ-approach has been development in this Bachelor thesis: This approach offers the possibility to specify the precise kind and content of the graphical representation. The ELVIZ-approach uses Model Driven Engineering (MDE) to automatically generate a transformation to transform data of the given system into data that conforms to the desired kind of graphical representation. This data can be used in a renderer to produce the real diagram.
So all in all it is possible to create a totally user-defined graphic representation, which is exactly suitable to the initial aim of the visualization.

Inhaltsverzeichnis

1 | Einleitung

„Software is invisible, disappearing into files on disks." [1]

Diese Eigenschaft von Software beschreibt Thomas Ball in seiner Veröffentlichung „Software Visualization in the Large" bereits im Jahr 1996. Als Konsequenz dieser Eigenschaft sieht er eine niedrigere Produktivität bei den Programmierern: Diese sind in ihrer Arbeit mit komplexen Softwaresystemen konfrontiert, die visuell nicht mehr für sie greifbar sind. Der einzige Anhaltspunkt ist der Quellcode. Da dieser mitunter allerdings hunderttausende Zeilen an Code umfasst, ist die Verstehenskomplexität eines vorliegenden Softwaresystems sehr hoch. Dieses Verstehen ist jedoch notwendig – sei es um ein bestehendes Softwaresystem weiterzuentwickeln, auf Fehler zu überprüfen oder in der heutigen Zeit – in der permanent neue Technologien entwickelt werden – an die neuesten Innovationen anzupassen. Doch nicht nur um bestehende Softwaresysteme zu erhalten ist das Verstehen oder die Analyse eines vorliegenden Quellcodes notwendig: Es wird beispielsweise auch benötigt um ein in Teamarbeit entstehendes System voranzubringen oder um Systeme miteinander vergleichen zu können.
Ein wichtiges Konzept, das den Verstehensprozess von Software vereinfachen kann und Software wieder „sichtbar" macht, ist dabei die Softwarevisualisierung: In einer Studie von Rainer Koschke aus dem Jahr 2003 wurde die Relevanz der Softwarevisualisierung für Softwarewartung, Reverse Engineering sowie Reengineering untersucht. Dabei hielten 42% der Befragten die Sofwarevisualisierung zum Verstehen von Softwaresystemen für wichtig und sogar 40% für absolut notwendig [21].

Darauf aufbauend stellt sich jedoch eine weitere Frage: Welche Visualisierung ist sinnvoll um Software eine Form zu geben und endlich wieder „sichtbar" zu machen? In den vergangenen Jahren wurden in der Literatur dafür einige Ansätze präsentiert: Angefangen mit einfachen, farbigen Linienrepräsentationen von Quellcodezeilen [1] bis hin zu Repräsentationen von Software als dreidimensionale Städte – sogenannte CodeCities [28].
Die vorhandenen Visualisierungstools lassen jedoch kaum Raum für eine individuelle Gestaltung einer eigenen Repräsentation. Dabei ist es für unterschiedliche Problemstellungen unabdingbar, unterschiedliche Visualisierungen erstellen zu können: Eine Visualisierung, die für eine Präsentation eines neuen Systems verwendet wird, benötigt natürlich eine andere Darstellungsart als eine Visualisierung, die darauf abzielt Code Smells [13] - wie beispielsweise Gottklassen - zu identifizieren.

Aus diesem Grund wird in der Bachelorarbeit ein Softwarevisualisierungsansatz entwickelt, der in der Lage ist, einen vorliegenden Quellcode in jeder nur denkbaren Weise zu visualisieren. Die genaue Art der Visualisierung soll dabei frei vom Anwender wählbar sein. Dadurch wird es möglich Software exakt passend zur vorliegenden Problemstellung in jeder nur vorstellbaren Weise „sichtbar" zu machen.

1.1 Problemstellung und Ziel

Wie in der Einleitung beschrieben, ist es das Ziel der Bachelorarbeit Software wieder „sichtbar" zu machen. Die Intention, die mit einer „Sichtbarmachung" von Software verfolgt wird, kann dabei allerdings ganz unterschiedlich sein: So kann eine Visualisierung beispielsweise die Verstehenskomplexität eines vorliegenden Systems in einem Softwarewartungsprozess reduzieren und dadurch eine erhöhte Produktivität sowie eine Reduktion von Zeitaufwand und Kosten im Wartungsprozess erzielen. Darüber hinaus kann eine Visualisierung auch dazu verwendet werden, um eine gemeinsame Kommunikationsbasis in einer Teamarbeit zu schaffen, was wiederum eine Erhöhung der Produktivität zur Folge hat. Zudem können Fakten über ein Softwaresystem anschaulich präsentiert werden. So ist es in einem Treffen mit dem Auftraggeber der Software möglich Fortschritte und teilweise Qualität der Software visuell nachzuweisen und zu kommunizieren. Auch zur Fehlersuche und bei der Suche von Code Smells kann die Softwarevisualisierung einen entscheidenden Beitrag leisten.

An diesem breiten Einsatzgebiet von Softwarevisualisierung ist bereits zu sehen, dass es viele *unterschiedliche* Anforderungen an die Visualisierungsart und den Visualisierungsinhalt geben kann: Die *Visualisierungsart* beschreibt dabei *wie* die vorhandenen Fakten dargestellt werden, während der *Visualisierungsinhalt* definiert *was* dargestellt wird.
In der Literatur gibt es einige Möglichkeiten Software visuell darzustellen [27] [28] [1] - Visualisierungsart und Visualisierungsinhalt sind dabei jedoch fest vorgegeben und es gibt kaum Möglichkeiten diese für die eigene Problemstellung zu personalisieren.
Angenommen ein Auftragnehmer will in einem Teamprojekt die prozentuale Verteilung der Methoden auf die einzelnen Klassen visualisieren, um Gottklassen auszuschließen und eine Grundlage zur Kommunikation und zum Vergleich zu haben, dann hätte er unterschiedliche Möglichkeiten: Er könnte gängige Tools verwenden - die bereits im Bereich der Softwarevisualisierung existieren - oder er müsste alles zur Erstellung der Visualisierung selber implementieren.
Bei den in der Literatur gängigen Methoden würden sich die CodeCities zur Visualisierung dieser Problemstellung eignen, da die Gebäude die Klassen repräsentieren und die Höhe der Gebäude durch die Anzahl der Methoden festgelegt wird. Doch die Visualisierung würde keinerlei Prozentangaben bieten und hätte darüber hinaus noch einiges an zusätzlichem Inhalt. Dies sorgt zum einen dafür, dass die Visualisierung für das gegebene Problem an Übersichtlichkeit verliert und zum anderen dafür, dass der Auftraggeber für den Visualisierungsinhalt zusätzliche Informationen benötigt - beispielsweise die Anzahl an Lines of Code der Klassen, die die Farbintensität der Gebäude festlegen.
Um eine für seine Problemstellung passende Visualisierung zu erstellen, bliebe ihm also noch die zweite Möglichkeit - die Visualisierung komplett selber zu implementieren. Dies würde einen sehr großen Aufwand bedeuten: Er müsste die notwendigen Daten - also die Anzahl der Methoden zugeordnet zu der jeweiligen Klasse - auslesen und er müsste einen Renderer implementieren, der ihm diese Daten in einer geeigneten Visualisierungsart darstellt. Als Visualisierungsart würde sich in diesem Fall ein Tortendiagramm anbieten, da

dieses prozentuale Verteilungen sehr übersichtlich und auf einen Blick erfassbar darstellen kann. Würde sich die Problemstellung jedoch ein wenig ändern, z.B. wenn er eine prozentuale Verteilung der Methoden auf die einzelnen Klassen *in einem Paket* haben wollte, so hätte er den gesamten Aufwand noch einmal, da er nun pro Paket ein Tortendiagramm bräuchte. Auch wenn sich die Struktur der Ausgangsdaten ändern würde, käme der gesamte Aufwand noch einmal auf den Auftragnehmer zu. Er hätte somit kaum Möglichkeiten der Wiederverwendbarkeit.

Dieses Problem soll in der Bachelorarbeit gelöst werden: Dafür soll ein allgemeiner Ansatz zur Softwarevisualisierung entwickelt werden, der es erlaubt zu *beliebigen Ausgangsdaten* eine Visualisierung mit *beliebigem Visualisierungsinhalt* und einer *beliebigen Visualisierungsart* zu erstellen. Dabei sollen Visualisierungsart und Visualisierungsinhalt als *getrennte* Artefakte vorliegen, die beliebig kombinierbar und wiederverwendbar sind - sobald sie einmal erzeugt wurden. Dieser Zusammenhang ist in Abb. 1.1 dargestellt.

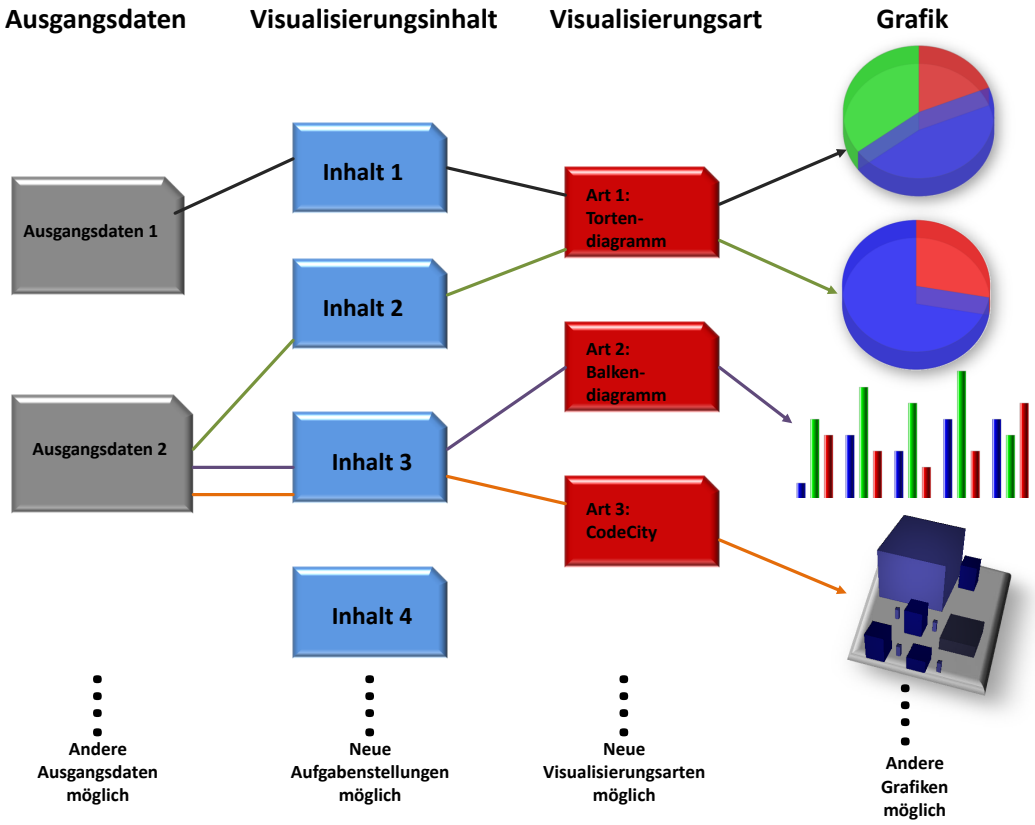


Abbildung 1.1: Schematische Darstellung der Zielsetzung der Bachelorarbeit.

In Abb. 1.1 ist zu erkennen, dass es im angestrebten Ansatz möglich sein soll, Visualisierungsinhalt - also eine bestimmte zu illustrierende Aufgabenstellung wie die Darstellung der prozentualen Verteilung von Methoden auf einzelne Klassen eines Systems - und die

Visualisierungsart - wie beispielsweise ein Tortendiagramm - zu spezifizieren. Diese Spezifikation soll in Artfakten gespeichert werden, so dass diese beliebig kombiniert und wiederverwendet werden können: So liegen in Abb. 1.1 als Visualisierungsart Tortendiagramme, Balkendiagramme und CodeCities vor und weitere Arten können über den Ansatz neu spezifiziert werden.
Wie in Abb. 1.1 zu erkennen, können diese Artefakte dann mit beliebigen Ausgangsdaten kombiniert und wiederverwendet werden um eine entsprechende Grafik zu erzeugen: Beispielsweise werden in Abb 1.1 die „Ausgangsdaten2" mit Inhalt 2 als Tortendiagramm dargestellt. Inhalt 2 kann dabei beispielsweise die prozentuale Verteilung der Methoden auf einzelne Klassen des Systems sein. Außerdem werden die „Ausgangsdaten 2" mit Inhalt 3 einmal als Balkendiagramm und einmal als CodeCity dargestellt. Inhalt 3 kann dabei eine beliebige andere Aufgabenstellung sein - beispielsweise die Darstellung der Attribute und die Anzahl der Statements verteilt auf die Klassen des betrachteten Systems.

Zusammenfassend lässt sich somit das Ziel der vorliegenden Arbeit in folgendem Satz festhalten:

In der vorliegenden Arbeit soll ein absolut generischer Ansatz zur Visualisierung von beliebigen Daten entwickelt werden, bei dem eine *beliebige* Visualisierungsart und *beliebiger* Visualisierungsinhalt *unabhängig* voneinander spezifiziert werden können und für unterschiedliche Ausgangsdaten *wiederverwendbar* sind.

Die allgemeine Herangehensweise zur Erarbeitung dieses Visualisierungsansatzes, die in der Bachelorarbeit zur Lösung des Problems verfolgt wurde, wird im nächsten Abschnitt kurz beschrieben.

1.2 Herangehensweise

Im Rahmen der Bachelorarbeit wurde zur Lösung des beschriebenen Problems folgende Herangehensweise gewählt: Im ersten Schritt wurden Machbarkeitsstudien entwickelt. Diese lieferten einen ersten Überblick über die Möglichkeiten, wie eine neue Visualisierung - bestehend aus Visualisierungsart und -inhalt - überhaupt erstellt werden kann und wie dies technisch realisierbar ist.
Dabei entstanden zwei Machbarkeitsstudien, die unterschiedliche Techniken verwenden: Die erste Machbarkeitsstudie basierte auf einem allgemeinen Java-Ansatz, der Klassen verwendete - beispielsweise eine Klasse „Pie" für eine Torte eines Tortendiagramms. Diese besaß wiederum eine Liste an Tortenstücken. Zur Erstellung einer neuen Grafik, konnten die dazugehörigen Instanzen anschließend mit den entsprechenden Werten, die zum Visualisierungsinhalt passen, belegt werden. Diese Daten sollten abschließend von einem passenden Renderer verarbeitet werden.

In einer zweiten Machbarkeitsstudie wurde dieser Prozess bereits geringfügig automatisiert, indem die Klassen und Klasseninstanzen nicht selber erzeugt werden mussten, sondern dies über eine Transformation erfolgte.
Aufgrund ihres Umfanges wurden diese beiden Machbarkeitsstudien nicht mit in die eigentliche Bachelorarbeit aufgenommen. Die Dokumentation der Machbarkeitsstudien befindet sich jedoch auf der beigelegten CD.
Aus den beiden Machbarkeitsstudien konnte danach ein *allgemeiner* Visualisierungsansatz entwickelt werden, der das beschriebene Problem löst. Dieser Ansatz wird in den nachfolgenden Kapiteln detailliert beschrieben. Darüber hinaus wurden Arbeitsschritte und ein Rollenkonzept erarbeitet, was in Kapitel 2.2 und 2.3 beschrieben wird. Diese spezifizieren die Arbeit mit dem entwickelten Visualisierungsansatz genauer und ermöglichen eine Arbeitsteilung und somit eine Konzentration auf jeweilige Fachgebiete bei der Erstellung einer neuen Visualisierung und Grafik an.
Abschließend wurde der entwickelte Ansatz validiert, indem eine konkrete Implementierung der notwendigen Werkzeuge erfolgte. Diese wurde dazu verwendet, um die Arbeitsschritte des Ansatzes für ein konkretes, real existierendes System zu durchlaufen und dieses so auf unterschiedliche Art und Weise „sichtbar" zu machen. Die Ergebnisse dieser Validierung werden in Abschnitt 4 präsentiert.
Zum Abschluss der Arbeit werden die aufgetretenen Herausforderungen noch einmal betrachtet, die gefundene Lösung darauf aufbauend bewertet und ein Ausblick, wie dieses Verfahren in Zukunft noch weiterentwickelt werden kann, präsentiert.

2 | ELVIZ - Der neue Visualisierungsansatz

Aus der zuvor genannten Problemstellung geht hervor, dass als Ziel der Bachelorarbeit ein Ansatz entwickelt werden soll, der schnell und automatisiert zu *beliebigen Ausgangsdaten* eine *beliebige Visualisierung* - bestehend aus Visualisierungsinhalt und Visualisierungsart - erstellt. In der vorliegenden Bachelorarbeit wurde dazu ein neuer Visualisierungsansatz - der ELVIZ-Ansatz (Every Layout Visualization approach) - entwickelt, der diese Kriterien ohne Einschränkungen erfüllt. Dieser Ansatz ist dabei nicht nur vollständig generisch bezüglich Visualisierungsinhalt, -art und Beschaffenheit der Ausgangsdaten, sondern kann darüber hinaus alle vorhandenen Techniken zur Implementierung ausschöpfen.
Im nachfolgenden Abschnitt erfolgt zunächst eine grobe Übersicht über die Bestandteile des ELVIZ-Ansatzes. Danach werden die Arbeitsschritte, die zur Erstellung einer neuen Grafik durchlaufen werden müssen, dargestellt und ein Rollenkonzept zur Arbeitsteilung präsentiert. Anhand der Arbeitsschritte wird dann in Abschnitt 2.4 detailliert ein Beispielablauf beschrieben. An diesem wird außerdem verdeutlicht, wie es zur Entscheidung für die einzelnen Bestandteile kam und welche konkreten Realisierungsmöglichkeiten sich an diesen Stellen anbieten.

2.1 Übersicht

Aus der Problemstellung geht hervor, dass sich sehr viele Daten - seien es Ausgangsdaten, die Visualisierungsart oder der Visualisierungsinhalt - *stetig verändern*. Aus diesem Grund wird für den zu entwickelnden Softwarevisualisierungsansatz eine Vorgehensweise benötigt, die sehr schnell und automatisiert auf Veränderungen reagieren kann:
Was sich an dieser Stelle anbietet ist die „Architecture of Choice for a **Changing** World" [24]. Mit diesen Worten wirbt die OMG (Object Management Group) um den Ansatz der Model Driven Architecture. Aus diesem Grund wurde entschieden, dass der ELVIZ-Ansatz eine modellgetriebene Arbeitsweise benötigt, wie sie im MDA-Ansatz präsentiert wird. Diese Art der Entwicklung und wie sie im ELVIZ-Ansatz verwendet wird, wird im nachfolgenden Abschnitt genauer erläutert.

2.1.1 Die Model Driven Architecture im Visualisierungsansatz

Bei der modellgetriebenen Entwicklung, die hauptsächlich im Jahr 2000 durch Einführung des MDA-Standards (Model Driven Architecture) der OMG aufkam, geht es darum bestimmte Teile von Software oder auch die gesamte Software automatisch aus Modellen zu generieren. Die Model Driven Architecture der OMG ist dabei nur eine Variante des Model Driven Engineerings (MDE) [2], bei der es darum geht aus einem plattformunabhängigen Modell (PIM) ein plattformspezifisches Modell (PSM) zu transformieren [18] (siehe Abb. 2.1). Dies hat den Vorteil, dass bei der Spezifizierung des plattformunabhängigen Modells ein höherer Abstraktionsgrad vorliegt, da hier unabhängig von plattformspezifischen Implementierungsdetails eine Konzentration auf den wesentlichen fachlichen Teil ermöglicht wird. Die Generierung des plattformspezifischen Modells soll im MDA-Ansatz dann automatisch durch eine Transformation erfolgen. Dadurch erhofft man sich eine höhere Produktivität und eine geringere Reaktionszeit bei Designänderungen [22]. Dies ist genau der Grund, aus dem die OMG den MDA-Ansatz als „Architecture of Choice for a Changing World" [24] bezeichnet.

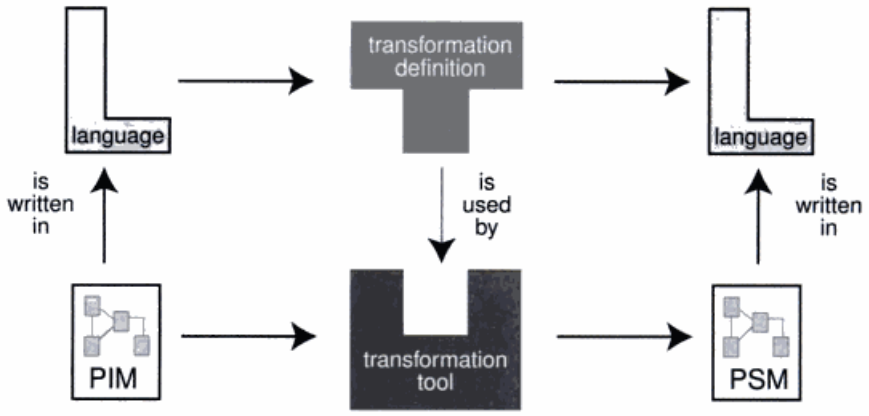


Abbildung 2.1: Der Ansatz der Model Driven Architecture entnommen aus dem Buch MDA Explained [18].

Da Modelle nicht nur im MDA-Ansatz, sondern im gesamten MDE-Konzept das Kernelement bilden, bezeichnet Jean Bézivin in seiner Veröffentlichung über MDE im UPGRADE, dem „Journal European for the Informatics Professional", im April 2004, dass das Grundprinzip im Vergleich zur objektorientierten Programmierung, bei der alles als Objekt betrachtet wird, darin besteht, dass alles als Modell angesehen wird: „Everything is a model" [2]. Im Prinzip wird es so möglich aus Modellen andere Modelle zu transformieren und so automatisch Quellcode - der in diesem Zusammenhang ebenfalls als Modell angesehen werden - zu erhalten. Das setzt jedoch voraus, dass die Modelle so präzise sind, dass sie automatisch interpretiert werden können [12].

In der Bachelorarbeit eignet sich der MDA-Ansatz an dieser Stelle sehr gut: Er bringt die nötige Wiederverwendbarkeit in den Ansatz, indem die Visualisierungsart als Sprache und dementsprechend als Metamodell für das PSM angesehen wird. Damit wird es möglich die zu visualisierenden Fakten, die bereits Informationen über Visualisierungsart und Visualisierungsinhalt beinhalten, stets sehr schnell aus unterschiedlichen Ausgangsdaten **automatisch** zu transformieren. Mit dieser modellgetriebenen Arbeitsweise konnte der ELVIZ-Ansatz entwickelt werden, der in Abb. 2.2 dargestellt ist.

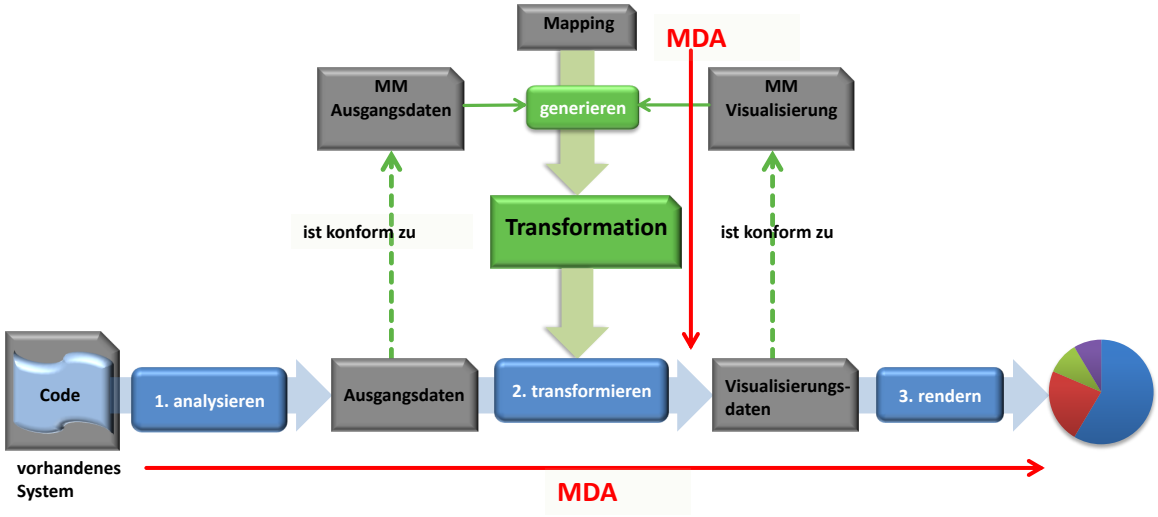


Abbildung 2.2: Der ELVIZ-Ansatz zur Erstellung einer Visualisierung.

