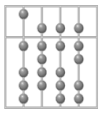
TECHNISCHE UNIVERSITÄT MÜNCHEN



# Diplomarbeit in Informatik

Technische Universität München

Fakultät für Informatik

Architektur eines Visualisierungswerkzeugs

für Anwendungslandschaften -

Anforderungsanalyse und Realisierung von Kernkomponenten

| | |
|---|---|
| Bearbeiter: | Christian M. Schweda |
| Aufgabensteller: | Prof. Dr. Florian Matthes |
| Betreuer: | André Wittenburg |
| Abgabedatum: | 12. Juli 2006 |

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den ..... .....

**Danksagung**

Ich möchte Herrn Professor Dr. Florian Matthes danken, nicht nur dafür, dass er Aufgabensteller dieser Arbeit war, sondern auch dass er in den Jahren meiner Tätigkeit als Hilfskraft an seinem Lehrstuhl mehr war, als ein ≫Weisungsbefugter≪ - ein kritischer Diskussionspartner in Belangen der Informatik allgemein und der *Softwarekartographie* im Speziellen.
Ich möchte mich auch bei den Mitgliedern des ≫SoCaTeams≪ bedanken:

- André, nicht nur für seine Betreuung dieser Arbeit, sondern auch dafür, dass er mir ermöglicht hat schon als Student Erfahrungen in der Forschung und am Forschungsgegenstand *Softwarekartographie* zu sammeln. Und dafür, dass er meine Ideen stets geschätzt und geprüft hat, auch gerade dann, wenn sie nicht rein pragmatischer Natur waren.

- Josef, für die gute und erfolgreiche Zusammenarbeit in all den Jahren, aber besonders bei der Formalisierung des Visualisierungsmodells, die ohne seine Ausdauer und Hartnäckigkeit wohl nicht zu Stande gekommen wäre. Und dafür, dass er nie einem Meinungsaustausch abgeneigt war, obwohl oder gerade weil er und ich nicht immer einer Meinung gewesen sind.

- Alex, nicht allein für seine Mitwirkung an der Betreuung des praktischen Teils dieser Arbeit, sondern auch dafür, dass er als geduldiger Praktiker stets bereit war theoretische ≫Umwege≪ in Kauf zu nehmen und sich ebenso zuverlässig wie konsequent um deren Umsetzung in der Praxis bemüht hat. Und dafür, dass er dabei immer ruhig geblieben ist.

Mein ganz besonderer Dank gilt den Personen im meinem privaten Umfeld, die mich in all den Jahren unterstützt haben, speziell meinen Eltern und meiner Partnerin für ihren Zuspruch, ihre Unterstützung und ihr Vertrauen in mich. Danke.

**Zusammenfassung**
Betriebliche Anwendungen nehmen eine wichtige Rolle in den Unternehmen von heute ein und sind in vielen Fällen essentiell für die betriebliche Wertschöpfung. Gleichzeitig bilden sie einen bedeutenden Kostenfaktor und sind darüber hinaus variablen Anforderungen von fachlicher Seite ausgesetzt. Deswegen muss das Management der Anwendungslandschaft, also der Gesamtheit der betrieblichen Anwendungen, in modernen Unternehmen als wesentlich angesehen werden.

Eine wichtige Grundlage für das Management der Anwendungslandschaft bildet die Dokumentation derselben, welche - wie im Forschungsprojekt Softwarekartographie am Lehrstuhl Software Engineering betrieblicher Informationssysteme (sebis) an der TU München gezeigt - häufig von Visualisierungen der Anwendungslandschaft oder ausgewählter Aspekte dieser Gebrauch macht. Diese Visualisierungen, Softwarekarten genannt, werden in den Unternehmen in der Regel mit nur rudimentärer Werkzeugunterstützung manuell erstellt, statt aus den gesammelten Informationen über die Anwendungslandschaft generiert zu werden. Diese Form der ≫losen Kopplung≪ zwischen den Informationen zum einen und ihrer Visualisierung zum anderen führt dabei zu verschiedensten Schwierigkeiten.

Um diese Schwierigkeiten zu beseitigen und Generierung von Softwarekarten aus den Informationen über die Anwendungslandschaft zu automatisieren, wurde am Lehrstuhl sebis ein Verfahren entwickelt, das unter Nutzung der Modelltransformation Visualisierungen automatisch aus Daten erzeugen kann. Die prinzipielle Realisierbarkeit dieses Verfahrens in einem Werkzeug ist an eben diesem Lehrstuhl durch eine prototypische Implementierung in einer Projektarbeit gezeigt worden.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Kapitel 1

# Einleitung und Motivation

Anwendungslandschaften, bestehend aus hunderten oder tausenden von betrieblichen Anwendungen, sind aus den Unternehmen gegenwärtig nicht mehr wegzudenken. Operative und unterstützende Prozesse werden heutzutage in der Mehrzahl der Unternehmen dabei nicht nur durch ein einzelnes sondern durch eine Vielzahl von Anwendungssystemen unterstützt oder erst ermöglicht. Dies bedingt sich im Besonderen durch die wachsende Bedeutung der Resource ≫Information≪, welche heute gemäß Gernert et. al. (vgl. [GA02]) mehr denn je zuvor als kritischer Faktor für den Unternehmenserfolg verstanden werden muss. Ein sorgfältiger Umgang mit dieser Resource, aber auch mit den Systemen der betrieblichen Informationsverarbeitung, darunter den betrieblichen Anwendungen, ist ergo unabdingbar als Maßnahme unternehmerischer Existenzsicherung.

Die Dokumentation von Anwendungslandschaften, wesentlicher Bestandteil im Management derselben, ist ein wichtiges Forschungsthema des Forschungsprojekts Softwarekartographie am Lehrstuhl für Software Engineering betrieblicher Informationssysteme an der Technischen Universität München. Im Rahmen dieses Forschungsprojekts, dem auch die vorliegende Arbeit zugerechnet werden kann, wurden in der industriellen Praxis bei renomierten Projektpartnern, wie z.B. der BMW Group oder Siemens, bestehende Verfahren zur Dokumentation von Anwendungslandschaften untersucht. Im Mittelpunkt standen dabei die verwendeten Visualisierungen (≫Softwarekarten≪ genannt), die trotz der Vielzahl der dargestellten Aspekte und der verwendeten Symbole, von Lankes et. al in [LMW05b] auf eine überschaubare Gruppe von vier Kartentypen konsolidiert werden konnten.

Auffällig ist, dass diese Visualisierungen vornehmlich von Hand und nur mit rudimentärer Werkzeugunterstützung erstellt werden. Die Werkzeugunterstützung beschränkt sich dabei in der Regel auf Repositorys, in welchen die Informationen über die Anwendungslandschaft gespeichert werden. Die in den Repositorys gespeicherten Objekte können dann auf die Visualisierung ≫gezogen≪ werden, wodurch ein Symbol entsteht. Die Beziehungen zwischen den Objekten können von diesen Werkzeugen meist nicht anhand graphischer Konzepte, wie z.B. die Schachtelung von Symbolen, umgewandelt werden. Deswegen erfolgt diese Übersetzung meist manuell durch Positionierung der Symbole in einem zeitaufwändigen

und fehleranfälligen Prozess. Darüber hinaus können Positionierungsinformationen von den meisten Werkzeugen nicht in einem umgekehrten Vorgang zur Modellierung von Beziehungen zwischen Objekten herangezogen werden.

Als Konsequenz dieser Tatsache wurden im Forschungsprojekt Softwarekartographie Methoden zur Automatisierung der Generierung von derartigen Darstellungen untersucht und erforscht. Als erfolgversprechendster Ansatz wurde dabei die Nutzung von Modelltransformationen entdeckt, welche von Lankes et. al. in [LMW05a] vorgestellt wurde. Zentraler Baustein dieses Ansatzes ist die Existenz zweier, eng gekoppelter objektorientierter Modelle. Zum einen ein Modell mit den Daten über die Anwendungslandschaft, wie es von vielen Projektpartnern des Forschungsprojekts bereits gepflegt wird. Zum anderen ein Modell, welches die Visualisierungen objektorientiert repräsentieren kann, d.h. über Klassen für Symbole, wie z.B. ein Rechteck, und Regeln, wie z.B. die Schachtelung verfügt. Die Entwicklung des zweitgenannten Modells, auch Visualisierungsmodell genannt, findet parallel zu der vorliegenden Arbeit am Lehrstuhl für Software Engineering betrieblicher Informationssysteme statt (siehe Ernst et. al. in [Er06]).

Aufbauend auf einen ersten Entwurf dieses Visualisierungsmodells wurde von Schweda in [Sc05a] ein prototypisches Werkzeug zur automatisierten Generierung von Visualisierungen realisiert und dadurch die Anwendbarkeit des Modells zumindest in Teilen validiert. Auf diese Vorarbeit setzt die vorliegende Arbeit auf, in welcher das Verfahren nach Lankes et. al aus [LMW05a] in den Rahmen einer erweiterbaren und flexiblen Softwarearchitektur eingebettet werden soll. Diese Architektur soll dann im praktischen Teil der Arbeit in Teilen realisiert werden.

## 1.1 Aufgabenstellung

Die Aufgabenstellung der Arbeit liegt in der Entwicklung einer komponentenbasierten Softwarearchitektur für ein Werkzeug, welches die automatisierten Generierung von Softwarekarten auf Basis der Konzepte eines bereits entwickelten Visualisierungsmodells, der Informationsmodellierung und der Modelltransformation unterstützt. Dazu sollen im Vorfeld Anwendungsfälle, die ein solches Werkzeug implementieren soll, und die zugehörigen Akteure erhoben, sowie deren funktionale und nicht-funktionale Anforderungen dokumentiert werden. Ausgehend von diesen Anforderungen soll aufbauend auf die in bereits abgeschlossenen Arbeiten im selben Umfeld gewonnenen Erkenntnisse, eine plattformunabhängige Komponentenarchitektur entworfen werden.

In dieser plattformunabhängigen Architektur werden für den praktischen Teil der vorliegenden Arbeit Kernkomponenten identifiziert und in einem weiteren Schritt in eine plattformspezifische Architektur überführt, welche den Gegebenheiten und Anforderungen der *Eclipse Rich Client Platform* angepasst ist. Diese plattformspezifische Komponentenarchitektur soll dann unter Berücksichtigung von Aspekten der Flexibilität und Erweiterbarkeit konkret implementiert werden. Das in diesem Schritt zu entwickelnde Werkzeug soll dann hinsichtlich seiner Architektur mit der im ersten Teil der Arbeit aufgestellten plattformu-

nabhängigen Architektur in Beziehung gesetzt werden. Auftretende Abweichungen sollen erklärt und in die Dokumentation der plattformspezifischen Architektur eingearbeitet werden. In einem abschließenden Abschnitt sollen Erkenntnisse aus der Design- und Entwicklungsphase konsolidiert und gegebenenfalls als Ansätze für weitergehende Forschungen und Entwicklungen formuliert werden.

## 1.2 Gliederung

Die vorliegende Arbeit folgt in ihrer Gliederung dem ≫klassischen≪ Softwareentwicklungsprozess, bestehend aus den Phasen *Analyse*, *Design* und *Implementierung*, wenngleich Kapitel- und Phasenübergänge nicht immer zusammenfallen.

Kapitel 2 gibt einen detaillierten Einblick in die Problemdomäne, in welche das in dieser Arbeit zu entwickelnde Werkzeug eingeordnet werden kann, und zeigt die momentan in diesem Umfeld verwendeten Verfahren sowie deren Nachteile auf. Auch werden Termini der Domäne entsprechend eingeführt und definiert.

Kapitel 3 identifiziert Rollen von Nutzern und führt Anwendungsfälle ein, aus welchen funktionale Anforderungen an das Werkzeug abgeleitet werden. Daran schließt sich in eine konkrete Beschreibung der graphischen Benutzeroberfläche des Werkzeugs inklusive der Angabe von potentiellen Benutzerinteraktionen an.

Kapitel 4 führt Termini und Konzepte der Lösungsdomäne ein und illustriert das dem Werkzeug zugrunde liegende Verfahren anhand dieser Begriffe.

Kapitel 5 zeigt ein plattformunabhängiges komponentenbasiertes Design, welches den im vorhergehenden Kapitel dargestellten Lösungsansatz implementiert und stellt die diesem Design zugrunde liegenden Architekturentscheidungen im Gesamtkontext dar.

Kapitel 6 führt die Zielplattform für die Implementierung anhand ihrer allgemeingültigen Design- und Aufbauprinzipien ein und zeigt Anpassungen des Designs an plattformspezifische Gegebenheiten, sowie konkrete Implementierungsaspekte der Kernkomponenten auf.

Kapitel 7 konsolidiert die Einsichten aus der Arbeit, zeigt Aspekte auf, die nicht adressiert werden konnten, und gibt Perspektiven für die weitere Erforschung sowie die Weiterentwicklung des Werkzeugs.

# Kapitel 2

# Management von Anwendungslandschaften

Betriebliche Informationssysteme sind aus Unternehmen heutzutage nicht mehr wegzudenken; sie unterstützen administrative Prozesse, wie z.B. die Rechnungslegung, oder ermöglichen die Durchführung operativer Prozesse, wie z.B. die Produktentwicklung. Sie wirken also an der Planung, Steuerung und Kontrolle der betrieblichen Prozesse, ebenso wie an der Interaktion des Unternehmens mit seiner Umwelt mit. Im Profil der Wirtschaftsinformatik [WK94] werden *betriebliche Informationssysteme* dabei als soziotechnische Systeme verstanden, die »menschliche« und »maschinelle« Komponenten umfassen. Dabei können die »maschinellen« Anteile noch weiter zerlegt werden, zum einen in Infrastrukturkomponenten, wie z.B. Hardware, Betriebssysteme oder Middleware, zum anderen in *betriebliche Anwendungssysteme*, die fachliche Daten-verarbeitenden und Geschäftsprozess-unterstützenden Softwarekomponenten. Die meisten betriebliche Anwendungen weisen dabei mindestens eine der folgenden Eigenschaften auf:

Betriebliches Informationssystem

Betriebliche Anwendungssysteme

**Langlebigkeit:** Anwendungen bleiben länger als einen Produktzyklus im Unternehmen im Einsatz.

**Investitionsgutcharakter:** Anwendungen sind teuer in Anschaffung oder Entwicklung und generieren bei der Ein- und Anbindung an bestehende Systeme weitere Kosten.

Im Kontext des betrieblichen Informationssystems, besonders im Hinblick auf die beteiligten Infrastrukturkomponenten sowie seine »menschliche« Dimension, kommen weitere Eigenschaften hinzu, wie z.B.:

**Betriebs- & Wartungsbedarf:** Auch die zur Anwendung gehörenden Infrastrukturkomponenten müssen bereitgestellt und gewartet werden.

**Schulungsbedarf:** Die Nutzer der Anwendung müssen mit ihrer Verwendung vertraut gemacht werden.

Doch nicht nur in der umfassenderen Betrachtung der betrieblichen Anwendung als Teil eines betrieblichen Informationssystems offenbart sich eine weitere Komplexitätsdimension des Beobachtungsgegenstandes, sondern auch beim Übergang von der isolierten Anwendung zur Gesamtheit der betrieblichen Anwendungen eines Unternehmens und ihrer Verbindungen untereinander, *Anwendungslandschaft* genannt. Die Bedeutung der Anwendungslandschaft als Managementgegenstand soll im folgenden Abschnitt beleuchtet werden.

Anwendungslandschaft

## 2.1 Managementgegenstand und -aufgaben

Die Anwendungslandschaft, die Gesamtheit der betrieblichen Anwendungssysteme und deren Verbindungen, bildet zusammen mit der IT-Infrastruktur und organisatiorischen (»menschlichen«) Komponenten die *Informationssysteminfrastruktur* eines Unternehmens. Um diese Begriffe noch deutlicher gegeneinander abzugrenzen, sei auf die vom Lehrstuhl für Software Engineering betrieblicher Informationssysteme in [se05] vorgestellte Gliederung für die *Enterprise Architecture (EA)* verwiesen (siehe Abbildung 2.1).

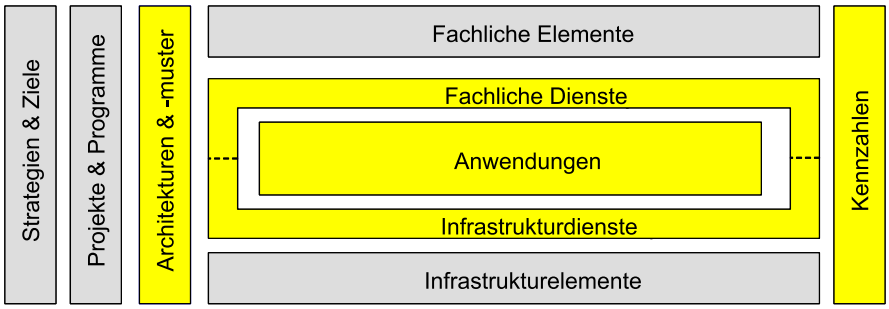



Abbildung 2.1: Einordnung des Managementgegenstandes *Anwendungslandschaft* in eine Gliederung einer Enterprise Architecture

Die in der Abbildung hervorgehobenen Bereiche sind dabei als Teil oder unmittelbar benachbart mit der Anwendungslandschaft zu sehen und sind daher am Managementgegenstand des Managements von Anwendungslandschaften beteiligt. Die Informationssysteminfrastruktur umfasst neben den markierten Bereichen noch weitere Aspekte, wie z.B. eine organisatorische, sowie fachliche Dimension und konkrete Infrastrukturkonzepte. Die Anwendungslandschaft selbst umfasst nur betriebliche Anwendungssysteme und deren Beziehungen untereinander, wobei konkrete Anwendungslandschaften nicht selten mehr als tausend Anwendungssysteme und zehntausend Beziehungen enthalten und nicht immer als das Ergebnis eines durchgängigen Managementprozesses, sondern vielmehr als ein durch viele und nur zum Teil koordinierte Mikromanagementprozesse »gewachsener« Zustand verstanden werden müssen. Nichtsdestotrotz oder gerade aufgrund ihrer inhärenten Komplexität tragen Anwendungslandschaften wesentlich zum »Kostenfaktor IT« bei, dessen

Management in den letzten Jahren an Bedeutung gewonnen hat. Eine typische Aufgabe des Managements von Anwendungslandschaften, wie sie der Lehrstuhl für Informatik 19 der Technischen Universität München in [se05] dargestellt hat, ist z.B. die Ablösung eines unternehmensweit eingesetzten Datenbankmanagementsystems. In diesem Zusammenhang müssen eventuell betroffene Produktivsysteme und unter Umständen deren Beitrag zur Unterstützung von Geschäftsprozessschritten ermittelt werden, so dass diese Gegebenheiten bei der Ablösung geeignete Berücksichtigung finden. Diese Aufgabe ist ohne die notwendigen Informationen über die bestehende Anwendungslandschaft, ihre betrieblichen Anwendungen und deren Abhängigkeiten von Infrastrukturkomponenten, wie z.B. Datenbankmanagementsystemen, nicht zu bewerkstelligen. Beispiele für weitere Aufgaben des Managements von Anwendungslandschaften, speziell motiviert durch Forderungen nach kosteneffizienterer Leistungserbringung in der IT sind nach [se05]:

**Konsolidierung von Infrastrukturkomponenten:** Verringerung der Anzahl verschiedenartiger, im Einsatz befindlicher Infrastruktursoftware, wie z.B. Datenbankmanagementsysteme, Betriebssysteme oder Webbrowser.

**Konsolidierung der Geschäftsprozessunterstützung:** Verringerung der Anzahl der Anwendungssysteme, die an verschiedenen Standorten denselben Geschäftsprozess oder Geschäftsprozessschritt unterstützen.

Derartige Aufgaben im Speziellen und damit das Management der Anwendungslandschaft im Gesamten gewinnen an Bedeutung, zum einen im Hinblick auf den steigenden Kostendruck, zum anderen auch in Hinblick auf die wachsende Zahl von Treibern für Veränderungen an der Anwendungslandschaft. Treibende Kräfte sind dabei einerseits technische Erfordernisse oder Neuerungen, wie die Nutzung von neuen Technologien, z.B. den Webservices, andererseits auch die Erfordernisse des Geschäfts und der fachlichen Abteilungen im Sinne einer konsequenten Aneinanderausrichtung von Geschäft und IT (siehe [Ad97]). Die Erfüllung dieser Erfordernisse bei gleichzeitiger Kostenreduktion ist nur durch den Übergang von einer »Spezialitätenschmiede«, die maßgeschneiderte Lösungen für geschäftliche Anforderungen liefert, hin zu einer »Software-Fabrik« möglich (siehe [RR05]), welche standardisierte, aber zugleich flexibel kombinierbare IT-Dienstleistungen anbietet. Diese Dienstleistungen sind dabei unabhängig von den diensterbringenden Anwendungssystemen zu sehen, so dass dem Management von Anwendungslandschaften eine Mittlerrolle zufällt, welche unter Berücksichtigung von Effektivitäts- und Effizienzsaspekten die Erbringung standardisierter Dienste durch bestehende und anzuschaffende Systeme sicherstellen muss. Eine derartige Aufgabe ist dabei allerdings von profunden Kenntnissen des zu verwaltenden Systems und damit von einer Dokumentation des Aufbaus desselben abhängig.

## 2.2 Dokumentation der Anwendungslandschaft

Die Anwendungslandschaft als zu verwaltender Gegenstand ist in ein Managementsystem eingebunden, zu dem unter anderem auch die Interessenten, nachfolgend *Stakeholder* ge-

nannt, gehören. Als Interessent der Anwendungslandschaft wird eine Person, Gruppe oder Rolle im Unternehmen bezeichnet, welche Interessen, nachfolgend *Concerns* genannt, an der Anwendungslandschaft zum Ausdruck bringt. Diese Definitionen stimmen mit den im IEEE Standard 1471-2000, *Recommended Practice for Architectural Description of Software-Intensive Systems* [IE00], verwendeten überein, anhand dessen weitere Begriffe eingeführt werden sollen. Dabei wird die Anwendungslandschaft als *softwareintensives System* verstanden, also als System, welches bei der Erfüllung seines Systemzwecks von der Nutzung von Software abhängig ist. Die Anwendbarkeit des Standards IEEE 1471 auf die Anwendungslandschaft und deren Beschreibung haben Lankes et al. in [LMW05a] illustriert. Anhand von Abbildung 2.2 soll der Querbezug zu bereits verwendeten Begriffen des Standards dargestellt werden: IEEE 1471

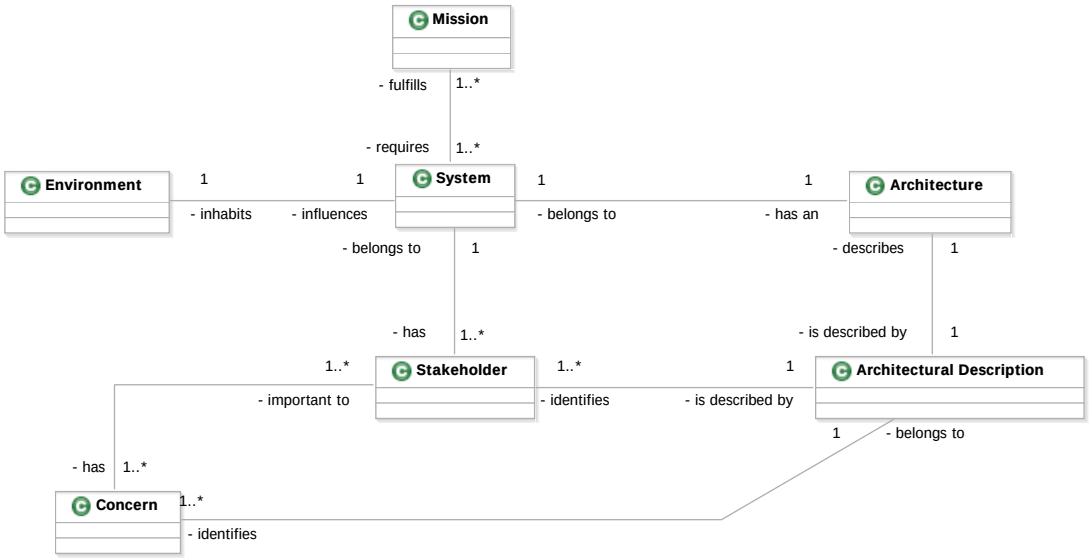



Abbildung 2.2: Ausschnitt aus dem konzeptuellen Modell des IEEE 1471

**System:** Die Anwendungslandschaft selbst, als System von Anwendungssystemen.

**Mission:** Der Zweck der Anwendungslandschaft, also die Unterstützung der einzelnen Geschäftsprozesse, sowie der Unternehmensziele und -strategien.

**Environment:** Das Umfeld der Anwendungslandschaft - Unternehmensstrukturen, wie z.B. Organisationseinheiten, oder das Umfeld des Unternehmens an sich, welche nicht in der Anwendungslandschaft modelliert werden; umfasst unter anderem die Informationssysteminfrastruktur (siehe Abschnitt 2.1, speziell Abbildung 2.1).

**Architecture:** Die grundlegende Organisation der Anwendungslandschaft, gebildet aus ihren Anwendungen, deren Beziehungen und den Prinzipien, die zu diesem Aufbau geführt haben.

**Stakeholder:** Die Interessenten der Anwendungslandschaft, also Personen, Gruppen, Rollen und Organisationseinheiten im Unternehmen mit Interessen und Anforderungen an die Anwendungslandschaft.

**Concern:** Die Interessen an der Anwendungslandschaft, also gewünschte Eigenschaften z.B. im Hinblick auf Betrieb, Entwicklung, Kosten oder Wartung.

Zentrales Element des Standards bildet die *Architectural Description*, also die Beschreibung der Architektur eines softwareintensiven Systems. Eine derartige Beschreibung enthält die oben geforderten Informationen über den Managementgegenstand Anwendungslandschaft. In Anbetracht der Komplexität und des Umfangs der Anwendungslandschaft kann man sich diese Architekturbeschreibung dabei als eine Sammlung von Informationen über die Anwendungslandschaft vorstellen, welche speziell auf die Stakeholder und deren Concerns zugeschnitten sein sollte. Die Bildung einer umfassenden und für die verschiedenen Kombinationen von Stakeholdern verwendbaren Informationssammlung stellt dabei einen aufwendigen Prozess dar, dessen Ergebnis sich je nach Unternehmensstruktur und angewandter IT-Aufbauorganisation (*zentral* kontra *dezentral* vgl Buchta et. al. in [BESC04]) deutlich unterscheiden kann. Lankes et al. etablieren in [LMW05b] in diesem Kontext für die Informationssammlung selbst, wie auch für deren Aufbauprinzipien den Begriff des *Modells*[1], welcher im folgenden Abschnitt definiert und angewendet werden soll.

## 2.3 Architekturbeschreibung als Modell

Ein *Modell* stellt die *Abbildung* eines natürlichen oder künstlichen Originals dar, wobei im allgemeinen nicht alle Attribute des Originals erfasst werden, sondern eine *Verkürzung* auf Attribute erfolgt, welche aus Sicht der Modellerschaffer für die Modellnutzer relevant erscheinen. Konsequenterweise sind Modelle ihren Originalen nicht eindeutig zugeordnet, sondern stellen für bestimmte modellbenutzende Subjekte zu fixen Zeitpunkten oder -intervallen eine Ersetzungsfunktion bereit, welche die konsistente Anwendung einer definierten Menge von Operationen gewährleistet (*Pragmatik*) [St73]. Modell

Für die Beschreibung der Architektur von Anwendungslandschaften bilden gewisse Aspekte, so dargestellt von Matthes et al. in [MW04], derselben, wie z.B. die darin enthaltenen Anwendungssysteme mit ihren Nutzerzahlen, die Grundlage. Die Gesamtheit dieser Aspekte wird dabei durch ein sogenanntes *semantisches Modell* abgedeckt. Die Bildung eines derartigen Modells erfolgt unter Verwendung spezifischer Modellierungstechniken und damit verbundener -konzepte, kann also als Instanziierung eines, durch das verwendete Modellierungsparadigma festgelegten *Metamodells* verstanden werden. Im Forschungsprojekt Softwarekartographie wurde das Paradigma der *objektorientierten Modellierung* als in der semantisches Modell

Metamodell

[1] An dieser Stelle sei auf die abweichende Verwendung des Begriffs *Model* im Standard IEEE 1471 hingewiesen, welche in Abschnitt 2.4 detaillierter diskutiert wird.

Praxis am häufigsten angewendetes Verfahren identifiziert, was nicht zuletzt durch die gute Werkzeugunterstützung begründet sein dürfte. In den Arbeiten von Halbhuber [Ha04], Brendebach [Br05] und Lauschke [La05] wurde dieses Paradigma der Modellierung in der Praxis unternehmerischer Architekturbeschreibung erfolgreich eingesetzt. Die Konzepte, aus den in diesen Arbeiten entwickelten Modellen, wurden von Buckl in [Bu05] unter anderem hinsichtlich ihrer semantischen Mächtigkeit und möglicher Austauschbarkeit miteinander verglichen und mit bestehenden Realisierungen für objektorientierte Modellierungssprachen und -werkzeuge in Bezug gesetzt.

Durch die Konzepte der objektorientierten Modellierung ausgedrückt, wird das semantische Modell als eine Menge von *Informationsobjekten* und deren *Beziehungen* untereinander verstanden. Diese sollten zu entsprechenden *Informationsklassen* und *Assoziationen* aus einem korrespondieren Metamodell der Informationsmodellierung konform sein. Dieses Metamodell wird nachfolgend *Informationsmodell* genannt. Informationsmodelle stellen also den Begriffsapparat (gewissermaßen die Vokabeln) zur Verfügung unter dessen Verwendung Architekturbeschreibungen für Anwendungslandschaften erstellt werden können und unterscheiden sich von Unternehmen zu Unternehmen stark. Halbhuber [Ha04], Brendebach [Br05] und Lauschke [La05] stellen in ihren Arbeiten objektorientierte Informationsmodelle, wie sie in der unternehmerischen Praxis gewonnen werden können, dar, die sich allerdings deutlich voneinander unterscheiden und aufgrund unterschiedlicher Abstraktionsniveaus, aber auch der ihnen zu Grunde liegenden Concerns als schwer vereinbar betrachtet werden müssen. Nichtsdestotrotz ist die Pflege der in ihnen enthaltenen Daten für das Unternehmen im Hinblick auf eine konsistente Dokumentation von großer Bedeutung. Aufgrund der Komplexität und des Umfangs einer derartigen Beschreibung ergibt sich die Notwendigkeit von geeigneten Verfahren zur Aufbereitung und Präsentation der Informationen. Ein solches Verfahren soll im folgenden Abschnitt vorgestellt werden.

Informationsmodell

## 2.4 Softwarekarten als Teil von Architekturbeschreibungen

Die Aufbereitung der Informationen aus einer Architekturbeschreibung für die Interessenten des dokumentierten Systems wird ebenfalls durch den Standard IEEE 1471 adressiert. Dazu werden die Konzepte *Viewpoint* und *View* eingeführt. Ein Viewpoint stellt einen speziellen »Blickwinkel« auf die Architekturbeschreibung dar und enthält Richtlinien für die Aufbereitung der Informationen in einen View (»Sicht«), sowie Verfahren für die Analyse der darin dargestellten Informationen. Ein View selbst wiederum enthält eine oder mehrere konkrete Aufbereitungen der Architekturdokumentation, im Standard als *Models* bezeichnet. Ein Viewpoint also kann als Verfahren für die Aufbereitung von Informationen verstanden werden, ein View als dessen konkrete Instanziierung im Kontext einer speziellen Architekturbeschreibung. Gesichtspunkte, welche bei der Bildung von Aufbereitungen beachtet werden sollen, werden dabei z.T. im Standard expliziert und können zu nachfolgender Liste ergänzt werden:

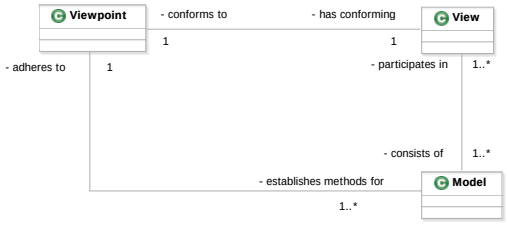



Abbildung 2.3: Beziehung Viewpoint-View-Model gemäß IEEE 1471

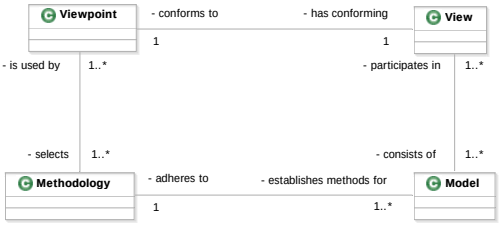



Abbildung 2.4: Beziehung Viewpoint-View-Methodology-Model verändert

**Filterung:** Ein Concern wird gemäß des Standards[2] durch einen Viewpoint auf die Architekturbeschreibung adressiert.

**Erfassbarkeit:** Ein Viewpoint sollte *Methodiken*[3] nutzen, welche die relevanten Aspekte leicht und schnell kenntlich machen und Vorschläge für mögliche Auswertungen machen können.

**Kommunizierbarkeit:** Ein View sollte für sich weitergebbar sein und dabei verständlich bleiben[4].

Der Begriff der Methodik wird im Rahmen des Standards nur implizit eingeführt und führt, wie u.A. von Schweda in [Sc05b] dargestellt, zu missverständlichen Formulierungen. Deswegen wird für Betrachtungen hinsichtlich der Definition von Methodiken für die Aufbereitung von IEEE 1471 Models nachfolgend das originale konzeptuelle Modell (siehe Abbildung 2.3) des Standards wie in Abbildung 2.4 dargestellt verändert.

Softwarekarte

Schichtenprinzip

Matthes et al. haben in [MW04] Verfahren im Kontext der Dokumentation von Anwendungslandschaften in der industriellen Praxis untersucht und die *Softwarekarte* (in logischer Kontinuität der Landschafts-Metapher) wie folgt definiert: »eine *Softwarekarte* ist eine graphische Repräsentation der Anwendungslandschaft oder von Ausschnitten selbiger«. Dabei ließ sich der grundlegende Aufbau einer Softwarekarte auf den Aufbau des entsprechenden topographischen Analogons zurückführen. Wie eine Landkarte folgt auch die Softwarekarte dem *Schichtenprinzip*, d.h. sie gruppiert gedanklich zusammengehörige Elemente zu einer *Schicht* (siehe Abbildung 2.5) genannten Gruppierung, die als ganzes sichtbar oder unsichtbar sein kann. Besondere Bedeutung kommt dabei der untersten Schicht einer Softwarekarte zu, dem sogenannten *Kartengrund*. Während bei der Erstellung einer Landkarte, prominente topographische Dimensionen (Aspekte), wie Längen-

[2]Der Standard etabliert hier die Kardinalitäten der Beziehung, stellt allerdings nicht explizit die Forderung nach einer Filterung auf.

[3]Der Begriff der *Methodik* kommt nur implizit in einem Assoziationsnamen vor.

[4]Der Standard fordert in diesem Zusammenhang nur minimale Dokumentationen zu einem View, schreibt aber keine konkreten Bestandteile einer kommunizierbaren Form vor.

