ulm university universität ulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Datenbanken und Informationssysteme

Integration der Modellierungs- und Laufzeitumgebung eines datenorientierten Prozess-Management-Systems

Masterarbeit an der Universität Ulm

Vorgelegt von:
Thomas Spindler
thomas.spindler@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert
Prof. Dr. Peter Dadam

Betreuer:
Dipl. Inf. Vera Künzle

2013

„Integration der Modellierungs- und Laufzeitumgebung eines datenorientierten Prozess-Management-Systems“
Fassung vom 4. Februar 2013

Bei der Erstellung dieser Diplomarbeit wurde freie Software eingesetzt:

Satz: PDF-LaTeX 2$_\varepsilon$
Druck: Druckerei der Universität Ulm

Danksagung

An dieser Stelle möchte ich denjenigen danken, die mich bei dieser Arbeit unterstützt haben. Mein besonderer Dank gilt meinen beiden Gutachtern Prof. Dr. Manfred Reichert und Prof. Dr. Peter Dadam, die mich mit den richtigen Worten zur richtigen Zeit auf dem Weg gehalten haben und diese Arbeit ermöglichten. Ferner danke ich meiner Betreuerin Vera Künzle, die mir mit Ratschlägen und Denkanstößen helfend zur Seite gestanden hat. Danke auch Julian Tiedeken, Hannes Beck und Stefan Schultz, die mich in die Grundlagen der Software-Prototypen eingeführt haben. Abschließend vielen Dank all denen, die mich während meines Studiums und bei dieser Masterarbeit tatkräftig unterstützt haben.

Inhaltsverzeichnis

1 Einleitung

Wirtschaftsunternehmen stehen in Konkurrenz zu anderen Unternehmen, die auch global verteilt sein können. Um konkurrenzfähig zu bleiben, ist es wichtig, Arbeitsabläufe und Prozesse regelmäßig überprüfen, und ggf. zu optimieren. Um dies rechnerunterstützt zu realisieren, benötigt man eine standardisierte Prozessbeschreibungssprache sowie ein Prozess-Management-System (PrMS), das den Prozess ausführt. BPMN [5] und Aristaflow [3] sind dafür zwei Beispiele.
Die klassische Vorgehensweise ist die Modellierung von Prozessen auf Basis einzelner Aktivitäten, die den Kern des Prozesses bilden. Eine Aktivität wird dann jeweils mit einer Anwendungsfunktion verknüpft. Die Verwaltung der Anwendungsdaten erfolgt selbstständig durch die eingebunden Anwendungsfunktionen. Das PrMS hat keine Kontrolle über die einzelnen Datenobjekte. Innerhalb des Datenflusses können lediglich einzelne Datenelemente (z.B. Attribute) integriert werden, die als Ein- und Ausgabeparameter mit den jeweiligen Aktivitäten verknüpft werden. Der Fortschritt eines Prozesses definiert sich ausschließlich über die abgearbeiteten Aktivitäten. Zur Laufzeit werden dem Endbenutzer die aktuell anstehenden Aktivitäten innerhalb von Arbeitslisten angeboten. Ein Prozess-Management-System stellt daher lediglich eine prozessorientierte Sicht zur Verfügung. Eine daten- und funktionsorientierte Sicht, die es dem Benutzer ermöglicht, entsprechende Datenänderungen und Funktionsaufrufe optional, d.h. außerhalb der obligatorisch geforderten Aktivitäten auszuführen, fehlt vollständig [10].
Am DBIS[1] Institut wurde deshalb ein alternatives Konzept erarbeitet, das Benutzern neben der Ausführung von Prozessen auch den gleichzeitigen Zugang zu Daten und Funktionen ermöglicht. Im Unterschied zur prozessorientierten Sicht schreitet der Prozess nur fort, wenn die notwendigen Daten vorhanden sind.

[1] Datenbanken und Informationssysteme

1.1 Motivation

Die Grundlage dieser Arbeit ist das PHILharmonicFlows-Konzept von Vera Künzle, das im Institut für Datenbanken und Informationssysteme (DBIS) an der Universität Ulm entwickelt wurde. Dieses Konzept umfasst sowohl eine Prozess-Beschreibungs-Sprache als auch eine operationale Semantik für die Ausführung. Des weiteren wird zur Laufzeit neben einer prozessorientierten Sicht auch eine daten- und funktionsorientierte Sicht zur Verfügung gestellt. Im Zuge der Evaluation wird dafür eine Modellierungsumgebung und eine Laufzeitumgebung entwickelt.
Durch verschiedene Arbeiten sind bereits Prototypen der Modellierungsumgebung [1] und der Laufzeitumgebung [23] entstanden. Sie realisieren bisher aber nur einen Teil des Konzepts von PHILharmonicFlows.

1.2 Ziel der Arbeit

Im Rahmen dieser Arbeit sollen weitere Teile der Modellierungs- und der Laufzeitumgebung implementiert werden. Ein Schwerpunkt dieser Arbeit liegt auf der Integration beider Umgebungen. Die Modellierungsumgebung kann jedoch keine vollständigen Prozesse persistent speichern, die später die Laufzeitumgebung laden und ausführen kann. Dies ist für den späteren Testlauf des Prototyps unbedingt notwendig. Weiterhin fehlen Komponenten, (z.B. eine Benutzerintegration) die für diese Arbeit benötigt werden.
Zusammengefasst ergeben sich folgende Schwerpunkte:

- Modellierungsumgebung
  - Datenbankverbindung
  - Makro-Prozesse
  - Benutzerintegration
- Laufzeitumgebung
  - Datenbankverbindung
  - Benutzerintegration

Dabei sind mehrere Dinge zu beachten:

- wenige Änderungen des Original Codes, um doppelte Arbeit zu vermeiden
- Korrekte Implementierung gegenüber dem PHILharmonicFlows-Konzepts
- Umsetzung der Usability und Design Konzepte, die in verschiedenen Diplomarbeiten erarbeitet wurden [27][21]

1.3 Gliederung

Kapitel zwei und drei beschreiben die Grundlagen, die für das Verständnis dieser Arbeit wichtig sind. Dabei wird zunächst auf das Konzept von PHILharmonicFlows eingegangen und später auf die Technologien, die in den Implementierungen verwendet werden (Entwicklungsumgebung, Datenbank-Technologie, etc.). Im vierten Kapitel wird der Entwicklungsstand jeder Teilimplementierung des Prototyps behandelt. Dies soll den aktuellen Stand als auch die Lücken und offenen Arbeitsfelder des Prototyps aufzeigen.
Im Anschluss daran werden die erarbeiteten Lösungen vorgestellt. Jeder Schwerpunkt erhält ein eigenes Kapitel, in dem verschiedene Lösungsansätze abgewogen werden. Danach wird die Umsetzung in einem grafischen und einem programmiertechnischen Teil gezeigt. Zuletzt wird eine Zusammenfassung der Ergebnisse geliefert.
Im letzten Kapitel werden die Ergebnisse dieser Arbeit zusammengefasst und ein Ausblick auf noch fehlende Implementierungen gegeben.

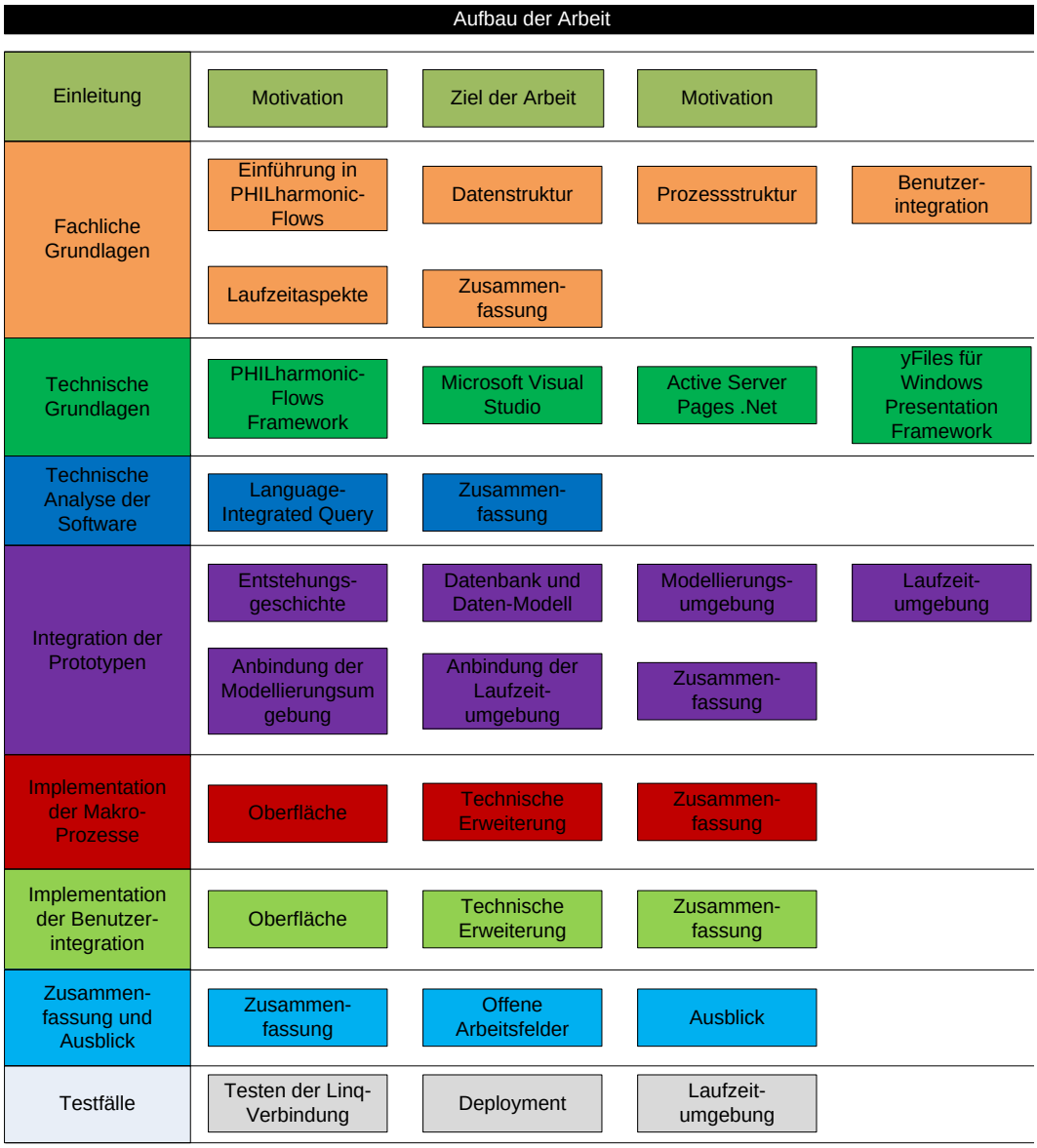


Abbildung 1.1: Aufbau der Arbeit)

2 Fachliche Grundlagen

Bevor mit der eigentlichen Beschreibung der Programmierarbeit an den Prototypen begonnen werden kann, erfolgt in diesem Kapitel zunächst die Klärung der fachlichen Grundlagen dafür.
Als erstes wird ein Überblick über den Aufbau von PHILharmonicFlows gegeben. Danach erfolgt die Vorstellung der Datenstruktur. Diese ist für ein datenorientiertes PrMS ein wichtiger Punkt. Im Anschluss wird auf die Prozesskomponente genauer eingegangen. Hierbei werden die Interaktion zwischen Daten und Prozess aufgezeigt und die Steuerung des Prozesses besprochen. Danach wird die Integration der Benutzerverwaltung und Rechtesteuerung behandelt. Zuletzt werden die notwendigen Erweiterungen für die Laufzeitaspekte vorgestellt.
In Abb. 2.1 ist eine kurze Übersicht über das Kapitel ersichtlich.

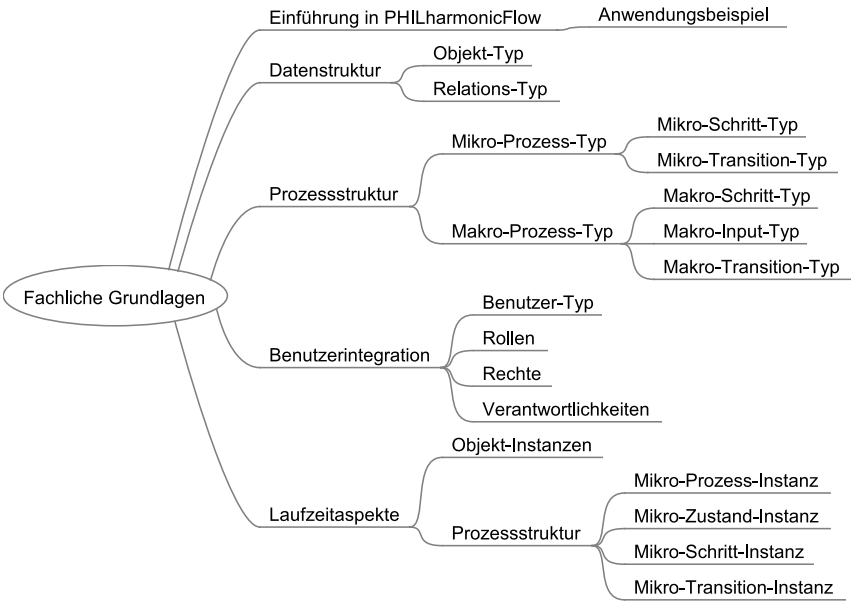


Abbildung 2.1: Übersicht über das Kapitel

2.1 Einführung in PHILhamonicFlows

In einem prozessorientierten PrMS sind die Prozesse über eine feste Abfolge von Aktivitäten definiert. Um Daten zwischen diesen Aktivitäten auszutauschen, müssen diese als Ein- und Ausgabeparameter modelliert werden. Dies bedeutet, dass Änderungen an Daten nur innerhalb einer fest vordefinierten Aktivität erfolgen können, die in einer festgelegten Reihenfolge ausgeführt werden müssen. Falls keine weitere Aktivität ein Datenfeld schreibt, kann dieses auch nicht geändert werden.
Das Ziel des PHILharmonicFlows-Konzepts ist die Entwicklung eines datenorientierten PrMS, welches die Daten in den Mittelpunkt stellt. Der Prozessfortschritt definiert sich über das Vorhandensein von Daten. Während der Prozessausführung sollten die Daten jederzeit veränderbar sein, damit die Gefahr eines Deadlock nicht besteht.
Wie in Abb. 2.2 zu sehen, ist PHILharmonicFlows in drei Strukturen aufgeteilt: die Datenstruktur, die Prozessstruktur und die Benutzerintegration. Diese Strukturen werden im Folgenden genauer vorgestellt.