Die Abb. 2.3 liefert eine Übersicht über die im ELVIZ-Ansatz vorkommenden Begriffe, die in der vorliegenden Bachelorarbeit fortwährend verwendet werden. Dies dient zunächst der ersten Orientierung. Eine detaillierte Beschreibung mit Realisierungsmöglichkeiten für die einzelnen Bestandteile wird in Abschnitt 2.4 gegeben.

Bei der Betrachtung der Grafik zum MDA-Ansatz aus Abb. 2.1 lassen sich die weiteren Bestandteile des MDA-Ansatzes den Bestandteilen des ELVIZ-Ansatzes aus Abb. 2.2 eindeutig zuordnen: Beim PIM, dem plattformunabhängigen Modell, handelt es sich somit um die Ausgangsdaten. Diese sind plattformunabhängig, da sie noch keine Informationen über die Visualisierungsart beinhalten. Die Ausgangsdaten sind nach Abb. 2.1 in einer bestimmten Sprache geschrieben. Dies ist genau das Metamodell der Ausgangsdaten.
Die Visualisierungsdaten bilden das plattformspezifische Modell, da diese intern bereits Informationen über die Visualisierungsart besitzen. Die Visualisierungsart wird dabei durch das Metamodell der Visualisierung beschrieben - die Spezifizierung der Visualisierungsart über das Visualisierungsmetamodell entspricht in diesem Fall somit der Sprache des PSM.

Begriff	Bedeutung
Code	Quellcode des Systems, von dem bestimmte Fakten und Zusammenhänge visualisiert werden sollen
Ausgangsdaten	Fakten über das vorhandenen Systems in geeigneter Datenstruktur
MM Ausgangsdaten	Beschreibt die Struktur, in der die Ausgangsdaten vorliegen
Transformation	Dient zur Überführung der Ausgangsdaten in die Visualisierungsdaten
Visualisierungsdaten	Ausgewählte Fakten des Systems, die bereits in der zur Visualisierungsart passenden Struktur vorliegen
MM Visualisierungsdaten	Das Metamodell der Visualisierungsdaten beschreibt die Visualisierungsart und gibt damit den Visualisierungsdaten die dazugehörige Struktur vor
Mapping	Enthält Informationen über den Visualisierungsinhalt, indem hier definiert wird, welche Elemente des MM der Ausgangsdaten auf welche Elemente des MM der Visualisierungsdaten abgebildet werden

Abbildung 2.3: Übersicht über die Begriffe des ELVIZ-Ansatzes.

Aus Abb. 2.2 geht hervor, dass MDA dabei nicht nur horizontal zur Generierung der Visualisierungsdaten eingesetzt wird, sondern auch vertikal zur Generierung der Transformation. Beide Dimensionen - die *horizontale* Verwendung von MDA und die *vertikale* Verwendung von MDA - werden im nachfolgenden genauer beschrieben.

MDA vertikal

Die Idee der Model Driven Architecture wird zunächst dazu verwendet die Transformation automatisch zu generieren: Hierzu werden das Metamodell der Ausgangsdaten, das Mapping sowie das Metamodell der Visualisierungsdaten als Ausgangsmodell genutzt. Über eine beliebige Transformationssprache ist es möglich aus diesen Informationen automatisch eine ausführbare Transformation zu generieren. Die so erzeugte Transformation, wird dann in der horizontalen Verwendung von MDA eingesetzt.

MDA horizontal

Bei der horizontalen Anwendung der Idee des MDA-Ansatzes wird die zuvor generierte Transformation verwendet: Dabei sollen die Ausgangsdaten, die dem PIM entsprechen, in die Visualisierungsdaten, die dem PSM entsprechen, transformiert werden. Dadurch wird es möglich die Visualisierungsdaten komplett automatisch zu erhalten. Diese müssen abschließend lediglich durch einen passenden Renderer in die reale Grafik überführt werden.

Durch einen derartigen Einsatz des Model Driven Engineerings sind die Spezifizierung der Visualisierungsart, des Visualisierungsinhalts und die eigentliche Ausführung zum Erhalt der realen Grafik stark voneinander abgekapselt, bieten dabei aber sehr viel Potential der automatischen Generierung, Wiederverwendung und Kombinierbarkeit unterschiedlicher Ausgangsdaten, Visualisierungsinhalte und Visualisierungsarten. Durch diese Abkapselung lassen sich konkrete Arbeitsschritte und eine Aufteilung dieser Schritt auf unterschiedliche Rollen festlegen. Diese werden im nachfolgenden Abschnitt präsentiert.

2.2 Allgemeine Arbeitsschritte

Im vorherigen Abschnitt wurde die Grundidee des ELVIZ-Ansatzes in einer groben Übersicht erläutert. An dieser Stelle soll nun die generelle Vorgehensweise in diesem Lösungsansatz definiert werden:
Aus diesem Grund wird eine Reihe von Arbeitsschritten definiert, die zur Erstellung einer neuen Grafik im ELVIZ-Ansatz durchlaufen werden müssen. Als Ergebnis können in jedem Arbeitsschritt unterschiedliche Artefakte erhalten werden. Da die Wiederverwendbarkeit dieser Artefakte Voraussetzung war, bedarf es zur Erstellung einer Grafik nicht das erneute Durchlaufen aller Arbeitsschritte.
In Abb. 2.4 ist eine Übersicht über die Arbeitsschritte und die daraus erhaltenen Artefakte dargestellt. Wie oft dabei ein bestimmter Arbeitsschritt gemacht werden muss, ist auf der linken Seite in Abb. 2.4 gezeigt.

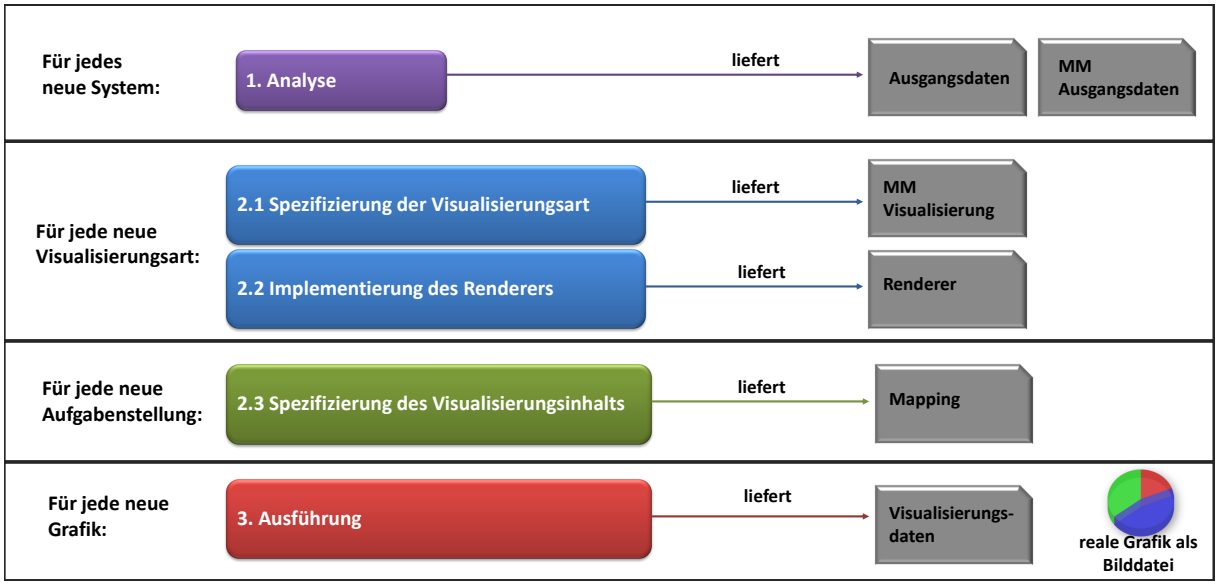


Abbildung 2.4: Arbeitsschritte des ELVIZ-Ansatzes und daraus gewonnene Artefakte.

In Abb. 2.4 ist zu sehen, dass der **erste Schritt** - die Analyse - zunächst darin besteht die Ausgangsdaten in einer geeigneten Datenstruktur für die nachfolgende Transformation bereitzustellen. Dies kann über eine Analyse des vorhandenen Systems geschehen. Hierzu können beispielsweise vorhandene Analysewerkzeuge genutzt werden, die den Quellcode eines Systems parsen und die extrahierten Fakten in geeigneter Datenstruktur als Ergebnis liefern.
Dieser Schritt muss *für jedes neue Softwaresystem* durchlaufen werden, für das eine visuelle Darstellung erstellt werden soll. Dies liegt daran, dass die Ausgangsdaten einmal in einer geeigneten Datenstruktur zur Verfügung gestellt werden müssen, damit sie überhaupt zur Erzeugung von Visualisierungsdaten in einer Transformation verfügbar sind.

Dies muss allerdings *nur einmal pro betrachtetem Softwaresystem* geschehen, da diese Daten natürlich Fakten über das gesamte System beinhalten können. Danach können diese Daten - da der Ansatz durch Verwendung von MDA mit einer Transformation arbeitet - für alle beliebigen Visualisierungsmöglichkeiten wiederverwendet werden. Sollte sich das System ändern, so muss an dieser Stelle eine neue Analyse durchgeführt werden.
Als Artefakte werden in diesem ersten Schritt die Ausgangsdaten sowie das Metamodell der Ausgangsdaten geliefert.

Der gesamte **zweite Schritt** befasst sich mit der Spezifizierung der Visualisierung: Dieser kann unterteilt werden in die Spezifizierung der Visualisierungsart (2.1) - die zunächst festlegt mit welchen visuellen Mitteln die Daten dargestellt werden sollen -, die Implementierung des dazugehörigen Renderers (2.2), der zu der spezifizierten Visualisierungsart die reale Grafik erzeugen kann und die Spezifizierung des Visualisierungsinhalts (2.3) - der festlegt, *was* dargestellt wird.
In **Schritt 2.1** muss zunächst die Visualisierungsart spezifiziert werden: Dies geschieht, indem die Elemente, die eine bestimmte Visualisierungsart haben soll, in einem Metamodell definiert werden. Dieser Schritt hat somit als Ergebnis, dass ein neues Visualisierungsmetamodell vorliegt, welches eine neue Visualisierungsart charakterisiert.
Eng damit verbunden ist der **Schritt 2.2**: Wird eine neue Visualisierungsart definiert, muss natürlich auch ein neuer Renderer implementiert werden, der in der Lage ist, diese Visualisierungsart in einer realen Grafik umzusetzen. Die Erstellung des Visualisierungsmetamodells und die Implementierung des Renderers müssen *einmal für jede neue Visualisierungsart* gemacht werden. Dieser Schritt muss so beispielsweise nur einmal für Tortendiagramme, einmal für Balkendiagramme, einmal für Liniendiagramme etc. durchlaufen werden. Wurde eine Visualisierungsart einmal über das Visualisierungsmetamodell definiert und der passende Renderer implementiert, so können diese beiden Artefakte danach immer wiederverwendet werden.
Zur Spezifizierung der Visualisierung gehört allerdings nicht nur die Art der Darstellung, sondern auch ihr Inhalt: Dies erfolgt in **Schritt 2.3** über die Spezifizierung des Visualisierungsinhalts. Folglich wird an dieser Stelle eine bestimmte Aufgabenstellung definiert - beispielsweise die Darstellung der prozentualen Verteilung von Methoden auf die einzelnen Klassen eines Systems. Als Artefakt wird in diesem Schritt das Mapping geliefert: Dieses sagt aus, welche Elemente aus dem Metamodell der Ausgangsdaten auf welche Elemente im Visualisierungsmetamodell abgebildet werden soll. Eine detaillierte Erläuterung des Mappings befindet sich in Abschnitt 2.9.
Die Spezifizierung des Visualisierungsinhalts muss *einmal pro Aufgabenstellung* durchlaufen werden. Existiert beispielsweise das Mapping für die Aufgabenstellung, dass eine prozentuale Verteilung der Methoden auf einzelne Klassen dargestellt werden soll, so kann dieses Mapping auch für unterschiedliche Visualisierungsarten wiederverwendet werden. Dies geschieht allerdings unter der Voraussetzung, dass das Mapping so allgemein spezifiziert wurde, dass es auf unterschiedliche Ausgangsdaten und unterschiedliche Visualisierungsarten angewendet werden kann.

Im **dritten Schritt** - der Ausführung - muss die Transformation nur noch generiert und ausgeführt werden sowie durch das Rendering die reale Grafik erzeugt werden. In diesem Schritt könnte ein Benutzer folglich vorhandene Ausgangsdaten auswählen, sich für eine vorhandene Visualisierungsart entscheiden und zur Generierung der Transformation das dazugehörige, vorhandene Visualisierungsmetamodell verwenden sowie ein bereits vorhandenes Mapping zur Spezifizierung des Visualisierungsinhalts. Damit könnte in diesem Schritt durch die modellgetriebene Arbeitsweise des ELVIZ-Ansatzes die Transformation automatisch generiert werden. Diese müsste abschließend nur noch ausgeführt werden und der zur Visualisierungsart passende Renderer müsste im abschließenden Rendering, die in der Transformation gewonnenen Visualisierungsdaten in die reale Grafik überführen.
Dieser letzte Schritt muss *einmal pro reale Grafik* durchlaufen werden, da es sich hierbei um die abschließende Ausführung zur Erstellung der realen Grafik handelt.

Diese Arbeitsschritte machen deutlich, dass unterschiedliche Aufgaben zur Erstellung einer neuen Grafik notwendig sind. Diese stammen dabei aus komplett unterschiedlichen Fachgebieten. Aus diesem Grund bietet sich für den in der Bachelorarbeit entwickelten ELVIZ-Ansatz ein Rollenkonzept an: Damit können sich die unterschiedlichen Rollen auf ihre Fachgebiete konzentrieren und die in ihrem Arbeitsschritt gewonnen Artefakte können von anderen Rollen in anderen Arbeitsschritten dieses Ansatzes wiederverwendet werden. Dieses Rollenkonzept wird im nächsten Abschnitt präsentiert.

2.3 Rollenkonzept

Bei der Betrachtung der Arbeitsschritte lassen sich vier unterschiedliche Fachbereiche erkennen: Die Analyse, die Modellierung, die Implementierung und die Ausführung. Da die in den ersten vier Arbeitsschritten produzierten Artefakte (Ausgangsdaten, Metamodell der Ausgangsdaten, Visualisierungsmetamodell, Mapping) im letzten Arbeitsschritt beliebig kombinierbar sind, sollte an dieser Stelle zur Erhöhung der Produktivität eine Unterteilung der Arbeitsschritte auf unterschiedliche Rollen stattfinden. So kann eine Konzentration auf die jeweiligen Fachbereiche stattfinden und dadurch eine Erhöhung der Produktivität erreicht werden.
Durch die vier unterschiedlichen Fachbereiche können vier Rollen identifiziert werden: *den Analysierer*, *den Modellierer*, *den Implementierer* und *den Ausführer*. Die Aufgaben dieser unterschiedlichen Rollen werden im nachfolgenden anhand der zuvor definierten Arbeitsschritte des ELVIZ-Ansatzes erläutert.

Der Analysierer

Der Analysierer bearbeitet den **ersten Schritt** - die Analyse: Er stellt damit die Ausgangsdaten eines vorhandenen Systems und das dazugehörige Metamodell bereit. Diese Rolle muss sich somit mit vorhandenen Analysewerkzeugen und passenden Datenstrukturen für die Ergebnisse der Analyse auskennen.

Der Modellierer

Der Modellierer übernimmt **Schritt 2.1** - die Spezifizierung der Visualisierungsart. Diese Rolle muss sich nicht nur eine sinnvolle Visualisierungsart überlegen, sondern sollte darüberhinaus mit der Metamodellierung vertraut sein, da die Visualisierungsart präzise definiert werden muss. Diese Rolle erstellt somit das Visualisierungsmetamodell für eine neue Visualisierungsart, welches daraufhin in Transformationen immer wiederverwendbar ist.

Der Implementierer

Der Implementierer kümmert sich um die Implementierung des Renderers (**Schritt 2.2**). Seine Aufgabe ist es, den neuen Renderer so zu implementieren, dass Visualisierungsdaten, die zu einer neuen Visualisierungsart konform sind, automatisch in eine reale Grafik überführt werden können. Diese Rolle sollte sich nicht nur mit diversen grafischen Implemen-

tierungsmöglichkeiten auskennen, sondern muss außerdem mit dem Visualisierungsmetamodell vertraut sein: Dabei muss er zum einen die abstrakte Syntax der Visualisierungsart kennen, wie sie vom Modellierer im Visualisierungsmetamodell spezifiziert wurde und zum anderen muss er wissen, wie die konkrete Syntax der Elemente aus dem Visualisierungsmetamodell aussieht, damit er diese Elemente in ihre entsprechende visuelle Darstellung überführen kann.

Der Ausführer

Der Ausführer bestimmt abschließend, wie die reale Grafik aussehen soll. Dazu gehören drei Aspekte: Es muss eine Entscheidung für eine Visualisierungsart getroffen werden, die Ausgangsdaten für das zu visualisierende System müssen gewählt werden und abschließend muss der Visualisierungsinhalt definiert werden. Aus diesem Grund übernimmt diese Rolle **Schritt 2.3** - das Mapping - und den letzten Schritt - die Ausführung zur Erstellung der realen Grafik.

Dieses Rollenkonzept verdeutlicht noch einmal die hohe Wiederverwendbarkeit einzelner Artefakte aus den unterschiedlichen Arbeitsschritten des hier entwickelten ELVIZ-Ansatzes.

Im nachfolgenden Abschnitt wird anhand eines Beispiels verdeutlicht, wie die einzelnen Bestandteile realisiert werden können und wie die dazugehörigen Artefakte in diesem Fall aussehen können.

2.4 Detaillierte Erläuterung des ELVIZ-Ansatzes

Der ELVIZ-Ansatz wird in diesem Kapitel der Bachelorarbeit detailliert beschrieben. Dazu wird das allgemeine Konzept dieses Ansatzes anhand eines Beispiels eingehend erläutert: Ein Auftragnehmer strebt an, eine visuelle Darstellung der Methoden auf die einzelnen Klassen eines Systems zu entwickeln - wobei ihm vollkommene Auswahlfreiheit in Bezug auf die Visualisierungsart gelassen wird.
Um dieses Beispiel möglichst übersichtlich zu halten, wird an dieser Stelle ein sehr kleines fiktives System verwendet. Dieses besteht lediglich aus einem Paket mit dem Namen „package1". In diesem Paket befinden sich zwei Klassen: „class1" und „class2", wobei „class1" zwei Methoden hat und „class2" lediglich eine Methode.
Dieses fiktive System dient an dieser Stelle lediglich zum Verstehen des ELVIZ-Ansatzes. Die gleiche Problemstellung - die prozentuale Verteilung der Methoden auf einzelne Klassen eines Systems zu visualisieren - wird jedoch auch noch einmal in Kapitel 4 zur Validierung der Lösung auf ein reales System angewendet.

2.4.1 Die Ausgangsdaten

Der Begriff „**Ausgangsdaten**" beschreibt die Daten, die die entsprechenden Informationen über ein zu betrachtendes System liefern: In den Ausgangsdaten befinden sich somit Daten und Fakten über das betrachtete System. In dem hier gewählten Beispiel beschreiben die Ausgangsdaten somit die Fakten des gewählten fiktiven Systems: Diese beinhalten, dass das System ein Paket mit dem Namen „package1" hat, welches wiederum aus zwei Klassen besteht - wobei eine Klasse zwei Methode hat und die andere Klasse nur eine Methode.
Diese Fakten müssen zunächst in einer geeigneten Form vorliegen, damit Teile von ihnen überhaupt als Visualisierungsinhalt verwendet werden können. Eine mögliche Struktur, die sich im hier betrachteten Beispiel und insbesondere auch für weitere Visualisierungen anbietet, sind *TGraphen* [6] [8]. Es kann im hier entwickelten Visualisierungsansatz jedoch auch jede andere Struktur für die Ausgangsdaten gewählt werden, da diese - wie im Ziel der Arbeit beschrieben - *beliebig* sein sollen. Warum sich gerade TGraphen anbieten und worum es sich bei dieser Datenstruktur handelt, wird im nachfolgenden Abschnitt genauer erläutert.

TGraphen

TGraphen wurden ursprünglich an der Universität Koblenz-Landau in der Arbeitsgruppe Ebert entwickelt [6] [8]. Der TGraph, der das im Beispiel verwendetet fiktive System beschreibt, ist in Abb. 2.5 zu sehen.

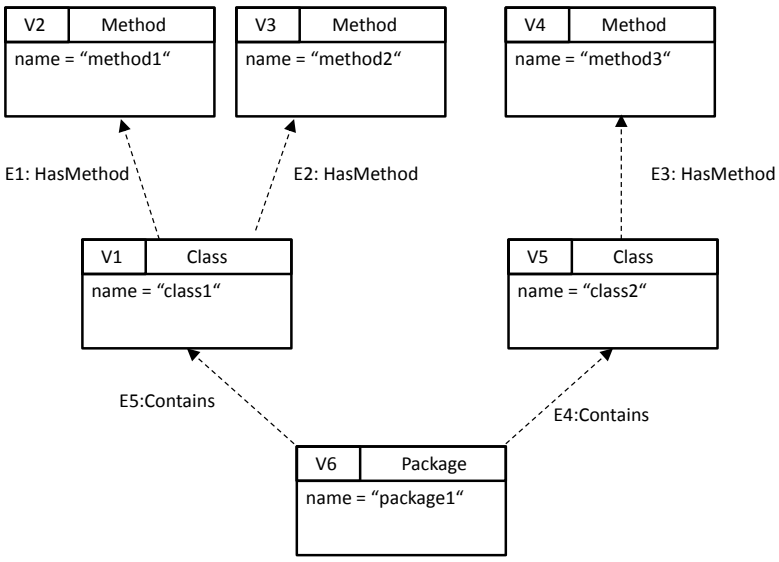


Abbildung 2.5: Beispielhafter TGraph des fiktiven Systems.

An diesem TGraphen sind bereits die grundsätzlichen Eigenschaften von TGraphen erkennbar: So handelt es sich bei TGraphen um „angeordnete”, „gerichtete” Graphen, die über „typisierte” und „attributierte” Knoten und Kanten verfügen [8]:
So ist im TGraphen der Abb. 2.5 zu sehen, dass die wichtigsten Graphelemente die Knoten und Kanten sind. Diese verfügen alle über einen entsprechenden *Typ* [4]. So hat der Knoten V1 im dargestellten Beispiel-TGraphen den Typ „Class” und die Kante E1 den Typ „HasMethod”. Dieser Typ legt die Menge der *Attribute* der Knoten und Kanten fest: So verfügen im oberen Beispiel alle Knoten des Typs „Package”, „Class” und „Method” über das Attribut „name”, das mit einem String belegt werden kann.
Da TGraphen darüber hinaus *gerichtete Graphen* sind, weist jede Kante des vorliegenden Graphen aus der Machbarkeitsstudie in eine bestimmte Richtung. Dadurch können im TGraphen der Abb. 2.5 die Beziehungen zwischen einzelnen Knoten herausgestellt werden: Beispielsweise ist die Methode mit dem Namen ‚method1”, die im Graphen über den Knoten V2 repräsentiert wird, eine Methode der Klasse „class1”, welche wiederum durch den Knoten V1 im TGraphen dargestellt ist. Diese Beziehung wird dabei durch die gerichtete Kante E1 vom Typ „HasMethod” illustriert, die von V1 auf V2 weist.
Neben dem Besitz von Typen, Attributen und gerichteten Kanten, können TGraphen auch noch *angeordnet* sein. Das bedeutet, dass die Menge der Knoten und Kanten geordnet ist und so auf den zu einem Knoten inzidenten Kanten eine Ordnung existiert [30]. Damit haben die in ein Graphelement ein– bzw. ausgehenden Kanten eine definierte Reihenfolge [17]. Beispielsweise geht von Package zuerst eine Kante zu „class1” und an zweiter Stelle eine Kante zum Knoten V5 mit Namen „class2”.

TGraphen eignen sich an dieser Stelle besonders gut als Datenstruktur für die Ausgangsdaten, da die Fakten über Quellcode sehr gut in einem Graphen darstellbar sind. Darüber hinaus existiert an der Universität Oldenburg bereits ein Analysewerkzeug, welches ein vorhandenes Java-System parsen kann und die Fakten über den Quellcode in Form eines TGraphen als Ergebnis liefert. Der Visualisierungsansatz ließe sich unter Verwendung von TGraphen somit sehr gut mit diesem vorhandenen Analysewerkzeug verbinden und würde diesem folglich als „Model Driven Software Visualization Service" dienen.
Der in der Bachelorarbeit entwickelte ELVIZ-Ansatz ist jedoch so allgemein gehalten, dass an dieser Stelle auch andere Datenstrukturen zur Repräsentation des Quellcodes möglich sind. Da der Ansatz so generisch gehalten wurde, ist es darüber hinaus sogar möglich jede Art von Daten zu visualisieren, das heißt, es muss sich an dieser Stelle nicht mehr unbedingt um Fakten aus einem Quellcode handeln.