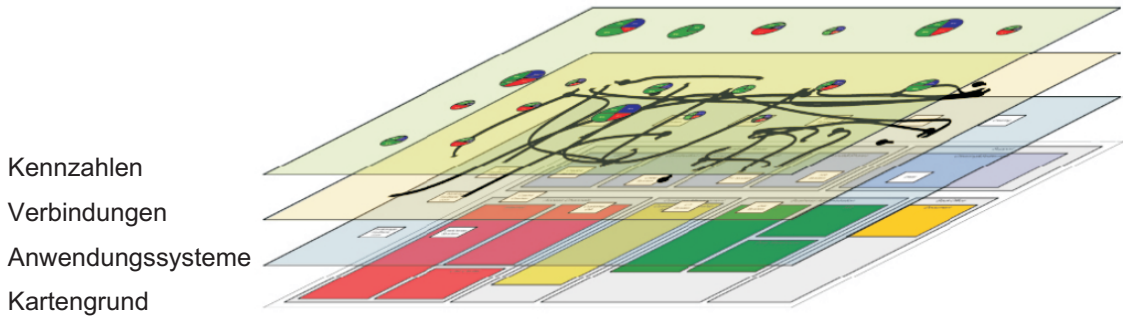



Abbildung 2.5: Darstellung des Schichtenprinzips (entnommen aus [se05])

und Breitengrad existieren, welche die Anordnung der Objekte auf dem Kartengrund bestimmen, fehlen bei Softwarekarten derartig ausgezeichnete Aspekte. Dennoch, so konnten Lankes et. al. in [LMW05b] eine Kategorisierung für Softwarekarten anhand der Aspekte auf dem Kartengrund entwickeln. Die dabei klassifizierten *Softwarekartentypen* sind:

Softwarekartentypen

- Clusterkarte (siehe Abbildung 2.6)
- Prozessunterstützungskarte (siehe Abbildung 2.7)
- Intervallkarte (siehe Abbildung 2.8)
- Karte ohne Kartengrund zur Verortung (Ad hoc-Karte)

Softwarekarten (siehe Abbildungen 2.6, 2.7 und 2.8) zeigen in der Praxis oft bis zu einige hundert betriebliche Anwendungssystem, Organisationseinheiten und Standorte, sowie deren Beziehungen untereinander - stellen dabei den (geplanten) Zustand zu einem festen Zeitpunkt in der Vergangenheit, Gegenwart oder Zukunft dar. Abhängig vom adressierten Concern werden auch weitere Aspekte von Informationssystemen, z.B. der Nutzungsintensität, Transaktionsraten oder Antwortzeiten durch eine Farbkodierung oder die Anbringung zusätzlicher Symbole, wie z.B. Ampeln dargestellt (vergleiche Beyer in [Be04]). Dabei spielen im Hinblick auf die oben geäußerten Anforderungen *Erfassbarkeit* und *Kommunizierbarkeit* Prinzipien zur Reduktion von Komplexität, z.B. das Schichtenprinzip, sowie die konsequente und durchgängige Verwendung von Symbolen und Farben eine Rolle. Zusätzlich kann die Erhaltung gewisser Layoutschemata die Wahrnehmbarkeit und den *Wiedererkennungseffekt* einer Softwarekarte erhöhen, wie dies in der topographischen Kartographie z.B. durch die immer gleiche Ausrichtung der Karte (Norden entspricht oben) erzielt wird.

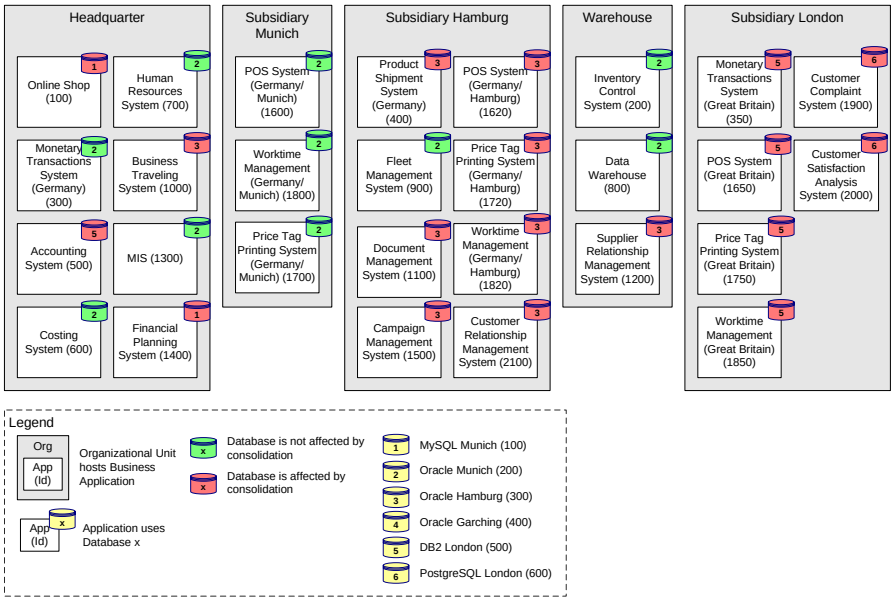



Abbildung 2.6: Softwarekarte: Clusterkarte (entnommen aus [se05])

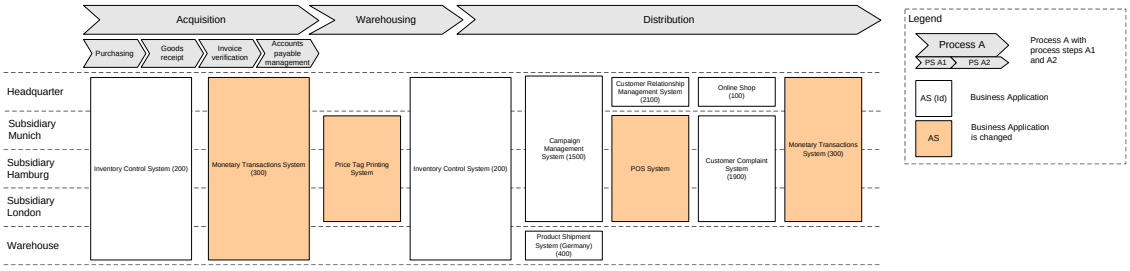



Abbildung 2.7: Softwarekarte: Prozessunterstützungskarte (entnommen aus [se05])

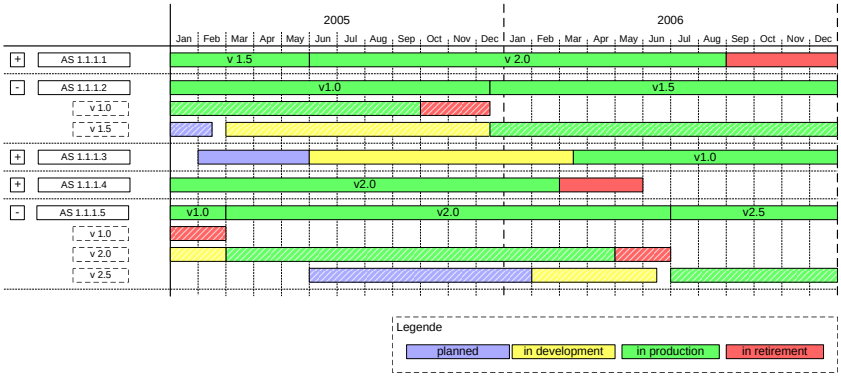



Abbildung 2.8: Softwarekarte: Intervallkarte (entnommen aus [se05])

## 2.5 Erstellung von Softwarekarten

Im Widerspruch zu den oben geäußerten Anforderungen an und Prinzipien für den Aufbau von Softwarekarten steht in der industriellen Praxis ihre Erstellung. Zwar werden in den Unternehmen die relevanten Informationen zumindest zum Teil werkzeuggestützt gesammelt, die Werkzeugunterstützung bei der Erstellung der Karten ist allerdings, wie vom Lehrstuhl für Software Engineering betrieblicher Informationssysteme in [se05] beschrieben, nicht durchgängig, die Karten müssen zumeist *per Hand* oder *semi-automatisch* erstellt werden. *Per Hand* bezeichnet dabei ein Verfahren, bei dem die symbolischen Repräsentationen von Informationsobjekten, wie z.B. betriebliche Anwendungssysteme, manuell auf die Zeichenfläche verbracht und positioniert werden müssen. Im Gegensatz dazu stellen *semi-automatische* Verfahren beschränkte Layoutingmechanismen, wie z.B. Graphlayout nach dem Spring- oder Sugiyama-Verfahren zur Verfügung. Bei beiden Verfahren ist ein hohes Maß an Benutzerinteraktion erforderlich, um praxisrelevante Karten wie die in den Abbildungen 2.6, 2.7 und 2.8 zu erstellen.

Der damit verbundene Aufwand mag sich besonders bei der Planung von künftigen Anwendungslandschaften oder der damit verbundenen Bildung von Szenarien für die Planlandschaft hinderlich erweisen, da die Szenarien unverbunden erstellt und einzeln visualisiert werden müssen. Vom Lehrstuhl für Software Engineering betrieblicher Informationssysteme wurden in [se05] Werkzeuge untersucht, die als Teil ihrer Unterstützung des *Enterprise Architecture Managements* verschiedene Formen der Visualisierung anbieten. Eine durchgängige Unterstützung von Softwarekarten, sowie deren *automatische* und *regelbasierte* Generierung war in keinem Werkzeug gegeben. Dies mag auf Grund der Vielfältigkeit der Visualisierungen und der ihnen zu Grunde liegenden Daten einerseits verständlich erscheinen, ist jedoch im Hinblick auf den Einsatz von Softwarekarten als Werkzeug im Management von Anwendungslandschaften ein nicht befriedigender Zustand. Besonders beim praktischen Einsatz zeigen sich dabei die Nachteile fehlender Generierungsmethoden. So werden z.B. Visualisierungen mit ≫ad hoc≪ eingefügten Anmerkungen erstellt, ohne dass die dort annotierten Informationen auch im semantischen Modell gepflegt werden. Meist sind derartige Informationen nicht einmal im Kontext des zu Grunde liegenden Informationsmodells vorgesehen, sind de facto vom Informationsbestand des Managementprozesses ≫entkoppelt≪. Diese Informationen werden losgelöst von den eigentlichen Informationen aus dem semantischen Modell gepflegt und führen so oftmals zu inkonsistenten Darstellungen. Lankes et al. haben in [LMW05b] ein Verfahren aufgezeigt, welches eine werkzeugunterstützte, automatische Generierung von Softwarekarten aus semantischen Modellen ermöglicht. Im Rahmen der vorliegenden Arbeit soll auf dieses Verfahren aufbauend die Architektur für ein Werkzeug entworfen und exemplarisch realisiert werden.

# Kapitel 3

# Anforderungsanalyse

Im Kapitel 2 wurde die Bedeutung von Softwarekarten als Werkzeug für das Management von Anwendungslandschaften dargestellt und die Schwierigkeiten bei deren Erzeugung aufgezeigt. Das in der vorliegenden Arbeit zu entwickelnde Werkzeug soll sich in das vorgestellte Umfeld integrieren und ein Verfahren zur automatischen Generierung von Softwarekarten realisieren. In diesem Kapitel erfolgt als erster Realisierungsschritt die Erhebung der Anforderungen an ein derartiges Werkzeug. Diese Erhebung beginnt mit Außensicht (≫Black-Box-Sicht≪) des zu realisierenden Werkzeugs in Abschnitt 3.1, identifiziert dort prominente Nutzerrollen und erhebt deren Anwendungsfälle.

Aus der Außensicht werden in der Folge konkrete Anforderungen gewonnen, die entsprechend einer Definition durch Sommerville in [So01] in *funktionale* (siehe Abschnitt 3.2) und *nicht-funktionale* (siehe Abschnitt 3.3) Anforderungen untergliedert werden. Dabei wird, soweit möglich, der Bezug zwischen den erarbeiteten Anforderungen und den im Rahmen der Enterprise Architecture Management Tool Survey 2005 vom Lehrstuhl für Software Engineering betrieblicher Informationssysteme in [se05] verwendeten Anforderungen hergestellt. Auch werden die konkreten Anforderungen mit den Anwendungsfällen in Beziehung gesetzt, aus denen sie jeweils hervorgegangen sind.

Abgeschlossen wird die Anforderungsanalyse durch eine Beschreibung der Benutzerschnittstelle des Werkzeugs in Abschnitt 3.4 sowie der in ihr enthaltenen Sichten und Dialoge. Dabei wird einer integrierten Benutzerschnittstelle besonderes Augenmerk geschenkt, in der auch ergonomische Aspekte Eingang finden.

## 3.1 Anwendungsfälle und Nutzerrollen

Die Beschreibung des Werkzeugs erfolgt in diesem Abschnitt aus der Perspektive der Nutzer anhand der von ihnen aufgeworfenen Anwendungsfälle. Diese Anwendungsfälle werden dabei zur besseren Referenzierbarkeit mit **UC*xx*** bezeichnet. Die verschiedenen Rollen von Nutzern dieses Werkzeugs werden ebenfalls eingeführt und in den Anwendungsfalldia-

grammen entsprechend durch Akteure repräsentiert.

### 3.1.1 Anwendungsfälle und Nutzerrollen auf Visualisierungen

Zentrale Anwendungsfälle des Werkzeugs und die mit ihnen assoziierten zentralen Rollen von Nutzern beziehen sich auf die Visualisierungen im Werkzeug. Einen Überblick über diese Rollen und Anwendungsfälle bietet dabei Abbildung 3.1, deren Elemente nachfolgend detailliert werden.

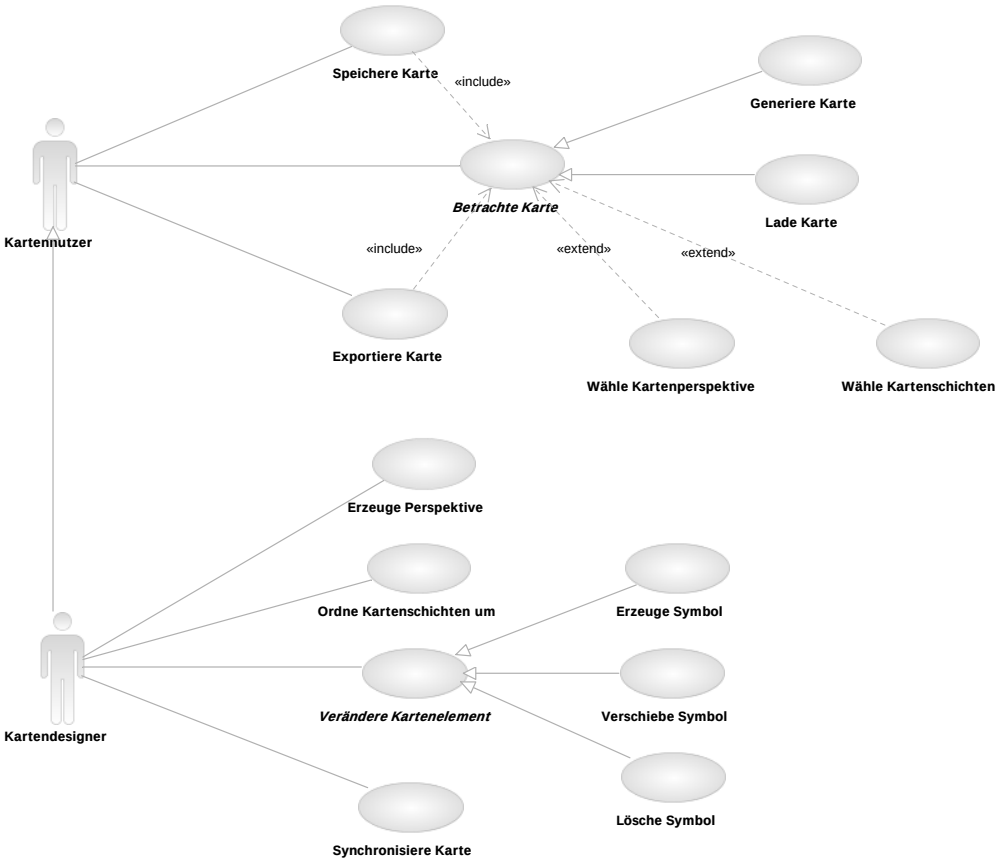



Abbildung 3.1: Zentrale Nutzerrollen und Anwendungsfälle

Die zentralen Rollen von Benutzern sind dabei:

**Kartennutzer** mit Rechten Softwarekarten sowie den ihnen zu grunde liegenden Daten zu lesen.

**Kartendesigner** mit Rechten Softwarekarten sowie die ihnen zu grunde liegenden Daten zu ändern.

Die Anwendungsfälle des Kartennutzers sind dabei:

**UC1** *Betrachte Karte:* Dieser Anwendungsfall ist ein *abstrakter* Anwendungsfall zum Betrachten einer Karte, der durch folgende konkrete Fälle realisiert wird:

- **UC1a** *Lade Karte:* In diesem Anwendungsfall wird eine bereits erzeugte und abgespeicherte Karte geladen und wieder angezeigt.
- **UC1b** *Generiere Karte:* In diesem Anwendungsfall wird eine neue Karte aus den Daten des semantischen Modells unter Anwendung eines konkreten Blickwinkels automatisch erzeugt.

**UC2** *Wähle Kartenschichten:* Dieser Anwendungsfall stellt eine Erweiterung von *UC1* dar und ermöglicht es dem Nutzer die angezeigten Schichten auszuwählen.

**UC3** *Wähle Kartenperspektive:* Dieser Anwendungsfall stellt eine Erweiterung von *UC1* dar und ermöglicht es dem Nutzer die anzuzeigende Perspektive auszuwählen.

**UC4** *Speichere Karte:* Dieser Anwendungsfall erlaubt es dem Benutzer eine generierte und eventuell veränderte Karte, welche durch einen *UC1* angezeigt wird, abzuspeichern.

**UC5** *Exportiere Karte:* Dieser Anwendugnsfall erlaubt es dem Benutzer eine Karte, welche durch einen *UC1* angezeigt wird, in ein Graphikformat zu exportieren.

Dem Kartendesigner stehen darüber hinaus weitere Anwendungsfälle zur Verfügung:

**UC6** *Ordne Kartenschichten um:* Dieser Anwendungsfall stellt eine Erweiterung von *UC1* dar und ermöglicht es dem Designer die Reihenfolge der angezeigten Schichten zu ändern.

**UC7** *Verändere Kartenelemente:* Dieser Anwendungsfall ist ein *abstrakter* Anwendungsfall und bildet die Basis für allgemeine Veränderungen an Kartenelementen gegenbenfalls mit Implikationen auf das semantische Modell. Er wird durch folgende konkrete Fälle realisiert:

- **UC7a** *Erzeuge Symbol:* Dieser Anwendungsfall erlaubt es dem Designer ein neues Symbol in eine Karte einzufügen.
- **UC7b** *Lösche Symbol:* Dieser Anwendungsfall erlaubt es dem Designer ein Symbol aus der Karte zu entfernen.
- **UC7c** *Verschiebe Symbol:* Dieser Anwendungsfall erlaubt es dem Designer die Position eines Symbols in der Karte zu verändern. Hier werden alternative Realisierungen unterschieden, bezüglich der Implikationen der Veränderung auf das *semantische Modell*.

**UC8** *Synchronisiere Karte:* Dieser Anwendungsfall erlaubt es dem Designer die Informationen, welche der Karte zugrunde liegen (ihr semantisches Modell), mit einem zentralen Modell abzugleichen und Änderungen zu propagieren.

### 3.1.2 Anwendungsfälle und Nutzerrollen auf Daten

Die Bearbeitung der Daten aus dem *semantischen Modell* (siehe 2.3) kann auch auf den Informationsobjekten direkt und nicht mittels der Visualisierung erfolgen. Einen Überblick über die in diesem Kontext zu unterscheidenden Rollen von Nutzern und deren Anwendungsfälle gibt Abbildung 3.2.

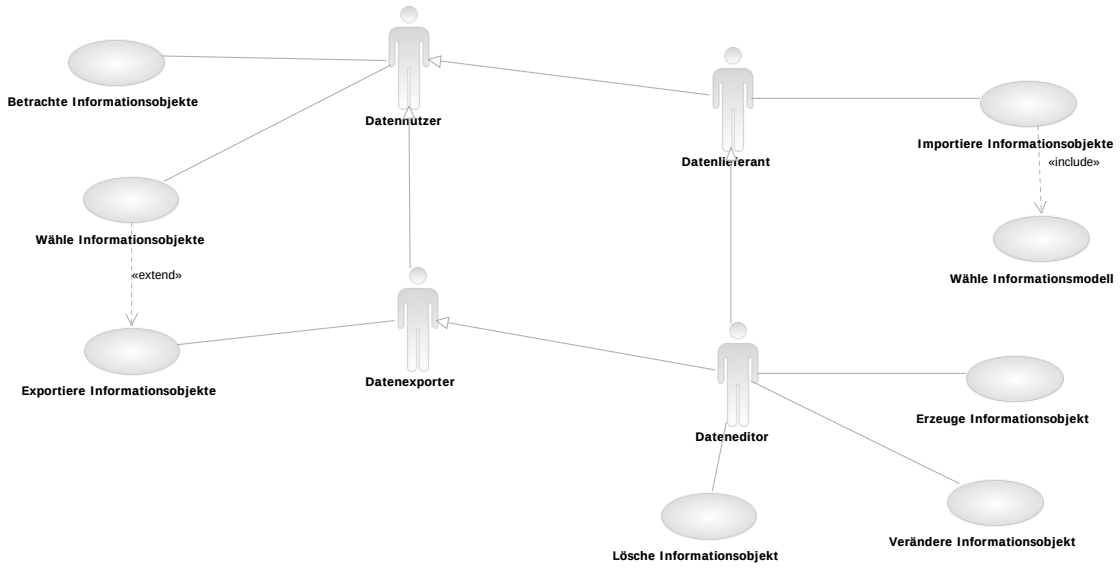



Abbildung 3.2: Nutzerrollen und Anwendungsfälle auf dem semantischen Modell

Die Rollen von Benutzern werden dabei wie folgt voneinander abgegrenzt:

**Datennutzer** hat lesende Rechte auf einer Repräsentation des semantischen Modells.

**Datenlieferant** kann neue semantische Modelle importieren.

**Datenexporter** kann die Informationsobjekte aus einem semantischen Modell oder einen Teil von ihnen exportieren.

**Dateneditor** hat Änderungsrechte auf semantischen Modellen.

Während bei den Karten der Anwendungsfall *Exportiere Karte* **UC5** dem Kartennutzer zugeordnet wurde, existiert auf den Daten eine eigene Rolle dafür, der *Exporter*. Dies geschah im Hinblick auf die Einbettung des Werkzeugs in die vorhandene Landschaft von Programmen zur Unterstützung der Dokumentation von Anwendungslandschaften, welche auf Datenexporte zur Synchronisation angewiesen sein könnten.

Die Anwendungsfälle des Datennutzers sind dabei:

**UC9** *Betrachte Informationsobjekte:* Dieser Anwendungsfall erlaubt es dem Nutzer alle Informationsobjekte im semantischen Modell zu betrachten oder sich Detailinformationen zu einem Informationsobjekt anzusehen.

**UC10** *Wähle Informationsobjekte:* Dieser Anwendungsfall erlaubt es dem Nutzer ein oder mehrere Informationsobjekte aus der Menge aller Objekte auszuwählen und gegebenfalls zu sortieren.

Dem Datenlieferanten stehen darüber hinaus weitere Anwendungsfälle zur Verfügung:

**UC11** *Importiere Informationsobjekte:* Dieser Anwendungsfall erlaubt es dem Datenlieferanten ein semantisches Modell, welches einem gegebenen Informationsmodell gehorcht, nutzbar zu machen.

**UC12** *Wähle Informationsmodell:* Dieser Anwendungsfall erlaubt es dem Datenlieferanten, das einem Import zugrunde liegende Informationsmodell auszuwählen.

Der Datenexporter verfügt neben den Anwendungsfällen des Datennutzers noch über folgenden Anwendungsfall:

**UC13** *Exportiere Informationsobjekte:* Dieser Anwendungsfall erlaubt es dem Datenexporter, alle oder ausgewählte Informationsobjekte aus dem semantischen Modell zu exportieren. Dieser Anwendungsfall kann durch *UC10* erweitert werden, d.h. die Auswahl von Informationsobjekten *kann* dem Export vorausgehen.

Der Dateneditor vereinigt die Anwendungsfälle von Datenexporter und Datenlieferant und verfügt darüber hinaus noch über folgende eigene Anwendungsfälle:

**UC14** *Erzeuge Informationsobjekt:* Dieser Anwendungsfall erlaubt es dem Dateneditor ein zum gegenwärtig gewählten Informationsmodell passendes Informationsobjekt in das semantische Modell einzufügen.

**UC15** *Verändere Informationsobjekt:* Dieser Anwendungsfall erlaubt es dem Dateneditor ein existierendes Informationsobjekt aus dem semantischen Modell unter Beachtung der Integritätsbedingungen aus dem gegenwärtig gewählten Informationsmodell zu verändern.

**UC16** *Lösche Informationsobjekt:* Dieser Anwendungsfall erlaubt es dem Dateneditor ein existierendes Informationsobjekt aus dem semantischen Modell unter Beachtung der Integritätsbedingungen aus dem gegenwärtig gewählten Informationsmodell zu verändern.

### 3.1.3 Anwendungsfälle und Nutzerrollen auf dem Informationsmodell

Wesentlicher Bestandteil des Werkzeugs ist, wie in Abschnitt 2.3 dargestellt, das dem semantischen Modell zugrunde liegende *Informationsmodell*. Dieses Modell, welches das »Vokabular« für die objektorientierte Modellierung einer Anwendungslandschaft enthält, wird einer eigenen Nutzerrolle verwaltet. Einen Überblick über deren Anwendungsfälle gibt Abbildung 3.3.

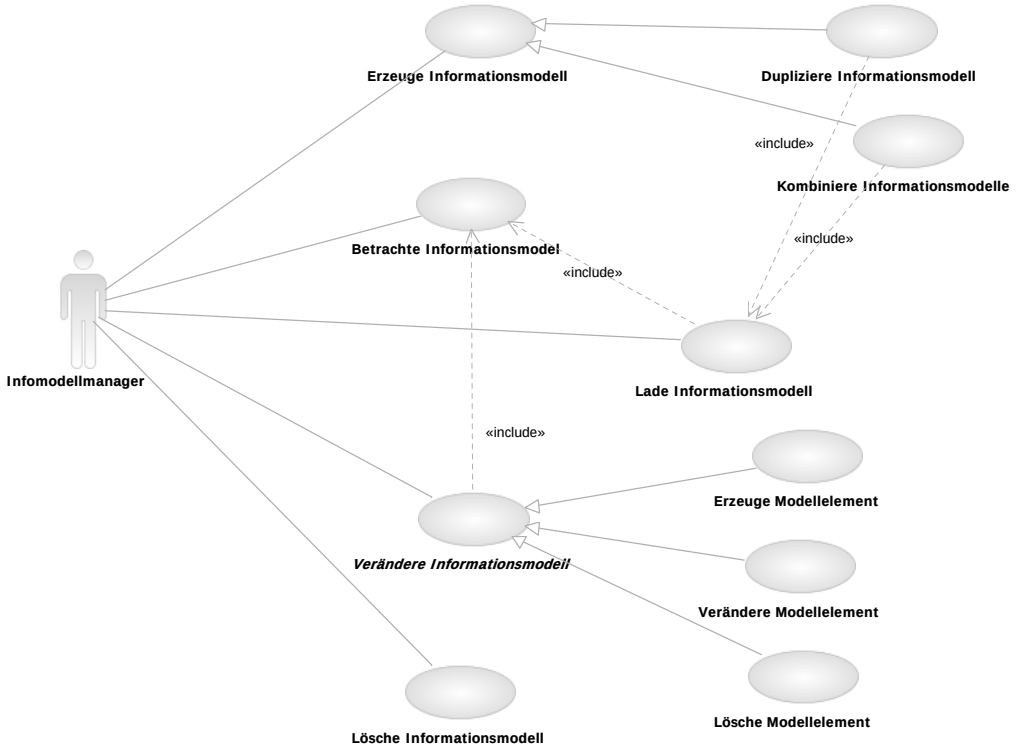



Abbildung 3.3: Nutzerrolle und Anwendungsfälle auf dem Informationsmodell

**Infomodellmanager** hat Berechtigungen Informationsmodelle zu lesen und zu schreiben.

Die Anwendungsfälle des Informationsmodellmanagers sind dabei die folgenden:

**UC17** *Erzeuge Informationsmodell:* Dieser Anwendungsfall erlaubt es dem Informationsmodellmanager ein neues, leeres Informationsmodell zu erstellen.

**UC17a** *Dupliziere Informationsmodell:* Dieser Anwendungsfall spezialisiert *UC17* und erlaubt es dem Informationsmodellmanager eine Kopie eines bestehenden Informationsmodells zu erstellen.

**UC17b** *Kombiniere Informationsmodelle:* Dieser Anwendungsfall stellt eine Spezialisierung von *UC17* dar und erlaubt es dem Informationsmodellmanager zwei verschiedene Informationsmodelle anhand gemeinsamer Elemente semiautomatisch zusammenzuführen.

**UC18** *Betrachte Informationsmodell:* Dieser Anwendungsfall erlaubt es dem Informationsmodellmanager die Klassen, Attribute, Assoziationen und Integritätsbedingungen im Informationsmodell zu betrachten.

**UC19** *Lade Informationsmodell:* Dieser Anwendungsfall erlaubt es dem Informationsmodellmanager ein bestehendes Informationsmodell zu laden.

**UC20** *Verändere Informationsmodell:* Dieser Anwendungsfall ist der *abstrakte* Basisanwendungsfall für die Veränderung von Elementen im aktuellen Informationsmodell.

**UC20a** *Erzeuge Modellelement:* Dieser Anwendungsfall spezialisiert *UC20* und erlaubt die Erstellung neuer Elemente im Informationsmodell.

**UC20b** *Verändere Modellelement:* Dieser Anwendungsfall spezialisiert *UC20* und erlaubt die Veränderung bestehender Elemente im Informationsmodell.

**UC20c** *Lösche Modellelement:* Dieser Anwendungsfall spezialisiert *UC20* und erlaubt die Löschung bestehender Elemente im Informationsmodell unter Beachtung gegebenenfalls bestehender Dependenzen.

**UC21** *Lösche Informationsmodell:* Dieser Anwendungsfall erlaubt es dem Informationsmodellmanager ein bestehendes Informationsmodell zu löschen.

### 3.1.4 Anwendungsfälle und Nutzerrollen auf den Visualisierungskonzepten

Den vom Werkzeug erzeugten Visualisierungen liegt ein ≫Vokabular≪ von Visualisierungskonzepten, das sogenannte *Visualisierungsmodell* (eine Definition folgt in Abschnitt 4.2) zugrunde. Dieses Modell wird von einer eigenen Nutzerrolle verwaltet. Einen Überblick über deren Anwendungsfälle gibt Abbildung 3.4. Die Anwendungsfälle des Visualisierungsmodellmanagers sind dabei weitgehend äquivalent zu denen des Informationsmodellmanagers, also zu *UC17* - *UC21*.

**Visualisierungsmodellmanager** hat Berechtigungen Visualisierungsmodelle zu lesen und zu schreiben.

Die Anwendungsfälle des Visualisierungsmodellmanagers sind folgende:

**UC22** *Erstelle Visualisierungsmodell:* Dieser Anwendungsfall erlaubt es dem Visualisierungsmodellmanager ein neues, leeres Visualisierungsmodell zu erstellen.

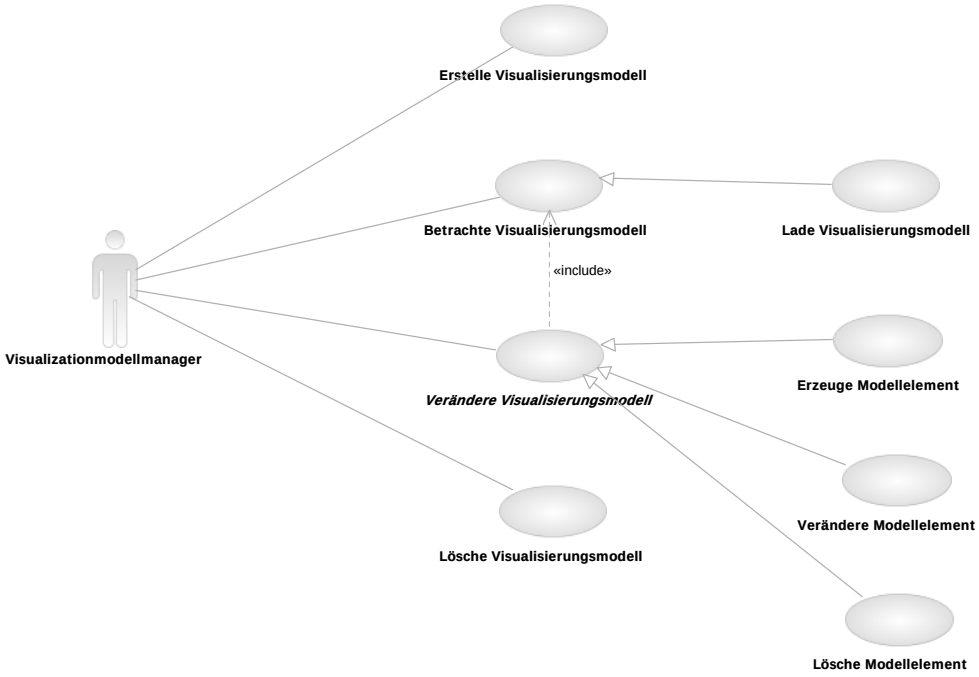



Abbildung 3.4: Nutzerrolle und Anwendungsfälle aus dem Visualisierungsmodell

**UC23** *Betrachte Visualisierungsmodell:* Dieser Anwendungsfall erlaubt es dem Visualisierungsmodellmanager die Klassen, Attribute, Assoziationen und annotierten Prädikate im Visualisierungsmodell zu betrachten.

- **UC23a** *Lade Visualisierungsmodell:* Dieser Anwendungsfall erlaubt es dem Visualisierungsmodellmanager ein bestehendes Visualisierungsmodell zur Betrachtung zu laden.

**UC24** *Verändere Visualisierungsmodell:* Dieser Anwendungsfall ist der *abstrakte* Basisanwendungsfall für die Veränderung von Elementen im aktuellen Visualisierungsmodell.

- **UC24a** *Erzeuge Modellelement:* Dieser Anwendungsfall spezialisiert *UC24* und erlaubt die Erstellung neuer Elemente im Visualisierungsmodell.
- **UC24b** *Verändere Modellelement:* Dieser Anwendungsfall spezialisiert *UC24* und erlaubt die Veränderung bestehender Elemente im Visualisierungsmodell.
- **UC24c** *Lösche Modellelement:* Dieser Anwendungsfall spezialisiert *UC24* und erlaubt die Löschung bestehender Elemente im Visualisierungsmodell.

**UC25** *Lösche Visualisierungsmodell:* Dieser Anwendungsfall erlaubt es dem Visualisierungsmodellmanager ein bestehendes Visualisierungsmodell zu löschen.