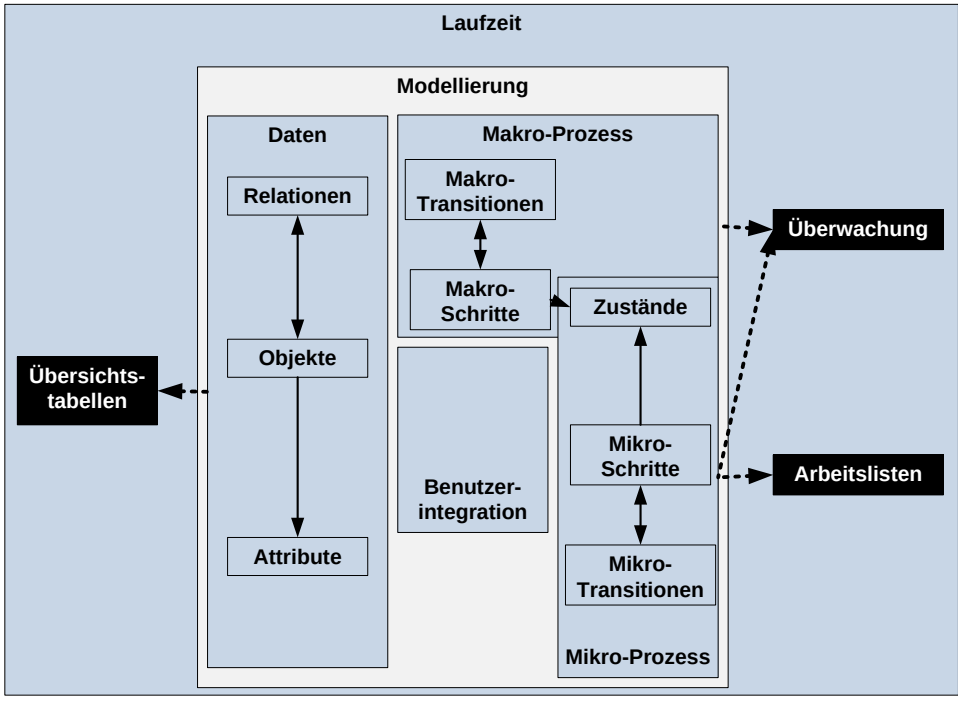


Abbildung 2.2: Metamodell von PHILharmonicFlows (nach [10])

Zur besseren Kennzeichnung der Modellstruktur werden diese als <Element>-*Typ* bezeichnet und während der Laufzeit als <Element>-*Instanz* bezeichnet. Diese genaue Kennzeichnung betrifft nur dieses Kapitel. In den nachfolgenden Kapiteln werden die ”-Typen”, wenn möglich weggelassen, um die Lesbarkeit zu erhöhen. Einzig die Objekt-Typen behalten ihre Kennzeichnung, da dies sonst zu Verwirrungen führt.
Um die Strukturen leichter zu verstehen, werden sie mit einem kleinen Beispiel veranschaulicht, das im folgenden Kapitel vorgestellt wird.

2.1.1 Anwendungsbeispiel

Das Anwendungsbeispiel stammt aus dem Personalmanagment [11]. Hier wird bewusst auf ein eigenes Beispiel verzichtet, um eine einfachere Einarbeitung in die Originalquelle [11] zu ermöglichen. Im Original ist das Beispiel ausführlicher erläutert, was im Rahmen dieser Arbeit aber keinen Mehrwert bringen würde.
Das Szenario beschreibt die `Stellenausschreibung` einer Firma. Diese Stellenausschreibung soll digital über ein PrMS mit Webinterface abgearbeitet werden. Über das Webinterface kann ein `Bewerber` eine `Bewerbung` abschicken, welche die notwendigen persönlichen Informationen enthält. Eine solche `Bewerbung` wird einem `Personalmanager` zugeleitet. Dieser entscheidet, ob ein `Bewerber` für die `Stellenbesetzung` geeignet ist. Falls nicht, wird die `Bewerbung` abgelehnt, bei Eignung beantragt der `Personalmanager` eine beliebige Anzahl von `Gutachten`. Diese werden von ausgewählten `Mitarbeitern` in den jeweiligen `Abteilungen` erstellt und mit einer kommentierten Empfehlung versehen. Die `Mitarbeiter` haben die Möglichkeit, das Erstellen eines solchen `Gutachtens` abzulehnen.
Sind die geforderten `Gutachten` beisammen, entscheidet der `Personalmanager` auf der Basis der `Gutachten`, ob ein `Bewerber` zum `Bewerbungsgespräch` eingeladen wird. Vor jedem `Bewerbungsgespräch` erhalten die `Prüfer Handouts` mit den wichtigsten Daten und Platz für Notizen. Nach dem `Bewerbungsgespräch` wird entschieden, ob der `Bewerber` für diese `Stelle` geeignet, ungeeignet oder für eine alternative `Stelle` geeignet ist. Sind alle `Bewerbungsgespräche` beendet, wird eine Entscheidung getroffen, welcher `Bewerber` die offene `Stelle` angeboten bekommt. Ist die offene `Stelle` besetzt, so werden alle noch offenen Verfahren zu Ende gebracht und der Gesamtprozess beendet.

2.2 Datenstruktur

Die Datenstruktur definiert, wie die Daten in einem Prozess gespeichert werden und wie diese untereinander vernetzt sind. Ein Kriterium bei der Entwicklung der Datenstruktur ist, dass die Daten in einer Relationalen Datenbank gespeichert werden. Aus diesem Grund gelten für die Datenstruktur dieselben Einschränkungen wie für Relationale Datenbanken, wie z.B. eindeutige Kennungen oder feste Datenstrukturen.

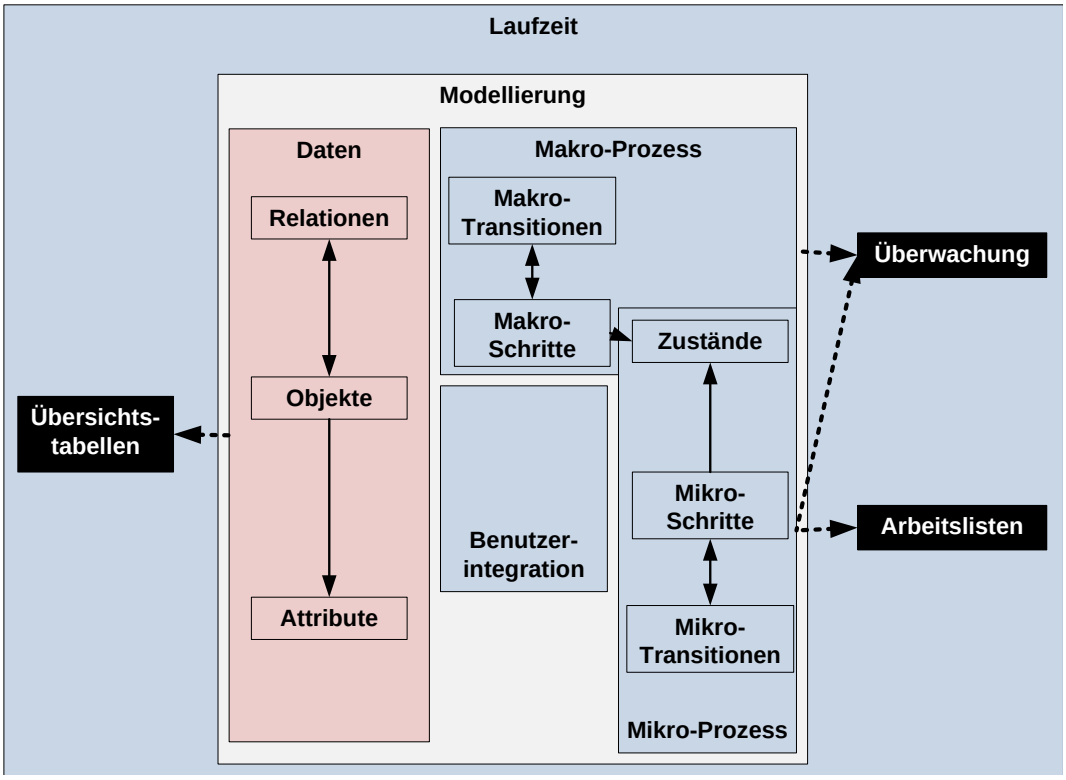


Abbildung 2.3: Metamodell von PHILharmonicFlows

Die Abb. 2.3 zeigt den behandelten Inhalt im Kontext.

2.2.1 Architektur

Kern der Datenstruktur ist der sogenannte *Objekt-Typ*. Dieser repräsentiert Objekte (z.B. Gutachten, Bewerbungsgespräch oder Bewerber) und enthält die Daten in sogenannten

Attribut-Typen. Die *Objekt-Typen* stehen mit Hilfe eines *Relations-Typs* in Beziehung zueinander.

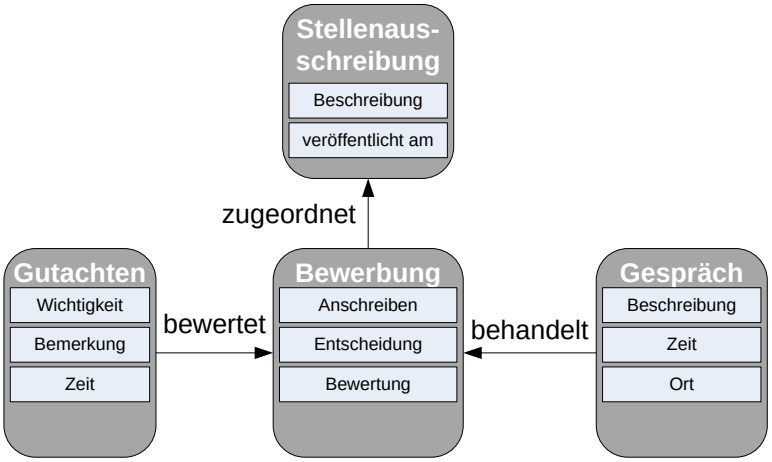


Abbildung 2.4: Beispiel einer Datenstruktur [11]

Die Abb. 2.4 zeigt die Datenstruktur des Beispiels. Die einzelnen Elemente werden nun genauer erklärt.

2.2.2 Objekt-Typ

Der *Objekttyp* ist ein Container für die Daten. Dieser kann eine Menge von *Attribut-Typen* enthalten, welche die Art der Daten repräsentieren. Ein *Attribut-Typ* wird durch einen Namen, einen Datentyp und einen empfohlenen Wert definiert. Die *Attribut-Typen* sind in Abb. 2.4 als weiße Kästchen dargestellt. In dem Beispiel besitzt das `Review` mehrere modellierte *Attribut-Typen*, wie `Zeit` oder `Bemerkung`. Die *Objekt-Typen* werden so genannten Datenebenen zugewiesen. Im Beispiel (Abb. 2.5) sind `Mitarbeiter`, `Stellenausschreibung` und `Bewerber` die erste Datenebene, `Bewerbung` ist die zweite Datenebene, und `Review` und `Interview` sind die dritte Datenebene. Die Aufteilung der Datenstruktur in die *Datenebene* ist wichtig, um verschiedene Beziehungsarten abzuleiten. Im Kapitel 2.3.2 werden diese Beziehungen genauer erläutert.

Daneben gibt es die sogenannten *Werte-Typen*. Diese besitzen keine Relations-Typen und werden in einer gesonderten Datenebene eingeordnet. Von diesen *Werte-Typen* können während der Modellierung Instanzen gebildet und mit Werten versehen werden. Attribut-

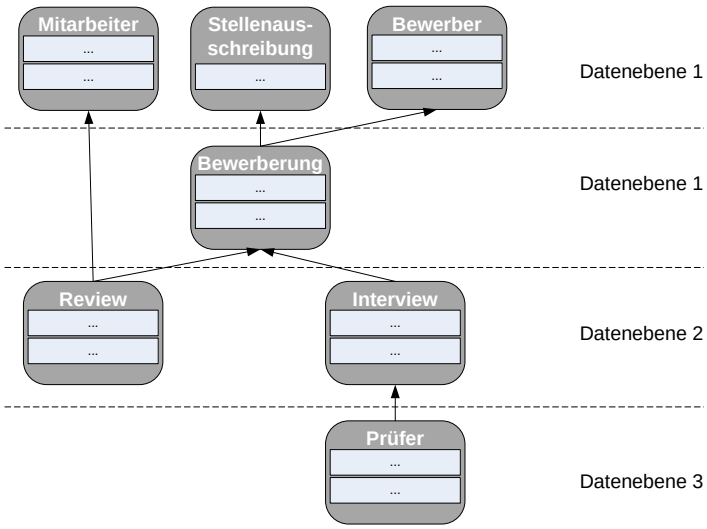


Abbildung 2.5: Datenebene des Beispiels

Typen von Objekt-Typen können auf die Attribut-Typen des *Werte-Typs* referenzieren und dürfen zur Laufzeit nur diese definierten Werte annehmen.

2.2.3 Relation-Typ

Die *Objekt-Typen* stehen in einem Verhältnis zueinander. Dieses Verhältnis wird durch einen *Relations-Typ* beschrieben. Wie in Abb. 2.4 dargestellt, wird dieser *Relations-Typ* als Pfeil zwischen den *Objekt-Typen* symbolisiert. Ein *Relations-Typ* besitzt einen Namen, um ihn für den Modellierer unterscheidbar zu machen. Wichtig ist, dass *Relations-Typen* nur Beziehungen mit unterschiedlichen Datenebenen erlaubt sind.

2.3 Prozessstruktur

Die Prozessstruktur definiert den Ablauf der Prozesse in PHILharmonicFlows. Hierbei wird zwischen zwei Prozessarten unterschieden, den *Mikro-Prozess-Typen* und den *Makro-Prozess-Typen*. Diese werden nun genauer erklärt.
Die Abb. 2.6 zeigt den hier behandelten Inhalt im Kontext.

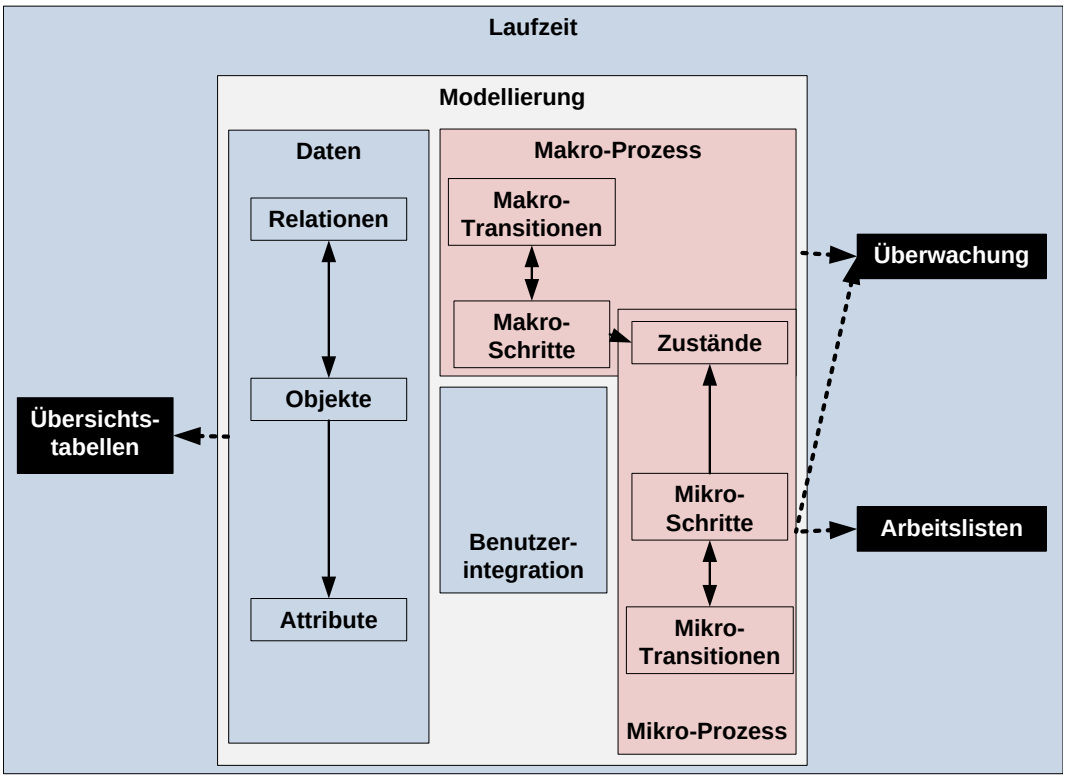


Abbildung 2.6: Metamodell von PHILharmonicFlows

2.3.1 Mikro-Prozess-Typ

Der *Mikro-Prozess-Typ* definiert den Bearbeitungsvorgang eines Objekt-Typs. Aus diesem Grund ist jedem Objekt-Typ exakt ein korrespondierender Mikro-Prozess-Typ zugeordnet. Beim Erstellen eines Objekt-Typs wird auch ein generischer minimaler *Mikro-Prozess-Typ* generiert. Ein *Mikro-Prozess-Typ* ist ein gerichteter Graph, in dem die Knoten sogenannte *Mikro-Schritt-Typen* sind und die Kanten sogenannte *Mikro-Transitions-Typen* (vgl. Abb. 2.7). Diese *Mikro-Schritt-Typen* werden zu sogenannten *Zustands-Typen* gruppiert. Dabei muss jeder *Miko-Schritt-Typ* genau einem *Zustands-Typen* zugewiesen sein.