Die Ausgangsdaten könnten an dieser Stelle direkt als Datenquelle für einen Renderer verwendet werden. Doch damit würde jede Aussicht auf Wiederverwendbarkeit der Visualisierungsart zunichte gemacht werden. Der Softwarevisualisierungsansatz muss an dieser Stelle somit ein Konzept bieten, dass die Wiederverwendbarkeit von Visualisierungsarten für die vielen unterschiedlichen Ausgangsdaten möglich macht. Aus diesem Grund kommt an dieser Stelle die Idee von MDA und die damit einhergehende Transformation zum Einsatz. Dafür muss jedoch das Metamodell der Ausgangsdaten vorliegen:
Im Fall des hier betrachteten Beispiels ist dementsprechend eine „Sprache" gesucht, die den TGraphen - in diesem Fall das plattformunabhängige Modell -, beschreibt, damit dies zur Realisierung der Transformation für die Visualisierung des Auftragnehmers im Beispiel verwendet werden kann. Im Falle von TGraphen ist dies „grUML" - die graph Unified Modeling Language. Diese wird im nachfolgenden Abschnitt genauer erläutert.

grUML - die Graph Unified Modeling Language

Bei grUML handelt sich um eine visuelle Sprache, die Metamodelle von Teilgraphen mittels einer Teilmenge der *Unified Modeling Language* modelliert [20]. Diese Metamodelle beschreiben dabei die Struktur der Daten.
Der vorhergehende Beispiel-TGraph aus Abb. 2.5 repräsentiert einen analysierten Ausschnitt eines einfachen Java-Quellcodes. Aus diesem Grund kann das grUML-Metamodell als Ausschnitt eines Java-Metamodells, wie in Abb. 2.6 gezeigt, dargestellt werden. An Abb. 2.6 ist zu erkennen, dass ein Paket beliebig viele Klassen haben kann und eine Klasse wiederum über beliebig viele Methoden verfügen kann. Dabei haben sowohl Methoden als auch Klassen und Pakete das Attribut „name", das mit einem String belegt werden kann.

Alle möglichen Graphelemente und deren Zusammenhänge lassen sich über das grUML-Metametamodell darstellen (siehe Abb. 2.7).

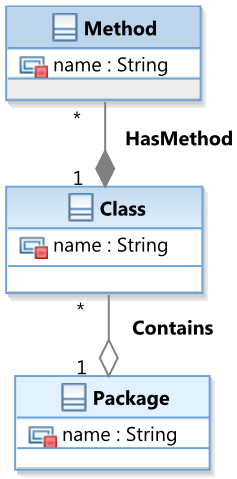


Abbildung 2.6: grUML-Schema des Beispiel-TGraphen.

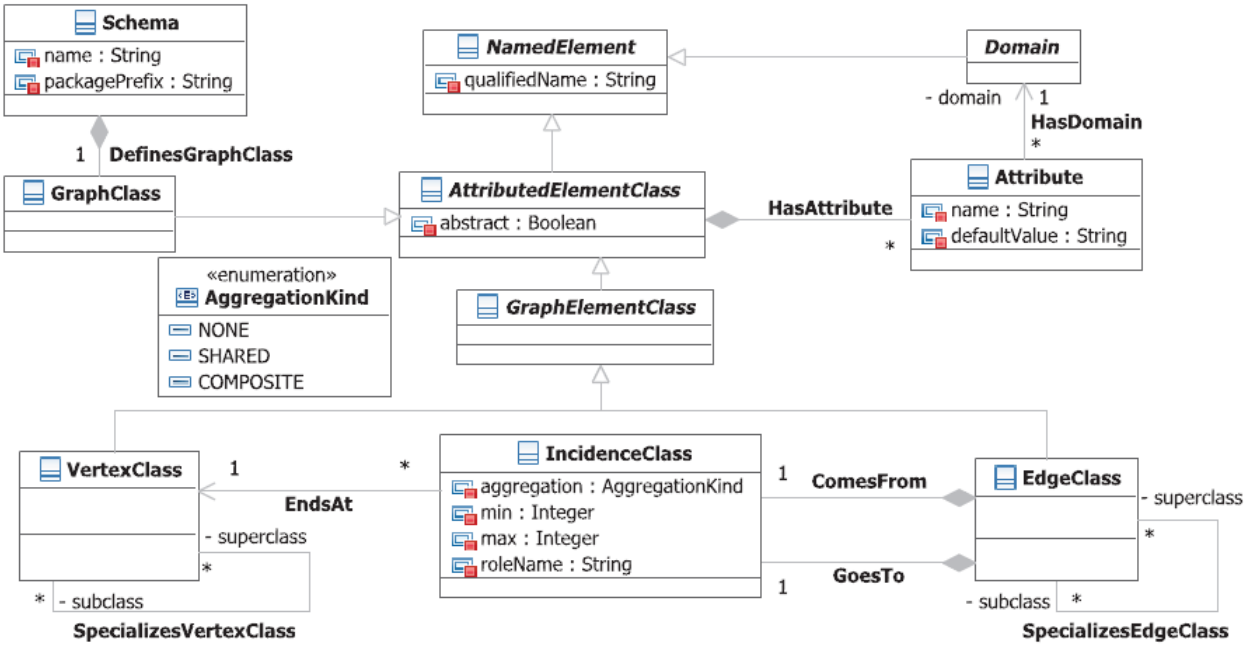


Abbildung 2.7: grUML Metametamodell aus Horn und Ebert [16].

An diesem Metametamodell sind erneut die grundlegenden Eigenschaften eines TGraphen erkennbar: Die Klasse „Schema" repräsentiert hier die Menge aller TGraphen, die dem Metamodell (in der obigen Abbildung wird das Metamodell als „Schema" bezeichnet) genügen. Dieses Schema besteht aus den grundsätzlichen Graphelementen - den Knoten (repräsentiert durch die Klasse „VertexClass") und Kanten (repräsentiert durch „EdgeClass"). Beide verfügen dabei über eine Spezialisierungshierarchie, so dass es zu jedem Knoten und zu jeder Kante wieder Unterknoten bzw. Unterkanten geben kann. Darüber hinaus ist zu sehen, dass sowohl Kanten als auch Knoten über Attribute verfügen können, da es sich bei beiden um Graphelemente („GraphElementClass") handelt, die wiederum von der abstrakten Klasse „AttributedElementClass" generalisiert werden und folglich durch die dargestellte Komposition über Attribute verfügen können.
Dieses grUML-Metametamodell beschreibt somit den Aufbau und die grundlegenden Eigenschaften aller TGraphen.

Nachdem die Ausgangsdaten und die Sprache zur Beschreibung dieser Daten in Form des grUML-Metamodells aus Abb. 2.6 für den Auftragnehmer vorliegen - wird nach Abb. 2.1 die *Transformationsdefinition* benötigt. Diese muss sowohl Informationen über die Visualisierungsart einbeziehen als auch Informationen über den Visualisierungsinhalt. Wie die *Transformation* im ELVIZ-Ansatz verwendet wird und wie die Transformationsdefinition erhalten wird, wird im nachfolgenden Text genauer beschrieben.

2.4.2 Die Transformation

Die *Transformation* ist das Herzstück des hier entwickelten modellgetriebenen Visualisierungsansatzes. Sie dient zur Transformation der Ausgangsdaten in die Visualisierungsdaten. Um diese Transformation ausführen zu können, bedarf es der Transformationsdefinition. Diese benötigt sowohl Informationen über die Visualisierungsart und den Visualisierungsinhalt als auch eine Transformationssprache, die zur Spezifizierung der eigentlichen Transformation verwendet wird. Diese drei Aspekte werden in den nachfolgenden Abschnitten genauer beschrieben.

Spezifizierung der Visualisierungsart - das Visualisierungsmetamodell

Da es sich bei dem in der Bachelorarbeit entwickelten ELVIZ-Ansatz um einen sehr variablen Ansatz handelt, soll die Art der Visualisierung durch den Anwender festgelegt werden können: Hierzu muss die angestrebte Visualisierungsart präzise definiert werden. Dies geschieht wie im MDA-Ansatz beschrieben über die Sprache, die das plattformspezifische Modell - die Visualisierungsdaten - beschreibt. Bei dieser Sprache handelt es sich dementsprechend um ein Metamodell, das als *„Visualisierungsmetamodell"* bezeichnet wird.
Im hier betrachteten Beispiel des Auftragnehmers kommt als gewünschte Visualisierungs-

art das Tortendiagramm in Betracht, da sich dieses sehr gut als Visualisierungsart für prozentuale Verteilungen eignet. Aus diesem Grund muss an dieser Stelle das Metamodell eines Tortendiagramms spezifiziert werden wie es in Abb. 2.8 dargestellt ist.

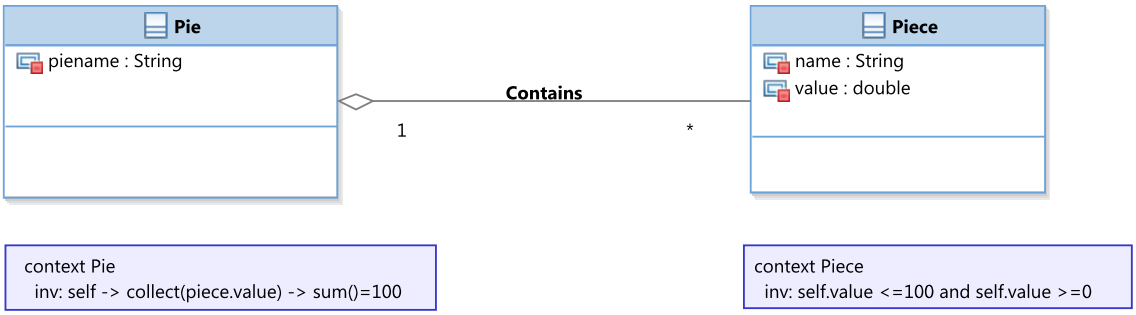


Abbildung 2.8: Metamodell des Tortendiagramms.

Das Metamodell besteht in diesem Fall aus zwei Klassen: Einer Klasse „Pie" für die gesamte Torte und einer Klasse „Piece" für die Tortenstücke. Eine Torte kann einen Namen haben, was über das Attribut „piename" spezifiziert wird, das mit einem String belegt werden kann. Ein „Piece" kann ebenfalls einen Namen haben (Attribut „name") und darüber hinaus hat das Tortenstück einen Wert (Attribut „value"), der die Größe festlegt. Über die zusätzlichen OCL-Constraints wird an dieser Stelle sichergestellt, dass die Summe aller Werte der Tortenstücke einer Torte den Wert 100 nicht überschreitet, da es sich bei der Anzeige der Werte in einem Tortendiagramm um Prozentwerte handelt. Abschließend wird über die Aggregation „Contains" sichergestellt, dass eine Torte beliebig viele Stücke besitzen kann.

Das Metamodell kann an dieser Stelle natürlich auch beliebig anders deklariert werden. Wichtig ist nur, dass das Metamodell für die spätere Transformation präzise definiert zur Verfügung steht.
Allgemein betrachtet wird in dem Visualisierungsmetamodell somit die Struktur der Visualisierungsdaten festgelegt, indem über entsprechende Klassen definiert wird, welche Elemente die Visualisierung haben kann - beispielsweise Torten und Tortenstücke - und über die Attribute können die Eigenschaften, die diese Elemente haben können, festgelegt werden. Hierzu gehören beispielsweise Namen und Größenangaben. Abschließend dienen die Assoziationen wie gewohnt dazu, die Beziehungen zwischen den einzelnen Elementen der Visualisierung zu beschreiben.
Insgesamt kann so über das Visualisierungsmetamodell die Art der Visualisierung präzise spezifiziert werden. Es sind mit dem ELVIZ-Ansatz somit alle nur denkbaren Visualisierungsarten möglich, die über ein Metamodell spezifiziert werden können. Die angestrebten Visualisierungsdaten, die das Ziel der Transformation sind, sind dann entsprechend konform zu diesem Visualisierungsmetamodell.

Spezifizierung des Visualisierungsinhalts - das Mapping

Neben der Spezifizierung der Visualisierungsart soll zusätzlich der Inhalt der Visualisierung frei spezifizierbar sein. Der Visualisierungsinhalt umfasst die Information welches Element des Metamodells der Ausgangsdaten durch welches Element des Visualisierungsmetamodells repräsentiert werden soll: Im Gegensatz zur Spezifizierung der Visualisierungsart, welche beschreibt *wie* die Inhalte dargestellt werden sollen, wird über das Mapping der konkrete Inhalt der Visualisierung festgelegt - also *was* in der Visualisierung dargestellt wird.
Im hier betrachteten Beispiel ist der Visualisierungsinhalt, dass die Tortenstücke Klassen repräsentieren sollen und die Größe der Stücke durch die Anzahl der Methoden gesetzt wird.
Das Mapping ist dabei abhängig von den Metamodellen: Der Zusammenhang im hier angegebenen Beispiel ist in Abb. 2.9 dargestellt.

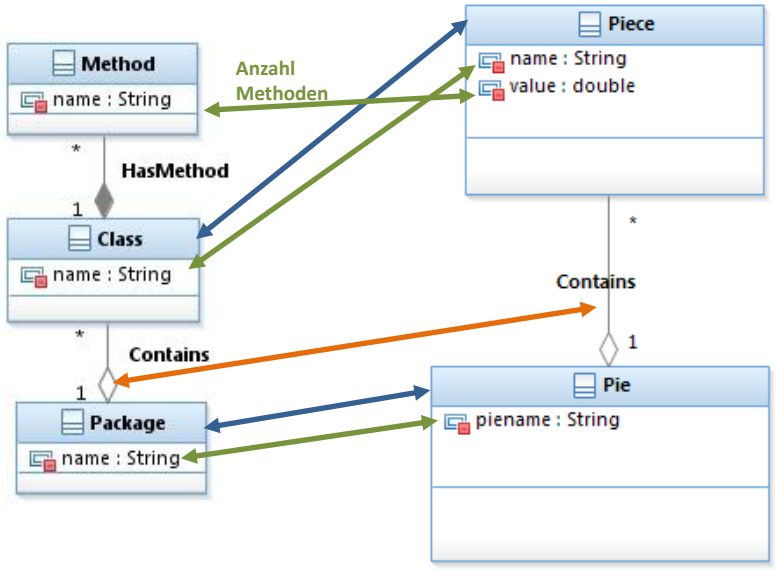


Abbildung 2.9: Mapping im Beispiel.

In 2.9 ist zu sehen, dass für **jedes** Element im Metamodell der Visualisierung ein entsprechendes Element im Metamodell der Ausgangsdaten angegeben werden muss - also für jede Klasse, Assoziation und für jedes Attribut im Visualisierungsmetamodell muss ein Pendant im Metamodell der Ausgangsdaten existieren. Die Instanzen dieser Elemente aus den Ausgangsdaten bilden dann die dazu gemappten Instanzen in den Visualisierungsdaten. Beispielsweise werden Elemente vom Typ „Class" im Ausgangsmetamodell

auf Elemente vom Typ „Piece" im Visualisierungsmetamodell gemappt.
Hierbei lassen sich generell zwei Arten von Mappings unterscheiden: Das Mapping zur Spezifizierung welches Element im Metamodell der Ausgangsdaten durch welches Element im Visualisierungsmetamodell repräsentiert werden soll („die Tortenstücke sollen die Klassen repräsentieren") und das Mapping zur Belegung der Attributwerte („die Werte der Tortenstücke werden durch die Anzahl der Methoden belegt") (siehe Abb. 2.9).

Zur Übersicht lässt sich das Mapping in einer Tabelle darstellen, wie sie für das hier gegebene Beispiel in Abb. 2.10 gezeigt ist. Dabei ist auf der linken Seite das Element im Visualisierungsmetamodell aufgelistet, in der Mitte was logisch als Visualisierungsinhalt angestrebt ist und auf der rechten Seite das Element aus dem Metamodell der Ausgangsdaten, das dazu passt. Beispielsweise sollen die Tortenstücke die Klassen repräsentieren: Daher müssen sie auf die Elemente vom Typ „Class" gemappt werden.

MM Visualisierung	Logischer Inhalt	MM Ausgangsdaten
Pie	Paket	Package
piename	Paketname	name-Attribut von Package
Piece	Klasse	Class
name	Name der Klasse	name-Attribut von Class
value	Anzahl Methoden der Klasse	Anzahl HasMethod-Kanten der Klasse
Contains	Klasse gehört zum Paket	Contains

Abbildung 2.10: Mappingtabelle für das Beispiel.

Die Umsetzung dieses Mappings ist an dieser Stelle abhängig von der gewählten Transformationssprache, die zur Transformation der Ausgangsdaten in die Visualisierungsdaten verwendet wird.
Im ELVIZ-Ansatz kann jede beliebige Transformationssprache verwendet werden, d.h. an dieser Stelle kann sowohl ATL - die ATLAS Transformation Language - oder QVT - Query View Transformation der OMG - oder eine beliebige andere Sprache verwendet werden. In ATL bestünde die Möglichkeit das Mapping über Transformationsregeln zu beschreiben.
Die Entscheidung für eine bestimmte Transformationssprache sollte dabei immer in Abhängigkeit von der Datenstruktur der Ausgangsdaten getroffen werden: Im hier betrachteten Beispiel wurden TGraphen zur Repräsentation der Quellcodefakten des fiktiven Systems verwendet. Aus diesem Grund bietet sich für dieses Beispiel eine Transformationssprache an, die speziell für TGraphen entwickelt wurde: Dies ist GReTL - die Graph Repository Transformation Language.

Diese verwendet für das Mapping *Anfragen*, die an den TGraphen der Ausgangsdaten gestellt werden. Die daraus gelieferten Ergebnisse bilden dann genau die Elemente in den Visualisierungsdaten für deren Elementtyp diese Anfrage in GReTL verwendet wurde. Als Anfragesprache auf TGraphen wird dabei GReQL - die Graph Repository Query Language - verwendet. Dieser Zusammenhang ist in Abb. 2.11 einmal für das Beispiel der Tortenstücke dargestellt:
Oben auf Abb. 2.11 befinden sich die Ausgangsdaten des Beispiels und ganz unten die daraus entwickelten Visualisierungsdaten im Beispiel. Abb. 2.11 beschreibt dabei beispielhaft das Mapping für die Tortenstücke. Diese sollen die Klassen repräsentieren. Aus diesem Grund wird in einem ersten Schritt eine Anfrage an die Ausgangsdaten gestellt, die die Klassen liefern soll (siehe Abb. 2.11).
Diese liefern dann als Ergebnis die beiden Instanzen vom Typ „Class" aus den Ausgangsdaten. Dieses Ergebnis wird intern von der Transformationssprache GReTL dazu verwendet, Instanzen vom Typ „Piece" zu bilden. Entsprechend finden sich in den Visualisierungsdaten zwei Instanzen vom Typ „Piece".

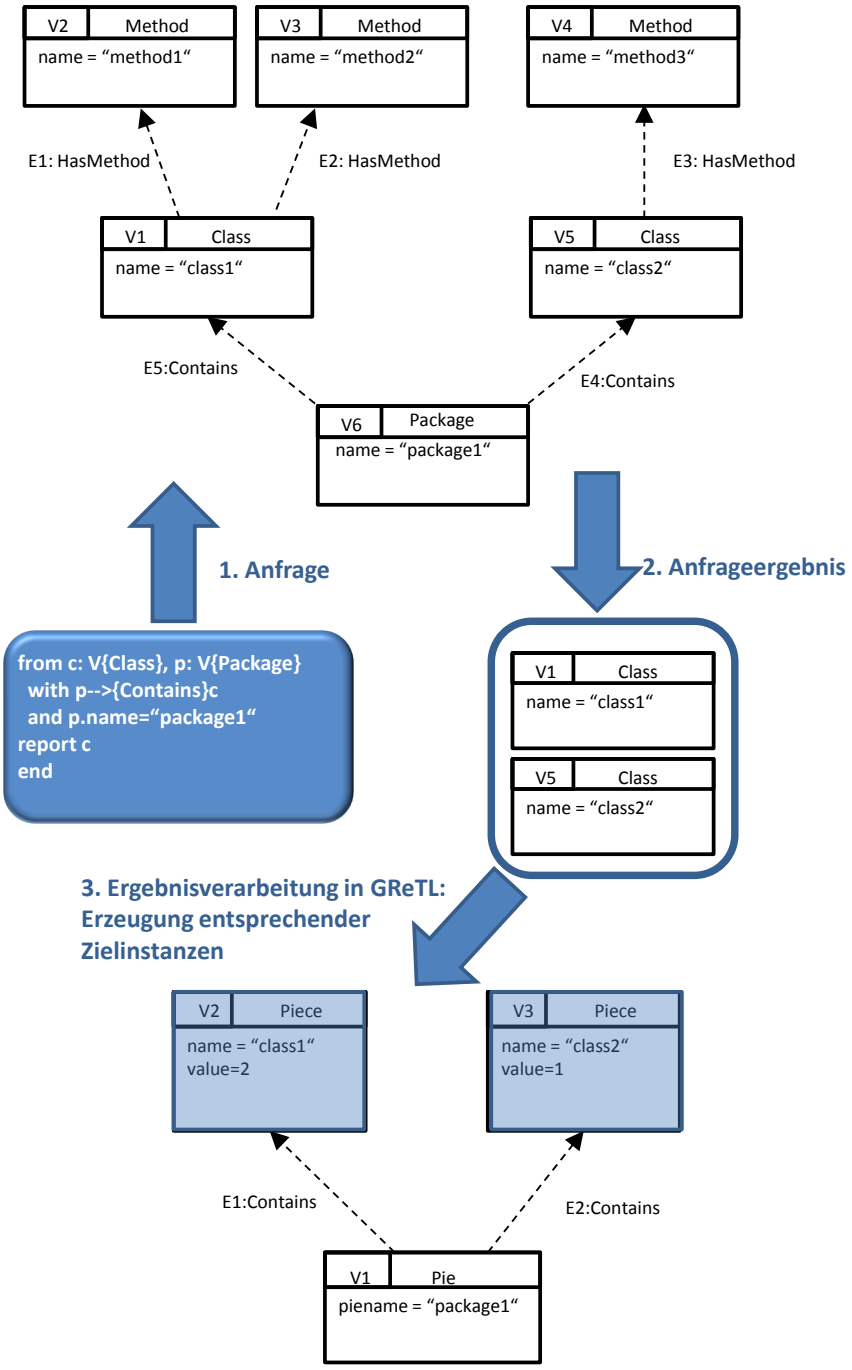


Abbildung 2.11: Beispiel für den Ablauf des Mappings und der Transformation für die Tortenstücke im fiktiven Fall des Auftragnehmers.

Da sowohl GReQL als auch GReTL im hier betrachteten Beispiel und für alle Visualisierungen, die mit dem ELVIZ-Ansatz erstellt werden und dabei auf TGraphen beruhen, verwendet werden können, werden diese Konzepte im nachfolgenden genauer erläutert. Es sei an dieser Stelle jedoch noch einmal darauf hingewiesen, dass der ELVIZ-Ansatz so generisch entwickelt wurde, dass auch jede andere Sprache - und deren spezifischen technischen Angebote zum Mapping - für die Transformation verwendet werden können.

GReQL

Für das Mapping wird im Beispiel eine Graphanfragesprache benötigt: Dies ist GReQL - die Graph Repository Query Language [4] - die seit Mitte der 90er Jahre an der Universität Koblenz-Landau entwickelt wird [20].
Bei GReQL handelt es sich um eine deklarative Graphanfragesprache, die auf TGraphen operiert und den Zugriff auf alle Eigenschaften eines Graphen, und dabei auch auf die Knoten und Kanten erlaubt. Dabei ist GReQL eine reine Anfragesprache: Dementsprechend ist es nicht möglich über GReQL Manipulationen an einem TGraphen vorzunehmen [11]. Neben dem Zugriff auf die einzelnen Elemente wie Knoten und Kanten eines TGraphen, bietet GReQL noch weitere Funktionen an: So können nicht nur die Grapheigenschaften wie Knoten- und Kantentypen oder der Grad eines Knoten angefragt werden, sondern GReQL bietet darüber hinaus noch mathematische Basis- und Statistikfunktionen wie die Summe oder den Durchschnitt an. Neben diesen mathematischen Funktionen ist es möglich in GReQL-Anfragen auch All- und Existenzquantoren zu verwenden. Zudem sind verschachtelte GReQL-Anfragen als Unteranfragen möglich, so dass mit vorherigen Teilergebnissen systematisch weitergearbeitet werden kann.
Die Ergebnisse einer GReQL-Anfrage lassen sich dann als Text-, XML- oder HTML-Datei speichern oder auch direkt in Java weiterverarbeiten, da die Ergebnisse von GReQL-Anfragen in Form des Containerdatentyps „JValue" ausgeben werden können [4].

