ulm university universität ulm

Technische Konzeption und Realisierung der Modellierungskomponente für ein datenorientiertes Prozess-Management-System

Andreas Pröbstle
andreas.proebstle@uni-ulm.de

Studiengang:	Informatik
1. Gutachter:	Prof. Dr. Manfred Reichert
2. Gutachter:	Prof. Dr. Peter Dadam
Betreuerin:	Dipl. Medieninf. Vera Künzle
begonnen am:	01.11.2010
beendet am:	28.04.2011

Inhaltsverzeichnis

Abbildungsverzeichnis

Codebeispiele

1
Einleitung

1.1. Motivation

Geschäftsprozessmanagement (BPM) verspricht eine höhere Flexibilität, Prozesskontrolle und Transparenz innerhalb komplexer Organisationen und zeigt sich somit als einer der wichtigsten Erfolgsfaktoren für die zukünftige Entwicklung. Jedoch zögern noch viele Unternehmen aus Mangel an Ressourcen oder Kompetenz mit der Einführung von BPM-Lösungen[1]. Allerdings zeigt sich ein großes Interesse an der Einführung von BPM in den Unternehmen. Die Notwendigkeit zum Einstieg zeigen auch Studien, die die richtige Beherrschung von BPM als Existenzvoraussetzung definieren [2].
In der Praxis zeigt sich jedoch das eingeschränkte Anwendungsfeld der herkömmlichen Prozess-Management-Software, die für fest durch Aktivitäten strukturierte Prozesse entworfen ist. Viele der heutigen Prozesse - gerade im Dienstleistungsbereich - sind jedoch nicht durch eine feste Abfolge von Aktivitäten geprägt. Vielmehr erfolgt die Ausführung der Arbeitsschritte flexibel und wird nur durch das Vorliegen der für einen Arbeitsschritt nötigen Daten bestimmt.
Eine aktuelle Entwicklung in diesem Bereich sind die datenorientierten Prozess-Management-Systeme. Diese beschreiben nicht feste Abfolgen der Bearbeitungsaktivitäten des Prozesses. Viel-

1 Laut einer Studie von trovarit [1] setzen 75% der Unternehmen bisher noch keine BPM Lösung ein

mehr bieten sie die Flexibilität Aktionen vorzunehmen sobald ihre Vorbedingungen erfüllt sind und die Ausführung der Aktion nicht der Konsistenz des Prozesses schadet.

1.2. Entwicklung eines Modellierungsprogrammes

Ziel dieser Arbeit ist die Implementierung eines Modellierungsprogrammes für das datenorientierte Prozess-Management-System PHILharmonicFlows[1] auf der Basis eines bestehenden Usability-Konzepts. Mit der Modellierungsumgebung soll der Benutzer graphisch unterstützt die Modelle für Prozesse entwerfen, aus denen zur Laufzeit automatisch die Aufgabenlisten und Datenformulare generiert werden. Da bei der Modellierung viele Regeln und Vorgehensweisen beachtet werden müssen ist es wichtig den Benutzer dabei zu unterstützen. Dies passiert sowohl durch verständliche Hinweise und Fehlermeldungen wie auch durch aktive Benutzerführung, indem fehlerhafte Vorgehensweisen erst gar nicht angeboten oder automatisch korrigiert werden. Für einen flexiblen und längerfristigen Einsatz soll die Modellierungsumgebung zudem möglichst plattformunabhängig implementiert und für spätere Änderungen und Erweiterungen offen konzipiert sein.

1.3. Struktur der Arbeit

Zunächst werden in Kapitel 2 die Grundlagen des PHILharmonicFlows Konzepts näher erläutert und anhand eines Beispiels deren praktischer Einsatz gezeigt. Im Anschluss gibt Kapitel 3 einen Überblick über die verwendeten Technologien zur Umsetzung. Außerdem werden Informationen über grundlegende verwendete Konzepte bei der Implementierung gegeben. In Kapitel 4 erfolgt eine kurze Einführung in die entwickelte Oberfläche um die Details der Implementierung besser verstehen und in den Gesamtkontext einordnen zu können. Anschließend erfolgt in Kapitel 5 eine Erläuterung der Implementierung. Dabei werden sowohl die Datenklassen als auch der Oberflächenentwurf beleuchtet. Die dabei aufgetretenen Abweichungen vom Oberflächenkonzept und deren Hintergründe wie auch die Erweiterungsschnittstellen werden in Kapitel 6 beschrieben.

1 Process, Humans and Information Linkage for harmonic Business Flows.

2

Grundlagen PHILharmonicFlows

Das Ziel des PHILharmonicFlows Projekts ist die Entwicklung eines datenorientierten Prozess-Management-Systems (PMS). Herkömmliche Prozess-Management-Systeme definieren Prozesse als Abfolge von Aktivitäten in denen jeweils festgelegte Arbeitsvorgänge erfolgen. Der Austausch der Anwendungsdaten zwischen den Aktivitäten erfolgt über einen parallel zum Prozessfluss entwickelten Datenfluss, der den Aktivitäten Ein- und Ausgabewerte zuordnet.
Umständlich ist bei dieser Vorgehensweise die Änderung von Daten außerhalb des normalen Prozessfortschritts. Beispielsweise kann sich in einem Bewerbungsprozess die Adresse eines Bewerbers zu jedem Zeitpunkt während des Prozessablaufs ändern. Da der Zugriff auf die Anwendungsdaten für Benutzer nur über Aktivitäten möglich ist, muss für alle möglichen Änderungen eine Aktivität bei der Modellierung berücksichtigt werden, was die Prozesse stark verkompliziert und der Übersicht schadet. PHILharmonicFlows stellt dagegen die Anwendungsdaten in den Mittelpunkt. Diese bestimmen direkt durch ihr Vorhandensein den Prozessfortschritt, wodurch ein ständiger integrierter Zugang für Datenänderungen und Prozessablaufsteuerungen ermöglicht wird.

2.1. Aufbau von PHILharmonicFlows

Das Framework teilt sich in die Modellierungs- und die Laufzeitumgebung auf. Für die weitere Betrachtung legen wir den Fokus auf die Modellierungsumgebung, deren Entwicklung Aufgabe dieser Arbeit ist. Die Modellierungsumgebung setzt sich dabei aus drei Hauptkomponenten zusammen. Diese ermöglichen die Modellierung der Datenstruktur, den Entwurf der Prozessstruktur und die Vergabe der Benutzerrechte. Die Aufteilung der Hauptkomponenten und die ihnen zugeordneten Funktionen können der Abbildung 1 entnommen werden.

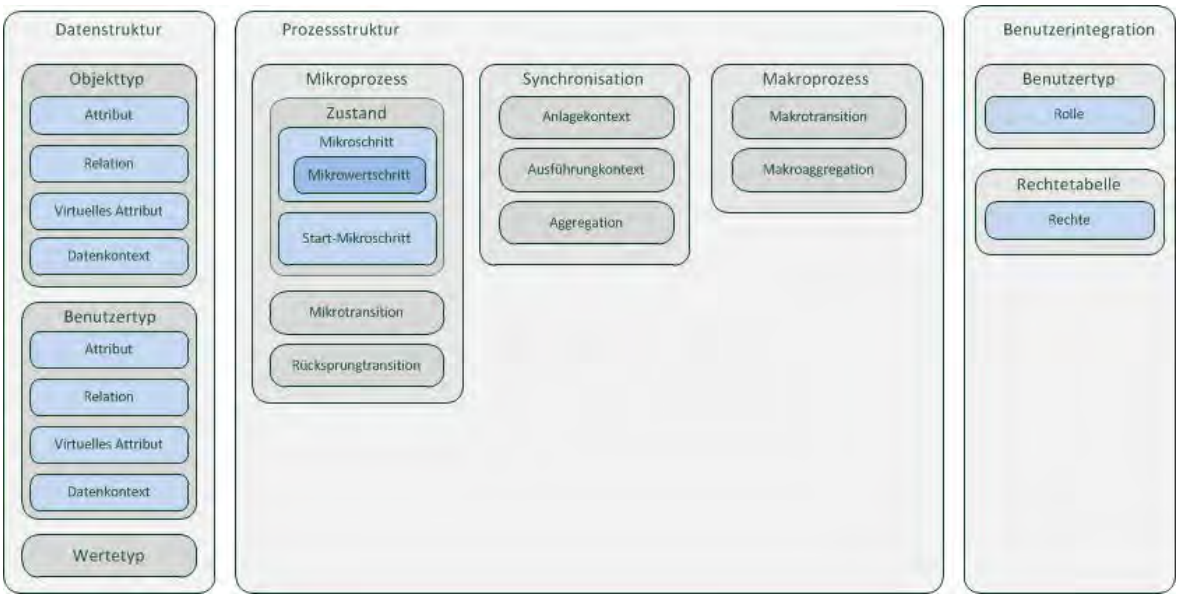

Abbildung 1.: Die Hauptkomponenten und ihre Funktionen (nach [17])

2.1.1. Datenstruktur

Die Datenstruktur definiert die Anwendungsdaten des Projekts. Sie wird durch die *Objekttypen*, deren *Attributen* und *Relationen* gebildet und kann anhand eines relationellen Datenmodells beschrieben werden. *Objekttypen* dienen dabei als Container für zusammengehörige *Attribute* und sind durch Schlüssel eindeutig identifizierbar. Die *Attribute* sind als Felder für einzelne Datenwerte konzipiert und werden durch einen gewählten Datentyp klassifiziert und einem Namen zur Anzeige definiert.
Einige der modellierten *Objekttypen* können zueinander in Beziehung stehen. Diese Beziehungen können mittels gerichteter *Relationen* dargestellt werden, die somit die Ab-

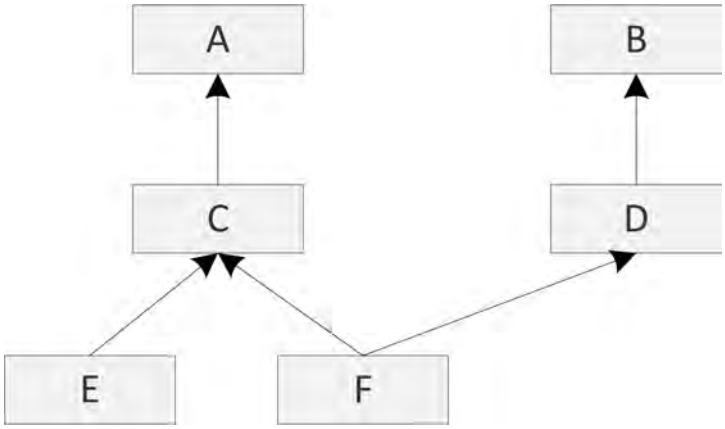

Abbildung 2.: Beispiele für zyklenfreie Datenstruktur

hängigkeit der Objektinstanzen zur Laufzeit anhand von 1:n Beziehung beschreiben. Mittels der *Relationen* lassen sich die *Objekttypen*, sofern sich durch die *Relationen* kein Zyklus ergibt, hierarchisch nach Ebenen anordnen (siehe Abbildung 2). Der obersten Ebene gehören dabei diejenigen *Objekttypen* an, die sich nicht auf andere beziehen. Beim Auftreten eines Zyklus wie links in Abbildung 3 zwischen A, C und F muss dieser vor der Ebenenermittlung aufgelöst werden. Dazu wird eine *Relation* als Zyklusrelation markiert und damit für die Berechnung der Ebenen nicht mehr als ausgehende Beziehung berücksichtigt. Somit kann der Zyklus aufgelöst werden und es ist eine hierarchische Anordnung wie im rechten Teilbild von Abbildung 3 zu sehen möglich, in der die Relation *von* A nach F markiert wurde. Bei vielen Prozessen ist es notwendig auf bestimmte Ereignisse, also der Belegung bestimmter *Attribute* mit speziellen Werten, gesondert zu reagieren. Diese Möglichkeit wird durch die *Datenkontexte* ermöglicht. Zur Definition der *Datenkontexte* werden logische Ausdrücke

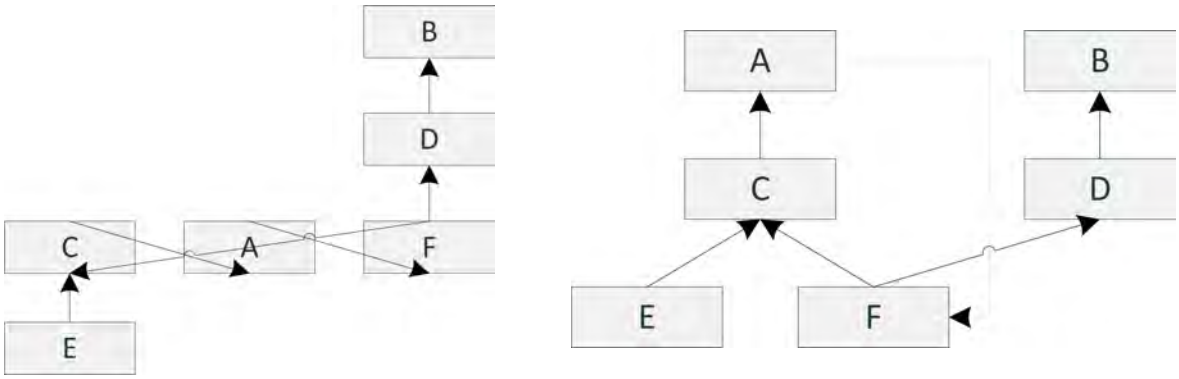

Abbildung 3.: Beispiele für Zyklus Datenstruktur

für die einzuschränkenden Attribute des *Objekttyps* erstellt, anhand derer zur Laufzeit die Eingaben überprüft werden können. Für die Definition der *Datenkontexte* kann es notwendig sein auf ein *Attribut* eines übergeordneten *Objekttyps* zuzugreifen um den logischen Ausdruck auf bestimmte Fälle einzuschränken. Um dies zu ermöglichen wurde das Konzept des *virtuellen Attributs* eingeführt. Dieses kann in einem *Objekttyp* angelegt werden und bietet lesenden Zugriff auf ein ausgewähltes *Attribut* eines übergeordneten *Objekttyps* und kann damit als lediglich lesbarer Zeiger verstanden werden.
Eine Sonderform der *Objekttypen* sind die *Benutzertypen*. Diese stellen Benutzer dar, die mit dem System interagieren können und verwalten somit noch die Zugriffsrechte und Änderungsbefugnisse, die in der Hauptkomponente Benutzerintegration verwaltet werden. Innerhalb der Datenstruktur werden die *Benutzertypen* jedoch ebenso wie die *Objekttypen* behandelt. Eine zweite Sonderform stellen die *Wertetypen* da. Diese können jedoch nicht in Relation zu anderen *Objekttypen* gesetzt werden und gehören keiner Datenebene an. Sie definieren stattdessen Wertebereiche für *Attribute*, die zur Laufzeit ausgewählt werden können.

2.1.2. Prozessstruktur

In der *Prozessstruktur* wird der Ablauf des modellierten Prozesses definiert. Diese setzt sich aus den beiden Teilbereichen *Mikro-* und *Makroprozess* zusammen.

Mikroprozess

Jedem Objekttyp ist ein *Mikroprozess* zugeordnet, der dessen Ausführungsverhalten beschreibt. *Mikroprozesse* werden durch *Mikroschritte*, *Transitionen* und *Zustände* gebildet. *Mikroschritte* definieren dabei Bedingungen, die für das Fortschreiten des Prozesses notwendig sind. Dabei kann zwischen *leeren*, *atomaren* und *wertespezifischen Prozessschritten* unterschieden werden. *Leere Mikroschritte* stellen dabei keine Bedingung und können somit zur Laufzeit immer erreicht werden. *Atomaren Mikroschritten* sind *Attribute* zugeordnet. Diesem zugeordneten *Attribut* muss ein Wert zugewiesen werden, damit der *atomare Mikroschritt* erreichbar ist. *Wertespezifische Mikroschritte* besitzen ebenso ein zugeordnetes Attribut. Jedoch reicht es hier nicht aus, dass dem Attribut ein beliebiger Wert zugewiesen wird. Ein *wertespezifischer Mikroschritt* erfordert einen bestimmten Wert für das Attribut um erreichbar zu sein. Dieser geforderte Wert wird in einem *Mikrowertschritt* des *Mikroschritts* definiert.

Für eine Untergliederung des *Mikroprozesses* sind zusammengehörige *Mikroschritte* in *Zuständen* zusammengefasst. Dabei erhält jeder Zustand einen die Änderungen beschreibenden Namen.
Mikrotransitionen bilden die möglichen Übergänge zwischen den *Mikroschritten* ab und bilden zusammen für jeden *Mikroprozess* einen gerichteten Graphen von dem *Startmikroprozessschritt* zu den *Endmikroprozessschritten*. Es kann zwischen *internen, externen impliziten* und *externen expliziten Transitionen* unterschieden werden. Falls Quelle und Ziel einer *Transition* demselben *Mikrozustand* angehören, spricht man von einer *internen Transition*. Diese sind immer als *implizit* eingestuft. Das heißt sie können zur Laufzeit schalten, sobald der folgende *Mikroschritt* erreichbar ist. *Externe Transitionen* verbinden dagegen *Mikroschritte* verschiedener *Zustände* miteinander. *Implizite externe Transitionen* können dabei genau wie *interne Transitionen* schalten, sobald der folgende *Mikroschritt* erreichbar ist. *Externe explizite Transitionen* benötigen dagegen die Bestätigung eines berechtigten Benutzers um zu schalten. Jeder *Mikroschritt* kann mehrere ausgehende *Transitionen* besitzen. Da zur Laufzeit jedoch nur eine *Transition* schalten darf ist jeder *Transition* eine eindeutige Prioritätsstufe zugeordnet. Dadurch ist ein eindeutiger Ablauf gewährleistet, da bei mehreren erfüllbaren *Transitionen* nur die erfüllbare mit der höchsten Priorität geschaltet wird.
Schleifentransitionen ermöglichen den Rücksprung zu anderen vorhergehenden *Zuständen* innerhalb des *Mikroprozesses*. *Schleifentransitionen* verbinden im Gegensatz zu anderen *Transitionen* zwei *Zustände* miteinander.
Jeder *Mikroprozess* besitzt mindestens einen *Startzustand* mit dem *Startmikroschritt* und beliebig viele *Endmikrozustände*, die jeweils nur einen *Endmikroschritt* enthalten dürfen. Mit Erreichen des *Endzustands* ist der *Mikroprozess* beendet.

Synchronisation der Mikroprozesse

Innerhalb der Datenstruktur sind die Beziehungen der Objekttypen untereinander definiert. Diese Abhängigkeiten besitzen zur Laufzeit ebenfalls Auswirkungen auf das Verhalten der Mikroprozesse untereinander. So sind Mikroprozessinstanzen höherer Ebenen oft von der Ausführung der Instanzen der unteren Ebenen abhängig. Aus diesem Grund müssen die Mikroprozesse des Gesamtprozesses untereinander synchronisiert werden.
Mittels der *Prozesskontexttypen* kann dabei festgelegt werden, dass die untergeordneten Mikroprozesse nur angelegt werden dürfen, solange sich der übergeordnete Mikroprozess in bestimmten Zuständen aufhält. Im Beispiel in Abbildung 4 ist der Beginn und das Ende des

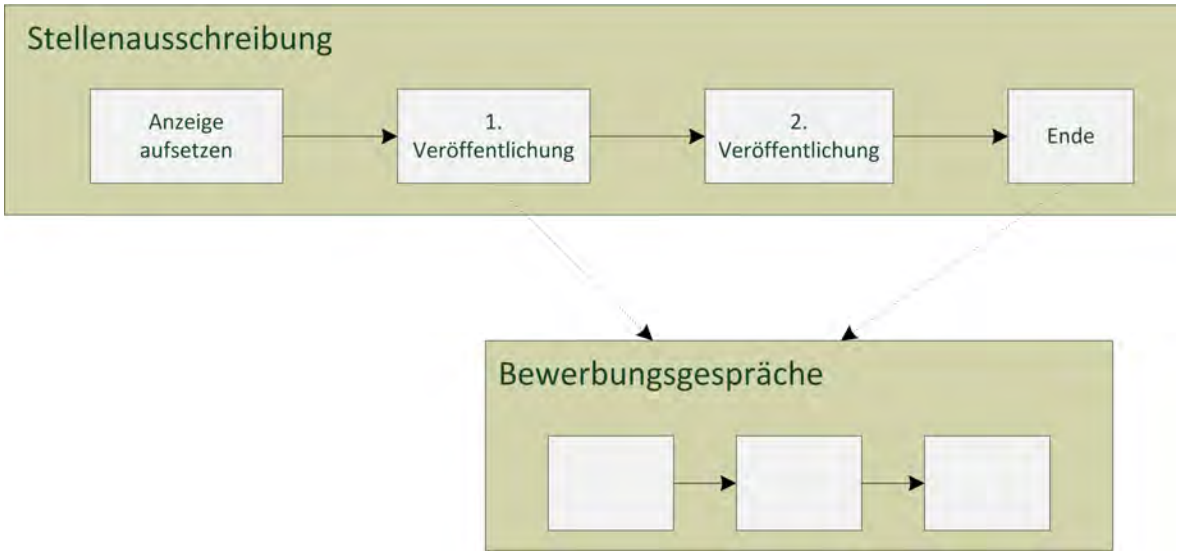

Abbildung 4.: Prozesskontext von Mikroprozess Instanzen

Prozesskontexttyps durch einen gestrichelten Pfeil zwischen den Zuständen Stellenausschreibung und Bewerbungsgespräch dargestellt. Dabei darf beispielsweise ein Bewerbungsgespräch nur zwischen der 1. Veröffentlichung und dem Ende der Bewerbersuche begonnen werden.
Mikroprozesse sind oft auf Informationen von untergeordneten Mikroprozessinstanzen angewiesen. Da eine Mikroprozessinstanz viele untergeordnete Instanzen besitzen kann müssen diese zusammengefasst werden. Dazu dient der *Aggregationstyp*. Die Aggregation erfolgt dabei nach dem Fortschritt der untergeordneten Prozesse zu einem ausgewählten Zustand.

Makroprozess

Für die Koordination der Zustände verschiedener Objekttypen untereinander gibt es die *Makroprozesse*. Jeder *Makroschritt* ist dabei einem Zustand eines Objekttyps zugeordnet. Somit können durch *Makrotransitionen* zwischen den *Makroschritten* die Abhängigkeiten zwischen Zuständen verschiedener Objekttypen dargestellt werden. Jeder *Makroprozess* besitzt genau einen Start*makroschritt* und mindestens einen *Endmakroschritt*.
Das Konzept der *Makroinputs* ermöglicht die Einmündung verschiedener *Makrotransitionen* mit verschiedener Semantik in einen Zustand. Dabei ist festgelegt, dass für *Makrotransitionen*, die in denselben Makroinput münden eine „and"-Semantik gilt und zwischen verschiedenen Makroinputs eine „or"-Semantik gilt.
Makrotransitionen haben als Ursprung also einen Makroschritt und enden in einem *Makroinput*, der zu einem Makroschritt gehört. *Rückwärtsgerichtete Transitionen* werden

bei der Konstruktion als Schleifen erkannt und ausgezeichnet. Sie sind als einzige ausgehende *Transitionen* aus den *Endmakroschritten* und als eingehende *Makrotransitionen* für den *Startmakroschritt* erlaubt.
Makrotransitionen werden anhand der Beziehung der referenzierten *Objekttypen* ihrer Quelle und ihres Ziels in *„top-down“*, *„bottom up“*, *„transverse“* und *„self“* eingeteilt. Dabei beschreibt eine *„top down“* Beziehung, dass der Objekttyp des Quell*mikrozustands* dem Objekttyp des Ziel*mikrozustands* direkt oder indirekt übergeordnet ist und *„bottom-up“* das Gegenteil davon. *„Transverse“* bedeutet, dass sich die beiden beteiligten *Objekttypen* auf einen gemeinsamen Objekttyp zurückführen lassen, jedoch nicht direkt aufeinander. *„Self“* bedeutet, dass Quelle und Ziel der *Makrotransition* derselbe Objektktyp sind. Somit ergibt sich für *Makrotransitionen* die Einschränkung, dass sich die Objekttypen von Quelle und Ziel auf einen gemeinsamen Objekttyp zurückverfolgen lassen müssen.

2.1.3. Benutzerintegration

Mit der Definition eines *Objekttyps* als *Benutzertyp* wird eine neue Benutzerrolle angelegt. Zur Laufzeit steht jede ihrer Instanzen für einen entsprechenden Anwender.
Benutzerrollen können anhand von Rechtetabellen Lese- und Schreibrechte auf die *Attribute* der *Objekttypen* zugeordnet werden. Diese können in *obligatorische* und *optionale* Rechte unterschieden werden. Dabei wird vom System überwacht, dass für jedes *Attribut* mindestens eine Benutzerrolle die Schreibrechte erhält um Deadlocks zu vermeiden.

2.2. Definition eines Anwendungsbeispiels

Um ein besseres Verständnis des Konzepts zu erlangen soll dieses noch anhand eines praktischen Beispiels anschaulicher dargestellt werden. Als Anwendungsbeispiel wird ein sehr vereinfachter Bewerbungsprozess verwendet (frei nach [6]). Die einzelnen Stationen des Prozesses sind dabei vereinfacht in Abbildung 5 dargestellt.
Dabei wird eine Stelle von einer Firma ausgeschrieben für die sich Bewerber mit einem Online-Formular bewerben können. Zur Entscheidungsfindung holt die Personalabteilung von einem Mitarbeiter der zuständigen Fachabteilung eine Beurteilung der Bewerbung ein. Die Beurteilung enthält dabei eine Empfehlung ob der Bewerber zu einem Vorstellungsgespräch eingeladen

werden soll. Falls die Empfehlung positiv ist erhält der Bewerber anschließend eine Einladung zum Vorstellungsgespräch.

2.3. Einordnung des Anwendungsbeispiels

Zur Anschauung sollen nun die Einheiten aus dem in Kapitel 2.2 beschriebenen Anwendungsbeispiel auf das Konzept übertragen werden. Dabei beschränkt sich die Einordnung auf die in der Arbeit behandelten Themen der Datenstruktur und der Mikroprozesse. Ausgangspunkt jedes PHILharmonicFlows Prozess ist die Datenstruktur. Diese setzt sich aus den Objekttypen zusammen. Im Beispiel sind die Stellenausschreibung, die Bewerbung, das Vorstellungsgespräch und die Beurteilung die Objekttypen. Benutzertypen sind der Bewerber und der Mitarbeiter. Die modellierte Datenstruktur ist dabei in Abbildung 6 dargestellt. Bewirbt sich ein Bewerber auf eine Stelle wird eine neue Bewerbung angelegt. Nachdem geprüft worden ist ob die Bewerbung die Anforderungen der Stelle erfüllt erfolgt die Beurteilung durch einen Mitarbeiter der Fachabteilung und bei einer Empfehlung die Einladung zum Vorstellungsgespräch. Um bei der Erstellung der Beurteilung Redundanzen zu vermeiden muss festgestellt werden, ob es ein Wiederbewerber ist. Dafür muss Beurteilung ein virtuelles Attribut auf den entsprechenden Wert in Bewerbung besitzen. Dieser Spezialfall eines Wiederbewerbers kann durch einen Datenkontext abgefangen und gesondert behandelt werden.
Für die Stellenausschreibung werden Wertetypen definiert. Die Wertetypen beschreiben dabei die Anforderungen für eine zugehörige Stelle.
Als Beispiel für den Mikroprozess wollen wir den des Objekttyps Bewerbung herausgreifen. Als erster Schritt erfolgt die Eingabe der Daten im Formular (`Erfassung`), anschließend erfolgt die Bestätigung der Daten (`Bestätigung`). Diese Schritte sind im Zustand `Bewerbung erfassen` zusammengefasst. Anschließend wird auf das Ergebnis der Beurteilung gewartet (`Beurteilung einholen`). Wenn das Ergebnis der Beurteilung eine Empfehlung war, schaltet der wertespezifische Mikroschritt `Beurteilung` als wahr und der Bewerber wird zu einem Vorstellungsgespräch eingeladen (`Vorstellungsgespräch einladen`). Ansonsten wartet der Prozess, da der Endzustand `Ablehnung` nur durch eine explizite Transition verbunden ist, da eine Ablehnung von einem Mitarbeiter der Personalabteilung persönlich autorisiert werden muss. Im Falle eines erfolgreichen Vorstellungsgesprächs schaltet der wertspezifische Mikroschritt `Einstellung` als wahr und der Prozess endet im Endzustand `Zusage`, ansonsten endet er im Endzustand `Ablehnung`.

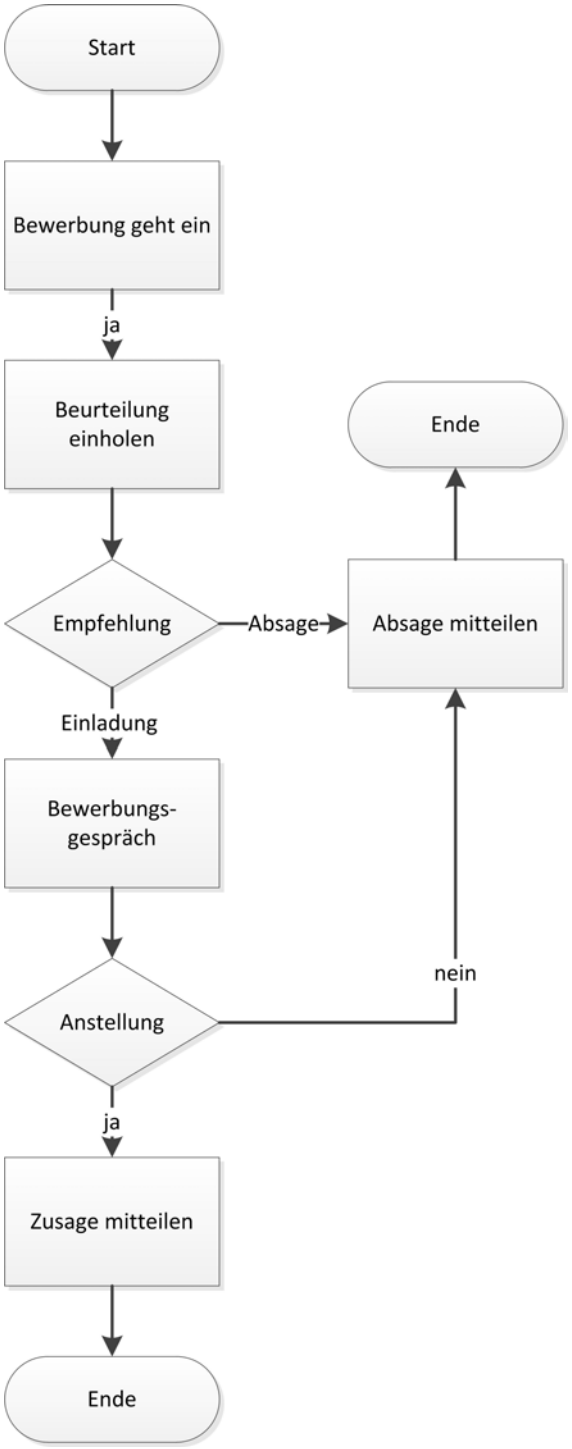

Abbildung 5.: Der Prozessablauf des Beispielprozesses

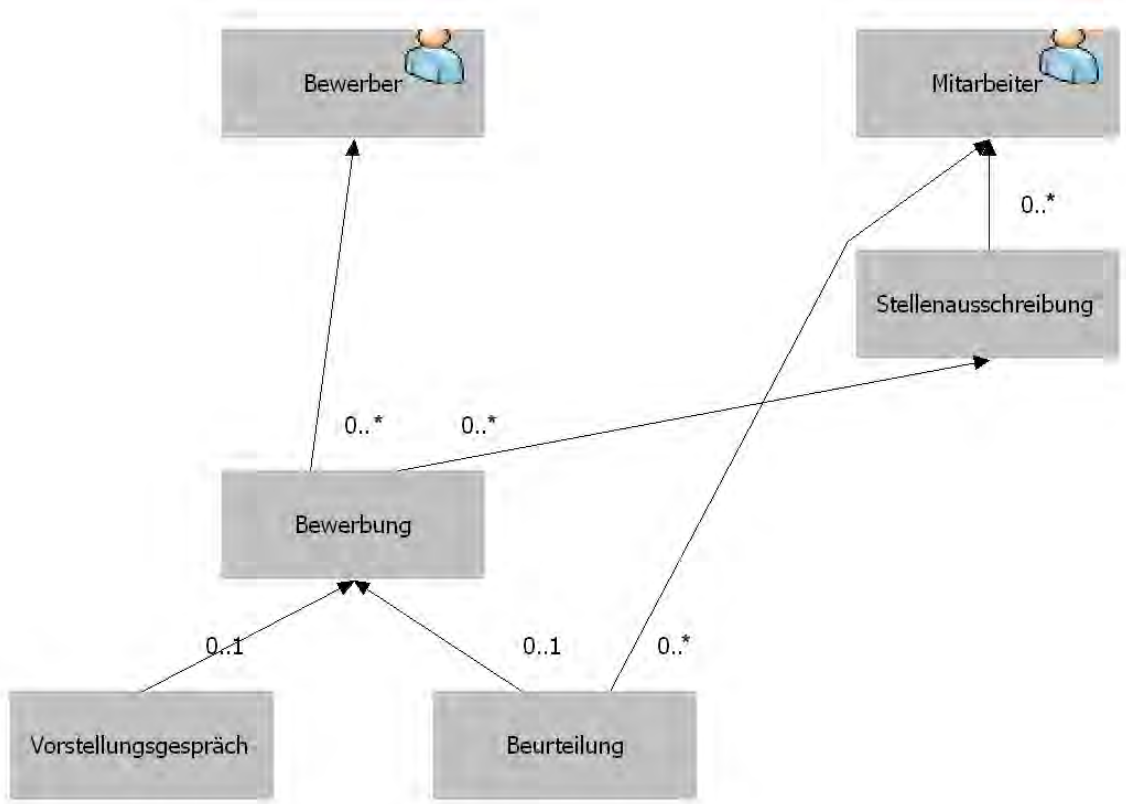

Abbildung 6.: Datenstruktur des Beispielprozesses

3
Technische Grundlagen

Dieses Kapitel soll die Entscheidungen für die bei der Implementierung gewählten Technologien erläutern und diese einführen. Darüber hinaus werden einige der wichtigsten verwendeten Konzepte vorgestellt.