Mikro-Schritt-Typ

Der *Mikro-Schritt-Typ* definiert, welcher Attribut-Typ geschrieben werden soll. Es existieren drei Arten von *Mikro-Schritt-Typen*:

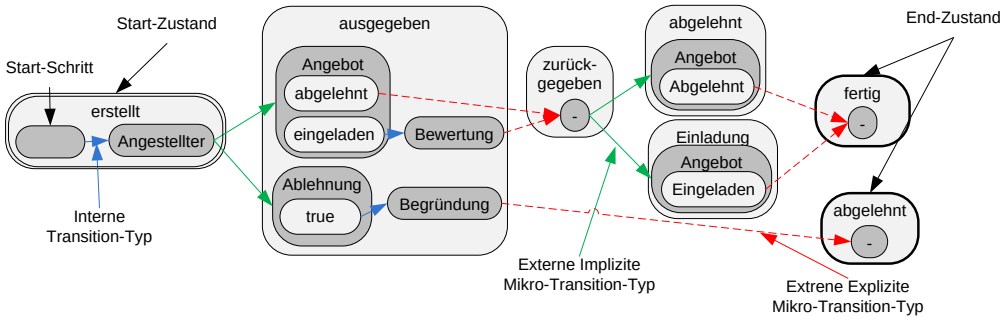


Abbildung 2.7: Beispiel eines Mikro-Prozess-Typs

- **leere Mikro-Schritt-Typ**
- **atomare Mikro-Schritt-Typ** referenziert jeweils einen Attribut-Typ oder einen Relations-Typ.
- **wertespezifische Mikro-Schritt-Typ** referenziert einen Attribut-Typ und beinhaltet mehrere untergeordnete Mikro-Werte-Schritt-Typen.

Die *Mikro-Schritt-Typen* werden durch *Mikro-Transitions-Typen* verbunden.

Mikro-Transitions-Typ

Der Mikro-Transitions-Typ (kurz Transitions-Typ) besitzt zwei Unterscheidungsmerkmale für *Mikro-Transition-Typen*: Verbindungs- und Schaltungsart. Die Verbindungsart (intern, extern) wird bestimmt durch die *Mikro-Schritte*, welche der *Mikro-Transitions-Typ* verbindet. Gehören die *Mikro-Schritt-Typen* zu demselben *Zustands-Typ*, wird dies als *interne Transition* bezeichnet, andernfalls als *externe Transition*.
Die Schaltungsart (implizit, explizit) definiert der Modellierer. Er kann zwischen automatischer Schaltung, der sogenannten *impliziten Transition* und der manuellen Schaltung, der sogenannten *expliziten Transition*, wählen. *interne Transitionen* sind immer *implizit*. Besitzt ein *Mikro-Schritt-Typ* mehrere ausgehende *Mikro-Transitions-Typen*, so müssen diese mit unterschiedlichen Prioritäten versehen werden.

Neben diesen *Mikro-Transitions-Typen* existieren sogenannte *Rücksprung-Transition-Typen*, die zwei *Zustands-Typen* miteinander verbinden. Dies wird genutzt, um auf einen

schon abgeschlossenen *Zustand-Typ* zurück zu springen. Ein solcher Rücksprung kann nur von einem Benutzer ausgelöst werden und ist daher immer explizit.

2.3.2 Makro-Prozess-Typ

Im Beispielszenario ist gefordert, dass mehrere Gutachten zu einer Bewerbung erstellt werden sollen. Mit Mikro-Prozess-Typen alleine wäre dies nicht machbar, da mehrere Objekt-Typen existieren und diese synchronisiert werden müssen. Aus diesem Grund existiert das Konzept des *Makro-Prozess-Typs*, um dem Modellierer die Möglichkeit zu geben, solche Gegebenheiten zu modellieren und Mikro-Prozess-Typen zu synchronisieren. Der *Makro-Prozess-Typ* dient zur Synchronisation der verschiedenen Mikro-Prozess-Typen. Der Makro-Prozess-Typ ist kein Prozess der abgearbeitet wird, sondern ein Steuerungsmechanismus für die Mikro-Prozess-Instanzen. Der *Makro-Prozess-Typ* ist vergleichbar mit den Petri-Netzen [6], in dem die Token Mikro-Prozess-Instanzen sind und die Petri-Transitionen den Makro-Transitions-Typen entsprechen.
Die Abb. 2.8 zeigt ein Beispiel eines *Makro-Prozess-Typs* in dem Beispielszenario.

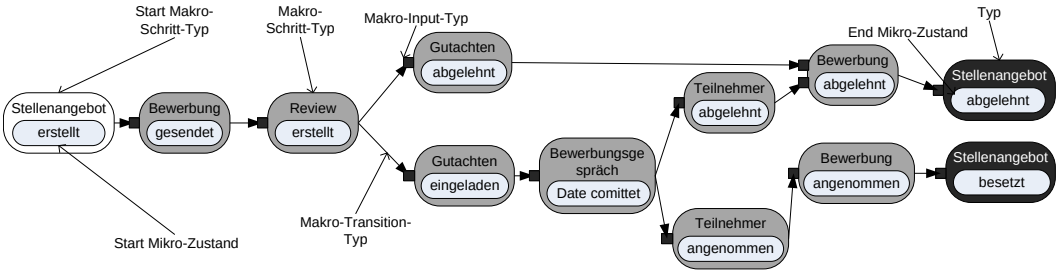


Abbildung 2.8: Beispiel eines Makro-Prozess-Typs

Ähnlich den Mikro-Prozess-Typen existiert der sogenannte *Makro-Schritt-Typ*, der auf einen Objekt-Typ und einen seiner Zustände verweist. Der Kontrollfluss wird über *Makro-Transitions-Typen* und *Makro-Input-Typen* realisiert.

Makro-Schritt-Typ

Der *Makro-Schritt* verweist auf genau einen Objekt-Typ und einen seiner Zustände. Der Start-Makro-Schritt und die End-Makro-Schritte müssen mit dem Mikro-Prozess korrespon-

dieren. Alle Zwischen-Schritte können nicht frei gewählt werden und müssen gewissen Regeln folgen, die in der Arbeit von Vera Künzle genau beschrieben sind [10].

Makro-Input-Typ

Die *Makro-Input-Typen* realisieren die Kontrollflusslogik in einem Prozess. Um einen *Makro-Schritt-Typ* zu aktivieren, muss mindestens ein *Makro-Input-Typ* erfüllt werden. Dies entspricht einer 'Or'-Semantik. Damit ein *Makro-Input-Typ* erfüllt ist, müssen alle eingehenden *Makro-Transitions-Typen* erfüllt sein. Dies entspricht der 'And'-Semantik.

Makro-Transitionen-Typ

Der *Makro-Transitions-Typ* verbindet zwei Makro-Schritte miteinander. Ähnlich wie die Mikro-Transition, stellen diese Bedingungen, die erfüllt sein müssen, bevor die involvierten Mikro-Prozess-Typen (Objekt-Typen) weiter schalten dürfen. Je nachdem auf welche Objekt-Typen die zwei *Makro-Schritte* verweisen, müssen verschiedene *Bedingungs-Typen* definiert werden. Im Folgenden gibt es vier verschiedene Transitionsarten, die abhängig

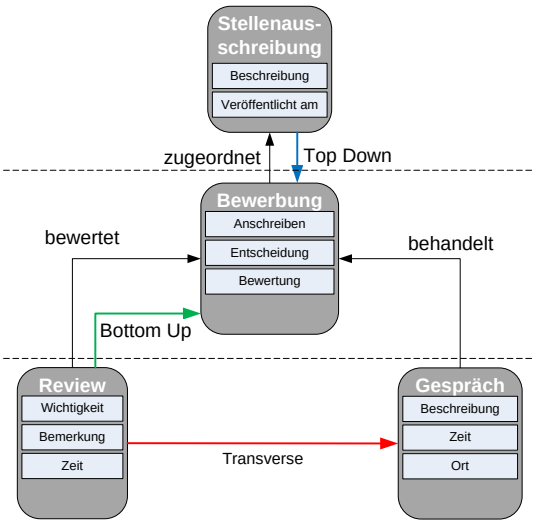


Abbildung 2.9: Beispiel der Bedingungen der Makro-Transition

von den verbundenen Objekt-Typen bestimmt sind:

Top-Down beschreibt eine Transition eines Objekt-Typ mit höherer Datenebene zu einem Objekt-Typ einer niedrigeren Datenebene. Dabei ist es wichtig, dass ein Pfad im Graphen zwischen den Objekt-Typen existiert. Zum Beispiel existiert zwischen `Gespräch` und `Gutachten` kein Pfad, während zwischen `Bewerbung` und `Gutachten` ein Pfad besteht (siehe Grüner Pfeil in Abb. 2.9). Eine solcher *Makro-Transitions-Typ* muss eine sogenannte *Prozess-Kontext-Typ* Bedingung definieren. Diese Bedingung aggregiert die höheren Mikro-Prozess-Instanzen bezüglich eines oder mehrerer ausgewählten Zustands-Typen.

Bottom-Up beschreibt den umgekehrten Fall von *Top-Down* und verlangt einen Pfad eines Objekt-Typs zu einem Objekt-Typ mit einer höheren Datenebene.
Eine solcher *Makro-Transitions-Typ* muss eine sogennante *Aggregations-Typ* Bedingung definieren. Während der Laufzeit wertet das System alle Instanzen des niedrigen Objekt-Typs aus, die von dem höheren referenziert werden, und aggregiert diese zu bestimmten Gruppen:

- #ALL: zählt alle Instanzen zusammen, die vom höheren Objekt-Typ referenziert werden.
- #IN: zählt alle Instanzen zusammen, die sich im ausgewählten Zustand des Makro-Schritts befinden.
- #BEFORE: zählt alle Instanzen zusammen, die sich vor dem ausgewählten Zustand befinden.
- #AFTER: zählt alle Instanzen zusammen, die sich nach dem ausgewählten Zustand befinden.
- #SKIPPED: zählt alle Instanzen zusammen, die den ausgewählten Zustand als 'Skipped' markiert haben.

Der Modellierer definiert ein Prädikat, das die Werte der aggregierten Gruppen auswertet. Die Bedingung ist erfüllt, wenn das Prädikat erfüllt ist.

Transverse beschreibt einen *Makro-Transitions-Typ* zwischen zwei Objekt-Typen, welchen einen gemeinsamen übergeordneten Objekt-Typ besitzen. Ähnlich wie bei dem *Aggregations-Typ* werden die Instanzen des Ziel-Objekt-Typs im Bezug zum Objekt-Typ aggregiert und die Bedingung anhand eines Prädikates ausgewertet.

Self beschreibt einen *Makro-Transitions-Typ*, der zwei *Makro-Schritt-Typen* miteinander verbindet, die denselben Objekt-Typ referenzieren. Dieser Typ stellt keine Bedingung.

2.4 Benutzerintegration

Das PHILharmonicFlows-Konzept umfasst auch die Verwaltung von Benutzern. Ein Grundgedanke ist, die Benutzer in das Modell zu integrieren und nicht als separates Organigramm, wie es in traditionellen PrMS eingesetzt wird. Aus diesem Grund werden Benutzer durch besondere Objekt-Typen, den sogenannten *Benutzer-Typen*, in das Datenmodell integriert. Über diese werden Beziehungen zwischen Benutzern und Anwendungsdaten hergestellt. Diese *Benutzer-Typen* erhalten dann Rechte (z.B. 'Attribut lesen' oder 'Zustand weiterschalten').

2.4.1 Benutzer-Typ

Der *Benuzer-Typ* besitzt über die Relations-Typen des *Benutzer-Typs* eine Menge von Makro- oder Mikro-Prozess-Typen, die mit ihm in Verbindung stehen. Aus dieser Menge kann zur Laufzeit eine Arbeitsliste erstellt werden, falls der *Benutzer-Typ* über entsprechende Rechte verfügt.
Die *Rechte* werden nicht an *Benutzer-Typen*, sondern an *Rollen-Typen* gebunden. Dieser kann eine oder mehrere *Rollen* besitzen.

2.4.2 Rollen

Rollen verbinden *Benutzer-Typen* mit *Rechten*, dabei gibt es verschiedene *Rollen-Typen*:

- **Benutzer-Rollen-Typen** werden an *Benutzer-Typen* gebunden.
- **Relation-Rollen-Typen** werden an einen *Benutzer-Typ* und an eine seiner Relationen gebunden.

Im weiteren Verlauf der Arbeit werden diese verschieden Arten von *Rollen-Typen* der Einfachheit halber als *Rolle* zusammengefasst. Im PHILharmonicFlows-Konzept existieren weitere *Rollen-Typen*, die für diese Arbeit nicht relevant sind [11].

2.4.3 Rechte

Die *Rechte* erlauben den *Rollen* die Attribut-Typen in der Datenstruktur zu *Lesen* und zu *Ändern*. Das System prüft, ob für jedes mögliche Recht mindestens eine Rolle zugewiesen ist, damit der Prozess in keine Deadlock-Situation gerät. Im Folgenden existieren Rechte für diese Elemente mit entsprechenden Möglichkeiten:

- Objekt-Instanz
  - Erstellen
  - Löschen
- Attribut-Instanz
  - Lesen
  - Optionales Schreiben
  - Verpflichtendes Schreiben
- Relation-Instanz
  - Lesen (Auf höhere Objekt-Typen zugreifen)
  - Schreiben (Den Objekt-Typ einem anderen höheren Objekt-Typ zuweisen)

2.4.4 Instanzspezifische Rechte

Die *instanzspezifischen Rechte* sind Rechte einer Rolle zu einem bestimmten Objekt-Typ. Bestimmt wird dieser Objekt-Typ über einen Pfad von Relationen. Zur Laufzeit werden die Objekt-Instanzen in zwei Gruppen eingeteilt: die abhängigen und die unabhängigen.
Die abhängigen Objekt-Instanzen sind über den Pfad der Relationen von der Benutzer-Instanz aus erreichbar. Für diese gelten die *instanzspezifischen Rechte*. Für die unabhängigen Objekt-Instanzen gelten die 'allgemeinen' Rechte.