### 3.1.5 Anwendungsfälle und Nutzerrollen auf den Visualisierungsvorlagen

Bei der Erstellung von Visualisierungen greift das Werkzeug auf definierte ≫Blickwinkel≪ oder ≫Vorlagen≪ für Visualisierungen, d.h. auf Regeln zurück, welche beschreiben, wie eine Visualisierung aus einem gegebenen semantischen Modell erstellt werden soll. Diese Transformationsregeln (eine Definition folgt in Abschnitt 4.3) werden von einer eigenen Nutzerrolle verwaltet. Einen Überblick über deren Anwendungsfälle gibt Abbildung 3.5.

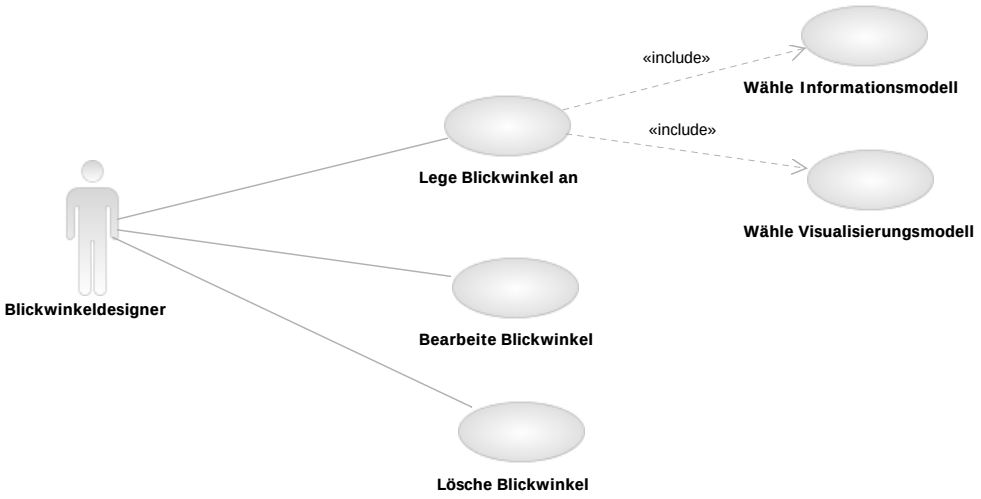



Abbildung 3.5: Nutzerrolle und Anwendungsfälle auf den Transformationsregeln

**Blickwinkeldesigner** hat Berechtigungen Blickwinkel (und die in ihnen enthaltenen Transformationsregeln) zu lesen und zu schreiben.

Die Anwendungsfälle des Blickwinkeldesigners sind folgende:

**UC26** *Lege Blickwinkel an:* Dieser Anwendungsfall erlaubt es dem Blickwinkeldesigner einen neuen (leeren) Blickwinkel zu erstellen.

- **UC26a** *Wähle Informationsmodell:* Dieser Anwendungsfall erlaubt es dem Blickwinkeldesigner zu einem gegebenen Blickwinkel das verwendete Informationsmodell (bzw. den Informationsmodellausschnitt) festzulegen.
- **UC26b** *Wähle Visualisierungsmodell:* Dieser Anwendungsfall erlaubt es dem Blickwinkeldesigner zu einem gegebenen Blickwinkel das verwendete Visualisierungsmodell (bzw. den Visualisierungsmodellausschnitt) festzulegen.

**UC27** *Bearbeite Blickwinkel:* Dieser Anwendungsfall erlaubt es dem Blickwinkeldesigner einen bestehende Blickwinkel zu verändern, indem er Transformationsregeln hinzufügt, verändert oder löscht.

**UC28** *Lösche Blickwinkel:* Dieser Anwendungsfall erlaubt es dem Blickwinkeldesigner einen bestehenden Blickwinkel zu löschen

## 3.2 Funktionale Anforderungen

Die in Abschnitt 3.1 dargestellten Anwendungsfällen werden in den folgenden Abschnitten durch funktionale Anforderungen konkretisiert und detailliert. Diese Anforderungen werden mit Rücksicht auf die Referenzierbarkeit mit **AF*x*** durchnummeriert.

**AF1** *Visualisierung von semantischen Modellen:* Aus den verschiedensten semantischen Modelle sollen sich unter Nutzung geeigneter Vorlagen Visualisierungen generieren lassen, welche aus Symbolen zusammengesetzt die Informationen der entsprechenden Modelle transportieren. Dabei sollen diese Visualisierungen dargestellt und mit kontextsensitiven Elementen versehen werden. Auch sollen die Visualisierungen in entsprechende graphische Formate (vgl. auch **ANF2**) zur Weiterverarbeitung in anderen Werkzeugen exportiert werden können.

**AF2** *Konfigurierbarkeit des Informationsmodells:* Im Hinblick auf die Unterschiedlichkeit der Anforderungen, welche verschiedene Unternehmen bei der Dokumentation ihrer Anwendungslandschaften haben können, soll das Werkzeug die freie Konfigurierbarkeit der Modelle für die Dokumentation unterstützen. Mit Verweis auf die Definitionen für Modell und Metamodell in Abschnitt 2.3, seien hier nur die Möglichkeiten beliebige *Konzepte*, wie z.B. Infrastrukturkomponenten, beliebige *typisierte Attribute*[1] und beliebige *Beziehungen* zu unterstützen, erwähnt. Dabei soll, soweit möglich, ein umfassende Unterstützung etablierter Metamodellierungskonzepte, z.B. aus der *Meta Object Facility* der OMG (vergleiche [OM06]), realisiert werden.

**AF3** *Flexibilität der Visualisierungskonzepte:* Im Hinblick auf die Unterschiedlichkeit der von verschiedenen Unternehmen auf Softwarekarten genutzten visuellen Konzepte, soll das Werkzeug die freie Konfigurierbarkeit des Konzeptkanons für die Visualisierungen unterstützen. Als *Visualisierungskonzepte* gelten dabei Symbole, wie z.B. Kreise, Chevren, Ellipsen oder Text, sowie Visualisierungsregeln, wie z.B. die Positionierungsregel der Schachtelung von Symbolen (siehe Abbildung 2.6).

Visualisierungskonzept

**AF4** *Flexibilität der Visualisierungsvorlagen:* Zwar lassen sich Softwarekarten z.B. über die Softwarekartentypen geeignet klassifizieren, jedoch liegen den verschiedenen Karten selbst desselben Typs unterschiedliche Regelwerke zugrunde. Um verschiedenste Arten von Visualisierungen unterstützen zu können, muss es dem Benutzer möglich

[1] Bei einem *typisierten* Attribut handelt es sich um ein Attribut mit einem festgelegten Datentyp, z.B. Ganzzahl oder Zeichenkette.

sein, die Regeln, welche einer Visualisierungsart zugrunde liegen, anzupassen oder um neue Regeln zu erweitern. So sollte es z.B. möglich sein, die Prozessunterstützung aus Abbildung 2.7 auf eine geringere Genauigkeit im Hinblick auf die Prozesskette zu adaptieren, d.h. nur Prozesse des Levels eins einzublenden.

**AF5** *Datenhaltung:* Das Repository des Werkzeugs übernimmt die Persistierung von Daten, welche zu einem gegebenen Informationsmodell passen. Daten aus externen Datenquellen sollen jedoch in das Repository des Werkzeugs importiert werden können. Darüber hinaus sollen geeignete Exportfunktionalitäten zur Verfügung gestellt werden, wodurch die Daten in ein Format überführt werden, das für die Weiterverarbeitung durch Mensch oder Maschine vorgesehen ist.

**AF6** *Modellierung im semantischen Modell:* Das Werkzeug soll Modellierungsfähigkeiten anbieten, bzw. das Anlegen neuer Instanzen von Klassen aus dem Informationsmodell durch die Benutzerschnittstelle, u.A. auch durch die Visualisierungen direkt unterstützen. Bereits bestehende Instanzen sollen hinsichtlich der Werte der zu ihnen gehörigen Variablen verändert werden können. Diese Veränderungen sollen mit dem Repository synchronisiert werden können.

Die oben aufgeführten funktionalen Anforderungen orientieren sich dabei an den Anforderungen, die vom Lehrstuhl für Software Engineering betrieblicher Informationssysteme in [se05] dargestellt worden sind. Die Verknüpfung dieser Anforderungen mit den konkreten Anwendungsfällen aus Abschnitt 3.1 wird in Tabelle 3.1 aufgezeigt[2]. In dieser Tabelle bedeutet das Symbol *****, dass der Anwendungsfall aus der entsprechenden Zeile durch die funktionale Anforderung aus der zugehörigen Spalte realisiert wird. Wie der Tabelle zu entnehmen ist, werden alle aufgeführten Anwendungsfälle in funktionale Anforderungen übersetzt.

## 3.3 Nicht-funktionale Anforderungen

Analog zu den funktionalen Anforderungen in Abschnitt 3.2 soll das Werkzeug hier in den Kontext seiner Nutzungsumgebung eingebettet und an deren organisatorische wie technische Gegebenheiten angebunden werden. Daraus können nicht-funktionale Anforderungen abgeleitet werden, die hier im Hinblick auf die Referenzierbarkeit mit **ANF*x*** durchnummiert werden.

**ANF1** *Einsatzsumgebung:* Das Werkzeug ist als eine Client-Server Anwendung mit einem Fat-Client gedacht. Mehrbenutzerfähigkeit soll besonders durch die zentrale objektorientierte Datenhaltung auf dem Server unterstützt werden. Es sollen idealerweise verschiedene Desktopbetriebssysteme, so z.B. Microsoft Windows XP, unterstützt werden, was durch den konsequenten Einsatz von J2SE 5.0 auf Basis der Sun Java Virtual Machine ermöglicht wird.

[2]In dieser Tabelle werden abstrakte Anwendungsfälle nicht berücksichtigt.

| | AF1 | AF2 | AF3 | AF4 | AF5 | AF6 |
|---|---|---|---|---|---|---|
| UC1a | * | | | | | |
| UC1b | * | | | | | |
| UC2 | * | | | | | |
| UC3 | * | | | | | |
| UC4 | * | | | | | |
| UC5 | * | | | | | |
| UC6 | * | | | | | |
| UC7a | | | | | | * |
| UC7b | | | | | | * |
| UC7c | | | | | | * |
| UC8 | | | | | | * |
| UC9 | | | | | * | * |
| UC10 | | | | | * | * |
| UC11 | | | | | * | |
| UC12 | | | | | * | |
| UC13 | | | | | * | |
| UC14 | | | | | | * |
| UC15 | | | | | | * |
| UC16 | | | | | | * |
| UC17 | | * | | | | |
| UC17a | | * | | | | |
| UC17b | | * | | | | |
| UC18 | | * | | | | |
| UC19 | | * | | | | |
| UC20a | | * | | | | |
| UC20b | | * | | | | |
| UC20c | | * | | | | |
| UC21 | | * | | | | |
| UC22 | | | * | | | |
| UC23 | | | * | | | |
| UC23a | | | * | | | |
| UC24a | | | * | | | |
| UC24b | | | * | | | |
| UC24c | | | * | | | |
| UC25 | | | * | | | |
| UC26 | | | | * | | |
| UC26a | | | | * | | |
| UC26b | | | | * | | |
| UC27 | | | | * | | |
| UC28 | | | | * | | |

Tabelle 3.1: Beziehungen zwischen Anwendungsfällen (**UC**) und funktionalen Anforderungen (**AF**)

**ANF2** *Unterstützung für Standards und Formate* Die Implementierung des Werkzeugs soll sich an den Standards im Umfeld der Eclipse Rich Client Platform (Version 3.1) [Ec06] und [Da05] orientieren. Darüber hinaus soll der *XML Metadata Interchange (XMI)* der OMG [OM05] für die Beschreibung von objektorientierten Modellen verschiedener Meta-Niveaus verwendet werden. Als weiteres Format für semantische Modelle, soll ein auf dem Microsoft Excel Spreadsheet Format (siehe Anhang [Mi03]) basierendes Format (vergleiche hierzu C) unterstützt werden. Die vom Werkzeug erzeugten Visualisierungen sollen darüber hinaus in geeignete Grafikformate exportierbar sein, einerseits als »Abzüge« in das *Portable Network Graphic (PNG)* [W3C03c]-Format, andererseits als interaktive Grafik mit Funktionen zum Ein- und Ausblenden von Schichten in das *Scalable Vector Graphic (SVG)* [W3C03d]-Format mit *ECMA-Script* [Ec99] Elementen.

Eine Beziehung zu den Anwendungsfällen aus Abschnitt 3.1 wird hier nicht hergestellt, da sich die nicht-funktionalen Anforderungen kaum auf Gegebenheiten der Außensicht des Werkzeugs beziehen.

## 3.4 Benutzerschnittstelle

Die Benutzerschnittstelle des Werkzeugs wird in diesem Abschnitt vorgestellt. Dabei wird ihr Aufbau an den von verwandten Modellierungswerkzeugen, wie z.B. dem IBM Rational Software Developer angelehnt. In Abbildung 3.6 wird der Gesamtaufbau der Benutzerschnittstelle des Werkzeugs dargestellt, die Abschnitte 3.4.1 bis 3.4.9 stellen die einzelnen Ansichten im Detail vor (vgl. Abbildung 3.6) und setzen sie mit den Anwendungsfällen aus Abschnitt 3.1 in Beziehung. Die Auflistung der Beziehungen zu den Anwendungsfällen im Detail findet sich in den Tabellen A.1 und A.2 in Anhang A. Dabei ist den Tabellen zu entnehmen, dass die Anwendungsfälle **UC17**-**UC21** auf dem Informationsmodell und **UC22**-**UC25** auf dem Visualisierungsmodell nicht in der Benutzerschnittstelle berücksichtigt wurden. Dies geschah im Falle des Informationsmodells mit Hinblick auf die gute Unterstützung der objektorientierten Modellierung durch externe Werkzeuge, wie z.B. dem *IBM Rational Software Modeler*, der in seiner Version 6 den Export in etablierte Standards, wie z.B. XMI unterstützt. Auch die Modellierung von Visualisierungskonzepten kann durch externe Werkzeuge der objektorientierten Modellierung unterstützt werden, wenngleich für diesen Einsatz eine Unterstützung bei der Formulierung der mathematischen Konkretisierung (siehe Abschnitt 4.2.2) nützlich wäre. Derartige Aspekte wurden jedoch von Ernst et. al. in [Er06] noch nicht eingehend untersucht und sind deswegen nicht Bestandteil der vorliegenden Arbeit. Die verschiedenen Anforderungen an die Benutzerschnittstelle werden in der Folge im Hinblick auf ihre Referenzierbarkeit mit **AB*xx*** durchnummeriert.

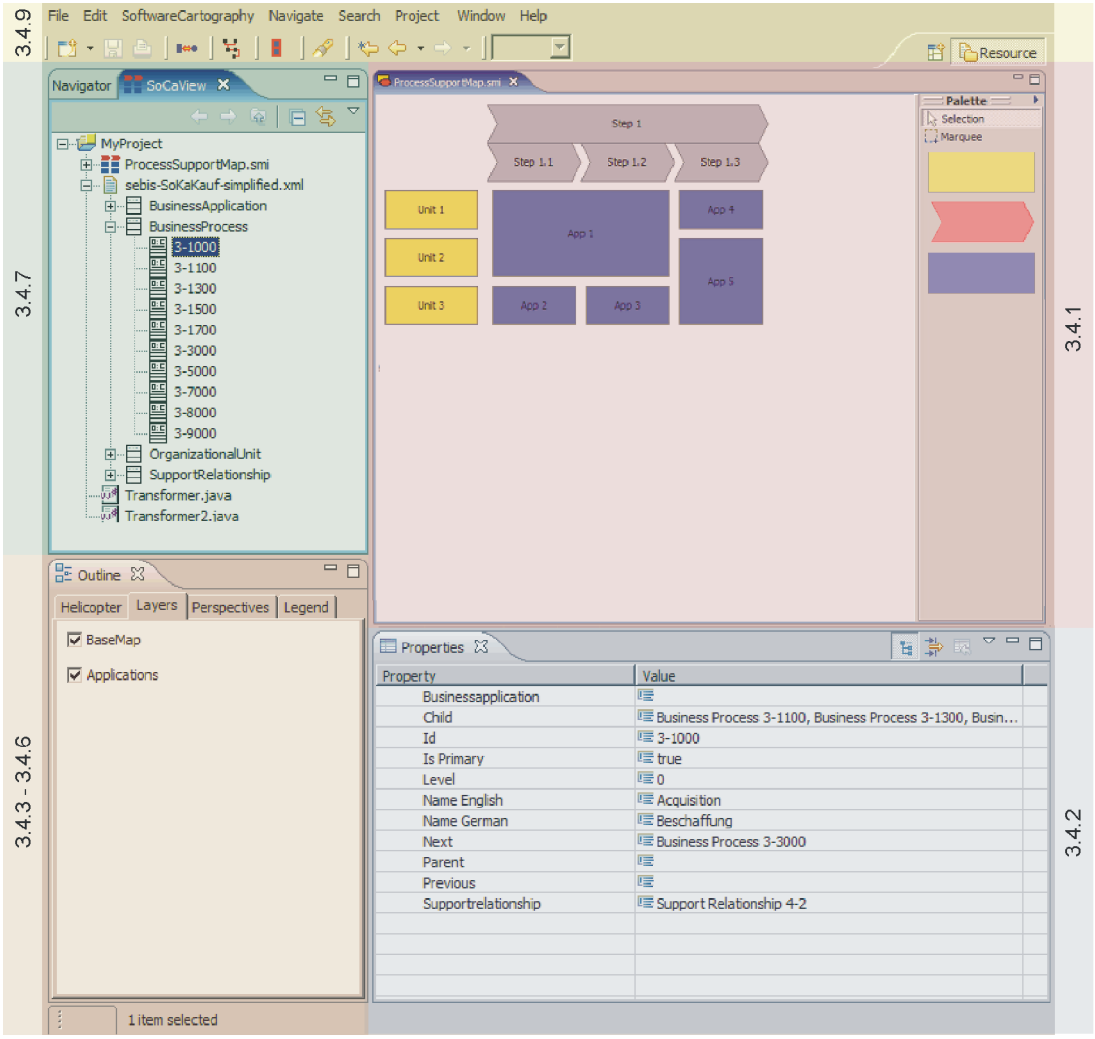



Abbildung 3.6: Gesamtansicht Benutzerschnittstelle

### 3.4.1 Benutzerschnittstelle - Modell

Im Modell (siehe Abbildung 3.7) wird die aktuelle Visualisierung (das entsprechende symbolische Modell) dargestellt. Zu diesem symbolischen Modell gehört auch eine Kopie des dargestellten Ausschnitts aus dem semantischen Modell. Teil der Ansicht ist dabei eine *Palette*, auf welcher die in der Visualisierung verwendeten Symbole zum Erstellen angeboten werden. Im Modell können folgende Operationen ausgeführt werden:

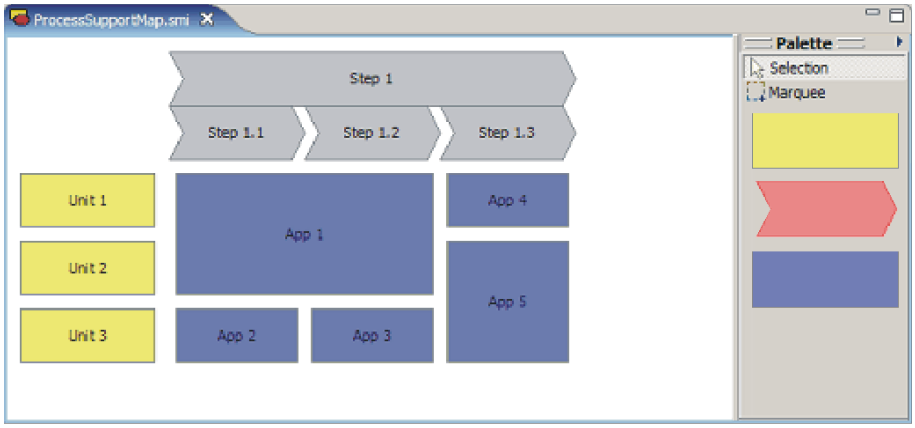



Abbildung 3.7: Benutzerschnittstelle - Modell

**AB1** *Betrachten des Modells:* Die Visualisierung kann im Modellfenster als Ganzes oder in Ausschnitten betrachtet werden. Dabei kann der Ausschnitt verschoben werden.

**AB2** *Zoom:* Die Visualisierung kann auf verschiedenen Stufen des Zooms dargestellt werden. Der Grad kann sowohl stufenweise erhöht werden, als auch aus einer Auswahlliste ausgewählt werden, wobei neben Prozentoptionen auch spezielle Stufen für die Anpassung an Seitenhöhe oder -breite zur Verfügung gestellt werden.

**AB3** *Auswählen eines Symbols:* Ein Symbol auf der Visualisierung kann durch Linksklick als ausgewählt markiert werden. Diese Selektion nimmt Einfluss auf die Anzeigen in korrespondierenden Ansichten, z.B. den *Eigenschaften* und dem *Datenbaum*.

**AB4** *Erstellen eines Symbols:* Ein Symbol kann von der Palette auf die Fläche des Modells gezogen werden (»drag-and-drop«), wodurch ein Informationsobjekt in der zur Visualisierung gehörigen Kopie des semantischen Modells erstellt wird, welches dem verwendeten Symbol entspricht. Optional wird anschließend der Fokus auf die Eigenschaftsansicht gesetzt, in welcher der Benutzer weitere Daten eingeben muss.

**AB5** *Löschen eines Symbols:* Ein Symbol kann mittels eines Kontextmenüs oder per Tastaturbefehl aus der Visualisierung entfernt werden. Dabei können zwei Ausprägungen des Löschbefehls unterschieden werden:

**AB5a** *Löschen ohne semantische Implikation:* Entfernt nur das Symbol, nicht aber das zugrunde liegende Informationsobjekt.

**AB5b** *Löschen mit semantischer Implikation:* Entfernt das Symbol und auch das/-die zum Symbol gehörigen Informationsobjekt(e) aus der zur Visualisierung gehörigen Kopie des semantischen Modells.

**AB6** *Verändern eines Symbols:* Ein Symbol kann innerhalb der Visualisierung verschoben oder bezüglich seiner Größe, respektive seines Verlaufs (bei Linien), verändert werden. Bei diesen Veränderungsbefehlen sind zwei unterschiedliche Betriebsmodi denkbar:

**AB6a** *Verändern der Visualisierung ohne semantische Implikation:* Verändert nur das Symbol und verhindert Veränderungen, welche die Semantik der Visualisierung verändern würden.

**AB6b** *Verändern der Visualisierung mit semantischer Implikation:* Verändert das Symbol und auch das/die zum Symbol gehörigen Informationsobjekt(e) aus der zur Visualisierung gehörigen Kopie des semantischen Modells. Dabei kann es sich bei dem veränderten Attribut oder der veränderten Beziehung um »read-only« Elemente handeln, so dass die Ausführung der semantischen Implikation unterbleibt.

### 3.4.2 Benutzerschnittstelle - Eigenschaften

In der Ansicht der Eigenschaften (siehe Abbildung 3.8) werden die Eigenschaften des aktuell ausgewählten Elements angezeigt. Die Auswahl kann sich dabei sowohl auf Symbole im Modell beziehen, als auch z.B. ein oder mehrere Informationsobjekte aus dem *Datenbaum* zum Inhalt haben. Die Eigenschaften des korrespondierenden, bzw. entsprechenden Informationsmodellobjekts werden dabei in einer Tabelle angezeigt, welche zum einen den Namen der Eigenschaft, zum anderen den gesetzten Wert enthält, wodurch folgende Operationen möglich gemacht werden:

**AB7** *Lesen eines Eigenschaftswerts:* Die Ansicht erlaubt es den Wert einer Eigenschaft eines Informationsobjekts zu lesen. Der Wert wird dabei in Form einer Zeichenkette angezeigt und farblich hervorgehoben, wenn es sich bei dem Wert um einen Default-Eintrag handelt. Der Name des korrespondieren Attributs wird ebenfalls angezeigt und markiert, falls das entsprechende Attribut verpflichtende Einträge erfordert.

**AB8** *Gruppierung nach Informationsobjekten:* Sind mehrere Informationsobjekte ausgewählt, so zergliedert sich die Ansicht in mehrere Tabellenelemente, welche via Klick aus- bzw. eingeklappt werden können.

**AB9** *Verändern eines Eigenschaftswerts:* Die Ansicht erlaubt es den Wert einer Eigenschaft eines Informationsobjekts zu verändern, sofern die Eigenschaft nicht als »read-only« markiert ist. Dabei stehen abhängig vom Typ der Eigenschaft verschiedene

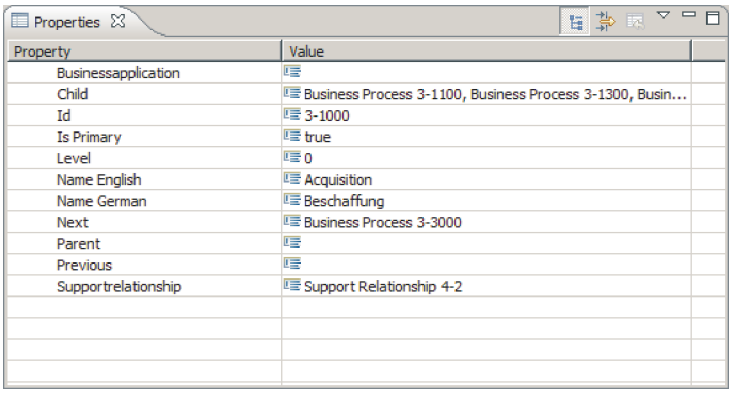



Abbildung 3.8: Benutzerschnittstelle - Eigenschaften

Funktionen zur Verfügung. Primitive Typen, wie z.B. Zahlen oder Zeichenketten, werden mittels eines Texteingabefeldes gesetzt, für Aufzählungs- oder Referenztypen stehen entsprechend vorbelegte Auswahllisten zur Verfügung. Veränderte Werte werden dabei nicht an die aktuelle Visualisierung im *Modell* propagiert.

**AB10** *Löschen eines Eigenschaftswerts:* Die Ansicht erlaubt es den Wert einer Eigenschaft eines Informationsobjekts zu löschen. Dies kann als eine Form des ≫Null-Setzens≪ verstanden werden, die jedoch nur auf Attribute angewendet werden kann, die im Informationsmodell als *optional* markiert worden sind. Beim Nullwert handelt es sich darüber hinaus nicht um eine Ausprägung des entsprechenden Datentyps, so muss z.B. bei einer Ganzzahl zwischen den Werten ≫0≪ und ≫null≪ unterschieden werden.

### 3.4.3 Benutzerschnittstelle - Helikopterperspektive

In der Helikopterperspektive (siehe Abbildung 3.9) wird die aktuelle Visualisierung aus dem *Modell* verkleinert dargestellt und mit einem Ausschnitt überlagert, welcher die aktuell sichtbaren Bereiche der Visualisierung überdeckt. In dieser Ansicht sollen dabei folgende Operationen unterstützt werden:

**AB11** *Helikopterperspektive betrachten:* In dieser Perspektive kann der Benutzer einen Verkleinerung des aktuellen Modells mit überlagertem Fensterausschnitt (Auswahlrechteck) betrachten.

**AB12** *Auswahlrechteck verschieben:* Wird im Modell nicht die gesamte Visualisierung angezeigt, so erscheint in der Helikopterperspektive ein Auswahlrechteck, welches den angezeigten Bereich der Visualisierung umfasst. Dieses Rechteck kann durch den Benutzer verschoben werden, wodurch auch der im Modell sichtbare Ausschnitt der Visualisierung verändert wird.

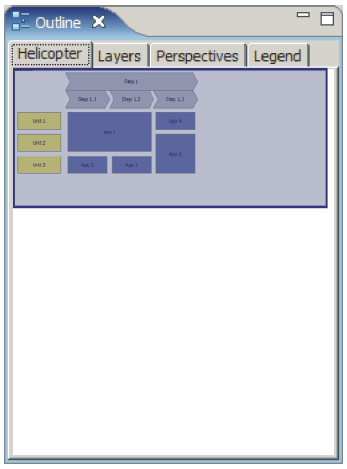



Abbildung 3.9: Benutzerschnittstelle - Helikopterperspektive

**AB13** *Auswahlrechteck zoomen:* Der Benutzer kann die Ausmaße des Auswahlrechtecks in der Helikopterperspektive verändern und so auf den Zoomstufe der Visualisierung im Modell Einfluss nehmen.

### 3.4.4 Benutzerschnittstelle - Legende

In der Legende (siehe Abbildung 3.10) wird die Legende für die aktuelle Visualisierung aus dem Modell angezeigt. Die Legende einer Visualisierung ist dabei ähnlich der Legende in der Kartographie [HGM02] als eine Beschreibung der verwendeten Symbole, Signaturen und Farben definiert. Der Benutzer kann keine Interaktionen mit der Legende durchführen, allerdings unterstützt die Ansicht der Legende Mehrsprachigkeit, d.h. es kann für verschiedene Spracheinstellungen des Werkzeugs oder für verschiedene Lokalisierungen eine andere Beschreibung hinterlegt sein.

**AB14** *Legende betrachten:* Der Benutzer kann eine Übersicht der im Modell verwendeten Symbole und Darstellungsregeln mit den entsprechenden Bedeutung betrachten.

### 3.4.5 Benutzerschnittstelle - Kartenschichten

In der Ansicht der Kartenschichten (siehe Abbildung 3.11) werden die Kartenschichten (vergleiche Abschnitt 2.4) der aktuellen Visualisierung aus dem *Modell* in ihrer momentanen Reihenfolge angezeigt. In dieser Ansicht sollen dabei folgende Operationen unterstützt werden:

**AB15** *Betrachten des Schichtenüberblicks:* Die Schichten der Visualisierung aus dem *Modell* werden in der Reihenfolge ihrer Schichtung (von »unten« nach »oben«) geord-

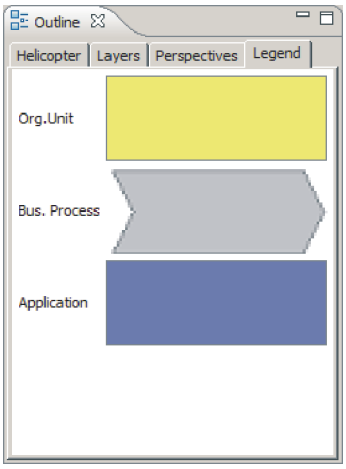



Abbildung 3.10: Benutzerschnittstelle - Legende

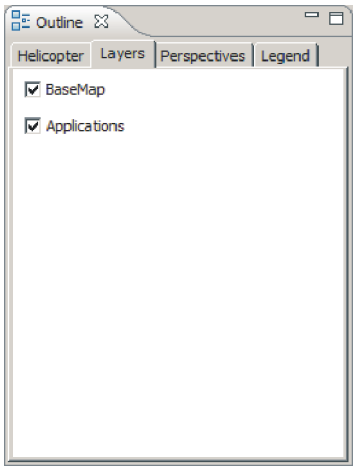



Abbildung 3.11: Benutzerschnittstelle - Kartenschichten

net zusammen mit ihren entsprechenden Namen angezeigt und ihren Sichtbarkeitsstati (»eingeblendet« oder »ausgeblendet«) angezeigt.

**AB16** *Anzeigen/Verbergen einer Schicht:* Eine Schicht kann ausgewählt und ihr zugehöriger Sichtbarkeitsstatus geändert werden. Die Symbole auf der Schicht werden analog im Modell ein- bzw. ausgeblendet, das momentan in den *Kartenperspektiven* ausgewählte Element ändert sich analog.

**AB17** *Umordnen der Schichtenfolge:* Eine Schicht kann »nach vorne« oder »nach hinten« sortiert werden, wodurch die korrespondierenden Symbole im Modell entsprechend versetzt werden. Dadurch ist es möglich nachträglich in den Aufbau der Visualisierung hinsichtlich der Überdeckung von Symbolen einzugreifen.

### 3.4.6 Benutzerschnittstelle - Kartenperspektiven

In der Ansicht der Kartenperspektiven (siehe Abbildung 3.12) werden die von der aktuellen Visualisierung aus dem *Modell* unterstützten Perspektiven (vergleiche Abschnitt 3.1.5) angezeigt. Dabei kann jede Visualisierung auch eine »Nullperspektive« unterstützen, die immer dann selektiert ist, wenn die in der Ansicht der *Kartenschichten* als sichtbar gewählten Schichten nicht zu einer der bisher definierten Perspektiven korrespondieren. Die in dieser Ansicht unterstützten Operationen sind dabei wie folgt definiert:

**AB18** *Betrachten der Perspektivenübersicht:* Die Perspektiven der Visualisierung aus dem *Modell* zuzüglich der »Nullperspektive« werden in alphabetischer Reihenfolge ihrer Benennung geordnet zusammen mit ihrem Status (»gewählt« oder »nicht-gewählt«) angezeigt.

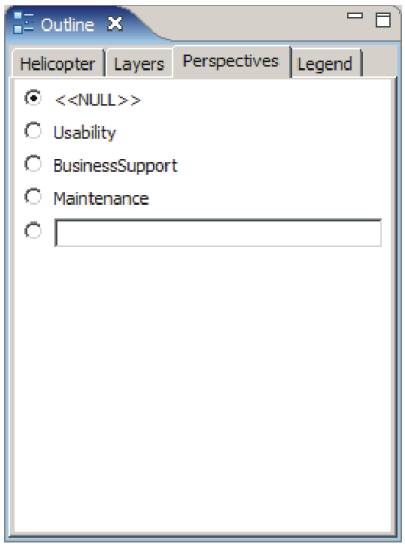



Abbildung 3.12: Benutzerschnittstelle - Kartenperspektiven

**AB19** *Aktivieren einer Perspektive:* Zu jedem Zeitpunkt der Anzeige einer Visualisierung im *Modell* ist eine Perspektive als aktiv markiert. Diese Markierung kann durch die Auswahl anderer *Kartenschichten* indirekt oder durch eine Klick auf eine Kartenperspektive direkt verändert werden. Letztgenannter Klick wiederum nimmt Einfluss auf die als sichtbar ausgewählten *Kartenschichten*.

**AB20** *Erstellen einer Perspektive:* Ist die ≫Nullperspektive≪ die aktuell gültige Perspektive, so kann der Benutzer über ein Kontextmenü eine neue Perspektive schaffen, welche genau die aktuell sichtbaren Schichten enthält, und kann diese Perspektive (über die *Eigenschaften*) mit einem Namen und einer Beschreibung versehen.

**AB21** *Löschen einer Perspektive:* Jede Perspektive mit Ausnahme der ≫Nullperspektive≪ kann durch den Benutzer über ein Kontextmenü gelöscht werden. Die korrespondierenden Schichten bleiben dabei aktiv, die Perspektive wechselt auf die ≫Nullperspektive≪.