3.1. Auswahl des Frameworks

Als Programmiersprache für die Implementierung war von Beginn an Java festgelegt. Als mögliche Bibliotheken für die graphenbasierten Editoren bieten sich damit die beiden quelloffenen Bibliotheken JGraph[16] und Eclipse GEF[13] an. Dabei fiel die Wahl nach der Evaluation der beiden auf das Eclipse GEF. Ausschlaggebend für diese Entscheidung war die bessere Dokumentation und Unterstützung durch Foren und Tutorials für GEF. Außerdem bietet GEF eine gute Integration innerhalb der Eclipse RCP Umgebung, die somit als Plattform für die Entwicklung des Frameworks genutzt werden konnte.
Bei Eclipse besteht derzeit die Wahl zwischen der Eclipse 3 Generation und Eclipse 4. Eclipse 4 stellt dabei eine flexiblere und durch Cascading Style Sheets (CSS) anpassbare Oberfläche zur Verfügung. Jedoch liegt der Fokus der Entwicklung noch auf der Eclipse 3 Reihe, sodass viele Plug-Ins wie zum Beispiel auch das GEF Eclipse 4 noch nicht unterstützen. Somit wird

die Eclipse 3 Generation in Verbindung mit dem GEF zur Implementierung von PHILharmonicFlows genutzt.

3.2. Das Eclipse Framework

In diesem Kapitel erfolgt eine Betrachtung der Technologien, die als Grundlage für die Implementierung des PHILharmonicFlows Rahmenwerks dienen. Grundlegendes Programm dabei ist Eclipse. Einerseits wird es in seiner Ausprägung als integrierte Entwicklungsumgebung (Eclipse IDE) zur Implementierung der Klassen genutzt. Darüber hinaus wird die Eclipse Rich Client Platform (RCP) mithilfe des Graphical Editing Frameworks (GEF) für die Darstellung der graphischen Benutzeroberfläche verwendet.

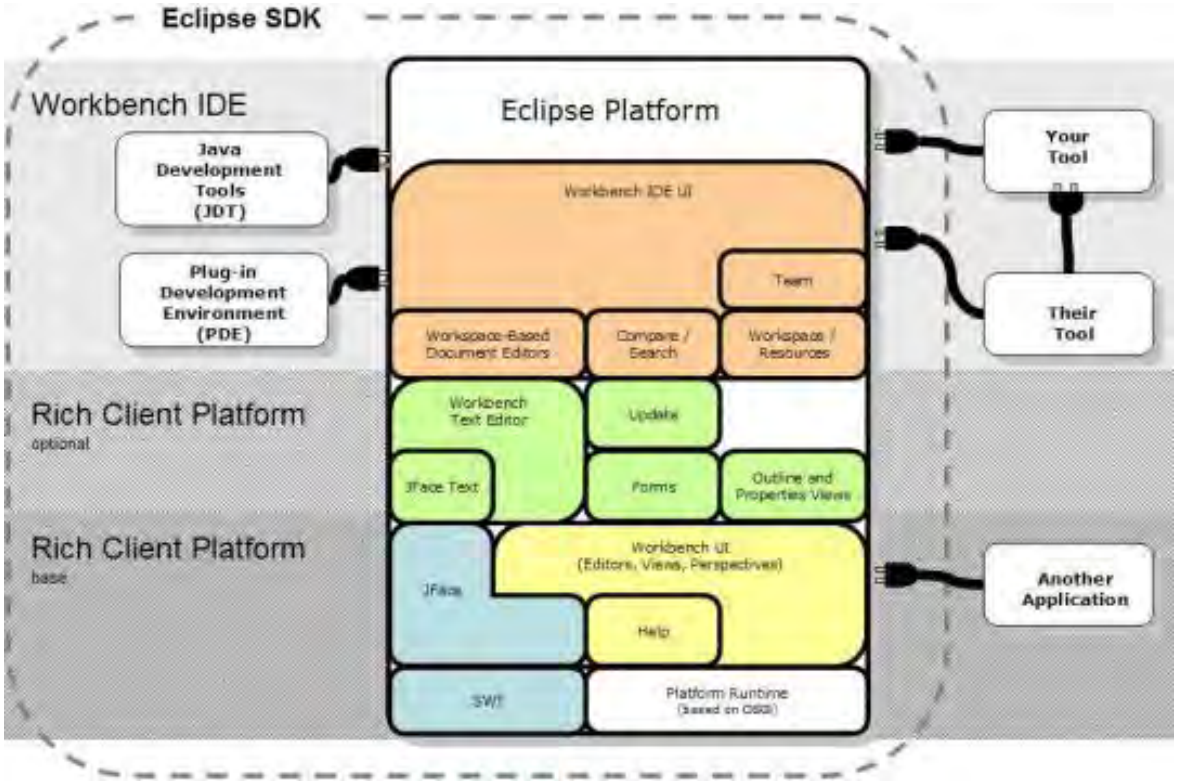

Abbildung 7.: Eclipse Architektur

3.2.1. Eclipse

Eclipse[11] ist eine quelloffene Softwareentwicklungsumgebung, die aus einer integrierten Entwicklungsumgebung (IDE) und einem erweiterbaren Plug-In System besteht. Ursprünglich von IBM als IDE für Java entwickelt wurde Eclipse 2001 schließlich in ein Open-Source Projekt überführt. Seit 2004 ist die Eclipse Foundation für die Weiterentwicklung zuständig [3].
Nachdem Eclipse zunächst nur als erweiterbare IDE konzipiert war, wurde mit Eclipse 3.0 ein Systemwechsel auf eine OSGi Service Platform basierende Architektur vollzogen [4]. Seitdem basiert Eclipse mit „Equinox“ auf einem Kern, der mithilfe von Plug-Ins die Funktionalitäten zur Verfügung stellt [siehe Abbildung 7]. Eclipse ist somit seitdem keine hartcodierte Gesamtlösung, sondern eine flexible, komponentenbasierte Lösung, die durch Hinzufügen weiterer Plug-Ins in den Plug-In Ordner der Eclipse Umgebung erweitert und angepasst werden kann. Die Plug-Ins sind vollständig in Java realisiert.
Für die Erstellung der Oberflächen nutzt Eclipse das eigens entwickelte schwergewichtige Standard Widget Toolkit (SWT). Schwergewichtig heißt, dass es zur Darstellung - ähnlich dem Abstract Window Toolkit (AWT) aus der Java Standardbibliothek und im Gegensatz zu Swing (ebenfalls in der Java Standardbibliothek), welches auf jeder Plattform gleich aussieht - die grafischen Elemente des Betriebssystems nutzt. Somit ist Eclipse nicht direkt plattformunabhängig, wird jedoch für die meisten Systeme und Architekturen bereitgestellt[1] und wird daher ebenfalls als plattformunabhängig bezeichnet. Je nach Verwendungszweck gibt es für die verbreitetsten Systeme, Windows, Linux und Macintosh, vorkonfigurierte Eclipse Pakete zum Download. Die derzeit aktuelle Eclipse Distribution ist Eclipse Helios (3.6.2) von Februar 2011.

3.2.2. Eclipse Rich Client Platform

Die Eclipse Rich Client Platform[12] (Eclipse RCP) umfasst den Eclipse Kern und diejenigen Plug-Ins, die für eine lauffähige Anwendung notwendig sind. Sie bietet damit eine Grundlage für die Entwicklung beliebiger eigenständiger Anwendungen, sogenannter Rich Client Applications, auf Basis eines OSGi-Frameworks [5]. Die so erzeugten Anwendungen sind plattformübergreifend auf allen Systemen und Architekturen nutzbar, auf denen Eclipse unterstützt wird. Die Definition der von der Anwendung benötigten Plug-Ins und die Auflistung der von der Anwendung benutzten Einstiegspunkte (die Modulbeschreibung) ist dabei in der plugin.xml des Projekts definiert. Das Oberflächendesign wird ebenfalls durch Eclipse verwaltet. Für die

1 Downloadseite: http://archive.eclipse.org/eclipse/downloads/drops/R-3.6.1-201009090800/index.php

Erstellung der GUI Komponenten verwendet Eclipse die eigene SWT Bibliothek. Da SWT die Darstellungskomponenten des jeweiligen Betriebssystems nutzt, erhält die erzeugte Anwendung ohne manuelle Anpassungen ein für die jeweilige Plattform typisches Aussehen.

3.2.3. Eclipse Oberfläche

Die Eclipse Oberfläche [Abbildung 8] bietet verschiedene Möglichkeiten zur Darstellung von Inhalten. Die wichtigsten Komponenten sind dabei der Editor, die Views und die Perspektiven.

Editor

Der Editorbereich ist der zentrale Bildschirmbereich für die Bearbeitung einer ausgewählten Datei. Die Darstellung der Daten kann sowohl in Textform, beispielsweise bei einem Quelltext-editor, wie auch in graphischer Form, zum Beispiel bei einem UML-Editor, erfolgen. Es können verschiedene Dateien in unterschiedlichen Editoren gleichzeitig geöffnet werden, jedoch darf es nur einen einzigen Editorbereich je Programmfenster geben in dem die einzelnen Editoren dann anhand von Tabreitern angeordnet werden.

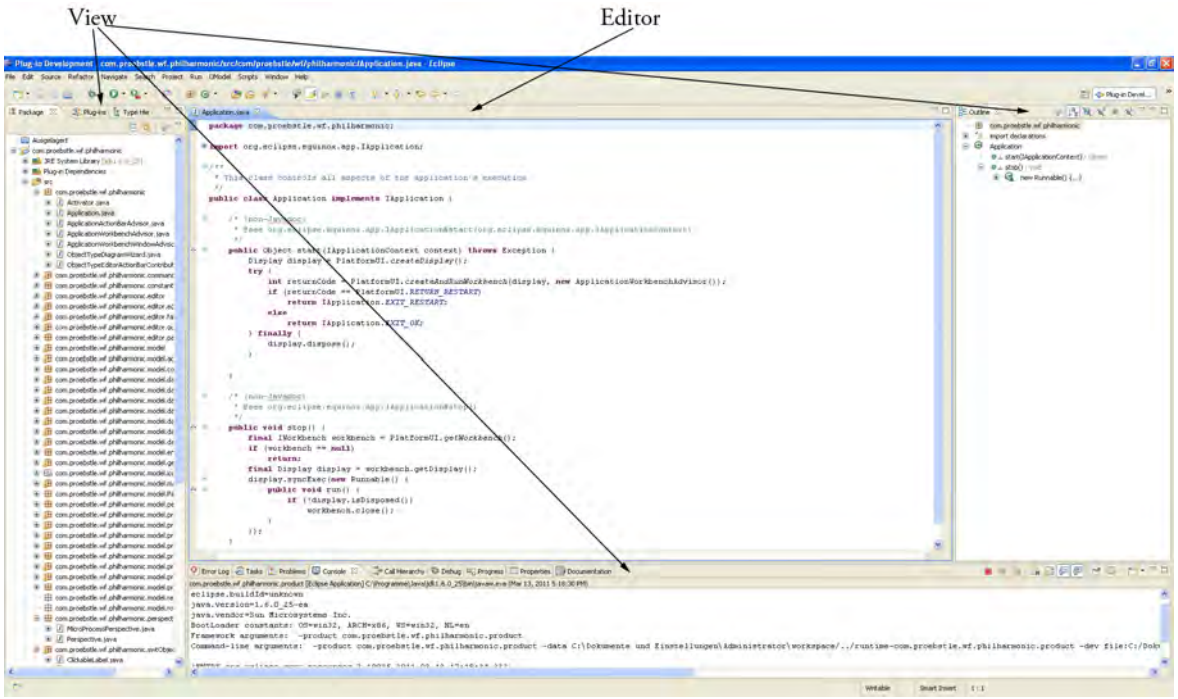

Abbildung 8.: Die Eclipse Oberfläche

View

Views sind frei bewegliche Fenster die durch Drag & Drop frei innerhalb des Programmfensters bewegt, angeordnet, gestapelt und geschlossen werden können. Sie können auch außerhalb des Hauptfensters als eigenständige Fenster angeordnet werden. Die Views ermöglichen sowohl die Anzeige von allgemeinen Informationen als auch die Detailanzeige von aktuell im Editor ausgewählten Objekten. Darüber hinaus können sie allgemeinen Zwecken wie der Navigation innerhalb einer Ordnerstruktur dienen.

Perspektive

Perspektiven sind festgelegte Anordnungen des Editors und ausgewählter Views auf der Programmoberfläche. Dabei ist in der Perspektive festgelegt welche Views angezeigt werden und wie diese auf dem Bildschirm angeordnet werden. Beim Hinzufügen einer neuen View wird dabei der Bereich einer bestehenden View oder des Editors in einem anzugebenden Verhältnis gesplittet, sodass die Fläche in zwei kleinere rechtwinklige Bereiche aufgeteilt ist. Durch diese Vorgehensweise wird verhindert, dass es leere Flächen in einer Perspektive gibt.

3.2.4. Graphical Editig Framework

Das Graphical Editing Framework (GEF)[13] ist ein Framework, welches die Erzeugung graphischer Editoren innerhalb der Eclipse Plattform ermöglicht. Es ist nach dem Model-View-Controller Entwicklungsmuster [siehe 3.3.3] aufgebaut und macht auch von den Entwurfsmustern der Softwareentwicklung regen Gebrauch. Das GEF besteht aus drei einzelnen Plug-Ins, wovon Draw2d und das GEF-Plugin für die weitere Betrachtungen im Rahmen dieser Arbeit wichtig sind.

Draw2d

Draw2d [15] ist ein auf Diagramme spezialisiertes Vektorgraphik-Toolkit für Eclipse. Es ist Teil des GEF Projekts, kann aber auch eigenständig verwendet werden. Draw2d ist ein nur auf SWT basierendes leichtgewichtiges Plug-In für die Anzeige graphischer Komponenten. Dabei stellt ein SWT Canvas die Zeichenfläche bereit auf der die Draw2d-Objekte, sogenannte Figuren (IFigure) platziert werden können. Figuren werden anhand einer Eltern-Kind Beziehung verwaltet, das

heißt die Elternfigur besitzt Grenzen, in denen sie und ihre Kinder gezeichnet werden können. Für die Anordnung der Kinder bietet Draw2d verschiedene Layout-Manager zur Auswahl. Für die graphischen Elemente stehen verschiedene Komponenten wie beispielsweise Rechtecke, Ellipsen, Label und Buttons zur Verfügung. Diese können durch Connections miteinander verbunden werden, deren Wegfindung anhand verschiedener Vorgaben und Arten gestaltet werden kann. Außerdem können die Connections mit Dekorationen wie Pfeilen oder Labeln versehen werden. Für einen besseren Überblick ermöglicht es die Erstellung einer Übersicht des Editors durch ein Thumbnail. Dieses kann in der Outline-View von Eclipse als navigierbares Übersichtsfenster verwendet werden.

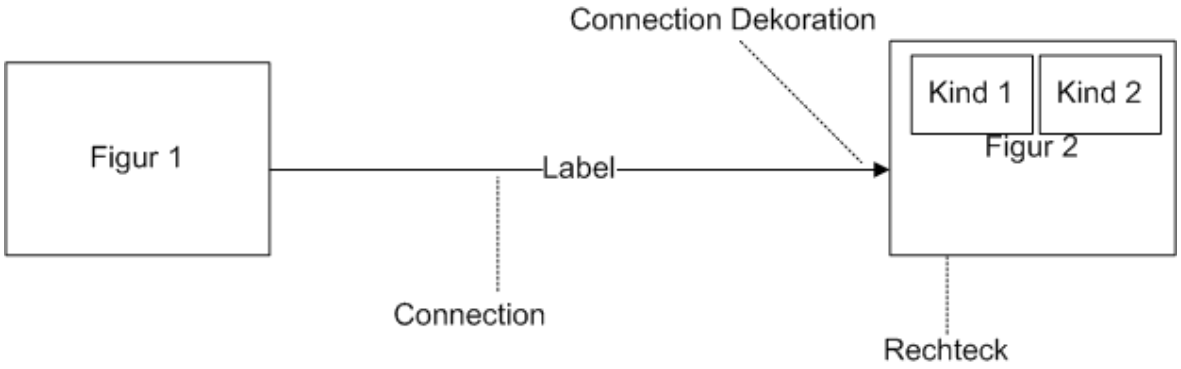

Abbildung 9.: Darstellung Draw2d

GEF -Plugin

Das GEF-Plugin bildet in GEF die Controller Schicht einer MVC-Architektur. Zentrales Interface dabei ist das EditPart. Für jedes Modellobjekt, dass im GEF-Editor editiert werden soll, muss ein EditPart existieren. Bei der Erzeugung des EditParts wird ein Listener für das zugrundeliegende Modell eingerichtet, der das EditPart über Änderungen der zugrundeliegenden Daten informiert. Anschließend wird aus dem EditPart die Draw2d Figur für die Darstellung im Editor erzeugt. Zur Definition der möglichen Operationen auf dem EditPart werden EditPolicies für bestimmte Editieroperationen angelegt. Diese definieren beispielsweise ob die Objekte verschoben oder skaliert werden dürfen oder welche Kindobjekte darauf abgelegt werden können. Ein EditPart kann über mehrere verschiedene EditPolicies verfügen, jedoch ist zu beachten, dass für jede Editieroperation nur eine EditPolicy existieren darf. Die Interaktion zwischen dem Benutzer und den EditParts erfolgt durch sogenannte Requests, welche die gewünschten

Änderungen kapseln. Ein Request wird dabei durch eine Benutzerinteraktion im Editor erzeugt. Das EditPart leitet das Request dann an die für diesen Typ des Request zuständige EditPolicy weiter. Anhand der EditPolicy wird daraus ein Command Objekt[1] für die gewünschte Aktion erstellt, das die Informationen zur Ausführung des Vorgangs enthält. Durch die Ausführung der execute-Methode der Kommando-Klasse erfolgt dann die gewünschte Operation.

3.3. Konzepte

An dieser Stelle sollen verschiedene Begriffe und Konzepte, die für das Verständnis der Arbeit wichtig sind, erläutert werden.

3.3.1. OSGi Service Platform

Die Spezifikation der OSGi Service Platform wird von der OSGi Alliance herausgegeben und beschreibt eine dynamische, modulare Softwareplattform für die Java Programmiersprache. Wir wollen uns hier nur auf die für das Verständnis der Eclipse Architektur nötigen Grundlagen konzentrieren. Ein OSGi-Framework besteht aus dem Kern und einzelnen Modulen. Die Module bestehen dabei in der Regel aus mehreren in verschiedene Pakete aufgeteilte Java-Klassen und einer Modulbeschreibung, in der auch Abhängigkeiten von anderen Modulen vermerkt sind. Anhand dieser ist es möglich einzelne Module zur Laufzeit hinzuzufügen, auszutauschen oder zu löschen. Die OSGi Spezifikation erlaubt somit ein aus einzelnen Modulen bestehendes dynamisches Gesamtsystem zu entwerfen.

3.3.2. Entwurfsmuster

Entwurfsmuster beschreiben bewährte Vorgehensweisen für häufig auftretende Probleme in der Softwareentwicklung. Sie beschreiben dabei Lösungsansätze und Wege und bestehen nicht aus fertigem Code wie Bibliotheken. Sie sind daher programmiersprachenunabhängig verwendbar. Im Gegensatz zu Architekturmustern, die das globalere Zusammenspiel der Komponenten einer Anwendung beschreiben, beziehen sich die Entwurfsmuster meist auf kleinere, umrissene Teilprobleme. Die Verwendung von Entwurfsmustern ist bei der Entwicklung hilfreich, da auf bereits bestehendes Wissen und Erfahrungen von anderen zurückgegriffen werden kann und

1 siehe „Kommando“ auf Seite 29

nicht jedes Problem mit einem eigenen neuen Ansatz erschlossen werden muss. Auch erleichtert es die Beschreibung und Dokumentation der Vorgehensweise indem man auf Bezeichnungen eines einheitlichen Vokabulars zurückgreifen kann. Die Entwurfsmuster sollten jedoch nicht als Wundermittel angesehen werden, die die Entwicklung guter Software garantieren. Oft verursacht der übermäßige Gebrauch von Entwurfsmustern zusätzliche Komplexität der Software und verhindert einfachere Lösungen. Daher sollte bei der Verwendung der Entwurfsmuster beachtet werden, dass sie ein bestehendes Problem lösen und sie nicht nur der Verwendung wegen Umgesetzt werden.
Populär wurden die Entwurfsmuster in der Softwareentwicklung mit der Veröffentlichung des Buchs „Design Patterns“ der sogenannten Viererbande[1] im Jahr 1994.

Kurzbeschreibung verschiedener Entwurfsmuster

Um ein besseres Verständnis und einen besseren Überblick zu gewinnen, sollen hier im Folgenden verwendete Entwurfsmuster vorgestellt werden.

Beobachter

Das Beobachter-Entwurfsmuster kann verwendet werden, wenn abhängige Klassen (Beobachter) über Änderungen einer Klasse (Subjekt) informiert werden sollen. Der Aufbau des Beob-

1 (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides)

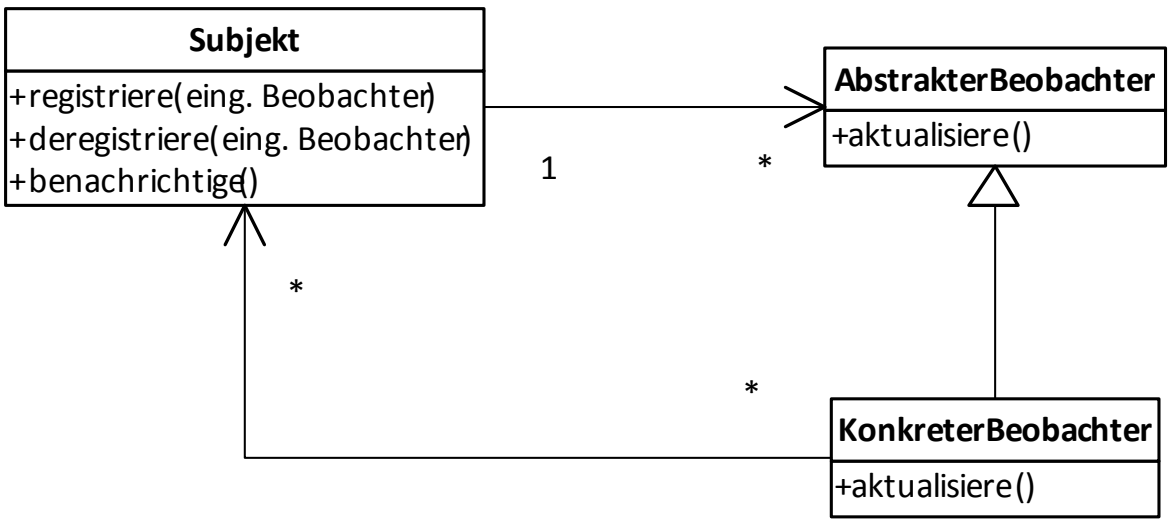

Abbildung 10.: Klassendiagramm des Beobachter-Entwurfsmusters

achter Entwurfmusters ist in Abbildung 10 dargestellt. Die Beobachter registrieren sich dabei im Subjekt. Das Subjekt schickt bei einer Änderung seines Zustands einen Broadcast an die Aktualisierungsmethode aller registrierten Beobachter, die auf diese Nachricht dann entsprechend reagieren können.

Kommando

Das Kommando-Entwurfsmuster unterstützt die Ausführung von Befehlen. Dazu werden alle für den gewünschten Vorgang benötigten Daten in einem Kommando-Objekt gespeichert, die in einer zentralen Kommando Liste verwaltet werden. Die tatsächliche Ausführung des Kommandos kann dadurch unabhängig von der Erstellung des Kommandos stattfinden. Durch die

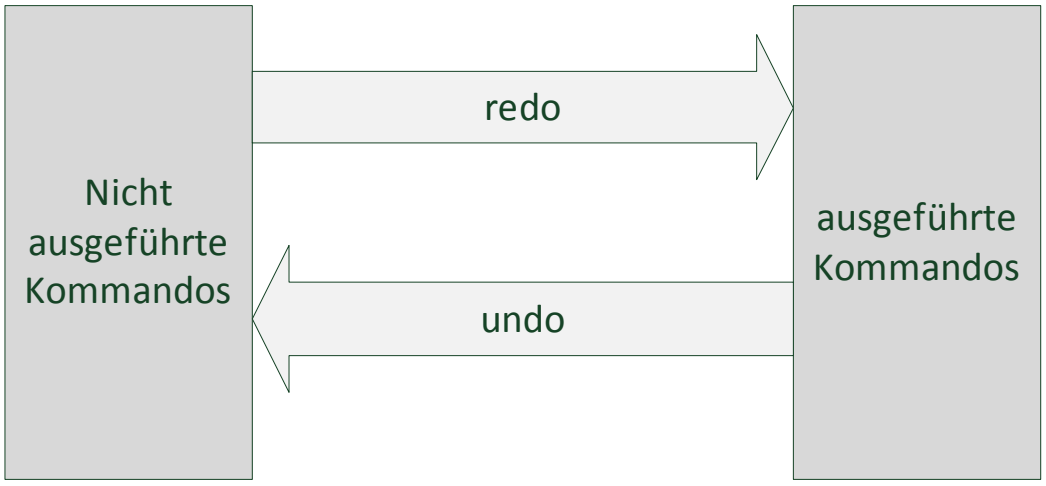

Abbildung 11.: Verwaltung der Kommandos

zusätzliche Speicherung der zur Umkehrung notwendigen Daten in den Kommando-Objektetn ist ein Rückgängig-Mechanismus möglich. Dazu müssen nur die ausgeführten Kommandos zusätzlich in einer zweiten Kommandoliste gespeichert werden. Durch Aufruf der Rückgängig-Methode werden die Befehle wieder zurückgesetzt und das Kommando in die ursprüngliche Liste zurückverschoben (siehe Abbildung 11).

3.3.3. Model-View-Controller-Architektur

Model-View-Controller (MVC) ist ein Muster der Softwaretechnik, welches die Datenlogik vom Benutzerinterface trennt und somit eine unabhängige Entwicklung voneinander erlaubt. Es wird heutzutage den Architekturmustern und nicht den Entwurfsmustern zugeordnet. Das

Muster sieht dabei die Aufteilung in die drei Einheiten Datenmodell (model), Präsentationsschicht (view) und Steuerung (controller) vor. Abbildung 12 zeigt eine Übersicht über das Zusammenspiel der Teilbereiche des Konzepts.
Das Modell verwaltet dabei die darzustellenden Daten und bietet die Methoden zur Änderung und zum Abruf der Daten an. Bei Änderungen der anzuzeigenden Daten des Modells werden die Controller benachrichtigt. Das Modell ist der einzige Speicherort für die persistenten Daten.
Die View ist als Präsentationsschicht für die Darstellung der Modelldaten zuständig. Im GEF besteht sie aus Draw2d Figuren. Ebenfalls Aufgabe der View ist es die Benutzerinteraktionen an den Controller weiterzugeben.
Der Controller dient als Bindeglied zwischen der View und dem Modell. Dabei nimmt er die von der View gemeldeten Benutzerinteraktionen entgegen und führt am Modell die zugeordneten Methoden aus. Auch reagiert er auf Änderungsmeldungen der Modelldaten und aktualisiert die View entsprechend. Der Controller ist somit der zentrale Punkt der Architektur, der sowohl Modell als auch View kennt. In GEF wird er durch EditParts im GEF-Plugin realisiert.

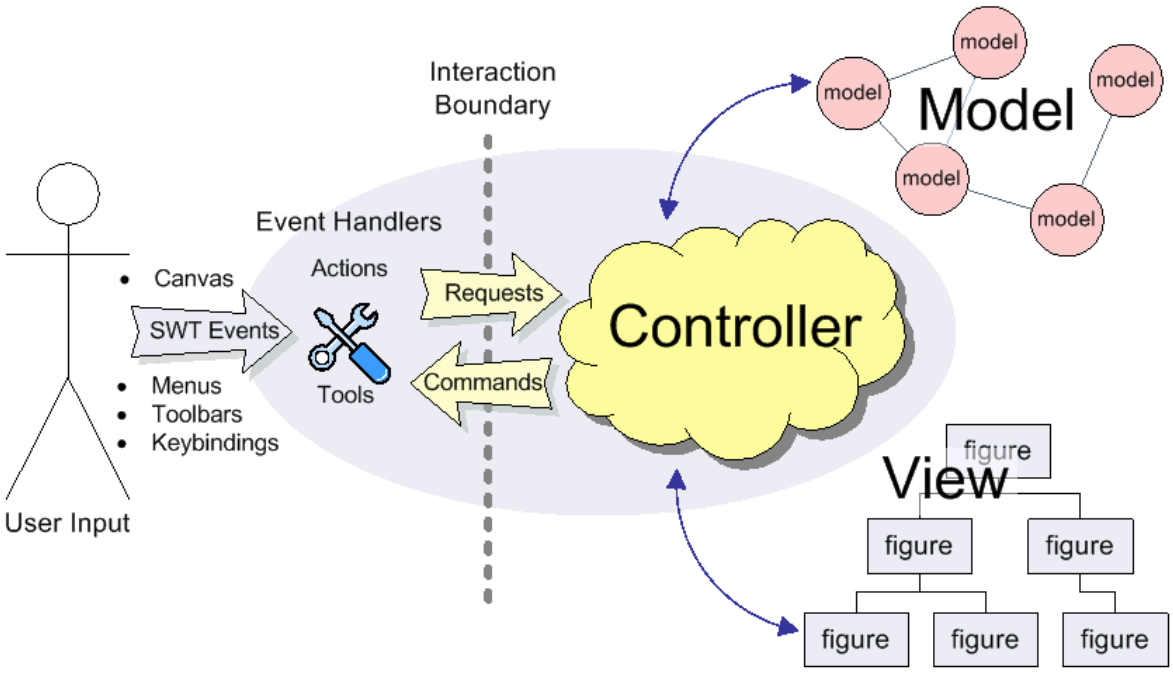

Abbildung 12.: Das MVC-Konzept in Eclipse[2]

4

Überblick Oberfläche

Dieses Kapitel bietet einen kurzen Überblick über die Gestaltung der Benutzeroberfläche des implementierten Frameworks, um die Einordnung der Implementierungsdetails zu erleichtern und die Zusammenhänge zu veranschaulichen. Als Ausgangsbasis für die Umsetzung der Benutzeroberfläche von PHILharmonicFlows dient das Usability-Konzept von Nicole Wagner[17].

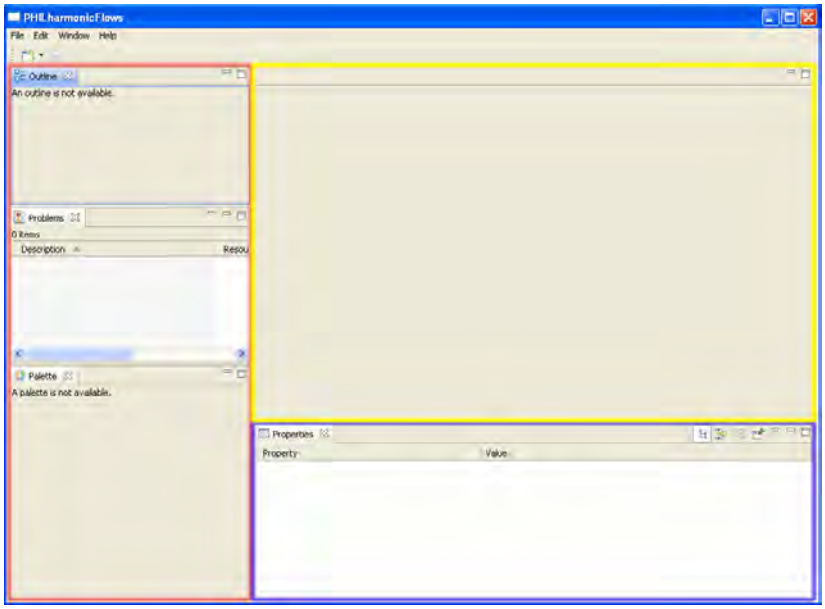

Abbildung 13.: Benutzeroberfläche bei Programmstart

Auf die Hintergründe für die Unterschiede und Abweichungen wird später in Kapitel 6.2 genauer eingegangen.

4.1. Anfangsbildschirm

Zum Programmstart bietet sich dem Benutzer die zu diesem Zeitpunkt noch weitgehend deaktivierte Oberfläche wie in Abbildung 13. Die Oberfläche unterteilt sich in die drei Bereiche Sidebar (rot umrandet), Editorbereich (gelb) und Eigenschaftsfenster (blau). Die Sidebar besteht wiederum aus drei einzelnen Teilfenstern, die ihre Funktionalität aber erst bei einem geöffneten Projekt erhalten. Die Teilbereiche sind die „Outline“, die eine verkleinerte Übersicht über den aktiven Editor bietet, das „Problems“ Fenster, in dem Fehlernachrichten gesammelt werden und der „Palette“, die Werkzeuge für die geöffneten Editoren anbietet und die Auswahl des gewünschten Editors ermöglicht. Geschlossene Fenster lassen sich über das „Window“ Menu wieder herstellen.
Um ein Projekt zu öffnen kann entweder ein neues mit dem Assistenten im „File“ Menu begonnen werden, oder ein bereits bestehendes über den „Open“ Dialog im „File“ Menu ausgewählt werden. Mit dem Öffnen eines Projekts startet der Editor für die Datenstruktur.

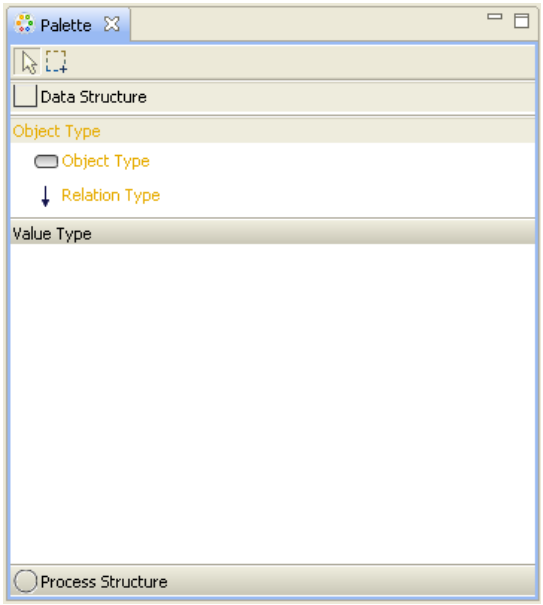

Abbildung 14.: Ansicht der Palette

4.2. Palette

Die Navigation zwischen den einzelnen Editoren ermöglicht die Palette in der Sidebar, die in Abbildung 14 einzeln zu sehen ist. Um die Navigation zu vereinfachen sind inhaltlich zusammengehörige Editoren innerhalb eines Hauptreiters zusammengefasst. Um die Unterscheidung zu vereinfachen sind die Hauptreiter durch ein Symbol vor der Ebenenbezeichnung hervorgehoben. Innerhalb einer aktivierten Hauptkomponente sind die zugehörigen Reiter der Unterkomponente, die jeweils den zugehörigen Editor öffnen, sichtbar. Dabei ist der jeweils aktive Palettenreiter aus Gründen der Übersichtlichkeit farblich hervorgehoben. Unterhalb des aktivierten Palettenreiters werden die Modellierungselemente für den gewählten Editor angeboten.