Im Beispiel sollte der `Mitarbeiter` nur seine `Gutachten` bearbeiten können, und die `Gutachten` seiner Kollegen lesen können. Seine eigenen `Gutachten` sind abhängige Instanzen, während die der Kollegen unabhängige Instanzen sind.

2.4.5 Verantwortlichkeiten

Neben den *Rechten* kennt PHILharmonicFlows auch *Verantwortlichkeiten*. Während *Rechte* an die Datenstruktur geknüpft werden, sind *Verantwortlichkeiten* an die Prozess-Struktur gebunden. Dies bedeutet, dass eine *Rolle* in die operationale Ausführung (z.B. eines Mikro-Prozess-Instanz) eingreifen kann.

- **Makro-Prozess-Verantwortung** erlaubt es, den Zustand-Typ des Makro-Prozesses zu ändern.
- **Mikro-Prozess-Verantwortung** erlaubt es, den Zustand-Typ und die Werte zu ändern.
- **Zustand-Verantwortung** erlaubt es, die Attribut-Typen des Zustand-Typs zu ändern.
- **Commit-Verantwortung** erlaubt es, explizite Transitionen-Typen zu aktivieren.

Eine verantwortliche Rolle bekommt automatisch alle Rechte einer Struktur. So kann gewährleistet werden, dass jedes Recht mindestens einer Rolle zugewiesen ist.

2.5 Laufzeitaspekte

Nachdem die *Typen* modelliert sind, werden diese in die Laufzeitumgebung geladen. Dabei werden in der Laufzeitdatenbank entsprechende Tabellen für die *Instanzen* der jeweiligen *Typen* generiert (vgl. Abb. 2.10).

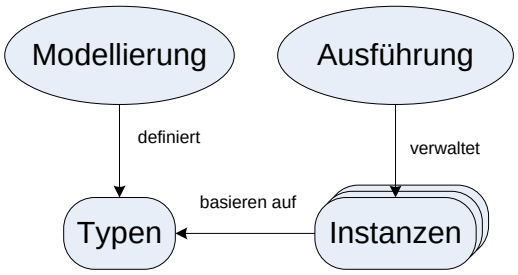


Abbildung 2.10: Typen und Instanzen (nach [23])

In den folgenden Kapiteln werden die technischen Unterschiede zwischen *Typen* und *Instanzen* erläutert. Jede Instanz enthält eine eindeutige ID zur Identifizierung, sowie einen

Zeitstempel, wann die Instanz erzeugt wurde und ein Feld für den Erzeuger der Instanz. Die Erweiterung um diese Felder wird in dem Kapitel nicht explizit erwähnt, da sie nur für Verwaltung und Nachvollziehbarkeit notwendig sind.

2.5.1 Objekt-Instanzen

Die Objekt-Instanzen erhalten keine neue Felder, jedoch erhält die Benutzer-Instanz jeweils ein Feld für 'Benutzernamen' und 'Passwort', damit sich ein Benutzer einloggen kann.

2.5.2 Prozessstruktur

Für die Abarbeitung und Verwaltung einer Prozess-Instanzs sind Markierungen notwendig, damit der Laufzeitumgebung bekannt ist, in welchem Zustand sich die Prozess-Instanz gerade befindet, und welche Entscheidungen der Benutzer getroffen hat.
In der operationalen Semantik [12] wird gezeigt, welche Markierungen existieren und wie die Übergänge dafür sind. Im Folgenden wird ein kurzer Überblick über die Markierung der jeweiligen Instanzen und die operationale Semantik der Markierungsübergänge gegeben. Ausführliche Informationen sind in den Quellen [12] und [10] zu finden.

Mikro-Prozess-Instanz

Eine Mikro-Prozess-Instanz wird erzeugt, sobald eine entsprechende Objekt-Instanz erzeugt wird. Ist dies geschehen, wird die Mikro-Prozess-Instanz als RUNNING markiert (vgl. Tabelle 2.1). Eine Mikro-Prozess-Instanz kann erst als FINISHED gekennzeichnet werden, wenn eine End-Zustand-Instanz einer Mikro-Prozess-Instanz mit CONFIRMED markiert wurde.

$\text{Prozess}_{\text{Mark}}$	**Erklärung**
RUNNING	Die Instanz wird momentan ausgeführt
FINISHED	Die Instanz hat einen Endzustand erreicht

Tabelle 2.1: Markierungen der Mikro-Prozess-Instanz [23]

Zustand-Instanz

Die Zustands-Instanz besitzt diverse mögliche Markierungen (vgl. Tabelle 2.2). Am Anfang ist jede Instanz mit der Markierung WAITING versehen, nur die Start-Zustand-Instanz, dieser wird als ACTIVATED markiert.

$Zustand_{Mark}$	**Erklärung**
WAITING	Der Zustand wurde noch nicht aktiviert, kann aber zu einem späteren Zeitpunkt aktiviert werden.
ACTIVATED	Der Zustand ist aktiviert.
CONFIRMED	Der Zustand ist abgeschlossen und der nächste Zustand wurde aktiviert.
SKIPPED	Der Zustand kann nicht mehr aktiviert werden, da er auf einem abgewählten, alternativen Pfad der Ausführung liegt.

Tabelle 2.2: Markierungen der Zustand-Instanz [23]

Der aktivierte Zustand bleibt solange ACTIVATED, bis eine externe Mikro-Transitions-Instanz aktiviert wird. Ist dies der Fall, wird die Markierung der Zustand-Instanz auf CONFIRMED gesetzt, und der nachfolgende Zustand-Instanz wird auf ACTIVATED gesetzt. Alle alternativen Zustand-Instanzen, die mit dem nun CONFIRMED markierten Zustand-Instanz verbunden sind, werden als SKIPPED markiert, falls diese keine weiteren eingehenden Transitionen-Instanz mit einer WAITING Markierung haben.

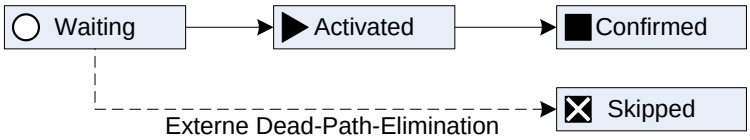


Abbildung 2.11: Zustandsübergänge [12]

Die Abb. 2.11 zeigt den Markierungswechsel der Zustands-Instanzen und Abb. 2.12 zeigt einen Mikro-Prozess-Instanz mit entsprechenden Markierungen.

Mikro-Schritt-Instanz

Ähnlich wie bei den Zustand-Instanzen gibt es verschiedene Markierungen für die Mikro-Schritt-Instanzen (vgl. Tabelle 2.3). Am Anfang ist jede Mikro-Schritt-Instanz mit WAITING

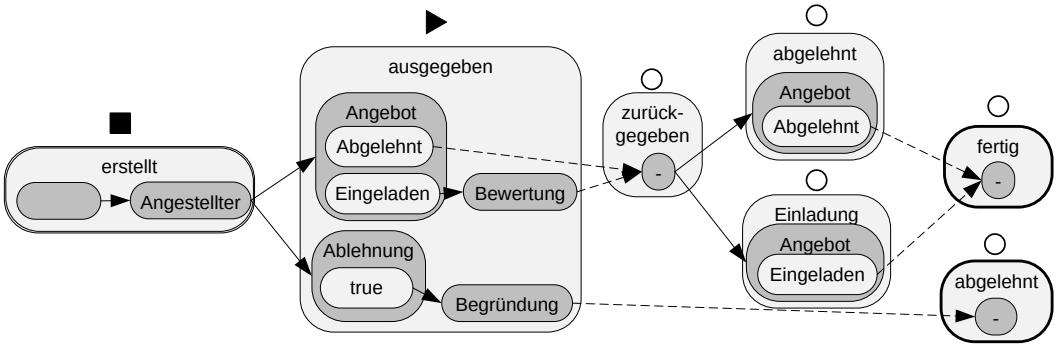


Abbildung 2.12: Beispielmarkierungen der Zustand-Instanz einer Prozess-Instanz

markiert. Sobald eine Zustand-Instanz auf ACTIVATED gesetzt wird, ändern sich alle Mikro-Schritt-Instanzen der Zustand-Instanz auf READY. Die mit der Mikro-Transition-Instanz verbunden Mikro-Schritt-Instanz, welche die Zustand-Instanz aktiviert hat, wird auf ENABLED gesetzt. Die Laufzeitumgebung ändert die Markierung von ENABLED auf ACTIVATED, wenn die Attribut-Instanz der Mikro-Schritt-Instanz einen Wert enthält. Falls der Mikro-Schritt-Typ kein Attribut-Typ zugewiesen ist, wird die Mikro-Schritt-Instanz automatisch auf ACTIVATED gesetzt.

$\text{Schritt}_{\text{Mark}}$	**Erklärung**
WAITING	Der Zustand, in dem der Schritt liegt, wurde noch nicht aktiviert.
READY	Der Zustand wurde aktiviert, der Schritt kann im Moment aber noch nicht erreicht werden.
ENABLED	Der Schritt kann durch Eingabe eines Wertes aktiviert werden.
ACTIVATED	Der Schritt ist aktiviert. Ein Wert wurde eingegeben.
BLOCKED	Der eingegebene Wert muss verändert werden, da er in keinen Wertebereich fällt, der durch Werteschritte definiert wurde.
UNCONFIRMED	Der Schritt wurde besucht, aber der dazugehörige Zustand wurde noch nicht komplett durchlaufen.
CONFIRMED	Der Zustand wurde verlassen und der Schritt wurde aktiviert.
BYPASSED	Der Schritt liegt in einem aktivierten Zustand, der Pfad auf dem er liegt wurde aber abgewählt.
SKIPPED	Der Schritt wurde nicht aktiviert.

Tabelle 2.3: Markierungen der Mikro-Schritt-Instanz [23]

Nachdem die Mikro-Schritt-Instanz auf ACTIVATED gesetzt wurde, wird geprüft, ob sie Werte-Schritt-Instanzen enthält. Falls dies so ist, überprüft die Laufzeitumgebung, ob mindestens eine Werte-Schritt-Instanz erfüllt wird. Wenn nicht, wird die Mikro-Schritt-Instanz als BLOCKED markiert und die Laufzeitumgebung wartet, bis eine Werte-Schritt-Instanz erfüllt ist, um die Mikro-Schritt-Instanz dann als UNCONFIRMED zu markieren. Falls die Mikro-Schritt-Instanz keine Werte-Schritt-Instanzen enthält, wird sie auf UNCONFIRMED gesetzt. Danach wird die nachfolgende Mikro-Schritt-Instanz auf READY gesetzt (vgl. 2.13).

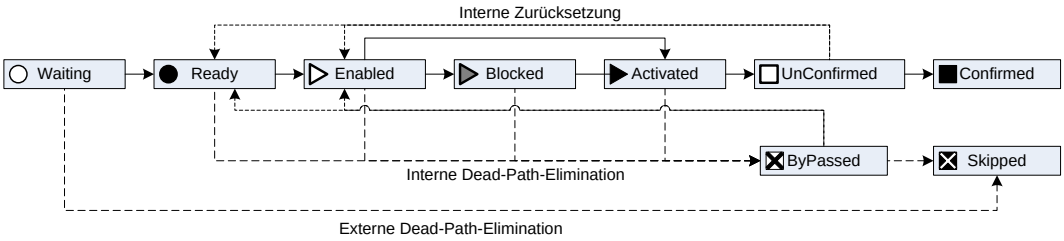


Abbildung 2.13: Zustandsübergänge [12]

Nach jedem Zustandsübergang überprüft die Laufzeitumgebung, ob Mikro-Schritt-Instanzen nicht mehr aktiviert werden können und setzt diese dann auf BYPASSED. Dazu wird ein sogennanter Death-Path Algorithmus eingesetzt. Entscheidend dabei ist, ob eine Transition-Instanz aktiviert wird oder als SKIPPED markiert ist.
Wird der Wert einer Attribut-Instanz geändert, das von einer Mikro-Schritt-Instanz referenziert wird, der schon als UNCONFIRMED markiert wurde, wird eine sogenannte Ausnahmebehandlung durchgeführt. In einer Ausnahmebehandlung wird die erste Mikro-Schritt-Instanz gesucht, deren Attribut-Instanzen geändert wurde. Diese wird auf ACTIVATED und alle nachfolgenden Mikro-Schritt-Instanzen werden auf WAITING gesetzt. Der Grund hierfür ist, dass durch die Änderung des Wertes ein alternativer Pfad an Mikro-Schritt-Instanzen gewählt werden könnte. Danach werden von der Laufzeitumgebung Zustandsübergange durchgeführt, bis dies nicht mehr möglich ist.
Im Beispiel (vgl. Abb. 2.14) könnte die Attribut-Instanz `Angebot` des `Ausgegeben` Zustand-Typs von `Abgelehnt` auf `Eingeladen` geändert werden, und der untere statt des oberen Pfades genommen wird. Die Ausnahmebehandlung gilt nur solange die aktuelle Zustand-Instanz betroffen ist. Angenommen, in der Zustand-Instanz `Ablehnung` würde die Attribut-Instanz `Angebot` geändert, kann die Laufzeitumgebung die Zustand-Instanz nicht

ändern, da dies auch Folgen für die Makro-Prozess-Instanzen und andere Mikro-Prozess-Instanzen hat. Eine Ausnahmebehandlung für einen solchen Fall wäre zu komplex und aus diesem Grund ist das Eingreifen eines Benutzers notwendig.

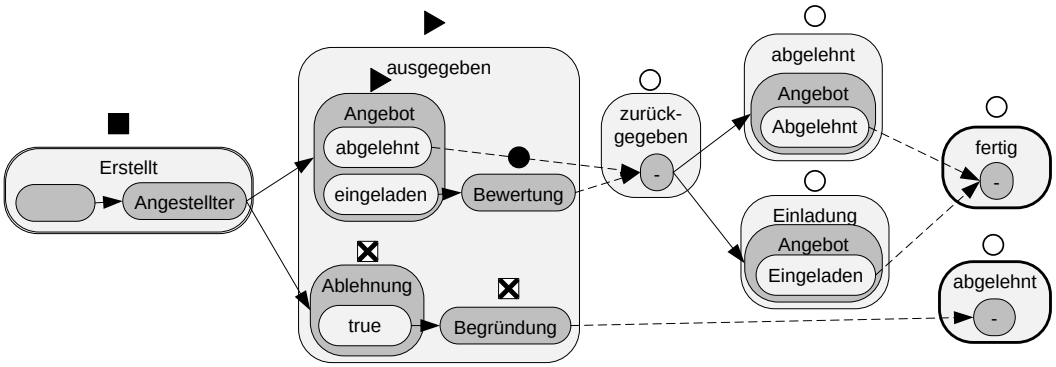


Abbildung 2.14: Zustandsübergänge [12]

Das Beispiel in der Abb. 2.14 zeigt, dass die Zustand-Instanz `Ausgegeben` aktiviert wurde, und die Mikro-Schritt-Instanz `Angebot` auf eine Eingabe des Benutzers wartet. Die Mikro-Schritt-Instanz *Bewertung* steht noch auf READY.

Mikro-Transition-Instanz

Ähnlich wie bei den Mikro-Schritt-Instanzen gibt es verschiedene Markierungen für die Mikro-Transition-Instanzen (vgl. Tabelle 2.4). Am Anfang ist jede Transition-Instanz als WAITING markiert. Sobald die Quelle der Transition-Instanz auf UNCONFIRMED oder CONFIRMED gesetzt ist, wird die Mikro-Transition-Instanz als READY markiert. Ist die Mikro-Transition-Instanz jedoch eine externe explizite Mikro-Transition-Instanz, wird diese auf CONFIRMABLE gesetzt. Erst ein Benutzer kann durch ein Dialogfenster die Markierung auf READY setzen lassen (vgl. 2.15).