### 3.4.7 Benutzerschnittstelle - Datenbaum

Im Datenbaum (siehe Abbildung 3.13) werden die Informationsobjekte aus der zur aktuellen Visualisierung gehörigen Kopie des semantischen Modells angezeigt. Als Gruppierungsmerkmal dienen dabei die korrespondierenden Klassen des Informationsmodells. Für die Benennung der Objekte werden dabei die Werte von Schlüsselattributen, wie im Informationsmodell definiert, verwendet. Als Operationen werden von der Datenansicht folgende unterstützt:

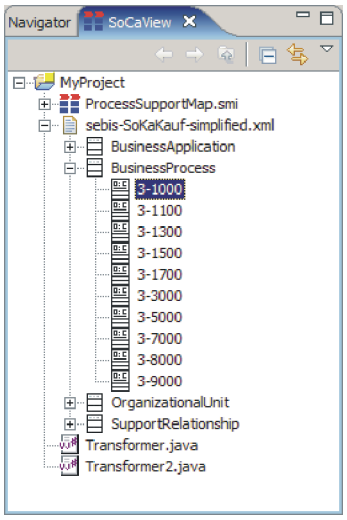



Abbildung 3.13: Benutzerschnittstelle - Datenbaum

**AB22** *Betrachten des Datenbaums:* Der Benutzer kann im Datenbaum die entsprechenden Informationsobjekte, gruppiert nach ihren Klassen navigieren.

**AB23** *Verbergen/Anzeigen der Instanzen einer Informationsmodellklasse:* Die Instanzen einer Klasse können durch Linksklick auf ein Symbol bei der entsprechenden Klasse »aus-«, bzw. »eingeklappt« werden.

**AB24** *Auswählen eines Informationsobjekts:* Eine Instanz im Datenbaum kann durch Linksklick als ausgewählt markiert werden. Diese Selektion nimmt Einfluss auf die Anzeige der *Eigenschaften*, sowie des *Modells*.

**AB25** *Auswählen einer Informationsmodellklasse:* Eine Klasse im Datenbaum kann durch Linksklick als ausgewählt markiert werden. Diese Selektion nimmt Einfluss auf die Anzeige der *Eigenschaften*. In dieser kann dann das »Benennungsattribut« für die Einträge in der Baumansicht ausgewählt werden; standardmäßig sind die Schlüsselattribute aus dem Informationsmodell gesetzt.

**AB26** *Anlegen einer neuen Instanz einer Informationsmodellklasse:* Durch ein Kontextmenü wird die Erstellung einer neuen Instanz einer ausgewählten Informationsmodellklasse in der zur aktuellen Visualisierung gehörigen Kopie des semantischen Modells angestoßen. Optional wird anschließend der Fokus auf die Eigenschaften gesetzt, wenn der Benutzer weitere Daten eingeben muss. Eine Visualisierung des neu entstandenen Objekts im aktuellen *Modell* wird dadurch nicht erzeugt.

**AB27** *Löschen eines Informationsobjekts:* Eine Instanz im Datenbaum kann durch ein Kontextmenü aus der zur aktuellen Visualisierung gehörigen Kopie des semantischen Modells gelöscht werden. Ein eventuell dieses Objekt repräsentierendes Symbol in der aktuellen Visualisierung im *Modell* bleibt dabei unberührt.

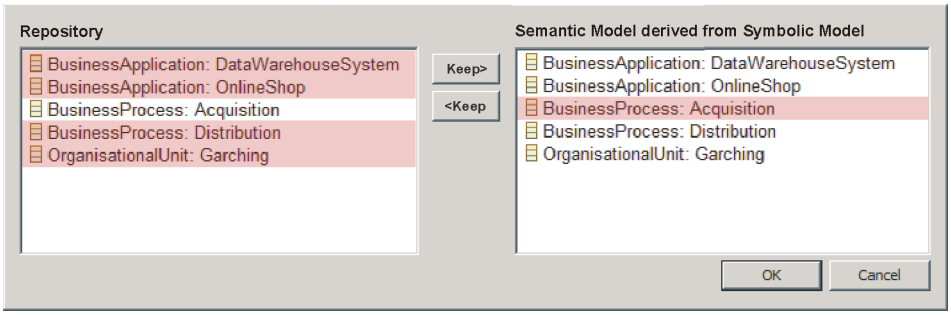



Abbildung 3.14: Benutzerschnittstelle - Synchronisierungsdialog

### 3.4.8 Benutzerschnittstelle - Synchronisierungsdialog

Werden Veränderungen an den zu konkreten Visualisierungen gehörigen Kopien des semantischen Modells, z.B. durch Benutzerinteraktion auf den Visualisierungen an sich (vgl. **AB6b**), auf den Eigenschaften (vgl. **AB9**) oder auf dem zugehörigen Datenbaum (vgl. **AB26**), vorgenommen, so betreffen diese zuerst einmal nur die Kopie an sich. Im Rahmen des Werkzeugs steht jedoch der Synchronisierungsdialog (siehe Abbildung 3.14) zur Verfügung, der es dem Benutzer erlaubt veränderte Kopien des semantischen Modells mit dem ursprünglichen semantischen Modell z.B. aus einer zentralen Datenhaltung abzugleichen. Dabei kann der Benutzer das *führende Modell* bei jeder Veränderung getrennt wählen, wobei eine Konsistenzprüfung durch das System angeschlossen wird. Im Rahmen dieses Dialogs werden folgende Operationen unterstützt:

**AB28** *Betrachten der Unterschiede zwischen den Modellen:* Hier kann der Benutzer sehen, welche Unterschiede zwischen dem kopierten semantischen Modell und dem ursprünglichen Modell bestehen. Dabei werden entsprechend hinzugekommene, bzw. gelöschte Elemente auf beiden Seiten hervorgehoben und veränderte Attributwerte und Assoziationen markiert.

**AB29** *Überschreiben des kopierten Modells:* Damit überschreibt der Benutzer den gewählten Wert oder Eintrag in der Kopie mit dem Wert aus dem ursprünglichen Modell. Die Visualisierung wird dadurch als ≫ungültig≪ markiert.

**AB30** *Überschreiben des ursprünglichen Modells:* Dadurch überschreibt der Benutzer den markierten Wert oder Eintrag im ursprüngliche Modell durch den entsprechenden Wert aus der Kopie.

### 3.4.9 Benutzerschnittstelle - Hauptmenü und zentrale Einstellungen

Das Hauptmenü des Werkzeugs erlaubt den Zugriff auf zentrale Funktionen und Einstellungen, passt sich dabei aber der aktuellen Auswahl des Benutzers an. Wesentliche

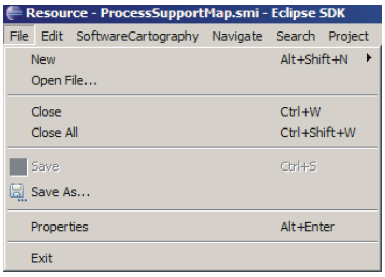



Abbildung 3.15: Benutzerschnittstelle - Hauptmenüleiste

grafische Bestandteile sind dabei die Menüleiste (siehe Abbildung 3.15) und der Dialog mit den Voreinstellungen (siehe Abbildung 3.16). Durch sie werden folgende Operationen unterstützt:

**AB31** *Speichern:* Der Benutzer kann ein verändertes symbolisches oder semantisches Modell abspeichern.

**AB32** *Exportieren:* Der Benutzer kann ein ausgewähltes symbolisches Modell in eine Datei, z.B. im SVG-, PNG- oder PDF-Format exportieren (vergleiche zu den Formaten **ANF2**).

**AB33** *Verändern der Grundeinstellungen:* Hier kann der Benutzer das zugrunde liegende Informations-, bzw. Visualisierungsmodell auswählen, indem er eine Datei, welche das zu verwendende Modell enthält und einem gegebenen Format gehorcht, referenziert.

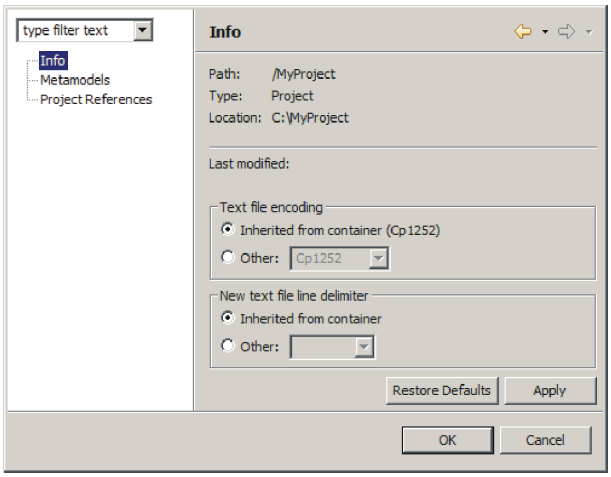



Abbildung 3.16: Benutzerschnittstelle - Dialog für Voreinstellungen

# Kapitel 4

# Verfahren zur Generierung von Softwarekarten

Im Kontext der in Kapitel 2 vorgestellten Problemdomäne haben Lankes et al. [LMW05a] ein Verfahren skizziert, welches die automatisierte Generierung von Visualisierungen, wie den Softwarekarten, aus den in Unternehmen gepflegten semantischen Modellen ermöglicht. Dieses Verfahren zielt dabei auf die folgenden Erfordernisse in diesem Umfeld ab:

- Anwendbarkeit für verschiedene Informationsmodelle
- Flexibilität der Art der Visualisierungen
- Anpassbarkeit der verwendeten Symbole

Das von Lankes et. al. vorgestellte Verfahren wurde in der Folge z.B. von Schweda in [Sc05a] auf seine Umsetzbarkeit hin untersucht und prototypisch in einem Werkzeug realisiert. Dieses Verfahren ist nun auch das Verfahren der Wahl für die vorliegende Arbeit, seine prinzipielle Funktionsweise wird in Abschnitt 4.1 vorgestellt. Die Abschnitte 4.2 bis 4.4 gehen auf die Kernkonzepte des Verfahrens exemplarisch ein.

## 4.1 Verfahrensbeschreibung

Das Verfahren von Lankes et. al. basiert auf der Annahme der Existenz zweier komplementärer, jedoch eng verbundener Modelle. Dies ist zum einen das in Abschnitt 2.3 vorgestellte *semantische Modell*, welches die Informationen über die Anwendungslandschaft beinhaltet. Zum anderen existiert ein *symbolisches Modell*, durch welches eine konkrete Visualisierung, wie z.B. die Softwarekarte in Abbildung 2.7, beschrieben wird. Dabei beschreibt ein symbolisches Modell nicht nur die Symbole auf einer Visualisierung, wie z.B. Rechtecke oder Chevren, sondern auch deren graphische Beziehungen, wie z.B. Nichtüberschneidung, durch einen Satz von Regeln. Die unterstützten Symbole und Regeln werden

Symbolisches Modell

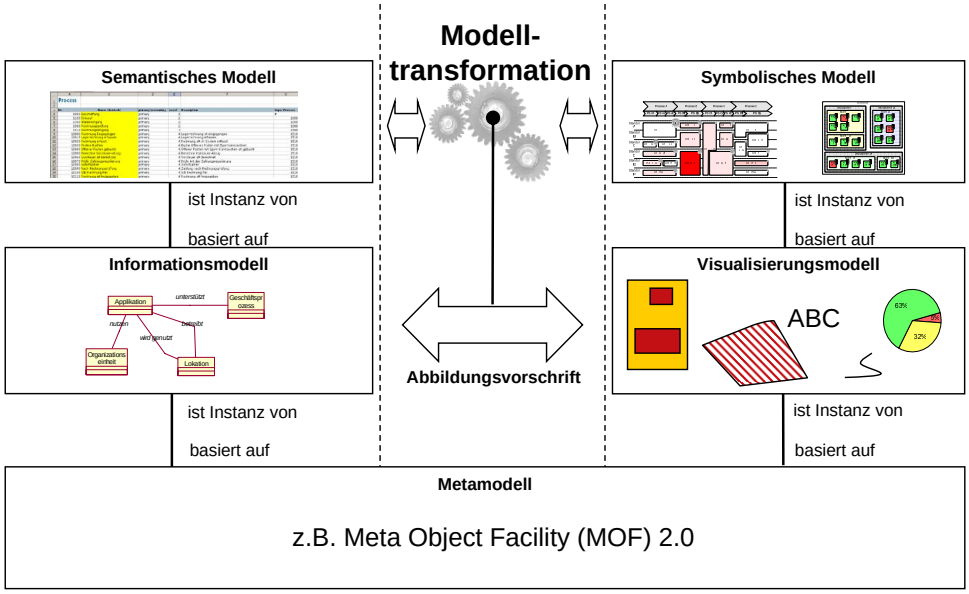



Abbildung 4.1: Schema des Verfahrens zur Generierung von Visualisierungen (eigene Darstellung nach [LMW05a])

dabei durch das dem symbolischen Modell zugrunde liegende Metamodell, das sogenannte *Visualisierungsmodell* (siehe Abschnitt 4.2), definiert.

Die enge Kopplung zwischen semantischem und symbolischem Modell beruht auf der Definition eines »Blickwinkels«, unter welchem die im semantischen Modell dokumentierte Anwendungslandschaft beschrieben ist und bildet einen Teil des Konzepts *Viewpoint* aus dem Standard IEEE 1471 [IE00], wie in Abschnitt 2.4 vorgestellt. Lankes et. al. wenden bei der Beschreibung der Kopplung, und damit für einen Teil der Definition des entsprechenden Viewpoints, das Verfahren der *Modelltransformation* (siehe Abschnitt 4.3) an. Die Kopplung wird konkret durch Transformationsregeln zwischen Konzepten aus dem Informations- und dem Visualisierungsmodell spezifiziert, z.B. könnte eine konkrete Regel bestimmen, dass »betriebliche Anwendungen« auf »Rechtecke« transformiert werden, welche wiederum mit dem »Namen« der entsprechenden Anwendung in einem »Textfeld« versehen sind. Durch die Anwendung vorstehender Regel werden alle Informationsobjekte aus dem semantischen Modell, die betriebliche Anwendungen darstellen, in Visualisierungsobjekte im symbolischen Modell übersetzt, welche Rechtecke repräsentieren.

Das vorgestellte Verfahren (siehe Abbildung 4.1) nimmt darüber hinaus im Sinne der Realisierbarkeit die Existenz eines gemeinsamen Metamodells für das Informations- und das Visualisierungsmodell an. Dieses Metamodell (siehe Abschnitt 4.3) sollte dabei die Konzepte der objektorientierten Modellierung unterstützen und muss seinerseits wiederum

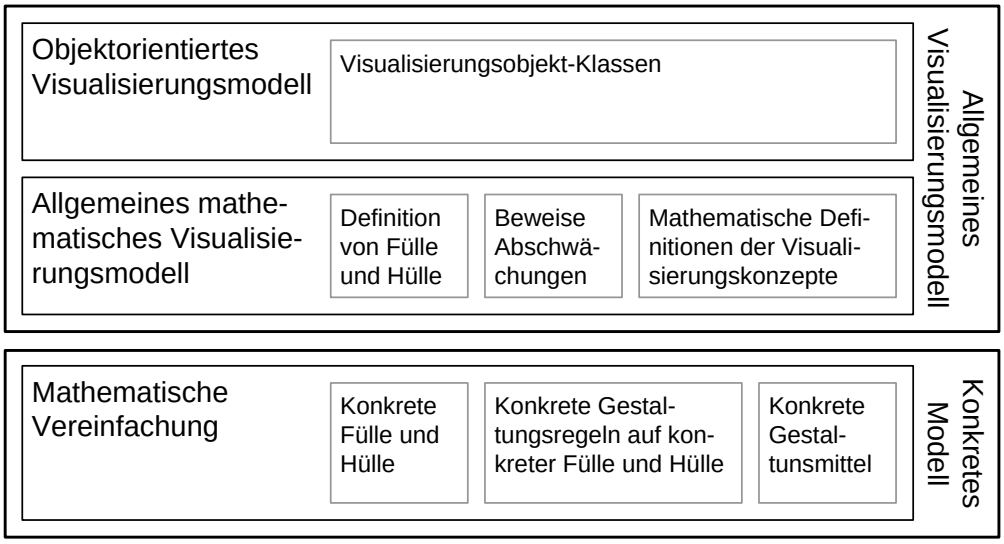



Abbildung 4.2: Dreischichtiges Visualisierungsmodell (eigene Darstellung nach [Er06])

durch die verwendete Modelltransformation unterstützt werden. Die Mächtigkeit verschiedener Metamodellierungssprachen sowie deren Unterstützung durch verschiedene Modelltransformationssprachen wurde von Buckl in [Bu05] im Vorfeld der vorliegenden Arbeit untersucht.

## 4.2 Visualisierungsmodell

Dreischichtiges Visualisierungsmodell

Eine Realisierung des von Lankes et. al. in [LMW05a] skizzierten *Visualisierungsmodells* findet sich bei Ernst et. al. in [Er06] in Form eines *dreischichtigen Visualisierungsmodells* (siehe Abbildung 4.2). Ein derartig umfangreiches Modell wird notwendig, um die verschiedenen Anforderungen, denen ein solches Modell unterliegt, adäquat zu adressieren; so sollte ein Visualisierungsmodell nach Ernst et. al.

- *einfach zu handhaben* sein, mit Hinblick auf die Verwendung bei der Spezifikation von Visualisierungen,
- *ausreichend formal definiert* sein, um eine geeignete Grundlage für die Generierung von Visualisierungen mittels eines Algorithmus zu bieten,
- *flexibel* sein, mit Hinblick auf die Einführung neuer Symbole und Regeln,
- *anpassbar* sein, um Verbesserungen an den Layoutverfahren zu erlauben, ohne das gesamte Modell ändern zu müssen und

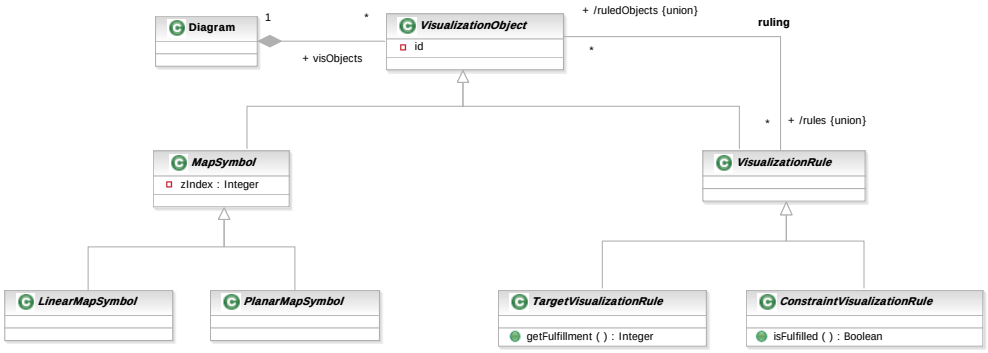



Abbildung 4.3: Zentrale Konzepte des Visualisierungsmodells (entommen aus [Er06])

- *berechenbar* sein, mit Hinblick auf die Entwicklung von Layoutverfahren.

Der mehrschichtige Aufbau des Visualisierungsmodells von Ernst et. al. erfüllt die oben geäußerten Anforderungen durch unterschiedliche Konzepte auf den verschiedenen Ebenen. Diese Ebenen werden in den folgenden Abschnitten vorgestellt.

### 4.2.1 Objektorientiertes Visualisierungsmodell

Auf der Ebene des *objektorientierten Visualisierungsmodells* werden die graphischen Konzepte anhand von Klassen, Attributen und Assoziationen eingeführt und können unter Verwendung der Notation für Klassendiagramme entsprechend dargestellt werden. Zentrale Konzepte auf dieser Ebene sind (siehe Abbildung 4.3):

**Diagramm** (*Diagram*) Ein Diagramm ist die Visualisierung als Ganzes und besteht aus den Visualisierungsobjekten, die an dieser Visualisierung beteiligt sind. Darüber hinaus könnte ein spezialisiertes Diagramm noch Metainformationen, z.B. über den Erschaffer des Diagramms oder sein Erstellungsdatum enthalten.

**Visualisierungsobjekt** (*Visualization Object*) Ein Visualisierungsobjekt ist ein Objekt, welches an einem Diagramm beteiligt ist. Dabei kann zwischen zwei Arten von Visualisierungsobjekten, den Gestaltungsmitteln und den Gestaltungsregeln unterschieden werden.

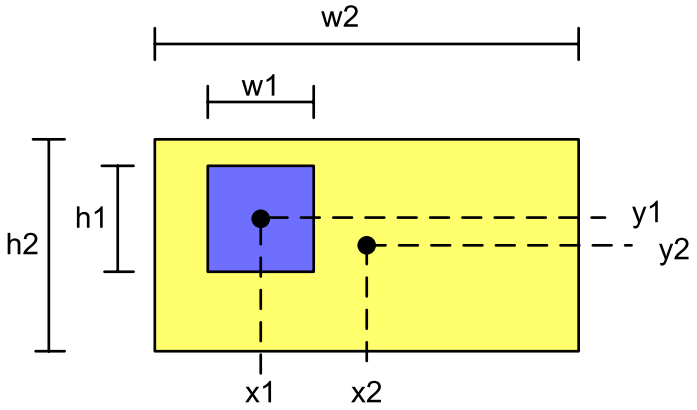
**Gestaltungsmittel** (*Map Symbol*) Ein Gestaltungsmittel ist ein graphisches Element, welches in einem Diagramm verwendet werden kann, z.B. ein Chevron. Die Instanz eines Gestaltungsmittels stellt dabei ein konkretes Symbol auf einer konkreten Visualisierung dar.

**Gestaltungsregel** (*Visualization Rule*) Eine Gestaltungsregel ist eine Regel, welche eine oder mehrere Instanzen von Visualisierungsobjekten hinsichtlich ihrer Positionierung oder anderer Aspekte ihrer Darstellung miteinander in Beziehung setzen kann, z.B. könnte eine konkrete Instanz einer Regel *Nesting* zwei Visualisierungsobjekte als ineinander geschachtelt festlegen.

**Regel-Beziehung** (*ruling*) Die Regel-Beziehung stellt eine bidirektionale Beziehung zwischen den Instanzen »regelnder« Gestaltungsregeln und der von ihnen »geregelten« Visualisierungsobjektinstanzen dar.

Gestaltungsvariablen

Von diesen allgemeinen objektorientierten Konzepten können spezielle Gestaltungsmittel und Gestaltungsregeln durch Subklassenbildung realisiert werden, die Attribute der Visualisierungsobjekte (auch als *Gestaltungsvariablen* bezeichnet) können in den Subklassen entsprechend ergänzt werden. Bei der Spezialisierung der Gestaltungsmittel führen Ernst et. al. eine Kategorisierung in *flächenhafte* (*planar*) und *linienartige* (*linear*) Gestaltungsmittel (*Map Symbol*) ein, die u.a. hinsichtlich der auf sie anwendbaren Gestaltungsregeln, aber auch hinsichtlich ihrer Handhabung in einer konkreten Visualisierung unterschieden werden müssen.




$$x_1 - \frac{w_1}{2} > x_2 - \frac{w_2}{2} \wedge x_1 + \frac{w_1}{2} < x_2 + \frac{w_2}{2} \wedge$$

$$y_1 - \frac{h_1}{2} > y_2 - \frac{h_2}{2} \wedge y_1 + \frac{h_1}{2} < y_2 + \frac{h_2}{2}$$

Abbildung 4.4: Auswirkungen der Gestaltungsregel »Nesting« auf zwei Gestaltungsmittelinstanzen

Bei der Spezialisierung der Gestaltungsregeln führen Ernst et. al. eine Kategorisierung in *Bedingungsregeln* (*Constraint visualization rule*) und *Zielregeln* (*Target visualization rule*) ein, die hinsichtlich ihrer Erfüllung unterschieden werden können. Bedingungsregeln unterscheiden zwei Erfüllungszustände (*erfüllt* und *nicht-erfüllt*). Durch die Forderung nach Erfüllung der Bedingungsregeln wird der Wertebereich der Gestaltungsvariablen entsprechend eingeschränkt. Ein Beispiel dazu findet sich in Abbildung 4.4. Zielregeln geben ihren Erfüllungsgrad auf einer Ordinalskala an, können also *mehr* oder *weniger* erfüllt sein. Dadurch wird der Bereich der gültigen Werte für die Gestaltungsvariablen nicht eingeschränkt, sondern lediglich gewisse Belegungen höher gewertet als andere. Generell dient die Ebene des objektorientierten Visualisierungsmodells der einfachen Handhabbarkeit und der Flexibilität des Gesamtmodells, z.B. in Hinblick auf die Definition von Modelltransformationen (siehe Abschnitt 4.3) oder die Erweiterung durch neue Visualisierungskonzepte.

### 4.2.2 Allgemeines mathematisches Visualisierungsmodell

Auf der Ebene des *allgemeinen mathematischen Visualisierungsmodells* werden die graphischen Konzepte durch Mengen, Funktionen und Prädikate formalisiert und ergänzen so die einfach handzuhabenden, aber nicht sehr formal definierten Konzepte der objektorientierten Ebene. Die Darstellung der Gestaltungsmittel und Gestaltungsregeln erfolgt auf der allgemeinen mathematischen Ebene in der Formelnotation, allerdings abstrakt und allgemein ohne konkrete Berechnungsvorschriften. Als zentrale Konzepte auf dieser Ebene sind definiert:

**Zeichenfläche:** Diese wird definiert als eine Menge von Punkten $\mathbb{K}$.

**Präsentationsinformation:** Diese wird definiert über eine Menge $\mathbb{P}$ der gültigen Werte für diese Information, z.B. durch die Menge der darstellbaren Farben.

**Gestaltungsmittel:** Dieses wird wie folgt definiert als ein Tupel $(p, r) \in \mathbb{G}$ zweier Funktionen: $p : \mathbb{K} \to \mathbb{B}$[1] und $r : \mathbb{K} \to \mathbb{P}$. Die Menge $\mathbb{G}$ ist dabei die Menge aller Gestaltungsmittel.

**Bedingungsregel:** Diese wird als eine Funktion wie folgt definiert: $\lambda : \mathbb{I}(\mathbb{G}) \to \mathbb{B}$[2].

**Zielregel:** Diese wird als eine Funktion wie folgt definiert: $\lambda : \mathbb{I}(\mathbb{G}) \to \mathbb{A}$, wobei es sich bei $\mathbb{A}$ um eine Menge von ordinal skalierten Werten handelt.

Anhand dieser Definitionen lassen sich die Bezeichnungen *Bedingungsregel* und *Zielregel* erschließen, handelt es sich bei den Bedingungsregeln um Bedingungen, welche die Menge der erlaubten Belegungen von Gestaltungsvariablen einschränken, wohingegen die Zielregeln Beitrag zu einer Art Zielfunktion leisten, welche über der eingeschränkten Menge zu einem optimalen Wert der Erfüllung gebracht werden soll.

[1] Dabei bezeichnet $\mathbb{B}$ die Menge der Wahrheitswerte.

[2] Der Operator $\mathbb{I}(\mathbb{G}_1)$ für $\mathbb{G}_1 \subseteq \mathbb{G}$ bezeichnet die Menge aller Instanzen von Gestaltungsmitteln aus $\mathbb{G}_1$.

Im Hinblick auf die Komplexität der algorithmischen Handhabung von Prädikaten und Funktionen der oben dargestellten allgemeinen Gestalt, geben Ernst et. al. eine Möglichkeit die zuhandhabenden Terme zu vereinfachen. Dies wird durch die Abschwächung der von Bedingungs- und Zielregeln aufgestellten Forderungen ermöglicht, welche sich auf die zwei weitere allgemein definierte Konzepte stützt. Dies sind die Konzepte, *Fülle* und *Hülle* einer Gestaltungsmittelinstanz, die geeignet gewählt zu »einfacheren« Prädikaten und Funktionen führen. Ernst et. al. bezeichnen die Verwendung von Fülle und Hülle allgemein als *Konkretisierung* und geben ein Beispiel für eine Konkretisierung an, welche in Abschnitt 4.4.1 beschrieben und im Rahmen des in dieser Arbeit zu realisierenden Werkzeugs verwendet wird.

Fülle und Hülle

## 4.3 Modelltransformation und Metamodell

Wie Buckl in [Bu05] zeigt, basieren Modelltransformationssysteme auf Metamodellen, die sich zum Teil explizit an Standards, wie z.B. der MOF [OM06] der Object Management Group (OMG), orientieren. Da sowohl das Informationsmodell als auch das oben vorgestellte Visualisierungsmodell als Instanzen des Metamodells aufgefasst werden können, muss das Metamodell die in den entsprechenden Modellen verwendeten Konzepte beinhalten. Ein *gemeinsames Metamodell* ist dabei zwar keine notwendige Voraussetzung für eine Modelltransformation, stellt jedoch in diesem Zusammenhang eine deutliche Vereinfachung dar, da die Transformation und die in ihr enthaltenen Modelloperationen, wie z.B. »new« oder »getClass«, sowohl bei Quell- als auch bei Zielmodellen bekannt und einheitlich definiert sind.

Gemeinsames Metamodell

Als wesentliche Anforderung an ein gemeinsames Metamodell muss die konsistente Unterstützung der Konzepte objektorientierter Modellierung erachtet werden, da die aus der Praxis gewonnenen Informationsmodelle (siehe Abschnitt 2.3) in diesem Paradigma modelliert worden sind. Den Umfang der verwendeten Konzepte betreffend, erscheint es jedoch besonders im Bereich der Informationsmodellierung wichtig zwischen Modellen aus verschiedenen »Phasen« der objektorientierten Modellierung zu unterscheiden. So können bei einem in der Analysephase (vgl. hierzu [Ba01]) erstellten Modell Konzepte, wie z.B. Assoziationsklassen, enthalten sein, welche im Rahmen einer fortgesetzten Konkretisierung des Modells in der Designphase dann semantisch äquivalent umgeformt werden. Aufbauend auf dieser Erkenntnis und auf die bei Buckl in [Bu05] durchgeführte eingehenden Analyse von Metamodellierungssprachen wird für die Realisierung des in der vorliegenden Arbeit zu beschreibenden Werkzeugs ein konkretes gemeinsames Metamodell ausgewählt und durch geeignete Softwarekomponenten unterstützt.

Die enge Bindung zwischen den Modellen wird, wie in Abschnitt 4.1 dargestellt, durch die Verwendung von Modelltransformation erzielt. Bei der *Modelltransformation* handelt es sich um eine Technologie, welche es erlaubt eines oder mehrere *Quellmodelle* in eines oder mehrere *Zielmodelle* zu überführen. Dies geschieht durch die Anwendung von *Transformationsregeln*, wobei die Menge der an einer konkreten Überführung beteiligten derartigen

Modelltransformation

Regeln die Beschreibung der Modelltransformation ergeben. Modelltransformationen finden in weiten Bereichen der Softwaretechnik Anwendung, z.B. bei der Erzeugung von SQL-DDL-Anweisungen aus einem in UML formulierten Datenbankschema.

Im Umfeld des in den letzten Jahren an Bedeutung gewinnenden Paradigmas der *Model Driven Architecture (MDA)* [OM01] entwickeln sich dabei Sprachen und Werkzeuge, welche bei stereotypen Transformationen, z.B. zwischen verschiedenen Abstraktionsniveaus der Modellierung, durch Automatisierung Unterstützung bieten sollen. Ein Vergleich verschiedener Sprachen und der ihnen zugrunde liegenden Paradigmen für die Regeldefinition findet sich bei Buckl in [Bu05].

Eine Entscheidung, nur eine konkrete Sprache im Rahmen des hier zu entwickelnden Werkzeugs zu unterstützen, kann allerdings auch auf Basis der Arbeit von Buckl nicht getroffen werden. Dies begründet sich u.A. auch dadurch, dass eine genaue Beschreibung der Anforderungen für die Mächtigkeit der Modelltransformation zwischen semantischem und symbolischen Modell zum jetzigen Zeitpunkt noch nicht existiert. Nichtsdestotrotz formulieren Ernst et. al. in [Er06] eine allgemeine Modelltransformation als Bindeglied zwischen semantischem und symbolischem Modell, wobei die in der Transformation enthaltenen Transformationsregeln zur Definition des *Viewpoints* (siehe Abschnitt 2.4) einer Visualisierung gehören.

## 4.4 Vom symbolischen Modell zur Grafik

Bei der Modelltransformation entsteht aus einem semantischen Modell ein symbolisches Modell mit Instanzen von Visualisierungsobjekten und deren Beziehungen untereinander. Die konkrete Visualisierung wiederum impliziert eine Belegung der Gestaltungsvariablen in der Art, dass die Gestaltungsregeln soweit möglich erfüllt sind (siehe Abschnitt 4.2.2). Soweit möglich bedeutet dabei, dass möglichst viele, optimalerweise alle[3] Bedingungsregeln erfüllt sind und die Zielregeln dabei den optimalen Wert annehmen. Die Ermittlung einer derartigen Belegung für die Gestaltungsvariablen, also das konkrete *Layouting* der Visualisierung ist jedoch auf Basis des allgemeinen Visualisierungsmodells nicht effizient möglich. Ernst et. al. formulieren eine mögliche Konkretisierung des Visualisierungsmodells basierend auf den Abschwächungen der Gestaltungsregeln unter Verwendung der Konzepte von Fülle und Hülle. Auf dieser kann die Berechnung eines Layouts effizient erfolgen. Diese Konkretisierung, die auch ihm Rahmen des hier zu erstellenden Werkzeugs implementiert werden soll, wird in den beiden folgenden Abschnitten 4.4.1 und 4.4.2 dargestellt.

Layouting

### 4.4.1 Konkretes mathematisches Visualisierungsmodell

Als eine mögliche Erfüllung der Anforderungen von Fülle und Hülle führen Ernst et. al. die Konzepte der *rechteckigen Fülle* (siehe Abbildung 4.5) und *rechteckigen Hülle*

[3] Ernst et. al. weisen in [Er06] darauf hin, dass ein symbolisches Modell implizit widersprüchlich sein kann, so dass ggf. unmöglich ist, alle Bedingungsregeln zugleich zu erfüllen.

Abbildung 4.5: Fülle (grau) einer Gestaltungsmittelinstanz

Abbildung 4.6: Hülle (grau) einer Gestaltungsmittelinstanz

(siehe Abbildung 4.6) ein. Dabei handelt es sich bei einer rechteckigen Fülle um das flächenmäßig größte Rechteck, welches komplett von der gegebenen Gestaltungsmittelinstanz eingeschlossen wird. Analog definiert sich die rechteckige Hülle als das Rechteck kleinster Fläche, welches die gegebene Gestaltungsmittelinstanz komplett umschließt. Die in Abschnitt 4.2.2 angedeuteten Abschwächungen der Bedingungsregeln auf Füllen und Hüllen treffen, wie bei Ernst et. al. allgemein gezeigt, auch auf ihre rechteckigen Realisierungen zu, auf denen wiederum die konkrete Berechnung von Prädikaten, z.B. der Überschneidung, auf Ungleichungen über den Parametern der entsprechenden Rechtecke zurückgeführt werden kann. So kann aus dem Satz von Bedingungsregeln einer konkreten Visualisierung ein Ungleichungssystem über einer Teilmenge der Gestaltungsvariablen der entsprechenden Gestaltungsmittelinstanzen errichtet werden. Die Zielregeln bilden eine Sammlung von Zielfunktionen, deren Werte über dem durch das Ungleichungssystem eingeschränkten Wertebereich der Gestaltungsvariablen optimiert werden müssen. Nach dieser Umwandlung ist die Problemstellung des *Layoutings* auf ein Problem der Optimierung zurückgeführt.