4.3. Oberfläche der Datenstruktur

Unterhalb der Datenstruktur Hauptreiter der Palette besteht die Wahl zwischen dem Objekttypen-Editor und dem Wertetypen-Editor. Dabei ist der Objekttypen-Editor der zentrale Ausgangspunkt jedes Projekts, da hier mindestens ein Objekttyp angelegt werden muss, um weiterzuarbeiten.

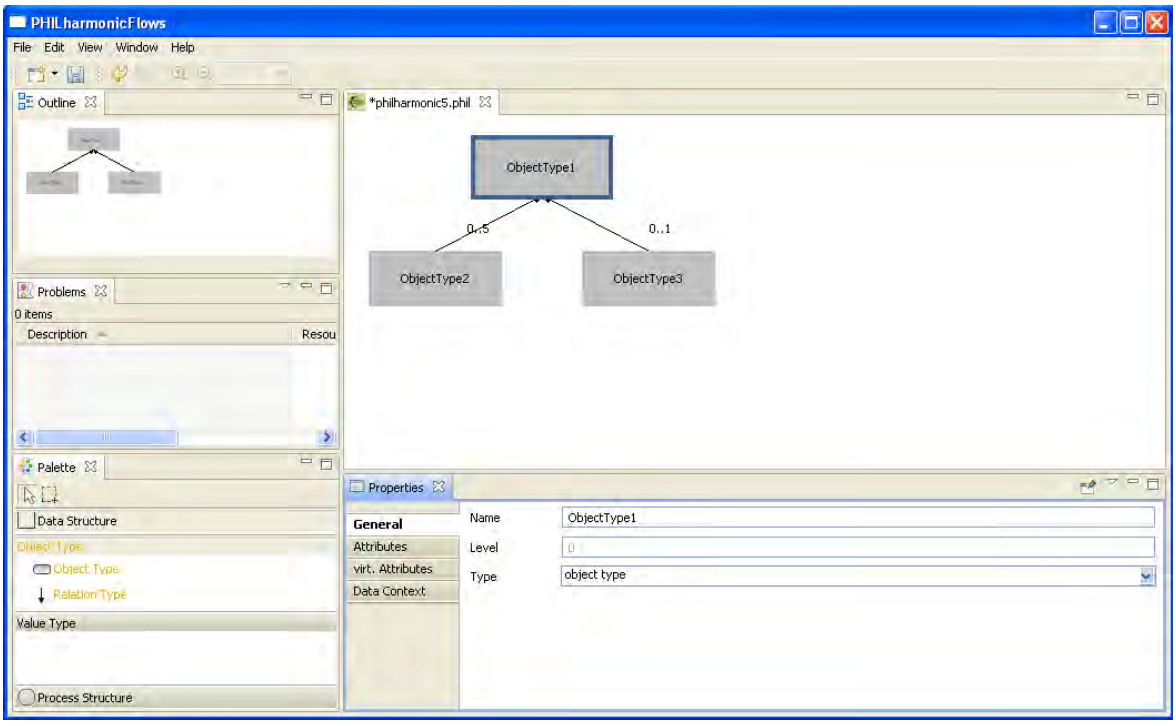

Abbildung 15.: Objekttypen-Editor

4.3.1. Objekttypen-Editor

Abbildung 15 zeigt das Framework mit geöffnetem Objekttypen-Editor. Im Editorfenster kann nun mit den in der Palette angebotenen Elementen „Object Type“ für Objekttypen und „Relation Type“ für Relationen die Datenstruktur modelliert werden. Dazu werden diese ausgewählt und im Editorfenster platziert. Die Objekttypen werden dabei im Editor als Rechtecke und die Relationen als Pfeile zwischen den Rechtecken dargestellt. Zur Unterscheidung der Objekttypen werden deren Namen mittig innerhalb der angezeigten Recktecke angezeigt. Zusätzlich wird in der oberen linken Ecke angezeigt ob und wie viele Datenkontexte zum Objekttyp existieren und rechts oben eine Figur, falls der Objekttyp als Benutzertyp typisiert ist. Alle Eigenschaften des ausgewählten Objekts werden zusätzlich detailliert im Eigenschaftsfenster unterhalb des Editors angezeigt. Dieses ist für eine einfachere Übersicht in mehrere Tabs unterteilt. Der gerade ausgewählte Objekttyp ist im Editor wie in Abbildung 15 („ObjectType1“ ist hier ausgewählt) zu erkennen mit einem blauen Rahmen markiert: Falls eine Relation ausgewählt ist, wird diese durch zwei schwarze Punkte an den Enden hervorgehoben. In der „Outline“ in der Sidebar wird eine navigierbare Übersicht über den Inhalt des Editorfensters gezeigt. Falls im Editor nicht der gesamte Bereich angezeigt werden kann, ist dabei der gerade dargestellte Bereich farbig hinterlegt.

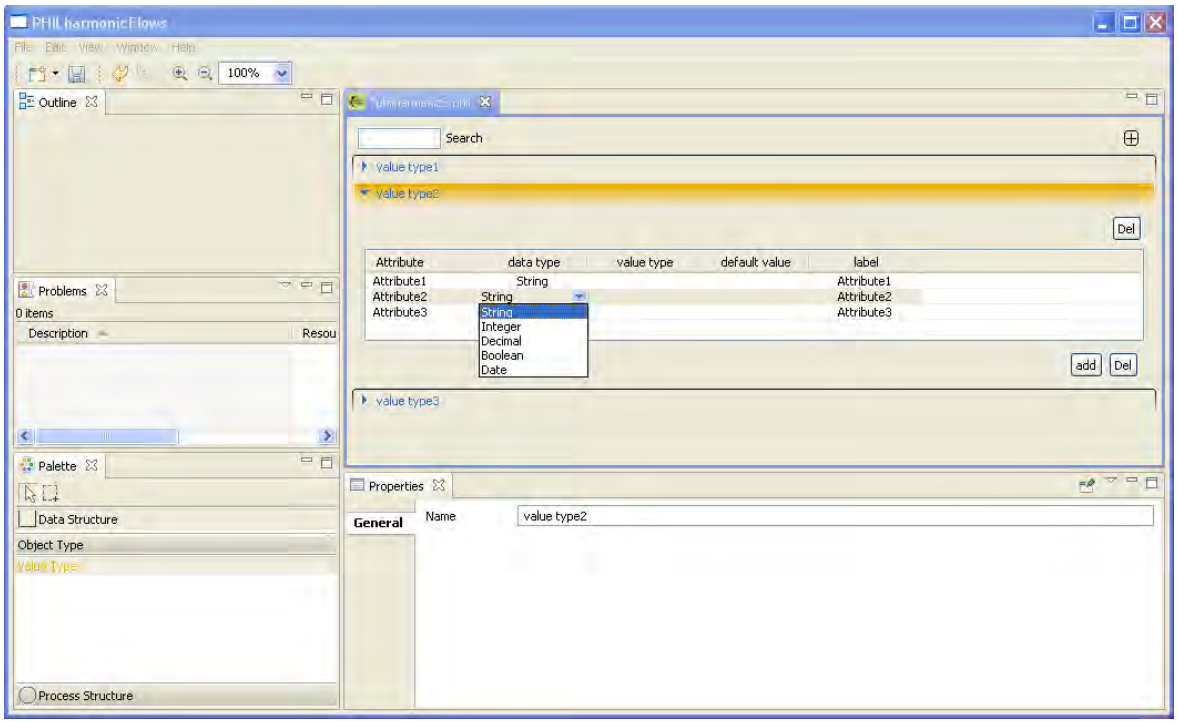

Abbildung 16.: Ansicht des Wertetypen-Editors

4.3.2. Wertetypen-Editor

Im Wertetypen-Editor erfolgt die Definition der Wertetypen, mit denen Wertebereiche einzelner Attribute beschrieben werden können. Wie in der Abbildung 16 zu sehen ist, werden bei geöffnetem Wertetypen-Editor in der Palette keine zugehörigen Modellierungselemente angeboten. Dies erübrigt sich, da die Editoroperationen direkt über Schaltflächen im Editorbereich erfolgen. Zu Beginn ist der Editorbereich bis auf eine Titelzeile leer. Diese bietet links ein Suchfenster, welches die Navigationsfunktion der ausgegrauten „Outline“ ersetzt. Auf der rechten Seite der Leiste befindet sich eine Schaltfläche zum Erzeugen neuer Wertetypen. Für jeden erzeugten Wertetyp wird eine sogenannte Sektion erstellt, die sich dem Benutzer zuerst als Textzeile mit dem Namen des Wertetyps zeigt. Falls der Name angeklickt wird öffnet sich der Wertetyp wie in Abbildung 16 bei „value type2“ zu sehen. Innerhalb der nun ausgeklappten Sektion bietet sich dem Benutzer eine Tabelle mit Schaltflächen zur Bearbeitung des Wertetyps. Als Orientierung für den Benutzer ist die Titelzeile des ausgewählten Wertetyps farbig hinterlegt.

4.4. Prozessstruktur

Innerhalb der Prozessstruktur Kategorie befindet sich der Mikroprozess-Editor. Dabei teilt sich der Editorbereich, wie in Abbildung 17 sichtbar, in zwei Teilfenster auf. Der obere Bereich, der sogenante Strukturkompass zeigt eine Übersicht der Datenstruktur. In diesem wird der gewünschte Objekttyp ausgewählt, der im darunter befindlichen Editor bearbeitet werden kann. Bei der Platzierung der Modellierungselemente muss beachtet werden, dass die Zustände (State Type) nur direkt auf der Editorfläche und die Mikroschritte (Micro Step) nur innerhalb von Zuständen platziert werden können. Dies vereinfacht im Gegensatz zu einem freien Platzieren mit einer nachträglichen Zuordnung mögliche Fehler bereits in der Konstruktion, die der Benutzer dann nicht im Nachhinein beheben muss. Als Hilfestellung bei der Konstruktion der Mikrotransitionen dienen die Halbkreise an den Seiten der Mikroschritte. Der Start-Mikroschritt besitzt als einziger Schritt keinen Halbkreis auf der linken Seite um deutlich zu signalisieren, dass keine eingehende Transition möglich ist. Der Halbkreis auf der rechten Seite wird bei allen Mikroschritten ausgeblendet, die alle Bedingungen für einen möglichen End-Mikroschritt erfüllen. Die Zuordnung von Attributen zu den Mikroschritten erfolgt durch einen Mausklick auf das hervorgehobene Feld innerhalb des Mikroschritts. Daraufhin öffnet sich das in Abbildung 17 rechts unten zu sehende Auswahlfenster, aus dem die gewünschte Referenz ausgewählt

werden kann. Mikrowertschritte für einen ausgewählten Mikroschritt können über das Eigenschaftsfenster angelegt und gelöscht werden.

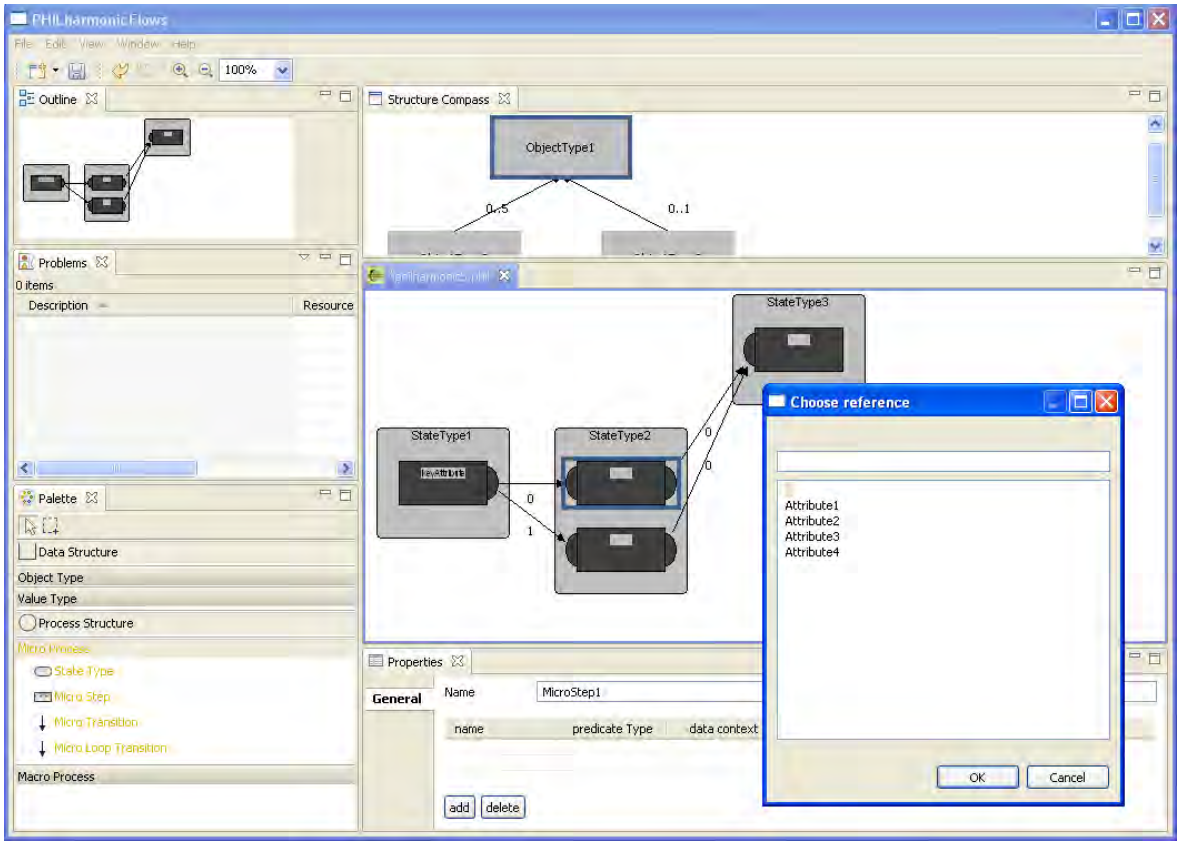

Abbildung 17.: Der Mikroprozess-Editor

5 Implementierung

Die Implementierung des Tools erfolgt innerhalb des Eclipse RCP [siehe 3.2.2]. Dabei wurden die Editoren der Arbeitsfläche teilweise mithilfe des GEF [siehe 3.2.4] implementiert und basieren auf der Model-View-Controller Architektur [siehe Seite 29]. Die Klassen sind somit in die Daten-Klassen, die den Modell Teil des MVC-Konzepts beschreiben und die Oberflächenklassen aufgeteilt. Die Oberflächenklassen beinhalten dabei sowohl die View als auch die Controller Komponente der MVC Architektur. Zur Veranschaulichung der Zusammenhänge werden unter anderem UML-Diagramme eingesetzt, die mit dem Programm UModel von Altova[1] erstellt sind.

5.1. Implementierung Paketstruktur

Bevor auf die Besonderheiten der einzelnen Klassen eingegangen wird, erfolgt zuerst ein Überblick über deren Aufgliederung. Bei der Implementierung entstanden eine Vielzahl verschiedener Klassen die für verschiedene Zwecke notwendig sind. Für eine bessere Übersicht über die Quellcodeklassen wurden diese anhand ihrer Verwendung systematisch in Pakete aufgeteilt.

1 UML Modellierungstool von Altova, welches alle 14 Diagrammtypen der UML 2.3 Spezifikation unterstützt.

Abbildung 18 bietet zur Veranschaulichung eine Übersicht über die für das Framework erstellten Pakete. Um diese übersichtlicher zu strukturieren wurde versucht diese anhand einer einheitli-

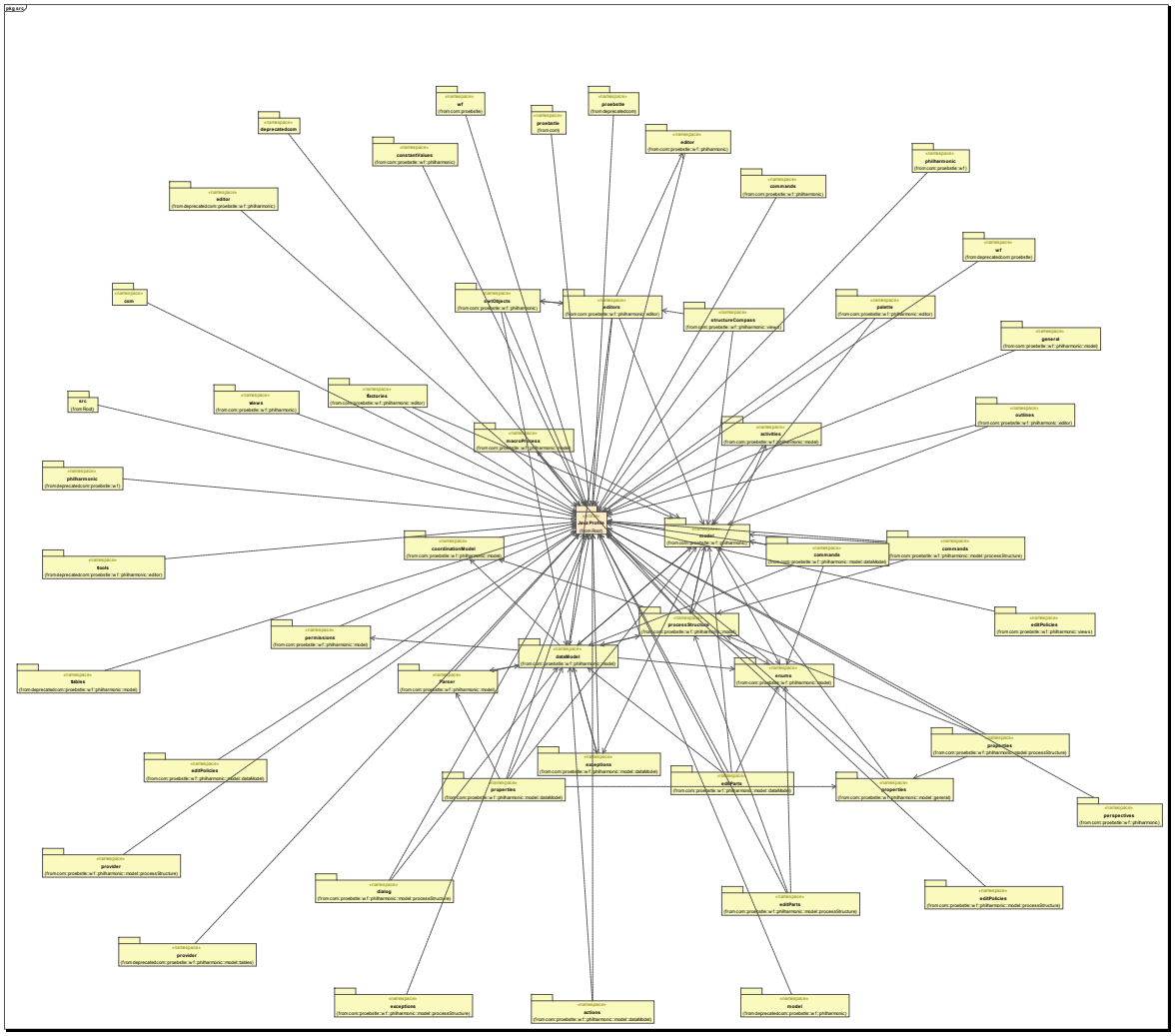

Abbildung 18.: Paketdiagramm

chen Systematik zu untergliedern. Diese Untergliederung erfolgt durch mehrere Hauptpakete, die zusammengehörige Unterpakete bündeln. Die Einteilung wurden dabei anhand der Funktion der Pakete und der Einordnung zu den Hauptkomponenten des Konzepts vorgenommen. Im folgenden soll zur Erleichterung ein kurzer Überblick über die Systematik der Gliederung gegeben werden. Eine genaue aufgegliederte Beschreibung der einzelnen Pakete ist im Anhang „A“, „Überblick über die Paketstruktur“ zu finden.

5.1.1. Paket für Editoren

Im Hauptpaket `philharmonic.editor` befinden sich die Klassen, die für die Erzeugung und Verwaltung der Editoren und der jeweiligen Übersichtsfenster zuständig sind. Editoren sind dabei die Fenster auf denen die bearbeitbaren Inhalte angezeigt werden, nicht jedoch deren graphische Inhalte. Ebenfalls befinden sich die Definitionen der Palette unterhalb dieses Hauptpakets.

5.1.2. Paket für die Inhalte

Die Klassen für die Inhalte befinden sich innerhalb des Hauptpakets `philharmonic.model`. Dabei befinden sich direkt im Hauptpaket die für die Verwaltung der Datenstruktur notwendigen Klassen. Außerdem befinden sich die grundlegenden, abstrakten Modellklassen, die in den verschiedenen Unterpaketen erweitert werden, ebenfalls in diesem Paket. Die in verschiedenen

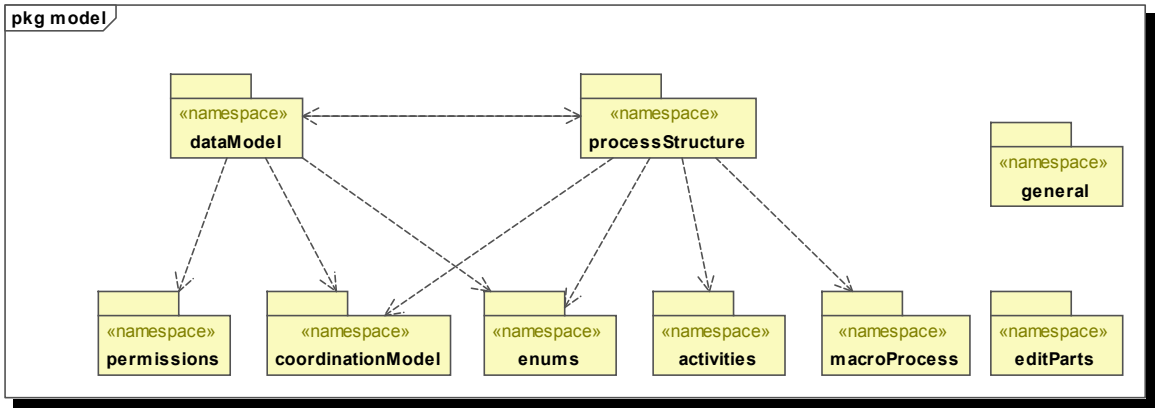

Abbildung 19.: Paketdiagramm des Modell-Hauptpakets

Klassen benutzten Aufzählungstypen sind im Unterpaket `enum` gesammelt. Das `editParts` Unterpaket definiert die abstrakten Klassen für alle in den Unterklassen erstellten EditParts. Im Unterpaket `general` sind schließlich von allen Klassen genutzte Bereiche und Funktionen für die porperties view definiert. Die anderen Pakete fassen jeweils die Klassen der jeweiligen Teilbereiche des PHILharmonicFlows-Konzepts zusammen. Daraus ergibt sich eine einfache verständliche Ordnung, die auf einfache Art erweitert werden kann, ohne in bestehende Pakete eingreifen zu müssen. Eine Übersicht über die Unterklassen des `philharmonic.model` Pakets bietet Abbildung 19.

Untergliederung der Inhaltspakete

Die einzelnen Pakete mit den Klassen für die Teilbereiche des PHILharmonicFlows-Konzepts sind ebenfalls nach einem systematischen Muster, wie in Abbildung 20 für das Daten Modell beispielhaft gezeigt, gegliedert. Dies bietet eine gute Übersicht über den Code, da diese Auftei-

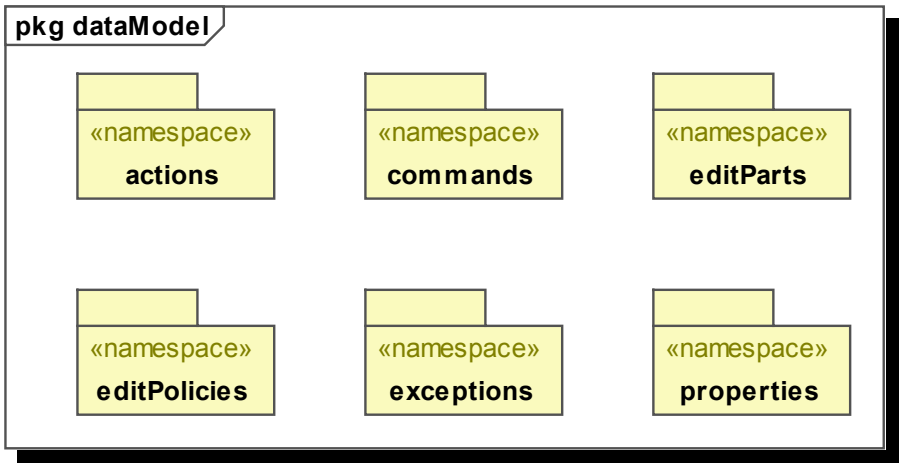

Abbildung 20.: Aufteilung Paket DataModel

lung in allen Paketen einheitlich durchgeführt ist. Die Datenklassen sind dabei direkt im Paket angeordnet. Das `actions` Paket sammelt die Klassen, die Aktionen für das Kontextmenu im Editor für den jeweiligen Teilbereich bieten. Im `commands` Unterpaket befinden sich die Operationen zum Ändern der Werte der Datenklassen. Die Klassen für die Generierung und Verwaltung der im Editor darzustellenden Objekte sind in den `editParts` Paketen aufzufinden. `EditPolicies` enthält die Klassen, die die erlaubten Aktionen innerhalb des Editors in dem jeweiligen Teilbereich beschreiben. Für mögliche Fehlerklassen ist das jeweilige `exceptions` Paket gedacht. Im Paket `properties` sind schließlich die Klassen, die die properties view für die einzelnen Datenklassen bilden.

5.2. Implementierung Daten-Klassen

In diesem Abschnitt werden der grundlegende Aufbau und die wichtigsten Funktionen und Methoden der Klassen, die das Modell des MVC-Konzepts bilden beschrieben. Eine Übersicht

über den Aufbau der Klassenhierarchie bietet Abbildung 21. Sämtliche Klassen des Modells erweitern dabei die `NamedObject` Klasse. Diese verwaltet einen unveränderbaren Schlüssel und einen änderbaren Namen. Damit kann jede Instanz der abgeleiteten Klassen eindeutig über den Schlüssel identifiziert werden. Der Schlüssel dient jedoch nur der internen Identifikation. Für die Identifikation gegenüber dem Benutzer ist der Name der Klasse gedacht. Dieser wird mit einem Standardwert initialisiert und kann nach den Bedürfnissen des Benutzers jederzeit geändert werden ohne dabei Bezüge zwischen den Klassen zu verletzen. Für die Speicherung der Daten

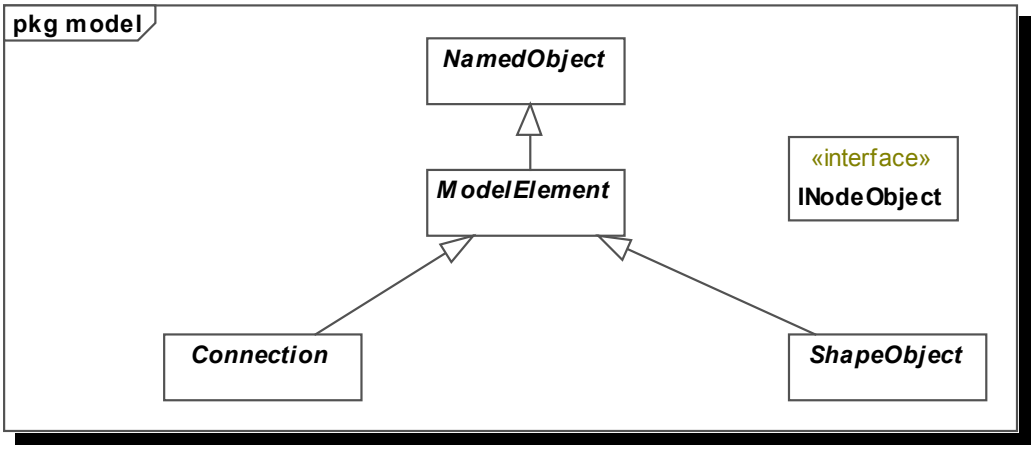

Abbildung 21.: Grundlegende Klassen

implementiert die Klasse das `Serializable` Interface. Dies ermöglicht eine Speicherung der Klasseninstanzen in einer seriellen Datei, aus der die Klassen mit den gespeicherten Werten beim erneuten Öffnen wieder hergestellt werden können. Außerdem implementiert die Klasse das Interface `IPropertySource`. Dieses unterstützt die Benachrichtigung von registrierten Klassen über Änderungen der Attributwerte der aktuellen Klasse nach dem Observer-Entwurfsmuster. An Änderungsmitteilungen interessierte Klassen müssen dabei die Schnittstelle `PropertyChangeListener` implementieren und sich als Listener registrieren. Die Benutzung dieses Entwurfsmusters ermöglicht die Erweiterung des grundlegenden Frameworks, ohne an den bestehenden Klassen Änderungen vornehmen zu müssen.
Die Klasse `ModelElement` bildet die Ausgangsbasis für alle Klassen, die in den graphischen Editoren dargestellt werden sollen und für die somit auch EditParts existieren. Um eindeutige Schlüssel für Kindelemente zu generieren enthält diese Klasse die Methode `getKeyForChild(Object childType)`. Diese speichert den höchsten bisher vergebenen Schlüssel für jeden Objekttyp in einem Hash und gibt bei ihrem Aufruf den nächst höheren zurück. Die Speicherung innerhalb eines Hashes ermöglicht einen schnellen Zugriff auf den

für den jeweiligen Objekttyp zuständigen Index. Als Index wird der Variablentyp BigInteger[1] gewählt, da somit auch bei großen oder oft überarbeiteten Modellen kein Überlauf wie bei der Verwendung von Integer auftreten kann.
Objekte in graphenbasierten Darstellungen können drei verschiedene Ausprägungen besitzen. Zunächst gibt es Objekte, die eigenständig platzierbar sind und zu keinen anderen Objekten in Verbindung stehen können. Um diese zu realisieren ist `ShapeObject` als Ausgangsbasis für solche im Editor eigenständig platzierbare Klassen gedacht, die unabhängig von Ankerpunkten platziert werden. Um den Ort, an dem diese dargestellt werden, zu verwalten besitzt sie Methoden zur Verwaltung von Ortskoordinaten.
Die Ausprägung für Objekte sind die sogenannten Knoten, die durch die letzte Ausprägung, den Kanten, miteinander verbunden werden um sie miteinander in Relation zu setzen. Als Grundlage für Knoten dient das Interface `INodeObject`. Dieses beschreibt die für Verwaltung von Kanten notwendigen Methoden. Dies ist als Interface umgesetzt, da die jeweilige Implementierung der Methoden sehr von der Verwendung der Kanten abhängig ist und sich daher keine sinnvolle Standardroutine implementieren lässt. Da Knoten auch immer selbstständig platzierbare Objekte sind, erweitern alle fertig implementierten Knotenklassen die Klasse `ShapeObject` und verwenden das Interface `INodeObject`.
Die abstrakte Klasse `Connection` bietet die grundlegenden Methoden für die Verbindung zweier Knotenobjekte. Die Knotenobjekte müssen dabei das Interface `INodeObject` implementieren um als Endpunkte der Verbindung in Frage zu kommen.

5.2.1. Datenstruktur

Die Datenstruktur stellt den Ausgangspunkt eines PHILharmonicFlows Workflows dar. Auf deren Basis sind die anderen Klassen erstellt und werden innerhalb dieser Instanzen verwaltet. Im folgenden werden nun die wichtigsten Klassen und ihre Zusammenhänge vorgestellt. Abbildung 22 bietet einen Überblick über Klassen der Datenstruktur. Aufgrund der Anzahl der verschiedenen Klassen wird im Folgenden nur auf ausgewählte Klassen mit den wichtigsten Strukturen eingegangen.