Beim Markieren der nächsten Mikro-Schritt-Instanz auf ENABLED oder höher, werden alle entsprechenden Mikro-Transitionen-Instanzen von READY auf ENABLED gesetzt. Danach wird die Mikro-Transition-Instanz mit der höchsten Priorität gesucht, und diese auf ACTIVATED gesetzt. Die anderen Mikro-Transition-Instanzen und deren Mikro-Schritt-Instanzen

$Schritt_{Mark}$	**Erklärung**
WAITING	Die Transition wurde noch nicht aktiviert, kann aber noch aktiviert werden.
CONFIRMABLE	Die Transition muss vom Benutzer weitergeschaltet werden.
READY	Die Transition kann aktiviert werden.
ENABLED	Die Transition wird aktiviert, wenn sie die höchste Priorität besitzt.
ACTIVATED	Die Transition hat die höchste Priorität und ist aktiviert.
UNCONFIRMED	Die Transition wurde aktiviert, der Zustand ist aber noch nicht abgeschlossen.
CONFIRMED	Der übergeordnete Zustand wurde abgeschlossen.
BYPASSED	Die Transition wurde deaktiviert, kann aber durch eine Ausnahmebehandlung noch aktiviert werden.
SKIPPED	Die Transition wurde nicht aktiviert.

Tabelle 2.4: Markierungen der Mikro-Transition-Instanzen [23]

werden als BYPASSED markiert. Wird die Ziel-Mikro-Schritt-Instanz auf CONFIRMED gesetzt, wird auch die Mikro-Transition-Instanz auf CONFIRMED gesetzt.

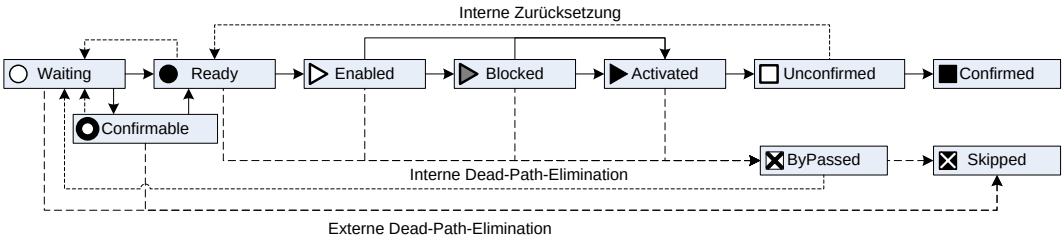


Abbildung 2.15: Zustandsübergänge [12]

2.5.3 Makro-Prozess-Instanz

Die Elemente für die Makro-Prozess-Instanz werden nicht erläutert, da sie nicht Teil dieser Arbeit sind.

2.6 Zusammenfassung

Die hier vermittelten Grundlagen geben einen Überblick über die Komplexität des PHILharmonicFlow-Konzeptes. In diesem Kapitel wurden nur die für diese Arbeit notwendigen Elemente behandelt. Ausführliche Informationen sind in der Dissertation von Vera Künzle [10] zu finden.

3 Technische Grundlagen

In diesem Kapitel werden die technischen Grundlagen erläutert, die zum Verständnis des Prototyps notwendig sind. Elementare Kenntnisse über die Programmiersprache c# [9] oder SQL [26] werden nicht näher erklärt. Als erstes wird das Framework von PHILharmonicFlows vorgestellt und die zugehörige Architektur erläutert. Danach wird die Entwicklungsumgebung erklärt. Sowohl das Modellierungsprojekt als auch das Laufzeitprojekt arbeiten beide in derselben Entwicklungsumgebung, daher muss nur eine vorgestellt werden. Im Anschluss werden die wichtigsten Frameworks und Bibliotheken besprochen, die sowohl in der Modellierungsumgebung als auch in der Laufzeitumgebung vorkommen.
In Abb. 3.1 ist eine kurze Übersicht über das Kapitel zu sehen.

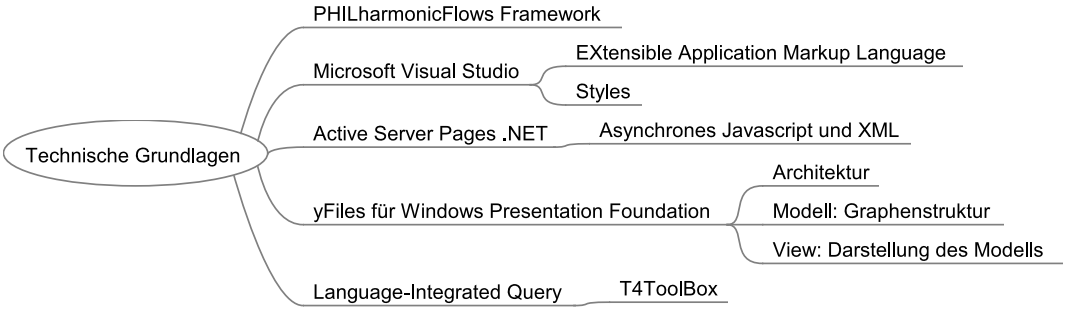


Abbildung 3.1: Übersicht über das Kapitel

3.1 PHILharmonicFlows Framework

Das Fundament des Framework sind zwei *Datenbanken*, eine für das Modell und die andere für die Laufzeitumgebung. Wie in der Einführung erwähnt, ist die reibungslose Einbindung einer Datenbank ein wichtiger Gesichtspunkt des Konzepts. Aus diesem Grund ist die Datenstruktur fast eins zu eins in eine Datenbank übertragbar. Als Datenbanksoftware wird der Microsoft MySQL-Server verwendet. Dank dem SQL Standard ist ein Wechsel auf eine

freie Alternativsoftware jederzeit möglich. Die Microsoft-Variante bietet jedoch den Vorteil, dass sie schon in der genutzten Entwicklungsumgebung Microsoft Visual Studio integriert ist.
Die Abb. 3.2 zeigt die Komponenten des Frameworks und ihr Zusammenspiel.

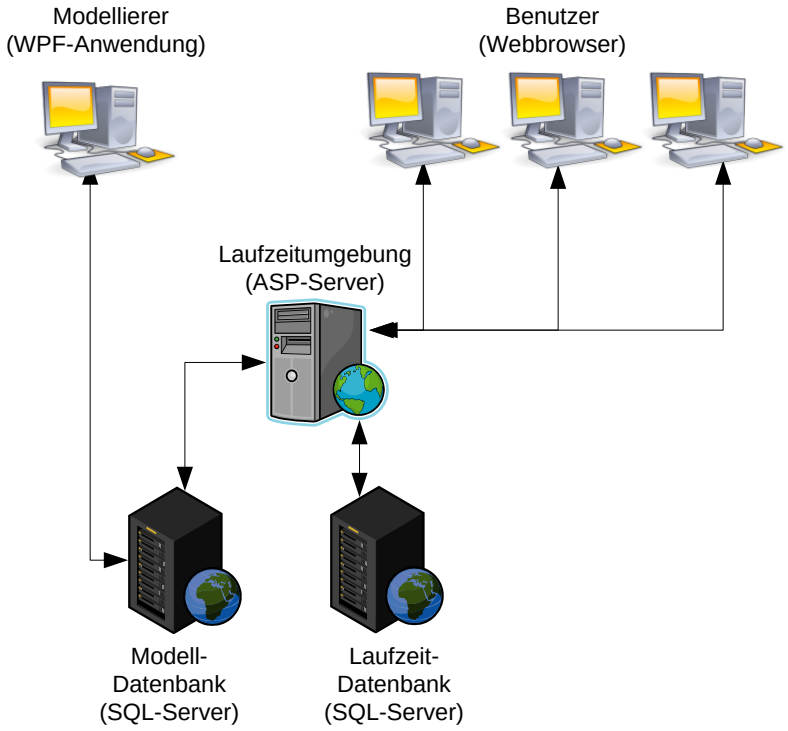


Abbildung 3.2: Architektur des PHILharmonicFlows-Konzepts

Die Modellierungsumgebung speichert und lädt die Modelle direkt in die Modell-Datenbank. Dies geschieht mit SQL-Anfragen und erlaubt daher ein beliebiges Netzwerkprotokoll. Auf eine Mehrbenutzerimplementierung wurde verzichtet. Die Modellierungsumgebung ist eine WPF-Anwendung, welche das yFiles-Framework verwendet, um die Prozess-Graphen zu visualisieren.
Soll ein fertiges Modell in Betrieb genommen werden, so werden in der Laufzeit-Datenbank entsprechende Tabellen angelegt. In dieser Datenbank werden die Markierungen der Prozesse, die Instanzen der Objekt-Typen und die Werte der Attribute abgespeichert. Die Daten-/Prozess-Struktur wird nicht vollständig übertragen, weshalb die Laufzeitumgebung sowohl eine Verbindung zur Laufzeit-Datenbank als auch zur Modell-Datenbank besitzt.

Welche Teile übertragen werden, ist im Kapitel 2.5 beschrieben. Genaue Implementierungsdetails sind in der Diplomarbeit [23] zu finden.
Die Laufzeitumgebung stellt für die Benutzer ein Web-Frontend zur Verfügung. Das Webfrontend ist durch die Active Server Pages realisiert. Dadurch benötigen die Benutzer lediglich einen Webbrowser, um mit der Laufzeitumgebung zu kommunizieren. Diese stellt dann Arbeitslisten, Übersichtslisten und generische Formulare bereit, damit der Benutzer mit dem Prozess interagieren kann.

3.2 Microsoft Visual Studio

Im Frühjahr 2011 wurde entschieden, sich von Eclipse-Framework [4] und der Programmiersprache Java [25] zu trennen und auf die Entwicklungsumgebung Microsoft Visual Studio und C# umzusteigen. Begründet wurde dies damit, dass Visual Studio zu diesem Zeitpunkt den besseren Oberflächeneditor besaß, und sich einfacher und vorlagengetreuer die Oberfläche erstellen lässt. Die Oberfläche wird mittels WPF [1] [9] gestaltet. Die Abb. 3.3 zeigt die Entwicklungsumgebung mit aktivem Oberflächen-Modellierer.
”WPF ist ein Präsentationssystem der nächsten Generation zum Erstellen von Windows-

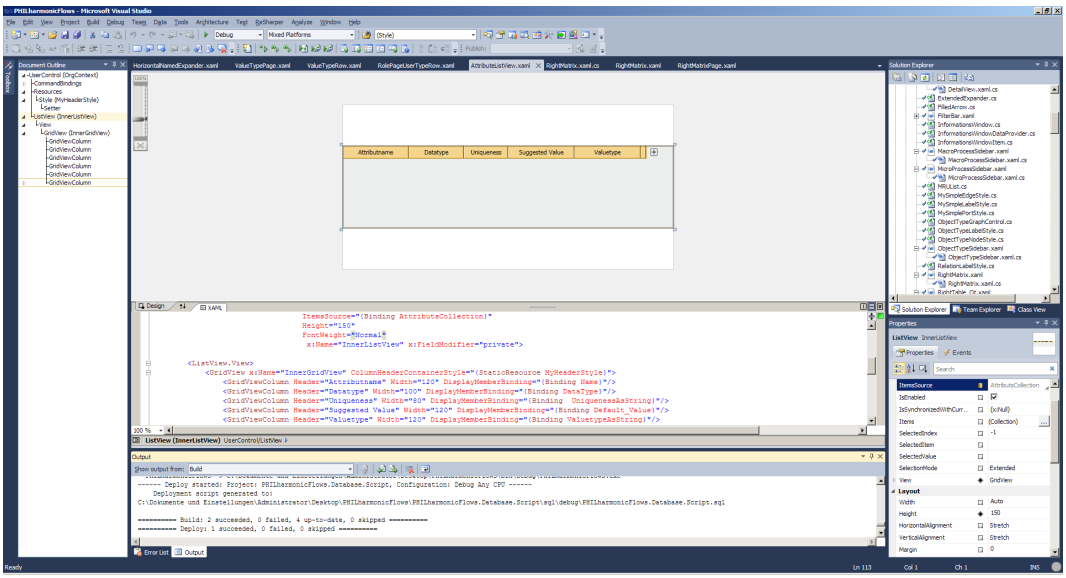


Abbildung 3.3: Microsoft Visual Studio 2010

[1] Windows Presentation Foundation

Clientanwendungen mit einem visuell herausragenden Benutzererlebnis."[2] WPF Anwendungen können als Windows Clientanwendung, als Silverlight-Anwendung oder für den Browser implementiert werden. Letzterer besitzt eine eingeschränkte Funktionalität gegenüber den anderen Varianten. Für Linux, iOS und Android gibt es die .Net-Implementation Mono [30]. Somit läuft PHILharmonicFlows auf allen Plattformen.
Ein weiterer wichtiger Punkt für WPF ist, dass für das Graphen-Framework yFiles [31] eine WPF Implementierung existiert. Die Verbindung yFiles, WPF und Visual Studio verspricht eine einfache und schnelle Oberflächengestaltung.
Für dieses Projekt wird Microsoft Visual Studio 2010 genutzt. Auf eine neuere Version wurde verzichtet, um mögliche Migrationsprobleme zu vermeiden. Als Variante wurde die Ultimate Edition gewählt. Diese bietet unter anderem einen Oberflächeneditor, eine integrierte Versionsverwaltung, eine Datenbankverwaltung und einen Editor für den Microsoft SQL Server. Zusätzlich werden mehrere Code-Analyse-Tools, eine GUI[3]-Validierung und Tools für das Überwachen von Projekten angeboten.

3.2.1 EXtensible Application Markup Language

EXtensible Application Markup Language (XAML) ist eine deklarative Programmiersprache für die Oberflächengestaltung von Anwendungen. XAML ist nach dem Vorbild von XML geformt und unterliegt auch dessen Formungsregeln.
Eingeführt wurde XAML in dem .Net-Framework 3.0 für die Oberflächengestaltung von WPF-Anwendungen. Durch die XML-Struktur und die Nutzung der allgemeinen Common-Language-Runtime (CLR) [16] kann XAML in anderen Anwendungsbereichen eingesetzt werden. Beispiele dafür sind Windows Workflow Foundation [17], Silverlight [18] oder Windows 8 Apps.
In XAML lassen sich nicht nur Oberflächen gestalten, sondern auch Ereignisse behandeln und dynamisch die Oberfläche ändern (z.B. kann bei einem 'MouseOver'-Ereignis die Hintergrundfarbe verändern werden). Zudem lassen sich CLR-Objekte einbinden und auf deren Attribute zugreifen. Damit können Designer große Teile einer Anwendung erstellen, ohne in einer Programmiersprache zu programmieren.
Ist es nicht möglich, eine gewünschte Eigenschaft in XAML zu bewerkstelligen, kann der Programmcode aufgerufen werden. Dieser wird nicht in der XAML-Datei integriert, sondern

[2] http://msdn.microsoft.com/de-de/library/vstudio/aa970268.aspx
[3] Graphical User Interface

in einer separaten Datei. Dabei ist die Programmiersprache egal, solange diese CLR unterstützt. Die umgekehrte Interaktion ist ebenso möglich. Indem in der XAML-Datei einem Element ein Namen gegeben wird, kann seitens der Programmiersprache darauf zugegriffen werden.
Im Anhang ist ein Codebeispiel für ein einfaches Dialogfeld gezeigt. Die Quelle B.1 ist die XAML-Datei, und die Quelle B.2 die zugehörige Codedatei. Diese Trennung zwischen Aussehen und Code erlaubt es einerseits, Oberflächen für verschieden Aufgaben wiederzuverwenden, oder die Programmiersprache zu wechseln. Andererseits können verschiedene Oberflächen (z.B. für Farbenblinde oder für Kinder) erstellt werden, die auf dieselbe Logik zugreifen.