Eine Beispielanfrage, wie sie im hier betrachteten Beispiel in Abb. 2.11 für die Tortenstücke verwendet wird, ist folgende GReQL-Anfrage:

```
from   c : V{Class}, p :V{Package}
     with   p -->{Contains}c
     and p.name ="package1"
report c
end
```

Diese Beispielanfrage wird im Fall des Auftragnehmers für das Mapping für die Tortenstücke benötigt, denn sie liefert die alle Klassen, die im Package mit dem Namen „package1" liegen.
Dabei zeigt sie bereits den groben Aufbau einer GReQL-Anfrage: Diese besteht aus drei Teilen - dem *from-Teil*, der zunächst alle relevanten Graphelemente deklariert, der *with-Klausel*, die als optionale Klausel dazu verwendet werden kann Beziehungen zwischen den

zuvor deklarierten Graphelementen zu spezifizieren und abschließend dem *report-Teil*, der die Erscheinung des Anfrageergebnisses beschreibt [11].
In der oben gezeigten Anfrage wird zunächst im from-Teil spezifiziert, dass nur die Knoten vom Typ „Class" betrachtet werden sollen. Diese werden in der weiteren Anfrage über die Variable „`c`" angesprochen („`c: V(Class)`") und die Knoten vom Typ „Package" im weiteren Verlauf über „`p`". Allgemein können so in der from-Klausel alle im Schema vorkommenden Knoten („`VKnotentyp`") und Kanten („`EKantentyp`") verwendet werden [17].
In der with-Klausel kann dann genauer spezifiziert werden, dass nur die Knoten und dementsprechend die Klassen als Ergebnis verlangt sind, die im Package mit dem Namen „package1" liegen. Hierzu muss es folglich eine Kante von p nach c vom Typ „Contains" geben und der Name von p muss „package1" sein. Dazu wird hier ein regulärer Pfadausdruck verwendet: So wird in der with-Klausel über die Notation „`p->{Contains} c and p.name ="package1"`" erreicht, dass nur die Knoten vom Typ „Class" betrachtet werden, die eine eingehende Kante vom Typ „Contains" mit einem Startknoten vom Typ „Package", dessen Namensattribut den Wert „package1" trägt, besitzen.
Abschließend sollen jetzt nur die Namen der Klassen als Anfrageergebnis zurückgegeben werden. Dies kann in der report-Klausel angegeben werden, indem über die Punktnotation auf das Attribut „name" verwiesen wird: „report c.name". Um die GReQL-Anfrage abzuschließen, folgt am Ende noch das Schlüsselwort „end".

Die *Auswertung der GReQL-Anfrage* erfolgt iterativ in zwei Schritten: Zunächst wird für jede mögliche Belegung der zuvor im from-Teil deklarierten Variablen überprüft, ob die in der with-Klausel angegebene Bedingung erfüllt wird. Wenn dies der Fall ist, wird im zweiten Schritt das Ergebnis ermittelt, wie es nach der Angabe in der report-Klausel dargestellt werden soll [17].
Im angegebenen Beispiel erfolgt die Auswertung der GReQL-Anfrage dann so, dass zunächst alle Knoten vom Typ „class" und alle Knoten vom Typ „Package" daraufhin überprüft werden, ob sie die in der with-Klausel angegebene Bedingung erfüllen und folglich über eine Kante „Contains" in entsprechender Beziehung zueinander stehen. Ist dies der Fall, so wird im zweiten Schritt der Wert des Attributs „name" des entsprechenden Klassenknotens ausgegeben. Das Ergebnis ist in diesem Fall eine einspaltige GReQL-Tabelle aus Strings, die die Namen aller Klassen enthält, die im Package „package1" liegen. Die Ergebnisstruktur der Anfrage ist dabei ein sogenannter *Bag*, d.h. das Ergebnis liegt als Multimenge vor, bei der bei bestimmten Anfragen auch Elemente doppelt vorkommen können. Ist für das Ergebnis eine Menge gewünscht, die keine doppelten Elemente enthält und die sich beispielsweise besser für die Anwendung einer Vereingungsfunktion eignet, so kann dies in der report-Klausel auch konkret über „report set" angegeben werden [17].

Diese Anfragesprache wird für das Mapping *in der Transformationssprache GReTL verwendet* um die passenden Instanzen zu den Ergebnissen aus der Anfrage zu erstellen (siehe Abb. 2.11). Wie dies in GReTL genau realisiert wird und was GReTL ist, wird im nachfolgenden Abschnitt beschrieben.

GReTL

Bei GReTL handelt es sich um eine operationale Modelltransformationssprache, die speziell für TGraphen konzipiert wurde. GReTL arbeitet dabei nach dem Prinzip, dass neben dem Zielmetamodell gleichzeitig auch der Zielgraph entwickelt wird [9]. Die eigentlichen GReTL-Operationen können dabei sowohl über eine speziell entwickelte GReTL API direkt in Java verwendet werden oder über eine einfache domänenspezifische Sprache [16]. GReTL verfügt nur über eine kleine Menge an Operationen, wurde jedoch so konzipiert, dass sie leicht erweiterbar ist [9].

Mit GReTL ist es möglich über einen kleinen Satz an Operationsbefehlen - bei denen es sich hauptsächlich um *create*-Operationen handelt - zu einem gegebenen Quellmodell das Zielmetamodell zu entwickeln und gleichzeitig die gegebenen Quellinstanzen in Instanzen des Zielmetamodells zu überführen [9]. Dieser Aspekt der Transformationssprache ist insbesondere im Zusammenhang für die Umsetzung des Softwarevisualisierungskonzeptes im Kontext der Bachelorarbeit interessant, da GReTL auch in der Lage ist, nur die Instanzen zu erzeugen, sofern bereits ein Zielmetamodell - wie beispielsweise ein Visualisierungsmetamodell - vorliegt [9].

GReTL – DSL

Um GReTL-Transformationen durchzuführen bietet sich die einfach verwendbare domänenspezifische Sprache an: Ein Beispiel für eine GReTL-Transformationsoperation in der GReTL-DSL ist die folgende Operation:

```
1 CreateVertexClass Pie <== from c : V{Class}, p :V{Package}
2                                    with p-->{Contains}c
3                                   and p.name ="package1"
4                           report c
5                           end
```

Diese Beispieloperation verwendet zum Mapping dabei die Beispielabfrage des vorherigen Abschnitts und erstellt den Knoten („VertexClass") mit dem Namen „Pie". Da bei GReTL gleichzeitig auch der Zielgraph entwickelt wird, werden „Instanzen", sogenannte „Images" von „Pie" im Zielgraphen erstellt, die mittels der GReQL-Anfrage ermittelt werden. Dies sind entsprechend zwei Tortenstücke (siehe Abb. 2.11).

An dieser Beispieloperation ist bereits der grundsätzliche Aufbau jeder Transformationsoperation erkennbar. Diese sind dabei von der folgenden Form [16]:

```
<GReTL operation> <schema properties> [<== <semantic expression>];
```

Die *GReTL-Operation* gibt dabei zunächst an um welche konkrete Transformationsoperation es sich dabei handelt. Danach werden die *„schema properties"* spezifiziert: So wird an dieser Stelle der Name einer zu erstellenden Klasse oder eines Attributs angegeben. Handelt es sich um eine Beziehung, werden an dieser Stelle außerdem noch die Multiplizitäten und Rollenbezeichner deklariert. Am Ende kann optional ein *semantischer Ausdruck* folgen. Dieser wird dabei über eine GReQL-Anfrage spezifiziert, die auf dem Quell-TGraphen ausgewertet werden kann. Dadurch werden sogenannte *„Archetypen"* für die Kanten und Knoten definiert und für jeden Archetyp wird im Zielmetamodell ein neues Element angelegt. Im Zielmodell wird für jedes Element des semantischen Ausdrucks dann ein sogenanntes *„Image"* angelegt. Ein Image kann dabei immer genau einem Archetypen zugeordnet werden [16].

Die genauen TGraph-Operationen sind dabei „*CreateVertexClass*", „*CreateAbstractVertexClass*", „*CreateAbstractEdgeClass*", „*CreateEdgeClass*", „*AddSubClass(es)*" sowie „*CreateAttribute(s)*".
Mit „*CreateVertexClass*" wird eine Knotenklasse im Zielmetamodell und gleichzeitig entsprechende Instanzen dieser Klasse im Zielgraphen erzeugt. Soll es sich dabei um eine abstrakte Klasse handeln, so wird entsprechend die Operation „*CreateAbstractVertexClass*" verwendet. Das Gleiche gilt für das Erstellen von Kantenklassen: Hier erzeugt „*CreateAbstractEdgeClass*" eine abstrakte Kantenklasse und „*CreateEdgeClass*" entsprechende Kantenklassen des Zielmetamodells sowie die dazugehörigen Instanzen, die aus dem Quellmodell folgen. Über die Operation „*AddSubClass*" können Generalisierungsbeziehungen zwischen einzelnen Klassen herausgestellt werden. Beispielsweise kann über „AddSubClass Statement IfStatement" ausgedrückt werden, dass eine zuvor angelegte Klasse IfStatement, von einer zuvor angelegten Klasse Statement erbt und folglich eine Generalisierungsbeziehung vorliegt. Handelt es sich um mehrere Unterklassen, so wird hier entsprechend die Mehrzahl verwendet, z.B. „AddSubClasses Statement IfStatement ElseStatement".
Eine weitere create-Operation ist „*CreateAttribute*". Diese erzeugt in einer über Punktnotation angegebenen Klasse das entsprechende Attribut. Um die Werte dieser Attribute zu belegen kann dies wiederum über eine GReQL-Anfrage im semantischen Ausdruck angegeben werden [9].

Insgesamt können so in der GReTL DSL über wenige create-Operationen ein Zielmetamodell - wie das Visualisierungsmetamodell - und entsprechende Instanzen im Zielmodell - wie den Visualisierungsdaten - erzeugt werden.

GReTL – API

Um GReTL direkt aus Java zu verwenden, existiert die GReTL API. Nachfolgend ist der erste Teil der GReTL Transformation zum Beispiel des Auftragnehmers zur Generierung seiner Visualisierungsdaten angegeben:

```
1  public class MyTransformation extends Transformation <Graph> {
2
3
4      //Konstruktor
5      public MyTransformation(Context c){
6            super(c);
7      }
8
9      //Transformationsbeschreibung
10     @Override protected Graph transform(){
11           VertexClass piece = new CreateVertexClass( context, "Piece"
                 ,
12                 "from  c : V(Class), p :V(Package)
13                 with  p -->{Contains}c and p.name =\"package1\"
14                 report c
15                 end").execute();
16                //...
17                // hier muessen weitere Operationen inklusive GReQL
                     -Anfragen angegeben
18                // werden fuer Pie, die Assoziation Contains und
                     alle Attribute
19                //...
20                return context.getTargetGraph();
21           }
22
23           //Mainmethode zum Ausfuehren der Transformation im Kontext:
24           public static void main(String[] a){
25                 Context c = new Context ("Zielschema" , "Zielgraph");
26                 c.addSourceGraph ( "Quellgraph" , Quellschema.instance
                      ().loadGraph(a[0]));
27                 GraphIO.saveGraphToFile(a[1], new MyTransformation(c).
                      execute());
28     }
29 }
```

In Zeile 11 bis 15 ist beispielhaft die Transformationsoperation dargestellt, die auch als Beispiel für die GReTL-DSL verwendet wurde:
Hier wird eine Klasse im Zielmetamodell erzeugt, die den Namen „Piece" hat. Dies sind somit die Tortenstücke. Gleichzeitig wird der semantische Ausdruck in Form der GReQL-Anfrage übergeben und somit können die entsprechenden Instanzen der Tortenstücke im

Zielmodell - den Visualisierungsdaten - erzeugt werden, so dass ein Tortenstück für jeden Knoten des Typs „Class" der mit dem Knoten mit Namen „package1" aus den Ausgangsdaten verbunden ist, erzeugt wird (siehe hierzu auch Abb. 2.11).

Insgesamt wird in diesem Beispiel eine eigene Transformation implementiert, die von der Klasse *Transformation* der API erbt. In der Methode *transform()* werden dann die auszuführenden Operationen angegeben und am Ende soll in dieser Methode über „return context.getTargetGraph()" der Zielgraph zurückgegeben werden. Dieser sieht im hier betrachteten Beispiel entsprechend wie im unteren Teil aus Abb. 2.11 aus.
Damit der gesamte Transformationsprozess ausgeführt werden kann, wird noch die main-Methode benötigt: In dieser wird der Kontext angelegt, indem der Konstruktor der Klasse „Context" mit den entsprechenden Parametern für Zielschema und Zielgraph aufgerufen wird. In diesen Kontext wird dann der Quellgraph über die Methode „addSourceGraph" eingelesen. Am Ende wird schließlich die selbst geschriebene Transformation instanziiert und ausgeführt und mit „saveGraphToFile" wird abschließend der Zielgraph in einer Datei gespeichert.

Das allgemeine GReTL Transformationsframework ist in Abb. 2.12 dargestellt.

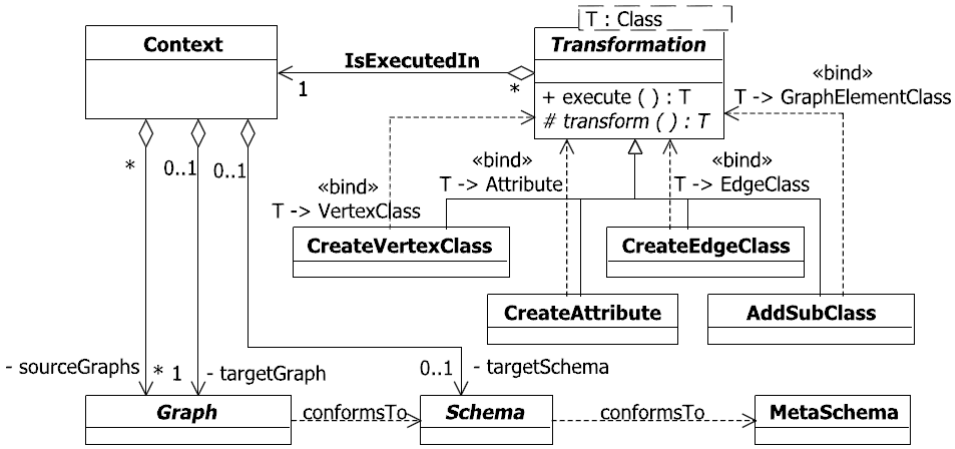


Abbildung 2.12: GReTL Transformationsframework [9].

Abb. 2.12 zeigt dabei, dass der Hauptbestandteil des GReTL Transformationsframeworks von der Klasse „Transformation" gebildet wird, die über zwei Methoden verfügt: „execute" und „transform".
Die Methode `execute()` kann aufgerufen werden um eine Transformation auszuführen. Dagegen ist die Methode `transform()` abstrakt. Diese Methode muss daher von jeder

konkreten Unterklasse überschrieben werden. Somit wird in den Unterklassen, die von den GReTL Transformationen „CreateVertexClass", „CreateEdgeClass", „CreateAttribute" und „AddSubClass" gebildet werden, das konkrete Transformationsverhalten implementiert.
Die eigentliche Ausführung der Transformation findet dann in einem Kontext statt, der in Abb. 2.12 durch die Klasse „Context" repräsentiert wird. Diese Klasse dient dazu, das Quellmetamodell einzulesen und den Transformationsprozess zu managen [9].

Insgesamt ist es so neben der GReTL DSL auch möglich über die GReTL API Transformationen direkt in Java auszuführen. In der Bachelorarbeit erfolgt eine spätere technische Realisierung zur Validierung des allgemeinen modellgetriebenen Softwarevisualisierungsansatzes in Java. Aus diesem Grund eignet sich die Verwendung der GReTL API an dieser Stelle eher, da sie direkt aus Java heraus verwendet werden kann.

Insgesamt ist es also im Visualisierungsansatz für das hier betrachtete Beispiel möglich aus dem TGraphen der Ausgangsdaten über eine GReTL-Transformation, die Informationen über die Visualisierungsart in Form des Visualisierungsmetamodells und über den Visualisierungsinhalt in Form von Mappings über GReQL-Anfragen vereint, die benötigten Visualisierungsdaten zu transformieren.
An dieser Stelle lässt sich jetzt auch exakter definieren, was die Visualisierungsdaten ganz genau sind: Es handelt sich hierbei um die Daten, die grafisch dargestellt werden sollen. Sie enthalten dabei sowohl Informationen über die Visualisierungsart als auch Informationen über den Visualisierungsinhalt. In dem hier betrachteten Beispiel, sollen beispielsweise die Tortenstücke die Klassen repräsentieren. Folglich muss es nun in den Visualisierungsdaten eine Instanz eines Objekts „Tortenstück" geben, das mit den entsprechenden Werten der zu repräsentierenden Klasse belegt ist.

Unabhängig vom Beispiel heißt das für den *allgemeinen* Visualisierungsansatz, dass er stets die Ausgangsdaten in die Visualisierungsdaten transformiert, indem er die Informationen über die Visualisierungsart als Metamodell erhält und die notwendigen Informationen über den Visualisierungsinhalt durch das Mapping. Somit sind die Spezifizierungen von Visualisierungsinhalt und Visualisierungsart wiederverwendbar. An dieser Stelle müsste die Transformation jedoch stets neu geschrieben werden. Um auch die Transformation automatisch generiert zu erhalten, wird der hier entwickelte Visualisierungsansatz so erweitert, dass die Transformation selber ebenfalls wieder als Modell angesehen wird, d.h. an dieser Stelle wird erneut der MDA-Ansatz zuhilfe genommen: So ist es möglich die Transformation aus dem Mapping und dem Visualisierungsmetamodell automatisch zu generieren.
Durch die Ausführung der Transformation werden die Visualisierungsdaten generiert. Diese müssen abschließend lediglich noch gerendert werden, damit eine reale Grafik erhalten werden kann. Dies ist in Abb. 2.2 durch den 3. Schritt - rendern - dargestellt und wird im nachfolgenden Abschnitt beschrieben.

2.4.3 Das Rendering

Abschließend können die durch die Transformation gewonnenen Visualisierungsdaten in einem Renderer genutzt werden. Dieser soll die Visualisierungsdaten in die reale Grafik überführen. Hierzu muss ein passender Renderer für das entsprechende Metamodell der Visualisierung implementiert werden oder - falls ein solcher Renderer bereits existiert - ausgewählt werden: An dieser Stelle ist die Implementierung somit auch nur einmal pro Visualisierungsart notwendig. Danach kann der Renderer stets wiederverwendet werden. Der Renderer kann dabei in einer beliebigen Sprache implementiert werden. Für die Validierung des ELVIZ-Ansatzes wurde ein Renderer für Tortendiagramme geschrieben. Der dazugehörige Quellcode befindet sich in Abschnitt 3.1.2. Dieser könnte an dieser Stelle dazu verwendet werden, um die zu den Visualisierungsdaten passende reale Grafik zu erzeugen wie sie in Abb. 2.13 dargestellt ist.

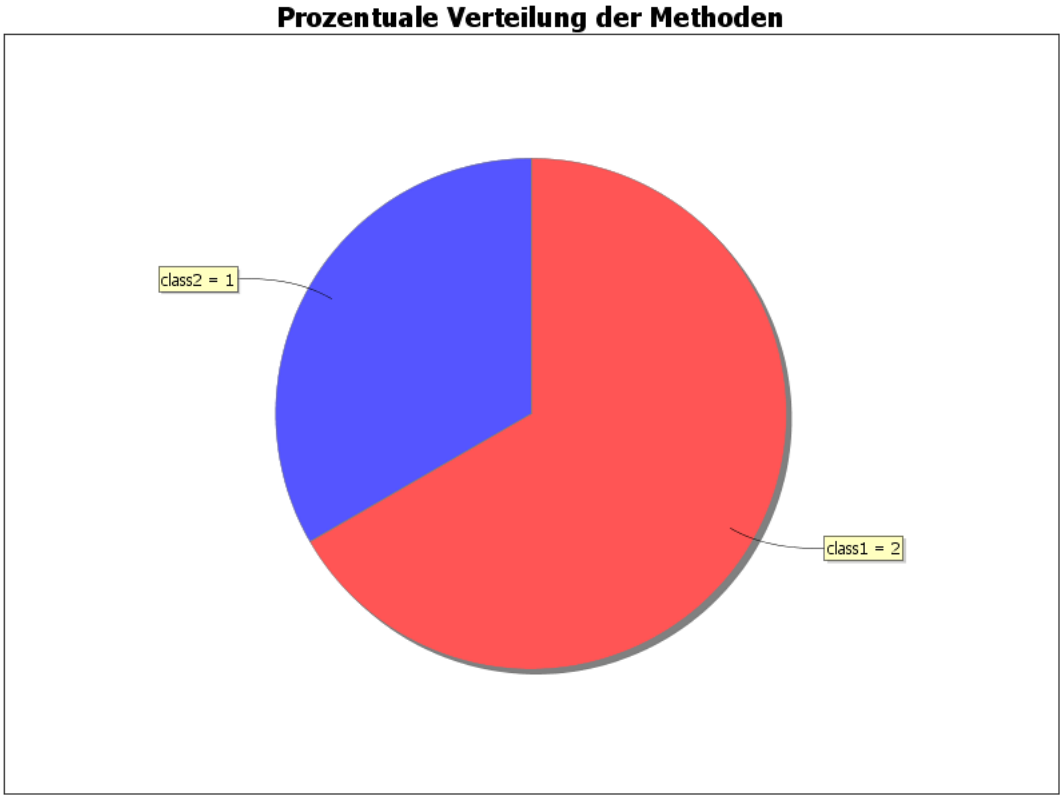


Abbildung 2.13: Tortendiagramm zur prozentualen Verteilung der Methoden auf die Klassen des fiktiven Systems.

Insgesamt wird es so mit dem hier erarbeiteten Visualisierungsansatz möglich aus beliebigen Ausgangsdaten eine komplett personalisierte Visualisierung zu erhalten, die auch für andere Ausgangsdaten wiederverwendbar ist. Dabei bleiben nicht nur Visualisierungsart, Visualisierungsinhalt und Ausgangsdaten komplett variabel, sondern auch die interne

technische Umsetzung bleibt frei wählbar. Dabei sind alle denkbaren Datenstrukturen für die Ausgangsdaten und die Visualisierungsdaten wählbar, die Wahl der Transformationssprache ist absolut frei wählbar und der Renderer kann mit einer beliebigen Programmiersprache und beliebigen APIs als Hilfsmittel implementiert werden. Neben dieser kompletten Auswahlfreiheit schafft es der Ansatz dennoch eine schnelle Generierung von Visualisierungen zu bieten, die darüber hinaus noch wiederverwendbar sind. Dies kann der in der Bachelorarbeit entwickelte ELVIZ-Ansatz bieten, da er durch die doppelte Verwendung der MDA-Arbeitsweise (siehe Abb. 2.2) sehr schnell durch eine Veränderung der Modelle eine neue Transformation sowie darauf aufbauend die notwendigen Visualisierungsdaten automatisch generieren kann. In diesem Fall kann die „Architecture of Choice for a Changing World" somit sehr gewinnbringend eingesetzt werden.

3 | Implementierung des ELVIZ-Ansatzes

Um die Validierung durchführen zu können, mussten Hilfsmittel für den beschriebenen allgemeinen Visualisierungsansatz implementiert werden, die die beschriebene Arbeitsweise des doppelt verwendeten MDA-Ansatzes umsetzen (siehe Abb. 2.2). Hierzu wurde die Java Visualization API - kurz JVizAPI - entwickelt. Diese setzt den allgemeinen ELVIZ-Ansatz mithilfe der in Teil 2.4 beschriebenen graphentechnischen Grundlagen um. Da diese API auch in Zukunft real einsetzbar ist, wird ihr Aufbau und ihre grundsätzliche Verwendung nachfolgend erläutert. Im letzten Teil dieses Kapitels wird die JVizAPI dann zur Validierung des Visualisierungsansatzes eingesetzt.

3.1 Die Java Visualization API

Die JVizAPI, die zur Validierung des Visualisierungsansatzes entwickelt wurde, wird im Folgenden beschrieben: Dazu wird zunächst der allgemeine Aufbau der API genannt und anschließend wird das vorherige Beispiel des Auftragnehmers, der eine visuelle Darstellung der prozentualen Verteilung der Methoden auf die Klassen seines Systems haben möchte, genutzt, um die Verwendung der JVizAPI deutlich zu machen.

3.1.1 Allgemeiner Aufbau der JVizAPI

Die in der Bachelorarbeit entwickelte JVizAPI vereinfacht die Erzeugung einer neuen Visualisierung - bestehend aus Visualisierungsart und Visualisierungsinhalt - und einer neuen realen Grafik. Dabei übernimmt die JVizAPI folgende Aufgaben: Die Generierung der Transformation zu einer Visualisierung und die Generierung einer Methode zur Erzeugung einer konkreten Grafik über eine Schnittstelle für einen passenden Renderer sowie die Speicherung der erzeugten Grafik in einer SVG-Datei.
Der allgemeine Aufbau der JVizAPI ist in dem Klassendiagramm in Abb. 3.1 dargestellt.