DataModell

Ursprung jedes PHILharmonicFlows Workflows ist die Klasse `DataModell`, die bei der Anlage eines neuen Projekts automatisch erstellt wird. Sie enthält direkte Verweise auf die wichtigsten

1 Variablentyp von Java, der die Darstellung beliebig großer Ganzzahlen ermöglicht

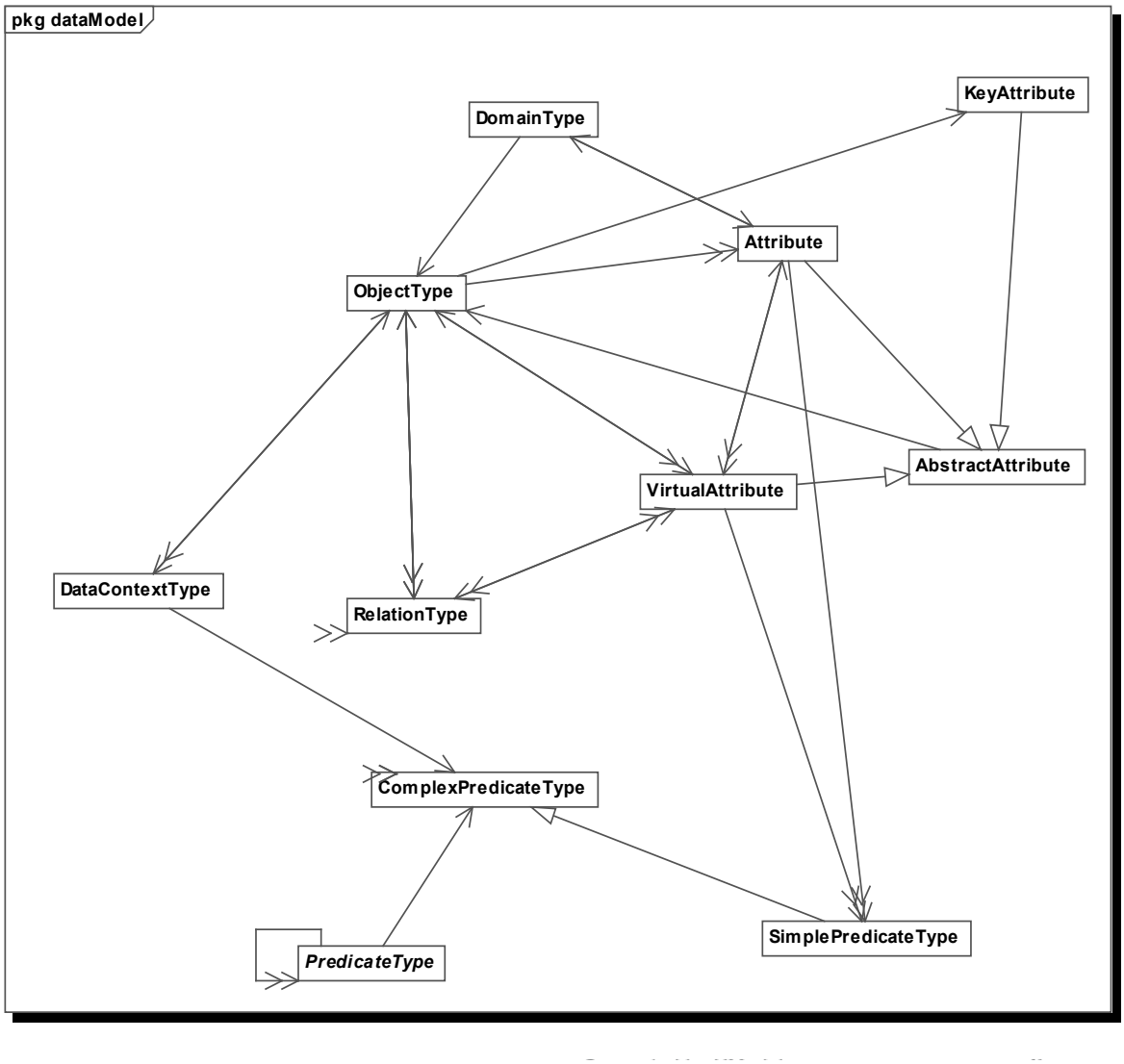

Abbildung 22.: Klassendiagramm Datenstruktur

Bestandteile des Workflows und verwaltet die Fehlermeldungen. Die Einordnung der Klasse ist anhand des Klassendiagramms in Abbildung 23 zu sehen. Die zugehörigen Objekttypen, die als Benutzer- oder Objekttyp definiert sind, werden in einer Liste verwaltet. Dies ermöglicht eine flexible Größe der Liste, bei der die Elemente in einer festen Ordnung verwaltet werden. Alternativ wäre noch die Speicherung anhand eines Hashes möglich, was einen schnelleren Zugriff auf die einzelnen Elemente ermöglichen würde. Jedoch könnte dann nicht mehr sichergestellt werden, dass sich die Reihenfolge der einzelnen Listenelemente nicht ändert und somit der Benutzer durch unterschiedliche Ausgaben verwirrt werden könnte. Die als Wertetyp definierten Objekttypen werden getrennt von den anderen Objekttypen in einer zweiten separaten Liste

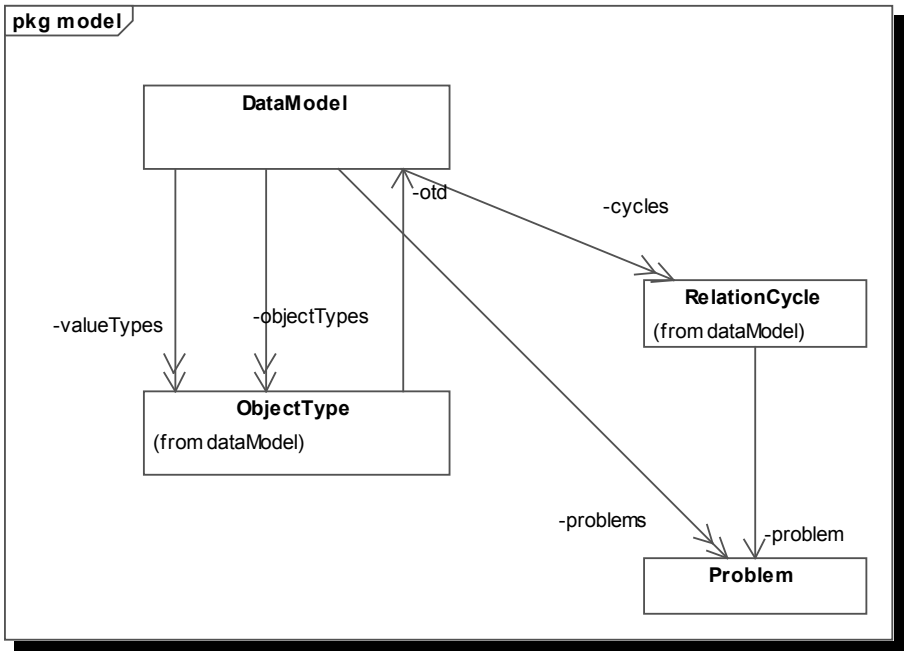

Abbildung 23.: Klassendiagramm DataModell

geführt. Dies vereinfacht die spätere Verwaltung, da bei Anfragen die jeweilige komplette Liste zurückgegeben werden kann. Bei Speicherung in einer gemeinsamen Liste müssten dagegen die einzelnen Objekttypen bei jeder Anfrage aufs neue einzeln auf ihren Typ untersucht werden und in einer neuen Teilliste zurückgegeben werden. Da Wertetypen laut Konzept auch nicht in Objekt- oder Benutzertypen überführt werden können, ist auch ein Wechsel zwischen den beiden Listen ausgeschlossen. In einer weiteren Liste verwaltet `DataModell` die noch nicht aufgelösten Zyklen. Die Verwaltung erfolgt innerhalb DataModell, da Zyklen meistens mehrere Relationen umfassen und sich über mehrere Objekttypen erstrecken können. Somit gibt es keine feinere Verwaltungseinheit für die Auflistung der noch offenen Zyklen. Zuletzt besitzt `DataModell` noch Referenzen auf alle im Workflow aufgetretenen Probleme, damit zentral ermittelt werden kann, ob er fehlerfrei ist, oder an welcher Stelle noch nachgebessert werden muss.

ObjectType

`ObjectType` beschreibt die Eigenschaften und Zusammenhänge der einzelnen Objekttypen. Objekttypen sind innerhalb von PHILharmonicFLows die zentralen Einheiten. Jeder `ObjectType` muss dabei direkt durch das `DataModell` referenziert sein.

```
1  //Definition einer rekursiven Hilfsfunktion zum Test auf Zyklen-
   //freiheit und Ermittlung des Levels
2  FUNCTION calculateLevel(HashMap visited, ObjectType root, int
   level)
3  INPUT: HashMap visited (bereits besuchte Objekttypen), Object-
   Type root (Ausgangspunkt der Suche), int level (Ausgangslevel)
4  OUTPUT: Integer (Berechnetes Level)
5  EXCEPTIONS: CycleExceptionContainer cycle(Zyklus mit Startele-
   ment erkannt), UnspecifiedCycleException unspecifiedCycle(Zyklus
   ohne Beteiligung des Startelements erkannt)
6  BEGIN
7      FOR alle Elternrelationen
8          IF Relation markiert als Zyklenrelation THEN
9              überspringen
10         ELSEIF Ziel der Relation = root THEN
11             //Falls Zyklus mit Beteiligung der Wurzel erkannt
               //wird, Hinzufügen der aktuellen Relation zur Zyklen
               //Ausnahme
12             cycle.add(aktuelle Relation)
13             überspringen
14         ELSEIF Ziel der Relation IN visited THEN
15             //Falls Ziel in diesem Pfad bereits besucht wurde,
               //existiert ein Zyklus ohne Beteiligung der aktuellen
               //Wurzel
16             unspecifiedCylce.add(aktuelle Relation)
17             überspringen
18         END IF
19         //Start der Rekursion mit aktueller Liste der besuchten
           //Objekttypen
20         visitedCopy = visited.copy()
21         visitedCopy.add(akueller Objekttyp)
22         int Vorgängerlevel = calculateLevel(visitedCopy, root,0)
23         IF Vorgänerlevel >= level THEN
24             //der Level muss nach Konzept mindestens eins größer
               //als der Vorgängerlevel sein
```

```
25            level = Vorgängerlevel + 1
26        EXCEPTION
27            WHEN CycleExceptionContainer container
28                //Falls in der Rekursion ein Zyklus erkannt
                  //wurde, wird die aktuelle Relation hinzugefügt
                  //um den Zyklenpfad zu ergänzen und die Ausnahme
                  //zum Sammelcontainer hinzugefügt.
29                container.add(aktuelle Relation)
30                cycle.add(container)
31    END FOR
32    IF cycle != null THEN
33        //Zyklus mit Wurzel erkannt, der versorgt werden muss,
          //Levelermittlung nicht möglich
34        RETURN cycle
35    ELSEIF unspecifiedCycle != null THEN
36        //Zyklus ohne Wurzelbeteiligung erkannt, daher Ermitt-
          //lung des Levels nicht möglich
37        RETURN unspecifiedCycle
38    END IF
39    RETURN level
40 END FUNCTION
41
42 //Rekursive Funktion correctChildsLevels überprüft ob die Level
   //der nachfolgenden Objekttypen geändert werden müssen
43 FUNCTION correctChildsLevels()
44 INPUT: -
45 OUTPUT: -
46 BEGIN
47    ObjectType ot
48    FOR alle Kindrelationen
49        IF Relation ist als Zyklusrelation markiert THEN
50            überspringen
51        END IF
52        ot = Ursprung(Relation)
53        IF Level(ot) <= Level(this)
```

```
54            //Falls Level des untergeordneten Objekttyps nicht
              //größer als das des Ausgangspunkts
55            ot.setLevel(this.level +1)
56        END IF
57        ot.correctChildsLevels()
58    END FOR
59 END FUNCTION
60
61 //Funktion addSourceConnection fügt die Relation als Elternre-
   //lation zum aktuellen Objekttyp hinzu.
62 FUNCTION addSourceConnection(Connection connection)
63 INPUT: Connection connection(Die neue ausgehende Verbindung)
64 OUTPUT: -
65 BEGIN
66    füge Relation zur Elternliste des Objekttypes hinzu
67    //Aufruf der rekursiven Funktion calculateLevel mit initial
      //leerer Map visited und dem eigenen Objekttyp
68    HashMap visited = new HashMap
69    int level = calculateLevel(visited, this, 0)
70    EXCEPTION
71        WHEN CycleExceptionContainer container
72            //Falls Zyklus mit Beteiligung des aktuellen Objekt-
              //typs erkannt, gebe dem DataModell den neuen Zyklus
              //bekannt
73            DataModell.addCycles(container)
74            RETURN
75        WHEN UnspecifiedCycleException container
76            //Falls Zyklus ohne Beteiligung des aktuellen Objekt-
              //typs erkannt, ist der Zyklus bereits bekannt
77            RETURN
78    //Falls kein Fehler aufgetreten ist, werden die Kinder auf
      //Korrektheit überprüft.
79    this.correctChildsLevels()
80 END FUNCTION
```

Beispielcode 1.: Erzeugen einer neuen Elternrelation

Die Ordnungsbeziehungen zwischen den verschiedenen Instanzen von `ObjectType` beschreiben dabei die `RelationType`. Diese werden getrennt nach ihrer Orientierung in zwei Listen gespeichert. Eine Liste enthält dabei die Elternrelationen und eine zweite die Kindrelationen. Elternrelationen bezeichnet dabei Relationen mit übergeordneten Objekttypen und Kindrelationen diejenigen mit untergeordneten Objekttypen. Objekttypen ohne Elternrelationen haben keine übergeordneten Objekte, gehören somit der geringsten Levelebene an und sind die Wurzeln der Beziehungsrelationen. Neue Elternrelationen des `ObjectType` werden dabei durch die `addSourceConnection` Methode, die in Beispielcode 1 als Pseudocode hinterlegt ist, hinzugefügt. Dabei wird ebenfalls versucht das Level für die Kindrelation zu ermitteln. Da dieses jedoch nur bei Zyklenfreiheit aller Vorgängerrelationen möglich ist, erfolgt parallel dazu eine Suche nach noch nicht aufgelösten Zyklen innerhalb der Vorgänger. Falls neue, noch nicht aufgelöste Zyklen erkannt wurden, werden diese analysiert und dem `DataModell` zur Speicherung übergeben. Beispiele für die möglicherweise auftretenden Zyklen sind in Abbildung 24 dargestellt. Bild 1 zeigt dabei den zyklenfreien Zustand zu Beginn. In Teilbild 2 entstehen durch die neue Relation A=>B zwei neue Zyklen, die als Zyklen mit Beteiligung der neuen Relation (im Code innerhalb eines CycleExceptionContainer zusammengefasst) erkannt werden. In Bild 3 wird eine mögliche Bearbeitung des Zustandes aus Bild 2 ohne Auflösung der vorhandenen

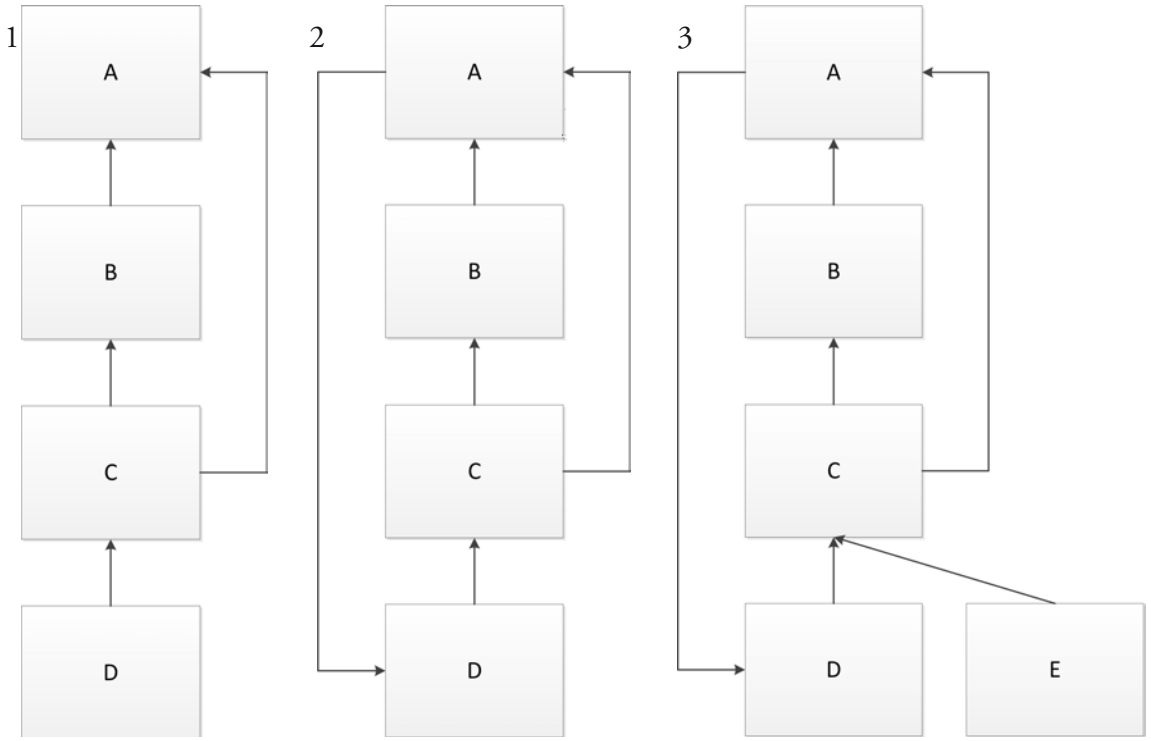

Abbildung 24.: Beispiel für Zyklenentwicklung

```
1  FUNCTION removeSourceConnection(Connection conn)
2  INPUT: Connection conn
3  OUTPUT: -
4  BEGIN
5      entferne Relation aus der Liste der Elternrelationen des
   Ob-    jekttyps
6      IF Level(Ziel(conn)) = Level(this) - 1 THEN
7          //Falls das Ziel der Relation genau einen Level kleiner
           //als das des Objekttyps ist, ist eine Leveländerung
           //möglich
8          int maxVorgLevel = max(Level(Elternrelationen))
9          IF maxVorgLevel + 1 < Level(this) THEN
10             //Falls niedrigeres Level errechnet, muss das Level
               //korrigiert werden.
11             Level(this) = maxVorgLevel + 1
12             FOR alle Kindrelationen
13                 //Einstieg ins überprüfen der Kinderlevel,
                   //Funktion hier nicht beschrieben.
14                 überprüfe Level
15             END FOR
16         END IF
17     END IF
18 END FUNCTION
```

Beispielcode 2.: Entfernen einer Elternrelation

Zyklen betrachtet. Mit dem Hinzufügen der neuen Relation E=>C entsteht kein neuer Zyklus. Jedoch ist eine Berechnung des Levels ebenfalls nicht möglich, da sich kein Objekt der Ebene 0 finden lässt, von der die Berechnung ausgehen kann. Im Code in Beispielcode 1 wird deshalb wie in Zeile 14 zu sehen eine UnspecifiedCycleException zurückgegeben um diesen Fall zu signalisieren.

Falls keine Zyklen erkannt wurden, erfolgt die Berechnung und Rückgabe des neuen Levels des `ObjectTypes`. Außerdem müssen bei Veränderungen die Level der Kindrelationen gegebenenfalls berichtigt werden. Hierbei kann jedoch nur eine Korrektur nach oben auftreten, da

durch das Hinzufügen einer zusätzlichen Relation das Level nur größer werden kann. Bei dem übergeordneten Objekttyp müssen beim Erstellen der Relation keine Seiteneffekte berücksichtigt werden.
Beim Löschen von Relationen, welches analog zum Pseudocode in Beispielcode 2 erfolgt, wird kontrolliert, ob das Level des zuvor untergeordneten Objekttyps korrigiert werden muss. Da bei Löschoperationen der Level nur geringer werden kann, reicht es aus die Level aller Vorgänger des `ObjectTypes` zu überprüfen und den maximalen Wert als Ausgangswert zu verwenden. Diese Überprüfung ist jedoch nur notwendig, falls der Level der an der Relation beteiligten Objekttypen sich genau um den Faktor 1 unterscheidet (siehe Beispielcode 2, Zeile 6). Ansonsten ist bereits bekannt, dass eine andere Relation die Grundlage für den Level bestimmt. Falls der Level korrigiert werden muss, werden alle Nachfolger des `ObjectTypes` ebenfalls rekursiv überprüft und dabei gegebenenfalls auch deren Werte korrigiert.
Als Datenstrukturen für die Speicherung der Elemente wurden generell Listen gewählt, da diese ihre Größe flexibel an die Anzahl der benötigten Elemente anpassen können. Diese wurden typisiert um falsche Zuordnungen bei der Implementierung zu verhindern.

RelationType

In der Klasse `RelationType` werden die Beziehungen der ObjectTypes untereinander gespeichert. Dabei wird der untergeordnete ObjectType als Quelle und der übergeordnete als Ziel der Relation bezeichnet. Außer den Eigenschaften der Relation, wie den Kardinalitäten, wird in dieser Klasse noch eine Liste der virtuellen Attribute geführt, die anhand dieser Relation definiert sind. Dies ist notwendig, damit beim Löschen der Relation die betroffenen virtuellen Attribute,

```
1  //Funktion zum Aktivieren der Verbindung
2  FUNCTION reconnect()
3  INPUT: -
4  OUTPUT: -
5  BEGIN
6      IF Relation nicht verbunden THEN
7          füge Relation zur Quelle als Elternrelation hinzu
8          füge Relation zum Ziel als Kindrelation hinzu
9      END IF
10 END FUNCTION
11
```

```
12 //Funktion zum Deaktivieren der Verbindung
13 FUNCTION disconnect()
14 INPUT: -
15 OUTPUT: -
16 BEGIN
17     IF Relation verbunden THEN
18         entferne Relation von der Quelle
19         entferne Relation vom Ziel
20         entferne Zyklen mit Beteiligung der Relation aus dem
           DataModell
21     END IF
22 END FUNCTION
23
24 //Funktion reconnect aktiviert die Verbindung mit den neu über-
   //gebenen Ziel und Quelle
25 FUNCTION reconnect(ObjectType newSource, ObjectType newTarget)
26 INPUT: ObjectType newSource (die neue Quelle), ObjectType new-
   Target (das neue Ziel)
27 OUTPUT: -
28 BEGIN
29     disconnect()
30     Quelle = newSource
31     Ziel = newTarget
32     reconnect()
33 END FUNCTION
```

Beispielcode 3.: Aktivierung einer Relation

die ohne die Relation keinen definierten Pfad besitzen, ebenfalls gelöscht werden können. Ansonsten müssten bei jedem Löschvorgang alle virtuellen Atribute erneut daraufhin überprüft werden, ob ihr Bezugsattribut noch erreichbar ist.
Wichtigste Methoden der Klasse sind die beiden `reconnect` Methoden und die `disconnect` Methode (siehe auch Beispielcode 3). Diese aktivieren die Relation, indem sie die Relation in den beteiligten Objekttypen registrieren und deaktivieren sie durch Löschung in den Objekttypen. Die Methode reconnect (Beispielcode 3, Zeile 2) aktiviert die Relation in den bereits

gesetzten Ziel- und Quellobjekttypen. Dies ist bei der Neuerstellung und nach einem Wiederherstellungsschritt erforderlich. Die `disconnect()` Methode löscht die Relation aus den beteiligten Objekttypen nach einer Löschoperation oder einem Rückgängigschritt. Die Methode `reconnect(ObjectType, ObjectType)` löst die bestehende Relation und aktiviert sie mit den neu angegebenen Ziel- und Quellobjekttypen. Dazu nutzt sie zum Lösen der Verbindung die `disconnect()` Methode (Zeile 29), setzt die neuen Endpunkte und aktiviert die Verbindung durch Aufruf von `reconnect()` (Zeile 32).

VirtualAttribute

`VirtualAttribute` dient zur Darstellung der virtuellen Attribute. Die Klasse enthält dabei einen Verweis auf das referenzierte Attribut. Um das Modell konsistent zu halten, muss garantiert sein, dass der Objekttyp des referenzierten Attributs jederzeit über Relationen erreichbar sein muss, er also vom referenzierten Objekttyp abgeleitet ist. Hierfür gibt es zwei verschiedene Möglichkeiten. Die naheliegendste Lösung wäre, bei jeder Änderung einer Relation alle virtuellen Attribute darauf zu überprüfen, ob die referenzierten Attribute noch erreichbar sind. Dabei müssten durch den Benutzer keine weiteren Angaben außer der Referenz erfolgen und es könnte jeder beliebige Pfad ermittelt werden. Diese Lösung erfordert eine zentrale Liste mit allen virtuellen Attributen, die ständig aktualisiert werden muss, um die virtuellen Attribute überprüfen zu können. Jedoch ergibt sich mit steigender Anzahl an virtuellen Attributen und Relationen ein großer Rechenaufwand, da bei jeder Änderung der Datenstruktur alle virtuellen Atribute der gesamten Datenstruktur erneut auf Plausibilität überprüft werden müssen. Daher wurde hier eine alternative Lösung gewählt. Bei dieser Variante wird bei der Erstellung des virtuellen Attributs ein Pfad von Relationen, über die das referenzierte Attribut erreicht werden kann, als Liste gespeichert. Zusätzlich dazu wird das virtuelle Attribut in allen Relationen des Pfads in einer Liste abgespeichert, um die Abhängigkeit verfügbar zu haben. Falls eine Relation, die Teil eines Pfads ist, gelöscht wird können die betroffenen virtuellen Attribute aus der Liste in der Relation ausgelesen werden. Diese können dann entweder gelöscht werden oder durch einen Alternativpfad versorgt werden. Beim Löschen des virtuellen Attributs genügt es, dieses bei den am gewählten Pfad beteiligten Relationen auszutragen.

SimplePredicateType

Zur Speicherung von Vergleichsoperationen dient die Klasse `SimplePredicateType`, deren Klassendiagramm Abbildung 25 zeigt. Das Ziel des Vergleichs kann ein Attribut oder ein

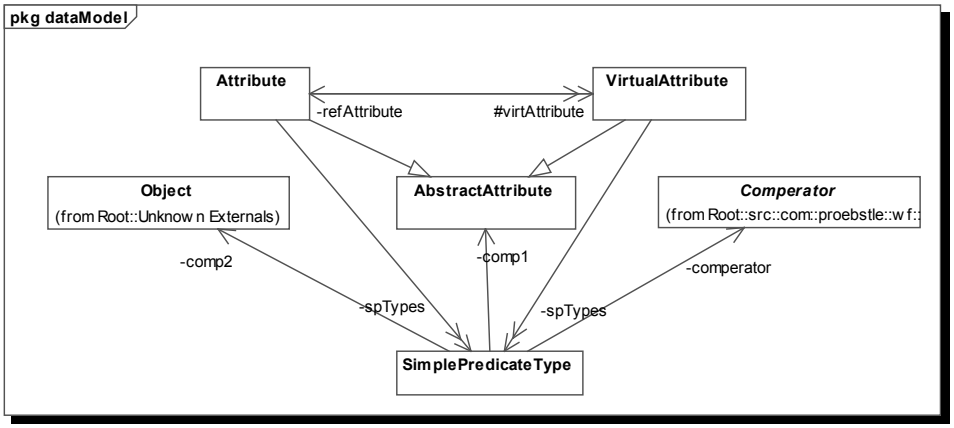

Abbildung 25.: Aufbau `SimplePredicateType`

virtuelles Attribut sein und wird daher als deren Oberklasse `AbstractAttribute` gespeichert. Als zweiter Operand dient eine Variable `Object`, da diese je nach Datentyp des zu vergleichenden Attributs verschiedene Typen symbolisieren kann. Als Vergleichsoperator dient die abstrakte Klasse `Comparator`, für die es für jede mögliche Ausprägung des Operators eine spezifische Klasse gibt. `Comperator` bietet für die Korrektheitsüberprüfung die Methode `validate(AbstractAttribute, Object)` an, damit sichergestellt wird, dass der Vergleich für das gewählte Attribut möglich ist.

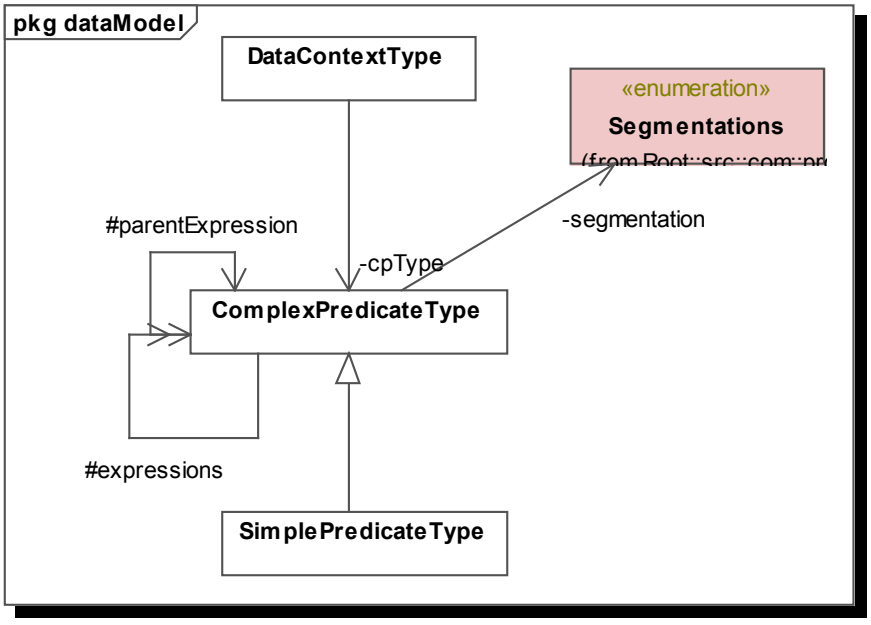

Abbildung 26.: Klassenaufbau der Prädikate

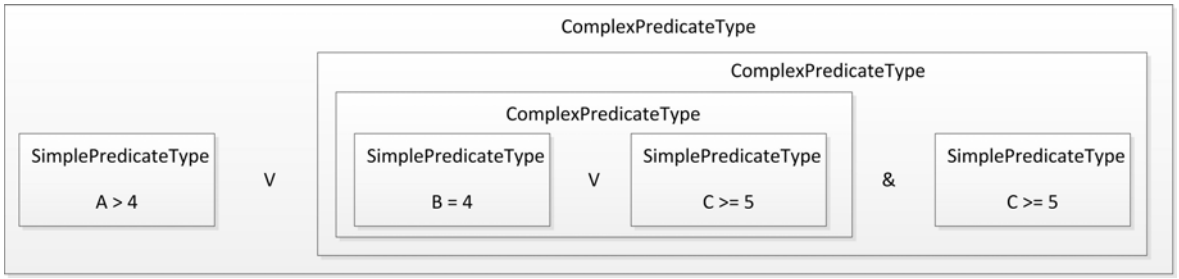

Abbildung 27.: Beispiel für Schachtelung der Prädikate

ComplexPredicateType

Mehrere `SimplePredicateTypes` können durch einen `ComplexPredicateType` (siehe Abbildung 26) zu komplexen logischen Ausdrücken verknüpft werden. Die Klasse `ComplexPredicateType` dient dabei als Container für alle Ausdrücke, die mit dem selben Operator (`segmentation` in Abbildung 26 genannt) verknüpft sind. Die verknüpften einzelnen Ausdrücke sind dabei in der Liste `expressions` vom Typ `ComplexPredicateType` gespeichert. Dadurch kann sowohl ein `SimplePredicateType`, der von `ComplexPredicateType` abgeleitet ist, als einfacher logischen Vergleich wie auch ein komplexer Ausdruck vom Typ `ComplexPredicateType` miteinander verknüpft werden. Diese Möglichkeiten zur Darstellung komplexer Ausdrücke sind in Abbildung 27 dargestellt.

DataContextType

Datenkontexte dienen dazu, den Bezug der `ComplexPredicateType` zum zugehörigen `ObjectType` herzustellen. Da dieser für jeden `DataContextType` unveränderlich ist, wird der `ObjectType` in einer finalen Variablen gespeichert. Eine weitere Wahrheitswert-Variable beschreibt die Erzeugung des Prädikats. Dies resultiert aus der bereits beschriebenen Schwierigkeit, die Eingabe komplexer Ausdrücke graphisch zu unterstützen. Die Variable gibt dabei an, ob das Prädikat über den graphischen Editor oder die Eingabezeile erzeugt wurde, damit die Anzeige entsprechend erfolgen kann.

5.2.2. Prozessstruktur

Die Prozessstruktur beschreibt den Prozessablauf innerhalb der Objekttypen. Somit muss für jeden Objekttyp genau eine Prozessstruktur existieren. Um dies sicherzustellen, wird bei der Erzeugung eines neuen Objekts automatisch eine Prozessstruktur mit einem minimalen Prozess erzeugt. Abbildung 28 bietet eine Übersicht über die Zusammenhänge der für die Verwaltung der Prozessstruktur benötigten Klassen. Im folgenden werden die wichtigsten Klassen mit ihren Zusammenhängen beleuchtet.