3.2.2 Styles

Eine Erweiterung von XAML sind die sogenannten Styles. Styles erlauben es, für verschiedene Oberflächenelemente das Aussehen zu ändern. Dabei kann Farbe und Form beliebig geändert werden. Zudem können das Verhalten oder die Ereignisse festgelegt werden (z.B. kann das 'Klick'-Geräusch eines Buttons durch ein anderes Geräusch ersetzt werden). Außerdem lassen sich Elemente durch Styles erweitern (z.B. kann ein Button um einen roten Rahmen erweitert werden, oder eine Liste um einen Sortierbutton).
Die Styles werden entweder lokal in den XAML-Dateien integriert oder global festgelegt. Globale Styles ändern das Aussehen jedes Elements, es sei denn es wird von lokalen Styles überschrieben. Dabei ist zu beachten, dass lokale Styles die globalen nicht ersetzen, sondern überschreiben. Beispiel: Definiert der gobale Style einen roten Rand, und der lokale Style eine blaue Schrift, so ist das Ergebnis ein Element mit rotem Rand und blauer Schrift.

Der Oberflächeneditor von Visual Studio zeigt das aktuelle Aussehen der Oberfläche mit allen angewendeten Styles an. In Abb. 3.4 sieht man den gleichen Button mit und ohne Styles. Das Listing 3.1 zeigt den notwendige Code dazu.

```
1 <StackPanel Margin="0,10,0,0" Orientation="Horizontal"
     HorizontalAlignment="Center">
```

Abbildung 3.4: Links mit und rechts ohne Styles

```
2         <Button Content="Ok" Height="23" Name="buttonOk" Width="75
              " Margin="0 0 5 0" Style="{StaticResource DefaultButton
              }" />
3         <Button Content="Ok" Height="23" Name="buttonOk" Width="75
              " Margin="0 0 5 0" />
4 </StackPanel>
```

Listing 3.1: XAML-Style für Button

3.3 Active Server Pages .NET

Active Server Pages (ASP) [13] ist ein serverseitiges Framework zum Darstellen von dynamischen Webseiten. Dynamisch bedeutet, dass die Webseiten für jeden Aufruf neu generiert werden und an den Browser und den Benutzer angepasst sind. Die Variante .Net besagt, dass die Generierung mit dem .Net-Framework programmiert wird.
Dabei nutzt ASP nur die Möglichkeiten von HTTP und HTML, und benötigt daher keine Plugins, wie z.B. Flash oder Silverlight. Somit ist ASP unabhängig von dem Betriebssystem des Aufrufers.

Ein Unterschied zu anderen serverseitigen Programmiersprachen (z.B. PHP [20]) ist, dass der Entwickler keine HTML- sondern WPF-Seiten erstellt. Der ASP Compiler generiert aus der WPF-Seite eine HTML-Webseite mit JavaScript. Dabei wird der komplette Funktionsumfang des .Net-Frameworks unterstützt. Somit kann eine .Net-Anwendug über ASP in eine Webseite umgewandelt werden.

3.3.1 Asynchrones Javascript und XML

Asynchrones Javascript und XML (Ajax) [29] ist ein Framework, das es ermöglicht, Datenaustausch mit dem Server zu betreiben, ohne die Seite im Browser neu zu laden. Die

Kommunikation läuft über die XMLHttpRequest Schnittstelle des Browsers. Die Schnittstelle schickt eine HTTP-Anfrage an den Server und übergibt Ajax die Antwort, sobald sie eingetroffen ist. Da Ajax nicht auf die Antwort warten muss, wird dies asynchron genannt. Die Antwort enthält kein HTML, sondern XML-Daten oder JSON-Objekte [8]. Ajax entschlüsselt die Antworten und verändert die Webseite, ohne diese neu zu laden.

Das Ajax Control Toolkit Projekt [19] integriert das Ajax Konzept in das ASP.Net-Framework. Dazu wird in die Webseite ein sogenannter ToolScriptManager eingefügt, der die Kommunikation mit dem Server übernimmt, und die Webseite entsprechend verändert. Die Abb. 3.5 zeigt den Unterschied einer Webanwendung mit und ohne Ajax.

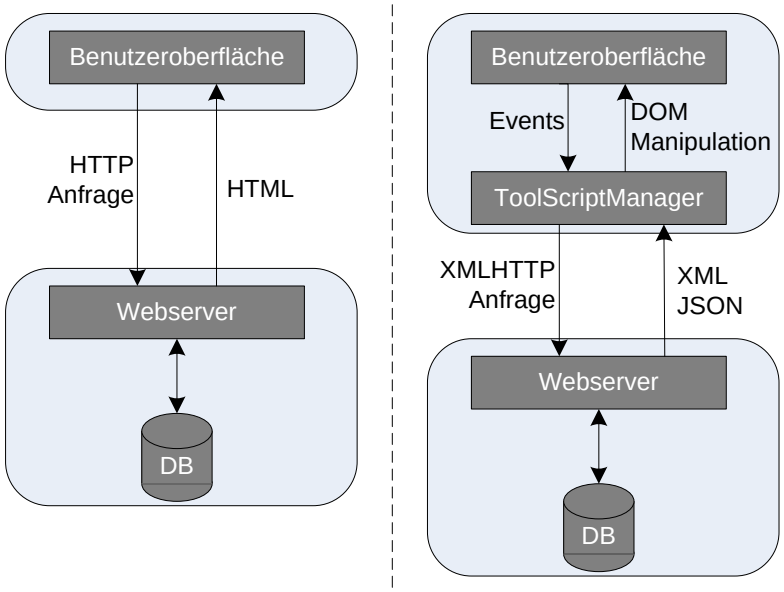


Abbildung 3.5: Webanwendung mit und ohne Ajax

3.4 yFiles für Windows Presentation Foundation

yFiles [31] ist ein Grafik-Framework zum Darstellen von Graphen, das von der Firma yWorks entwickelt und vertrieben wird. Diese bietet sowohl eine Java- als auch eine .Net-Implementation an. Neben den umfangreichen Möglichkeiten zur Visualisierung des Graphen, bietet yFiles Möglichkeiten der Manipulation und Interaktion (z.B. Löschen/Erstellen von Knoten/-

Kanten).
Zudem gibt es Algorithmen zur automatischen Anordnung der Knoten sowie zum Bestimmen des kürzesten Pfades im Graphen. Daneben gibt es fertige Funktionen:

- Hinein-/Herauszoomen aus dem Graphen
- Redo-/Undomöglichkeiten
- Zwischenablage
- Level-of-Detail
- Export des Graphen in XML
- Bildexport
- Druckfunktion

Für den Einstieg in yFiles bietet sich ein Online Developer Guide [32] an. Daneben gibt es Tutorials und Programmbeispiele, die jedoch sehr generisch sind und nur einen kleinen Teil der Möglichkeiten abdecken.
Die Mindestanforderung für yFiles sind das Microsoft .Net-Framework ab Version 3.5 mit dem Servicepack 1 und Microsoft Visual Studio ab Version 2008.

3.4.1 Architektur

Die Anzeigekomponenten von yFiles folgen dem Model-View-Controller (MVC) Paradigma. Diese teilt die Klassen in drei Gruppen auf:

- Model: Graphenstruktur- und Repräsentationsklassen
- View: Darstellungsklassen
- Controller: Benutzersteuerung

Abb. 3.6 zeigt das MVC-Muster und die Übertragung der yFile Architektur. Der *IGraph* hält die Daten und Zustände der Graphenstruktur und ist daher das Modell im MVC-Muster. Die *GraphControl* Klasse ist der Viewer, welcher den Graphen visualisiert. Er stellt ihn in einer WPF-Komponente dar. Diese Komponente fängt die Benutzerinteraktionen auf (z.B. Mausklicks oder Tastatureingaben), und leitet sie an den Controller weiter. Der Controller wird durch das Interface *IInputMode* implementiert. Dieser entscheidet, ob nur die Anzeige geändert oder das Modell manipuliert wird. Bei einer Änderung von LoD [4] wird nur die An-

[4] Level-of-Detail

zeige und nicht das Modell geändert.

Für jede Komponente des MVC-Musters existieren fertige Implementationen, die ein um-

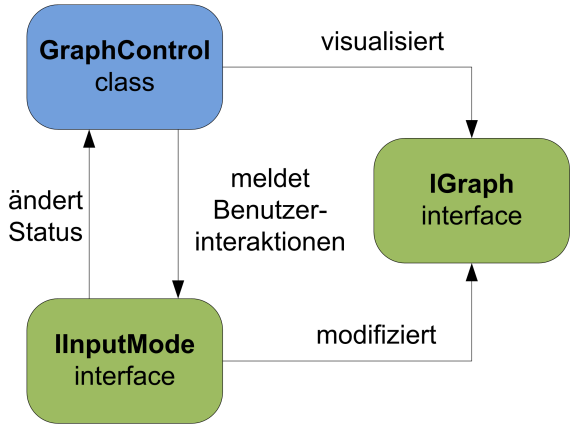


Abbildung 3.6: MVC (nach [32])

fangreiches Grundgerüst bieten. Die Implementierungen hat einige Einstellmöglichkeiten, um das Verhalten der Benutzerinteraktion oder der Darstellung zu beeinflussen. Die fertigen Implementationen sind modular aufgebaut und erlauben das Ersetzen von Modulen. Damit kann das Verhalten der Benutzerinteraktion und die Darstellung angepasst werden. Einige Verhaltensmuster (z.B. Bewegen von Knoten) sind fest implementiert und erfordern eine Neuimplementierung, falls dieses Verhalten geändert werden soll.

3.4.2 Modell: Graphenstruktur

Die zentrale Klasse für die Graphenstruktur ist die Implementation des *IGraph*-Interfaces. Sie ist Container, Factory und Manager in einem. Alle Abfragen oder Manipulationen des Modells werden an den *IGraph* gestellt, der dazu ausreichend Methoden bietet. Dies bedeutet auch, dass kein Modellelement (*ModelItems*) außerhalb des *IGraphen* existiert.
Neben den klassischen Modellelementen wie Kanten (*Edges*) und Knoten (*Nodes*) gibt es noch *Labels*, *Ports* und *Bends*. In Abb. 3.7 sind die Beziehungen der Klassen zueinander dargestellt.

Bends sind Teilabschnitte einer Kante. Dadurch ist es möglich, statt nur gerader Verbindungen auch Ecken zu modellieren. *Ports* sind Start-(*Sourceport*) und Endpunkte (*Targetport*) von Kanten. Knoten werden nicht direkt, sondern über die Ports miteinander verbunden.

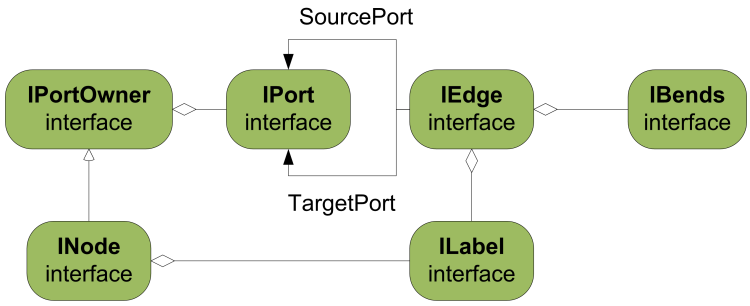
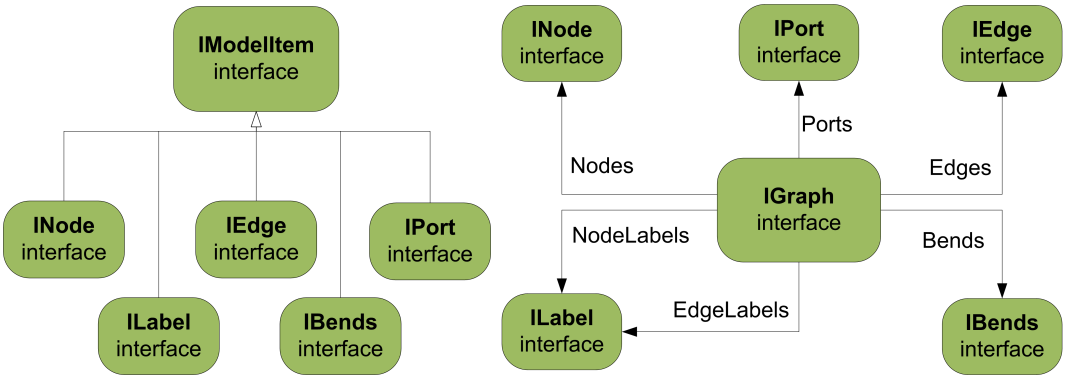


Abbildung 3.7: Beziehungen (nach [32])



(a) Vererbungsdiagramm nach [32] (b) Graphenstruktur nach [32]

Abbildung 3.8: Struktur des Graphenmodells

Andere Modellelemente (z.B. *Labels* oder *Nodes*), können auch Ports besitzen, deswegen referenzieren Ports auf das *Portowner*-Interface, welches die Elemente besitzen. Durch *Labels* können Texte im Modell visualisiert werden. *Labels* können relativ zu einem Modellelement erzeugt werden.
Abb. 3.8(a) zeigt die Vererbungshierarchie und Abb. 3.8(b) die Datenstruktur des *IGraph*-Containers.

3.4.3 View: Darstellung des Modells

Zur Darstellung des Modells hat yFiles keine neuen Elemente implementiert, sondern setzt auf die vorhandenen Oberflächenelemente von WPF auf. *ILabels* werden durch Textboxen, *INodes* durch Rechteck-Elemente und *IEdges* durch Linien-Elemente dargestellt. Auf diese Elemente hat der Entwickler keinen Zugriff. Alle Ereignisse dieser Elemente werden vom

GraphControl aufgefangen, in Modell-Elemente umgesetzt und so an den *IInputMode* weitergeleitet (vgl. Abb. 3.9).

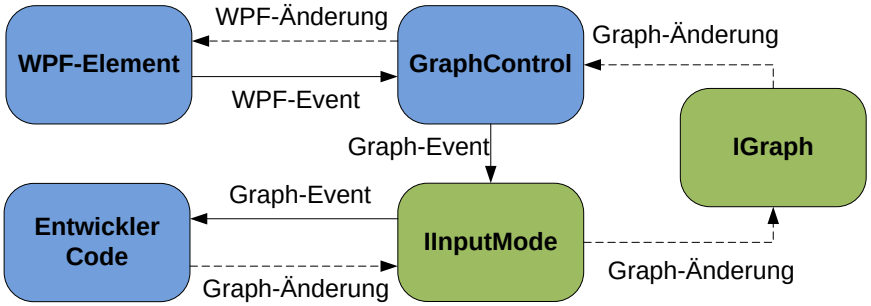


Abbildung 3.9: Verlauf der Events vom Auslöser zum Entwickler

Der *IInputMode* bietet dem Entwickler *Listener*, um die *Events* aufzugreifen und darauf zu reagieren. Somit wird die Struktur der Darstellung vor dem Entwickler verborgen.
Die Visualisierung des Graphen basiert auf WPF-Elementen, so hat der Entwickler die Möglichkeit, die Darstellung nach den Regeln der WPF abzuändern. Dazu können entweder XAML-Styles oder eine Implementierung des *StyleControler*-Interface verwendet werden. In der Interface-Variante bekommt der Entwickler die *IModelItem*-Elemente übergeben und muss einen *Canvas-Container* zurück geben. In diesen Containern können Steuerelemente (z.B. Comboboxen) eingefügt und mit Handlern versehen werden.