Hieran ist erkennbar, dass sich eine Hilfs-API für den hier entwickelten allgemeinen Visualisierungsansatz bereits mit nur vier Klassen umsetzen lässt: In diesem Fall die Klasse „VizTransformation", die Klasse „ClassGenerator", die Klasse „SVGGenerator" und die abstrakte Klasse „Renderer".
Die Klasse „ClassGenerator" ist dabei nur eine Hilfsklasse. Sie wird dazu verwendet den Quellcode dynamisch zu erzeugen und Java-Klassen zu generieren. Dieser „ClassGenera-

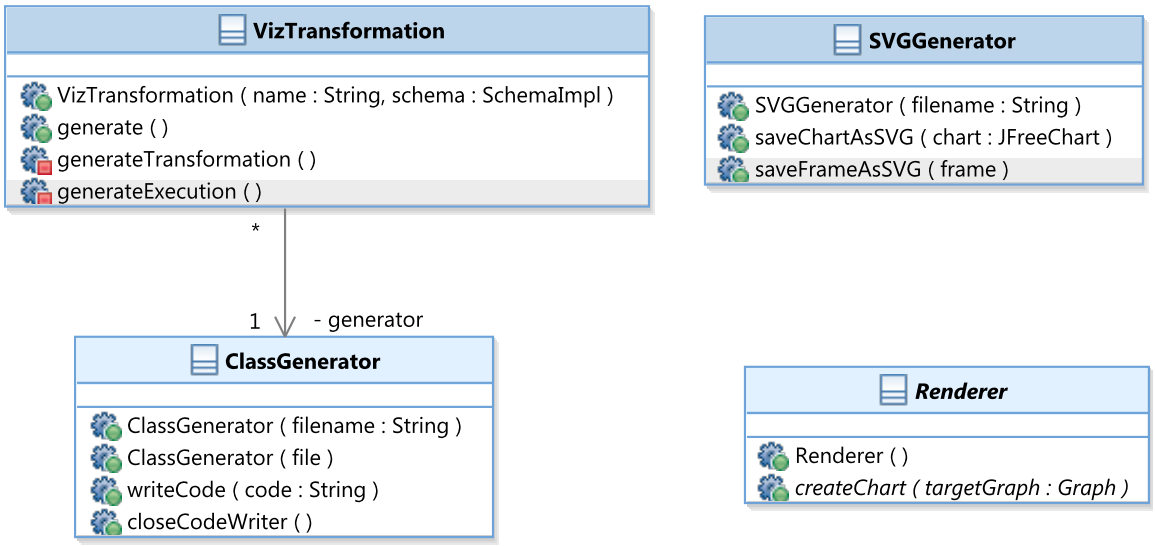


Abbildung 3.1: Das Klassendiagramm der Java Visualization API zur Übersicht.

tor" wird dabei intern von der Klasse „VizTransformation" verwendet.
Die Klasse „VizTransformation" ist die wichtigste Klasse der JVizAPI: Sie dient zur automatischen Generierung der Transformation und der Transformationsausführung. Dafür besitzt sie den Konstruktor `VizTransformation(String name, SchemaImpl schema)`. Über diesen wird eine Instanz der VizTransformation erzeugt und über Aufruf der Methode „generate" werden intern die Methoden „generateTransformation" und „generateExecution" aufgerufen. Daraufhin werden automatisch Klassen zur Transformation von Ausgangs-TGraphen in Visualisierungsdaten generiert, die zu dem Schema konform sind, das als Parameter dem Konstruktor der „VizTransformation" übergeben wurde.
Die abstrakte Klasse „Renderer" dient als Schnittstelle für einen konkreten Renderer: Hierzu muss der zum vorher definierten Schema passende Renderer vom Benutzer implementiert werden, dabei von dieser Klasse erben und dementsprechend die Methode „createChart()" schreiben. Die abstrakte Klasse Renderer erbt intern von der Klasse „JFrame". Es stehen an dieser Stelle somit die Methoden von JFrame direkt zur Verfügung. Damit nachfolgend die SVG-Datei korrekt erzeugt werden kann, sollte die Methode „createChart" so erzeugt werden, dass sie die zu speichernde Grafik in einem JFrame anzeigt. Beispielsweise kann an dieser Stelle ein JPanel erzeugt werden und abschließend kann in der Methode „createChart" dieses JPanel über „setContentPane(panel)" dem JFrame hinzugefügt werden. Konkrete Beispiele hierzu finden sich auch noch in den nachfolgenden Validierungen.
Abschließend existiert noch die Klasse „SVGGenerator": Diese erzeugt ein SVG aus dem JFrame, das der Renderer geliefert hat. Dabei wird die Methode „saveFrameAsSVG" in der generierten Klasse zur Ausführung der generierten Transformation aufgerufen. Ein Anwender der API muss sich folglich nicht um die Speicherung seiner Grafik im SVG-Format kümmern, da dies bereits intern von der API realisiert wird.

Die JVizAPI ist dabei allerdings nur eine Möglichkeit den allgemeinen Softwarevisualisierungsansatz, wie er in der Bachelorarbeit entwickelt wurde, umzusetzen. Dieser kann natürlich auch mit beliebigen anderen Techniken realisiert werden.

3.1.2 Verwendung der JVizAPI zur Erstellung einer neuen Visualisierung und Grafik

Da die JVizAPI auch in Zukunft real einsetzbar ist, wird ihre Verwendung im nachfolgenden anhand eines Beispiels erläutert: Dabei wird erneut das Beispiel vom Beginn der Arbeit verwendet - ein Auftragnehmer möchte eine Grafik erstellen, die eine prozentuale Verteilung der Methoden auf die Klassen seines Systems zeigt. Die dabei verwendeten Arbeitsschritte entsprechen den Arbeitsschritten aus Teil 4 der Bachelorarbeit unter Verwendung des beschriebenen Rollenkonzepts.

1. Analyse des vorhandenen Systems

Zunächst müssen die Ausgangsdaten des fiktiven Systems in einer geeigneten Datenstruktur vorliegen. Wie in Kapitel 2.4 der Arbeit beschrieben, wird dafür ein TGraph verwendet. Dieser ist in Abb. 2.5 dargestellt und kann durch den Analysierer durch ein Parsen des Systems über geeignete Analysewerkzeuge erhalten werden.

2. Generierung einer neuen Visualisierungsart durch den Modellierer

Zunächst kann eine neue Visualisierungsart durch den Modellierer anhand des Visualisierungsmetamodells spezifiziert werden. In dem hier betrachteten Beispiel wurde als sinnvolle Visualisierungsart das Tortendiagramm identifiziert. Aus diesem Grund muss an dieser Stelle das Metamodell des Tortendiagramms spezifiziert werden, wie es in Abb. 2.8 zu sehen ist. Damit das Visualisierungsmetamodell in der JVizAPI zur Generierung der Transformation verwendet werden kann, muss es über das sogenannte „Java Graphenlabor" als grUML-Schema eingegeben werden.
Beim Java Graphenlabor handelt sich um eine effiziente API zur Verarbeitung von TGraphen [20] in Java. Mit dieser ist es möglich TGraphen zu manipulieren und anzufragen: So lassen sich Kanten, Knoten und Attribute erstellen und deren Werte anfragen. Darüber hinaus bieten diese Bibliotheken noch weitere Unterstützung bei der Arbeit mit TGraphen an, wie beispielsweise die Abfrage von Knoten- und Kantentypen [11], Traversierungsfunktionen für alle Knoten und Kanten oder die Modifikation von TGraphen zur Laufzeit [20]. Über die JGraLab Bibliotheken ist es möglich mit TGraphen, die über Millionen von

Knoten und Kanten verfügen, zu arbeiten [11]. Dies ist im Kontext der Bachelorarbeit sehr nützlich, da die Ergebnis-TGraphen eines analysierten Quellcodes sehr groß werden können. Darüber hinaus lässt sich an dieser Stelle das Schema über JGraLab direkt textuell in Java eingeben: Im hier angegebenen Beispiel sieht diese Eingabe wie folgt aus:

Zunächst muss ein neues Schema und eine neue GraphClass erzeugt werden. Dies kann über die folgenden Befehle in JGraLab erreicht werden:

```
1 Schema schema = new SchemaImpl("PieDiagramSchema", "pieDiagramSchema"
    );
2 GraphClass gc = schema.createGraphClass("PieDiagramGraphClass");
```

Danach können die entsprechenden Knoten und Kanten, die den Assoziationen und Klassen des gewünschten Metamodells für die Visualisierung entsprechen, angelegt werden und der GraphClass hinzugefügt werden:

```
1  //Nodes:
2  VertexClass pie = gc.createVertexClass("Pie");
3  pie.addAttribute("piename", schema.getDomain("String"));
4
5  VertexClass piece = gc.createVertexClass("Piece");
6  piece.addAttribute("name", schema.getDomain("String"));
7  piece.addAttribute("value", schema.getDomain("Integer"));
8
9  //Edges:
10 EdgeClass edge1 = gc.createEdgeClass("Contains", pie, 1, 1,"pie",
11 AggregationKind.NONE, piece, 0, 100, "piece", AggregationKind.NONE);
```

Abschließend muss das Schema lediglich über den Befehl „commit" gespeichert werden:

```
1 schema.commit("src/", CodeGeneratorConfiguration.NORMAL);
```

Die JVizAPI bietet nun die Klasse „VizTransformation" an, um für diese Visualisierung automatisch eine Transformation und eine Methode zur Ausführung der Transformation zu generieren. Diese Klasse verfügt über den Konstruktor „`VizTransformation(String name, SchemaImpl schema)`" und die Methode „`generate()`". Um eine neue Visualisierung zu erstellen, muss an dieser Stelle lediglich ein passender Name gewählt werden - beispielsweise „Piediagram" - und das dazu passende Schema über das Java-Graphenlabor wie zuvor beschrieben angelegt werden. Diese beiden Objekte können dann zur Erzeugung einer Instanz der „VizTransformation" verwendet werden. Über diese Instanz kann dann der Aufruf der „generateTransformation"-Methode aufgerufen werden. Dies sieht in Java wie folgt aus:

```
1 VizTransformation v = new VizTransformation("PieDiagram",
    PieDiagramSchema.instance());
2 v.generate();
```

Nach Aufruf dieser Methode werden automatisch zwei neue Klassen generiert: Eine Klasse, die die GReTL-Transformation beinhaltet und als Name den gewählten „name" der VizTransformation besitzt (in diesem Fall „PieDiagram") und eine Klasse zur Ausführung dieser Transformation („PieDiagramExecution"). Letztere kann im weiteren Verlauf zur Erzeugung einer konkreten Grafik immer wiederverwendet werden.

Alternativ hätte der Modellierer zusätzlich entscheiden können, dass *visuelle Attribute* festgelegt werden können. Dann bestünde beispielsweise die Möglichkeit die Farben einzelner Tortenstücke zu setzen. In diesem Fall hätte er die Visualisierung auch entsprechend des Metamodells in Abb. 3.2 spezifizieren können. Hier wäre zusätzlich die Farbe einzelner Tortenstücke über das Attribut „color" wählbar und es ist entscheidbar, ob die Torte dreidimensional dargestellt werden soll oder nicht. Dies kann über den Boolean „isThreeDimensional" in der Klasse „Pie" gesetzt werden.

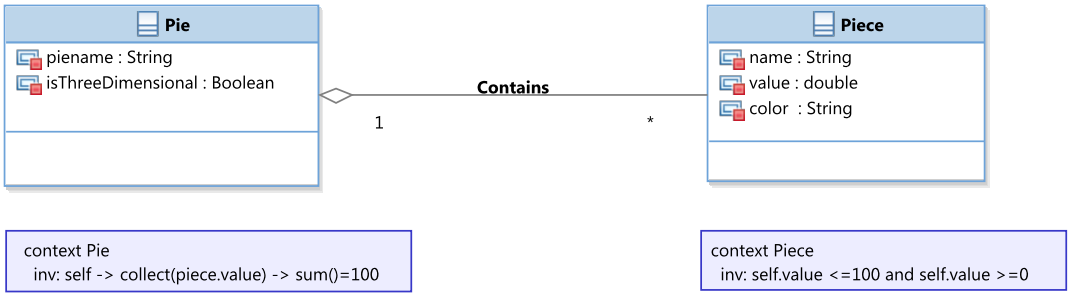


Abbildung 3.2: Metamodell eines Tortendiagramms mit visuellen Attributen.

Durch die Verwendung weiterer Attribute handelt es sich um ein neues Schema. Insgesamt muss somit eine neue Transformation generiert werden. Hierzu muss zunächst wieder das Schema in JGraLab angelegt werden, wie es bereits zuvor gemacht wurde. Zusätzlich werden außerdem die beiden neuen Attribute hinzugefügt:

```
1 piece.addAttribute("color", schema.getDomain("String"));
2 pie.addAttribute("IsThreeDimensional", schema.getDomain("Boolean"));
```

Danach können die Klassen zur Transformation und deren Ausführung wie zuvor angegeben über die JVizAPI automatisch generiert werden.

3. Die Implementierung des Renderers

Nachdem die Visualisierungsart spezifiziert wurde und die entsprechende Transformation sowie die execute-Methode zur Ausführung dieser Transformation über die JVizAPI automatisch generiert wurden, wird zum Aufruf der execute-Methode der Renderer, der die Elemente aus dem Visualisierungsmetamodell in die korrekten grafischen Elemente

überführt, benötigt. Die JVizAPI bietet für das Rendering intern die abstrakte Klasse „Renderer" an und die Klasse „SVGGenerator". Der „SVGGenerator" wird intern zum Speichern der erzeugten Grafik im SVG-Format verwendet, so dass der Implementierer des Renderers sich um die Speicherung nicht zu kümmern braucht. Er muss lediglich eine Klasse schreiben, die von der abstrakten Klasse „Renderer" der JVizAPI erbt. Die abstrakte Klasse Renderer erbt von der Klasse JFrame, so dass in dem selbst geschriebenen Renderer alle Methoden dieser Klasse direkt zugreifbar sind. Bei der Implementierung des passenden Renderers muss die Methode „`public abstract void createChart(Graph targetGraph)`" implementiert werden. Die zu erstellende Grafik muss dann nur dem JFrame über die Methode „setContentPane" hinzugefügt werden. Im hier verwendeten Beispiel müsste dementsprechend ein Renderer geschrieben werden, der aus den Visualisierungsdaten ein reales Tortendiagramm erzeugt. Dieser kann wie folgt implementiert werden:

```
1  public class PieDiagramRenderer extends Renderer{
2
3    public PieDiagramRenderer(){}
4
5    public void createChart(Graph targetGraph) {
6      GreqlEvaluatorFacade evaluator = new GreqlEvaluatorFacade(
           targetGraph);
7
8      //read Pies
9      JValueList pies = evaluator.evaluate("from c: V{Pie} report c end
         ").toJValueList();
10     JPanel panel = new JPanel(new GridLayout(cakes.size()/4+1, pies.
         size()/2 ));
11
12     //Create Pies:
13     for (int i=0; i<pies.size(); i++){
14       //Get Pieces of current Pie
15       JValueList pieces = evaluator.evaluate("from p: V{Piece}, c: V{
           Pie} with c-->{Contains}p and c.piename="+"\""+cakes.get(i).
           toVertex().getAttribute("piename")+"\""+" report p end").
           toJValueList();
16       //set Values
17       DefaultPieDataset dataset = new DefaultPieDataset();
18       for (int a = 0; a < pieces.size(); a++) {
19         dataset.setValue(pieces.get(a).toVertex().getAttribute("name"
             ).toString(),
20         (Number)pieces.get(a).toVertex().getAttribute("value"));
21       }
22       JFreeChart chart = ChartFactory.createPieChart(pies.get(i).
           toVertex().getAttribute("piename").toString(), dataset,
           false, false, false);
23       PiePlot plot = (PiePlot) chart.getPlot();
24       plot.setLabelGenerator(new StandardPieSectionLabelGenerator("
           {0} = {2}", NumberFormat.getNumberInstance(), NumberFormat.
```

```
            getPercentInstance()));
25          panel.add(new ChartPanel(chart));
26          }
27      setContentPane(panel);
28      }
29  }
```

In diesem Renderer wird die freie Bibliothek JFreeChart verwendet. Mit dieser ist es recht einfach die Visualisierungsdaten in die realen Grafikelemente zu überführen.
In Zeile 9 werden zunächst die „Pie(s)" eingelesen. Danach werden in einer Schleife alle Stücke dieses Kuchens ausgelesen und daraus ein Datensatz mit den dazugehörigen Namen und Werten der Stücke erstellt. Mit diesem lässt sich dann über den Befehl `createPieChart` ein Tortendiagramm mit diesen Werten und beliebigen Farben der Stücke erstellen (Zeile 22).

4. Ausführung der Transformation zur Erzeugung einer konkreten Grafik:

Nachdem die notwendigen Klassen über die JVizAPI wie zuvor beschrieben, automatisch generiert wurden, kann die execute-Methode in der generierten Klasse zur Ausführung der Transformation aufgerufen werden. Diese Klasse hat den Namen der Transformation und angehängt daran das Wort „Execution". Im hier betrachteten Beispiel wurde folglich von der JVizAPI die Klasse „PieDiagramExecution" erzeugt. Diese verfügt über die Methode „`public void execute(String filename, String sourceGraph, String targetGraph, HashMap<String, String> greqlQueries, Renderer renderer) throws GraphIOException`". Diese benötigt zum Aufruf die folgenden Parameter: Über den String „filename" wird der gewünschte Name der SVG-Bilddatei eingegeben, über den String „sourceGraph" muss dann der Pfad angegeben werden, an dem sich der Ausgangs-TGraph für die Transformation befindet. Der Pfad, der für den String „targetGraph" eingetragen wird, dient zur Festlegung des Speicherortes für den in der Transformation erzeugten Zielgraphen - dementsprechend als Speicherort für die Visualisierungsdaten. Darüber hinaus müssen als Parameter die GReQL-Anfragen, die das Mapping spezifizieren, mittels einer HashMap sowie eine Instanz des zur Visualisierung passenden Renderers, wie er im vorhergehenden Abschnitt implementiert wurde, übergeben werden.
Der Ausführer muss an dieser Stelle folglich den *Ausgangs-TGraphen* wählen und das *Mapping* spezifizieren. Der Ausgangs-TGraph liegt dem Ausführer vor, da er in Schritt 1 durch den Analysierer zur Verfügung gestellt wurde.

Das Mapping: Anhand des Metamodells der Ausgangsdaten, wie es in Abb. 2.6 angegeben ist, kann das Mapping spezifiziert werden und folglich der Inhalt der Visualisierung festgelegt werden.
Im hier betrachteten Beispiel ist es gewünscht, dass die prozentuale Verteilung der Methoden auf die Klassen eines Pakets visuell dargestellt wird. Der logische Inhalt, das

Metamodell der Visualisierungsdaten und das Metamodell der Ausgangsdaten entsprechen dabei exakt dem Beispiel aus Abschnitt 2.4. Aus diesem Grund kann die in Abb. **??** dargestellte Mappingtabelle im folgenden als Übersicht verwendet werden.
Nachfolgende müssen daher die GReQL-Anfragen, die für alle Elemente aus dem Metamodell erstellt werden müssen, folglich die entsprechenden Elemente aus den Ausgangsdaten liefern. Hierzu müssen also GReQL-Anfragen für „Pie", „Piece", die Attribute „piename", „name", „value" und die Assoziation „Contains" geschrieben werden. Wird als Visualisierungsart das Tortendiagramm *mit* visuellen Attributen gewählt, so müssen an dieser Stelle zusätzlich auch noch GReQL-Anfragen für die Attribute „color" und „isThreeDimensional" definiert werden.
Die benötigten GReQL-Anfragen werden im Folgenden genauer beschrieben:
Zunächst soll die Torte das Paket mit dem Namen „package1" repräsentieren. Aus diesem Grund muss die GReQL-Anfrage den Knoten des Typs „Package" liefern, dessen Attribut „name" mit „package1" belegt wurde. Für den **Pie** sieht die GReQL-Anfrage daher wie folgt aus:

```
from p: V{Package}
with p.name="package1"
report p
end
```

Der Name der Torte - das Attribut **„piename"** - kann in diesem Fall einerseits durch das gewählte Paket gesetzt werden oder über eine selbst gewählte Konstante. Die nachfolgende GReQL-Anfrage zeigt, wie in GReQL das Attribut „piename" auf die Konstante „Prozentuale Verteilung der Methoden" gesetzt wird. Hierbei sollte jedoch beachtet werden, dass das Attribut „piename" zur Identifizierung der jeweiligen Torte dient. Eine Konstante hat somit die Auswirkung, dass bei der Erzeugung mehrerer Torten alle Torten identisch behandelt werden. Da im Ausgangs-TGraphen jedoch nur ein Paket mit Namen „package1" vorhanden ist, kann eine Konstante ohne weitere Auswirkungen gewählt werden:

```
from m: keySet(img_Pie)
reportMap m -> "Prozentuale Verteilung der Methoden"
end
```

Die **Tortenstücke** der gewählten Visualisierung repräsentieren die Klassen des Pakets, dazu werden über eine GReQL-Anfrage für „Piece" alle Klassen des Pakets „package1" ausgelesen. Dies sind genau die Knoten vom Typ „Class", die über eine Kante vom Typ „Contains" mit dem Knoten vom Typ „Package" verbunden sind, dessen Attribut „name" mit „package1" belegt ist. Dieser Zusammenhang wird durch die folgende GReQL-Anfrage ausgedrückt, die für die Tortenstücke die Knoten vom Typ „Class" zurückgibt, die diese Bedingung erfüllen:

```
from c: V{Class}, p: V{Package}
with p-->{Contains}c and p.name="package1"
report c
end
```

Der **Name der Stücke** soll folglich durch die Namen der entsprechenden Klasse belegt werden. Daher müssen an dieser Stelle nur die Ursprungsknoten, die für die „Piece(s)" verwendet wurden über „keySet(img_Piece)" ausgewählt werden und das Namensattribut des dazugehörigen „Piece(s)" muss lediglich den Wert des „name"-Attributs des Ursprungsknotens übernehmen. Dies ist genau das, was in der folgenden GReQL-Anfrage umgesetzt wird:

```
from m: keySet(img_Piece)
reportMap m -> m.name
end
```

Das Attribut „value" soll im gegebenen Beispiel die Anzahl der Methoden darstellen. Aus diesem Grund wird die Anzahl der Methoden für die jeweiligen Klassen über eine GReQL-Anfrage ausgelesen: Dazu werden die Ursprungsknoten erneut über „keySet(img_Piece)" gewählt und das Attribut „value" wird dann auf die Anzahl der Kanten vom Typ „HasMethod" gesetzt, die von der betrachteten Klasse ausgehen:

```
from m: keySet(img_Piece)
reportMap m->count(
                from e: E{HasMethod}
                with startVertex(e).name=m.name
                report e
                end)
end
```

Abschließend muss die Kante noch spezifiziert werden. Diese entspricht in der gegebenen Aufgabenstellung genau der Assoziation „Contains" des Ausgangs-TGraphen, so dass lediglich diese Kante angefragt werden muss:

```
from e: E{Contains}
reportSet e, startVertex(e), endVertex(e)
end
```

Insgesamt können die für die gewählte Aufgabenstellung spezifizierten Mappings wie folgt für die JVizAPI deklariert werden. Dabei sind die Namen aus dem Visualisierungsmetamodell die Schlüssel der HashMap (key-Parameter der `put` -Methode für HashMaps) und der Wert - der zweite String, der der HashMap über den `put`-Befehl übergeben wird - die GReQL-Anfrage, die die Ergebnisse für diese Elemente des Visualisierungsmetamodell liefert:

```
1 HashMap<String, String> greqlQueries = new HashMap<String, String>();
2 greqlQueries.put("Pie", "from p: V{Package} with p.name=\"package1\"
    report p end");
3 greqlQueries.put("piename","from m: keySet(img_Pie) reportMap m -> \"
    Prozentuale Verteilung der Methoden\" end");
4 greqlQueries.put("Piece", "from c: V{Class}, p: V{Package} with p-->{
    Contains}c and p.name=\"package1\" report c end");
5 greqlQueries.put("name", "from m: keySet(img_Piece) reportMap m -> m.
    name end");
6 greqlQueries.put("value", "from m: keySet(img_Piece) reportMap m->
    count(from e: E{HasMethod} with startVertex(e).name=m.name report
    e end) end");
7 greqlQueries.put("Contains", "from e: E{Contains} reportSet e,
    startVertex(e), endVertex(e) end");
```

Nachdem das Mapping und der Ausgangs-TGraph somit vorliegen, kann die execute-Methode wie folgt aufgerufen werden:

```
1         //Execute Transformation to get Visualization as SVG
2                 PieDiagramRenderer renderer = new PieDiagramRenderer
                      ();
3                 PieDiagramExecution p = new PieDiagramExecution();
4                 try {
5                         p.execute("Prozentuale Verteilung der
                              Methoden","src/AusgangsTGraph.tg", "src/
                              targetGraph.tg", greqlQueries, renderer);
6                 } catch (GraphIOException e) {
7                         e.printStackTrace();
8                 }
```

Als Ergebnis wird zum einen der Zielgraphen aus der GReTL-Transformation und zum anderen die gewünschte gerenderte Grafik als svg-Datei erhalten. Für das angebene Beispiel sind diese in Abb. 3.3 und Abb. 3.4 dargestellt.