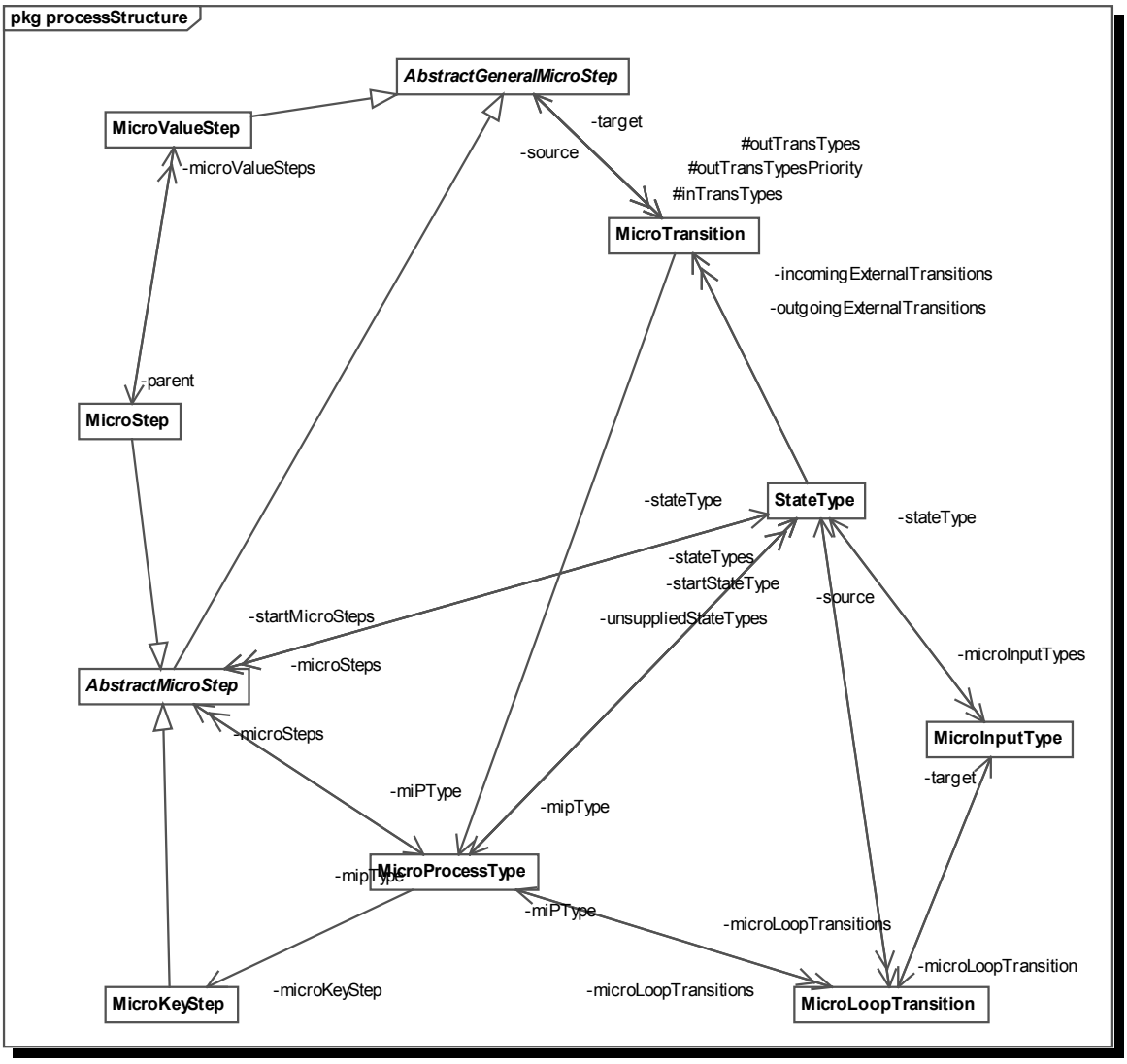

Abbildung 28.: Klassendiagramm Prozessstruktur

MicroProcessType

Die verwaltende Klasse für die Mikroprozessstruktur ist `MicroProcessType`, die für jeden `ObjectType` genau einmal bei der Erzeugung angelegt wird. MicroProcessTyp besitzt dabei wie auch aus Abbildung 29 ersichtlich ist, direkte Verweise auf die grundlegenden Elemente des Mikroprozesses. Dies sind die zugehörigen Zustandstypen (StateType), den Start-Mikroprozessschritt (MicroKeyStep) und alle eigenständigen Mikroprozessschritte (AbstractMicroStep). Die Liste `stateTypes` enthält dabei alle zum Mikroprozess gehörenden Zustandstypen. Die zusätzliche Liste der `unsuppliedStateTypes` enthält dagegen nur die Zustandstypen, die im Mikroprozess noch nicht durch eingehende Transitionen versorgt sind. Dies dient der einfachen Auffindung der noch nicht angeschlossenen Elemente, die auch bei der Anordnung gesondert berücksichtigt werden müssen. Zusätzlich existiert mit einem gesonderten finalen Verweis auf den `startStateType` eine Verknüpfung mit dem Startzustand des Prozesses, mit dem die Modellierung beginnt. Die einzelnen Mikroprozessschritte sind in der Liste `microSteps` aufgeführt. Zur einheitlichen Speicherung sind sie in der abstrakten, übergreifenden Varian-

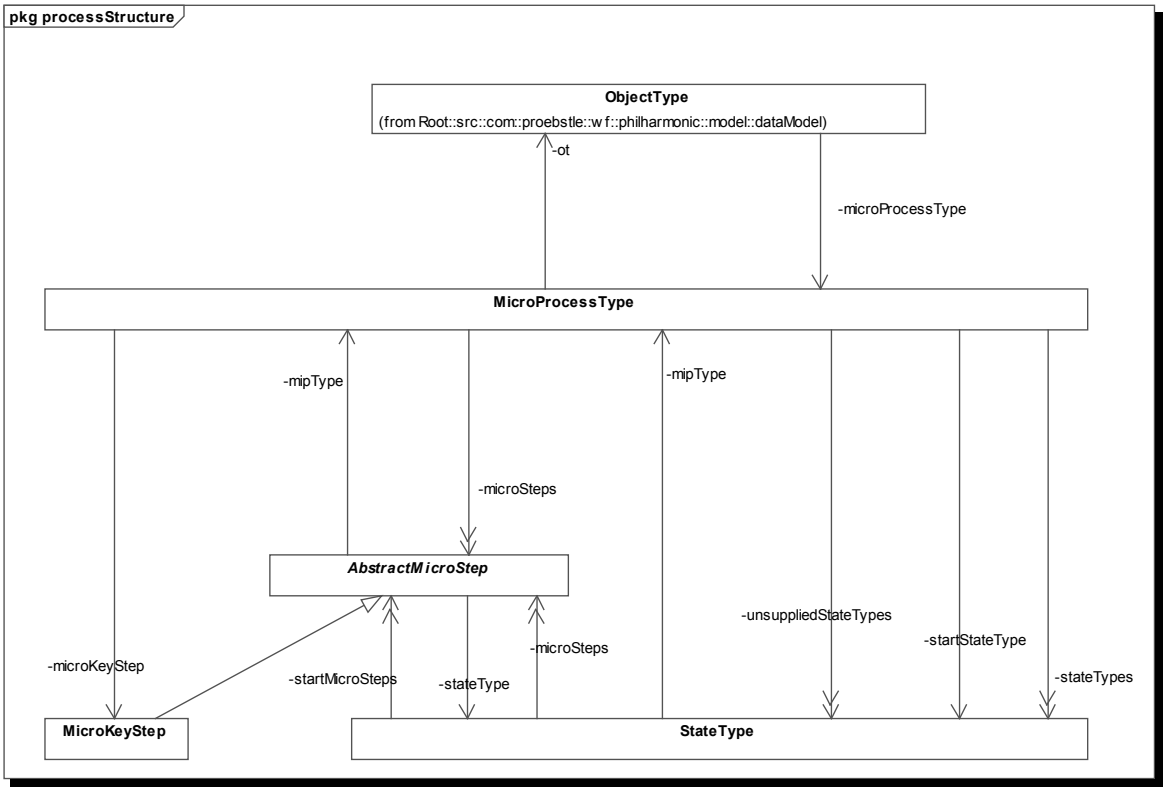

Abbildung 29.: Abhängigkeiten MicroProcessType

te `AbstractMicroStep` verwaltet. Für den Start-Mikroprozessschritt existiert zur einfachen Verwaltung, da er sich nicht ändert, ebenfalls ein finaler direkter Verweis.
Der Konstruktor erstellt beim Erzeugen einer neuen Instanz automatisch einen minimalen, initialen Prozess, der aus einem Startzustand mit Start-Mikroprozessschritt und einem Endzustand besteht. Dieser leere Standardprozess kann durch den Benutzer dann an seine Anforderungen angepasst werden.

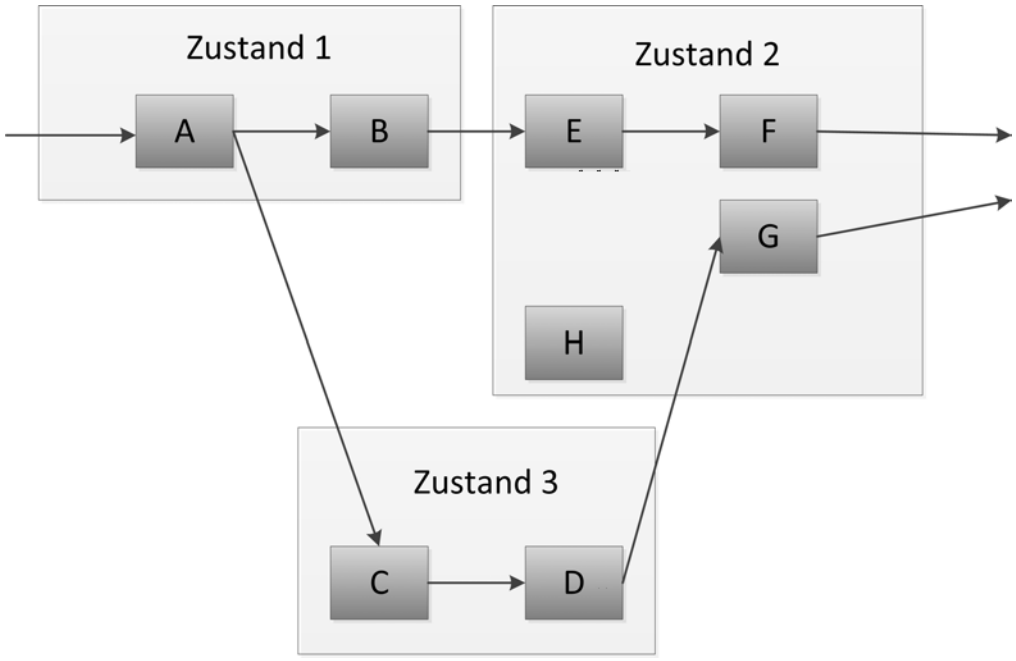

Abbildung 30.: Ausschnitt eines Beispielgraphs mit externen und internen Transitionen

StateType

Die einzelnen Zustände werden durch `StateType` dargestellt. In Abbildung 28 ist die Einordnung der StateType innerhalb der Prozessstruktur zu sehen. Einen genaueren Einblick bietet Abbildung 31, die deren wichtigste Abhängigkeiten beschreibt. Die Klasse verwaltet die ihr zugehörigen Mikroprozessschritte innerhalb der Liste `microSteps`. Die nicht durch interne Transitionen versorgten Mikroprozessschritte werden zusätzlich in einer separaten Liste, der Liste `startMicroSteps`, gespeichert. Die Liste `startMicroSteps` enthält somit alle Mikroprozessschritte, die durch externe Transitionen versorgt werden oder Schritte, die keine eingehende Transition besitzen. Im Beispiel in Abbildung 30 wären folglich die Schritte E, G und H in der `startMircoSteps` Liste. Dies ermöglicht eine einfachere Vorgehensweise bei der Anordnung

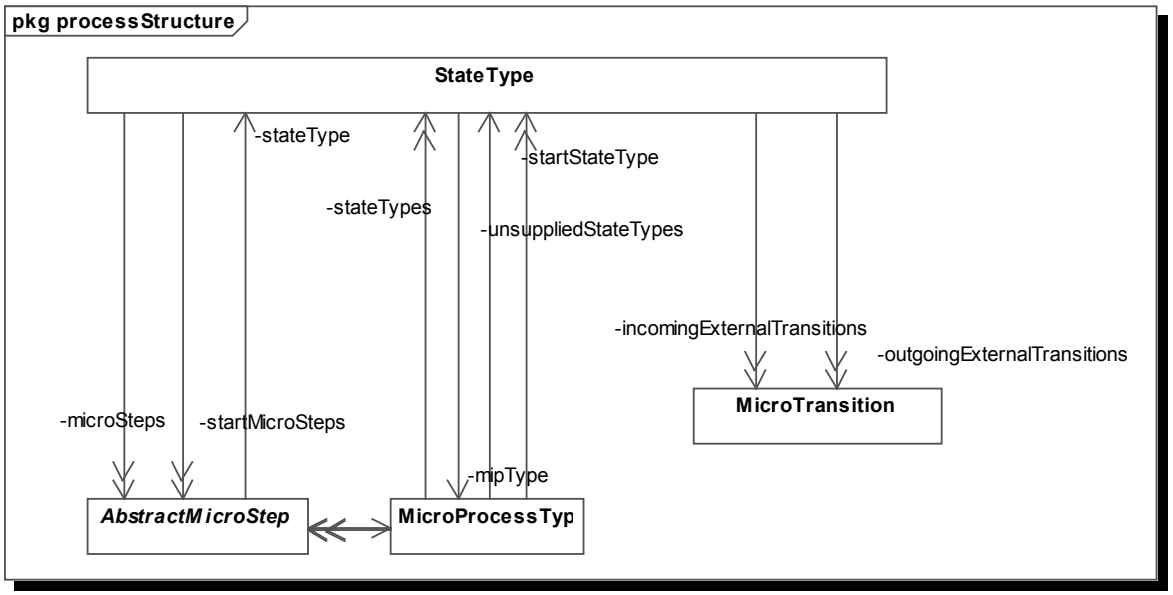

Abbildung 31.: Klassendiagramm ausgehend von StateType

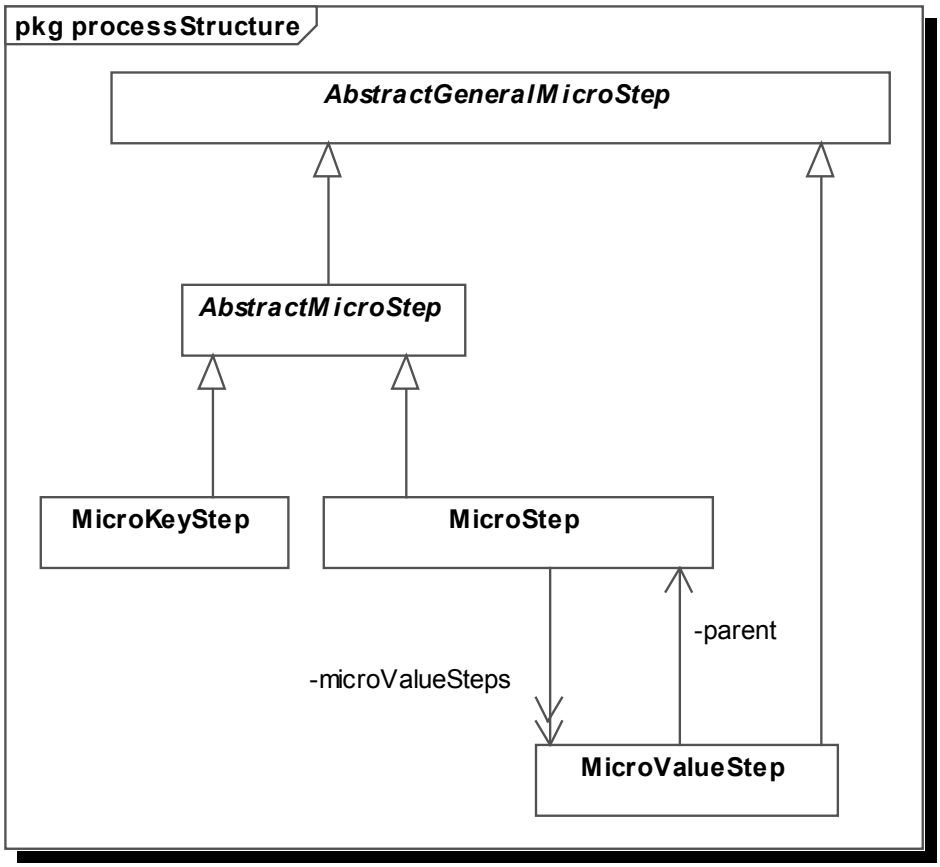

Abbildung 32.: Hierarchisches Klassendiagramm der Mikroprozessschritte

der Schritte, da diese wie in 5.3.5 beschrieben, von diesen Schritten ausgeht. Die Beziehungen zum Mikroprozess sind bereits unter `MicroProcessType` erwähnt. Außerdem besitzt die Klasse Listen mit Referenzen auf die eingehenden und ausgehenden externen Transitionen. Daraus lassen sich die vorhergehenden und nachfolgenden Zustände erkennen. Außerdem kann dadurch einfach erkannt werden, ob ein Zustand noch nicht versorgt wird.

Prozessschritte

Der Begriff Prozessschritte wird hier übergreifend für die verschiedenen möglichen auftretenden Varianten (`AbstractMicroStep`, `MicroStep`, `MicroValueStep`) verwendet. Die Varianten lassen sich anhand der in Abbildung 32 zu sehenden Hierarchie zueinander in Bezug setzen. Die gemeinsame Oberklasse für die verschiedenen Prozessschritte ist dabei der `AbstractGeneralMicroStep`.

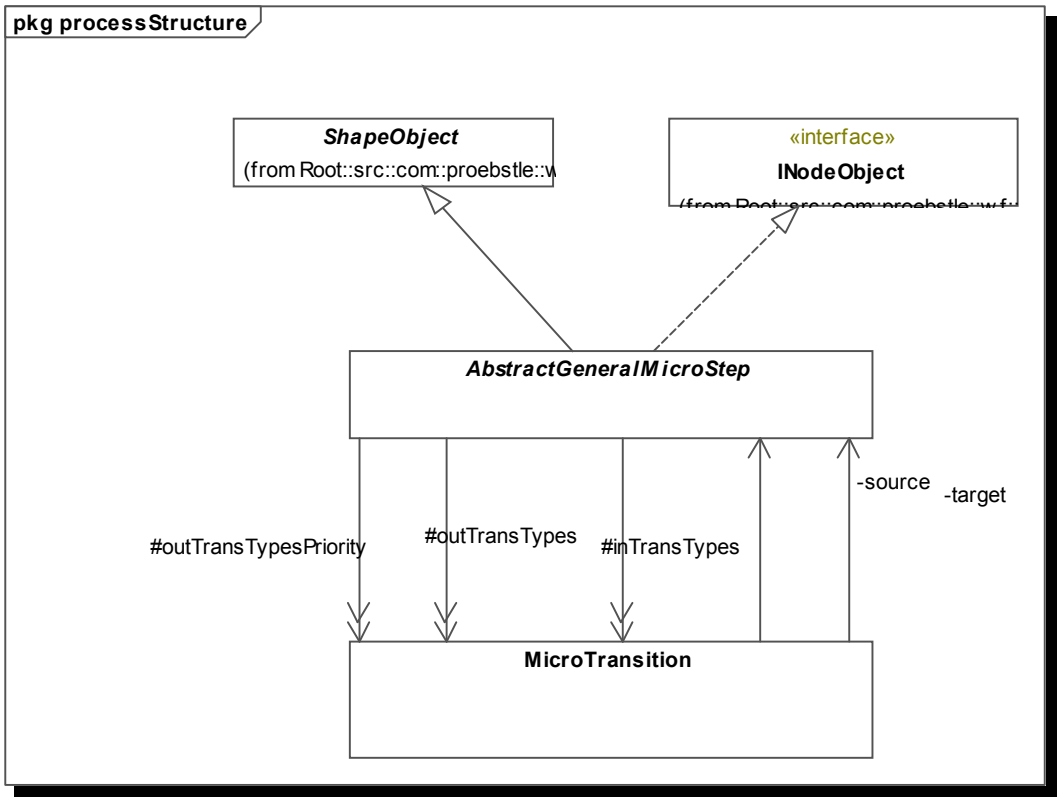

Abbildung 33.: Beziehungen der Prozessschritte zu Mikrotransitionen

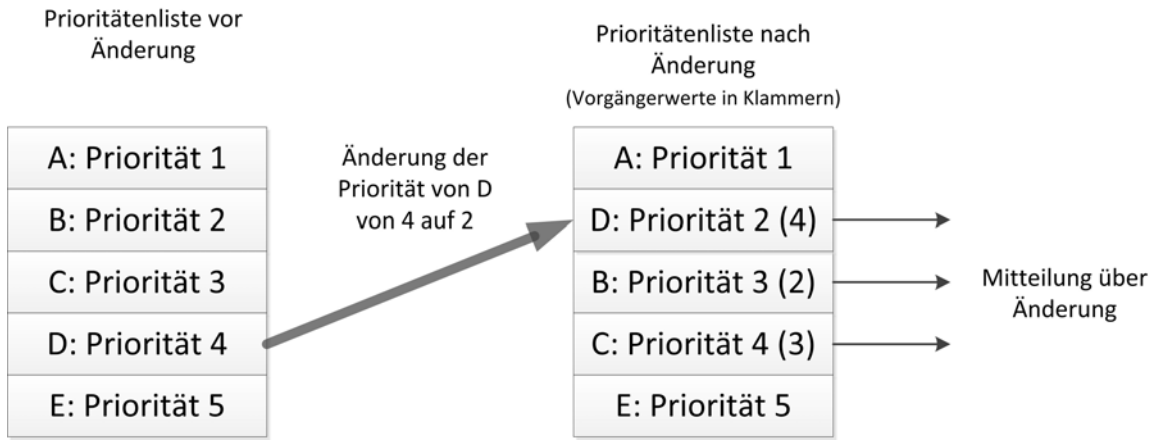

Abbildung 34.: Beispiel für Prioritätenänderung

AbstractGeneralMicroStep

`AbstractGeneralMicroStep` dient als gemeinsame Oberklasse für alle Varianten von Prozessschritten und erweitert die Klasse `ShapeObject` für im Editor eigenständige Objekte. Sie implementiert zusätzlich das bereits zu Beginn des Kapitels beschriebene Interface `INodeObject` und bietet somit die Möglichkeiten Transitionen daran anzudocken. Die zwischen den Prozessschritten und den Mikrotransitionen bestehenden Abhängigkeiten sind in Abbildung 33 zu sehen. Die Liste outTransTypes enthält dabei die vom Prozessschritt abgehenden Transitionen und inTransTypes die in den Prozessschritt mündenden Transitionen. Mit diesen ist es möglich die Vorgänger und Nachfolger des Prozessschritts zu ermitteln. Außerdem kann dadurch festgestellt werden, ob einem Prozessschritt eingehende oder ausgehende Transitionen für die Korrektheit fehlen. Bei mehreren ausgehenden Transitionen ist es notwendig Prioritäten für diese zu verwalten. Hierfür dient die Liste outTransTypesPriority in der die Transitionen analog zu ihrer Priorität einsortiert sind. Da die Prioritäten so zentral gespeichert werden, garantiert

```
1  //Funktion addSourceConnection fügt eine ausgehende Verbindung
   //zum Prozessschritt hinzu
2  FUNCTION addSourceConnection(Connection connection)
3  INPUT: Connection connection (die neue ausgehende Verbinding)
4  OUTPUT: -
5  BEGIN
```

```
6       //Iteration über alle ausgehende Transitionen
7       FOR outTransTypes
8           IF connection.target = outTransType.target THEN
9               //Falls bereits eine Verbindung zwischen den glei-
                //chen Prozessschritten besteht, wird eine Fehler-
                //meldung zurückgegeben.
10              THROW doppelte Transition
11          END IF
12      END FOR
13      //Hinzufügen zur ausgehenden Transitionsliste und Prioritä-
        //tenliste
14      outTransTypes.add(connection)
15      outTransTypePriority.add(connection)
16      IF connection.target.stateType != this.stateType THEN
17          //Falls die Transition in einem anderen StateType endet,
            //muss sie als externe Transition im StateType gespei-
            //chert werden.
18          this.stateType.addSourceConnection(connection)
19      END IF
20
21  END FUNCTION
```

Beispielcode 4.: Hinzufügen einer ausgehenden Transition

dies die Eindeutigkeit der Prioritäten. Für eine effizientere Verfügbarkeit der Priorität wird diese in den Mikrotransitionen ebenfalls zwischengespeichert. Bei einer Änderung der Priorität einer Transition wird diese in der Liste dann an die jeweilige Position verschoben und so die gewünschte neue Ordnung hergestellt. Die von der Neuordnung betroffenen Mikrotransitionen, also alle in dem Bereich zwischen der jetzigen Priorität und der vorherigen Priorität, werden anschließend über die Änderung benachrichtigt. Abbildung 34 zeigt ein Beispiel zur Veranschaulichung des Vorgangs.
Beim Hinzufügen neuer Transitionen zu einem Prozessschritt [siehe Beispielcode 4] wird zuerst sichergestellt, dass bisher keine Mikrotransition zwischen den beiden beteiligten Prozessschritten besteht (siehe Zeile 7-12). Anschließend wird die neue Transition zur ausgehenden Transitionsliste und ans Ende der Prioritätsliste eingefügt. Falls das Ziel der Transition in einem anderen

```
1  //Teilfunktion zur Berechnung der neuen Ordnung des Schritts
2  FUNCTION recalculateOrderAfterAdding(int newOrder)
3  INPUT: int newOrder (Ordnung des neuen Vorgängerschritts)
4  OUTPUT: -
5  BEGIN
6      IF newOrder >= aktuelle Ordnung THEN
7          Ordnung = newOrder + 1
8          FOR ausgehende Transitionen
9              ausgehendeTransition.recalculateOrderAfterAdding(Ord-
                   nung)
10         END FOR
11     END IF
12 END FUNCTION
13
14 //Funktion addTargetConnection fügt eine eingehende Verbindung
   //zum Prozessschritt hinzu
15 FUNCTION addTargetConnection(Connection connection)
16 INPUT: Connection connection (die neue eingehende Verbinding)
17 OUTPUT: -
18 BEGIN
19     //Iteration über alle eingehende Transitionen
20     FOR inTransTypes
21         IF connection.source = inTransType.source THEN
22             //Falls bereits eine Verbindung zwischen den glei-
               //chen Prozessschritten besteht, wird eine Fehler-
               //meldung zurückgegeben.
23             THROW doppelte Transition
24         END IF
25     END FOR
26     //Hinzufügen zur eingehenden Transitionsliste
27     inTransTypes.add(connection)
28     IF istVersorgt() = falsch THEN
29         //Falls der Prozessschritt noch nicht durch eingehende
           //Transitionen versorgt ist, diese Methode unterscheidet
           //sich je nach Prozessschritttyp
```

```
30          setze als versorgt
31      END IF
32      recalculateOrderAfterAdding(connection.source.Order)
33      IF connection.source.stateType != this.stateType THEN
34          //Falls die Transition in einem anderen StateType be-
            //ginnt muss sie als externe eingehende Transition im
            //StateType gespeichert werden.
35          this.stateType.addTargetConnection(connection)
36      END IF
37
38  END FUNCTION
```

Beispielcode 5.: Hinzufügen einer eingehenden Mikrotransition

Zustandstyp als der Ursprung ist, wird die Transition im Zustand ebenfalls als ausgehende Transition gespeichert werden (siehe Zeile 16-19). Diese gesonderte Speicherung ist wie bereits in StateType beschrieben zur Erkennung des nachfolgenden Zustands notwendig. Beim Löschen einer ausgehenden Transition wird diese wieder aus den beiden Listen des Prozessschritts entfernt und die zwischengespeicherten Prioritäten in den anderen ausgehenden Mikrotransitionen berichtigt.

Beim Hinzufügen von eingehenden Mikrotransitionen wird in der Methode `addTargetConnection`[Beispielcode 5] geprüft, ob der Prozessschritt durch eine andere Transition bereits versorgt ist (Zeile 28). Die Überprüfung der Versorgung unterscheidet sich dabei je nach Art

```
1   //rekursive Funktion zur Berechnung der Ordnung eines Prozess-
    //schritts nach einer Löschoperation, das heißt, wenn die Ord-
    //nungszahl sinkt.
2   FUNCTION recalculateOrderAfterRemoving()
3   INPUT: -
4   OUTPUT: -
5   BEGIN
6       //Ermittlung der maximalen Vorgängerordnung
7       maxPrevOrder = maximale Vorgängerordnung
8       IF maxPrevOrder + 1 < aktuelle Ordnung THEN
```

```
9          //Falls die berechnete Ordnung kleiner ist, wird die
           //Ordnung des Prozessschritts korrigiert
10         Ordnung = maxPrevOrder + 1
11         FOR ausgehende Transitionen
12             //Korrektur der nachfolgenden Transitionen
13             ausgehendeTransition.recalculateOrderAfterRemoving
14         END FOR
15     END IF
16 END FUNCTION
```

Beispielcode 6.: Ordnungskorrektur nach dem Löschen einer Transition

des Prozessschritts und ist daher in den einzelnen konkreten Klassen genauer definiert. Zuvor nicht versorgte Schritte werden daraufhin als versorgt markiert. Diese Markierung ist notwendig, da nur an versorgte Schritte weitere Transitionen angedockt werden können. Abschließend erfolgt eine Kontrolle ob die Ordnung des Prozessschritts geändert werden muss. Dabei wird die aktuelle Ordnung mit der Ordnung des neuen Vorgängers verglichen (Beispielcode 5, Zeile 6) und wenn nötig korrigiert. Falls eine Änderung erfolgt ist, müssen die Nachfolger des aktuellen Schritts ebenfalls überprüft werden (Zeile 9), da diese beeinflusst sein können. Beim Löschen einer eingehenden Verbindung muss nach dem Entfernen aus der Liste der eingehenden Transitionen überprüft werden, ob der Prozessschritt noch versorgt ist. Außerdem wird geprüft, ob sich eine Änderung der Ordnung ergeben hat. Dies erfolgt gemäß Beispielcode 6. Dabei wird zuerst die maximimale Ordnung aller Vorgänger nach der Löschoperation ermittelt (Zeile 7). Falls die neu errechnete Ordnung kleiner ist als die zuvor berechnete (Zeile 8), wird diese korrigiert und die nachfolgenden Prozessschritte ebenfalls überprüft (Zeile 13). Zur Effizienzsteigerung werden dabei nur dann die Nachfolger kontrolliert, wenn Änderungen beim aktuellen Prozessschritt aufgetreten sind, ansonsten wird die Rekursion abgebrochen.

AbstractMicroStep

`AbstractMicroStep` ist die gemeinsame Oberklasse für die selbstständig einem Zustand untergeordneten Klassen `MicroKeyStep` und `MicroStep`. Diese besitzen damit eine eigene Ordnung und sind direkt einem `StateType` zugeordnet.

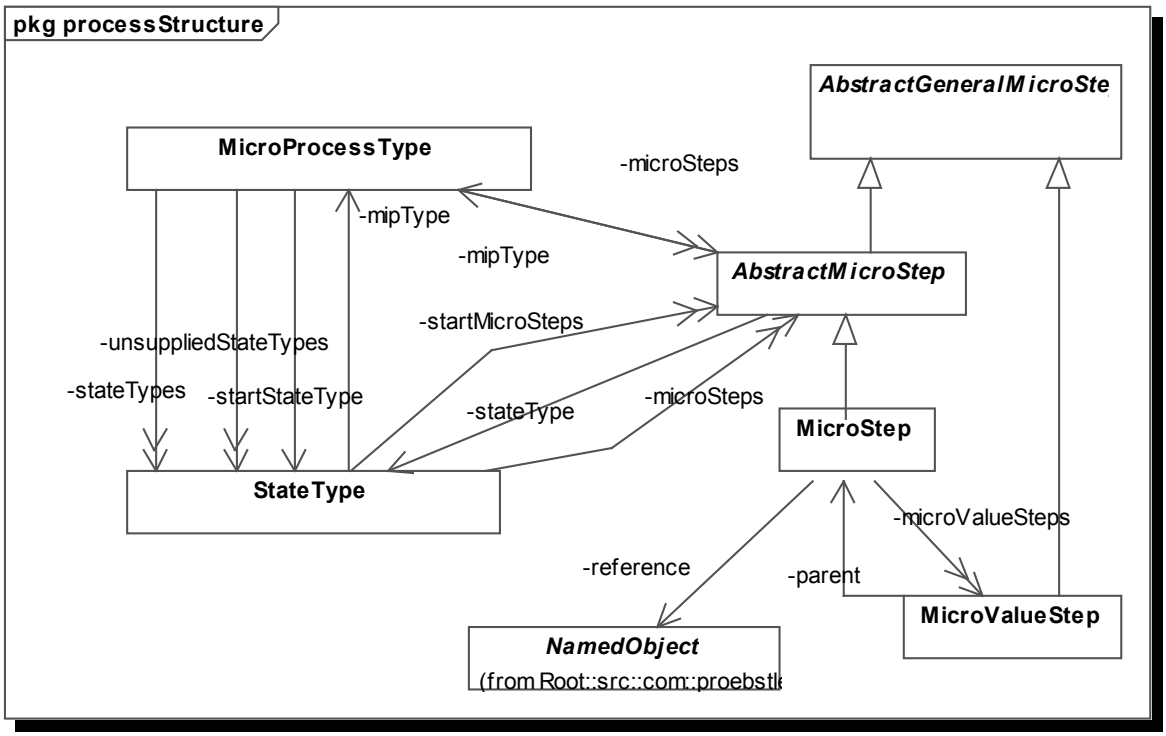

Abbildung 35.: Beziehungen der Prozessschritte zu Mikrotransitionen

MicroKeyStep

Start-Mikroprozessschritte werden durch die Klasse `MicroKeyStep` dargestellt. Die Klasse ist, wie aus Abbildung 32 ersichtlich ist, eine direkte Ableitung von `AbstractMicroStep`. Da ein Start-Mikroprozessschritt den Anfang eines Prozesses markiert, wird die Ordnung immer als Stufe 0 zurückgegeben und ist nicht änderbar. Auch gibt die `IsSupplied()` Methode bei Anfragen nach dem Versorgungsstatus immer wahr zurück, da an den Start-Mikroprozessschritt immer ausgehende Mikrotransitionen angedockt werden können. Zusätzlich wird die zuvor beschriebene geerbte `addTargetConnection` Methode zum Hinzufügen neuer eingehender Transitionen überschrieben und gibt nur eine Fehlermeldung zurück, da eingehende Transitionen für den Startschritt nicht erlaubt sind, da er sonst nicht mehr der erste Schritt im Prozess sein kann.

MicroStep

`MicroStep` beschreibt die Mikroschritte. Diese besitzen, falls es sich um Mikroschritte vom Typ „atomar“ handelt, eine Referenz auf ein Attribut oder eine Relation. Diese wird als deren

kleinster gemeinsamer Obertyp `NamedObject` gespeichert. Beim Setzen und Löschen der Referenz wird automatisch der Typ des Mikroschritts passend gewählt. Dies minimiert die Möglichkeit von Benutzerfehlern und erspart dem Benutzer zusätzlichen Aufwand. Falls dem Mikroschritt Mikro-Wertschritte zugeordnet sind, werden diese ebenfalls im Mikroschritt verwaltet. Dieser benötigt deshalb eine angepasste `isSupplied()` Methode, da die Versorgungsbedingung erfüllt ist, wenn alle Mikro-Wertschritte versorgt sind oder der Mikroschritt selbst versorgt ist.

MicroValueStep

`MicroValueStep` erweitert die `AbstractGeneralMicroStep` um Methoden zur Verwaltung der ihm zugewiesenen Wertbeschränkungen der Typen `ComplexPredicateType` oder `DataContextType`. Dabei wird darauf geachtet, dass jeder Schritt nur eine Beschränkung besitzen darf, die auf die Referenz des übergeordneten Mikroschritts passt. Die Ordnung eines Mikro-Wertschritts wird im übergeordneten Mikroschritt verwaltet, da die einzelnen Mikro-Wertschritte dieselbe Ordnung besitzen.

MicroTransition

`MicroTransition` erweitert die `Connection` Klasse um die Mikrotransitionen darzustellen. Der Ursprung und das Ziel der Mikrotransition werden mit der Methode `reconnect(Source, Target)` gesetzt. Beim Aktivieren der Verbindung innerhalb der Methode `reconnect()` wird geprüft, ob der Ursprung und das Ziel dem selben Zustand angehören, um zu entscheiden ob es eine interne oder externe Transition ist. Bei externen Transitionen wird der Schalttyp standardmäßig auf implizit gesetzt. Die Schaltpriorität der Mikrotransitionen wird wie bereits bei `AbstractGeneralMicroStep` erwähnt durch die Position innerhalb einer Liste bestimmt. Die Position in der Liste und damit die Priorität wird ebenfalls direkt in `MicroTransition` in einer Variablen zwischengespeichert. Dies erfolgt aus Gründen der Effizienz, da öfter Anfragen nach der Priorität erfolgen und so nicht jedes Mal erneut die Liste nach der Position der Transition durchsucht werden muss.