### 4.4.2 Layouting als Optimierungsproblem

Das Optimierungsproblem, welches durch Anwendung einer Konkretisierung des Visualisierungsmodells aus einem symbolischen Modell entsteht, konkret zu lösen, bedeutet das Layout der korrespondierenden Visualisierung zu berechnen. Dies stellt selbst bei der »einfachen« Konkretisierung über rechteckige Füllen und Hüllen ein komplexes Problem dar; die durch das Ungleichungssystem formulierten Randbedingungen der Optimierung sind in den meisten Fällen nicht linear in den Werten der Gestaltungsvariablen. Die Zielfunktionen sind nicht automatisch miteinander vergleichbar, so dass das zu suchende Optimum ein *Pareto Optimum* (siehe [Zi87]) darstellt. Die daraus resultierende Komplexität kann zwar durch weitere Annahmen hinsichtlich der Konkretisierung reduziert werden, dennoch bleibt das Layouting ein Optimierungsproblem, welches z.B. mit den Methoden der nichtlinearen Optimierung angegangen werden muss.

Auf ein effizientes Lösungsverfahren für dieses Optimierungsproblem soll das Augenmerk

im Rahmen der vorliegenden Arbeit und des zu realisierenden Werkzeugs nicht gerichtet werden. Es soll lediglich anhand eines einfachen Verfahrens die effektive Lösbarkeit demonstriert werden.

# Kapitel 5

# Plattformunabhängige Architektur

In diesem Kapitel soll die Architektur, für das in dieser Arbeit zu beschreibende Werkzeug, vorgestellt werden, welches die in Kapitel 3 aufgezählten Anforderungen auf Basis des in Kapitel 4 gezeigten Verfahrens erfüllt. Die vorzustellende Architektur wird dabei von verschiedenen Blickwinkeln aus betrachtet, wobei Abschnitt 5.3 die statischen Aspekte darstellt, Abschnitt 5.4 die dynamischen Prozesse im Werkzeug an examplarischen Abläufen illustriert und Abschnitt 5.5 Aspekte von Verteilung und Betrieb beleuchtet. Dem Ganzen vorangestellt sind ein Abschnitt 5.1, der die geplante Realisierung der Lösung aus Kapitel 4 aufzeigt und ein Abschnitt 5.2, der die grundsätzlichen Architekturentscheidungen des Systems zusammenfasst und in einem konsistenten Gesamtbild darstellt.

## 5.1 Lösungsüberblick

Ausgehend von dem in Abbildung 4.1 schematisch dargestellten Verfahren zur automatisierten Generierung von Visualisierungen aus semantischen Modellen wurde von Schweda in [Sc05a] die prinzipielle Funktionsweise einer Realisierung des Verfahrens beschrieben. Diese Beschreibung der Funktionsweise und die in ihr enthaltene Gliederung des Werkzeugs in Komponenten, wie sie Ausgangspunkt der vorliegenden Arbeit war, findet sich in Abbildung 5.1. Dabei werden die verarbeiteten Modelle durch Notizen (mit eingeklappter Ecke) und die Komponenten durch Rechtecke repräsentiert, die Pfeile zwischen den verschiedenen Objekten erläutern den Datenfluss bei der Erstellung und Darstellung einer Visualisierung. Der dargestellte abstrakte Aufbau wurde zu einer abstrakten Architektur konsolidiert, die in Abbildung 5.2 vorgestellt wird. Diese Architektur operiert dabei auf folgenden Artefakten:

**Informationsmodell** ist definiert analog zu Abschnitt 2.3 als Begriffsapparat, unter dessen Verwendung Architekturbeschreibungen für Anwendungslandschaften erstellt werden können.

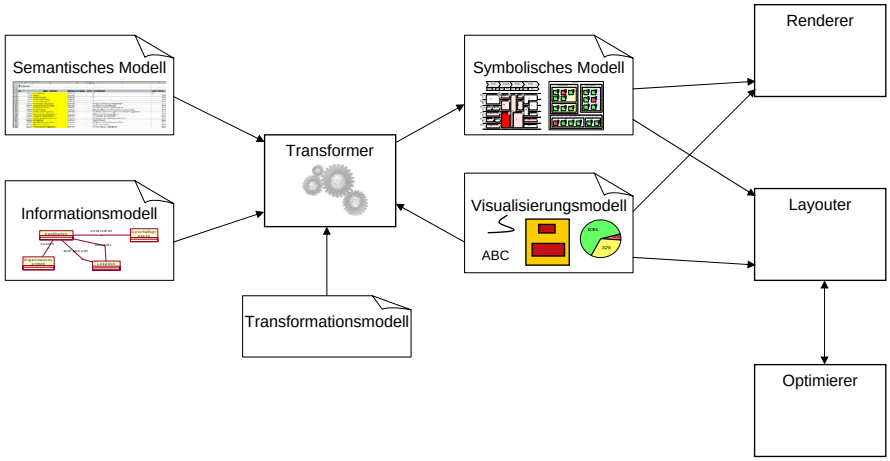



Abbildung 5.1: Abstrakter Aufbau (eigene Darstellung nach [Sc05a], an UML angelehnte Notation)

**Semantisches Modell** ist definiert analog zu Abschnitt 2.3 als Gesamtheit von Aspekten der Beschreibung der Architektur von Anwendungslandschaften, sowie als Instanz des Informationsmodells.

**Visualisierungsmodell** ist definiert analog zu Abschnitt 4.1 als Begriffsapparat, unter dessen Verwendung Visualisierungen objektorientiert und konkret berechenbar beschrieben werden können.

**Symbolisches Modell** ist definiert analog zu Abschnitt 4.1 als Gesamtheit der Symbole auf einer Visualisierung, deren graphische Beziehungen untereinander, sowie als Instanz des Visualisierungsmodells.

**Transformationsmodell** ist definiert analog zu Abschnitt 4.3 als Menge von Regeln, welche ein Quellmetamodell auf ein Zielmetamodell abbilden, sowie als Instanz von Transformationsregelklassen aus einem, durch die Transformationsfazilität definierten, Transformationsmetamodell.

**Optimierungsproblem** ist definiert analog zu Abschnitt 4.4 als eine Sammlung von Bedingungen, welche den Lösungsraum einschränken, und eine Zielfunktion, deren Wert über dem Lösungsraum optimiert werden soll.

Darüber hinaus verfügt die abstrakte Architektur über verschiedene Komponenten, die in Abbildung 5.2 mit Ausnahme des Repositorys eingeführt werden. Diese Komponenten und ihre Funktion werden hier kurz, soweit für das Verständnis der in Abschnitt 5.2 vorgestellten Architekturentscheidungen notwendig, beschrieben, da eine ausführliche Erläuterung der Funktion in Abschnitt 5.3 folgt:

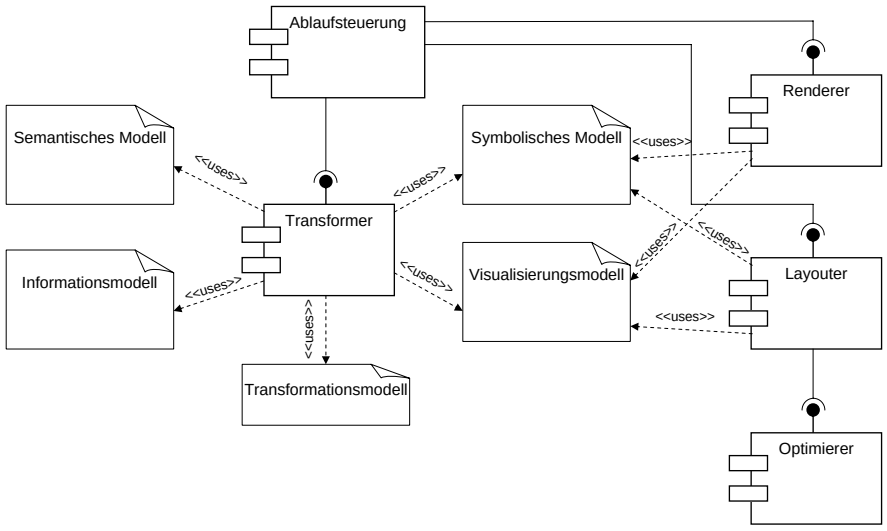



Abbildung 5.2: Abstrakte Architektur der zu entwickelnden Lösung

**Repository** dient als Speicher für Modelle aller Art.

**Ablaufsteuerung** steuert die Orchestrierung der anderen Komponenten.

**Transformer** transformiert regelbasiert ein Quell- in ein Zielmodell.

**Layouter** übersetzt zwischen symbolischem Modell und Optimierungsproblem. Dabei dient ein symbolisches Modell als Eingabe für eine Optimierung, deren Ergebnisse wiederum als Werte in Modell zurückgepflegt werden.

**Optimierer** löst ein Optimierungsproblem.

**Renderer** stellt ein gelayoutetes symbolisches Modell dar.

Aus den Eigenschaften dieser Komponenten und der Artefakte wurde eine Grundidee für die Architektur des zu realisierenden Werkzeugs gewonnen, welche im Abschnitt 5.2 vorgestellt wird.

## 5.2 Architekturentscheidungen

Im Systementwicklungsprojekt von Schweda [Sc05a] wurden am Prototyp Komponenten einer Architektur für ein Werkzeug für die Generierung von Visualisierungen von Anwendungslandschaften identifiziert. Diese Komponentengliederung wurde in weiten Bereichen in die statische Komponentenarchitektur (siehe Abschnitt 5.3) des hier zu entwickelnden Werkzeugs übernommen. Als fundamentale Erkenntnis wurden aus der Vorarbeit Ge-

meinsamkeiten der Komponenten zum einen und Besonderheiten der Choreographie der Komponenten zum anderen abgeleitet:

- Die Komponenten führen fast ausschließlich Operationen auf objektorientierten Modellen und deren Metamodellen aus.
- Es sind ≫synonyme≪ Komponenten denkbar, also Komponenten, welche identische Funktion haben, diese aber unterschiedlich realisieren.
- Es sind zusätzlich Komponenten denkbar, welche weitere Funktionalitäten realisieren, ohne dass diese bereits vorgesehen waren.
- Die Orchestrierung (Aufrufreihenfolge) der Komponenten ist nicht fix, sondern muss flexibel gestaltbar sein.
- Es sind Komponenten denkbar, die zu einem Aufruf eine typsichere Parametrisierung brauchen.

Eine grundsätzliche Architekturentscheidung, welche die oben aufgeführten Besonderheiten adressiert, ist die Gliederung des Werkzeugs in *Model-Services*. Bei einem *Model-Service* handelt es sich dabei in Anlehnung an die Definition des Begriffes *Web Service* durch das W3C in [W3C04c] um ein Softwaresystem, welches

Model-Service

- Maschine-Maschine-Interaktion auf objektorientierten Modellen unterstützt,
- Verteilungs- und Netzwerkfähigkeiten anbietet,
- Modellaustausch (und Verhandlung) auf Basis eines gemeinsamen Protokolls ermöglicht und
- Schnittstellenbeschreibung in maschinenlesbarer Form anbietet.

Die konkrete Realisierung eines Model-Service beruht auf Software- und/oder Hardwarekomponente, welche den in der Beschreibung des Service vorgestellten Dienst tatsächlich erbringen. Die Hardwarekomponenten, auf welchen die verschiedenen Realisierungen zu den verschiedenen Services betrieben werden, können verschieden physikalische Rechner sein, welche dann über ein Netzwerk interagieren. Zwischen diesen Realisierungen werden objektorientierte (Meta-)Modelle in geeigneter Repräsentation ausgetauscht. Wesentlicher Bestandteil dieses Austausches ist dabei die Verhandlung eines den beiden Realisierungen bekannten Metamodells, als dessen Instanzen die verschiedenen Modelle gesehen werden können. Dieser Austausch stellt dabei ein Einsatzszenario z.B. für XMI-serialisierte Modelle (siehe OMG XMI Spezifikation in [OM05]) als ≫Payload≪ einer SOAP-kodierten Nachricht (siehe W3C SOAP Spezifikation in [W3C04b]) dar. Die Schnittstellenbeschreibung wiederum beinhaltet dieselbe Problematik einer Metamodellverhandlung zwischen Dienstanbieter (*Service Provider*) und Dienstnutzer (*Service Requester*), so dass auch hier geeignete Verfahren auf Basis etablierter Standards zum Einsatz kommen müssen.

Ein wichtiger Aspekt bei der Definition eines Model-Services ist die Unterstützung von Transaktionen. A priori sollte jede Operation auf einem Model-Service atomar ablaufen und ihren eigenen Transaktionskontext haben. Darüber hinaus sollte es jedoch auch möglich sein, einen erweiterten und mehrere Operationen übergreifenden Transaktionskontext zu starten. Model-Services, welche diese Möglichkeit anbieten, werden nachfolgend als *transaktionale Model-Services* bezeichnet.

Transaktionaler Model-Service

Bei der Definition eines Model-Services ist die Unterstützung einer *Benachrichtigungsfunktion* ein weiterer Aspekt, d.h. ein *benachrichtigender Model-Service* ist ein Dienst, welcher eine Liste von Empfängern verwaltet, die automatisiert benachrichtigt werden, falls ein für sie relevantes Ereignis eintritt (vergleiche hierzu Web-Service Notification [OA05]).

Benachrichtigender Model-Service

Ein weiteres essentielles Element der auf Model-Services basierenden Architektur, welches speziell durch die dynamische Erweiterbarkeit des Vorrats an Diensten bedingt wird, ist eine zentrale Verwaltung der zur Verfügung stehenden Dienste, die sogenannte *Model-Service Registry*. Diese Registry übernimmt dabei einen Teil der Funktionalität, welche der sogenannte *Service Broker* in der generellen Architektur der Webservices (vgl. W3C in [W3C04c]) anbietet, beschränkt sich dabei jedoch stark auf die ≫Bekanntmachung≪ der einzelnen Model-Services und delegiert weitergehende Aspekte an die direkte Interaktion zwischen anbietenden und nachfragenden Agenten.

Model-Service Registry

Ein letztes Element stellt das Kommunikationsnetz der Model-Services dar, welches als Abstraktion einer Punkt-zu-Punkt-Verbindung zwischen den einzelnen Diensten gesehen werden kann. Dieses Verbindungsnetzwerk wird durch den sogenannten *Model-Bus* gebildet, welcher Funktionalitäten für einen Endpunkt-zu-Endpunkt Austausch von Modellen zwischen verschiedenen Model-Services zur Verfügung stellt. Darüber hinaus erfolgt die Adressierung der Model-Services über Funktionen, welche vom Model-Bus im Zusammenspiel mit der Model-Service Registry zur Verfügung gestellt werden. Spezielle Anforderungen, sowie eine Realisierung eines Model-Bus, sowie ein Einsatzszenario im Umfeld der Model Driven Architecture werden im MDDi-Projekt (siehe [Bl05]) dargestellt.

Model-Bus

Trotz der begrifflichen Nähe stellt die vorliegende Architektur keine *Service Oriented Architecture (SOA)* im engeren Sinne dar, da speziell der vom W3C in [W3C04c] geforderte Aspekt der Erbringung von ≫business-level operations≪ durch den Service als nicht gegeben gesehen werden muss. Nichtsdestotrotz sind viele Aspekte einer SOA in der dargestellten Architektur realisiert, so z.B. die *lose Kopplung*, der *nachrichtenbasierte Informationsaustausch*, die *Ortstransparenz der Dienste* und die Verwendung einer *Dienstregistrierung*.

## 5.3 Statische Komponentenarchitektur

In diesem Abschnitt werden die einzelnen Komponenten[1] des zu realisierenden Werkzeugs vorgestellt. Falls anwendbar, werden die Komponenten dabei in Gestalt eines *Model-Service* dargestellt und ihrem Ausführungskontext entsprechend hinsichtlich von Transaktionsunterstützung oder Benachrichtigungsfunktionalitäten kategorisiert. Bei der Beschreibung der Komponenten wird darüber hinaus eine Beschreibung der von der Komponente realisierten Schnittstelle gegeben. Für diese Beschreibung werden zusätzlich zu den in UML2 definierten Datentypen noch weitere Datentypen und Klassen gebraucht. Einige dieser Typen sollen, da sie häufig vorkommen, hier vorweg definiert werden, andere Typen werden bei den entsprechenden Komponenten eingeführt:

- ***Model*** ist ein objektorientiertes Modell, bestehend aus *Elements*, repräsentiert, z.B. durch einen Objektgraphen im Arbeitsspeicher oder eine geeignete Serialisierung, z.B. in XMI.
- ***Element*** ist ein Objekt in einem objektorientierten Modell, vergleichbar dem *EMOF::Reflection::Element* aus MOF.
- ***URI*** ist ein *Uniform Resource Identifier* (siehe Definition durch die Internet Engineering Task Force in [IE94]), also ein eindeutiger Identifikator für eine Resource.
- ***URL*** ist ein *Uniform Resource Locator* (siehe [IE94]), also eine tatsächlich auflösbare Adressierung im Internet.

### 5.3.1 Komponente: Repository

Das Repository stellt die zentrale objektorientierte Datenhaltung für das Werkzeug dar. Die darin gehaltenen Modelle sind in Instanzen des gemeinsamen Metamodells (siehe Abschnitt 4.3) beschrieben und lassen sich dabei als einem der folgenden Bereiche zugehörig betrachten:

**Informationsmodell:** Die Datenhaltung enthält ein Informationsmodell, dem die Beschreibung der Anwendungslandschaft gehorcht.

**Semantisches Modell:** Die Datenhaltung enthält ein semantisches Modell, das die Beschreibung der Anwendungslandschaft enthält und dem Informationsmodell gehorcht.

**Visualisierungsmodell:** Die Datenhaltung enthält ein spezifisches Visualisierungsmodell, das eine Erweiterung des werkzeuginhärenten Kernvisualisierungsmodells darstellt.

[1] Die in den folgenden Abschnitten aufgezählte Liste ist als möglichst umfangreiche Sammlung von Ideen zu verstehen, aus denen für die konkrete Realisierung Komponenten ausgewählt und in Teilen ihrer Funktionalität umgesetzt werden sollen.

**Symbolisches Modell:** Die Datenhaltung enthält ein oder mehrere symbolische Modelle, welche Teile der Beschreibung der Anwendungslandschaft visualisieren und dem Visualisierungsmodell gehorchen.

**Viewpointdefinition:** Die Datenhaltung enthält ein oder mehrere Viewpoint-Definitionen, welche die Transformationsregeln für die Generierung von Visualisierungen enthalten und auf den Elementen korrespondierender Informations- und Visualisierungsmodelle aufbauen.

Im einfachsten Falle bietet das Repository *lesenden und erzeugenden* Zugriff, auf die in ihm enthaltenen Modelle, wobei die Erzeugung von neuen Modellen nur im Falle von neuen symbolischen Modellen verwendet wird. Dieses Repository der einfachsten Version lässt sich als *zustandsloser Model Service* mit folgender Schnittstelle realisieren:

- ***read(uri:URI):Model*** liest das Modell, welches durch den URI identifiziert wird.
- ***create(uri:URI, content:Model)*** erzeugt ein neues Modell unter dem angegeben URI mit dem angegebenen Inhalt.

Eine komplexere Variante eines Repositorys könnte darüber hinaus auch schreibenden Zugriff auf die in ihm enthaltenen Modelle bieten, wodurch speziell Veränderungen an symbolischen und semantischen Modellen ermöglicht würden. In diesem Kontext wäre allerdings auch die Unterstützung transaktionaler Veränderungen sinnvoll, die sich nur über einen *zustandsbehafteten Model Service* realisieren liese. Dessen Schnittstelle sollte neben obigen Methoden für ein Repository und der Möglichkeit ein Isolationslevel (siehe Date in [Da00]) zu setzen, noch folgende transaktionale Methoden anbieten:

- ***create(elementUri:URI, value:Element) raises TransactionException*** erzeugt das referenzierte Element im Modell und verbindet es mit dem URI oder erzeugt eine Fehlermeldung, wenn das nicht möglich ist.
- ***update(elementUri:URI, newValue:Element) raises TransactionException*** verändert das durch den URI referenzierte Element eines Modells auf den angegebenen neuen Wert oder erzeugt eine Fehlermeldung, wenn das nicht möglich ist.
- ***delete(elementUri:URI) raises TransactionException*** löscht das durch den URI referenziert Element eines Modells oder das entsprechende Modell oder erzeugt eine Fehlermeldung, wenn das nicht möglich ist.
- ***commit() raises TransactionException*** Versucht die letzte Transaktion (seit dem letzten Commit) abzuschließen oder erzeugt eine Fehlermeldung, wenn das nicht möglich ist.
- ***rollback()*** Versucht die letzte Transaktion (seit dem letzten Commit) rückgängig zu machen.

Ein noch komplexerer Fall von Repository lässt sich realisieren, wenn andere Komponenten, welche Daten aus diesem Repository nutzen, sich als ≫zu benachrichtigend≪ registrieren können. Diese Registrierungen greifen, falls auf einem entsprechenden Modellelement die angegebene Operation ausgeführt wurde. Die Benachrichtigungsfunktion des Repositorys wird dabei auf einer angegebenen Komponente aufgerufen, welche die spezifische Schnittstelle ***Listener*** implementiert, über welche die Benachrichtigungen stattfinden. Zusätzlich definiert das Repository noch einen Datentyp ***Operation***, welcher eine Aufzählung der Operationen, auf welche eine Benachrichtigung spezialisiert werden kann, enthält. Mögliche Literale dieser Aufzählung wären: *Create*, *Update*, ... Die Schnittstellen der Methoden des Repositorys für den lesenden Zugriff verändern sich wie folgt:

- ***read(String uri:URI, listener:Listener, watchedOperation:Operation):Model*** liest das Modell, welches durch den URI identifiziert wird, und registriert ein zu benachrichtigendes Subjekt für die Benachrichtigung im Falle des Auftretens einer Operation des angegebenen Typs.
- ***read(String elementUri:URI, listener:Listener, watchedOperation:Operation):Element*** liest ein Element aus einem Modell, welches durch den URI identifiziert wird, und registriert ein Subjekt für die Benachrichtigung im Falle des Auftretens einer Operation des angegebenen Typs.

Weitere Erörterungen über die verschiedenen Funktionalitäten eines objektorientierten Repositorys und die verschiedenen Level der Transaktions- bzw. Nachrichtenunterstützung finden sich noch detaillierter im MOF Object Lifecycle RFP (vgl. [OM04]).

### 5.3.2 Komponente: Modelltransformer

Der Modelltransformer stellt die Transformationskomponente für objektorientierte Modelle aus Abschnitt 4.3 dar. Diese Komponente muss Viewpoint-Definitionen aus dem Repository interpretieren und die daraus gewonnenen Transformationensregeln konkret über dem semantischen Modell des Projekts zur Ausführung bringen. Dabei müssen eventuell nicht gesetzte Parameter in der Viewpointdefinition vor der Ausführung der Transformation, durch das umgebende System bereits belegt worden sein. Die Viewpoint-Definitionen werden im Repository selbst wiederum als Modell abgespeichert, welches einem *Transformationsmetamodell* gehorcht. Ein derartiger Ansatz der Repräsentation von Transformationen als Modell wird von der ATLAS group in [AT05] dargestellt. Der Modelltransformer selbst kann als *Model-Service* mit folgender Schnittstelle realisiert sein:

- ***transform(source:Model, rules:Model, sourceMetamodel:Model, targetMetamodel:Model):Model raises TransformationException*** transformiert das gegebene Quellmodell unter Anwendung der angegebenen Regeln und unter Verwendung der angegebenen Metamodelle in ein Zielmodell oder erzeugt einen Fehler, wenn die Transformation, z.B. aufgrund eines fehlenden Parameters, fehlschlägt.

Die Komponente des Modelltransformers kann dabei unter Verwendung verschiedener Transformationssprachen realisiert sein. Eine Liste möglicher Alternativen findet sich bei Buckl in [Bu05], dort werden auch die Vor- und Nachteile der verschiedenen Systeme beschrieben.

### 5.3.3 Komponente: System für regelbasiertes Schließen

Das System für regelbasiertes Schließen erlaubt es auf gegebenen Modellen anhand ihrer Metamodelle logische (deduktive) Schlüsse zu ziehen (vgl. z.B. Dietrich in [Di03]). So könnte ein Informationsmodell Annotationen enthalten, welche es einer Deduktionskomponente erlauben, z.B. Geschäftsprozessunterstützungen transitiv zu traversieren - also in der Situation, in der ≫Geschäftsprozess A≪ von der ≫betrieblichen Anwendung B≪ unterstützt wird und darüber hinaus ≫Geschäftsprozess C≪, sich in die ≫Geschäftsprozesse A und D≪ zerlegen lässt, folgern, dass die ≫betriebliche Anwendung B≪ auch den ≫Geschäftsprozess C≪ (transitiv) unterstützt. Derartige Deduktionen spielen allerdings nicht nur auf der Seite des semantischen Modells eine Rolle, sondern könnten auch auf Seite des symbolischen Modells Anwendung finden, evtl. um dort redundante Regeln zu erkennen und vor dem Aufruf des Layouters zu eliminieren. Das System für regelbasiertes Schließen lässt sich durch einen *Model-Service* mit folgender Schnittstelle realisieren:

- ***derive(source:Model, sourceMetamodel:Model):Model*** liest das Quellmodell und fügt gemäß der Regeln für logisches Schließen aus dem entsprechenden Metamodell auch ableitbare (transitive) Beziehungen und Attribute in das Zielmodell ein.
- ***reduce(source:Model, sourceMetamodel:Model):Model*** liest das Quellmodell und elimiert Beziehungen und Attribute, die gemäß der Regeln für logisches Schließen aus dem entsprechenden Metamodel auch ableitbar (transitiv) wären.

Bei der Realisierung eines Systems für regelbasiertes Schließen könnte dabei z.B. auf Ontologie-basierte Sprachen, z.B. der W3C OWL (vgl. [W3C03b]) zurückgegriffen werden. Allerdings würde die Umsetzung eines konkreten Systems für regelbasiertes Schließen umfangreiche Voruntersuchungen benötigen, die nicht im Rahmen dieser Arbeit angesiedelt sein können. Deswegen ist ein derartiges System nicht teil der zu realisierenden Anwendung.

### 5.3.4 Komponente: Layouter

Der Layouter übersetzt die Gestaltungsmittel- und Gestaltungsregelinstanzen mit ihren belegten oder noch zu berechnenden Gestaltungsvariablen gemäß des in Abschnitt 4.4.2 vorgestellten Verfahrens in ein Optimierungsproblem mit Randbedingungen und geeigneten Zielfunktionen. Dabei ist es möglich, dass ein Layouter nur einen Teil der Gestaltungsmittel- und Gestaltungsregelinstanzen in ein derartiges Problem übersetzt und weitere

Variablen und Regeln zum Layout durch einen anderen Layouter beibehält. Dies ist besonders dann sinnvoll, wenn mehrere Layouter nacheinander zum Einsatz kommen, so z.B. ein Layouter für das Layout flächiger Objekte kombiniert mit einem Layouter für das Layout von linienartigen Objekten. Nach der Übersetzung in das Optimierungsproblem wird dieses an einen Optimierer übergeben, dessen Ergebnisse wiederum vom Layouter in das symbolische Modell eingepflegt werden. Der Layouter selbst lässt sich dabei als *Model-Service* mit folgender Schnittstelle realisieren:

- ***layout(symbolicModel:Model, visualizationModel:Model):Model*** öffnet das symbolische Modell, wandelt die entsprechenden Instanzen von Visualisierungsobjekte aus dem angegebenen Visualisierungsmodell in ein Optimierungsproblem um, gibt dies an der Optimierer weiter und pflegt die Ergebnisse der Optimierung in das symbolische Modell ein.

Der Layouter ist dabei sehr eng an die Schnittstelle der von ihm verwendeten Optimierer gekoppelt, da die Repräsentation der Optimierungsprobleme sich von Optimierer zu Optimierer unterscheiden könnte. Eine allgemein geeignete Schnittstelle wird dabei im Abschnitt 5.3.5 vorgestellt.

### 5.3.5 Komponente: Optimierer

Der Optimierer löst das vom Layouter übergebene Optimierungsproblem durch eine geeignete Belegung der darin enthaltenen Variablen. Dabei kann je nach Gestalt des Problems eine andere Form von Optimierer nötig werden. Während ein Problem mit linearen Randbedingungen und einer linearen Zielfunktion mittels Simplex-Verfahren gelöst werden könnte, verlangen komplexere Randbedingungen oder Zielfunktionen nach fortgeschritteneren Lösungsalgorithmen, wie z.B. genetischen Algorithmen (vgl. hierzu Buttelmann et. al. in [BL04]). Je nach Art des verwendeten Optimierers unterscheidet sich auch möglicherweise die geeignete Repräsentation des Optimierungsproblems und damit auch die Schnittstelle zwischen Layouter und Optimierer. Eine allgemeingültige Schnittstelle ist durch die Klasse ***OptimizationProblem*** zu definieren, welche die mathematischen und logischen Bedingungen, sowie die Zielfunktion geeignet auf Basis von Termbäumen (vgl. hierzu Aho et.al. in [ASU86]) oder allgemeiner Termgraphen, d.h. Objektgraphen repräsentiert. Die Schnittstelle des Optimierers vereinfacht sich dann zu einer einzigen Methode:

- ***solve(problem:OptimizationProblem):OptimizationProblem raises SolverException*** liest ein Optimierungsproblem ein und führt eine Optimierung darüber aus oder erzeugt einen Fehler, u.a. dann wenn das angegebene Problem einer Optimierungsproblemklasse angehört, die von diesem Optimierer nicht bearbeitet werden kann.

### 5.3.6 Komponente: Renderer

Der Renderer nimmt ein symbolisches Modell, in dem die Gestaltungsvariablen der Gestaltungsmittelinstanzen z.B. durch vorhergehende Layouter zugewiesen worden sind, und stellt es in einem erwünschten Ausgabeformat dar. Dabei können verschiedene Varianten des Renderers unterschieden werden:

**Exporter:** Exportiert das symbolische Modell in eine Datei eines gewissen Graphikformats.

**Displayrenderer:** Zeigt das symbolische Modell in einem Bildschirmfenster an.

**Interaktiver Displayrenderer:** Zeigt das symbolischen Modell in einem Bildschirmfenster an und erlaubt dem Benutzer mit dem Modell zu interagieren.

Je nach Art des Renderers ist auch seine Realisierung durch die verschiedenen Arten von Model-Services möglich. Der Exporter und die nicht interaktive Variante des Displayrenderers können durch *Model-Services* mit einer einfachen Schnittstelle wie folgt realisiert werden:

- ***render(symbolicModel:Model, visualizationModel:Model)*** zeigt das gegebene symbolische Modell unter Nutzung der Informationen aus seinem Metamodell an.
- ***render(symbolicModel:Model, visualizationModel:Model, filename:URL)*** exportiert das gegebene symbolische Modell unter Nutzung der Informationen aus seinem Metamodell in eine Datei mit dem angegebenen Namen.

Die nicht interaktive Variante des Displayrenderers kann darüber hinaus noch als Nachrichtenempfänger realisiert werden und die Visualisierung dynamisch an Veränderungen des ihr zugrunde liegenden symbolischen Modells anpassen.

Die interaktive Variante des Displayrenderers kann nur durch einen Model-Service mit *Benachrichtigungsfunktion* realisiert werden, zumindest dann, wenn das zugrunde liegende symbolische Modell entsprechend der Benutzerinteraktion angepasst werden soll. Durch die Interaktion würde dann entsprechend eine Ausführung eines Inverse-Layouters (siehe Abschnitt 5.3.7) angestoßen. Die Schnittstelle des interaktiven Displayrenderers könnte wie folgt realisiert sein:

- ***render(symbolicModel:Model, visualizationModel:Model, listener: Listener)*** zeigt das gegebene symbolische Modell unter Nutzung der Informationen aus seinem Metamodell an und informiert bei durch Benutzerinteraktion hervorgerufene Veränderungen das zu benachrichtigende Subjekt.

Darüber hinaus muss die interaktive Variante des Displayrenderers immer als Nachrichtenempfänger realisiert werden, um die Visualisierung dynamisch an Veränderungen des ihr zugrunde liegenden symbolischen Modells anpassen zu können.

### 5.3.7 Komponente: Inverse-Layouter

Der Inverse-Layouter nimmt in gewisser Weise die Umkehrung der Aufgabe des Layouters wahr. Er nimmt ein symbolisches Modell, in dem die Variablen aller Gestaltungsmittelinstanzen belegt sind, und errechnet, welche Gestaltungsregeln aus einem angegebenen Regelkanon auf diesen Instanzen erfüllt sind. Der Regelkanon enthält dabei Gestaltungsregeln, welche in der vorliegenden Visualisierung eine semantische Information (also eine Information aus dem semantischen Modell) transportieren können. Das Ergebnis ist ein symbolisches Modell mit Gestaltungsmittel- und Gestaltungsregelinstanzen. Diese Funktionalität lässt sich durch einen *Model-Service* mit folgender Schnittstelle realisieren:

- ***invlayout(symbolicModel:Model, visualizationModelExcerpt:Model): Model*** durchsucht das angegebene symbolische Modell nach möglichen Instanzen der Regeln aus der angegebenen Menge von Gestaltungsregeln und produziert ein um erkannte Regeln erweitertes symbolisches Modell.

Darüber hinaus kann der Inverse-Layouter auch als Nachrichtenempfänger realisiert werden und damit dynamisch durch die Veränderungen, die von einem interaktiven Displayrenderer am symbolischen Modell durchgeführt werden, angestoßen werden.

### 5.3.8 Komponente: Ablaufsteuerung

Die Ablaufsteuerung wird durch die Entscheidung für einen *Model Bus* notwendig (siehe Abschnitt 5.2) und regelt die Zusammenarbeit der einzelnen Model Services. Aufgabe der Ablaufsteuerung ist es, einen in Instanzen eines Modells für die Ablaufsteuerung beschriebenen Ablauf wie angegeben auszuführen. Dabei registriert sich die Ablaufsteuerung, wo durch den Ablauf vorgesehen, z.B. als Nachrichtenempfänger für einen *benachrichtigenden Model-Service* oder aber fungiert selbst als Nachrichtenquelle für weitere Model-Services. Eine Ausführung eines Ablaufs durch die Ablaufsteuerung ist dabei immer an eine Instanz einer Dienstregistrierung gebunden, welche die URIs der Dienste in entsprechende URLs übersetzt. Die Schnittstelle der Ablaufsteuerung ist dabei wie folgt realisiert:

- ***execute(workflowModel:Model, registry:ModelServiceRegistry) raises ExecutionException*** führt den als Modell angegebenen Ablauf aus und verwendet dabei die angegebene Dienstregistrierung für die Auflösung von URIs. Produziert ein aufgerufener Model-Service einen Fehler oder kann die Dienstregistrierung einen URI nicht auflösen, so wird dieser unter Angabe des erzeugenden Verursachers als Ausführungsfehler propagiert.
- ***execute(workflowModel:Model, registry:ModelServiceRegistry, monitor: ExecutionMonitor) raises ExecutionException*** führt den als Modell angegebenen Ablauf aus, verwendet dabei die angegebene Dienstregistrierung für die Auflösung von URIs und gibt den Fortschritt der Ausführung an einen angegebenen

Monitor weiter. Produziert ein aufgerufener Model-Service einen Fehler oder kann die Dienstregistrierung einen URI nicht auflösen, so wird dieser unter Angabe des erzeugenden Verursachers als Ausführungsfehler propagiert.