Der Quellcode 3.2 ist ein einfaches Beispiel, um die Darstellung eines Labels zu verändern. In dem Beispiel wurde um das Label ein abgerundetes Rechteck eingefügt, und der Text des Labels ist der Name des Objekts.

```
1 <Style x:Key="RelationMaxCardLabelTemplate" TargetType="
     y:LabelControl" >
2         <Setter Property="Template">
3         <Setter.Value>
4                 <ControlTemplate TargetType="y:LabelControl">
5                         <Border CornerRadius="5"  MinWidth="10">
6                         <TextBlock Text="{Binding Path=Name}"
                             FontSize="13"/>
```

```
                        </Border>
                </ControlTemplate>
        </Setter.Value>
        </Setter>
</Style>
```

Listing 3.2: XAML-Style für Labels

Dasselbe Ergebnis kann durch die Implementierung der SimpleAbstractLabelStyle-Klasse erreicht werden (vgl. Quellcode 3.3). Diese Klassen implementiert das ILabelStyle-Interface.

```
protected override CanvasContainer CreateVisual(ILabel label,
    IRenderContext renderContext){
        //Casten des Objekts, welches im Tag gespeichert ist
        var relation = label.Tag as relation;

        //Erstellen des Containers
        var container = new CanvasContainer();
        //Rahmen erstellen
        var border = new Border{CornerRadius = new Thickness(5),
            MinWidth=10,};
        //Textfeld erstellen
        var textBlock = new TextBlock{text = relation.Name,
            FontSize = 13,};
        //Textfeld in den Rahmen, Rahmen in den Container
        border.Child=textBlock;
        container.Add(border);

        //Ausrichten
        ArrangeByLayout(container, label.Layout, true);
        return container;
}
```

Listing 3.3: Interface-Style Implementation für Labels

3.5 Language-Integrated Query

Die Bibiothek *Language-Integrated Query*[5] (Linq) ist ein Programmierkonzept von Microsoft, das es ermöglichen soll, SQL-Abfragen besser in die Programmiersprache zu integrieren. Der klassiche Weg einer SQL-Abfrage ist, zunächst einen Abfrage-String zu erstellen, diesen an eine Datenbank-Verbindungsklasse zu übergeben und als Ergebnis ein zweidimensionales Array zu erhalten. Linq ermöglicht es, anstatt einem String einen Programmcode zu schreiben, der von der Entwicklungsumgebung validiert werden kann. Quelltext 3.4 zeigt eine solche Linq-Abfrage an das Microsoft Northwind Datenbankbeispiel.

```
1 Northwnd db = new Northwnd(@"c:\northwnd.mdf");
2
3 // Query for customers in London.
4 IQueryable<Customer> custQuery =
5     from cust in db.Customers
6     where cust.City == "London"
7     select cust;
```

Listing 3.4: Linq-Abfrage nach [15]

Um Linq nutzen zu können, müssen die Tabellen der Datenbank als Klassen implementiert sein. Der Programmierer arbeitet nun nicht mehr mit SQL-Strings und Arrays, sondern auf Objekten, und kann somit die Vorteile der Objektorientierung nutzen. Zudem überwacht Linq die Referenzen zwischen den Objekten und passt die Referenz des anderen Objekts an, falls bei einem der beiden die Referenz geändert wird. Bei einem Commit werden automatisch alle Werte und referenzierte Objekte geprüft und Veränderungen in die Datenbank übertragen und ggf. auch neue Zeilen in die Tabellen eingefügt. Falls eine Datenbankregel verletzt wird (z.B. ein fehlender Fremdschlüssel) wird der Commit abgebrochen, zurückgesetzt und eine Fehlermeldung geworfen.
Da die Klassen spezielle Annotationen benötigen, werden ab der Version Microsoft Visual Studio 2008 entsprechende Werkzeuge mitgeliefert, die solche Klassen aus bestehenden Datenbanken erzeugen.
Dieses Konzept wurde auf einfache Listen und Arrays erweitert. Zusätzlich werden an-

[5] sprachintegrierte Abfrage

dere Datenstrukturen (z.B. XML[6], JSON[7]) oder Webservices (z.B. Amazon Booksearch) unterstützt. Um die verschiedenen Varianten des Linq zu unterscheiden und dennoch eine einheitliche Benennung zu wahren, werden alle Varianten Linq-to-*Variante* genannt.

3.5.1 T4ToolBox

Ein Nachteil bei Linq-to-SQL ist jedoch, dass Veränderungen manuell an der Datenbank, als auch in den Klassen erfolgen müssen. Diese manuelle Änderung ist bei einem umfangreichen Fall zeitaufwändig und fehleranfällig. Falls dann das Klassenmodell nicht zur Datenbank passt, kann Linq die Daten nicht lesen/schreiben und gibt eine generische Fehlermeldungen aus. Aus diesem Grund versuchen die Softwareentwickler, nur eine Seite zu ändern und die andere automatisch anzupassen. Meist wird die Datenbankseite geändert und das Klassenmodell wird automatisch angepasst. Dies wird *database-driven* (Datenbank getrieben) genannt [24].
Ein Nachteil dieser Methode ist, dass modifizierter Programmcode durch das generierte Klassenmodell überschrieben wird und dadurch nachträgliche Modifizierungen verloren gehen. Dazu kommt, dass Eigenschaften der Objektorientierung, wie Sichtbarkeit oder Vererbung nicht berücksichtigt werden, da dies eine Datenbank im Regelfall nicht kennt.

T4ToolBox verfolgt eine andere Methode, die sogennante *model-driven* Methode (Modell getrieben). Hier wird aus der Datenbank ein Klassendiagramm erzeugt, an dem gearbeitet wird. Veränderungen an dem Klassendiagramm werden dann sowohl auf die Datenbank, als auch auf die generierten Klassen übertragen. In diesem Klassendiagramm können objektorientierte Eigenschaften, wie Vererbung oder Sichtbarkeit, berücksichtigt werden. Die Abb. 3.10 zeigt einen Ausschnitt des Klassendiagramms für PHILharmonicFlows.

3.6 Zusammenfassung

Die Technologie ASP kommen nur in der Laufzeitumgebung zum Einsatz, während das yFiles-Framework nur in der Modellierungsumgebung benutzt wird. Linq wird in beiden Um-

[6] Extensible Markup Language
[7] JavaScript Object Notation

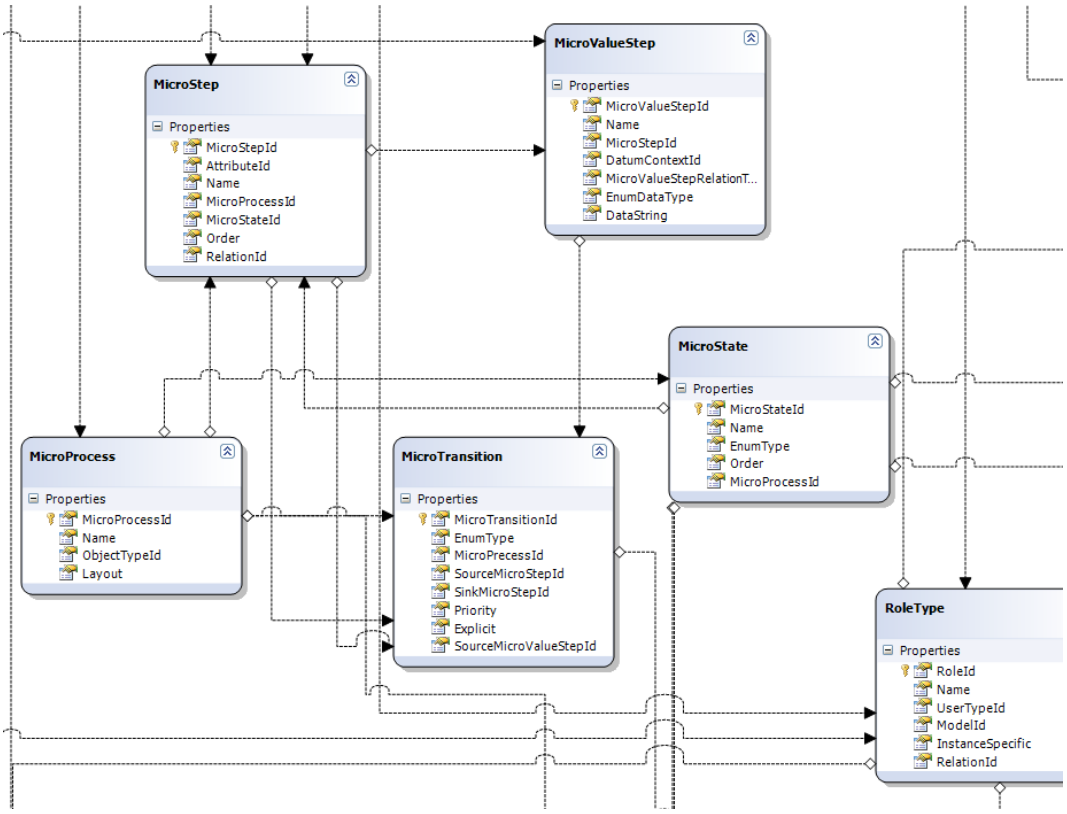


Abbildung 3.10: Ausschnitt des Klassendiagramms von PHILharmonicFlows

gebungen eingesetzt. Die T4ToolBox ist eine für diesen Prototyp neue Technologie. Grundsätzlich wird keine neue oder experimentelle Technologie eingesetzt, sondern auf bewährte Software zurückgegriffen.

4 Technische Analyse der Software

Nachdem die fachlichen und technischen Grundlagen behandelt sind, werden in diesem Kapitel die technischen Komponenten des aktuellen Prototyps analysiert und fehlende Komponenten aufgezeigt. Dabei werden Punkte aufgelistet, die während der Einarbeitungszeit, als auch während der Implementierung aufgefallen sind. Es wird jedoch nur auf die relevanten Komponenten für diese Arbeit eingegangen. Die restlichen fehlenden Komponenten werden im Kapitel 8 der Zusammenfassung besprochen. Ziel dieses Kapitels ist es selbstverständlich, nicht andere Entwickler zu kritisieren, sondern die offenen Arbeitsfelder und Schwierigkeiten in der Software zu zeigen. Viele Arbeitsfelder blieben offen, weil sie nicht Teil des Schwerpunktes des jeweiligen Entwicklers waren.
In Abb. 4.1 ist eine kurze Übersicht über das Kapitel zu sehen.

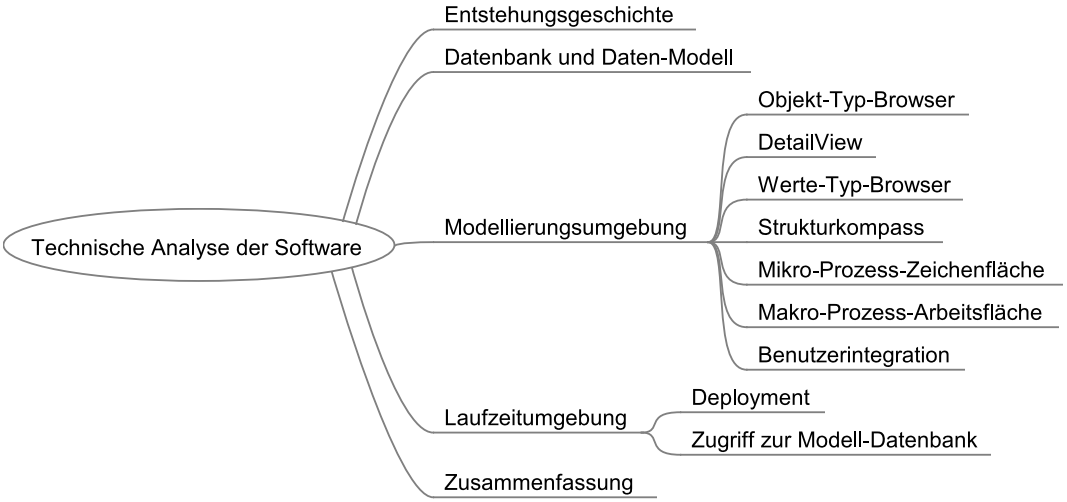


Abbildung 4.1: Übersicht über das Kapitel

Der Prototyp ist von verschiedenen Entwicklern in verschiedenen Programmiersprachen erstellt worden. Jeder hat seinen persönlichen Programmierstil mit einem anderen Programmierschwerpunkt. Deswegen ist der Prototyp geprägt von verschiedenen Codestilen

und offenen Arbeitsfeldern. Ein Schwerpunkt dieser Arbeit ist es, diese Arbeitsfelder zu schließen.
Dieses Kapitel gibt zuerst einen kurzen Rückblick über die Historie des Prototyps und zeigt die Altlasten aus dessen Entstehungsphase. Danach wird die vorhandene Datenbank und das Datenmodell erläutert, da diese sowohl für die Modellierungsumgebung, als auch für die Laufzeitumgebung wichtig sind. Anschließend wird die Modellierungsumgebung vorgestellt. Dieser Abschnitt wird wegen seiner Komplexität in mehrere Unterkapitel aufgeteilt. Zum Schluss wird die Laufzeitumgebung besprochen.

4.1 Entstehungsgeschichte

Das Usability-Konzept und Design für den PHILharmonicFlows Prototyp ist unabhängig vom verwendeten Framework entwickelt worden. Der erste Prototyp wurde von dem Diplomanden Andreas Pröbstle [21] für das Eclipse-Framework[4] konzipiert und implementiert. Die Implementierung in dem Eclipse-Framework wurde später verworfen, da die Oberflächengestaltung mit den Java-Werkzeugen zu komplex war und nicht zu dem gewünschtem Ergebnis führte. Deshalb folgte ein Wechsel auf das .Net-Framework, da yFiles-Framework eine bessere Möglichkeit der Oberflächengestaltung ermöglicht. Daraufhin implementierte Julian Tiedeken auf Basis der Eclipse-Implementierung den Prototyp in der Programmiersprache C# und WPF neu. Ziel dieser Implementierung war, dass zeitnah ein Grundgerüst entstehen sollte, an dem Studenten weiterentwickeln konnten. Aus diesem Grund waren viele Konstrukte aus der Eclipse Version übernommen, aber nicht ausgearbeitet oder an C# angepasst worden. Ein weiteres Arbeitsfeld in der Implementierung war die nicht voll funktionierende Datenbankanbindung und die nur teilweise gespeicherten Modelle. Das Laden der Modelle wurde bei der Weiterentwicklung nicht behoben, sondern durch einen XML-Im-/Export umgangen.
Parallel zur Erweiterung der Modellierungsumgebung implementierte Stefan Schultz die Laufzeitumgebung. Den Oberflächenentwurf entwickelte Christian Scheb [22]. Die Laufzeitumgebung wurde zunächst unabhängig vom Prototyp des Modellierungsumgebung ausgearbeitet, um paralleles Arbeiten zu ermöglichen. Sie ist die operationale Ausführung der Prozesse. In der aktuellen Version der Laufzeitumgebung werden bisher nur die Mikro-Prozesse unterstützt.