5.3. Implementierung Oberfläche

Das zentrale Element der Bearbeitungsoberfläche bilden die Editoren. Die einzelnen Editoren sind dabei als Untereinheiten eines übergeordneten, diese verwaltenden Editors realisiert. Wie Abbildung 36 zeigt, ist der übergeordnete Editor in der Implementierung durch den `PHILharmonicFlowsEditor` realisiert, der den `MultiPageEditorPart` von Eclipse erweitert. `MultiPageEditorPart` ist eine abstrakte Klasse der Eclipse Rich Client Platform , die ein Grundgerüst für einen Editor bietet, der aus mehreren einzelnen Editorseiten besteht. Um die beiden verschiedenen Editortypen, die in Kapitel 4 bereits beschrieben wurden, allgemein im Editor verwalten zu können, wurde das einheitliche Interface `IAbstractEditor` eingeführt. Dieses beschreibt die Methoden, die jede Editorseite implementieren muss um als Unterseite des `PHILharmonicFlowsEditor` in Frage zu kommen.
Zur Vereinfachung von Implementierungen weiterer graphenbasierter Editoren nach dem Graphical Editig Framework dient die Klasse `AbstractGraphicalEditorPage`, die den `Gra-`

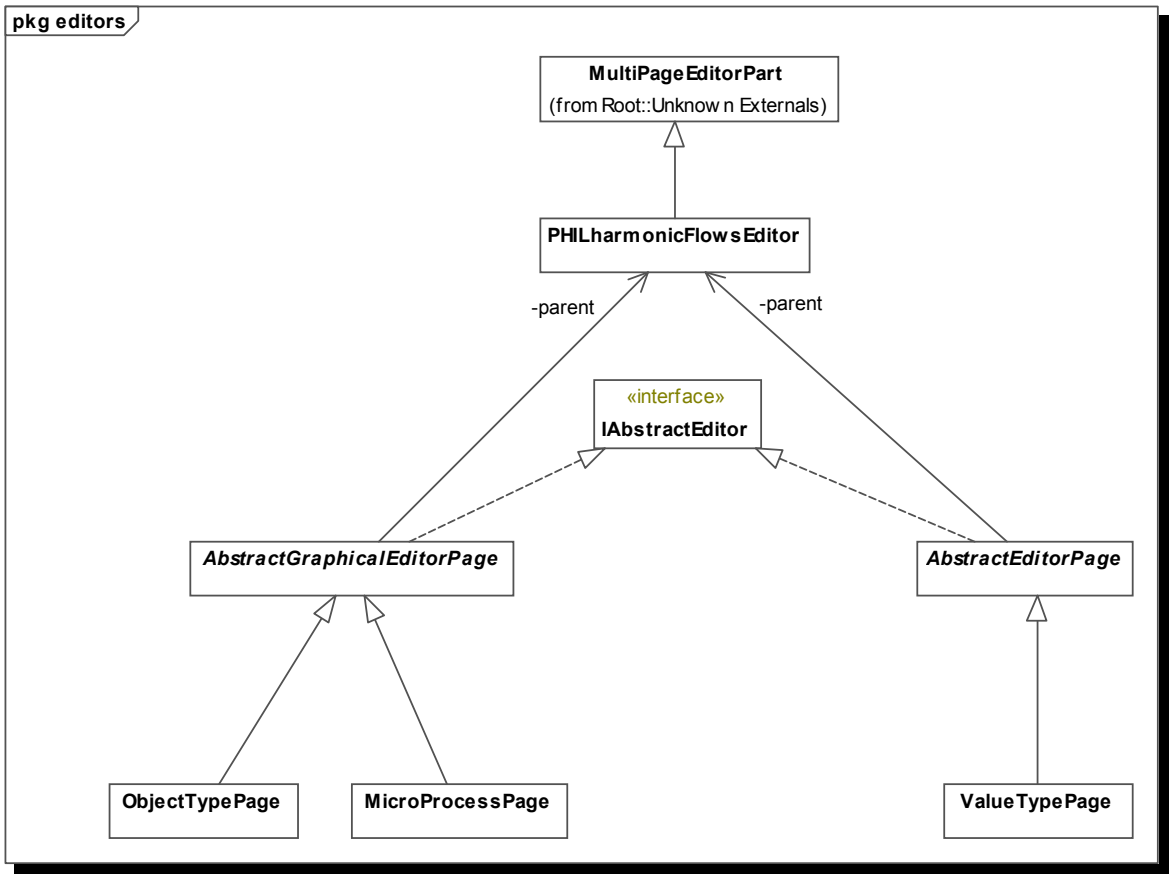

Abbildung 36.: Zusammenhang Editoren

phicalEditor des GEF erweitert und das IAbstractEditor Interface implementiert, sodass sie eine abstrakte Oberklasse für graphenbasierte Editorseiten des PHILharmonicFlowsEditor bildet. Konkrete Implementierungen auf dieser Basis sind der Objekttypen-Editor (ObjectTypePage) und der Mikroprozess-Editor (MicroProcessPage).
Im Gegensatz dazu bietet die AbstractEditorPage eine Grundimplementierung für all diejenigen Editorseiten, die wie der Wertetypen-Editor (ValueTypePage) nicht graphenbasiert aufgebaut sind.

5.3.1. PHILharmonicFlowsEditor

Der PHILharmonicFlowsEditor ist für das Öffnen und Speichern der Daten zuständig. Darüber hinaus initialisiert und verwaltet er die zentralen, gemeinsamen Daten und Funktionen der einzelnen Editoren.
Die Editoren besitzen einen gemeinsamen CommandStack, der für das Ausführen, Rückgängigmachen und Wiederholen von Operationen zuständig ist. Dies garantiert, dass keine syntaktischen Fehler bei der Rückgängigmachung von Befehlen erfolgen können, da dieses in der Reihenfolge der ursprünglichen Ausführung geschieht, die ja gezwungenermaßen korrekt war.
Auch das Outline Fenster wird zentral durch den PHILharmonicFlowsEditor verwaltet und erhält von diesem den jeweils geöffneten Editor als Grundlage der Übersicht zugewiesen. Die Outline bietet bei allen graphenbasierten Editoren eine verkleinerte, navigierbare, vollständige Übersicht des Editor. Die Implementierung der Outlineklasse erfolgt innerhalb der internen Klasse PHILHarmonicFlowsOutlinePage, da die beiden Klassen eng miteinander verzahnt sind.

Eigenschaften der Palette

Die Palette, deren Umsetzung in Abbildung 37 dargestellt wird, hilft bei der Erzeugung neuer Objekte in den Editoren. Dazu bietet sie die möglichen Objekte zur Auswahl, die selektiert und im Editor entsprechend angelegt werden können. Außerdem ermöglicht sie den Wechsel zwischen den Teileditoren durch Auswahl des entsprechenden Reiters. Die Palette wird ebenfalls zentral im PHILharmonicFlowsEditor erzeugt und verwaltet, damit sie einheitlich für alle Editorseiten genutzt werden kann.
Die Reiter und Einträge der Palette werden in der PHILharmonicFlowsEditorPaletteFactory Klasse innerhalb der createPalette Methode angelegt. Die Instanzierung der Palette erfolgt wiederum in einer internen Klasse des PHILharmo-

`nicFlowsEditor`, der `CustomPalettePage`. Die Lösung durch eine interne Klasse wurde gewählt, da auf einige der privaten Werte des Editors zurückgegriffen werden muss. Bei der Gestaltung der Palettenreiter trat das Problem auf, dass sich die Farben und das Verhalten der Palettenreiter von der Klasse `DrawerEditPart` nicht direkt anpassen lassen. Die Ursache für die fehlende Anpassungsmöglichkeit ist, dass die in Eclipse für die Erzeugung der Palette zuständigen Klassen sich im Paket `org.eclipse.gef.internal.ui.palette` des GEF und dessen Unterpaketen befinden. Auf diese ist der Zugriff jedoch eingeschränkt, sodass deren Methoden nicht angesprochen und die Klassen nicht erweitert werden können. Als zusätzliche Schwierigkeit ergibt sich dabei, dass die Farbgebung innerhalb einer privaten Methode erfolgt, sodass selbst nach erlangtem Zugriff nur eine Neuimplementierung der Klasse und aller übergeordneten Klassen dieses Problem lösen würde. Aufgrund der tiefen Integration der Palette in das System würde dies aber eine weitreichende Neuimplementierung des Frameworks voraussetzen, womit die Lösung voraussichtlich nicht mehr zu Updates der Eclipse Plattform kompatibel wäre. Da eine direkte Anpassung der Palette somit nicht realisierbar ist, musste auf alternative, weniger komfortable Wege zurückgegriffen werden. Als Lösung des Problems erfolgt die Steue-

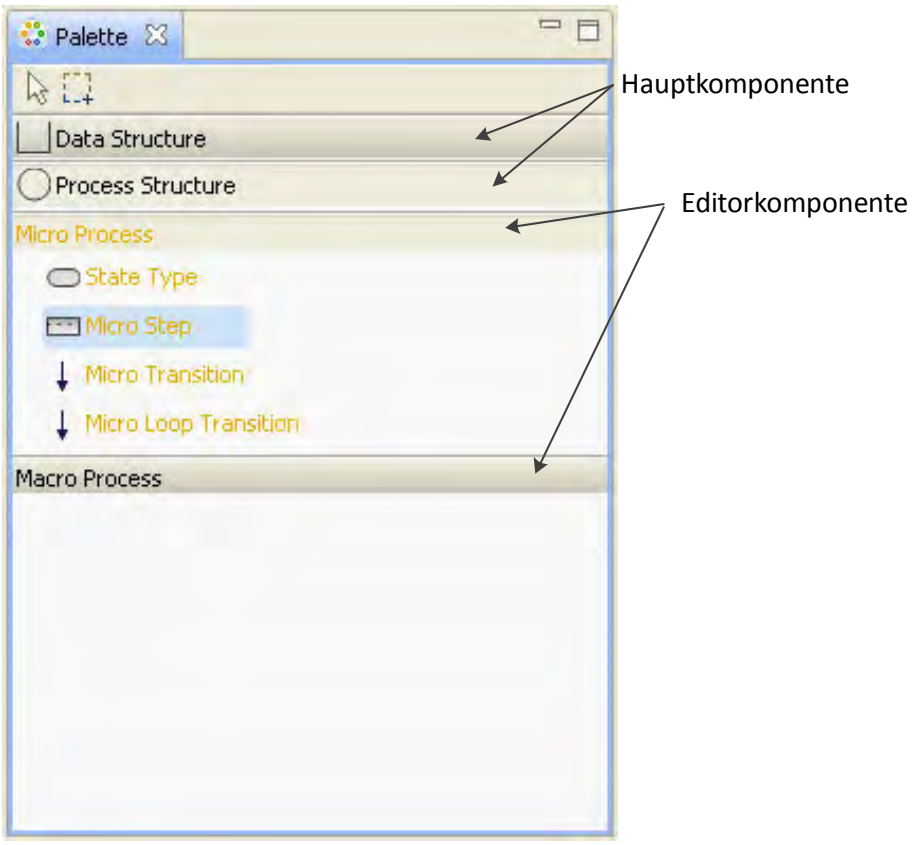

Abbildung 37.: Ansicht Palette

rung der Palette indirekt über einen Listener in dem ebenfalls die Auswahl des aktiven Editors erfolgt. Konkret ist dieser Vorgang innerhalb eines `SelectionChangedListener` realisiert, der in der `createControl` Methode der `CustomPalettePage` angelegt wird. Der Listener wird bei jeder Änderung der Reiterauswahl aktiviert. Daraufhin wird zuerst überprüft ob der neu gewählte Reiter der ersten oder zweiten Ebene angehört. Falls ein Reiter der übergeordneten Ebene gewählt ist, wird überprüft, ob in der diesem untergeordneten Ebene bereits ein Reiter aktiviert ist und somit geöffnet dargestellt wird. In allen Fällen in denen ein Reiter der untergeordneten Ebene aktiviert ist, wird der jeweilige Editor sofort geöffnet um den Benutzer nicht zu verwirren. Der Zugriff auf den aktivierten Reiter kann über eine Basisklasse des Palettenreiters realisiert werden. Über diese Basisklasse lässt sich jedoch nur Einfluss auf die Darstellung der Schrift nehmen um den ausgewählten Eintrag optisch hervorzuheben.

Das Eigenschaftsfenster

Das Eigenschaftsfenster (properties View) bietet eine Übersicht über die Eigenschaften des im Editor ausgewählten Objekts an und erlaubt die Änderung der Eigenschaften des selektierten Objekts. Für eine bessere Benutzerführung ist hier das tabbed properties view Plug-In des Eclipse RCP verwendet. Dieses unterstützt nämlich die Aufteilung des Eigenschaftsfensters in mehrere Tabreiter, die jeweils zusammengehörige Eigenschaften geordnet darstellen. Die am linken Rand befindlichen Reiter dienen dabei zur Auswahl des gewünschten Tabs. Die Tabs setzen sich wiederum aus einzelnen, untereinander angeordneten Sektionen zusammen, die jeweils in einer eigenen unabhängigen Klasse definiert werden. Die einzelnen Sektionen sind in Abbildung 38 zur Veranschaulichung blau umrandet.

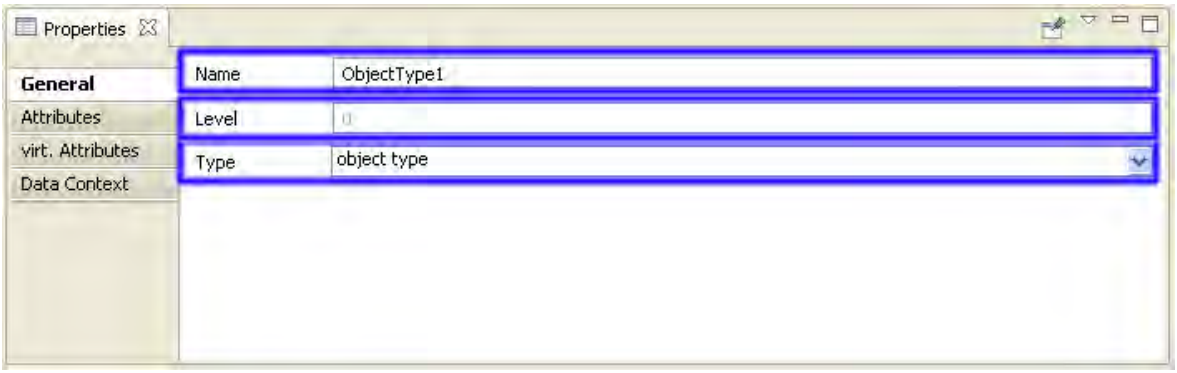

Abbildung 38.: Beispiel Eigenschaftsfenster mit Tabreitern

Dieser modulare Aufbau ermöglicht eine große Flexibilität, da die Kategorien nicht fest im Code festgelegt werden, sondern nur innerhalb der plugin.xml des Objekts definiert werden. Die einzelnen Sektionen wiederum werden ebenfalls innerhalb der plugin.xml auf die Kategorien verteilt und dabei die Reihenfolge der Anzeige festgelegt. Außerdem kann noch angegeben werden für welche ausgewählten Datentypen die ausgewählte Sektion im Eigenschaftsfenster angezeigt werden soll. Somit kann das Eigenschaftsfenster ohne großen Aufwand anders angeordnet oder für neue Datentypen angepasst werden und die erstellten Sektionen für andere Zwecke wiederverwendet werden.
Die Definition der Kategorien erfolgt dabei im Erweiterungspunkt *org.eclipse.ui.views.properties.tabbed.propertyTabs* in der `plugin.xml`. Die einzelnen Sektionen werden dem Plug-In im Erweiterungspunkt *org.eclipse.ui.views.properties.tabbed.propertySections* mit den dazugehörigen Klassen der Implementierung bekanntgemacht und den einzelnen Tabs zugeordnet.

5.3.2. Erzeugung neuer Modell-Klassen

Die Anlage von neuen Instanzen der Klassen erfolgt nach dem Kommando-Entwurfsmuster. Die Kommando-Klasse wird zur weiteren Verwendung in einem zentralen Kommando-Stack im `PHILharmonicFlowsEditor` gespeichert. Dadurch ist es möglich alle Aktionen rückgängig zu machen und danach ebenso wiederherzustellen. Die dazu notwendigen Kommando-Klassen befinden sich zur einfachen Übersicht in den *.commands Paketen der Implementierung.

5.3.3. Implementierung des Objekttypeneditors

Der Objekt Type Editor dient der grundlegenden Modellierung der Datenstrukur. Nach der Auswahl des Objekt Type Editors in der Palette wird im `PHILharmonicsFlowsEditors` durch Aufruf der `setPerspective` Methode des `IAbstractEditor` Interfaces die zugehörige Perspektive für die Oberfläche geladen. Diese besteht, wie bereits in Kapitel 4.3.1 erwähnt und in Abbildung 15 zu sehen, aus 5 Oberflächenelementen. Am linken Bildschirmrand werden dabei zuoberst das Outline Fenster, darunter das Fehlerfenster und die Palette angezeigt. Rechts davon nimmt die größere Fläche der graphenbasierte Editor ein. Unterhalb ist das Eigenschaftsfenster für die Anzeige der Details des im Editor ausgewählten Objekts platziert.
Die Palette bietet in der Object Type Editor Perspektive die Möglichkeit einen neuen Objekttyp zu erzeugen und einen neuen Relationentyp anzulegen, der zwei Objekttypen miteinander in Beziehung setzt.

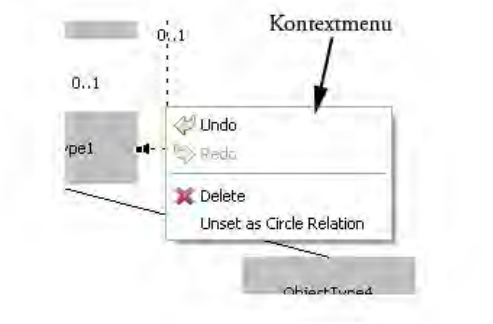

Abbildung 39.: Kontextmenu im Object Type Editor für eine Relation

Der graphische Editor wird durch einen `ScrollingGraphicalViewer` realisiert, der in der Methode `createGraphicalViewer` erzeugt wird. Der `ScrollingGraphicalViewer` unterstützt die Anzeige von graphenbasierten Inhalten und bietet zusätzlich eine native Unterstützung für die Auswahl des sichtbaren Bildschirmausschnitts um auch über die Bildschirmgrenzen hinausgehende Graphen zu realisieren. Dieses Verhalten dient der intuitiven Benutzerführung. Für Editieroperationen bietet der Viewer ein per Rechtsklick erreichbares Kontextmenu, wie es heutzutage in den meisten Anwendungen üblich ist. Dieses ist in Abbildung 39 beispielhaft dargestellt. Dessen Einträge werden in der Klasse `ObjectTypeEditorContextMenuProvider` zentral definiert.
Die für die Steuerung und Erzeugung der im Editor darzustellenden Objekte notwendigen EditParts werden durch eine Fabrikklasse erzeugt. Diese Fabrikklasse ist die `ObjectTypeEditPartFactory`. Die zugrundeliegende Zeichenfläche des Editors bildet die Klasse `ObjectTypeDiagramEditPart`, die somit einer graphischen Darstellung von `ObjectTypeDiagram` entspricht. Auf diesem werden die `ObjectTypeEditPart` für die Objekttypen platziert und können durch die `RelationTypeEditPart`, die Relationen verwalten, zueinander in Beziehung gesetzt werden.
ObjectTypeDiagramEditPart bildet die leere Zeichenfläche, auf der die weiteren EditParts angeordnet werden können. In der Methode `createEditPolicies` wird dem EditPart eine `XYLayoutEditPolicy` zugewiesen, um die Kindobjekte anhand XY-Koordinaten anzuordnen. Dies ist notwendig um die Objekttypen für die Anordnung selbstständig platzieren zu können und somit eine Anzeige anhand ihrer Level zu ermöglichen. Außerdem wird festgelegt, dass nur `ObjectTypeEditParts` auf diesem EditPart platziert werden dürfen, da für die Erzeugung von Relationen Ursprung und Ziel definiert sein müssen und diese somit an `ObjectTypeEditParts` platziert werden. Um einen durchgezogenen, farbigen Auswahlrahmen um das selektierte Objekt zu erhalten, wird der in der Klasse `SelectionBorderEditPolicy` eigens definierte Rahmen als Auswahlrahmen für die Objekte festgelegt.

ObjectTypeEditPart Der Controller für das `ObjectType` ist das `ObjectTypeEditPart`. Das EditPart hat eine EditPolicy, die sein Löschen erlaubt. Dabei wird die definierte Operation `ObjectTypeDeleteCommand` aufgerufen, die ebenfalls die zugrundeliegenden Modelldaten löscht. Eine weitere `Policy` beschreibt das `ObjectTypeEditPart` als Knoten und erlaubt damit das Anlegen und Verändern von Verbindungen. Diese sind hier durch die `RelationTypes` realisiert. Somit können Relationen als Pfeile zwischen den betroffenen Objekttypen dargestellt werden. Im Editor dargestellt wird das `ObjectType` durch ein Rechteck, in dem der Name steht [siehe Abbildung 42]. Falls das zugehörige `ObjectType` als UserType definiert ist, wird dabei rechts oben ein Symbol für einen Benutzer dargestellt. Dem zugehörigen `ObjectType` zugeordnete Datenkontexte werden durch die Anzeige der zugeordneten Anzahl visualisiert.
Die Anzeige aller weiteren Eigenschaften eines ausgewählten ObjectType und die Möglichkeit zur Änderung erfolgt im Eigenschaftsfenster. Diese ist zur Strukturierung in vier Kategorien unterteilt - *General*, *Attributes*, *virt. Attributes* und *Data Context* - die jeweils zusammengehörige Eigenschaften bündeln. In der *Generals* Kategorie ist eine Änderung des Namens

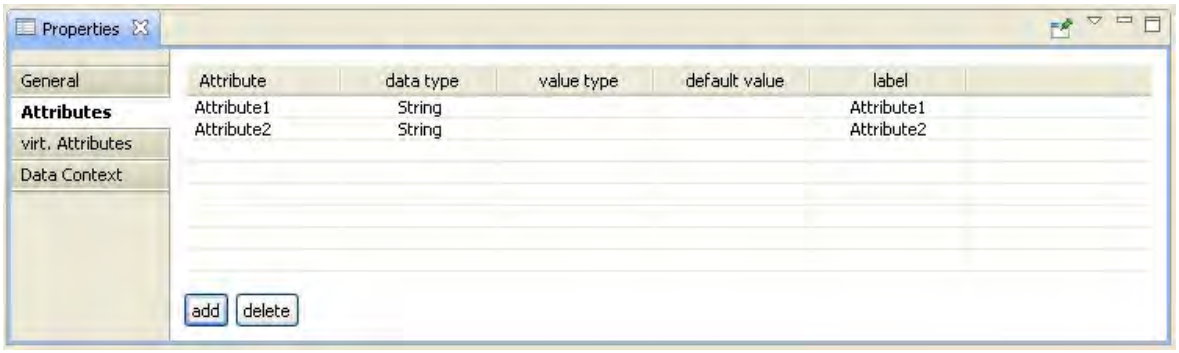

Abbildung 40.: Attribute Kategorie der properties view

und des Objekttyps möglich. Zusätzlich wird die aktuelle Ebene des Objekttyps angezeigt. Die *Attributes* Kategorie [siehe Abbildung 40] bietet eine Tabelle für die Anzeige der vorhandenen Attribute des Objekttyps. Den einzelnen Zellen der Tabelle sind zur Änderung der Attributwerte Editoren zugewiesen. Abhängig von der Spalte sind dies sowohl reine Texteditoren wie auch Combobox Editoren, aus denen vorgegebene Werte ausgewählt werden können um Fehleingaben zu vermeiden. Unterhalb der Tabelle befinden sich zwei Schaltflächen, eine zum Erzeugen neuer Attribute und eine zum Löschen des gerade in der Tabelle selektierten Attributs.
Die *virt. Attributes* Kategorie ist ähnlich der Attributes Kategorie aufgebaut und unterscheidet sich nur in den Spalten der Tabelle. Da zum Anlegen neuer virtueller Attribute jedoch

Abbildung 41.: Data Context Kategorie der properties view

festgelegt werden muss, auf welches Ausgangsattribut sich dieses bezieht, öffnet sich bei Auswahl der Schaltfläche „add“ ein Assistent. Die Auswahl des zugrundeliegenden Attributs erfolgt dabei zweistufig um dem Benutzer einen besseren Überblick über die Auswahlmöglichkeiten zu geben. Zuerst muss aus den angebotenen übergeordneten Objekttypen das gewünschte Ziel ausgesucht werden. Anschließend werden in einer weiteren Combobox die dafür möglichen Attribute zur Auswahl bereitgestellt.

Die *Data Context* Kategorie, die in Abbildung 41 zu sehen ist, enthält ein `cTabFolder`, also horizontal angeordnete Tabreiter zur Untergliederung des Inhalts. Jeder dieser Tabreiter stellt dabei einen eigenen Datenkontext dar. An letzter Position steht ein zusätlicher Tabreiter zur Erzeugung neuer Datenkontexte. Die Tabs der Datenkontexte bestehen aus einzelnen Eintragszeilen, die jeweils eine Wahrheitsüberprüfung darstellen und mit einem „und“ Operator miteinander verknüpft sind. Einzelne Zeilen können durch die am jeweiligen rechten Zeilenrand befindlichen Schaltfläche gelöscht werden.

RelationTypeEditPart Die `RelationTypeEditParts` sind für die Darstellung und Verwaltung der `RelationTypes` zuständig. Das Löschen des EditParts und der zugrundeliegenden Modelldaten wird durch eine `EditPolicy` geregelt. Die Darstellung im graphischen Editor erfolgt als Linie mit einer Pfeilspitze auf das übergeordnete Element wie in Abbildung 42 zu sehen. Zur einfachen Unterscheidung wird die Linie einer normalen Relation durchgezogen dargestellt und eine gestrichelte Linie gezeichnet, falls die Relation als Zyklusrelation markiert ist. Normale Relationen docken dabei an den horizontalen Seiten der `ObjectTypeEditParts` an, sodass sie eine direkte Verbindung zwischen den Elementen bilden. Zyklusrelationen docken dagegen an der linken Seite an. Damit ihr Verlauf, gerade bei selbstbeziehenden Object Types, besser sichtbar wird sind sie horizontal leicht versetzt von der Objektkante dargestellt (wie die Beziehung zwischen „ObjectType2“ und „ObjectType1“ in Abbildung 42). Falls eine Relation Teil eines nicht aufgelösten Zyklus ist, erscheint sie im Editor in roter Farbe um den

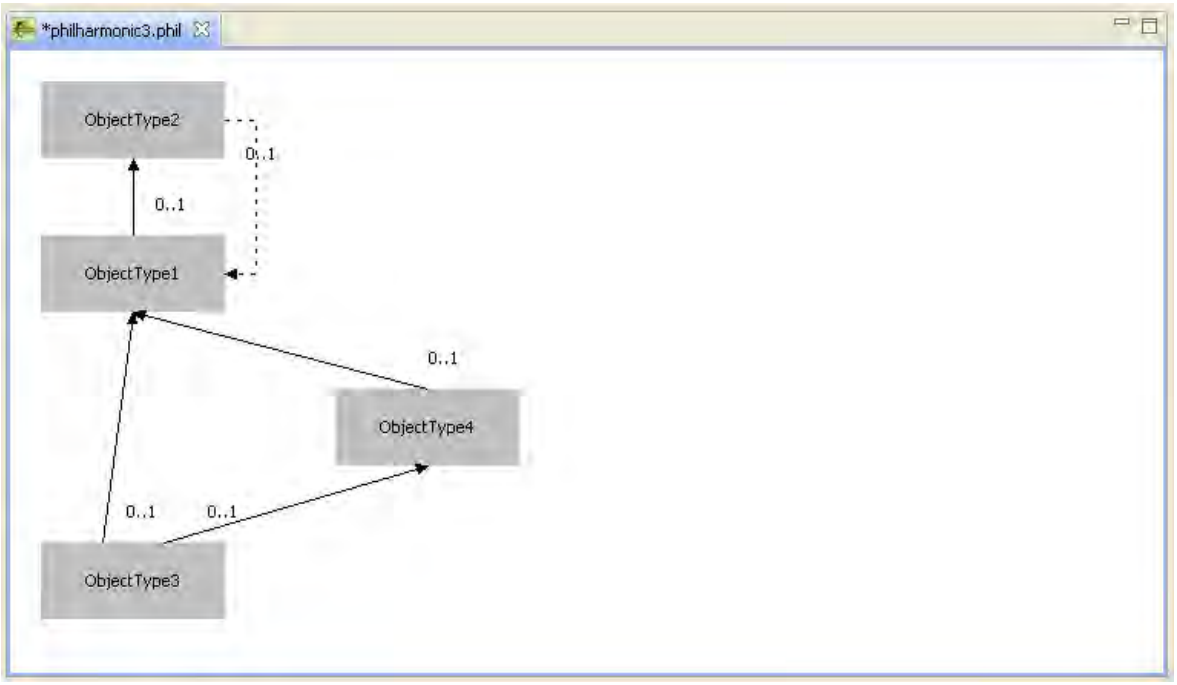

Abbildung 42.: Darstellung von Relationen im Objekttypen-Editor

nicht gelösten Fehler zu verdeutlichen, ansonsten in schwarz. Die Kardinalität der Relation wird für eine einfache Erkennung durch zwei Ziffern neben ihrem Ursprung angezeigt.
Im Eigenschaftsfenster existiert bei einem `RelationTypeEditPart` nur die *General* Kategorie. Diese stellt die Eigenschaften der Relation in fünf Zeilen dar. Zuoberst ist wie bei allen Objekten die Namenssektion angeordnet. Unterhalb befindet sich eine zweizeilige Sektion, die es erlaubt die Kardinalitäten der Relation zu bestimmen. Bei der Eingabe wird überprüft, dass die maximale Kardinalität nie kleiner als die minimale sein kann und dies gegebenenfalls automatisch korrigiert um Fehler auszuschließen. Unter dieser befinden sich noch zwei Sektionen für die Anzeige des Ursprungs und des Ziels der Relation.

Anordnung im Objektypeneditor

Die vertikale Anordnung der Objekttypen im Editor ergibt sich aus den berechneten Ebenen. Ausgangspunkt der Ebenenberechnung sind dabei die Objekttypen ohne Elternrelationen. Dabei werden Zyklusrelationen nicht berücksichtigt. Die horizontale Anordnung der Elemente der ersten Ebene erfolgt nach dem Zeitpunkt der Erzeugung des Modellobjekts. Die Anordnung erfolgt in post-order Traversierung. Für eine schönere Darstellung wird versucht das übergeordnete Element immer mittig über den untergeordneten Objekttypen anzuordnen. Objekttypen

die mehr als ein übergeordnetes Element besitzen werden dabei anhand des ersten auftretenden übergeordneten Objekttyps angeordnet. Bei erneutem Auftreten werden sie in der Anordnung ignoriert.

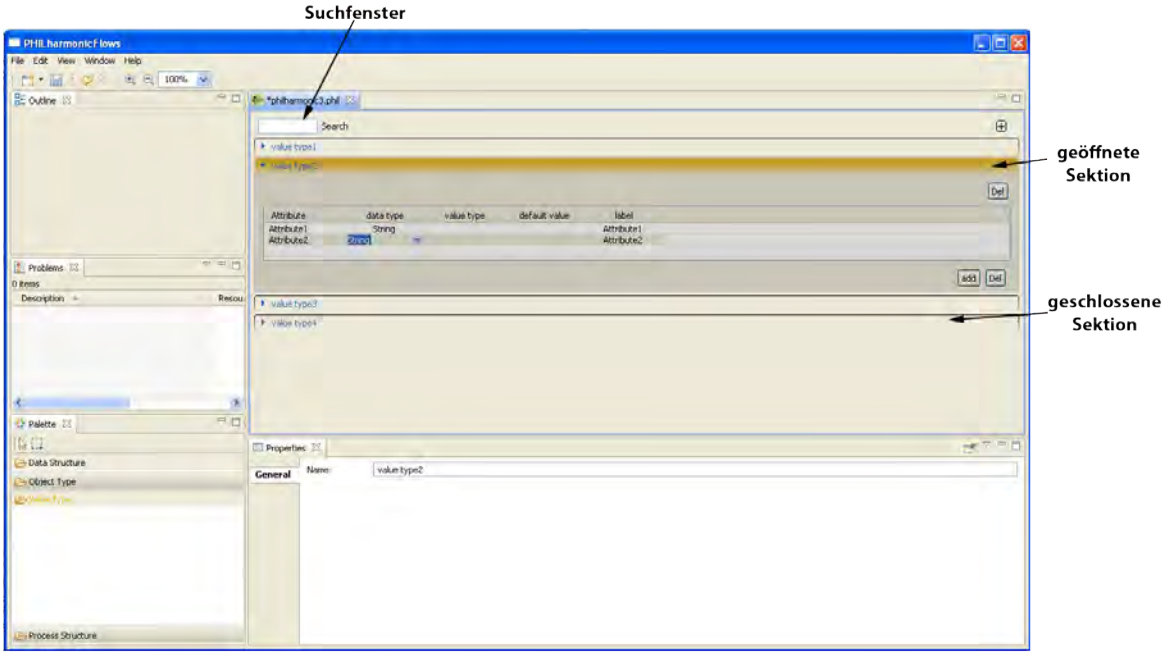

Abbildung 43.: Ansicht der Wertetypen-Editor Perspektive

5.3.4. Wertetypen-Editor

Der Wertetypen-Editor, zur Veranschaulichung in Abbildung 43 dargestellt, basiert im Gegensatz zum Objekttypen-Editor nicht auf dem GEF, sondern ist allein mithilfe des SWT implementiert. Die Bearbeitung der Wertetypen erfolgt ausschließlich über Schaltflächen. Deshalb wird in der Palette für den Editor kein Werkzeug bereitgestellt. Die Perspektive ist ebenso wie für den Object Type Editor aufgebaut, wobei in der `Outline` nur ein grauer Hintergrund zu sehen ist. Die Wertetypen werden innerhalb von `Sections` dargestellt. Dies sind graphische Objekte, die eine immer sichtbare Titelzeile besitzen. Zusätzlich dazu besitzen sie eine ausklappbare Fläche für weitere Inhalte, auf denen SWT-Objekte platziert werden können. In der Implementierung sind die Einzelheiten der Wertetypen in diesem ausklappbaren Bereich platziert, der in Abbildung 43 in einer Beispielsektion dunkelgrau hinterlegt ist. Die Einzelheiten der Wertetypen werden in der Tabelle dargestellt. Bei der Auswahl einer `Section` durch den Be-

nutzer wird die zuvor geöffnete geschlossen um die Übersicht zu erleichtern. Zusätzlich wird die ausgewählte `Section` optisch durch eine farbig hinterlegte Titelzeile hervorgehoben.
Für eine einfachere Durchsuchbarkeit der Liste der `Sections` gibt es eine Suchfunktion, die alle nicht den Suchbegriff enthaltenden `Sections` ausblendet. Dies war nur über einen Umweg zu realisieren, da auch ausgeblendete SWT-Elemente Platz auf der Oberfläche beanspruchen. Somit war es nicht ausreichend, die `Sections` als unsichtbar zu setzen. Für die Umsetzung wurde in einer von `Sections` abgeleiteten Klasse die `computeSize` Methode überschrieben, damit immer korrekte Größenangaben für den Layoutmanager der Oberfläche zurückgeliefert werden und sich keine Lücken zwischen den angezeigten `Sections` bilden.