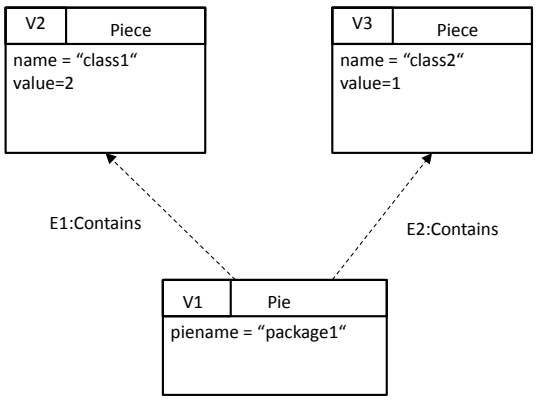


Abbildung 3.3: Ziel-TGraph der Beispieltransformation.

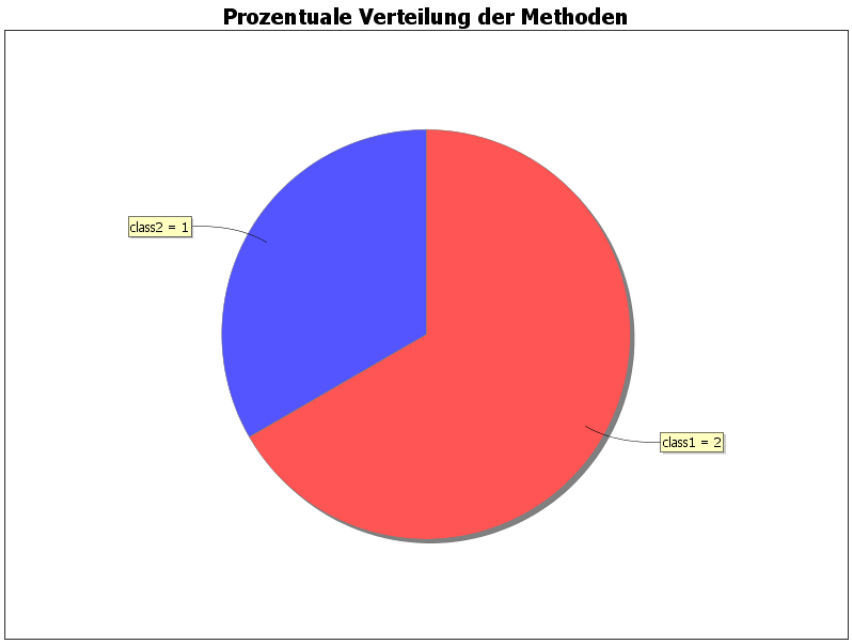


Abbildung 3.4: Tortendiagramm zur prozentualen Verteilung der Methoden auf die Klassen des fiktiven Systems.

Möchte der Ausführer zusätzlich die Farben der Tortenstücke bestimmen, so hätte an dieser Stelle auch die Visualisierung für Tortendiagramme mit visuellen Attributen gewählt werden können. Hier hätten jedoch noch zwei zusätzliche GReQL-Anfragen geschrieben werden müssen: Eine zum Belegen des Attributs „color" der Klasse „Piece" und eine zum Belegen des Attributs „isThreeDimensional" der Klasse „Pie". Diese hätten wie folgt aussehen können:

Das Attribut **„isThreeDimensional"** muss lediglich durch einen Boolean gesetzt werden, wobei „true" gewählt wird, falls eine dreidimensionale Darstellung gewünscht ist und „false", falls die Darstellung lediglich einen zweidimensionalen Plot haben soll:

```
from m: keySet(img_Pie)
reportMap m -> true
end
```

Das Attribut „color" kann durch Stringwerte aus einer Liste belegt werden, die jeweils die Farbe repräsentieren:

```
from m: keySet(img_Piece), n: list(\"green\", \"blue\")[id(m)-2]
reportMap m->n
end
```

Um die entsprechende Transformation ausführen zu können, müssen an dieser Stelle die dazu passenden Klassen zur Transformation und Ausführung gewählt werden. Um deutlich zu machen, dass es sich hierbei um eine andere Visualisierungsart als die vorherige handelt, wurde der Name „pieDiagramV" verwendet und es muss natürlich ein entsprechend passender Renderer verwendet werden, der in der Lage ist, die visuellen Angaben umzusetzen. Dieser befindet sich auf der beigelegten CD. Die Ausführung kann dann wie folgt stattfinden:

```
1  //Execute Transformation to get Visualization as SVG
2          PieDiagramRendererV renderer = new
               PieDiagramRendererV();
3          PieDiagramVExecution p = new PieDiagramVExecution();
4          try {
5                  p.execute("Prozentuale Verteilung der
                       Methoden","src/AusgangsTGraph.tg", "src/
                       targetGraph.tg", greqlQueries, renderer);
6          } catch (GraphIOException e) {
7                  e.printStackTrace();
8          }
```

Als Ergebnis werden der entsprechende Zielgraph, der nun über die zusätzlichen Attribute und ihre Belegung durch die in den GReQL-Anfragen angegebenen Werte verfügt und als konkrete Grafik, die in Abb. 3.5 dargestellte Scalable Vector Graphic, geliefert.

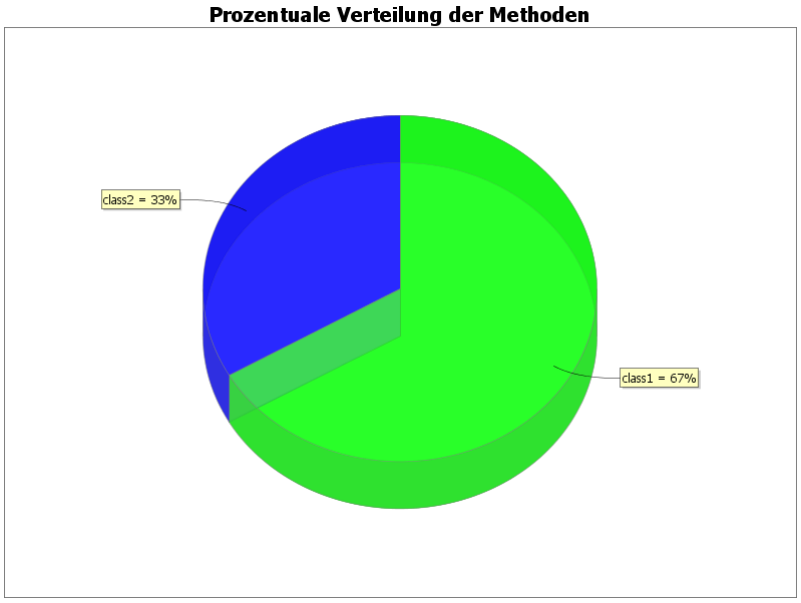


Abbildung 3.5: Tortendiagramm mit selbst gewählten visuellen Attributen.

4 | Validierung des Visualisierungsansatzes - Anwendungen des Konzeptes auf reale Systeme

Um den ELVIZ-Ansatz zu validieren, wird dieser unter Zuhilfenahme der zuvor beschriebenen JVizAPI für ein real existierendes System eingesetzt - die GPSPrintAPP.
Bei der GPSPrintAPP handelt es sich um eine APP für Android, die sich im BETA-Stadium befindet. Die APP dient dazu, aktuelle GPS Koordinaten mit Angabe der Genauigkeit dieser Daten anzuzeigen. Die GPS-Koordinaten können vom Benutzer unter Angabe einer Ortsbezeichnung gespeichert werden. Zusätzlich können die gespeicherten Orte in eine Textdatei exportiert werden. Der Quellcode dieser APP ist im Internet frei verfügbar.

4.1 Validierung und praktische Anwendung 1 - Das Tortendiagramm

In diesem Abschnitt werden zwei unterschiedliche Tortendiagramme erstellt - zunächst ein Tortendiagramm mit nur einer Torte und darauffolgend eine Grafik mit mehreren gleichzeitig erzeugten Torten. Die genauen Inhalte dieser beiden Visualisierungen werden im nachfolgenden genauer beschrieben.

4.1.1 Das einfache Tortendiagramm

Die Visualisierung für Tortendiagramme wie sie zuvor einmal an einem fiktiven Beispiel durchgeführt wurde, wird zur Validierung der Lösung mit den konkreten Analysedaten der GPSPrintApp verwendet. Hierzu können die generierte Transformation, die Transformationsausführung sowie der dazu bereits vorhandene Renderer wiederverwendet werden. Lediglich das Mapping muss an dieser Stelle durch den Ausführer der Transformation angepasst werden. Dieses soll die prozentuale Verteilung der Methoden auf die Klassen der GPSPrintAPP darstellen. Der TGraph der GPSPrintAPP ist konform zum SOAMIG-Metamodell:
Dieses Metamodell gehört ursprünglich zum SOAMIG-Projekt. Die Projektpartner sind die pro et con Innovative Informatikanwendungen GmbH, das Institut für Softwaretechnik der Universität Koblenz-Landau, das Offis in Oldenburg sowie die Amadeus Germany

GmbH. Bei dem SOAMIG-Projekt geht es darum sogenannte „Legacy-Systeme" - Altsysteme von Unternehmen, die im Laufe der Jahre stetig gewachsen sind, jedoch immer noch auf alten Technologien aufsetzen - an die neuen Technologien anzupassen. Dabei wird der Ansatz der Software-Migration verwendet und folglich werden die Altsysteme in eine neue Umgebung überführt, ohne dass sich die Funktionalität verändert. Dazu wurden Transformationen von Legacy-Systemen in serviceorientierte Architekturen verwendet [25].
Das dazugehörige SOAMIG-Metamodell verfügt dabei sowohl über einen Teil, der Java-Konstrukte spezifiziert als auch über einen Teil der Cobol definiert. In der Bachelorarbeit wird lediglich der umfangreiche Java-Teil des SOAMIG-Metamodells, der über 86 Assoziationen und 63 Klassen verfügt, verwendet, da lediglich Java-Systeme analysiert werden. Der an dieser Stelle relevante Ausschnitt aus dem SOAMIG-Metamodell ist in Abb. 4.1 dargestellt. Bei diesem Ausschnitt werden aus Gründen der Übersichtlichkeit nur die für die Problemstellung relevanten Klassen und Assoziationen gezeigt:
Die Klasse „SourceFile" beschreibt die Quelldateien des Systems und besitzt das Attribut „sourcePath" , welches den Pfad der Quelldatei als String speichern kann. Diese Klasse „SourceFile" erbt von der abstrakten Klasse „JavaFile". Diese ist wiederum über die Assoziation „HasTopLevelClassType" mit der abstrakten Klasse „ClassType" verbunden. Diese repräsentiert den Klassentyp. Dabei erbt „ClassType" von der abstrakten Klasse „JavaType" und verfügt folglich über die Attribute „modifier", „name" und „typeString". Außerdem erbt die Klasse „ClassType von der abstrakten Klasse „PackagableElement". Von dieser Klasse erbt auch die Klasse „JavaPackage". Mit dieser kann über die Assoziation „PackageContainsElement" die Zugehörigkeit einer bestimmten Klasse zu einem bestimmten Paket ausgelesen werden.
Die Klasse „MethodType" repräsentiert die Klassen. Es handelt sich hierbei somit auch um einen „JavaType". Für Methodenaufrufe existiert die Klasse „MethodCall" im SOAMIG-Metamodell.
Bestimmte Eigenschaften von Klassen und Methoden können über die Beziehungen von „ClassType", „MethodType" und „MethodCall" zu „DataObject" angefragt werden: Die Assoziation „HasField" bezieht sich auf die Attribute einer Klasse, die Assoziation „HasMethod" auf die Anzahl der Methoden und die Assoziation „CallsMethod" auf die Methodenaufrufe.

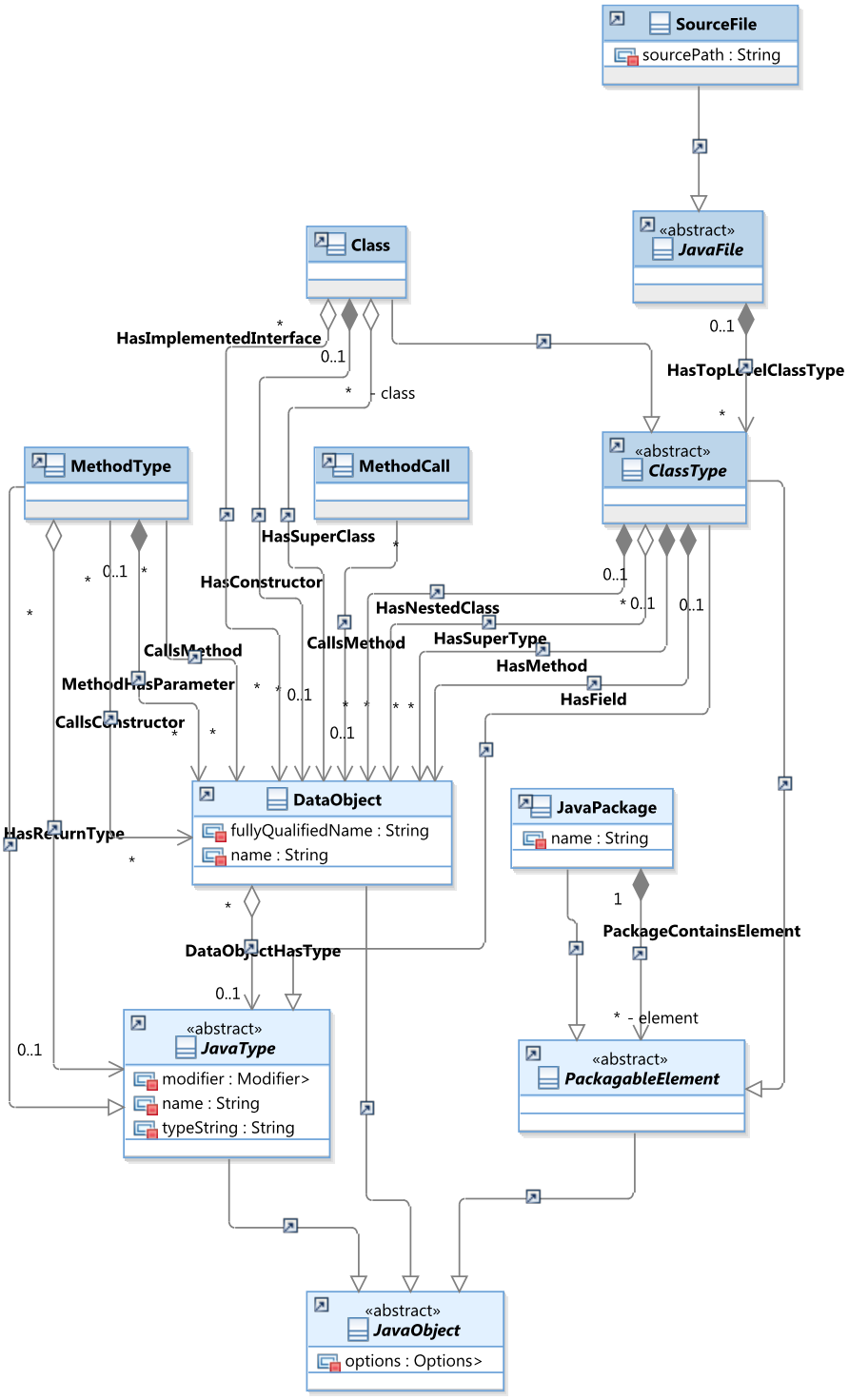


Abbildung 4.1: Auschnitt aus dem SOAMIG-Metamodell.

Anhand dieses Metamodells verändert sich die dritte Spalte der vorherigen Mappingtabelle. Diese ist in Abb. 4.2 dargestellt.

MM Visualisierung	Logischer Inhalt	MM Ausgangsdaten
Pie	Paket	JavaPackage
piename	Paketname	Konstante „Prozentuale Verteilung der Methoden“
Piece	Klasse	ClassType
name	Name der Klasse	name-Attribut von ClassType
value	Anzahl Methoden der Klasse	Anzahl HasMethod-Kanten von ClassType
Contains	Klasse gehört zum Paket	PackageContainsElement

Abbildung 4.2: Mappingtabelle für die Darstellung der prozentualen Verteilung der Methoden auf die Klassen der GPSPrintAPP.

Die dazugehörigen GReQL-Anfragen lassen sich wie folgt aufstellen:

```
1  HashMap<String, String> greqlQueries = new HashMap<String, String>();
2  greqlQueries.put("Pie", "from p: V{frontend.java.JavaPackage}
3                                  with p.name=\"gpsprint\"
4                                  report p
5                                  end");
6  greqlQueries.put("piename","from m: keySet(img_Pie)
7                              reportMap m -> \"Prozentuale Verteilung
                                  der Methoden\"
8                              end");
9  greqlQueries.put("Piece", "from e: E{frontend.java.
       PackageContainsElement}," +
10                               "p: V{frontend.java.JavaPackage}, b:
                                    V{frontend.java.ClassType} " +
11                               "with p.name=\"gpsprint\" and p--e->b
                                     "+
12                               "reportSet b
13                               end");
14 greqlQueries.put("name", "from m: keySet(img_Piece)
15                           reportMap m -> m.name
```

```
16                                  end");
17  greqlQueries.put("value", "from m: keySet(img_Piece) " +
18                   "reportMap m->count(from b: V{frontend.java.ClassType
                        }, c: V{frontend.java.DataObject} " +
19                   "with b-->{frontend.java.HasMethod}c and b.name=m.
                        name
20                   report c.name end)
21                   end");
22  greqlQueries.put("Contains", "from e: E{frontend.java.
        PackageContainsElement}," +
23                                     "p: V{frontend.java.JavaPackage}, b:
                                          V{frontend.java.ClassType} " +
24                                     "with p.name=\"gpsprint\" and p--e->b
                                          "+
25                                     "reportSet e, startVertex(e),
                                          endVertex(e) end");
```

Für „Pie" wird das Element vom Typ „JavaPackage" ausgelesen, dessen Namensattribut auf „gpsprint" gesetzt ist. Das Attribut „piename" wird mit der Konstanten „Prozentuale Verteilung der Methoden" belegt.
Zur Erzeugung der „Piece(s)" werden die Elemente vom Typ „ClassType" ausgelesen, die im Paket „gpsprint" enthalten sind und folglich über die Assoziation „PackageContainsElement" mit diesem verbunden sind. Das Namensattribut des Stücks wird dann auf das Namensattribut des Elements vom Typ „ClassType" gesetzt und das Attribut „value" wird mit der Anzahl der Methoden dieser Klasse belegt: Dieses entspricht der Anzahl der DataObjects die über eine Kante „HasMethod" mit dem jeweiligen Element vom Typ „ClassType" verbunden sind. Abschließend werden für die Assoziation „Contains" die Kanten vom Typ „PackageContainsElement" angefragt, die den vorherigen Bedingungen entsprechen.

Nach Aufruf der execute-Methode, kann die Visualisierung erhalten werden, die in Abb. 4.3 dargestellt ist.

In Abb. 4.3 zu erkennen, dass die Klasse „GPSItem" mit 22 Methoden, die höchste Anzahl an Methoden besitzt, während die Klassen „Help" und „ViewItem" überhaupt keine Methoden besitzen.

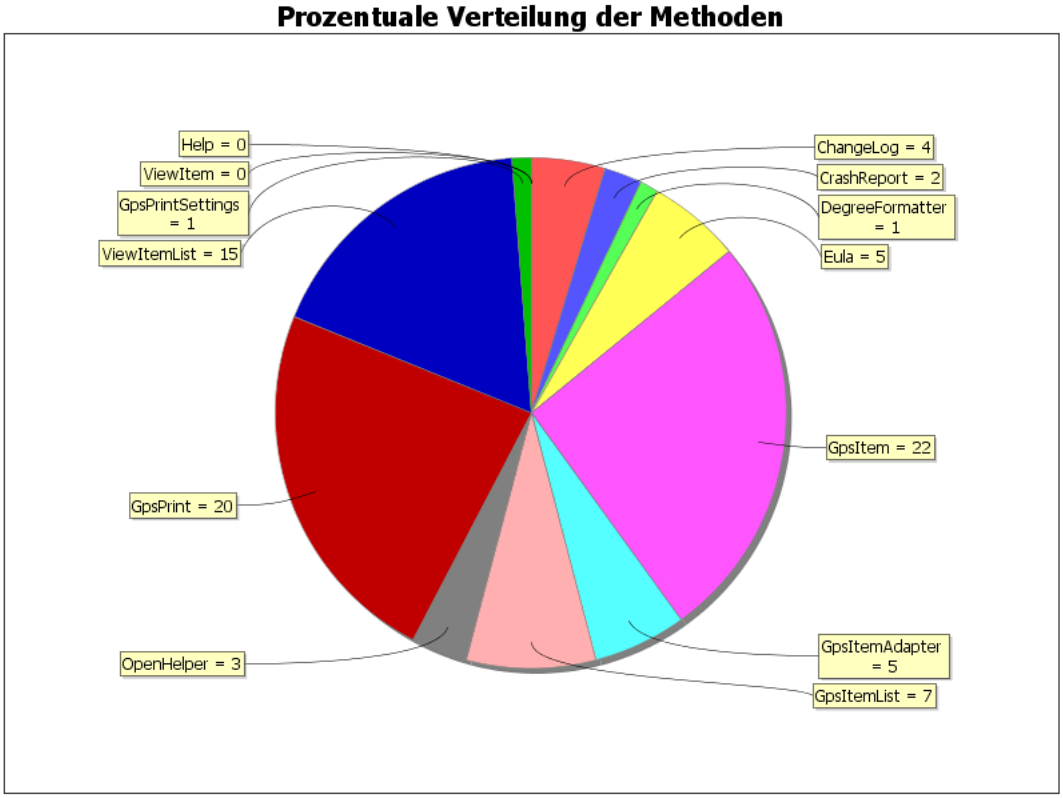


Abbildung 4.3: Tortendiagramm der GPSPrintAPP zur prozentualen Verteilung der Methoden auf die Klassen.

4.1.2 Diagramm mit mehreren Torten erzeugen

Mit einer veränderten Aufgabenstellung und gleichbleibender Visualisierungsart ist es auch möglich mehrere Tortendiagramme gleichzeitig zu erstellen, wenn das Mapping für „Pie" mehrere Elemente liefert. Als veränderte Aufgabenstellung sollen in diesem Fall die Aufrufe der Methoden visualisiert werden. Dazu soll eine Torte eine Klasse repräsentieren, das Tortenstück steht für die Methoden dieser Klasse und die Größe des Stücks wird durch die Anzahl der Methodenaufrufe gesetzt. Damit lässt sich die Mappingtabelle aus Abb. 4.4 aufstellen.

MM Visualisierung	Logischer Inhalt	MM Ausgangsdaten
Pie	Klasse	ClassType
piename	Klassenname	name-Attribut von ClassType
Piece	Methode	DataObject mit HasMethod-Kante zu ClassType
name	Name der Methode	name-Attribut des DataObjects
value	Anzahl Aufrufe der Methoden	Anzahl CallsMethod des DataObjects
Contains	Methode gehört zur Klasse	HasMethod

Abbildung 4.4: Mappingtabelle für die Darstellung der prozentualen Methodenaufrufe der GPSPrintAPP.