Die Ablaufsteuerung unterstützt konditionale Ablaufstücke, die nur dann von der Ablaufsteuerung angestoßen werden, wenn in einem gegebenen Modell ein vorgegebener Bool'scher Ausdruck auf wahr auswertet. Eine weitere Funktion einer Ablaufsteuerung könnte durch die Anbindung an die Benutzerschnittstelle realisiert werden, mittels welcher der Benutzer an geeigneter Stelle um Eingabe von Daten (Parametern) für den Ablauf gebeten wird.

### 5.3.9 Komponente: Dienstregistrierung

Die Dienstregistrierung ist ebenfalls essentieller Bestandteil einer Architektur mit einem *Model Bus*. Die Aufgabe der Registrierung ist dabei die Abbildung der eindeutigen Namen von Model-Services (realisiert durch URIs) auf tatsächliche Instanzen von Realisierungen, die den entsprechenden Model-Service realisieren. Die Realisierungen werden dabei unter Angabe des Protokolls durch eine ≫physikalische Adresse≪ in Form eines URL lokalisiert. Die Dienstregistrierung kann dabei klassiche Funktionen, wie z.B. die Lastverteilung zwischen verschiedenen Realisierungen desselben Dienstes, aber auch die Reaktion auf mögliche Ausfälle einer Realisierung übernehmen. Die Schnittstelle der einfachsten Form einer Dienstregistrierung wäre dabei wie folgt zu realisieren:

- ***resolveModelService(modelServiceId:URI):URL***
  ***raises NoSuchServiceException*** versucht die Adresse einer Realisierung für den angegebenen Model-Service zu finden oder erzeugt einen Fehler, falls keine derartige bekannt ist.
- ***registerModelService(modelServiceId:URI, modelServiceAddress:URL)***
  registriert die Adresse einer Realisierung für den angegebenen Model-Service.
- ***unregisterModelService(modelServiceAddress:URL)*** löscht die Adresse einer Realisierung für alle von ihr bereitgestellten Model-Services.
- ***unregisterModelService(modelServiceAddress:URL, modelServiceId:URI)***
  Löscht die Adresse einer Realisierung für den angegebenen Model-Service.

## 5.4 Komponenteninteraktionsarchitektur

Aufgrund der Ausführung der Komponenten als Model-Services sind prinzipiell verschiedenste Orchestrierungen der Dienste, realisiert durch verschiedene Programmierungen der Ablaufsteuerung, denkbar. Um die Funktionsweise der Ablaufsteuerung mit Benutzerinteraktion und das Zusammenspiel der Model-Services zu illustrieren, werden beispielhafte

Abläufe jeweils kurz beschrieben, mit einem Diagramm dargestellt und zum korrespondierenden Anwendungsfall aus Abschnitt 3.1 in Beziehung gesetzt.

**W1:** *Lade Karte* Dieser Ablauf (siehe Abbildung 5.3) wird durch den *Kartennutzer* ausgelöst und bildet die Funktionalität aus dem Anwendungsfall *Lade Karte (UC1a)* ab. Dabei wird das Visualisierungsmodell, sowie ein symbolisches Modell aus dem Repository geladen und dem Renderer zur Anzeige übergeben.

**W2:** *Generiere Karte* Dieser Ablauf (siehe Abbildung 5.4) wird durch den *Kartennutzer* angestoßen und realisiert die Funktionalität aus dem Anwendungsfall *Generiere Karte (UC1b)*. Dabei werden das Informationsmodell, das Visualisierungsmodell, ein semantisches Modell sowie die Regeln für die Generierung aus dem Repository gelesen und vom Modelltransformer in ein symbolisches Modell transformiert. Dieses Modell wird zusammen mit dem Visualisierungsmodell an den Layouter übergeben, der wiederum die Instanzen der Gestaltungsregeln in ein konkretes Optimierungsproblem übersetzt, das nachfolgend von einem Optimierer gelöst wird. Die Ergebnisse der Optimierung werden vom Layouter in das symbolische Modell übertragen, welches abschließend zusammen mit dem Visualisierungsmodell an den Renderer zur Anzeige übergeben wird.

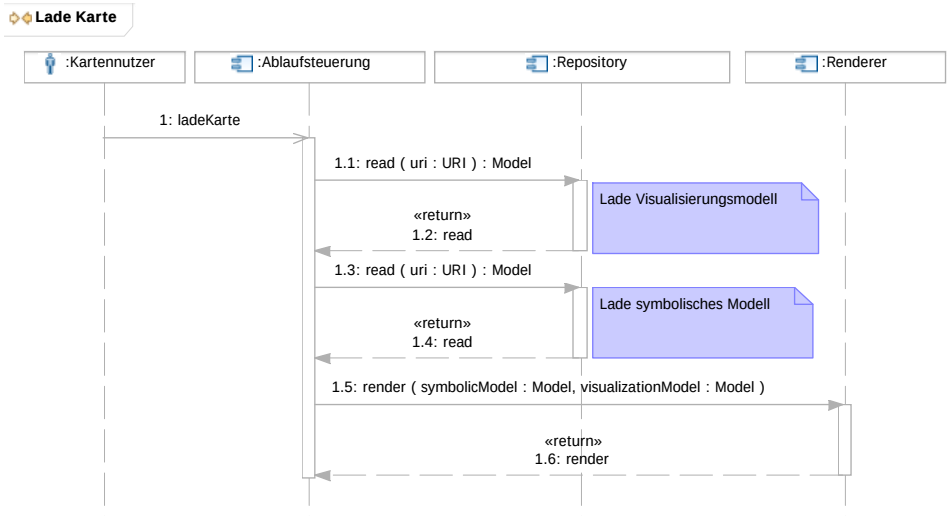



Abbildung 5.3: Aktivitätsdiagramm *Lade Karte*

## 5.5 Deploymentarchitektur

Die grundlegende Entscheidung für die Gliederung des Werkzeugs in Model-Services erlaubt verschiedene Szenarien für das tatsächliche Deployment. Die Realisierung der Model-Services kann dabei auf mehrere Rechner verteilt erfolgen. In der Folge sollen drei mögliche

Szenarien für eine Deploymentarchitektur textuell eingeführt und durch geeignete Vor- und Nachteile motiviert werden:

**Einzelplatzdeployment:** Bei dieser Variante für die Verteilungsarchitektur werden die Model-Services durch Realisierungen erbracht, die alle auf einem Rechner installiert sind. Diese Variante zeichnet sich durch ihren geringen Installations- und Wartungsaufwand auf, ist jedoch nicht mehrbenutzerfähig und im Hinblick auf die Rechnerleistung, die spezielle Realisierungen benötigen, kritisch.

**Mehrbenutzerdeployment:** Bei dieser Variante einer Deploymentarchitektur wird die Realisierung für den Repository Model-Service auf einem zentral zugänglichen Server installiert, so dass die auf verschiedenen Clients installierten (Rest-)Anwendungen darauf zugreifen können. Diese Variante ermöglicht Mehrbenutzerbetrieb, bringt allerdings den erhöhten Installations- und Wartungsbedarf einer verteilten Anwendung mit sich.

**Hochperformanzdeployment:** Bei dieser Variante der Verteilungsarchitektur wird die Realisierung des Layouter Model-Services und der daran angeschlossene Optimierer auf einen eigenen Rechner installiert, der für performante Ausführung von Berechnungen besser geeignet ist, als die Clients, auf denen die Restanwendung inklusive bzw. exklusive des Repository Model-Service installiert ist. Diese Variante erlaubt schnellere Erstellung von Visualisierungen, bringt allerdings Nachteile einer verteilten Architektur im Hinblick auf Installation und Wartung mit sich.

Weitere Szenarien sollen im Rahmen dieser Arbeit nicht mehr vorgestellt werden, sind jedoch denkbar. Besonders beim rechenintensiven Layouter Model-Service mit angeschlossenem Optimierer sind z.B. auch Varianten mit statischen oder dynamischen Lastverteilungsparadigmen, z.B. Round-Robin realisierbar.

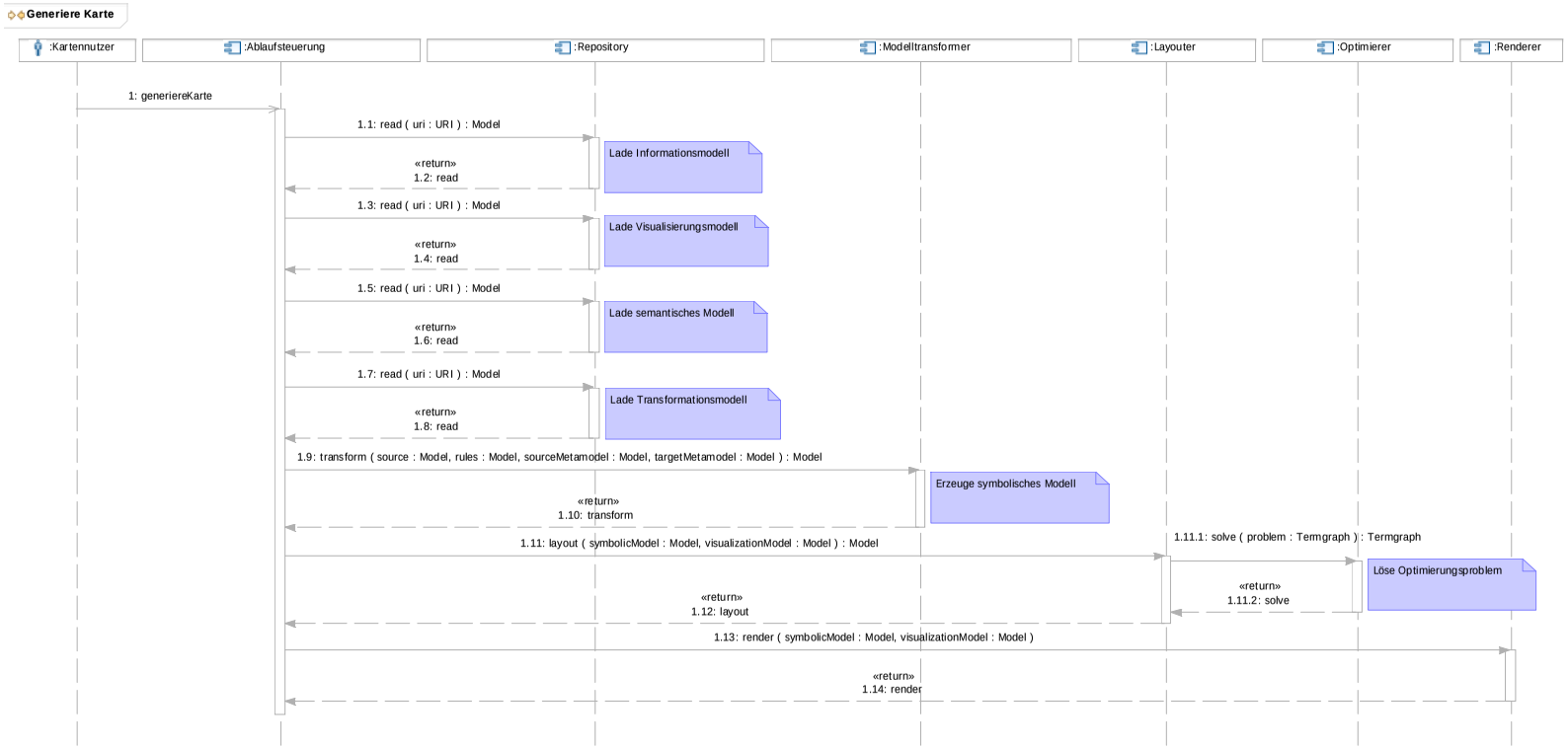



Abbildung 5.4: Aktivitätsdiagramm *Generiere Karte*

# Kapitel 6

# Plattformspezifische Architektur und Implementierung

Als Plattform für die Realisierung ausgewählter Kernkomponenten des in dieser Arbeit zu erstellenden Werkzeugs wurde die *Eclipse Rich Client Platform (RCP)* ausgewählt. Die Gründe für diese Entscheidung werden in Abschnitt 6.1 vorgestellt und an zentralen Eigenschaften und Funktionalitäten der Eclipse RCP festgemacht. Die konkrete Übertragung der allgemeinen Konzepte der plattformunabhängigen Architektur, speziell der Model-Services aus Abschnitt 5.2, auf die Gegebenheiten der Plattform werden in 6.2 vorgestellt. In Abschnitt 6.3 wird die Auswahl der in dieser Arbeit realisierten Komponenten motiviert, die in 6.3.1 bis 6.3.6 hinsichtlich ihrer Implementierung untersucht werden. Abschließend wird in Abschnitt 6.4 die Verteilungsarchitektur des erstellten Werkzeugs dargestellt

## 6.1 Eclipse Rich Client Platform

Nach dem Selbstverständnis der Eclipse Community handelt es sich bei Eclipse um eine Plattform, „die von Grund auf in ihrem Design darauf ausgelegt worden ist die Erstellung von Werkzeugen zur Web- oder Applikationsentwicklung zu unterstützen. Dabei stellt die Plattform direkt nur eine kleine Menge von Funktionen für den Endbenutzer zur Verfügung. Ihre Hauptaufgabe besteht darin, die schnelle Entwicklung von Funktionen in einer integrierten Umgebung auf Basis einer *Plug-In* Architektur zu unterstützen.“ [IB06].

Weiter lassen sich in der Beschreibung der Eclipse Plattform in [IB06] die Hauptfunktionen derselben, wie folgt zusammengefasst, finden:

- Bereitstellung einer gemeinsamen graphischen Benutzeroberfläche, die durch Plug-Ins erweitert werden kann, mit einer Standardnavigation,
- Unterstützung verschiedenster Betriebssystemplattformen durch geeignete Abstraktion in der Programmierschnittstelle und

- Bereitstellung von Funktionalitäten für die dynamische Ermittlung und das dynamische Laden von Plug-Ins sowie für deren Betrieb.

Die zentralen Begriffe der Architektur der Eclipse Plattform sind dabei die Begriffe *Plug-In*, *Erweiterungspunkt*, *Erweiterung* und *Sicht*, die wie folgt definiert sind. Ein *Plug-In* ist ein Beitrag zur einer konkreten Installation einer Eclipse Plattform, welcher die Plattform um Funktionalitäten erweitert. Dabei müssen diese Funktionalitäten nicht a priori für den Endbenutzer zugänglich sein, sondern nur von anderen Plug-Ins genutzt werden. Der Beitrag einer Funktionalität durch ein Plug-In erfolgt über die *Erweiterung* eines von der Plattform vordefinierten *Erweiterungspunkts* oder durch das Bereitstellen einer zusätzlichen *Sicht*.

Plug-In

Ein *Erweiterungspunkt* definiert einen Kontext, in den ein Plug-In neue Funktionen einbringen kann, sowie einen eindeutigen Namen, über den der Erweiterungspunkt referenziert wird. Erweiterungspunkte werden von der Plattform an sich oder aber von anderen Plug-Ins definiert. Im einfachsten Fall (wie z.B. beim Erweiterungspunkt ***org.eclipse.help.contexts***) stellt eine *Erweiterung* zu einem Erweiterungspunkt lediglich eine statische Sammlung von (X)HTML-Dateien zur Verfügung, die als kontextabhängige Hilfe eingebunden werden.

Erweiterungspunkt

Erweiterung

Komplexere Fälle von Erweiterungspunkten verlangen nach Erweiterungen durch Java-Klassen, welche eine spezifische und durch die Beschreibung des Erweiterungspunkts definierte Schnittstelle implementieren. Ein Beispiel hierfür bietet der Erweiterungspunkt ***org.eclipse.ui.intro***, der die Angabe einer Klasse erwartet, welche die von der Schnittstelle ***org.eclipse.ui.intro.IIntroPart*** definierten Methoden implementiert und dadurch eine Willkommensschirm erstellt.

Die Erweiterungen werden allgemein durch Plug-Ins realisiert, welche in der Regel aus mehr als einer Ressource (z.B. Java-Klasse oder XHTML-Datei) bestehen und mehr als einen Erweiterungspunkt erweitern können. Die Festlegung, welche Klasse oder Ressource aus dem Plug-In, dabei welchen Erweiterungspunkt erweitert, geschieht in der Konfigurationsdatei des Plug-Ins, dem sog. *Manifest*. In diesem Manifest können darüber hinaus noch Parameter für Erweiterungspunkte gesetzt werden, welche auf die Funktion oder Anwendbarkeit der angegebenen Erweiterung Einfluss nehmen. So kann z.B. der Erweiterungspunkt ***org.eclipse.ui.newWizards***, der Dialoge zum Erstellen neuer Ressourcen als Erweiterungen erwartet, auf bestimmte Projekte innerhalb von Eclipse beschränkt werden, so dass eine C++-Klasse nur in einem C++-Projekt und eine Java-Klasse nur einem Java-Projekt angelegt werden kann.

Das Manifest eines Plug-Ins wird von der Laufzeitumgebung der Plattform beim Start eingelesen. Hier ermittelt die Laufzeitumgebung die installierten Plug-Ins und deren im Manifest definierte Abhängigkeiten untereinander und stellt fest, bei welchen Plug-Ins alle Voraussetzungen erfüllt sind, d.h. alle abhängigen Plug-Ins geladen werden könnten. Diese Plug-Ins werden dann als »nutzbar« markiert, fehlen entsprechende Ressourcen, so wird das Plug-In als »gesperrt« markiert.

Bei dieser Form der Abhängigkeitsprüfung werden durch die Laufzeitumgebung zwar die Manifest-Dateien untersucht, nicht aber tatsächlich Ressourcen, wie z.B. Klassen oder Dateien geladen. Dieses Laden erfolgt erst dann, wenn die Funktionalität eines nutzbaren Plug-Ins konkret durch Endbenutzerinteraktion oder aber durch ein anderes Plug-In angefordert wird. Zu diesem Zeitpunkt wird das angeforderte Plug-In bei der Registrierung nachgefragt und geladen, wodurch sich im Erfolgsfalle der Status auf »aktiv« ändert. Erst jetzt sind die Ressourcen des nachgefragten Plug-Ins im Arbeitsspeicher und können in Aktion treten.

Sicht

Eine *Sicht* stellt eine besondere Form einer Erweiterung dar, welche sich an den Erweiterungspunkt ***org.eclipse.ui.views*** ankoppelt. Eine Sicht leistet einen Beitrag zur Benutzerobefläche der Plattform und visualisiert z.B. den Inhalt von Dateien eines spezifischen Formats in textueller oder graphischer Form. Dabei greift die Sicht auf Funktionen des *Standard Widget Toolkit (SWT)* (vergleiche dazu Daum in [Da05]) für Steuerelemente, wie z.B. Eingabefelder zurück oder verwendet die Abstraktion, die *Draw2D* (vergleiche Moore et. al in [Mo04]) anbietet, um Symbole auf einer Zeichenfläche anzuzeigen.

## 6.2 Architektur- und Implementierungsentscheidungen

Die in Abschnitt 5.2 illustrierten Architekturentscheidungen, welche der plattformunabhängigen Architektur zugrunde liegen, werden in diesem Abschnitt auf die besonderen Gegebenheiten der Eclipse Rich Client Platform angepasst. Dabei muss besonders die durch die Model-Services ermöglichte Erweiterbarkeit umgesetzt werden. Dies kann auf der Zielplattform durch eine konsequente Umsetzung der Komponenten im Allgemeinen und der Model-Services im Speziellen als Plug-Ins realisiert werden. Einen Überblick über die realisierten Plug-Ins in der Zielarchitektur bietet Abbildung 6.1, die in ihr dargestellten Plug-Ins werden nachfolgend erklärt.

Bei der Umsetzung der Model-Service-Architektur auf die Eclipse Plattform spielt der Erweiterungspunkt ***de.softwarekartographie.socatool.ModelService***, definiert im gleichnamigen Plug-In, eine zentrale Rolle. Dieser Erweiterungspunkt wird von allen Plug-Ins, welche einen Model-Service realisieren, erweitert. Dadurch kann gewährleistet werden, dass die Realisierungen von Model-Services durch die Registrierung für Erweiterungen und Erweiterungspunkte, wie sie die Laufzeitumgebung der Plattformumgebung anbietet, gefunden werden können. Dieser Mechanismus wird in Abschnitt 6.3.6 bei den Aspekten der Umsetzung der Dienstregistrierung detailliert beschrieben.

Ein wesentlicher Aspekt der Model-Services - ihre Verteilbarkeit - wie sie in Abschnitt 5.2 vorgestellt wird und für die Verteilungsszenarien aus Abschnitt 5.5 notwendig ist, wird nicht direkt durch die Plug-In Mechanismen der Eclipse RCP unterstützt. Um dennoch Verteilbarkeit gewährleisten zu können und damit das Konzept des Model-Services als Ganzes umzusetzen, wird in Abschnitt 6.4 eine Möglichkeit vorgestellt, welche aufbauend auf das im Moment in der Entwicklung befindliche Projekt der »Model Driven Development integration (MDDi)« (siehe Blanc in [Bl05]) Verteilung ermöglichen könnte.

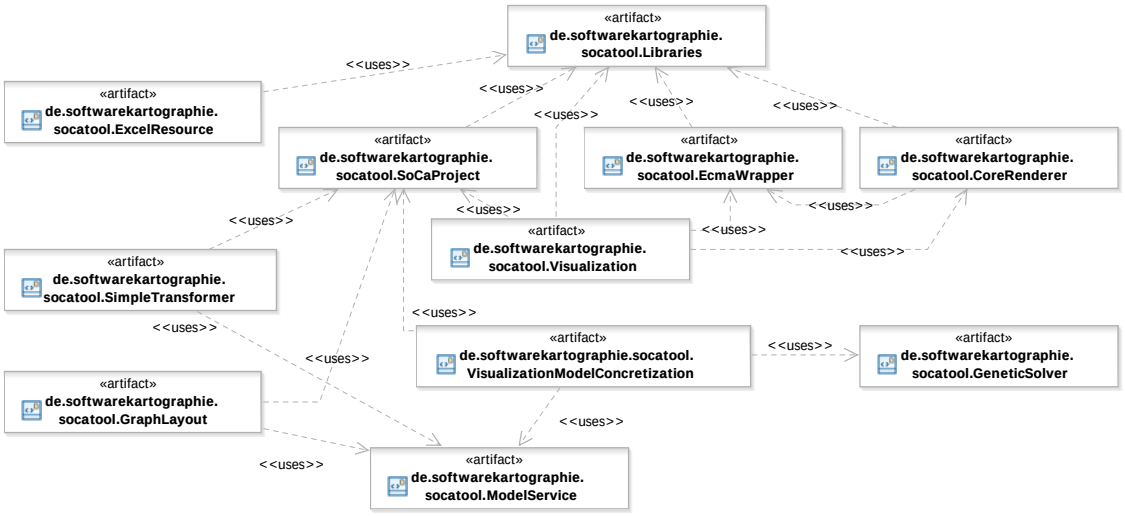



Abbildung 6.1: Überblick über die realisierten Plug-Ins

Weiterhin wurde im Hinblick auf eine Integration in die Eclipse RCP die Projektdefinition der Plattform, der Erweiterungspunkt ***org.eclipse.core.resources.natures*** erweitert und so im Plug-In ***de.softwarekartographie.socatool.SoCaProject*** das *SoCaProject* geschaffen, das neben den klassischen Eigenschaften eines Projekts in Eclipse noch zwei weitere Attribute beinhaltet, welche die Angabe eines XMI-serialisierten Informations- und eines ebenso serialisierten Visualisierungsmodells ermöglichen. Diese Modelle werden für den Kontext des gesamten Projekts als ≫gesetzt≪ betrachtet, alle semantischen und symbolischen Modelle, welche in dem Projekt auftreten, müssen den gewählten Metamodellen gehorchen.

Der Kern für Informations- sowie Visualisierungsmodelle, durch den die Interpretation und Verarbeitung dieser Modelle durch das Werkzeug erst möglich wird, also die zentralen Klassen und Assoziationen, welche von allen denkbaren Modellen beinhaltet werden müssen, wurden als unveränderliche Modelle (***http://www.softwarekartographie.de/socatool/informationmodel/core*** und ***http://www.softwarekartographie.de/socatool/visualizationmodel/core***) im Werkzeug verankert. Die Abhängigkeiten dieser beiden Modelle untereinander, sowie die Abhängigkeiten, welche tatsächlich genutzte Informations- und Visualisierungsmodelle zu diesen beiden Kernmodellen aufweisen, werden in Abbildung 6.2 veranschaulicht.

Das Paket ***http://www.softwarekartographie.de/socatool/informationmodel/core***, welches von allen Informationsmodellen importiert werden muss, enthält die abstrakte Klasse ***Informationsobjekt*** (vgl. Abbildung 6.4), von der alle Klassen erweiterter Informationsmodelle erben müssen. Ein derartiges Informationsmodell, welches im Rahmen dieser Arbeit als Beispiel für die Realisierung gewählt wurde, stellt Abbildung 6.3 dar.

Das Paket ***http://www.softwarekartographie.de/socatool/visualizationmodel/***

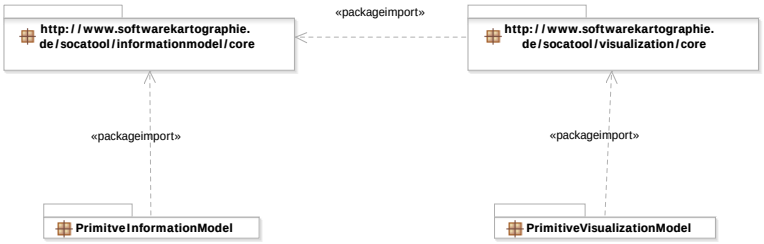



Abbildung 6.2: Beziehung zwischen den Kernmodellen und den genutzten Erweiterungen

***core*** bildet das Kernvisualisierungsmodell, welches von allen konkreten Visualisierungsmodellen importiert werden muss. Die schon in Abschnitt 4.2.1 (vgl. Abbildung 4.3) eingeführten Konzepte des objektorientierten Visualisierungsmodells von Ernst et. al. (vgl. [Er06]) werden im Kernvisualisierungsmodell (siehe Abbildung 6.4) um Klassen und Assoziationen erweitert, welche eine Kombination von semantischen und symbolischen Informationen in einem Modell erlauben. Diese gemeinsame Speicherung erlaubt die Umsetzung der nichtsynchronen Bearbeitung von semantischen Informationen, wie sie in Abschnitt 3.4 z.B. in Anforderung **AB6b** dargestellt wurde. Ein konkretes Visualisierungsmodell, wie es im Rahmen dieser Arbeit als Beispiel für die Realisierung gewählt wurde, stellt Abbildung 6.5 dar.

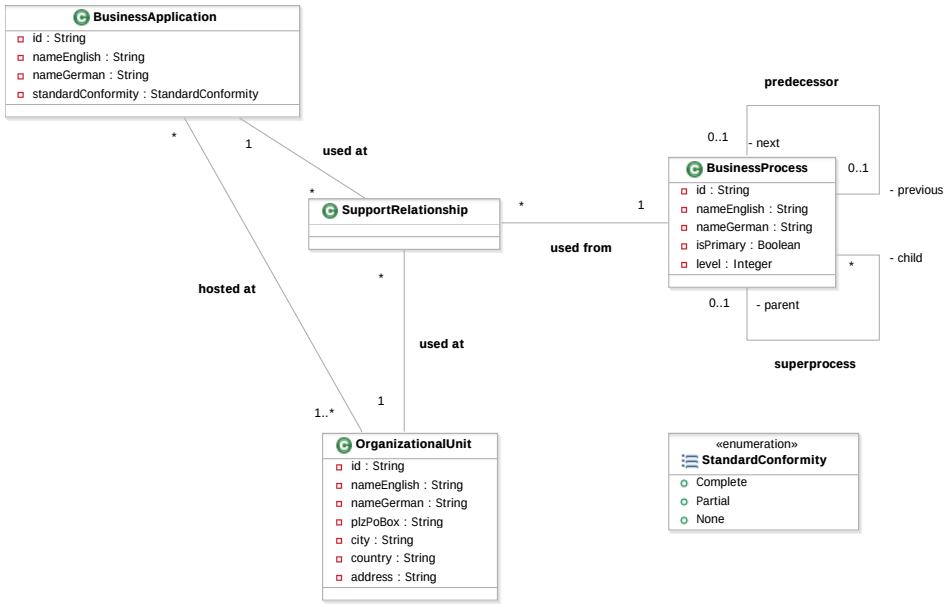



Abbildung 6.3: Beispielhaftes Informationsmodell

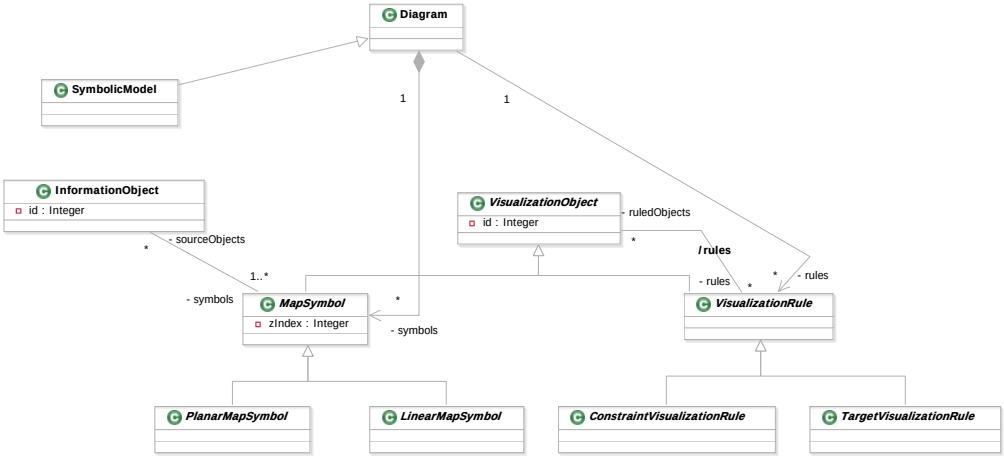



Abbildung 6.4: Aufbau des Kernvisualisierungsmodells

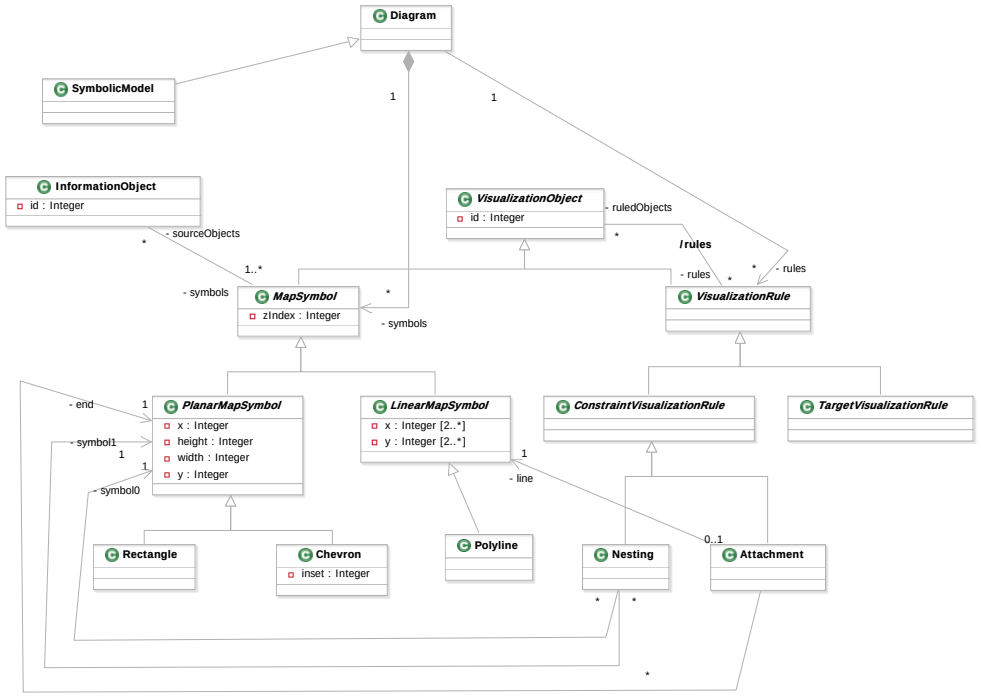



Abbildung 6.5: Beispielhaftes Visualisierungsmodell

Als gemeinsamens Metamodell für die im Werkzeug verwendeten Informations- und Visualisierungsmodelle wurde das ECore-Metamodell des Eclipse Modelling Frameworks (EMF) (vgl. Moore et. al in [Mo04]) ausgewählt, welches sich neben seiner konsistenten Integration in die Plattform durch die von ihm angeboteten Modellierungskonzepte auszeichnet. Die Konzepte orientieren sich dabei an den Konzepten von MOF 2.0 [OM06] und übersteigen, wie von Buckl in [Bu05] gezeigt, hinsichtlich des Umfangs der unterstützten Konzepte sogar *Essential MOF - EMOF*.

Dennoch bietet dieses Metamodell keine direkte Unterstützung für die Modellierung von Prädikaten oder Darstellungsfunktionen, wie sie ein konkretes mathematisches Visualisierungsmodell beinhalten muss. Deswegen wurde die von ECore zur Verfügung gestellte Möglichkeit allgemeine Modellelemente, wie z.B. Klassen, Referenzen oder Attribute mit Anmerkungen, sog. *Annotationen* zu versehen, genutzt. In diesen Annotationen (Instanzen der Annotationsklasse ***org.eclipse.emf.ecore.EAnnotation***) wurden für das Werkzeug interpretierbare Zusatzinformationen über Gestaltungsmittel und -regeln hinterlegt. Bei diesen Zusatzinformationen handelt es sich um:

Annotation

**Präsentationsinformationen** für Gestaltungsmittel, wodurch diese auf einfachere Symbole, sog. *graphische Primitive* zurückgeführt werden.

**Mathematisch-logische Informationen** für Gestaltungsmittel und -regeln, wodurch Gestaltungsvariablen mathematisch definiert und mittels Operationen zu Bedingungen bzw. Zielfunktionen zusammengesetzt werden können.