5.3.5. Details des Mikroprozess-Editors

Der Mikroprozess Editor [siehe Abbildung 44] ist ein weiterer auf dem GEF basierender Editor. Auf der Oberfläche ist zusätzlich zu den bisher bekannten Komponenten ein weiteres Fenster oberhalb des Editors zu sehen, der Strukturkompass. Der Strukturkompass besitzt ebenfalls eine

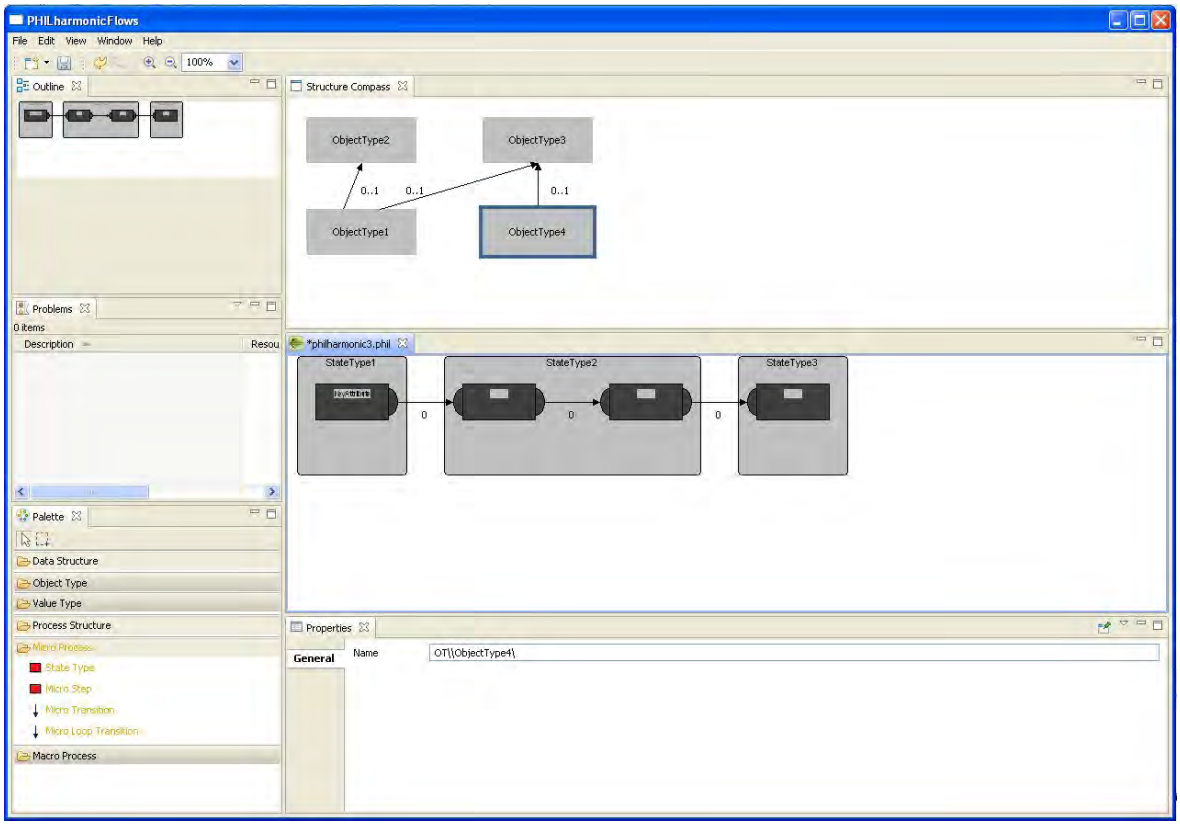

Abbildung 44.: Ansicht der Mikroprozess-Editor Perspektive

graphenbasierte Darstellung und zeigt die im Objekttypen-Editor erstellte Datenstruktur an. Aus der angezeigten Datenstruktur wird vom Benutzer der Objekttyp ausgewählt, für den der Mikroprozess im Editorfenster bearbeitet werden soll. Da Eclipse nur ein Editorfenster unterstützt, ist der Strukturkompass innerhalb einer View realisiert, die die Anzeige von GEF-Elementen unterstützt. Um Änderungen der Datenstruktur über den Strukturkompass zu verhindern, werden die EditParts des Objekttypen-Editors für die Verwendung abgeleitet und dabei die `createEditPolicies` Methode leer überschrieben. Dies verhindert Editieroperationen, da somit für die im Strukturkompass angezeigten Objekte keine Editieroperationen definiert sind. Bei der Anzeige des Strukturkompasses muss beachtet werden, dass der darzustellende Input des Strukturkompasses beim Schließen und Öffnen eines neuen Datenmodells ebenfalls erneuert wird, da dieser nicht von den automatischen Routinen mit dem neuen Modell versorgt wird. Der Editor zeigt immer den Mikroprozess zu dem gerade im Strukturkompass ausgewählten Objekttyp an und erlaubt dessen Bearbeitung. Für die Bearbeitung stellt die Palette wieder wie beim Objekttypen-Editor die verfügbaren Modellierungselemente bereit.

MicroProcessTypeEditPart Als graphische Repräsentation für den jeweils gewählten Mikroprozess verwaltet der `MicroProcessTypeEditPart` die Zeichenfläche. Diese zeigt zu Beginn den minimalen Mikroprozess, der erweitert werden kann. Auf der Zeichenfläche können dabei nur Elemente des Typs `StateTypeEditPart` platziert werden, die die Zustände des Mikroprozesses darstellen. Die verschiedenen Level werden dabei in horizontaler Richtung nacheinander angeordnet.

StateTypeEditPart Das `StateTypeEditPart` übernimmt die graphische Verwaltung der `StateType` Klasse, die die Zustände des Mikroprozesses beschreibt. Im Editor werden sie durch ein Rechteck mit abgerundeten Kanten dargestellt. Zur besseren Unterscheidung für den Benutzer wird der Name des Zustands mittig am oberen Rand des Zustandes angezeigt. Innerhalb der Zustände können beliebig viele `MicroStepEditParts` platziert werden. Dabei ist es erlaubt jeden Zustand mit Ausnahme des Startzustands jederzeit zu löschen, unabhängig mit welchen Kindelementen er belegt ist. Die Kindelemente werden ebenfalls mitgelöscht.

MicroStepEditPart Das `MicroStepEditPart` bildet den Controller für einen Mikroschritt. Daher besitzt es eine `NodeEditPolicy`, die es als Knoten ausweist, an den eine Verbindung zu einem anderen Knoten angedockt werden kann, um den Prozessfluss abzubilden. Die optische Darstellung im Editor erfolgt als Rechteck das für jeden Schritt dieselbe Breite besitzt. Die Höhe der einzelnen Mirkoschritte ist abhängig von ihrem anzuzeigenden Inhalt. Zur optischen Unterscheidung der verschiedenen Mikroschritt-Typen und der möglichen Verwendung dienen Halbkugeln an den Seiten, an denen auch die Transitionen angedockt werden. Bei einem Start-Mikroschritt wird der Halbkreis nur an der rechten Seite angezeigt, da dieser nur ausgehende

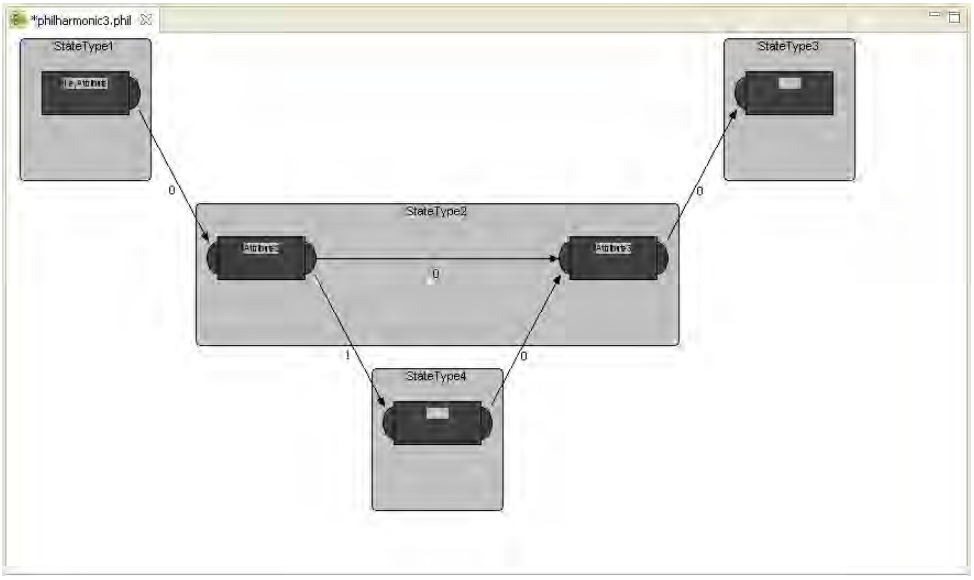

Abbildung 45.: Darstellung eines Mikroprozesses

Transitionen besitzen darf. Der Start-Mikroschritt ist damit der einzige Mikroschritt, der keinen Halbkreis auf der linken Seite besitzt. Bei allen Steps, die als End-Mikroschritt des Prozesses infrage kommen, existiert nur die Halbkugel auf der linken Seite, solange keine ausgehende Verbindung angelegt ist. Alle sonstigen Figuren besitzen die Halbkreise auf beiden Seiten. Dieses System bietet dem Benutzer eine einfache optische Hilfestellung bei der Konstruktion des Prozesses und erleichtert die Auffindung von toten Punkten. Die visuelle Darstellung der Halbkreise erfolgt dabei durch zum Teil unter das Rechteck geschobene Kreise. Die Zuweisung einer Referenz zu einem Mikroschritt erfolgt über die Anwahl der hellgrauen Titelzeile des Rechtecks durch einen Mausklick. Da eine direkte Anzeige einer Auswahlliste direkt im Editor an

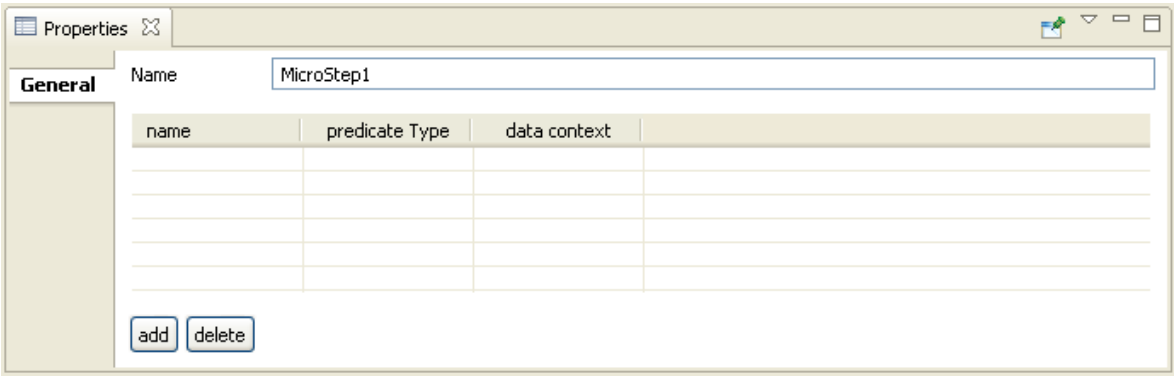

Abbildung 46.: Eigenschaftsfenster mit Mikrowertschritt-Tabelle

den eingeschränkten Möglichkeiten von Draw2d scheitert, welches keine Comboboxen oder anwählbare Listen vorsieht, ist die Auswahl in einen Dialog ausgelagert. Das sich öffnende Dialogfenster zeigt die möglichen Referenzen dabei in einer durchsuchbaren Auswahlliste [siehe Abbildung 17]. Der Name des ausgewählte Attributs wird anschließend in der Figur innerhalb der Titelzeile angezeigt [siehe Abbildung 45]. Dem `MicroStepEditPart` können `MicroValueStepEditParts`, die die Mikrowertschritte darstellen, als Kindelemente zugewiesen werden. Deren Erzeugung erfolgt jedoch nicht über die Auswahl eines Werkzeugs in der Palette und der Platzierung im Editor, sondern sie werden direkt über das in Abbildung 46 zu sehende Eigenschaftsfenster für den gerade ausgewählten Mikroschritt erzeugt und gelöscht. In der Tabelle werden die bereits erstellten Mikrowert-Schritte und ihre zugehörigen Filter angezeigt. Die Filter können durch Anklicken der Spalte angepasst werden.

MicroValueStepEditPart Die dem zugrundeliegenden Mikroschritt zugewiesenen Mikrowert-Schritte werden durch die Klasse `MicroValueStepEditPart` verwaltet und gezeichnet. Dieses agiert durch die Zuweisung der `NodeEditPolicy` ebenfalls als Knoten für Transitionen. Im Editor werden die erzeugten Figuren dabei innerhalb der Micro Steps als untereinander angeordnete Rechtecke visualisiert, die mit den Filterwerten bezeichnet sind. Falls die anzuzeigende Fläche für die Darstellung nicht ausreicht wird der komplette Filterwert in einem Tooltip angezeigt.

MicroTransitionEditPart Diese steuern die Darstellung der Mikrotransitionen. Die Anzeige im Editor erfolgt dabei als Pfeil in der Flussrichtung des Graphen. Die Anzeige der Priorität der Transition erfolgt unterhalb der Transition mithilfe eines `MidpointLocator`, der die Priorität damit immer mittig zwischen den beiden Enden der Transition platziert. Im Eigenschaftsfenster der Transitionen gibt es die Möglichkeit deren Priorität mittels einer Combobox abzuändern. In dieser werden möglichen Prioritäten angeboten, die der Transition zugewiesen werden kön-

Abbildung 47.: Eigenschaftsfenster einer Mikrotransition

nen. Zusätzlich ist es hier bei externen Mikrotransitionen möglich das Ausführungsverhalten auszuwählen. Diese Auswahl ist auch über das Kontextmenu der Transition möglich. Um das Ausführungsverhalten auch im Editor auf den ersten Blick erkenntlich zu machen, werden die expliziten Transitionen zusätzlich mit einer gepunkteten Pfeillinie dargestellt.

Anordnung der Mikroschritte im Editor

Die Anordnung der Komponenten des Mikroprozesses muss mehrere Bedingungen berücksichtigen. Zunächst besitzen die Mikroschritte Level Werte, die ihre Abfolge und somit die horizontalen Spalten in denen sie angezeigt werden müssen beschreiben. Da alle Mikroschritte im Editor mit einer festen Breite dargestellt werden, ergibt sich in der Breite über alle Level ein einheitliches Raster. Eine nur dem Level folgende Anordnung, wie in Abbildung 48 beispielhaft erfolgt, ist jedoch nicht sinnvoll, da die Zuordnung der Mikroschritte zu Zuständen so optisch nur indirekt ersichtlich und damit schlecht erkennbar ist.

Abbildung 48.: Lineare Anordnung von Mikroschritten

Für eine bessere Übersicht werden die zusammengehörigen Mikroschritte daher innerhalb eines einen Zustand darstellenden Rechtecks dargestellt [siehe Abbildung 49]. Für die Anordnung werden die zusammengehörenden Mikroschritte zuerst Zustands-lokal - ohne Berücksichtigung von Mikroschritten anderer Zustände - anhand ihres Levels angeordnet. Da Zustände

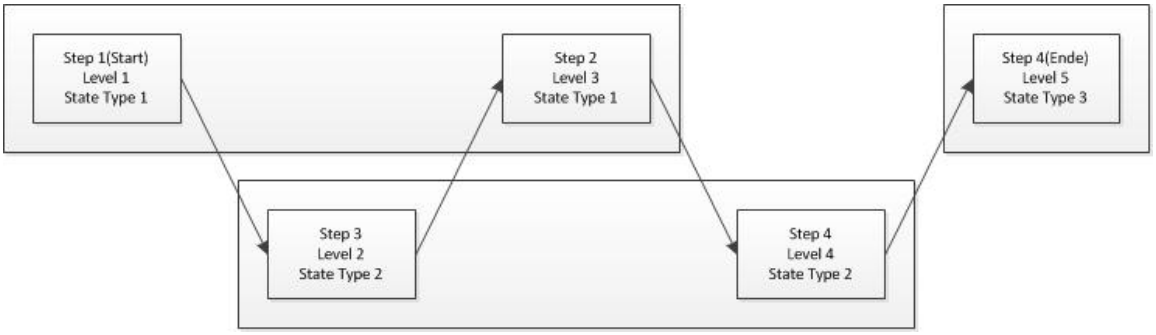

Abbildung 49.: Anordnung der Mikroschritte, gegliedert in Zustände

keinen von einem einzigen Startpunkt ausgehenden durchgehenden Graphen bilden müssen, ergibt sich bei der Traversierung innerhalb eines Zustandes das Problem des Startpunkts der Anordnung um alle Mikroschritte zu berücksichtigen. Dies wird gelöst, indem die Traversierung bei allen Mikroschritten des Zustands mit eingehenden externen Transitionen beginnt. In vertikaler Richtung ist kein festes Zeilenraster möglich, da Mikroschritte, abhängig davon ob sie Mikrowertschritte besitzen, mit unterschiedlichen Höhen dargestellt werden müssen. Daher erfolgt die Berechnung in post-order Traversierung. Die maximale Höhe eines Mikroschritts gibt dann die Höhe des aktuellen Pfades vor. Anschließend werden die so berechneten Zustände ebenfalls auf der Zeichenfläche angeordnet. Die Anordnung erfolgt ebenfalls in post-order Traversierung analog zur Anordnung der Mikroschritte. Die zu berücksichtigenden Pfade sind dabei die in den Zuständen separat gespeicherten externen Transitionen.

```
1  //Funktion zum Anordnen der Mikroschritte eines Zustands
2  FUNCTION ordneStateTypeAn
3  INPUT: -
4  OUTPUT: -
5  BEGIN
6  FOR Ziele eingehender externe Transitionen
7      IF Ziel bereits platziert THEN
8      //Falls der Mikroschritt bereits zuvor durch eine andere
       //externe Transition platziert worden ist
9          continue
10     ELSE
11         ordneMicroStepAn(Ziel)
12     END IF
13 END FUNCTION
14
15 //Rekursive Funktion zum Platzieren der Mikroschritte
16 FUNCTION ordneMicroStepAn(MicroStep microStep)
17 INPUT: MicroStep microStep (der anzuordnende Mikroschritt)
18 OUTPUT: -
19 BEGIN
20     IF microStep hat interne ausgehende Transition THEN
21         //Falls ein interner Nachfolger existert, wird dieser
           //zuerst platziert, da post-order Traversierung
```

```
22        ordneMicroStepAn(Ziel ausgehender Transition)
23    ELSE
24        platziere microStep
25    END IF
26 END FUNCTION
```

Beispielcode 7.: Pseudocode für Anordnung der Mikroschritte

6
Diskussion

6.1. Erweiterungsmöglichkeiten des Frameworks

Das Framework stellt die Grundfunktionen für den Entwurf eines PHILharmonicFlows Projekts bereit und kann um weitere Funktionen ergänzt werden. Um dies zu unterstützen wurde versucht das System möglichst modular zu implementieren, sodass bei Erweiterungen möglichst wenig am bisherigen Code verändert werden muss. Als Grundlage für mögliche Erweiterungen werden hier die Schnittstellen zum aktuellen System offengelegt und die Ausgangspunkte für die weitere Entwicklung bereitgestellt.

6.1.1. Entwurf eines zusätzlichen Editors

Das Framework kann um zusätzliche Editoren erweitert werden. Graphenbasierte Editoren sollten dabei die bereits zuvor beschriebene abstrakte Klasse `AbstractGraphicalEditorPage` erweitern, während konventionelle Editoren auf der `AbstractEditorPage` Klasse aufbauen sollten [siehe 5.3].

Für einen neuen Editor muss zusätzlich eine neue PaletteDrawer Schaltfläche in der Palette angelegt werden, die den Wechsel auf die neu erstellte Editorseite ermöglicht. Mit dieser werden ebenfalls die Modellierungswerkzeuge für GEF-Editoren bereitgestellt. Die Anlage eines PaletteDrawer erfolgt in der `PHILharmonicFlowsEditorPaletteFactory` innerhalb der `createPalette()` Methode. Hierbei muss zusätzlich die zweistufige Verwaltung der Palette beachtet werden. Die erste Stufe bilden die Hauptkomponenten, die die Einträge der Palette in Kategorien untergliedern und die Zweite die Editorkomponenten, die als Wählschalter für einen Editor dienen und dessen Werkzeuge bereitstellen. Diese Gliederung ist in Abbildung 50 anhand einer Beispielansicht dargestellt. Hauptkomponenten müssen der Liste `mainPaletteDrawers` hinzugefügt werden um eine problemlose Verwaltung der Palette zu ermöglichen. Für die einzelnen Editorkomponenten werden im Gegensatz dazu jeweils statische öffentliche Variablen angelegt. Dies ist notwendig, um das Umschalten der Editoren zu realisieren, da abgefragt werden muss welcher Palettenreiter gerade ausgewählt worden ist.
Der Editor muss anschließend im `PHILharmonicsFlowsEditor` als weitere Editorseite bekannt gemacht werden. Dies erfolgt durch eine Abänderung der `createPages()` Methode des `PHILharmonicsFlowsEditor`. In dieser wird durch den Aufruf von `addPage` mit der neuen Editorseitee als Parameter die neue Editorseite hinzugefügt. Die von `addPage` zurückgegebene Seiten-ID muss in einer privaten Variablen gespeichert werden um sie für spätere Aufrufe verfügbar zu haben. Anschlie-

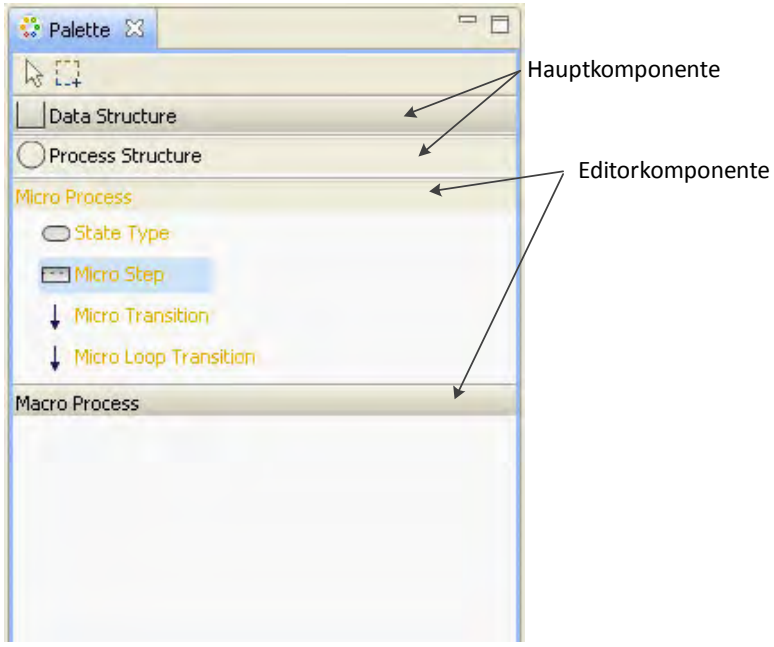

Abbildung 50.: Navigationskonzept der Palette

ßend muss noch mittels der Methode `setPageText` ein Seitenname festgelegt werden. Um die Auswahl des Editors über die Palette zu ermöglichen ist eine Änderung innerhalb der internen `CustomPalettePage` Klasse des `PHILharmonicFlowsEditor` notwendig. In der `createControl` Methode der `CustomPalettePage` muss dafür der SelectionChanged-Listener des paletteViewers ergänzt werden. Dies erfolgt genauer in der `selectActivePage` Methode durch Hinzufügen einer Vergleichsoperation des aktiven Paletteneintrags mit dem der neu erstellten Editorseite.

6.1.2. Implementierung zusätzlicher Modell-Klassen

Für zusätzliche Modell-Klassen sollten die bereits in Kapitel 5.1.2 beschriebenen jeweiligen abstrakten Klassen erweitert werden um eine einheitliche Struktur beizubehalten. Für Erzeugungs-, Änderungs- und Löschoperationen sollten Kommando-Klassen zwischengeschaltet werden, um eine einheitliche Vorgehensweise von Rückgängig- und Wiederholoperationen anzubieten und somit ein konsistentes, einheitliches Verhalten im gesamten Ablauf zu erhalten und den Benutzer nicht zu verwirren. Die Benachrichtigung über Änderungen an den bisher implementierten Klassen erfolgt nach dem Observer Entwurfsmuster über registrierte Listener und unterstützt somit die problemlose Erweiterung. An Informationen interessierte Instanzen müssen sich somit nur als Listener registrieren um über die Änderungen informiert zu werden.

6.1.3. Integration des Eigenschaftsfensters

Das Eigenschaftsfenster erhält das aktuell ausgewählte Objekt über ein `SelectionChangedEvent`. Dieses wird bei GEF-Editoren automatisch abgesendet, bei den anderen Editoren muss das Versenden im Programmcode explizit erfolgen. Dafür ist es notwendig, dass der Editor den ISelectionProvider implementiert und Änderungen der Auswahl an die registrierten SelectionChangedListener versendet.
Als Anzeigefenster für die Eigenschaften ist im `PHILharmonicsFlows`Editor eine TabbedPropertiesView eingestellt. Diese besteht wie bereits zuvor erwähnt aus Kategorien, die aus Sections aufgebaut sind. Die Definition der Kategorien und ihres jeweiligen Aufbaus erfolgt dabei flexibel in der Projektbeschreibung in der zugehörigen plugin.xml Datei. Für die Änderung muss dabei nicht die xml Datei direkt geändert werden, sondern Eclipse bietet dafür eine angepasste Oberfläche, die in Abbildung 51 dargestellt ist. Dabei sind auf der linken Hälfte der Baum mit den Erweiterungspunkten und auf der rechten Hälfte die Eigenschaften des gerade gewählten Eintrags zu sehen. Die für das Eigenschaftsfenster zuständigen Punkte sind in

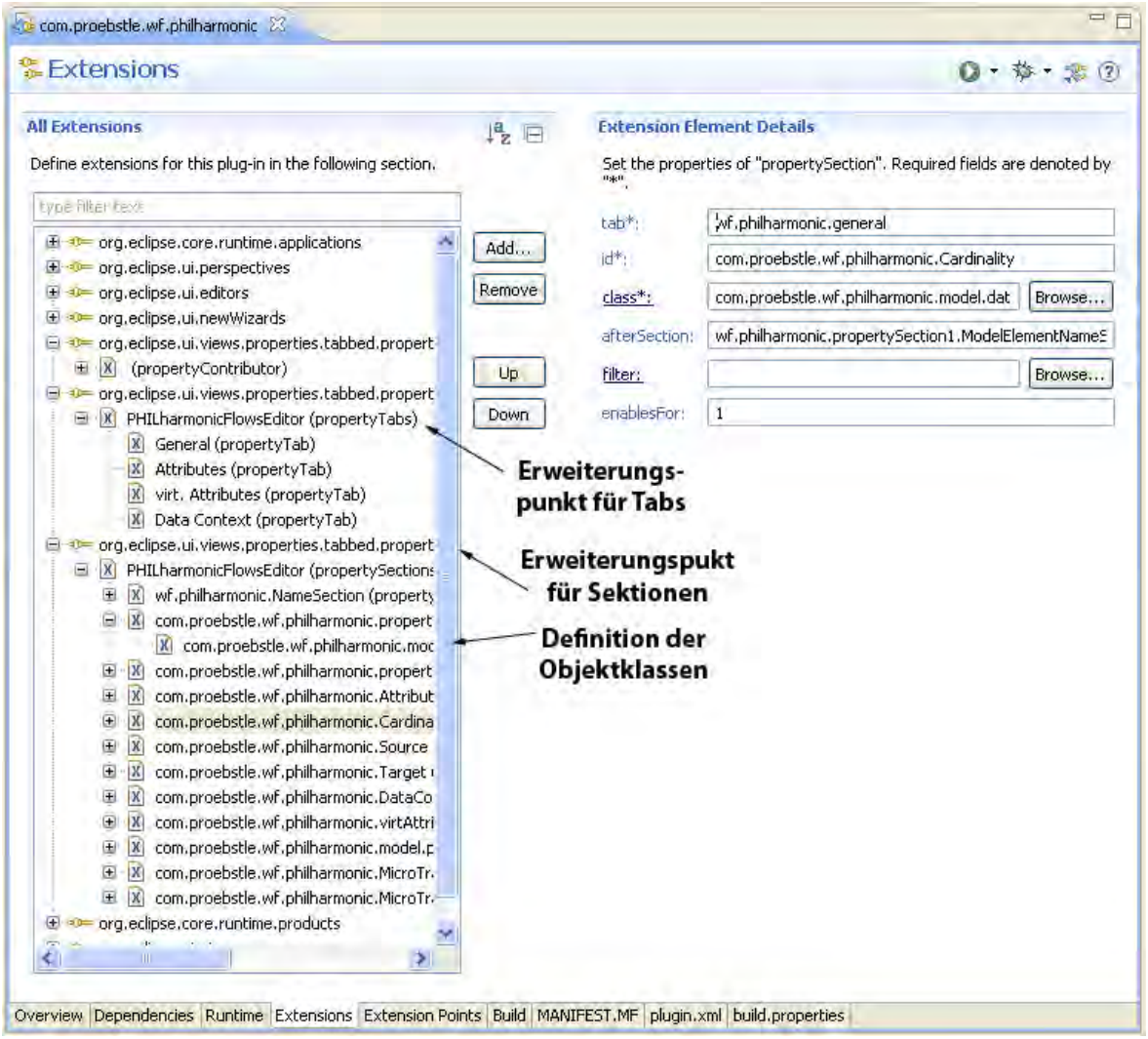

Abbildung 51.: Erweiterungspunkte für das Eigenschaftsfenster in der plugin.xml

der Abbildung für eine bessere Übersicht expandiert. Im Erweiterungspunkt `org.eclipse.ui.views.properties.tabbed.propertyTabs` werden die Tabs für die Kategorien des Erweiterungsfensters definiert. Im Bild sind hier die bereits definierten Tabs General, Attributes, virt. Attributes und DataContext zu sehen, die im PHILharmonicFlowsEditor angelegt sind. Im Erweiterungspunkt `org.eclipse.ui.views.properties.tabbed.propertySections` sind die einzelnen Sektionen mit dem Editor verknüpft. In den Detaileinstellungen in der rechten Bildschirmhälfte werden sie einem Tab zugeordnet und die angezeigte Reihenfolge festgelegt. Jede Sektion besitzt dabei noch die im Beispiel gezeigten Kindelemente, in denen definiert wird, für welche ausgewählten Objektklassen sie im Eigenschaftsfenster angezeigt werden soll.

6.2. Abwandlungen zum Oberflächenkonzept

Gegenüber dem ursprünglichen Oberflächenkonzept wurden einige Änderungen vorgenommen. Diese sind zum einen durch Änderungen am PHILharmonicsFlows Konzept oder praktischen Erwägungen, die sich bei der Umsetzung ergeben haben, bedingt. Andere sind durch die Eigenheiten des Eclipse RCP und allgemeine Darstellungsprobleme ausgelöst. Im Folgenden soll eine kurze Übersicht über einige Änderungen und deren Hintergründe gegeben werden.

6.2.1. Aufspaltung der Navigationsleiste

Die Navigationsleiste besteht im Konzept wie in Abbildung 52 links zu sehen aus einem zusammenhängenden Block, der verschiedene Aufgaben erfüllt. Die Position der einzelnen Unterfenster ist dabei abhängig von der ausgewählten Komponente verschoben, da die Reiter fest untereinander angeordnet sind. In der Implementierung (Abbildung 52 rechts) wurde die Navigationsleiste in die drei eigenständige Teilbereiche aufgespalten, die sich aus dem Konzept ab-

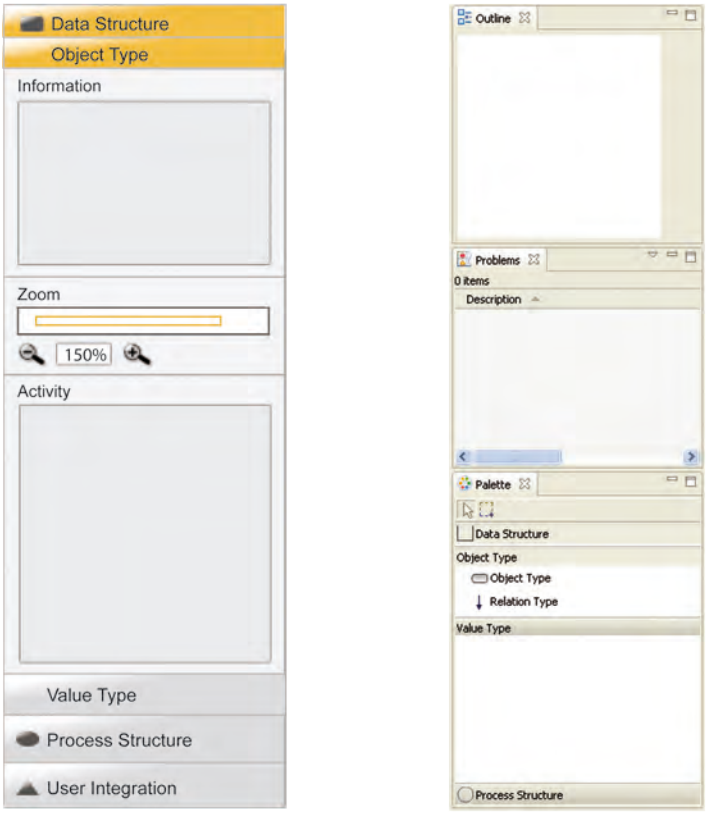

Abbildung 52.: Navigationsleiste im Konzept (links) und in der Implementierung (rechts)

leiten lassen. Dies sind die Palette für die Auswahl der Komponenten und die Bereitstellung der Werkzeuge, die Outline für das Übersichtsfenster und das „Problems“-Fenster für Fehlerhinweise. Vereinfacht wird diese Aufteilung durch das Eclipse Konzept, das die Palette, die Outline und das „Problems“-Fenster bereits als eigenständige Einheiten kennt. Darüber hinaus bietet es dem Benutzer die Möglichkeit einer freien Platzierung und Größenanordnung der einzelnen Teilbereiche. Außerdem findet sich dadurch die Position des Übersichtsfensters und des Informationsbereichs unabhängig von einem Reiterwechsel immer an der selben Position was die Übersicht begünstigt. Im Oberflächenentwurf war als Teil eines durchgehenden Farbkonzepts ebenfalls eine gelbe Hinterlegung der Titelleiste der ausgewählten Komponente vorgesehen. Dies ist jedoch wie bereits in Kapitel 5.3.1 erklärt nicht möglich, da Eclipse die Gestaltung des ausgewählten Eintrags in privaten Methoden vornimmt und die Klasse für Änderungen und Erweiterungen sperrt. Als Ersatz dafür ist die Schrift des jeweils aktiven Eintrags hervorgehoben.