Daraus lassen sich die folgenden GReQL-Anfragen aufstellen:

```
1  HashMap<String, String> greqlQueries = new HashMap<String, String>();
2  greqlQueries.put("Pie", "from a: V{frontend.java.JavaFile}, b: V{
       frontend.java.ClassType},
3                           c: V{frontend.java.DataObject}
4                           with a-->{frontend.java.HasTopLevelClassType
                               }b
5                           reportSet b
6                           end");
7  greqlQueries.put("piename","from m: keySet(img_Pie)
8                              reportMap m -> m.name
9                              end");
10 greqlQueries.put("Piece", "from m: keySet(img_Pie), b: V{frontend.
       java.ClassType},
11                           c: V{frontend.java.DataObject} " +
12                          "with b-->{frontend.java.HasMethod}c and m.
                               name=b.name
13                          report c
14                          end");
15 greqlQueries.put("name", "from m: keySet(img_Piece)
16                          reportMap m -> m.name
17                          end");
18 greqlQueries.put("value", "from m: keySet(img_Piece)
```

```
19                                  reportMap m -> (count(from e: E{frontend.
                                        java.CallsMethod}
20                                                  with endVertex(e).
                                                        name==m.name
21                                                  reportSet
                                                        startVertex(e)
                                                        end))
22                                  end");
23  greqlQueries.put("Contains", "from b: V{frontend.java.ClassType},
24                          c: V{frontend.java.DataObject}, " +
25                                  "e: E{frontend.java.HasMethod}, a: V{
                                        frontend.java.JavaFile} " +
26                                  "with a-->{frontend.java.
                                        HasTopLevelClassType}b " +
27                                  " and  b--e->c "+
28                                  " reportSet e, startVertex(e),
                                        endVertex(e)
29                                  end");
```

Für „Pie" werden die Klassen ausgelesen - dementsprechend die Elemente vom Typ „ClassType" angefragt, die eine Beziehung „HasTopLevelClassType" zu einem Element vom Typ „JavaFile" haben. Als Name für die Torte wird dann der entsprechende Name der Klasse ausgelesen (siehe Zeile 7-9).
Die Stücke des jeweiligen Tortendiagramms werden durch die Methoden belegt, die zu dieser Klasse gehören. Diese können durch die Beziehung „HasMethod" angefragt werden. Der Name des Stücks wird entsprechend auf den Namen der Methode gesetzt und der Wert wird mit der Anzahl der Methodenaufrufe belegt. Dafür werden in Zeile 18-22 die Anzahl der Kanten vom Typ „CallsMethod" ausgelesen.
Abschließend werden die Kanten vom Typ „HasTopLevelClass" mit den zuvor genannten Bedingungen für die Kanten vom Typ „Contains" in den Visualisierungsdaten angefragt (Zeile 23-29).

Nach der Ausführung der Transformation werden die in Abb. 4.5 und 4.6 dargestellten Tortendiagramme erhalten. Die gesamten Torten befinden sich dabei in *einer* svg-Datei. Aufgrund der Vielzahl der Methoden wurde aus Gründen der Übersichtlichkeit die Grafik an dieser Stelle in Abb. 4.5 und 4.6 aufgeteilt. Da die Grafik aller Torten, auf der alle Labels sichtbar sind, zur direkten Einbindung ins A4-Format zu groß war, befindet sie sich auf der beigelegten CD.
An Abb. 4.5 und Abb. 4.6 ist jetzt beispielsweise zu sehen, welche Methoden niemals aufgerufen werden, z. B. die Methode „getView" der Klasse „GPSItemAdapter", während die Methode „show" der Klasse „Eula" 11-mal aufgerufen wird. Die Torten „CrashReport", „PreferenceActivity" und „SQLiteOPenHelper" sind nicht zu sehen: Diese erzeugen Torten ohne Stücke, da sie keine Methoden haben.

Abbildung 4.5: Tortendiagramm der GPSPrintAPP zur Anzeige der Methodenaufrufe - Teil1.

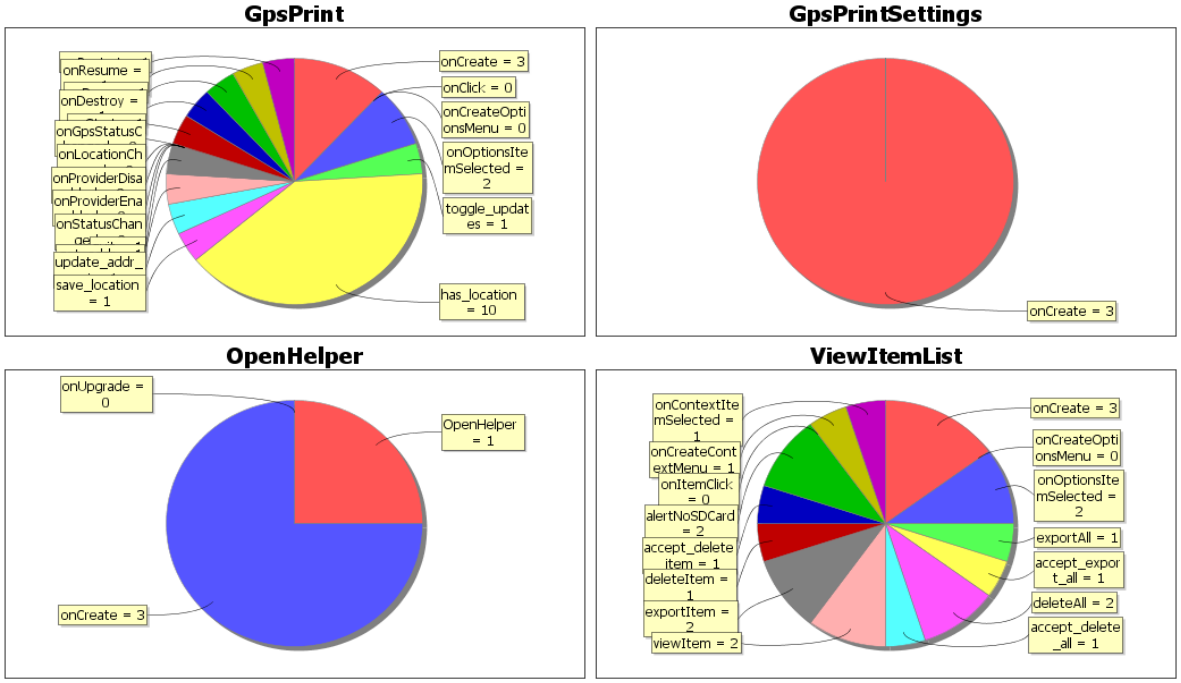


Abbildung 4.6: Tortendiagramm der GPSPrintAPP zur Anzeige der Methodenaufrufe - Teil2.

4.2 Validierung und praktische Anwendung 2 - Das Balkendiagramm

Neben dem Tortendiagramm ist es natürlich auch möglich eine andere Standardvisualisierungsart zu wählen. Dies kann zum Beispiel ein Balkendiagramm sein. Hierzu muss die Visualisierungsart zunächst wieder durch den Modellierer über ein Metamodell spezifiziert werden. Dieses ist in Abb. 4.7 dargestellt.

Abbildung 4.7: Metamodell eines Balkendiagrammes.

Dieses besteht aus einer Klasse „CoordinateSystem", welche das gesamte Koordinatensystem präsentiert und einer Klasse „Bar", die für die Balken in diesem Koordinatensystem steht. Dieser Balken kann über die Assoziation „Contains" einem bestimmten Koordinatensystem zugeordnet werden.
Die Klasse „CoordinateSystem" verfügt über ein Attribut „systemname", welches den Namen des Koordinatensystems als String enthält, und über die Attribute „xaxis" und „yaxis". Diese enthalten die Namen für die y-Achse sowie für die x-Achse.
Die Klasse „Bar" besitzt die Attribute „name" und „value" - wobei „name" den Namen des Balkens enthält und „value" die Höhe des Balkens festlegt.

Dieses muss dann wie gewohnt als Schema über JGraLab eingegeben werden, damit die Klassen zur Transformation und deren Ausführung über die JVizAPI erzeugt werden können.

Zusätzlich muss der dazu passende Renderer wie gewohnt vom Implementierer umgesetzt werden. Dies kann in diesem Fall wieder unter Zuhilfenahme der JFreeChartAPI geschehen:

```
1  public class BarDiagramRenderer extends Renderer {
2    public void createChart(Graph targetGraph) {
3          GreqlEvaluatorFacade evaluator = new GreqlEvaluatorFacade(
               targetGraph);
4          // read Coordinate System
5          JValueList cSystem = evaluator.evaluate("from c: V{
               CoordinateSystem} report c end").toJValueList();
6          JPanel panel = new JPanel(new GridLayout(cSystem.size(), 0));
7          // Create Systems:
8          for (int i = 0; i < cSystem.size(); i++) {
9                  // Get Bars in current Coordinate System
10                 JValueList bars = evaluator.evaluate("from p: V{Bar},
                       c: V{CoordinateSystem} with c-->{Contains}p and c
                       .systemname=""\"cSystem.get(i).toVertex().
                       getAttribute("systemname") + "\" report p end").
                       toJValueList();
11                 // set Values
12                 DefaultCategoryDataset dataset = new
                       DefaultCategoryDataset();
13                 for (int a = 0; a < bars.size(); a++) {
14                         dataset.setValue((Number) bars.get(a).
                               toVertex().getAttribute("value"), cSystem.
                               get(i).toVertex().getAttribute("xachses").
                               toString(),bars.get(a).toVertex().
                               getAttribute("name").toString());
15                 }
16         JFreeChart chart = ChartFactory.createBarChart(cSystem.get(i)
               .toVertex().getAttribute("systemname").toString(), cSystem
               .get(i).toVertex().getAttribute("xachses").toString(),
               cSystem.get(i).toVertex().getAttribute("yachses").toString
               (), dataset, PlotOrientation.VERTICAL,false, true, false);
17                 CategoryPlot plot = chart.getCategoryPlot();
18         panel.add(new ChartPanel(chart));
19         }
20         setContentPane(panel);
21   }
22 }
```

Dieser Renderer ist dem Renderer für Tortendiagramme sehr ähnlich: Hier werden in Zeile 5 zunächst die Koordinatensysteme ausgelesen und danach werden in einer Schleife alle Koordinatensysteme durchlaufen, die dazugehörigen Balken ausgelesen (Zeile 10) und der dazugehörige Datensatz erstellt (Zeile 12). Mit diesem kann dann abschließend über die JFreeChartAPI ein Balkendiagramm mit dem Befehl `createBarChart` erzeugt werden (Zeile 16).

Zuguterletzt muss der Ausführer der Transformation lediglich das Mapping spezifizieren. Das Balkendiagramm eignet sich beispielsweise um die Anzahl der Methoden je Klasse darzustellen. Dies ermöglicht wiederum die Identifizierung einer Gottklasse anhand eines sehr hohen Balkens. In diesem Fall stellen die Balken somit die Klassen dar und die Höhe der Balken wird durch die Anzahl der Methoden der jeweiligen Klasse festgelegt. Für diese Aufgabenstellung lässt sich somit die Mappingtabelle aus Abb. 4.8 aufstellen.

MM Visualisierung	Logischer Inhalt	MM Ausgangsdaten
CoordinateSystem	Paket	JavaPackage
systemname	Paketname	Konstante „Prozentuale Verteilung der Methoden“
xaxis	Beschriftung „Klassen“	Konstante „Klassen“
yaxis	Beschriftung „Anzahl Methoden“	Konstante „Anzahl Methoden“
Bar	Klasse	ClassType
name	Name der Klasse	name-Attribut von ClassType
value	Anzahl Methoden der Klasse	Anzahl HasMethod-Kanten von ClassType
Contains	Klasse gehört zum Paket	PackageContainsElement

Abbildung 4.8: Mappingtabelle für die Darstellung der prozentualen Verteilung der Methoden auf die Klassen für das Balkendiagramm.

Damit lassen sich folgende GReQL-Anfragen aufstellen:

```
1  HashMap<String, String> greqlQueries = new HashMap<String, String>();
2  greqlQueries.put("CoordinateSystem", "from p: V{frontend.java.
       JavaPackage}
3                                       with p.name=\"gpsprint\"
4                                       report p
5                                       end");
6  greqlQueries.put("systemname","from m: keySet(img_CoordinateSystem)
7                                  reportMap m -> \"Anzahl der Methoden
                                       der Klassen\"
8                                  end");
9  greqlQueries.put("xaxis","from m: keySet(img_CoordinateSystem)
10                              reportMap m -> \"Klassen\"
11                              end");
12 greqlQueries.put("yaxis","from m: keySet(img_CoordinateSystem)
13                              reportMap m -> \"Anzahl Methoden\"
14                              end");
15 greqlQueries.put("Bar", "from e: E{frontend.java.
       PackageContainsElement},” +
```

```
16                                  "p: V{frontend.java.JavaPackage}, a:
                                       V{frontend.java.JavaFile},
17                                   b: V{frontend.java.ClassType} " +
18                                  "with a-->{frontend.java.
                                       HasTopLevelClassType}b
19                                  and p.name=\"gpsprint\" and p--e->b "
                                       +
20                                  "reportSet b end");
21  greqlQueries.put("name", "from m: keySet(img_Bar)
22                            reportMap m -> m.name
23                            end");
24  greqlQueries.put("value", "from m: keySet(img_Bar) " +
25                  "reportMap m->count(from b: V{frontend.java.ClassType
                        },
26                  c: V{frontend.java.DataObject} " +
27                  "with b-->{frontend.java.HasMethod}c and b.name=m.
                        name
28                  report c.name end)
29                  end");
30  greqlQueries.put("Contains", "from e: E{frontend.java.
        PackageContainsElement},"  +
31                  "p: V{frontend.java.JavaPackage}, a: V{frontend.java.
                        JavaFile},
32                  b: V{frontend.java.ClassType} " +
33                  "with a-->{frontend.java.HasTopLevelClassType}b
34                  and p.name=\"gpsprint\" and p--e->b "+
35                  "reportSet e, startVertex(e),endVertex(e)
36                  end");
```

Die GReQL-Anfragen für „CoordinateSystem", „Bar", „name", „value" und „Contains" entsprechen den GReQL-Anfragen, die für das Tortendiagramm in Abschnitt 4.1.1 verwendet wurden, da hier der gleiche Visualisierungsinhalt und dasselbe Metamodell der Ausgangsdaten vorliegt.
Für „xaxis" und „yaxis" werden die Konstanten „Anzahl der Methoden der Klassen" und „Anzahl Methoden" verwendet (Zeile 9-14).

Damit wird das in Abb. 4.9 dargestellte Balkendiagramm erhalten. An diesem Diagramm ist - wie auch bereits beim Tortendiagramm in Abb. 4.3 - sehr schön zu sehen, dass die meisten Methoden in den Klassen „GPSItem" und „GPSPrint" vorliegen.

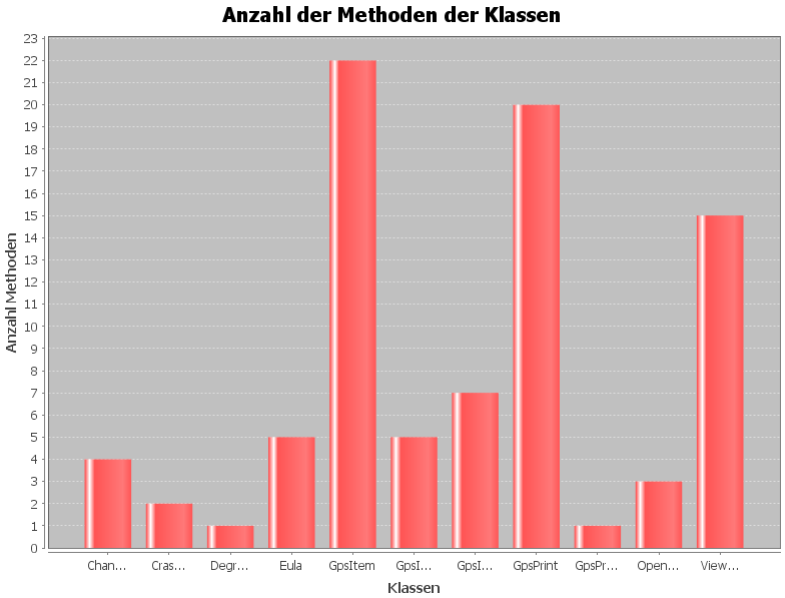


Abbildung 4.9: Balkendiagramm der GPSPrintAPP zur Anzeige der Anzahl der Methoden pro Klasse.

4.3 Validierung und praktische Anwendung 3 - CodeCity

Mit dem in der Bachelorarbeit entwickelten Softwarevisualisierungsansatz wird es auch möglich andere gängige Visualisierungen umzusetzen: So können mit diesem Ansatz auch sogenannte CodeCities erstellt werden.
Bei der CodeCity handelt es sich um eine Softwarevisualisierung, die Softwaresysteme in Form von Städten mit unterschiedlichen Distrikten und Gebäuden darstellt. Sie wurde im Jahr 2007 erstmals von Michele Lanza und Richard Wettel vorgestellt, da es bis zu diesem Zeitpunkt kaum 3D Darstellungen zur Softwarevisualisierung gab [27]. Diese Art der Visualisierung gewann 2008 den ersten Preis beim „Riconoscimento ated-ICT" TicinoWettbewerb [26]. Zudem haben die Autoren Wettel und Lanza im Jahr 2011 in einer empirischen Studie nochmals nachgewiesen, dass CodeCities zum einen die Aufgabenkorrektheit erhöhen und zum anderen für eine Zeitersparnis bis zur Beendigung einer Aufgabe sorgen. Insgesamt handelt es sich somit um eine nachgewiesen sinnvolle Visualisierung, die zur Analyse von Softwaresystemen bereits real existiert.

Um diese Visualisierung mit dem in der Bachelorarbeit entwickelten Softwarevisualisierungsansatz umsetzen zu können, muss zunächst die Visualisierung über das Metamodell spezifiziert werden. Aus den Angaben in der Literatur kann dieses wie in Abb. 4.10 dargestellt umgesetzt werden.

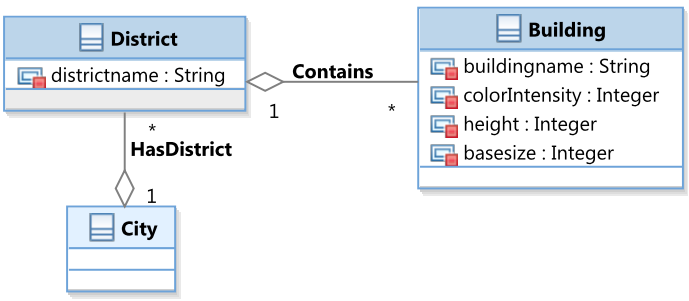


Abbildung 4.10: Metamodell der CodeCity.

Aus dem Metamodell in Abb. 4.10 wird ersichtlich, dass eine CodeCity aus verschiedenen Distrikten besteht, die jeweils über einen Namen verfügen. Ein Distrikt besteht wiederum aus beliebig vielen Gebäuden, die einen Namen haben, eine bestimmte Höhe, eine Grundfläche und einen Integerwert zur Bestimmung der Farbintensität.
Mit diesem Metamodell kann wie gewohnt ein Schema in JGraLab angelegt werden und dieses kann zur Generierung der Transformation sowie der Transformationsausführung in der JVizAPI genutzt werden.

Der dazugehörige Renderer sieht deutlich komplexer aus, als der eines Balken- oder Tortendiagramms. In der Originalimplementierung von Lanza und Wettel wurde dazu OpenGL verwendet [26].Im Rahmen der vorliegenden Arbeit wurde hingegen ein Rendering über Java 3D gewählt. Der Quellcode dazu befindet sich auf der beigelegten CD. Er umfasst über 400 Codezeilen mit aufwendigen Berechnungen beispielsweise zu korrekten Anordnungen der variablen Anzahl an Gebäuden. Dies verdeutlicht noch einmal wie sinnvoll das in der Bachelorarbeit präsentierte Rollenkonzept ist, da sich ein Implementierer so vollständig dieser Aufgabe widmen kann.
Ein weiterer Renderer, der in der Lage ist das originalgetreue Rendering vorzunehmen, ließe sich an dieser Stelle natürlich auch über OpenGL implementieren.

Abschließend muss der Ausführer noch das Mapping spezifizieren, wie es in der Literatur für CodeCities vorgegeben ist: Dementsprechend repräsentiert die City das gesamte Softwaresystem, die Distrikte stehen für die unterschiedlichen Pakete des Systems und die Gebäude in diesen Distrikten für die Klassen in diesen Paketen. Die Höhe des jeweiligen Gebäudes wird dann durch die Anzahl der Methoden dieser Klasse festgelegt, die Grundfläche durch die Anzahl der Attribute und die Farbintensität durch die Anzahl der Codezeilen der betrachteten Klasse. Dabei gilt: Je mehr Zeilen Code eine Klasse besitzt, umso intensiver ist die Farbe. Aus dieser Zuordnung kann die Mappingtabelle, die in Abb. 4.11 dargestellt ist, aufgestellt werden.

MM Visualisierung	Logischer Inhalt	MM Ausgangsdaten
City	Gesamtes System	Gesamter Graph
District	Paket	JavaPackage
districtname	Name des Pakets	Name-Attribut von JavaPackage
Building	Klasse	ClassType
buildingname	Name der Klasse	name-Attribut von ClassType
colorIntensity	Anzahl an LoC der Klasse	name-Attribut von ClassType
height	Anzahl Methoden der Klasse	Anzahl HasMethod-Kanten von ClassType
basesize	Anzahl Attribute der Klasse	Anzahl HasField-Kanten von ClassType
HasDistrict	Paket gehört zum System	Da gesamter Graph gewählt wird keine solche Verbindung
Contains	Klasse gehört zum Paket	PackageContainsElement

Abbildung 4.11: Mappingtabelle für die CodeCity.

Damit ergeben sich die folgenden GReQL-Anfragen:

```
1  HashMap<String, String> greqlQueries = new HashMap<String, String>();
2  //packages:
3  greqlQueries.put("District", "from p: V{frontend.java.JavaPackage}
       with p.name=\"gpsprint\" report p end");
4  greqlQueries.put("districtname","from m: keySet(img_District)
       reportMap m -> m.name end");
5
6  //classes
7  greqlQueries.put("Building", "from e: E{frontend.java.
       PackageContainsElement}," +
8                                  "p: V{frontend.java.JavaPackage}, a:
                                      V{frontend.java.JavaFile}, b: V{
                                      frontend.java.ClassType} " +
9                                  "with a-->{frontend.java.
                                      HasTopLevelClassType}b and p.name
                                      =\"gpsprint\" and p--e->b "+
10                                 "reportSet b end");
11 greqlQueries.put("buildingname", "from m: keySet(img_Building)
       reportMap m -> m.name end");
12
13 //number of statements:
14 greqlQueries.put("color", "from m: keySet(img_Building) reportMap m->
       count" +
15                                 "(from file: V{frontend.java.
                                      SourceFile}, c: V{frontend.java.
                                      ClassType}, " +
16                                 "stmt: V{frontend.java.JavaStatement}
                                       with file-->*stmt " +
17                                 "and not file-->{frontend.java.ext.
                                      CallsMethod}stmt " +
18                                 "and file-->{frontend.java.
                                      HasTopLevelClassType}c and c.name=
                                      m.name report stmt end) end");
19
20 //number of methods:
21 greqlQueries.put("height", "from m: keySet(img_Building) " +
22                                 "reportMap m->count(from b: V{
                                      frontend.java.ClassType}, c: V{
                                      frontend.java.DataObject} " +
23                                 "with b-->{frontend.java.HasMethod}c
                                      and b.name=m.name report c.name
                                      end) end");
24
25 //number of attributes:
26 greqlQueries.put("basesize", "from m: keySet(img_Building) " +
```

```
27                                  "reportMap m->count(from b: V{
                                        frontend.java.ClassType}, c: V{
                                        frontend.java.DataObject} " +
28                                  "with b-->{frontend.java.HasField}c
                                        and b.name=m.name report c.name
                                        end) end");
29
30  greqlQueries.put("Contains", "from e: E{frontend.java.
        PackageContainsElement}," +
31                                  "p: V{frontend.java.JavaPackage}, a:
                                        V{frontend.java.JavaFile}, b: V{
                                        frontend.java.ClassType} " +
32                                  "with a-->{frontend.java.
                                        HasTopLevelClassType}b and p.name
                                        =\"gpsprint\" and p--e->b "+
33                                  "reportSet e, startVertex(e),
                                        endVertex(e) end");
```

Da für „City" und „HasDistrict" keine Einschränkungen bestehen, da das gesamte System für die City betrachtet wird, werden diese für die Implementierung außer Acht gelassen. Damit wird es nicht möglich mehrere Cities in einer einzigen Ausführung zu erstellen.
In Zeile 6 werden die Elemente vom Typ „JavaPackage" wie bereits zuvor für die Torten- und Balkendiagramme ausgelesen und ihre Ergebnisse zur Erzeugung der Distrikte verwendet. Der Distriktname entspricht dabei dem Namen des Pakets (Zeile 7).
Für die Gebäude werden die Klassen ausgelesen. Auch an dieser Stelle wird eine GReQL-Anfrage verwendet, die auch bereits bei den Balken- und Tortendiagrammen zum Einsatz kam (Zeile 10-14).
Die Lines of Code können nicht mit dem SOAMIG-Metamodell ausgelesen werden. Stattdessen werden die Anzahl der Statements ausgelesen. Dies entspricht der Anzahl der Kanten, die in einem beliebigen Pfad vom entsprechenden „JavaFile" der jeweiligen Klasse zu der abstrakten Klasse „JavaStatement" führen (Zeile 17-22).
Für die Anzahl der Methoden und für die Assoziation „Contains" konnte wieder eine GReQL-Anfrage verwendet werden, die bereits vorher für Balken- und Tortendiagramme) aufgestellt wurde (Zeilen 24-26 und Zeilen 33-36).
Die Anzahl der Attribute kann in GReQL zuguterletzt über die Anzahl der DataObjects ausgelesen werden, die mit der entsprechenden Klasse über eine Kante vom Typ „HasField" verbunden sind.