Die Beschreibung der Präsentationsinformationen für ein Gestaltungsmittel stützt sich, wie oben eingeführt, auf elementare Symbole. Diese *graphische Primitive* genannten Symbole werden durch die gewählte Ausgabeplattform direkt unterstützt. Um eine möglichst große Vielfalt von Ausgabeplattformen zu unterstützen, wurden nur Primitive ausgewählt, welche von den Plattformen SVG, Draw2D und PDF direkt unterstützt werden. Dies sind die Primitive *Ellipse*, *Rechteck*, *Abgerundetes Rechteck*, *Polygon*, *Polylinie* und *Schrift*. Aus diesen Primitiven werden komplexere Gestaltungsmittel, wie z.B. das RechteckMit-Text oder das Chevron zusammengesetzt. Die zur Ausführung dieser Zusammensetzung notwendigen Präsentationsinformationen werden in der Annotation ***supports*** unter dem Schlüssel ***de.softwarekartographie.visualization*** an die Klasse des entsprechenden Gestaltungsmittels annotiert und sind in einer mit ECMA-Script (vgl. [Ec99]) angereicherten XML-Notation kodiert. Nachfolgendes Codesegment, entnommen aus der entsprechende Annotation zum Gestaltungsmittel *Rectangle* (vgl. Klasse in Abbildung 6.5) soll die Funktionsweise der Präsentationsinformationen illustrieren:

Graphisches Primitiv

```
1 <root>
2   <figure typeuri="ScriptableRectangle" id="rect"/>
3   <figure typeuri="ScriptableLabel" id="label"/>
4   <script>
5     <![CDATA[
6       rect.x = source.x - source.width/2;
```

```
 7         rect.y = source.y - source.height/2;
 8         rect.width = source.width;
 9         rect.height = source.height;
10         label.x = source.x - source.width/2;
11         label.y = source.y - source.height/2;
12         label.width = source.width;
13         label.height = source.height;
14         label.text = source.text;
15       ]]>
16   </script>
17 </root>
```

In diesem Codesegment werden in den Zeilen 2 und 3 Instanzen von graphischen Primitiven des entsprechenden Typs gebildet und mit IDs versehen. In den Zeilen 6 bis 14 wird das ECMA-Script definiert, welches bei der Respräsentation einer Instanz der annotierten Gestaltungsmittelklasse ausgeführt werden soll. In diesen Zeilen finden Umrechnungen statt, welche die Werte der Gestaltungsvariablen der Instanz (im Script als ***source*** bezeichnet) verwenden, um aus ihnen die Werte zu berechnen, mit denen die Primitive belegt werden sollen. Auffällig sind dabei die Umrechnungen der Koordinaten in den Zeilen 6 und 7 bzw. 10 und 11; diese werden notwendig, da die Gestaltungsmittel allgemein über die x- und y-Koordinate ihres Mittelpunktes definiert werden, wohingegen die Koordinatensysteme der Ausgabeplattformen die linke, obere Ecke eines Primitivs als Bezugspunkt verwenden.

Die Beschreibung der mathematisch-logischen Informationen für Gestaltungsmittel erfolgt durch die Definition von Variablen in den Annotationen zu den Gestaltungsmitteln. Diese Variablen werden zu Termen, welche in den Annotationen zu den Gestaltungsregeln definiert werden, zusammengeführt, wodurch Bedingungen oder Zielfunktionen repräsentiert werden können. Diese Informationen sind in einer XML-Syntax notiert, welche sich an MathML [W3C03a] bzw. OpenMath [Op04] orientiert, ohne jedoch deren Syntax exakt einzuhalten. Die Entwicklung einer eigenen Syntax (eine Definition dieser Syntax findet sich in Anhang B) war notwendig, da weder MathML noch OpenMath konsistent typisierte Ausdrücke und Variablen unterstützen. Dies ist jedoch essentiell, da Gestaltungsmittel über Attribute verschiedenster Datentypen verfügen können und Gestaltungsregeln je nach Kategorie selbst wieder streng typisierte Terme enthalten, die entweder (im Fall einer Bedingungsregel) zu einem Bool'schen Wert oder (im Fall einer Zielregel) zu einer reellen Zahl evaluiert werden.

Die mathematisch-logischen Informationen für Visualisierungsobjekte sind dabei spezifisch für eine gewählte Konkretisierung des allgemeinen Visualisierungsmodells. Dies macht es möglich für verschieden Konkretisierungen in der Annotation ***supports*** einen eigenen Schlüssel anzulegen, welcher die spezifischen Formulierungen der Gestaltungsmittel und -regeln enthält. Beispielhaft wurde in vorliegender Arbeit die Konkretisierung unter Nutzung rechteckiger Füllen und Hüllen (vgl. Abschnitt 4.4.1) umgesetzt und hierfür der Schlüssel ***de.softwarekartographie.primitivelayout*** eingeführt.

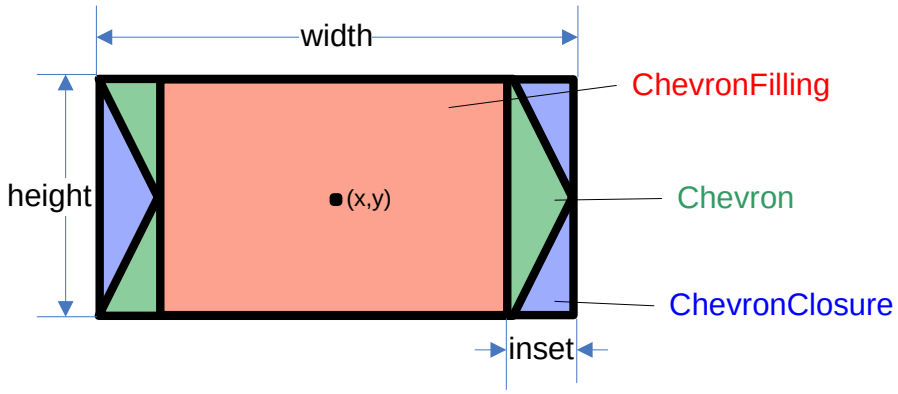



Abbildung 6.6: ***ChevronFilling***, ***ChevronClosure*** und ***Chevron*** mit ihren Gestaltungsvariablen

Anhand der nachfolgenden Codesegmente sollen die Funktionsweise und die Notation der mathematisch-logischen Informationen in einem konkreten Visualisierungsmodell illustriert werden. Das erste Codesegment ist dabei der entsprechenden Annotation zum Gestaltungsmittel *Chevron* (vgl. Klasse in Abbildung 6.5) entnommen:

```
1 <simplification>
2   <filling>ChevronFilling</filling>
3   <closure>ChevronClosure</closure>
4 </simplification>
```

In diesem Codesegment werden zwei Subklassen der allgemeinen rechteckigen Fülle bzw. Hülle angegeben (***ChevronFilling*** und ***ChevronClosure***), welche wiederum die Vorschriften zur Berechnung von Position, Breite und Höhe der entsprechenden Vereinfachungen enthalten. Diese Berechnungsvorschriften lassen sich leicht aus den, in Abbildung 6.6 dargestellten graphischen Beziehungen zwischen Gestaltungsmittel, Fülle und Hülle ableiten. Diese Vorschriften werden an den Attributen der Fülle (***ChevronFilling***) in der entsprechenden Klasse im konkreten Visualisierungsmodell als Annotationen angebracht und lauten dabei wie folgt:

**x:**

```
1 <var name="source.x"/>
```

**y:**

```
1 <var name="source.y"/>
```

**width:**

```
1 <sub>
2   <var name="source.width"/>
```

```
3     <mult>
4       <var name="source.inset"/>
5       <const value="2"/>
6     </mult>
7   </sub>
```

**height:**

```
1   <var name="source.height"/>
```

Durch diese Annotationen werden die Werte der Attribute ***x***, ***y*** und ***height*** direkt auf die Werte der entsprechenden Attribute der Gestaltungsmittelinstanz, also des Chevrons (bezeichnet durch ***source***), zurückgeführt. Der Wert des Attributs ***width*** der Fülle wird wiederum nach folgender Formel aus den Werte der entsprechenden Attribute der Gestaltungsmittelinstanz berechnet: $width = source.width - 2 * source.inset$. Analog zu diesen Termen existieren auch zu den Attributen der Hülle (***ChevronClosure***) entsprechende Annotationen mit Berechnungsvorschriften.

Auch die Gestaltungsregeln werden in ähnlicher Weise mit mathematisch-logischen Informationen zu ihrer Konkretisierung annotiert. Der Aufbau dieser Informationen soll an folgendem Codebeispiel illustriert werden, welches sich unter dem Schlüssel ***de.softwarekartographie.primitivelayout*** (als Konkretisierung unter Annahme rechteckiger Füllen und Hüllen) in der Annotation ***supports*** bei der Gestaltungsregel *Nesting* findet:

```
1   <definition type="constraint">
2     <expression>
3       <geq>
4         <sub>
5           <var name="symbol0.closure.x"/>
6           <div>
7             <var name="symbol0.closure.width"/>
8             <const value="2"/>
9           </div>
10        </sub>
11        <sub>
12          <var name="symbol1.filling.x"/>
13          <div>
14            <var name="symbol1.filling.width"/>
15            <const value="2"/>
16          </div>
17        </sub>
18      </geq>
19    </expression>
20    <expression>
21      <geq>
22        <add>
```

```
23          <var name="symbol1.filling.x"/>
24          <div>
25            <var name="symbol1.filling.width"/>
26            <const value="2"/>
27          </div>
28        </add>
29        <add>
30          <var name="symbol0.closure.x"/>
31          <div>
32            <var name="symbol0.closure.width"/>
33            <const value="2"/>
34          </div>
35        </add>
36      </geq>
37    </expression>
38    <!-- ... -->
39  </definition>
```

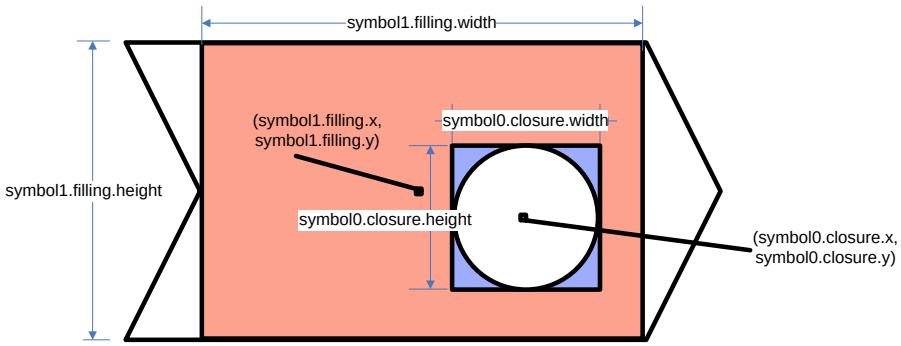



Abbildung 6.7: ***Nesting*** eines Kreises in einem Chevron ausgedrückt durch rechteckige Füllen und Hüllen

In diesem Codesegment wird in der Zeile 1 der Typ der Regel, hier eine Bedingungsregel, angegeben. In den Zeilen 2-19 wird eine erste Bedingung in Form eines Bool'schen Terms beschrieben, der in mathematischer Notation wie folgt lautet: $symbol0.closure.x - symbol0.closure.width/2 \geq symbol1.filling.x - symbol1.filling.width/2$. Kann dieser Term nach „wahr“ evaluiert werden, so bedeutet das für die Graphik, dass die linke Kante der rechteckigen Hülle von »symbol0« eine größere oder dieselbe x-Koordinate besitzt wie die linke Kante der rechteckigen Fülle von »symbol1«. Bei dieser Bedingung handelt es sich um eine Variante des ersten Terms, wie er in Abbildung 4.4 auf zwei rechteckige Gestaltungsmittelinstanzen angewandt wird. Eine Graphik, welche die Anwendung des Terms auf rechteckige Füllen und Hüllen illustrieren soll, findet sich in Abbildung 6.7. Analog formulieren die Zeilen 20-37 die entsprechende Bedingung auf der rechten Kante

der rechteckigen Fülle bzw. Hülle. Ausgelassen wurden in diesem Codebeispiel die Terme, welche die Bedingungen auf den unteren bzw. oberen Kanten der rechteckigen Fülle und Hülle etablieren. Eine vollständige Beschreibung der Notation, wie sie für die Formulierung dieser Terme verwendet wird, findet sich in Anhang B.

## 6.3 Implementierungen der Komponenten

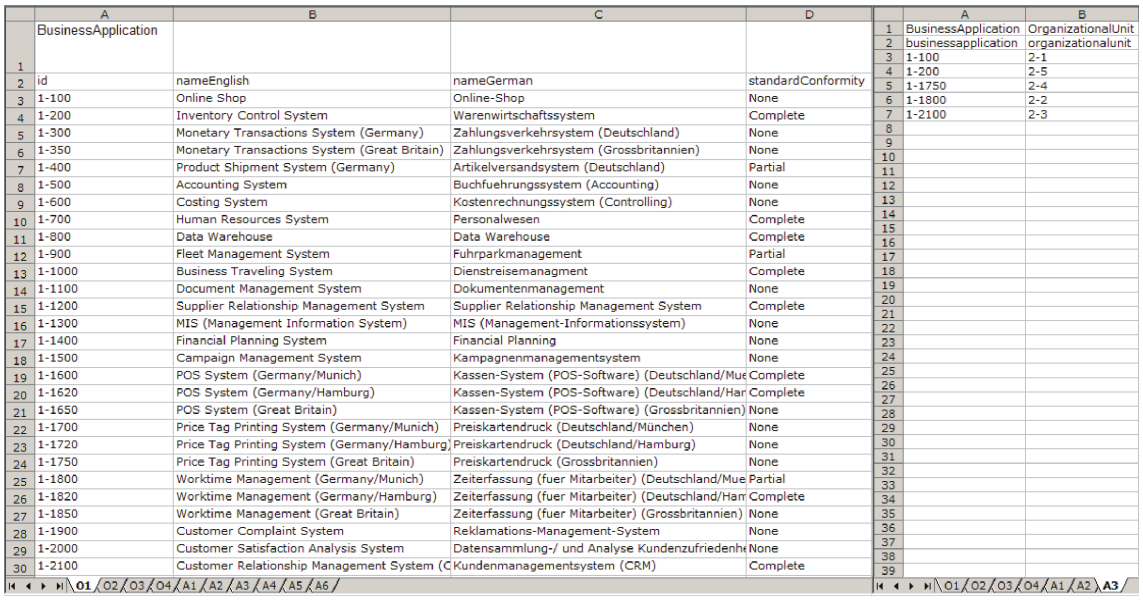
Aus der in Abschnitt 5.3 vorgestellten plattformunabhängigen Komponentenarchitektur wurden »Kernkomponenten« ausgewählt, welche konkret realisiert worden sind. Ein wesentliches Kriterium für die Auswahl einer Komponente war dabei die Beteiligung dieser Komponente an den in Abschnitt 5.4 exemplarisch vorgestellten zentralen Abläufen im Werkzeug, welche in der erstellten Version des Werkzeugs ausführbar sein sollten. Unter diesem Gesichtspunkt wurde auf die Realisierung der Komponenten *System für regelbasiertes Schließen* und *Inverse-Layouter* verzichtet, die Funktionalitäten einiger anderer Komponenten wurden nur zum Teil realisiert, was in den folgenden Abschnitten bei den einzelnen Komponenten angemerkt wird.

Die zentralen Abläufe werden darüber hinaus nicht automatisiert und durch eine *Ablaufsteuerung* kontrolliert abgearbeitet, sondern durch den Benutzer interaktiv gestartet. Eine konfigurierbare *Ablaufsteuerung* ist in der momentanen Version des Werkzeugs nicht enthalten, die Abläufe werden in Java kodiert. In diesem Zusammenhang wird auf das laufende Eclipse MDDi Projekt (vgl. Blanc in [Bl05]) verwiesen, in Rahmen dessen die Entwicklung einer deklarativen Ablaufsteuerung geplant ist, auf welche eine Weiterentwicklung des vorliegenden Werkzeugs zurückgreifen könnte.

### 6.3.1 Komponente: Repository

Bei der Realisierung des Repositorys wurde in weiten Bereichen auf die vom Eclipse Modelling Framework (EMF) zur Verfügung gestellten Implementierungen von Ressourcen zurückgegriffen. In EMF ist eine Ressource eine Menge von Objekten, auf die mittels reflektiver Funktionen zugegriffen werden kann. Entsprechend ist die allgemeine Schnittstelle von Resourcen durch ***org.eclipse.emf.ecore.resource.Resource*** definiert und erlaubt die Suche nach Objekten über den Wert ihres ID-Attributs. Ein Objekt selbst stellt über die Implementierung der Schnittstelle ***org.eclipse.emf.ecore.EObject*** Methoden zur Verfügung, um auf die Werte von Attributen und Referenzen, aber auch auf die zugehörigen Metaobjekte, d.h. Klassen zugreifen zu können. Durch die allgemeine Schnittstelle des Repositorys können verschiedene Realisierungen genutzt werden, so z.B. das Elver EMF Repository [El06]. Im Rahmen dieser Arbeit wurde eine eigene Repository Implementierung - die ***ExcelResource*** - geschaffen, die weiter unten erklärt wird.

Das oben eingeführte *SoCaProject* stellt einen gemeinsamen Zugriff auf alle in seinem Repository enthaltenen Modelle und Metamodelle über ein ***org.eclipse.emf.ResourceSet*** zur Verfügung, in welchem die Ressourcen gemeinsam verwaltet und ihre Querbezüge her-



| | A | B | C | D |
|---|---|---|---|---|
| 1 | BusinessApplication | | | |
| 2 | id | nameEnglish | nameGerman | standardConformity |
| 3 | 1-100 | Online Shop | Online-Shop | None |
| 4 | 1-200 | Inventory Control System | Warenwirtschaftssystem | Complete |
| 5 | 1-300 | Monetary Transactions System (Germany) | Zahlungsverkehrsystem (Deutschland) | None |
| 6 | 1-350 | Monetary Transactions System (Great Britain) | Zahlungsverkehrsystem (Grossbritannien) | None |
| 7 | 1-400 | Product Shipment System (Germany) | Artikelversandsystem (Deutschland) | Partial |
| 8 | 1-500 | Accounting System | Buchfuehrungssystem (Accounting) | None |
| 9 | 1-600 | Costing System | Kostenrechnungssystem (Controlling) | None |
| 10 | 1-700 | Human Resources System | Personalwesen | Complete |
| 11 | 1-800 | Data Warehouse | Data Warehouse | Complete |
| 12 | 1-900 | Fleet Management System | Fuhrparkmanagement | Partial |
| 13 | 1-1000 | Business Traveling System | Dienstreisemanagment | Complete |
| 14 | 1-1100 | Document Management System | Dokumentenmanagement | None |
| 15 | 1-1200 | Supplier Relationship Management System | Supplier Relationship Management System | Complete |
| 16 | 1-1300 | MIS (Management Information System) | MIS (Management-Informationssystem) | None |
| 17 | 1-1400 | Financial Planning System | Financial Planning | None |
| 18 | 1-1500 | Campaign Management System | Kampagnenmanagementsystem | None |
| 19 | 1-1600 | POS System (Germany/Munich) | Kassen-System (POS-Software) (Deutschland/Mue | Complete |
| 20 | 1-1620 | POS System (Germany/Hamburg) | Kassen-System (POS-Software) (Deutschland/Har | Complete |
| 21 | 1-1650 | POS System (Great Britain) | Kassen-System (POS-Software) (Grossbritannien) | None |
| 22 | 1-1700 | Price Tag Printing System (Germany/Munich) | Preiskartendruck (Deutschland/München) | None |
| 23 | 1-1720 | Price Tag Printing System (Germany/Hamburg) | Preiskartendruck (Deutschland/Hamburg) | None |
| 24 | 1-1750 | Price Tag Printing System (Great Britain) | Preiskartendruck (Grossbritannien) | None |
| 25 | 1-1800 | Worktime Management (Germany/Munich) | Zeiterfassung (fuer Mitarbeiter) (Deutschland/Mue | Partial |
| 26 | 1-1820 | Worktime Management (Germany/Hamburg) | Zeiterfassung (fuer Mitarbeiter) (Deutschland/Har | Complete |
| 27 | 1-1850 | Worktime Management (Great Britain) | Zeiterfassung (fuer Mitarbeiter) (Grossbritannien) | None |
| 28 | 1-1900 | Customer Complaint System | Reklamations-Management-System | None |
| 29 | 1-2000 | Customer Satisfaction Analysis System | Datensammlung-/ und Analyse Kundenzufriedenhe | None |
| 30 | 1-2100 | Customer Relationship Management System (C | Kundenmanagementsystem (CRM) | Complete |

O1 | O2 | O3 | O4 | A1 | A2 | A3 | A4 | A5 | A6

| | A | B |
|---|---|---|
| 1 | BusinessApplication | OrganizationalUnit |
| 2 | businessapplication | organizationalunit |
| 3 | 1-100 | 2-1 |
| 4 | 1-200 | 2-5 |
| 5 | 1-1750 | 2-4 |
| 6 | 1-1800 | 2-2 |
| 7 | 1-2100 | 2-3 |

O1 | O2 | O3 | O4 | A1 | A2 | A3

Abbildung 6.8: Arbeitsblätter in einer ***ExcelResource***

gestellt werden. Beim Erstellen eines SoCaProjects werden automatisch zwei Modelle, das Kerninformationsmodell ***http://www.softwarekartographie.de/socatool/informationmodel/core*** und das Kernvisualisierungsmodell ***http://www.softwarekartographie.de/socatool/visualizationmodel/core***, in das ResourceSet des Projekts geladen. Hat der Benutzer für das Projekt konkrete Informations- und Visualisierungsmodelle spezifiziert, so werden diese danach geladen und deren Referenzen und Vererbungsbeziehungen zu Klassen aus den Kernmodellen aufgelöst.

Nachdem die entsprechenden Metamodelle erfolgreich geladen werden konnten, ist über das ResourceSet des *SoCaProjects* der Zugriff auf semantische und symbolische Modelle möglich, die unter Nutzung der korrespondierenden Informations- und Visualisierungsmodelle interpretiert werden können. An dieser Stelle kommt auch eine im Rahmen dieser Arbeit erstellte Implementierung von ***org.eclipse.emf.ecore.resource.Resource*** im Plug-In ***de.softwarekartographie.socatool.ExcelResource*** zum Einsatz, die es ermöglicht semantische Modelle im XML Spreadsheet Format (vgl. hierzu Anhang C) zu interpretieren. Diese Modelle werden in Microsoft Excel in Arbeitsblättern (vgl. Abbildung 6.8) modelliert, die entweder Instanzen einer Klasse (Arbeitsblätter **O<xx>**) oder Verknüpfungen (Arbeitsblätter **A<xx>**) enthalten. Der oben dargestellte Ansatz würde prinzipiell auch ändernden Zugriff auf die Daten aus den verschiedenen semantischen und symbolischen Modellen ermöglichen, für die Realisierung in der vorliegenden Arbeit jedoch wurde dieser Zugriff lediglich auf Änderungen an symbolischen Modellen beschränkt.

### 6.3.2 Komponente: Modelltransformer

Das Plug-In ***de.softwarekartographie.socatool.SimpleTransformer*** beinhaltet eine mögliche Realisierung eines Modelltransformers und stellt einen Model-Service dar. Über die Schnittstelle dieses Service können ein Quellmodell und sein entsprechendes Metamodell, sowie ein Zielmetamodell angegeben werden, welche dann von einer Java-Klasse im Transformer, in welcher die Transformationsvorschriften (das Transformationsmodell) konkret implementiert sind, in ein Zielmodell überführt werden. Diese Implementierung nutzt dabei die von EMF zur Verfügung gestellten Methoden für den reflektiven Zugriff auf die Objekte aus Modell und Metamodell und erzeugt über die ebenfalls von EMF bereit gestellten Fabrikmethoden aus der Klasse ***org.eclipse.emf.ecore.EFactory*** Instanzen von Klassen aus dem Zielmetamodell.

Durch die Spezifizierung der Transformationsvorschriften als Java-Code in dieser Realisierung eines Modelltransformers war es möglich die Mächtigkeit einer Programmiersprache zu nutzen, um diese Vorschriften zu beschreiben. Zugleich wurde aber auch der Nachteil in Kauf genommen, dass der Designer für Transformationsmodelle diese Programmiersprache beherrschen muss. Ein vereinfachtes Verfahren zur Festlegung von Transformationsvorschriften wäre wünschenswert, so dass diese, wie von Ernst et. al. in [Er06] dargestellt, in einer deklarativen Sprache formuliert werden können.

### 6.3.3 Komponente: Layouter

Das Layouting von symbolischen Modellen wurde in zwei Schritte zerlegt. Dies geschah in Übereinstimmung mit der von Ernst et. al. in [Er06] vorgeschlagenen Konkretisierung für das Visualisierungsmodell, nach welcher zuerst das Layout der flächenartigen Gestaltungsmittel berechnet werden soll und dann aufbauend auf die Ergebnisse dieses Layouts die linienartigen Gestaltungsmittel gelayoutet werden sollen. Die beiden realisierten Layouter finden sich in den Plug-Ins ***de.softwarekartographie.socatool.VisualizationModelConcretization*** und ***de.softwarekartographie.socatool.GraphLayout***, ersterer führt das Layout flächenartiger Getaltungsmittel durch, zweiterer errechnet das optimale Routing für linienartige Gestaltungsmittel.

Bei der Umsetzung des Layouters für flächenartige Gestaltungsmittel wurden die Konzepte von *Fülle* und *Hülle* (vgl. hierzu Abschnitt 4.4.1) durch rechteckige Objekte realisiert. Die Korrespondenz zwischen einem Gestaltungsmittel und seinem Füllen- bzw. Hüllenrechteck wurde dabei, wie in Abschnitt 6.2 dargestellt, im verwendeten Visualisierungsmodell annotiert. Analog wurden auch die Gestaltungsregeln, wie in Abschnitt 6.2 dargestellt, annotiert, so dass auf Basis dieser Annotationen durch den Layouter aus einem symbolischen Modell ein durch Termgraphen repräsentiertes Optimierungsproblem gewonnen werden kann. Dieses Optimierungsproblems wird einer eigenen Komponente, dem Optimierer, zur Lösung übergeben.

Das Routing der linienartigen Gestaltungsmittel wird von einem Layouter übernommen,


Abbildung 6.9: Ausschnitt aus einem symbolischen Modell

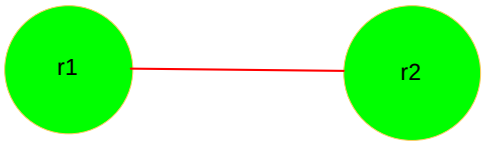



Abbildung 6.10: Zum Ausschnitt 6.9 korrespondierer Ausschnit aus einem Graphen

der ein symbolisches Modell in eine geeignete Repräsentation als Graph überführt. Bei dieser Umwandlung werden Annotationen genutzt, welche flächenartige Gestaltungsmittel allgemein als *Knoten* auszeichnen. Die Gestaltungsregel ≫Attachment≪ wurde entsprechend annotiert, so dass sie für die Ermittlung der an eine Instanz eines flächenartigen Gestaltungsmittels angeknüpften Instanzen linienartiger Gestaltungsmittel verwendet werden kann. Diese Umwandlung wird beispielhaft an einem Ausschnitt aus einem symbolischen Modell (siehe Abbildung 6.9) und dem korrespondierenden Ausschnitt aus einem Graphen (siehe Abbildung 6.10) illustriert. Diese Knoten-Kanten-Repräsentation der Visualisierung wird dann an eine einfache Form eines Layouters übergeben, welche eine Mittelpunktsverbindung berechnet. Zusätzlich wurde in einer parallelen Arbeit ein Layouter für ein orthogonales Linienrouting entwickelt, der über dieselbe Schnittstelle angebunden werden kann.

Die Zweiteilung des Layouters, wie sie in vorliegendem Werkzeug realisiert wurde, setzt, wie oben beschrieben, eine effektiv und effizient berechenbare Konkretisierung des Visualisierungsmodells um, bringt aber die von Ernst et. al. in [Er06] beschriebenen Nachteile mit sich. Zu diesen Nachteilen zählt speziell das möglicherweise nicht global-optimale Layout der Instanzen linienartiger Gestaltungsmittel, da diese an den bereits positionierten Instanzen flächenartiger Gestaltungsmitteln ausgerichtet werden, statt in einem gemeinsamen Layoutingschritt auch auf die Positionierung der Instanzen flächenartiger Gestaltungsmittel Einfluss zu nehmen.

Dennoch werden auch mögliche andere Konkretisierungen des Visualisierungsmodells, welche z.B. Teile des Visualisierungsmodells auf lineare Optimierungsprobleme abbilden, durch das Plug-In ***de.softwarekartographie.socatool.VisualizationModelConcretization*** unterstützt. So kann dieses Plug-In allgemeine mathematisch-logische Annotationen in einen Termgraphen übersetzen und einem Optimierer übergeben. Dadurch wäre es z.B. möglich eine zweite Konkretisierung in das Visualisierungsmodell zu annotieren und ein Layout auf Basis derselben zu berechnen.

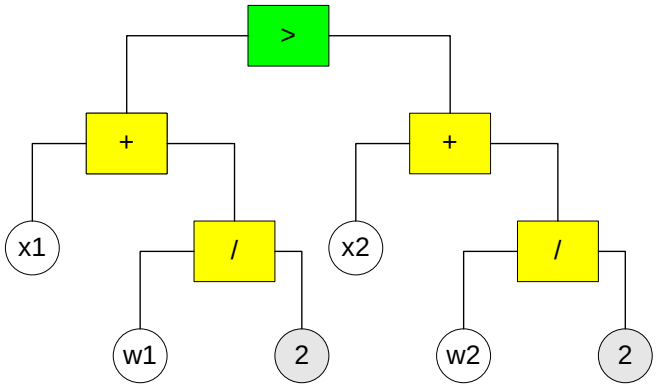



Abbildung 6.11: Repräsentation einer Bedingung eines Optimierungsproblems durch einen Termbaum

### 6.3.4 Komponente: Optimierer

Bei einem Optimierungsproblem, welches auf Basis der gewählten Konkretisierung des Visualisierungsmodells aus einem symbolischen Modell durch den Layouter generiert werden kann, handelt es sich im allgemeinen um ein nicht-linear restringiertes Problem, dessen Lösungsraum nicht zwingend im mathematischen Sinne als kompakt bezeichnet werden kann, mit einer nicht-linearen Zielfunktion. Abbildung 6.11 illustriert den Aufbau eines Termgraphen am Beispiel einer Bedingung. In dieser Abbildung werden

- Knoten, welche Bool'sche Operatoren enthalten, durch grüne Rechtecke,
- Knoten, welche reelle Operatoren enthalten, durch gelbe Rechtecke,
- Knoten, welche reelle Variablen enthalten, durch gelbe Kreise und
- Knoten, welche reele Konstanten enthalten, durch graue Kreise repräsentiert.

Bei der Lösung eines Optimierungsproblems müssen im allgemeinen mathematisch und numerisch aufwändige Lösungsverfahren zum Einsatz kommen, deren Implementierung nicht Teil der vorliegenden Arbeit gewesen sind. Durch Verwendung der ≫Third-Party≪-Bibliothek JGAP [MR06] war es jedoch möglich einen *genetischen Algorithmus* in das realisierte Werkzeug einzubinden. Dabei wurden auch einige Anpassungen am Algorithmus und den eingesetzten genetischen Operatoren vorgenommen, um das Laufzeitverhalten positiv zu beeinflussen. Eine eingehende Untersuchung der Konfiguration des Algorithmus und der Auswahl der genetischen Operatoren war jedoch nicht Teil der vorliegenden Arbeit und sollte im Hinblick auf potentielle, noch nicht genutzte Möglichkeiten zur Optimierung in zukünftigen Forschungen untersucht werden.

### 6.3.5 Komponente: Renderer

Im Rahmen der vorliegenden Arbeit wurden zwei verschiedene Arten von Renderern realisiert, zum einen ein nicht-interaktiver Diplayrenderer auf Basis des Eclipse *Graphical Editor Frameworks (GEF)* (vgl. hierzu Moore in [Mo04]), zum anderen ein Exporter für PDF-Dateien. Gemeinsame Basis der beiden Renderer bildet dabei das Plug-In ***de.softwarekartographie.socatool.CoreRenderer***, welches entsprechend der Präsentationsannotationen aus dem Visualisierungsmodell von einer abstrakten Fabrik zur Verfügung gestellte graphische Primitive instantiiert und deren Werte gemäß der Angaben aus dem in der Präsentationsannotation enthaltenen ECMA-Script belegt. Dabei können verschiedene Fabriken für Primitive verwendet werden, die zu den entsprechenden Zielformaten kompatibel sind. Ein minimales ≫Vokabular≪ wird dabei als unterstützt vorausgesetzt, wie bereits in Abschnitt 6.2 dargestellt.

Im Falle des nicht-interaktiven Diplayrenderers werden die Primitive durch Draw2D Objekte (abgeleitet von ***org.eclipse.draw2d.IFigure***) repräsentiert, welche von GEF an eine Sicht zur Anzeige übergeben werden. Auf dieser Sicht könnten mittels von GEF angebotenen Funktionalitäten auch Interaktionen ermöglicht werden, deren Realisierung jedoch nicht Bestandteil dieser Arbeit war. Lediglich einige Interaktionen wurden ermöglicht, so z.B. die Möglichkeit Schichten der Visualisierung auf ≫sichtbar≪ oder ≫unsichtbar≪ zu setzen.

Der PDF-Exporter repräsentiert die Primitive durch die von PDF zur Verfügung gestellten graphischen Elemente. Diese Elemente werden, je nach Zugehörigkeit zu einer Schicht, entsprechend einer durch den Exporter angelegten Schicht im PDF zugeordnet und können in dieser Visualisierung entsprechend ein- respektive ausgeblendet werden.

### 6.3.6 Komponente: Dienstregistrierung

Die Realisierung der Dienstregistrierung erfolgte unter Verwendung von der Eclipse Rich Client Platform direkt zur Verfügung gestellter Funktionen. Dabei wurde eine Plattform-Funktion verwendet, mittels welcher es möglich ist, zu einem gegebenen Erweiterungspunkt, alle bekannten Erweiterungen zu ermitteln. Da die Model-Services, wie in Abschnitt 6.2 beschrieben, als Erweiterungen des durch ein Plug-In definierten Erweiterungspunktes ***de.softwarekartographie.socatool.ModelService*** realisiert wurden, kann die Komponente Dienstregistrierung die Plattform-Funktion zur Ermittlung einer Liste der bekannten Erweiterungen diese Erweiterungspunktes verwenden. Ein konkreter Model-Services kann in dieser Liste über seinen eindeutigen Namen gefunden und eine Instanz des Services bei Anforderung durch die Ablaufsteuerung geladen bzw. erzeugt werden.

## 6.4 Verteilungsaspekte

Von den in Abschnitt 5.5 vorgestellten Verteilungsszenarien wird in der vorliegenden Version des Werkzeugs nur die Installation auf einem Rechner unterstützt. Verteilte Installationen und verteilter Betrieb wären dabei jedoch durch die Nutzung von ≫Third-Party≪-Bibliotheken möglich. Ein Szenario für die Verteilung des Repositorys könnte unter Nutzung des *EMF Persistency-Projektes* (vgl. hierzu [El06]) realisiert werden, welches die Persistierung von EMF-Objekten in eine (über Netzwerk anzusprechende) Datenbank ermöglicht. Dadurch könnten darüber hinaus Funktionalitäten für die transaktionale Bearbeitung von Daten durch eine Abbildung auf entsprechende Datenbankfunktionen zur Verfügung gestellt werden.