6.2.2. Darstellung der Datenstruktur

Abbildung 53 zeigt die Anregungen zur Positionierung der Objekttypen im Editor und der Wegfindung der Relationen zwischen den Objekttypen aus dem Konzept. Bei der Umsetzung in die Praxis zeigten sich dabei aber mehrere Probleme und Widersprüche. Das hauptsächliche Problem betrifft dabei die rechtwinklige Linienführung im Manhattan Stil. Da es kaum möglich ist die Linien nicht komplett überlagerungsfrei zu positionieren, tritt das in Abbildung 55 zu sehende Problem auf, dass sich die Relationen nicht mehr auseinanderhalten lassen und somit Ursprung und Ziel nicht mehr erkennbar sind. Ein weiteres Problem ist die häufige Kollision der rechtwinkligen Linien mit Objekttypen, die dadurch wie ebenfalls in Abbildung 55 dargestellt von den Linien durchschnitten werden. Diese beiden Probleme könnten durch eine Kollisionserkennung vermieden werden, die jedoch aufwendig implementiert werden müsste, mehrere zusätzliche Knicke in den Linien schaffen würde und trotzdem Kompromisse bei der Überlagerung eingehen müsste.
Aufgrund dieser Probleme wurde bei der Implementierung eine direkte Wegfindung gewählt, die jedoch versucht Kollisionen mit anderen Objekttypen zu vermeiden. Diese Wegfindung ist zum Vergleich in Abbildung 54 mit der selben Datenstruktur wie die Konzeptdarstellung in Abbildung 53 abgebildet.
Die zweite Änderung gegenüber dem Konzept betrifft die Anordnung der jeweils übergeordneten Objekttypen, die nicht wie im Konzept am linken Rand sondern zentral über allen untergeordneten platziert sind. Dies ermöglicht eine Reduzierung der Wege der Relationen und ermöglicht damit in den meisten Fällen einen besseren Überblick.

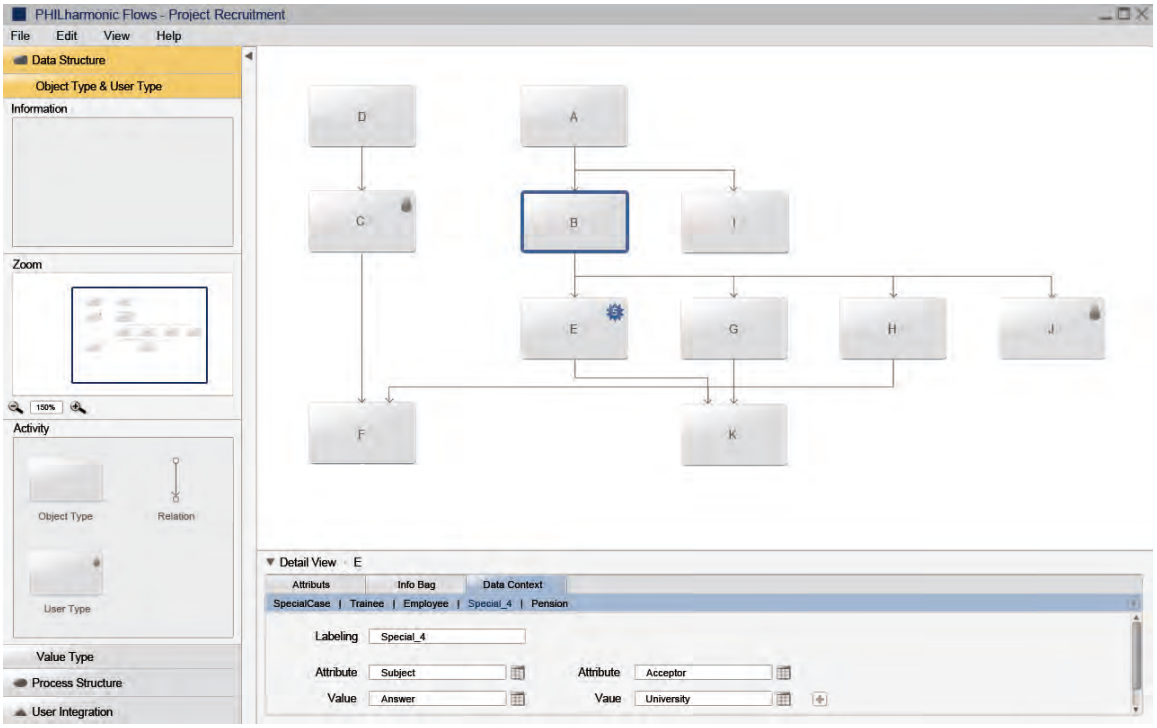

Abbildung 53.: Datenstruktur Konzept

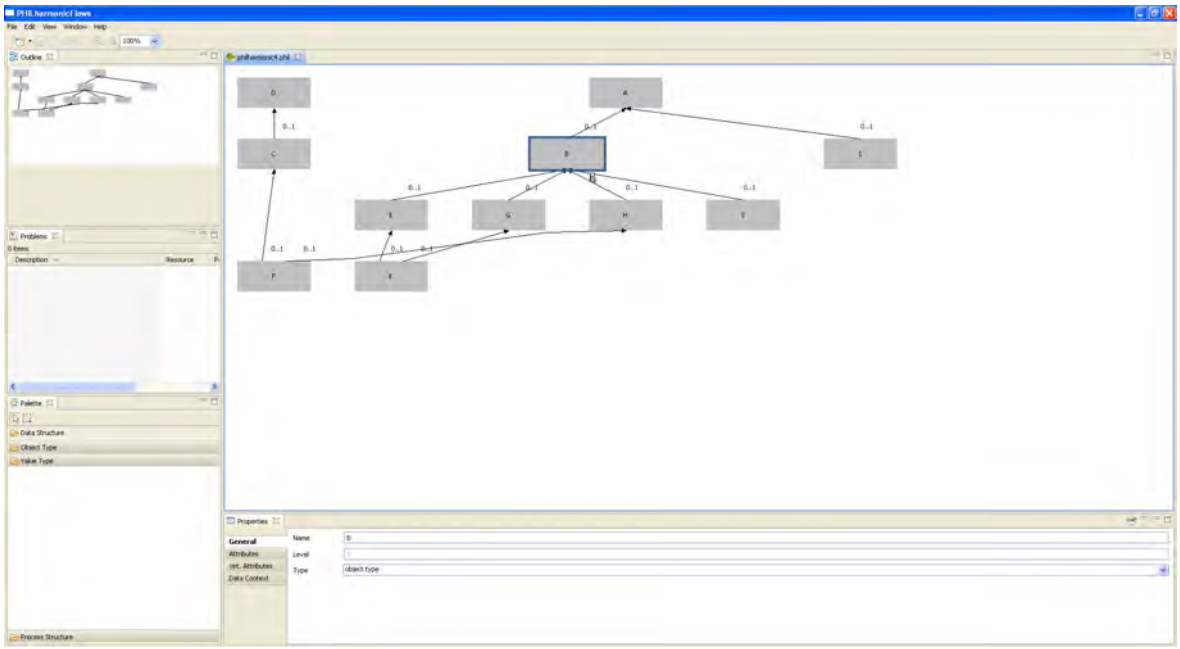

Abbildung 54.: Datenstruktur Implementierung

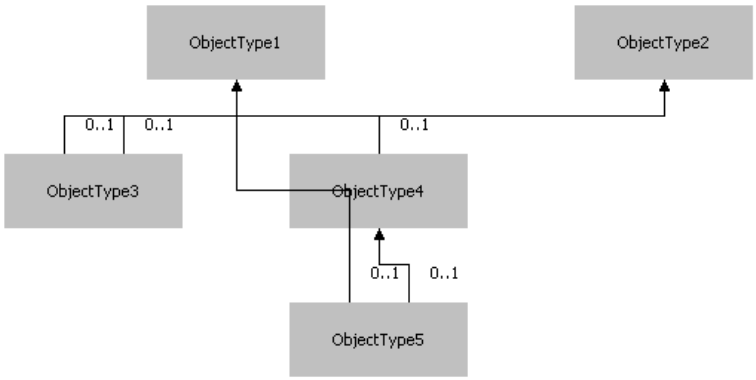

Abbildung 55.: Alternative Datenstruktur mit rechtwinkliger Linienführung (Manhattan Stil)

6.2.3. Abwandlungen des Eigenschaftsfensters

Das Eigenschaftsfenster [Abbildung 57] weist ebenfalls Unterschiede zu der Vorlage des Konzepts [siehe Abbildung 56] auf. Die Reiter zur Auswahl der Kategorie sind im Konzept vertikal am oberen Fensterrand positioniert. Dies verkürzt jedoch die im Fenster verfügbare Fläche für

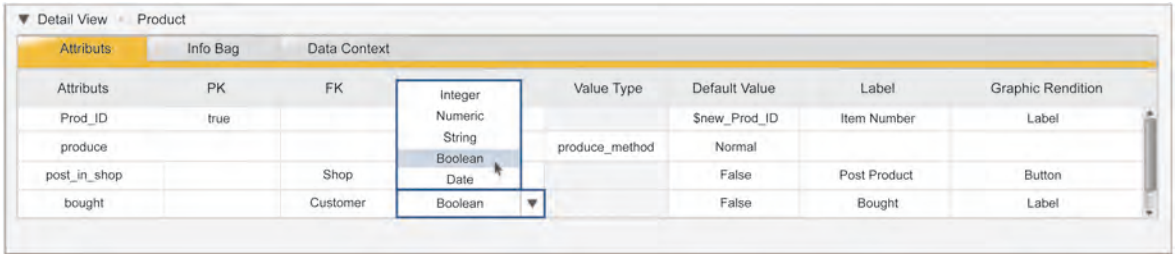

Abbildung 56.: Eigenschaftsfenster Konzept

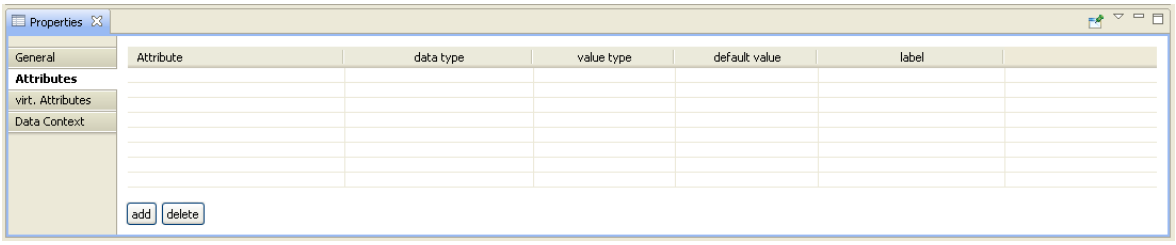

Abbildung 57.: Eigenschaftsfenster Implementierung

die Darstellung der Inhalte. Da sich jedoch beispielsweise bei Attributen unter Umständen recht lange Tabellen ergeben führt dies zu einer schlechteren Übersicht und einem häufigen Scrollen. Da zusätzlich die verfügbare Breite des Bildschirms nicht ausgenutzt wird und sich diese voraussichtlich durch die zunehmende Verbreitung von breitformatigen Bildschirmen noch zusätzlich erhöht, wurden die Tabs stattdessen am linken Rand untereinander angeordnet. Dies maximiert die für Inhalte nutzbare Höhe des Fensters.
Bei der farbigen Hervorhebung des gewählten Reiters musste von den Konzeptfarben abgewichen werden, da Eclipse hierauf ebenfalls keine Zugriffsmöglichkeiten bietet. Somit ist die nicht änderbare Auswahlfarbe weiß beibehalten und die Schrift des ausgewählten Reiters wird fett hervorgehoben.
Inhaltlich sind die Kategorien um einen zusätzlichen Reiter für allgemeine Einstellungen ergänzt. Ebenso sind mehrere kleinere Ergänzungen wie Schaltflächen erfolgt, die zur Bedienung des Programms notwendig sind.

6.2.4. Unterschiede der Prozessstruktur

Abgesehen von den durch die veränderte Darstellung der Datenstruktur verursachten Unterschiede zeigen sich in der Darstellung nur kleinere Unterschiede zum Konzept (vgl. Abbildung 59 und Abbildung 58). Eine Änderung betrifft dabei die Darstellung der Mikroschritte. Dabei gibt nun das Erscheinungsbild der Mikroschritte durch die unterschiedliche Darstellung der Halbkreise bereits Hinweise auf die aktuellen Möglichkeiten der Mikroschritte im Mikroprozesses. Dies wurde bereits in 5.3.5 genauer erklärt. Eine wichtigere Änderung betrifft die Konstruktion der Zustände. Laut Konzept war geplant, die Mikroschritte frei im Mikroprozess zu platzieren und durch anschließendes Umfahren mit einem Auswahlrahmen die Schritte zu einem Zustand zusammenzufassen. Dabei können sich jedoch Probleme ergeben, da die gewünschten zu umfahrenden Schritte nicht unbedingt nahe beieinander liegen sondern über den ganzen Mikroprozess verteilt sein können. Auch stellt sich die Frage, wie die nachträgliche Zuordnung von Mikroschritten erfolgen soll. Außerdem stellt dies den Benutzer vor die Aufgabe am Ende zu kontrollieren ob alle Mikroschritte einem Zu-

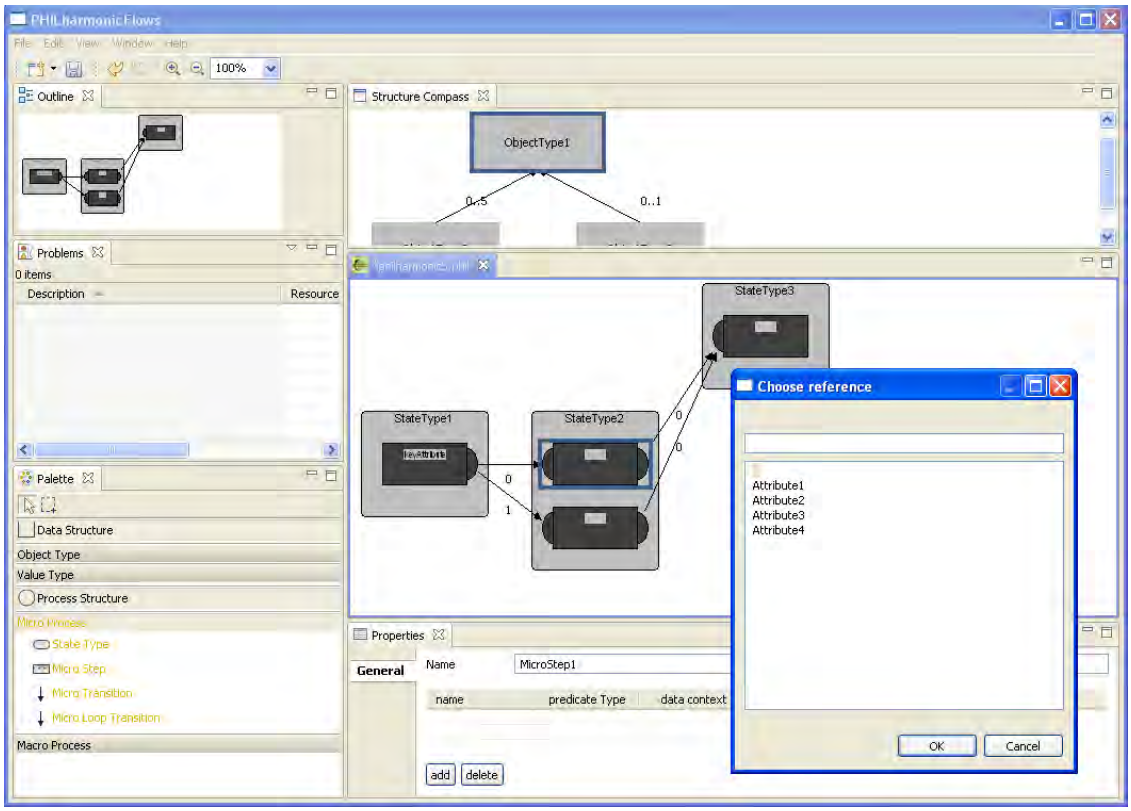

Abbildung 58.: Prozessstruktur Implementierung

stand zugeordnet wurden. Auch ergeben sich damit Probleme in der Verwaltung der zugehörigen Klassen, da somit Mikroschritte verwaltet werden müssen, die keinem Zustand angehören. In der Implementierung werden daher zuerst die Zustände erstellt, auf denen anschließend die Mikroschritte positioniert werden können. Dies stellt die korrekte Zuordnung der Mikroschritte zu den Zuständen bereits bei der Konstruktion sicher und erspart dem Benutzer nachträgliche Zuordnungen.

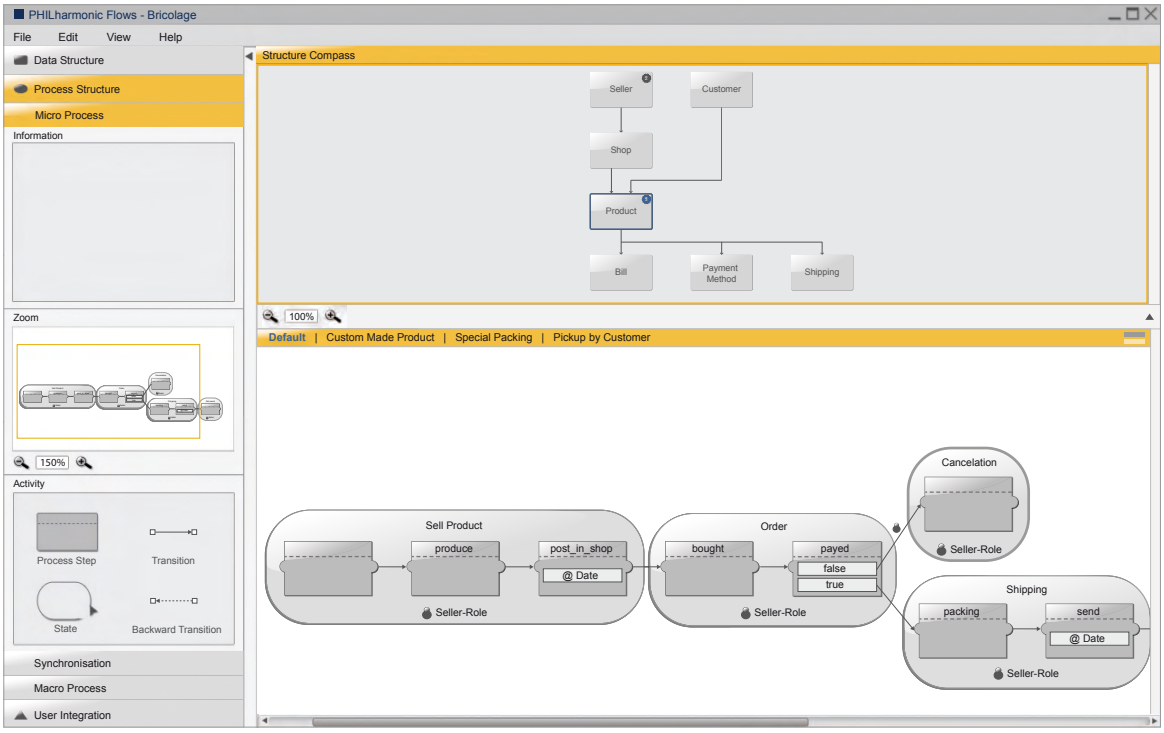

Abbildung 59.: Prozessstruktur Konzept

7
Zusammenfassung und Ausblick

7.1. Zusammenfassung

Die Aufgabe der Diplomarbeit bestand darin, einen Prototypen für die Modellierungsumgebung von PHILharmonicFlows auf der Basis der bisherigen Veröffentlichungen zum Modellierungsframework und des bestehenden Usability-Konzepts[17] zu entwickeln. Hierfür wurden zunächst die benötigten Klassen für die Datenverwaltung und ihre Zusammenhänge erarbeitet und mittels UML Klassendiagrammen eine Datenstruktur modelliert. Auf dieser Grundlage wurde die Implementierung der Modellierungskomponente mittels Eclipse RCP und Eclipse GEF begonnen.

Bei der Implementierung zeigte sich, dass die Eclipse Plattform nur eine begrenzte freie Gestaltungsmöglichkeit der Oberfläche erlaubt. Gerade eine Änderung der Farbe der Elemente, die keinen Einfluss auf die Abläufe hätte ist oft nicht möglich. Dies könnte in weiteren Eclipse Versionen verbessert werden. Eine der wichtigsten Verbesserungen der nächsten Version Eclipse 4 soll ja die Vereinfachung der Oberflächendarstellung unter anderem mittels Cascading Style Sheets sein. Insgesamt gesehen hat sich Eclipse RCP aber als eine gute Wahl für die Plattform

bestätigt, da es für viele benötigte Funktionalitäten Grundlagen bereitstellt und damit die Entwicklung vereinfacht. Auch zwingt es durch seinen Aufbau sich an bewährte Entwicklungsmuster zu halten und garantiert damit ein flexibleres Endprodukt.
Weitere Schwierigkeiten ergaben sich bei der graphischen Umsetzung der Modellierungseditoren. Dabei zeigte sich, dass eine dauerhafte Anordnung anhand der Level für einen Benutzer nicht sehr intuitiv ist, da die neu angelegten Objekte dadurch nicht an der eingefügten Stelle erscheinen, sondern immer auf dem niedersten Level. Außerdem erweist sich die Anordnung der einzelnen Objekte als sehr komplex, weil dabei die Level berücksichtigt werden müssen und die darzustellenden Strukturen durch mögliche Teilungen und Zusammenführungen des Prozessgraphen nicht wie bei blockstrukturierten Graphen allgemein ermittelbar sind.

7.2. Ausblick

Im Rahmen des PHILharmonicFlows Konzepts ist die Realisierung eines kompletten Frameworks für datenorientierte Prozesse geplant. Das vorgesehene Prozess-Management-System besteht aus einem Modellierungstool und einer Laufzeitkomponente. In einem ersten Schritt wurde nun mit der Entwicklung des Modellierungstools begonnen. Dieses besitzt bisher einen Editor für die Datenstruktur und den Mikroprozess. In einem weiteren Schritt ist die Erweiterung auf die Makroprozesse und die Rechtevergabe möglich.
Dabei gibt das im Rahmen dieser Diplomarbeit entstandene grundlegende Framework einen Ausgangspunkt, das die Möglichkeit einer Erweiterung vorsieht und diese durch allgemein bekannte Konzepte unterstützt. Somit sollte es möglich sein das Framework zu erweitern um einen vollständigen Editor für PHILharmonicFlows zu erhalten.

Literaturverzeichnis

[1] Trovarit: http://www.trovarit.com/veroeffentlichungen/presse/erste-ergebnisse-der-studie-business-process-management%94-fuer-die-mehrheit-ist-bpm-neuland..html, zuletzt besucht am 23.04.2011.

[2] Kurzlechner W.: Die BPM-Prognosen 2011 von Gartner, http://www.cio.de/strategien/2263262/. Zuletzt besucht am 23.04.2011.

[3] Eclipse: http://www.eclipse.org/org/. Zuletzt besucht am 20.04.2011.

[4] Eclipse Projects: http://www.eclipse.org/osgi/. Zuletzt besucht am 20.04.2011.

[5] Delpa S.: Understanding how Eclipse plug-ins work with OSGi, http://www.ibm.com/developerworks/opensource/library/os-ecl-osgi/index.html. Zuletzt besucht am 03.03.2011.

[6] Künzle, V., Reichert, M.: Towards Object-aware Process Management Systems: Issues, Challenges, Benefits. In: Proc. BPMDS'09. (2009) 197-210

[7] Künzle V., Reichert M.: Integrating Users in Object-aware Process Management Systems: Issues and Challenges. In: Proc. BPD'09. (2009) 29-41

[8] Künzle, V., Weber, B., Reichert, M.: Object-aware Business Processes: Fundamental Requirements and their Support in Existing Approaches. International Journal of Information System Modeling and Design 2(2) (2010)

[9] Künzle V., Reichert M.: PHILharmonicFlows: Towards a Framework for Object-aware Process Management. Journal of Software Maintenance and Evolution: Research and Practice (2011)

[10] Künzle V., Reichert M.: A Modeling Paradigm for Integrating Process and Data at the Micro Level. In: Proc. BPMDS'11, Springer (2011) *(accepted for publication)*

[11] Eclipse: The Eclipse Foundation open source community website. http://www.eclipse.org/. Zuletzt besucht am 14.04.2011.

[12] Eclipse: Eclipse Rich Client Platform. http://www.eclipse.org/rcp/. Zuletzt besucht am 14.04.2011.

[13] Eclipse: Eclipse GEF. http://www.eclipse.org/gef/. Zuletzt besucht am 14.04.2011.

[14] Moore B., Dean D., Gerber A., Wagenknecht G. Vanderheyden P.: Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework. http://www.redbooks.ibm.com/redbooks/pdfs/sg246302.pdf. Zuletzt besucht am 25.11.2010.

[15] Eclipse: Eclipse Draw2d. http://www.eclipse.org/gef/draw2d/. Zuletzt besucht am 29.12.2011.

[16] JGraph: JGraph. http://www.jgraph.com/jgraph.html/. Zuletzt besucht am 14.12.2011.

[17] Wagner N.: Entwicklung eines Usability-Konzepts für die Modellierungsumgebung eines datenorientierten Prozess-Management-Systems, Diplomarbeit, Universität Ulm, (2010)

Bildnachweis

[1] Überblick Eclipse Architektur, http://www.jdg2e.com/ch08.architecture/doc/index.html

[2] MVC-Konzept im Eclipse GEF, http://www.linuxtopia.org/online_books/eclipse_documentation/eclipse_gef_draw2d_plug-in/topic/org.eclipse.gef.doc.isv/guide/eclipse_gef_draw2d_guide.html

A

Überblick über die Paketstruktur

- **philharmonic** Dieses Paket enthält die notwendigen Klassen um die Rich-Client-Application[siehe 3.2.2] zu starten.
- **philharmonic.commands** Dieses Paket enthält den benötigten Command zum Öffnen eines neuen Editors.
- **philharmonic.constantValues** In diesem Paket befinden sich die gesammelten Konstanten, die das Aussehen der Oberfläche und die festgelegten Farben des Farbkonzepts bereitstellen.
- **philharmonic.editor** Enthält für die graphischen Editoren notwendige Hilfsklassen.
- **philharmonic.editor.editors** Hier befinden sich die Implementierungen aller Editoren von PHILharmonicFlows.
- **philharmonic.editor.factories** Dieses Paket enthält die für die Erzeugung von EditParts notwendigen Fabriken. Diese erzeugen abhängig von dem übergebenen Modell-Objekt das jeweils passende EditPart für ide Darstellung im Editor.
- **philharmonic.editor.outlines** Hier befinden sich die Implementierungen für die Outlines der einzelnen Editoren.
- **philharmonic.editor.palette** In diesem Paket befinden sich die Klassen, die den Inhalt der Creation-Palette, die der Navigation und der Erzeugung neuer Komponenten dient, regeln.

- **philharmonic.model** In diesem Paket befinden sich die den verschiedenen Modelldaten zugrundeliegenden abstrakten Klassen und die zur Verwaltung der kompletten Datenstruktur benötigte DataModell Klasse.
- **philharmonic.model.activities** Enthält die Klassen des activity Modells.
- **philharmonic.model.coordinationModel** Hier befinden sich die Klassen zur Verwaltung der Daten des Coordination Modells.
- **philharmonicmodel.dataModel** Dieses Paket enthält die zur Verwaltung des Data Modells benötigten Klassen.
- **philharmonic.model.dataModel.actions** Implementierungen der für den ObjectTypeEditor[siehe 5.3.3]nötigen Actions.
- **philharmonic.model.dataModel.commands** Alle Klassen die Befehle in den Data Structure Editoren[siehe 5.3.5] ausführen.
- **philharmonic.model.dataModel.editParts** Die für die Realisierung des Object Type Editors notwendigen Controller(Edit Parts).
- **philharmonic.model.dataModel.editPolicies** Hier befinden sich die für den Object Type Editor notwendigen externen `Edit Policies`.
- **philharmonic.model.dataModel.exceptions** In diesem Paket befinden sich die für die Erkennung und Verwaltung der Zykluserkennung im Object Type Editor befindlichen Klassen.
- **philharmonic.model.dataModel.properties** Enthält die für die Data Structure Editoren implementierten Sections für die `properties view`.
- **philharmonic.model.enums** Die in den Daten verwaltenden Klassen notwendigen Enumerationen werden in diesem Paket definiert.
- **philharmonic.model.general.properties** In diesem Paket sind die abstrakten grundlegenden Klassen zur Implementierung der `properties view` enthalten. Außerdem befindet sich die Sektion für die Benennung des ausgewählten Objekts hier.
- **philharmonic.model.icons** Paket für die Speicherung der in der Implementierung benötigter Icons.
- **philharmonic.model.macroProcess** Implementierungen der für den Macro Process zugehörigen Strukturen.
- **philharmonic.model.Parser** In diesem Paket befinden sich die Klassen, die zum Parsen und Verwalten der Data Context Objekte benötigt werden.
- **philharmonic.model.permissons** Hier befinden sich die Klassen für die Verwaltung der Berechtigungen.

- **philharmonic.model.processStructure** Die Klassen für die Verwaltung der Daten der Mikro Prozesse sind in diesem Paket implementiert.
- **philharmonic.model.processStructure.commands** Dieses Paket enthält die Implementierungen der Befehle, die in den Prozess Struktur Editoren ausgeführt werden können.
- **philharmonic.model.processStructure.dialogs** Dialoge, die im Prozess Struktur Editor benötigt werden, sind in diesem Paket realisiert.
- **philharmonic.model.processStructure.editParts** Controller für die Realisierung der Prozess Struktur Editoren anhand der MVC-Architektur.
- **philharmonic.model.processStructure.editPolicies** Die externen `Edit Policiy` Klassen für den Editor der Prozess Struktur befinden sich in diesem Paket.
- **philharmonic.model.processStructure.exceptions** Klassen, die der Ausnahmebehandlung im Prozess Struktur Editor dienen sind hier implementiert.
- **philharmonic.model.processStructure.properties** Die Sektionen die der Darstellung der Eigenschaften der Prozess Struktur Elemente in der `properties view` dienen befinden sich hier.
- **philharmonic.model.processStructure.provider** Provider für in der Prozess Struktur verwendete Dialoge.
- **philharmonic.perspectives** Die verschiedenen vorgegebenen Perspektiven für die Editieroperationen sind in diesem Paket festgeschrieben.
- **philharmonic.swtObjects** Erweiterungen von SWT-Klassen zur Darstellugn benötigter Objekte befinden sich an diesem Ort.
- **philharmonic.views.editPolicies** Die `Edit Policies` für in Views benutzten graphischen Editoren sind in diesem Paket enthalten.
- **philharmonic.views.structureCompass** Die für den Struktur Kompass benötigten Klassen.

B

Glossar

Abstract Window Toolkit

Das Abstract Window Toolkit (AWT) ist die ursprüngliche Bibliothek von Java für die Entwicklung grafischer Benutzeroberflächen und ist Bestandteil der „Java Platform, Standard Edition". Es ist als schwergewichtige Komponente einzuordnen, da es zur Darstellung die nativen Routinen des Betriebssystems nutzt. Daher ist die Darstellung der Benutzeroberfläche auch immer an das Aussehen der Benutzeroberfläche angepasst und bietet nur eine beschränkte Auswahl an Oberflächenelementen.

Cascading Style Sheets

Cascading Style Sheets (abgekürzt CSS) ist eine Sprache, die die konkrete Darstellung (Schriftart, Schriftgröße, Farbe usw.) eines strukturierten Dokuments beschreibt. Die Trennung ermöglicht eine einfache, zentrale Änderung der Darstellung und die Möglichkeit mehrere Dokumente mit einer einheitlichen Darstellung durch eine gemeinsame Darstellungsbeschreibung zu versehen. Sie können damit entfernt mit Formatvorlagen in Textverarbeitungsprogrammen verglichen werden.

Java Platform, Standard Edition

Die „Java Platform, Standard Edition“ - kurz Java SE - bietet eine grundlegende Plattform für Java und besteht aus der Java Virtual Machine und einer Sammlung der wichtigsten Bibliotheken. Aufbauend auf Java SE werden die meisten Java Versionen, beispielsweise die Java Runtime Environment (JRE) als Laufzeitumgebung und das Java Development Kit (JDK) für die Softwareentwicklung, entwickelt.

OSGi Alliance

Die OSGi Alliance wurde 1999 als nicht-profitorientierte Organisation gegründet. Sie ist ein Zusammenschluss vieler Firmen und Organisationen, die die Entwicklung des Standards für die „OSGi Service Platform“ vorantreiben.

Standard Widget Toolkit

Das Standard Widget Toolkit (SWT) ist eine Bibliothek zur Erstellung grafischer Oberflächen. Es gehört im Gegensatz zu Swing und AWT nicht zu „Java SE“ und muss daher mit der jeweiligen Applikation mitgeliefert werden. Es ist als Konkurrenz zu Swing einzuordnen, basiert jedoch auf einem schwergewichtigen Aufbau. Dies bedeutet, dass es die nativen Funktionen des Betriebssystems zur Erzeugung der grafischen Komponenten nutzt und diese nicht selbst rendert.

Swing

Swing ist ein Toolkit für die Erstellung grafischer Benutzeroberflächen und Teil der „Java Platform, Standard Edition“ . Ziel seiner Entwicklung war eine größere Auswahl an Komponenten als das vorhergehende AWT zu bieten. Die Komponenten sind leichtgewichtig und werden damit direkt durch Java gerendert. Somit ist ihr Aussehen unabhängig vom Betriebssystem. Jedoch ist es durch Verwendung von verschiedenen Skins möglich das Aussehen der Komponenten zu verändern.

Viererbande

Die Viererbande, oft auch als „Gang of Four“ oder nur kurz „GoF“ bezeichnet sind die vier Autoren des Buchs „Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software“ Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides.

Ich erkläre, dass ich die Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ulm, den 28.04.2011

_____

Andreas Pröbstle