Nach Ausführung der execute-Methode wird die in Abb. 4.12 dargestellte Grafik als svg-Datei geliefert.

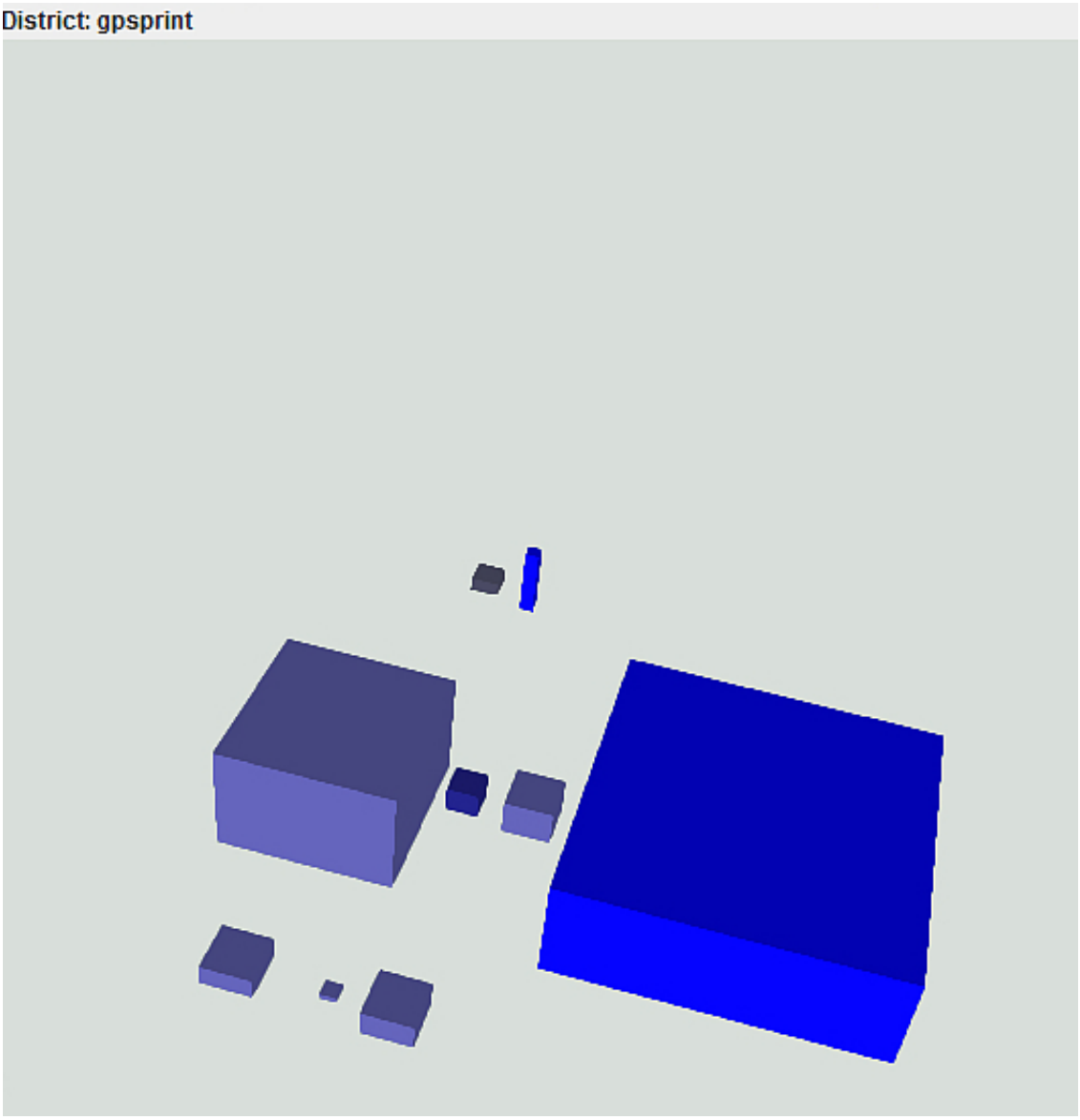


Abbildung 4.12: CodeCity der GPSPrintAPP.

In Abb. 4.12 sind zwei recht große Gebäude in der mittleren Reihe erkennbar: Dies zeigt, dass eine recht große Anzahl an Methoden, aber auch recht viele Attribute vorliegen. Es ist allerdings auch erkennbar, dass nur 8 Gebäude vorhanden sind, obwohl die GPSPrintAPP über 13 Klassen verfügt. Dies liegt daran, dass 5 Klassen keine Attribute oder keine Methoden haben - damit wird ihre „basesize" oder ihre „height" entsprechend auf 0 gesetzt und damit sind sie nicht sichtbar.
An dieser Stelle sind zudem noch keine Namen der Gebäude und damit der Klassen zu

sehen. Diese Information ist aufgrund des Metamodells in den Visualisierungsdaten verfügbar. In Zukunft kann dafür beispielsweise ein Renderer implementiert werden, der eine anklickbare Anwendung erzeugt, so dass zusätzliche Informationen angezeigt werden. Dies ist ohne weitere Probleme umsetzbar, da die einzige Voraussetzung darin besteht, dass der Renderer die Grafik in einem JFrame präsentiert.

Die Visualisierungsart der CodeCities wurde auch noch einmal zur Visualisierung der entwickelten JVizAPI verwendet, die den ELVIZ-Ansatz zur Validierung umsetzt. Die dazugehörige CodeCity ist in Abb. 4.13 zu sehen.

Abbildung 4.13: CodeCity der JVizAPI.

Auf dieser CodeCity sind nur drei Gebäude abgebildet - obwohl die eigentliche JVizAPI aus vier Klassen besteht. Dies resultiert daraus, dass die Klasse „Renderer" keine Attribute besitzt. Aus diesem Grund wird die „basesize" des Gebäudes für den Renderer auf 0 gesetzt. Dementsprechend ist dafür das Gebäude nicht zu sehen.

5 | Schluss

Nachdem der in der Bachelorarbeit entwickelte modellgetriebene Softwarevisualisierungsansatz in den vorhergehenden Abschnitten eingehend beschrieben und validiert wurde, wird in diesem Kapitel darauf eingegangen, welche besonderen Herausforderungen und Probleme bei der Entwicklung aufgetaucht sind, wie die Lösung darauf aufbauend bewertet werden kann und welche Möglichkeiten sich in Zukunft zur Weiterentwicklung des ELVIZ-Ansatzes bieten.

5.1 Herausforderungen und Bewertung des ELVIZ-Ansatzes

Die besondere Herausforderung der Bachelorarbeit lag darin, einen komplett generischen Ansatz zu entwickeln: Generisch in Bezug auf die Ausgangsdaten, auf die Visualisierungsart und auf den Visualisierungsinhalt. Eine Lösung, die in der Bachelorarbeit dafür gefunden werden konnte, lag in der doppelten Verwendung der Model Driven Architecture und damit einhergehend in der Transformation beliebiger Ausgangsdaten in die zur Visualisierungsart passenden Visualisierungsdaten.
Da der ELVIZ-Ansatz so generisch gehalten wurde, hängen mögliche Schwächen dieses Ansatzes immer von den verwendeten Techniken zur Umsetzung ab:
In der Validierung im vorherigen Teil der Arbeit konnten beispielsweise vorhandene Grundlagen der Graphentechnik sehr gewinnbringend eingesetzt werden. Dabei gibt es sowohl einige Vorteile als auch einige Nachteile, die diese Realisierung hat:
Die Ausgangsdaten sind in dieser Implementierung konform zum SOAMIG Metamodell. Aus diesem Grund sind die Inhalte der Visualisierung stark abhängig von den Möglichkeiten, die das SOAMIG Metamodell bietet. Beispielsweise sind in diesem Metamodell keine Informationen über die Lines of Code einzelner Klassen verfügbar. Übertragen auf den allgemeinen Ansatz besagt dies, dass der Inhalt der Visualisierung stark abhängig vom Metamodell der Ausgangsdaten ist. Dies ist jedoch eine Schwachstelle der spezifischen Umsetzung mit SOAMIG unter Verwendung der Analysewerkzeuge, die es bereits an der Universität Oldenburg gibt. Der allgemeine Ansatz hat diesen Nachteil nicht, da er beliebige Ausgangsdaten und diverse Analysewerkzeuge zulässt.
Auch die Verwendung einer bestimmten Transformationssprache hat stets Vor- und Nachteile. Die Entscheidung für eine Transformationssprache sollte dabei immer unter Berücksichtigung dieser Vor- und Nachteile und unter Einbeziehung der Struktur der Ausgangsdaten geschehen:
Zur Transformation der Ausgangsdaten in die Visualisierungsdaten wurde in der beispielhaften Implementierung GReTL verwendet, da die Ausgangsdaten als TGraphen vorla-

gen. GReTL unterstützt jedoch bei der Anlegung von Attributen lediglich die Datentypen String, Integer, Double, Long und Boolean. Andere Datentypen müssen dementsprechend im Renderer gesondert behandelt werden: So wurde beispielsweise in der GReTL-Transformation aus Abschnitt 3.1.2 für das Attribut „color" ein String anstelle des Datentyps „Color" von Java AWT verwendet. Der Renderer musste dies entsprechend auslesen und in ein Objekt vom Typ „Color" umwandeln. Dies ist jedoch der einzige Nachteil den GReTL als Transformationssprache mit sich bringt. Der große Vorteil bei der Verwendung von GReTL ist, dass es speziell für die Arbeit mit TGraphen entworfen wurde. Darüber hinaus ist GReTL so konzipiert, dass es neben dem konkreten TGraphen auch das Schema anlegt. Dadurch wurde die automatische Generierung der Transformation aus dem Metamodell der Visualisierung vereinfacht. Die alternative Verwendung von ATL oder QVT hätte sich an dieser Stelle nicht angeboten, da sie die direkte Arbeit mit TGraphen nicht unterstützen.
Auch bei der internen Umsetzung der bestehenden Renderer gibt es in Zukunft noch Verbesserungspotential: Beispielsweise ist es in zukünftigen Arbeiten erstrebenswert mit diesem Ansatz anklickbare CodeCities zu erstellen. Damit wird es mit diese Ansatz möglich nicht nur Grafiken zu erstellen, sondern interaktive Anwendungen, die das Softwaresystem repräsentieren.

Insgesamt wird deutlich, dass die Schwächen, die der ELVIZ-Ansatz haben kann, stets von den Implementierungsentscheidungen abhängt. Die Implementierungsentscheidungen können aber natürlich auch Vorteile mit sich bringen: So erreicht der ELVIZ-Ansatz nicht nur, dass die in der Defintion des Zieles der Bachelorarbeit betrachteten Aspekte - Ausgangsdaten, Visualisierungsart und Visualisierungsansatz - komplett generisch sind, sondern auch, dass die Implementierungstechnik vollständig frei wählbar ist. Damit wird es in diesem Ansatz möglich die Vorteile bestimmter Techniken zu nutzen und zu kombinieren: Dies gewährleistet den fortwährenden Einsatz innovativer Techniken. Damit profitiert der Visualisierungsansatz auch in Zukunft von Vorteilen zukünftiger Technologien: Er bleibt damit nicht nur stets aktuell, sondern kann stets die passendsten und schnellsten Technologien verwenden. So können beispielsweise die Vorteile einer neuen Transformationssprache mit diesem Ansatz vollständig genutzt werden.
So konnte das in der Arbeit angestrebte Ziel erreicht werden und darüberhinaus ein Ansatz mit Zukunftspotential entwickelt werden. Folglich konnte für das anfängliche Problem eine sehr gute Lösung gefunden werden.
Grenzen dieses Ansatzes finden sich genau dann, wenn zu bestimmten Ausgangsdaten kein Metamodell verfügbar ist oder wenn eine gewünschte Visualisierungsart nicht präzise über ein Metamodell spezifiziert werden kann.
Ansonsten ist der ELVIZ-Ansatz zur Erstellung komplett personalisierter Visualisierungen mit beliebigen Technologien sehr gut einsetzbar.

5.2 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde ein neuer Visualisierungsansatz - der ELVIZ-Ansatz - entwickelt. Dieser kann Software auf jede beliebige Art und Weise wieder „sichtbar" zu machen. Dies kann die Produktivität in einem Softwareentwicklungsprozess steigern, die Verstehenskomplexität reduzieren, Schwachstellen von Software leichter identifizierbar machen oder als übersichtliche Kommunikationsgrundlage dienen. Dabei wurde darauf geachtet, dass möglichst viele Teile - wie die Spezifizierung der Visualisierungsart über das Visualisierungsmetamodell, die Spezifizierung des Visualisierungsinhalts über das Mapping oder vorhandene Ausgangsdaten - wiederverwendbar sind. Durch Einsatz der Model Driven Architecture konnten darüber hinaus große Teile des Ansatzes automatisch generiert werden, so dass der ELVIZ-Ansatz an dieser Stelle noch weitere Möglichkeiten der Zeitersparnis bei der Erstellung neuer Grafiken bietet.
Darüber hinaus ist die gewählte Lösung komplett generisch - sowohl in Bezug auf Visualisierungsart, -inhalt, Beschaffenheit und Inhalt der Ausgangsdaten als auch in Bezug auf Implementierungsmöglichkeiten. Dies gewährleistet, dass der ELVIZ-Ansatz auch in Zukunft weiter einsetzbar bleibt und stets Vorteile neuer Technologien einbeziehen kann. Da er in Zukunft weiter einsetzbar ist und eine Möglichkeit bietet, komplett personalisierte „Sichtbarmachungen" von Software zu erstellen, wurde er bereits beim diesjährigen SATToSE Seminar in Koblenz vorgestellt. Hierbei handelt es sich um ein internationales „Seminar on Advanced Techniques and Tools for Software Evolution", das dieses Jahr im Rahmen der SoTeSoLa Summer School (Research 2.0 event on software technologies and software languages) stattfand [19].
In der Validierung in Teil 4 der Bachelorarbeit konnte gezeigt werden, dass dieser Ansatz real einsetzbar ist. Durch die JVizAPI und die verwendeten Grundlagen der Graphentechnik existiert jetzt eine mögliche Implementierung dieses Ansatzes. Diese Implementierung kann zukünftig verwendet werden, da sie ohne Einschränkungen funktioniert. Mit dieser wird es möglich viele neue Visualisierungsarten zu erstellen: Die generierten Klassen zur Transformation und Transformationsausführung sowie die passenden Renderer und eine Sammlung von Mappings für unterschiedliche Aufgabenstellungen, könnten dabei in einem gemeinsamen Repository gespeichert werden, so dass nach und nach sehr viele unterschiedliche Möglichkeiten der „Sichtbarmachung" zur Ausführung bereitstehen. Weiterhin sind zukünftig noch andere Implementierungen möglich, so dass auch Vorteile anderer eventuell neuer Technologien, in diesem Ansatz Anwendung finden.
Eine weitere Idee für eine zukünftige Arbeit wäre es den ELVIZ-Ansatz zu nutzen, um interaktive Anwendungen von Softwaresystemen zu entwickeln, die ein Softwaresystem repräsentieren.

Alles in allem kann somit gesagt werden, dass das anfängliche Problem durch den hier präsentierten ELVIZ-Ansatz ohne Einschränkungen gelöst werden konnte. Dieser Ansatz ist in Zukunft weiterhin einsetzbar: Dabei kann sowohl die in der Bachelorarbeit implementierte JVizAPI zur Erstellung einer neuen Visualisierung und Grafik genutzt werden als auch neue Implementierungen mit anderen innovativen Techniken erstellt werden, die

der Arbeitsweise des in der Bachelorarbeit präsentierten Ansatzes folgen.
Insgesamt ist es so möglich Software unter Verwendung vielfältiger Technologien auf jede beliebige Art und Weise „sichtbar" zu machen.

6 | Übersicht über den Inhalt der CD

Auf der beigelegten CD befinden sich alle notwendigen Libraries, Quellcode, Texte und Grafiken. Abb. 6.1 zeigt die Ordnerstruktur: Dabei existieren zunächst die drei Hauptordner: 1-Text, 2-Quellcode und Artefakte und 3-Grafiken.

Als Texte befinden sich das PDF der Bachelorarbeit sowie die Ergebnisse der Machbarkeitsstudien auf der CD.
Unter 2-Quellcode und Artefakte befinden sich im Ordner 1-Libraries alle notwendigen Bibliotheken für Java (Batik, JFreeChart, JGraLab, smgschema und die JVizAPI als JAR). Im Ordner 2-JVizAPI befindet sich der Quellcode der API.
Unter 3-Wiederverwendbare Artefakte befinden sich alle Artefakte, die im Laufe der Bachelorarbeit entwickelt wurden und in Zukunft weiter verwendet werden können, da sie wiederverwendbar sind. Dazu gehören die 1-Ausgangsdaten, d.h. die TGraphen des fiktiven Systems, der GPSPrintAPP und der JVIZAPI sowie 2-Transformationen und Renderer. In diesem Ordner befinden sich die ausführbaren Transformationen von Tortendiagrammen, Balkendiagrammen und CodeCities. In Abb. 6.1 ist er grün markiert, d.h. er unterteilt sich dabei in die Unterordner Tortendiagramm, Balkendiagramm und CodeCity um die Übersicht zu gewährleisten.
Dasselbe gilt für den Ordner 4-Validierung, indem sich der Quellcode zu allen durchgeführten Validierungen befindet und den Ordner 3-Grafiken, indem sich alle erzeugten SVG-Grafiken befinden.

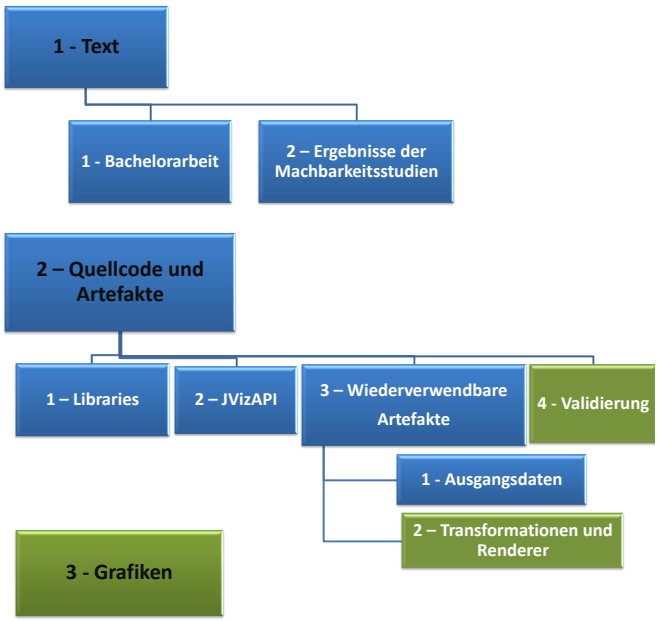


Abbildung 6.1: Ordnerübersicht der beigelegten CD.

Literaturverzeichnis

[1] Thomas Ball and Stephen G. Eick. Software visualization in the large. *Computer*, 29(4):33–43, April 1996.

[2] Jean Bézivin. In Search of a Basic Principle for Model Driven Engineering. *UPGRADE – The European Journal for the Informatics Professional*, 5(2):21–24, 2004.

[3] Jean Bézivin. Model Driven Engineering: An Emerging Technical Space. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, chapter 2, pages 36–64. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2006.

[4] Daniel Bildhauer. *Auswertung der TGraphanfragesprache GReQL 2*. VDM Verlag Dr. Müller, Saarbrücken, 2008.

[5] Daniel Bildhauer and Jürgen Ebert. Querying Software Abstraction Graphs. In *Working Session on Query Technologies and Applications for Program Comprehension (QTAPC 2008), collocated with ICPC 2008*, 2008.

[6] Jürgen Ebert. A Versatile Data Structure For Edge-Oriented Graph Algorithms. *Communications ACM*, 30(6):513–519, 1987.

[7] Jürgen Ebert and Daniel Bildhauer. Reverse Engineering Using Graph Queries. In Andy Schürr, Claus Lewerentz, Gregor Engels, Wilhelm Schäfer, and Bernhard Westfechtel, editors, *Graph Transformations and Model Driven Engineering*, LNCS 5765. Springer, 2010.

[8] Jürgen Ebert and Angelika Franzke. A declarative approach to graph based modeling. In Ernst Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science*, pages 38–50. Springer Berlin / Heidelberg, 1995.

[9] Jürgen Ebert and Tassilo Horn. GReTL: an extensible, operational, graph-based transformation language. *Software and Systems Modeling*, pages 1–21.

[10] Jürgen Ebert, Bernt Kullbach, Volker Riediger, and Andreas Winter. GUPPRO. Generic Understanding of Programs - An Overview. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.

[11] Jürgen Ebert, Volker Riediger, and Andreas Winter. Graph Technology in Reverse Engineering, The TGraph Approach. In Rainer Gimnich, Uwe Kaiser, Jochen Quante, and Andreas Winter, editors, *10th Workshop Software Reengineering (WSR 2008)*, volume 126, pages 67–81, Bonn, 2008. GI.

[12] Jean-Marie Favre. Towards a basic theory to model model driven engineering. In *In Workshop on Software Model Engineering, WISME 2004, joint event with UML2004*, 2004.

[13] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[14] Ralf Pohlmann/aptico GmbH. SVG Tutorial - Version 1.1, 2011. url: http://svg.tutorial.aptico.de/svg-workshop.pdf, (7.06.2012).

[15] Tassilo Horn. Model migration with gretl. In *Transformation Tool Contest*, Malaga, 2010.

[16] Tassilo Horn and Jürgen Ebert. The GReTL Transformation Language. In Jordi Cabot and Eelco Visser, editors, *Theory and Practice of Model Transformations*, volume 6707 of *Lecture Notes in Computer Science*, pages 183–197. Springer Berlin / Heidelberg, 2011.

[17] Manfred Kamp. GReQL - eine Anfragesprache für das GUPRO–Reporsitory. In Andreas Winter, H. Stasch, Rainer Gimnich, and Jürgen Ebert, editors, *GUPRO — Generische Umgebung zum Programmverstehen*, pages 173–202. Föllbach, 1998.

[18] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[19] Universität Koblenz. SaTToSE 2012 - 5th Seminar Series on Advanced Techniques and Tools for Software Evolution , 2012. url: https://github.com/SoTeSoLa/SoTeSoLa/wiki/SATToSE-2012, (20.10.2012).

[20] Universität Koblenz-Landau. Graphentechnologie, 2011. url: http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IST/AGEbert/MainResearch/ Graphentechnologie, (20.05.2012).

[21] Rainer Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance*, 15(2):87–109, March 2003.

[22] ingenieurbüro für softwaretechnologie Markus Voelter. Modellgetriebene Software-

entwicklung, url: http://www.voelter.de/data/articles/MDSD.pdf , (18.06.2012).

[23] Hausi A. Müller, Scott R. Tilley, and Kenny Wong. Understanding software systems using reverse engineering technology perspectives from the rigi project. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering - Volume 1*, CASCON '93, pages 217–226. IBM Press, 1993.

[24] OMG. OMG Model Driven Architecture, 2012. url: http://www.omg.org/mda/, (03.10.2012).

[25] pro et con Innovative Informatikanwendungen GmbH. KMU-innovativ: Informations- und Kommunikationstechnologien (IKT),Verbundprojekt SOAMIG: Migration von Legacy-Software in serviceorientierte Architekturen, 2012. url: http://www.soamig.de/ , (05.10.2012).

[26] Richard Wettel. CodeCity Homepage, 2010. url: http://www.inf.usi.ch/phd/wettel/codecity.html , (03.10.2012).

[27] Richard Wettel and Michele Lanza. Visualizing Software Systems as Cities. In *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on*, pages 92–99, 2007.

[28] Richard Wettel and Michele Lanza. Codecity: 3d visualization of large-scale software. In *In ICSE Companion '08: Companion of the 30th ACM/IEEE International Conference on Software Engineering*, pages 921–922. ACM, 2008.

[29] Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: a controlled experiment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 551–560, New York, NY, USA, 2011. ACM.

[30] Andreas Winter, Manfred Kamp, Angelika Franzke, Jürgen Ebert, and Peter Dahm. TGraphen und EER-Schemata – Formale Grundlagen. In Andreas Winter, H. Stasch, Rainer Gimnich, and Jürgen Ebert, editors, *GUPRO — Generische Umgebung zum Programmverstehen*, pages 51–66. Föllbach, 1998.

[31] Andreas Winter, Hans H. Stasch, Rainer Gimnich, and Jürgen Ebert, editors. *GUPRO - Generische Umgebung zum Programmverstehen.* Föllbach, 1998.

Versicherung

Hiermit versichere ich, dass ich diese Arbeit selbsständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt habe.

Oldenburg, den 22. Oktober 2012 _____

Marie-Christin Ostendorp