Weitere Verteilungsszenarien, welche die Auslagerung der Realisierung für einen einzelnen Model-Service auf einen anderen Computer beinhalten, ließen sich durch die Nutzung der Model-Bus Realisierung aus dem MDDi-Projekt (siehe Blanc in [Bl05]) in die Tat umsetzen. Der Model-Bus lässt sich dabei nahtlos in die bestehende Infrastruktur, bereitgestellt durch den Erweiterungspunkt ***de.softwarekartographie.ModelService*** integrieren. Die Erweiterung würde dann über einen Model-Service realisiert, der selbst nur als *Proxy* für den eigentlichen, entfernten Model-Service fungiert, d.h. also die Anfrage gemäß des Model-Bus Datenaustauschformats serialisiert und an die Zieladresse überträgt, um die von dort erhaltene Antwort wieder zu deserialisieren und an den Aufrufer zurückzugeben. Dabei lässt sich auch die im MDDi Projekt realisierte Dienstregistrierung nutzen, deren Aufruf bei der Konfiguration des entsprechenden Proxys nach seiner Erstellung erfolgt. Die dort bezogene Adresse wird im Proxy-Model-Service gespeichert und für die Ausführung der tatsächlichen Anfrage genutzt.

# Kapitel 7

# Zusammenfassung und Ausblick

Abschließend fasst dieses Kapitel die erbrachten Leistungen und die theoretischen Ergebnisse der vorliegenden Arbeit sowie die in der praktischen Realisierung des Werkzeugs gewonnen Erkenntnisse zusammen (vgl. Abschnitt 7.1). Darüber hinaus werden in Abschnitt 7.2 Fragestellungen aufgezeigt, welche nicht Teil der Aufgabenstellung vorliegender Arbeit waren, doch als thematisch ≫benachbart≪ anzusehen sind. Weiterhin werden Fragestellungen aufgeworfen, welche sich erst während der Durchführung der Arbeit ergeben haben, und als Anknüpfungspunkte für zukünftige Forschungen dienen können.

## 7.1 Zusammenfassung

In Kapitel 3 wurden die Anforderungen eines ≫Visualisierungswerkzeugs für Anwendungslandschaften≪ sowie die zentralen Nutzerrollen und Anwendungsfälle erhoben. Dabei wurden Erkenntnisse, die in ausgewählten Arbeiten zum Bereich des ≫Managements von Anwendungslandschaften≪ zu finden waren, mit einbezogen, soweit diese in spezifischen Anforderungen an das Werkzeug resultierten. Darüber hinaus wurden geeignete oder im Einsatzumfeld des Werkzeugs weit verbreitete Formate für Informationsmodellierung und Graphikverarbeitung angesprochen.

In Kapitel 4 wurde das Verfahren zur automatischen Generierung von Visualisierungen von Anwendungslandschaften, welches dem Werkzeug zugrunde liegt, kurz vorgestellt und hinsichtlich möglicher Realisierungs- bzw. Konkretisierungsansätze detailliert. Darüber hinaus wurden bezüglich spezieller Aspekte des Verfahrens, z.B. der Modelltransformation und ihres des gemeinsamen Metamodells, ausgewählte Arbeiten in diesem Bereich einbezogen, um die in der Realisierung zu treffenden Entscheidungen geeignet zu untermauern.

In Kapitel 5 wurde aufbauend auf einen, in einer vorhergehenden Arbeit erstellten Architekturentwurf eine plattformunabhängige Architektur für ein Visualisierungswerkzeug für Anwendungslandschaften entwickelt. Dabei konnte durch die Entscheidung für eine servicezentrische Architektur bestehend aus Model-Services und einem Model-Bus dem

Wunsch nach Flexibilität und Erweiterbarkeit geeignet Rechnung getragen werden. Ein Querbezug zu laufenden Projekten im MDA Umfeld wurde im Sinne einer thematischen Fundierung aufgebaut.

In Kapitel 6 wurde eine geeignete Plattform für die Realisierung von Kernkomponenten gefunden und hinsichtlich ihrer Unterstützung für die servicezentrische, plattformunabhängige Architektur untersucht. Dabei konnte gezeigt werden, wie Kernaspekte dieser Architektur, speziell die Model-Services geeignet und erweiterbar umgesetzt werden können. Der Zuschnitt der Komponenten, wie er in der plattformunabhängigen Architektur gewählt wurde, konnte durch die Realisierung ausgewählter Komponenten validiert werden. Diese Komponenten sind ein weiterer Teil des Ergebnisses der Arbeit, welches - neben vorliegender Ausarbeitung - die Umsetzung des beschriebenen Werkzeugs auf Basis der Eclipse Rich Client Plattform ist. Die Benutzerschnittstelle dieses Werkzeugs mit ihren verschiedenen Ansichten, sowie der Menüleiste mit den Knöpfen zum Auslösen der Transformation, bzw. der verschiedenen Layouter wird in Abbildung 7.1 dargestellt.

## 7.2 Ausblick

Von den erhobenen Anforderungen und vorgestellten Anwendungsfällen für ein Visualisierungswerkzeug für Anwendungslandschaften konnten im Umfang der Arbeit zahlreiche Aspekte nur rudimentär oder gar nicht adressiert werden. Dies trifft auf die Übertragung in die plattformspezifische Architektur und auf die Umsetzung des Werkzeugs gleichermaßen zu. Zukünftige Arbeiten könnten aufbauend auf die in vorliegender Arbeit entwickelten Grundlagen, ausgewählte noch nicht adressierte Aspekte realisieren. Prominent tritt dabei der Aspekt der Erstellung von neuen Visualisierungskonzepten, wie Gestaltungsmitteln und -regeln hervor. Für diesen Aspekt könnte auf Basis der in vorliegender Arbeit entwickelter Beschreibungssprache ein geeigneter Editor entwickelt werden.

Die Mächtigkeit der Komponente für die Modelltransformation zwischen semantischem und symbolischem Modell könnte erforscht werden, was zu einer konkreten Beschreibung des notwendigen Funktionsumfangs führen sollte. Diese Beschreibung wäre besonders im Hinblick auf die Verwendung intuitiv erfassbarer Sprachen für die Definition von Transformationsmodellen wichtig. Darüber hinaus wären Aspekte einer Editor-gestützten Definition von Transformationsmodellen zu adressieren.

Ebenfalls im Zusammenhang mit der Modelltransformation wären Aspekte der Umkehrung eines gegebenen Transformationsmodells von Interesse, insbesondere, wenn das Werkzeug für die graphische Modellierung von Informationsobjekten verwendet werden soll. Da die Modelltransformation auch eine essentielle Rolle im Bereich der MDA spielt, wären hier mögliche Querbezüge zu untersuchen.

Die im Rahmen der plattformunabhängigen Architektur vorgeschlagenen Komponenten, welche in vorliegender Arbeit nicht realisiert werden konnten, bieten Anknüpfungspunkte für Folgearbeiten und Forschungen. Dies betrifft im Besonderen das *System für regelbasier-*

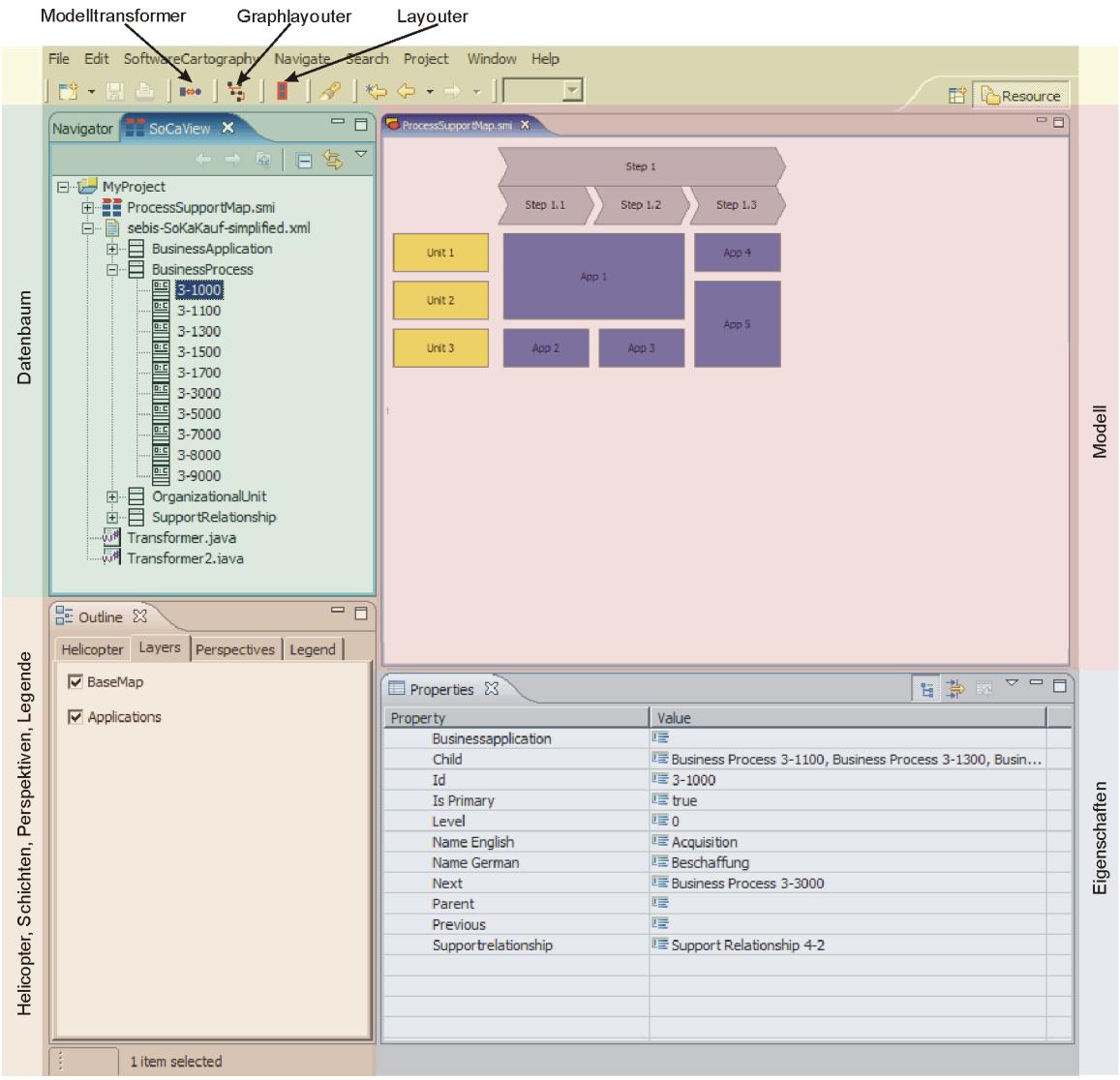



Abbildung 7.1: Benutzerschnittstelle des erstellten Werkzeugs

*tes Schließen*, in dessen Kontext logische Deduktionen auf semantischen und symbolischen Modellen untersucht und gegebenenfalls formalisiert werden könnten. Darüber hinaus bietet die gewählte Realisierungsplattform vielgestaltige Anknüpfungspunkte für interaktive Visualisierungen, in deren Zusammenhang auch die Entwicklung von geeigneten Konzepten für einen *Inverse-Layouter* untersucht werden müssten.

Die Realisierung der Komponenten *Optimierer* sowie die in vorliegender Arbeit verwendete Konkretisierung des Visualisierungsmodells sind prototypischer Natur, so dass im Hinblick auf eine zeiteffiziente Generierung von Visualisierungen alternative Konkretisierungen untersucht werden könnten. Dabei verdienen besonders Aspekte hinsichtlich der Komplexität des aus ihnen erwachsenden Optimierungsproblems, sowie seiner Problemklasse den Fokus der Forschung, da diese maßgeblich für den Einsatz effizienter Verfahren sind. In diesem Zusammenhang sei auf die laufende Arbeit von Ernst et. al. in [Er06] und die darin aufgeworfenen Forschungsfragen verwiesen.

Abschließend sei noch auf die zahlreichen Forschungsfragen verwiesen, die aus dem Einsatz dieses Werkzeugs bei einem industriellen Projektpartner des Forschungsprojekts Softwarekartographie erwachsen könnten. Besonders die Akzeptanz des Werkzeugs in der Praxis und seine Anwendbarkeit auf praxisrelevante Anwendungsfälle wären von Interesse, zum einen im Hinblick auf die Erhebung neuer oder der Verfeinerung bestehender Anforderungen, zum anderen mit Blick auf die Validierung des von Lankes et. al. vorgeschlagenen Ansatzes zur automatischen Generierung von Softwarekarten, wie er diesem Werkzeug zugrunde liegt.

# Anhang A

# Anforderungsanalyse - Anwendungsfälle und Benutzerschnittstelle

In diesem Anhang werden zwei Tabellen aus Abschnitt 3.4 nachgereicht, welche die Beziehungen zwischen den Anwendungsfällen (**UC**) und den Anforderungen an die Benutzerschnittstelle (**AB**) darstellen. Dabei repräsentiert das Symbol ***** die vollständige Abdeckung des in der entsprechenden Zeile aufgeführten Anwendungsfalls durch die zugehörigen Anforderungen an die Benutzerschnittstelle, wohingegen das Symbol **o**, auf eine teilweise Abdeckung verweist. Farblich hervorgehoben sind in der Tabelle die Anwendungsfälle **UC17**-**UC25**, die nicht direkt durch Teile der Benutzerschnittstelle abgedeckt, sondern, wie in Abschnitt 3.4 kurz illustriert, durch externe Werkzeuge zur objektorientierten Modellierung realisiert werden.

| | AB1 | AB2 | AB3 | AB4 | AB5a | AB5b | AB6a | AB6b | AB7 | AB8 | AB9 | AB10 | AB11 | AB12 | AB13 | AB14 | AB15 | AB16 | AB17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| UC1 | * | * | * | | | | | | * | | | | * | | | * | * | | |
| UC1a | * | | | | | | | | * | | | | * | | | * | * | | |
| UC1b | * | | | | | | | | * | | | | * | | | * | * | | |
| UC2 | | | | | | | | | | | | | | | | | | * | |
| UC3 | | | | | | | | | | | | | | | | | | | |
| UC4 | | | | | | | | | | | | | | | | | | | |
| UC5 | | | | | | | | | | | | | | | | | | | |
| UC6 | | | | | | | | | | | | | | | | | | | * |
| UC7 | | | * | * | * | * | * | * | | | | | | | | | | | |
| UC7a | | | * | * | | | | | | | | | | | | | | | |
| UC7b | | | * | | * | * | | | | | | | | | | | | | |
| UC7b | | | * | | | | * | * | | | | | | | | | | | |
| UC8 | | | | * | | * | | * | | | | | | | | | | | |
| UC9 | | | | | | | | | | | | | | | | | | | |
| UC10 | | | | | | | | | | | | | | | | | | | |
| UC11 | | | | | | | | | | | | | | | | | | | |
| UC12 | | | | | | | | | | | | | | | | | | | |
| UC13 | | | | | | | | | | | | | | | | | | | |
| UC14 | | | | | | | | | | | | | | | | | | | |
| UC15 | | | | | | | | | | | * | | | | | | | | |
| UC16 | | | | | | | | | | | | | | | | | | | |
| UC17 | | | | | | | | | | | | | | | | | | | |
| UC17a | | | | | | | | | | | | | | | | | | | |
| UC17b | | | | | | | | | | | | | | | | | | | |
| UC18 | | | | | | | | | | | | | | | | | | | |
| UC19 | | | | | | | | | | | | | | | | | | | |
| UC20 | | | | | | | | | | | | | | | | | | | |
| UC20a | | | | | | | | | | | | | | | | | | | |
| UC20b | | | | | | | | | | | | | | | | | | | |
| UC20c | | | | | | | | | | | | | | | | | | | |
| UC21 | | | | | | | | | | | | | | | | | | | |
| UC22 | | | | | | | | | | | | | | | | | | | |
| UC23 | | | | | | | | | | | | | | | | | | | |
| UC23a | | | | | | | | | | | | | | | | | | | |
| UC24 | | | | | | | | | | | | | | | | | | | |
| UC24a | | | | | | | | | | | | | | | | | | | |
| UC24b | | | | | | | | | | | | | | | | | | | |
| UC24c | | | | | | | | | | | | | | | | | | | |
| UC25 | | | | | | | | | | | | | | | | | | | |
| UC26 | | | | | | | | | | | | | | | | | | | |
| UC26a | | | | | | | | | | | | | | | | | | | |
| UC26b | | | | | | | | | | | | | | | | | | | |
| UC27 | | | | | | | | | | | | | | | | | | | |
| UC28 | | | | | | | | | | | | | | | | | | | |

Tabelle A.1: Beziehungen zwischen Anwendungsfällen (**UC**) und Anforderungen an die Benutzerschnittstelle(**AB**)

| | AB18 | AB19 | AB20 | AB21 | AB22 | AB23 | AB24 | AB25 | AB26 | AB27 | AB28 | AB29 | AB30 | AB31 | AB32 | AB33 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| UC1 | * | | | | * | | | | | | | | | | | |
| UC1a | * | | | | * | | | | | | | | | | | |
| UC1b | * | | | | * | | | | | | | | | | | |
| UC2 | | | | | | | | | | | | | | | | |
| UC3 | | * | | | | | | | | | | | | | | |
| UC4 | | | | | | | | | | | | | | * | | |
| UC5 | | | | | | | | | | | | | | | * | |
| UC6 | | | | | | | | | | | | | | | | |
| UC7 | | | | | | | | | | | | | | | | |
| UC7a | | | | | | | | | | | | | | | | |
| UC7b | | | | | | | | | | | | | | | | |
| UC7c | | | | | | | | | | | | | | | | |
| UC8 | | | | | | | | | | | | | | | | |
| UC9 | | | | | * | | | | | | * | * | * | | | |
| UC10 | | | | | | | * | | | | | | | | | |
| UC11 | | | | | * | | | | | | | | | | | |
| UC12 | | | | | | | | | | | | | | | | * |
| UC13 | | | | | * | | | | | | | | | | | |
| UC14 | | | | | | | | | | | | * | * | | | |
| UC15 | | | | | | | | * | | | * | | * | | | |
| UC16 | | | | | | | | | * | | * | | * | | | |
| UC17 | | | | | | | | | | | | | | | | |
| UC17a | | | | | | | | | | | | | | | | |
| UC17b | | | | | | | | | | | | | | | | |
| UC18 | | | | | | | | | | | | | | | | |
| UC19 | | | | | | | | | | | | | | | | |
| UC20 | | | | | | | | | | | | | | | | |
| UC20a | | | | | | | | | | | | | | | | |
| UC20b | | | | | | | | | | | | | | | | |
| UC20c | | | | | | | | | | | | | | | | |
| UC21 | | | | | | | | | | | | | | | | |
| UC22 | | | | | | | | | | | | | | | | |
| UC23 | | | | | | | | | | | | | | | | |
| UC23a | | | | | | | | | | | | | | | | |
| UC24 | | | | | | | | | | | | | | | | |
| UC24a | | | | | | | | | | | | | | | | |
| UC24b | | | | | | | | | | | | | | | | |
| UC24c | | | | | | | | | | | | | | | | |
| UC25 | | | | | | | | | | | | | | | | |
| UC26 | | o | o | | | | | | | | | | | | | |
| UC26a | | | | | | | | | | | | | | | | o |
| UC26b | | | | | | | | | | | | | | | | o |
| UC27 | | o | o | | | | | | | | | | | | | |
| UC28 | | * | | * | | | | | | | | | | | | |

Tabelle A.2: Beziehungen zwischen Anwendungsfällen (**UC**) und Anforderungen an die Benutzerschnittstelle(**AB**) (fortgesetzt)

# Anhang B

# Notation für mathematisch-logische Informationen im Visualisierungsmodell

Die für die Annotation von mathematisch-logischen Informationen zu Gestaltungsregeln im Visualisierungsmodell verwendete XML-Syntax soll in diesem Kapitel unter Nutzung einer EBNF-Notation (vgl. W3C z.B. in [W3C04a]) dargestellt werden. Dabei werden konventionsgemäß nicht-terminale Symbole unterstrichen notiert. Jede gültige Annotation zu einer Gestaltungsregel wird dabei durch den nicht-terminalen Typ Annotation repräsentiert.

```
Annotation ::=
  <definition type=''target''>RealExpression</definition> |
  <definition type=''constraint''>(BooleanExpression)+</definition>

BooleanExpression ::= <expression>BooleanTerm</expression>

RealExpression ::= <expression>RealTerm</expression>

BooleanTerm ::=
  <and>BooleanTerm (BooleanTerm)+</and> |
  <or>BooleanTerm (BooleanTerm)+</or> |
  <not>BooleanTerm</not> |
  <var name=''ID''/> | <const value=''BooleanValue''/> |
  <gr>RealTerm RealTerm</gr> |
  <geq>RealTerm RealTerm</geq>
```

```
RealTerm ::=
  <add>RealTerm (RealTerm)+</add> |
  <sub>RealTerm (RealTerm)+</sub> |
  <mult>RealTerm (RealTerm)+</mult> |
  <div>RealTerm (RealTerm)+</div> |
  <abs>RealTerm</abs> |
  <var name=''ID''/> | <const value=''RealValue''/>

BooleanValue ::= true | false

RealValue ::= ("-")?[1-9][0-9]*(",[0-9]*[1-9])? |
  ("-")?0(",[0-9]*[1-9])?

ID ::= [a-zA-Z][a-zA-Z0-9]*
```

In oben definierter Syntax werden mathematische oder logische Operationen durch Tags repräsentiert. Die Zuordnung kann aus Tabelle B.1 entnommen werden.

| Tagbezeichnung | Parameterbezeichnung | Beschreibung | Mathematisch-logische Interpretation |
|---|---|---|---|
| and | $b_1 ... b_n$:BooleanTerm | Repräsentiert das logische UND über alle Kindterme | $\bigwedge_{i=1}^{n} b_i$ |
| or | $b_1 ... b_n$:BooleanTerm | Repräsentiert das logische ODER über alle Kindterme | $\bigvee_{i=1}^{n} b_i$ |
| not | $b$:BooleanTerm | Repräsentiert die logische Negation des Kindterms | $\neg b$ |
| gr | $r_1, r_2$:RealTerm | Repräsentiert den strikten Vergleich der Kindterme | $r_1 > r_2$ |
| geq | $r_1, r_2$:RealTerm | Repräsentiert den Vergleich der Kindterme | $r_1 \geq r_2$ |
| add | $r_1 ... r_n$:RealTerm | Repräsentiert die SUMME über alle Kindterme | $\sum_{i=1}^{n} r_i$ |
| sub | $r_1 ... r_n$:RealTerm | Repräsentiert die DIFFERENZ vom ersten und der SUMME aller weiteren Kindterme | $r_1 - \sum_{i=2}^{n} r_i$ |
| mult | $r_1 ... r_n$:RealTerm | Repräsentiert das PRODUKT über alle Kindterme | $\prod_{i=1}^{n} r_i$ |
| div | $r_1 ... r_n$:RealTerm | Repräsentiert den QUOTIENTEN vom ersten und dem PRODUKT aller weiterer Kindterme | $r_1 / \prod_{i=2}^{n} r_i$ |
| abs | $r$:RealTerm | Repräsentiert den BETRAG des Kindterms | $\lvert r \rvert$ |

Tabelle B.1: Zuordnung zwischen Tags und mathematisch-logischen Operationen

# Anhang C

# Excel XML Spreadsheet Format

Der folgende Code ist aus einem semantischen Modell repräsentiert im Excel XML Spreadsheet Format entnommen und bezüglich der Verwendung von Namensräumen normalisiert worden. Repräsentiert wird hier ein Arbeitsbereich mit einer speziellen Dokumenteneigenschaft (»CustomDocumentProperty«), die auf das zugehörige Informationsmodell verweist. Das dargestellte Arbeitsblatt enthält Informationen über Instanzen der Klasse »Business Applications«, die nächste Zeile referenziert entsprechende Attribute der Klasse, eine konkrete Instanz folgt.

```
<?xml version="1.0"?>
<?mso-application progid="Excel.Sheet"?>
<Workbook xmlns:o="urn:schemas-microsoft-com:office:office"
  xmlns:dt="uuid:C2F41010-65B3-11d1-A29F-00AA00C14882"
  xmlns="urn:schemas-microsoft-com:office:spreadsheet">

  <o:CustomDocumentProperties>
    <o:metamodel dt:dt="string">
      http://www.softwarekartographie.de/socakauf.ecore
    </o:metamodel>
  </o:CustomDocumentProperties>

  <Worksheet Name="O1">
    <Table>
      <Row>
        <Cell>
          <Data Type="String">BusinessApplication</Data>
        </Cell>
      </Row>
      <Row>
        <Cell>
          <Data Type="String">id</Data>
```

```
        </Cell>
        <Cell>
          <Data Type="String">nameEnglish</Data>
        </Cell>
        <Cell>
          <Data Type="String">nameGerman</Data>
        </Cell>
        <Cell>
          <Data Type="String">standardConformity</Data>
        </Cell>
      </Row>
      <Row>
        <Cell>
          <Data Type="String">1-100</Data>
        </Cell>
        <Cell>
          <Data Type="String">Online Shop</Data>
        </Cell>
        <Cell>
          <Data Type="String">Online-Shop</Data>
        </Cell>
        <Cell>
          <Data Type="String">None</Data>
        </Cell>
      </Row>
    </Table>
  </Worksheet>
</Workbook>
```

# Literaturverzeichnis


[Ad97] Adler, G.: *An den Geschäftsprozessen ausrichten.* In *Diebold Management Report*, Jgg. Nr. 5, 1997.

[ASU86] Aho, A.; Sethi, R.; Ullman, J.: *Compilers - Principles, Techniques and Tools.* Addison-Wesley-Longman, Reading, Menlo Park, New York, 1986.

[AT05] ATLAS group - LINA & INRIA: *ATL: Atlas Transformation Language.* 2005. http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/subprojects/ATL/doc/ATL_User_Manual[v00.09].pdf (abgerufen am 2005-11-21).

[Ba01] Balzert, H.: *Lehrbuch der Software-Technik.* Spektrum, Akad. Verlag, Heidelberg, Berlin, 2. Auflage, 2001.

[Be04] Beyer, N.: *Kennzahlen zur Beschreibung von Anwendungslandschaften und ihre Visualisierung auf Softwarekarten.* Bachelor-Arbeit, Fakultät für Informatik, Technische Universität München, 2004.

[BESC04] Buchta, D.; Eul, M.; Schult-Croonenberg, H.: *Strategisches IT-Management.* Gabler Verlag, Wiesbaden, 2004.

[BL04] Buttelmann, M.; Lohmann, B.: *Optimierung mit Genetischen Algorithmen und eine Anwendung zur Modellreduktion.* In: at - Automatisierungstechnik 52, S.151–163, 2004.

[Bl05] Blanc, X.: *Model Bus - A MODELWARE Whitepaper.* http://www.eclipse.org/proposals/eclipse-mddi/main_data/ModelBusWhitePaper_MDDI.pdf (abgerufen am 2006-06-08), 2005.

[Br05] Brendebach, K.: *Integrierte Modelle und Sichten für das IT-Management.* Diplomarbeit, Fakultät für Informatik, Technische Universität München, 2005.

[Bu05] Buckl, S.: *Modell-basierte Transformationen von Informationsmodellen zum Management von Anwendungslandschaften.* Diplomarbeit, Fakultät für Informatik, Technische Universität München, 2005.

[Da00] Date, C.: *An introduction to database systems.* Addison-Wesley-Longman, Reading, Menlo Park, New York, 7. Auflage, 2000.

[Da05] Daum, B.: *Rich-Client-Entwicklung mit Eclipse 3.1.* dpunkt-Verlag, Heidelberg, 2005.

[Di03] Dietrich, J.: *The Mandarax Manual.* http://mandarax.sourceforge.net/docs/mandarax.pdf (abgerufen am 2006-06-13), 2003.

[Ec99] Ecma International: *Ecma Script Language Specification - 3rd Edition.* 1999. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf (abgerufen am 2006-05-29).

[Ec06] Eclipse community: *Rich Client Platform.* 2006. http://wiki.eclipse.org/index.php/RCP (abgerufen am 2006-05-29).

[El06] Elver Group:. *EMF Hibernate.* The Elver Group, 2006. http://www.elver.org/hibernate/index.html (abgerufen am 2006-06-23).

[Er06] Ernst, A. et al.: *Using Model Transformation for Generating Visualizations from Repository Contents - An application to Software Cartography.* Technischer Bericht, Technische Universität München, Lehrstuhl für Informatik 19 (sebis), 2006 (in Veröffentlichung).

[GA02] Gernert, C.; Ahrend, N.: *IT-Management: System statt Chaos - ein praxisorientiertes Vorgehensmodell.* Oldenbourg Verlag, München, 2. Auflage, 2002.

[Ha04] Halbhuber, T.: *Entwicklung eines Informationsmodells für das IT-Management.* Diplomarbeit, Fakultät für Maschinenwesen, Technische Universität München, 2004.

[HGM02] Hake, G.; Grünreich, D.; Meng, L.: *Kartographie.* de Gruyter Verlag, Berlin, New York, 8. Auflage, 2002.

[IB06] IBM Corporation: *Eclipse Platform Plug-in Developer Guide v3.1.* 2006. http://www.eclipse.org/downloads/download.php?file=/eclipse/downloads/drops/R-3.1-200506271435/org.eclipse.platform.doc.isv.3.1.pdf.zip (abgerufen am 09.06.2006).

[IE94] IETF: *Universal Resource Identifiers in WWW.* Internet Engineering Task Force, 1994. http://www.ietf.org/rfc/rfc1630.txt (abgerufen am 2006-05-30).

[IE00] IEEE: *IEEE Std 1471-2000 for Recommended Practice for Architectural Description of Software-Intensive Systems.* IEEE Computer Society, 2000.

[La05] Lauschke, S.: *Softwarekartographie: Analyse und Darstellung der IT-Landschaft eines mittelständischen Unternehmens.* Bachelor-Arbeit, Fakultät für Informatik, Technische Universität München, 2005.

[LMW05a] Lankes, J.; Matthes, F.; Wittenburg, A.: *Architekturbeschreibung von Anwendungslandschaften: Softwarekartographie und IEEE Std 1471-2000.* In (Liggesmeyer, P.; Pohl, K.; Goedicke, M., Hrsg.): *Software Engineering 2005*, P-64 von Lecture Notes in Informatics (LNI) - Proceedings, Essen, 2005.

[LMW05b] Lankes, J.; Matthes, F.; Wittenburg, A.: *Softwarekartographie: Systematische Darstellung von Anwendungslandschaften.* In (Ferstl, O. et al., Hrsg.): *Wirtschaftsinformatik 2005*, Bamberg, 2005.

[Mi03] Microsoft: *INFO: Microsoft Excel 2002 and XML (Q288215).* Microsoft Inc., 2003. http://support.microsoft.com/kb/288215/EN-US/(abgerufen am 2006-05-29).

[Mo04] Moore, B. et al.: *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework.* http://www.redbooks.ibm.com/redbooks/pdfs/sg246302.pdf (abgerufen am 2006-05-29), 2004.

[MR06] Meffert, K.; Rotstan, N.: *What is JGAP?* http://jgap.sourceforge.net/ (abgerufen am 2006-06-27), 2006.

[MW04] Matthes, F.; Wittenburg, A.:. *Softwarekarten zur Visualisierung von Anwendungslandschaften und ihren Aspekten - Eine Bestandsaufnahme.* Technischer Bericht TB0402, Technische Universität München, Lehrstuhl für Informatik 19 (sebis), 2004.

[OA05] OASIS: *Web Services Base Notification.* Organization for the Advancement of Structured Information Standards, 2005. http://docs.oasis-open.org/wsn/wsn-ws_base_notification-1.3-spec-pr-02.pdf (abgerufen am 2006-05-30).

[OM01] OMG: *MDA Guide Version 1.0.1.* Object Management Group, 2001.

[OM04] OMG: *MOF 2.0 Facility and Object Lifecycle Specification, ad/2004-04-02.* Object Management Group, 2004.

[OM05] OMG: *Meta Object Facility (MOF) 2.0 XMI Mapping Specification, v2.1.* Object Management Group, 2005.

[OM06] OMG: *Meta Object Facility (MOF) Core Specification, version 2.0, formal/06-01-01.* Object Management Group, 2006.

[Op04] OpenMath: *The OpenMath Standard.* The OpenMath Society, 2004. http://om-candidate.activemath.org/standard/om20-2004-06-30/omstd20.pdf (abgerufen am 2006-06-22).

[RR05] Rausch, K.; Rothe, A.: *Von der Industrie lernen - Steuerung der IT nach industriellen Maßstäben.* In (Ferstl, O. et al., Hrsg.): *Wirtschaftsinformatik 2005*, Bamberg, 2005.

[Sc05a] Schweda, C.: *Atomare und komplexe Darstellungsregeln der Softwarekartographie - Identifikation, Konzeption und prototypische Implementierung.* Abschlussvortrag Systementwicklungsprojekt, Technische Universität München, Fakultät für Informatik, 2005.

[Sc05b] Schweda, C.: *IEEE 1471: Rahmen, Begriffsdefinition und konzeptuelles Modell für Architekturdokumentation.* Seminararbeit, Technische Universität München, Fakultät für Informatik, 2005.

[se05] sebis: *Enterprise Architecture Management Tool Survey 2005.* Technische Universität München, Lehrstuhl für Informatik 19 (sebis), 2005.

[So01] Sommerville, I.: *Software Engineering.* Pearson Studium, München, 6. Auflage, 2001.

[St73] Stachowiak, H.: *Allgemeine Modelltheorie.* Springer-Verlag, Wien, 1973.

[WK94] WKWI: *Profil der Wirtschaftsinformatik.* In: Wirtschaftsinformatik 36(1), 1994.

[W3C03a] W3C: *Mathematical Markup Language (MathML) Version 2.0 (Second Edition).* World Wide Web Consortium, 2003. http://www.w3.org/TR/MathML2/ (abgerufen am 2006-06-22).

[W3C03b] W3C: *OWL Web Ontology Language.* World Wide Web Consortium, 2003. http://www.w3.org/TR/owl-features/(abgerufen am 2006-06-13).

[W3C03c] W3C: *Portable Network Graphics (PNG) Specification (Second Edition).* World Wide Web Consortium, 2003. http://www.w3.org/TR/PNG/ (abgerufen am 2006-05-29).

[W3C03d] W3C: *Scalable Vector Graphics (SVG) 1.1 Specification.* World Wide Web Consortium, 2003. http://www.w3.org/TR/SVG11/(abgerufen am 2006-05-29).

[W3C04a] W3C: *Extensible Markup Language (XML) 1.0 (Third Edition).* World Wide Web Consortium, 2004. http://www.w3.org/TR/2004/REC-xml-20040204/#sec-documents (abgerufen am 2006-07-03).

[W3C04b] W3C: *SOAP Version 1.2 Part 1: Messaging Framework.* World Wide Web Consortium, 2004. http://www.w3.org/TR/soap12-part1/(abgerufen am 2006-05-30).

[W3C04c] W3C: *Web Services Architecture.* World Wide Web Consortium, 2004. http://www.w3.org/TR/ws-arch/(abgerufen am 2006-05-30).

[Zi87] Zimmermann, H.:. *Operations Research - Methoden und Modelle.* Vieweg & Sohn Verlag, Braunschweig, 1. Auflage, 1987.