Anforderungen

Die abgeleiteten Anforderungen sind:

- Voll funktionsfähiges Abspeichern der Modelle
- Verbindung der Prototypen
- Beheben der Workarounds durch vollwertige Lösungen

4.2 Datenbank und Daten-Modell

Die Datenbank und die Tabellen wurden so entworfen, dass sie möglichst wenig Speicher belegen. Dies lässt sich daran erkennen, dass die Typgrößen der Spalten gering gehalten sind. Außerdem wurde auf die Originaltreue der Namen geachtet, so dass Spaltennamen wie `Order` entstanden. `Order` ist ein reserviertes Wort in SQL und gibt die Ordnung der Mikro-Prozess-Schritte im Modell an.
Mit der Neuentwicklung auf C#-Basis wurden die Klassen mit Hilfe von Linq-SQL aus der Datenbank generiert. Dabei traten mehrere Fehler auf, (z.B. Linq-SQL kann keine n:m Relationen auflösen). Dadurch konnten keine Daten aus der Datenbank in den Modellierungsumgebung eingelesen werden. Hinzu kamen einige Linq-SQL Bugs, die bei ungünstigen SQL-Abfragen [2] Fehler warfen. In der Folge wurde die Funktionalität der Klassen soweit reduziert, dass keine Fehler mehr auftraten und das Lesen aus der Datenbank wieder funktionierte (entfernt wurde die Rechte, der Makro-Prozess und die Mikro-Prozesse). Diese Änderungen waren für die Bachelor-Arbeit von Hannes Beck zu wenig und aus diesem Grund ist beschlossen worden, auf den Einsatz der Datenbank zu verzichten.
Dieselben Probleme im Linq führten bei der Entwicklung der Laufzeitumgebung dazu, dass auf den Einsatz von Linq verzichtet wurde. Als Ersatz wurden die Daten über eine manuell entwickelte Klasse in die Datenbank geschrieben. Diese Daten wurden in eigens programmierten Klassen gehalten.
So existieren nun drei Klassenmodelle im Prototyp: Das generierte Modell, das von Hannes Beck und das von Stefan Schultz. Bei kleinen Änderungen an der Datenbank, wie z.B. Umbenennen von Spalten, müssen deshalb alle drei Klassenmodelle angepasst werden.
Das nächste Ziel dieser Arbeit war, die Datenbank in die Modellierungsumgebung und in die Laufzeitumgebung einzubinden. Dabei sollte möglichst nur ein Klassenmodell genutzt

werden, damit Änderungen nicht redundant an verschiedenen Stellen vorgenommen werden müssen.

Anforderungen

Die abgeleiteten Anforderungen sind:

- Beheben der Linq-Fehler
- Vereinheitlichung der Klassenmodelle
- Einbindung der Datenbank

4.3 Modellierungsumgebung

In diesem Kapitel werden die einzelnen Komponenten der Modellierungsumgebung betrachtet, Fehler aufgezeigt und fehlende Funktionen beschrieben.
Das Kernelement der Modellierungsumgebung ist die Arbeitsfläche in der Mitte des Fensters. Dort werden alle Informationen visualisiert und manipuliert. Die Arbeitsfläche wird in weitere Teilflächen unterteilt, die entweder zur Steuerung der anderen Teilflächen dient oder Informationen zu ausgewählten Objekten bereitstellt. Wie in Abb. 4.2 zu sehen, ist die Arbeitsfläche unten um eine Detailansicht erweitert worden. Um die Oberfläche übersichtlich zu halten, werden einige Einstellungen durch Dialoge ermöglicht. Wie in Abb. 4.2 zu sehen ist, existieren neben der Arbeitsfläche in der Mitte zusätzlich die Navigationssidebar (kurz Sidebar) auf der linken Seite sowie die Menüleiste.
Im Menü befinden sich die klassichen Funktionen wie ’Projekt öffnen’, ’Speichern’, ’Projekt Schließen’ und ’Programm Beenden’. Funktionen wie ’Projekt Löschen’ oder ’Projekt Deployen’ existieren bisher nicht.
Die Sidebar ist einem Akkordeon nachempfunden [28], d.h. es expandiert immer nur ein Bereich, während die anderen minimiert sind. Wird ein Bereich in der Sidebar ausgewählt, so wechselt die Arbeitsfläche entsprechend dem gewählten Bereich. Die Sidebar enthält auch Platzhalter für nicht implementierte Arbeitsflächen, bei diesen wird nur eine leere Arbeitsfläche geladen. Des Weiteren enthält sie auch Elemente zur Steuerung und Interaktion der Arbeitsfläche, oder stellt Informationen zur Verfügung, die bezüglich der Arbeitsfläche wichtig sind.

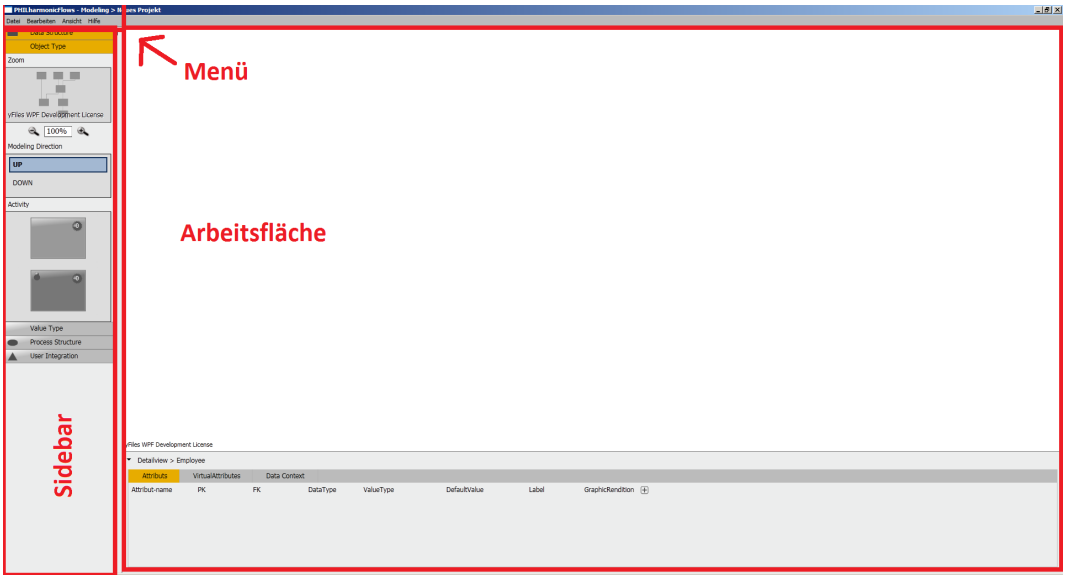


Abbildung 4.2: Objekt-Typ-Browser

In den folgenden Kapiteln werden die für diese Arbeit relevanten Komponenten besprochen. Relevant sind diejenigen, die während der Arbeit verändert wurden.

4.3.1 Architektur

Die Architektur der Modellierungsumgebung (vgl. Abb. 4.3) ist einfach und kompakt gehalten und lässt sich in drei Bereiche einteilen: Oberfläche, Verwaltung und Datenbank.

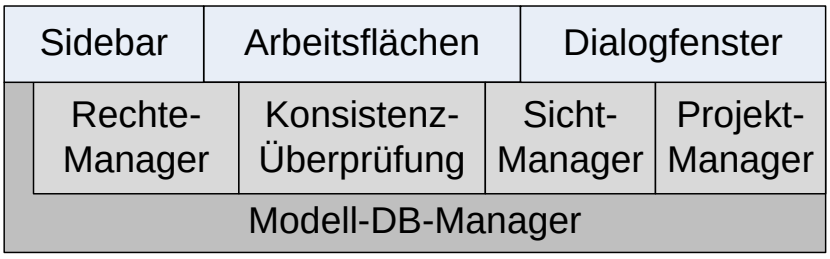


Abbildung 4.3: Architektur der Modellierungsumgebung

Der Zugriff zur Datenbank wird über den `Database-Manager` geregelt. Der Database-Manager bekommt die Verbindungsinformationen zur Datenbank und erstellt über die Linq-

Technologie eine Verbindung. Linq kümmert sich um die SQL-Abfragen, erstellt aus den Tabellen die Klassen-Instanzen und setzt die Referenzen. Das Übertragen der Änderungen in den Instanzen übernimmt Linq. Bis auf wenige Ausnahmefälle (Erstellen des Projekts) interagieren die Verwaltung und die Oberfläche nur mit den Instanzen.

Die Verwaltung besteht aus mehreren Klassen:

- Rechte-Manager (`RightManager`)
- Sicht-Manager (`ViewManager`)
- Projekt-Manager (`ProjectManager`)
- Konsistenz-Überprüfung (`<Type>ConsistyManager`)

Der Sicht-Manager verwaltet die Arbeitsflächen. Zu jeder Arbeitsfläche hält dieser eine Instanz und stellt sie in den Vordergrund, wenn dies gewünscht ist. Auf diese Instanzen kann über 'static'-Methoden zugegriffen werden, um Interaktionen zwischen den Oberflächenelementen zu ermöglichen.
Der Projekt-Manager öffnet, speichert, schließt und deployed das geladene Projekt. Er erhält die Befehle von der Oberfläche, überprüft diese, gibt ggf. eine Warnmeldung aus, (z.B. ob ein Projekt wirklich geschlossen werden möchte) und schickt diese dann an den Datanbank-Manager weiter.
Der Rechte-Manager ermittelt die Rechte und Verantwortlichkeiten einer Rolle zu einem gegebenen Element, oder setzt die Rechte bzw. Verantwortlichkeiten für die Rollen. Der Rechte-Manager wird von der Oberfläche und dem Konsistenz-Manager genutzt.
Der Konsistenz-Manager dient zur Überprüfung der Instanzen. Jeder Typ (z.B. Objekt-Typ) besitzt einen eigenen sogenannten `<Type>ConsistyManager`. Zusammengefasst werden diese als `ConsistyManager` bezeichnet.
Alle Methoden des Rechte-Managers und der Konsistenz-Überprüfung sind als 'static' implementiert und arbeiten ausschließlich mit den Instanzen.

Die Oberflächenelemente (Sidebar, Arbeitsfläche und Dialogfenster) dienen zum Anzeigen von Informationen und zur Interaktion mit dem Modell. Sie holen vom Datenbank-Manager die benötigten Instanzen und zeigen diese an. Änderungen werden direkt an den Instanzen ausgeführt.

4.3.2 Objekt-Typ-Browser

Der `Objekt-Typ-Browser` wird über ”Datastructur->Objecttype” in der Navigationsbar aufgerufen und dient zur Darstellung der Datenstruktur in Form eines Graphen. Grafisch ist die Komponente bereits vollständig (vgl. 4.4), während es im Bereich Interaktion und Layout noch offene Arbeitsfelder gibt. Ein Interaktions-Arbeitsfeld betrifft die Bearbeitung

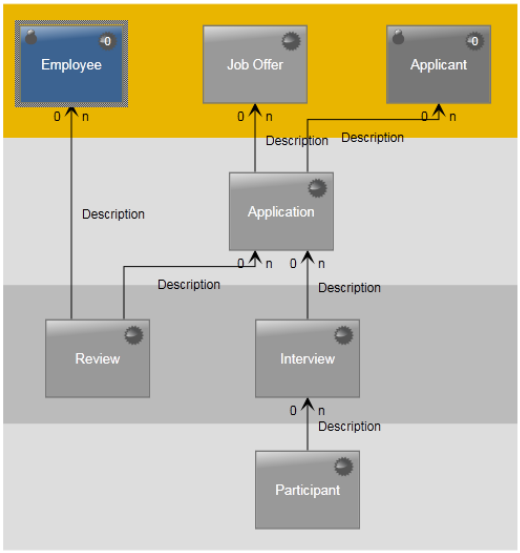


Abbildung 4.4: Objekt-Type-Browser

der Kardinalitäten und der Name einer Relation. Diese werden in einem Dialogfeld editiert, das über einen Doppelklick auf den Pfeil geöffnet wird. Aus Usability-Sicht ist eine direkte Bearbeitung im Graphen selbst erwünscht.
Ein Layout-Arbeitsfeld ist die Anordnung der Knoten. Aus Darstellungssicht ist die aktuelle Anordnung nicht optimal. Sie wird über einen vorgefertigten yFiles-Layout-Algorithmus ausgeführt, der ersetzt oder angepasst werden sollte, um die Darstellung zu verbessern. Ein weiteres offenes Arbeitsfeld ist das Layout der Labels. Sie überlagern sich oder liegen auf Linien. Eine Anpassung des Layout-Algorithmus ist wünschenswert.

Anforderungen

Die abgeleiteten Anforderungen sind:

- Direkte Bearbeitung der Kardinalitäten und der Namen der Relationen im Graphen

- Verbesserung des Layout-Algorithmus des Graphen
- Verbesserung des Layout-Algorithmus der Labels

4.3.3 DetailView

Die `DetailView` ist eine Erweiterung des Objekt-Typ-Browsers und erscheint, sobald ein Objekt-Typ ausgewählt wurde. In der `DetailView` werden die Attribute des ausgewählten Objekt-Typs angezeigt. Die Anzeige enthält für den Entwickler wichtige Informationen (z.B. Primärschlüssel oder Fremdschlüssel (vgl. Abb. 4.5)). Es wird dem Benutzer keine Möglichkeit gegeben, die Attribute zu bearbeiten oder zu löschen. Mit Hinzukommen des Werte-Typs (siehe Kapitel 4.3.4) ist eine Spalte für den Verweis auf den Werte-Typ gewünscht, falls das Attribut einen besitzt.

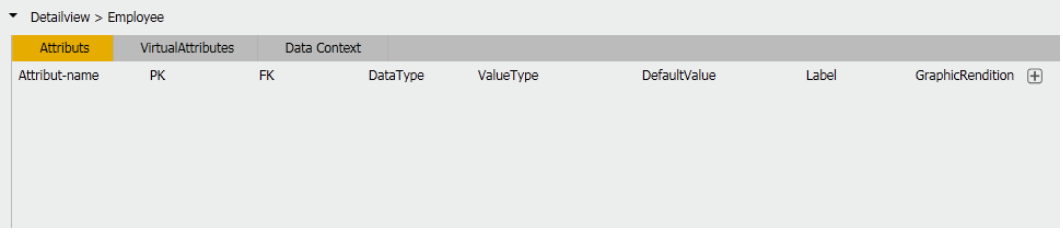


Abbildung 4.5: Screenshot der `DetailView`

Anforderungen

Die abgeleiteten Anforderungen sind:

- Anpassung der Spaltenelemente der Liste
- Optionen für Bearbeiten und Löschen von Attributen
- Hinzufügen der Verweise auf Attributen von Werte-Typen

4.3.4 Werte-Typ-Browser

Der `Werte-Typ-Browser` wird über ”Datastructur->Valuetype” in der Navigationsbar aufgerufen. Diese Arbeitsfläche wird zum Verwalten der Werte-Typen verwendet. Die Werte-

Typen werden in einer Akkordeon-Struktur angezeigt. Um das Suchen der Werte-Typen zu vereinfachen, ist die Arbeitsfläche um eine `Filterbar` erweitert (vgl. 4.6).

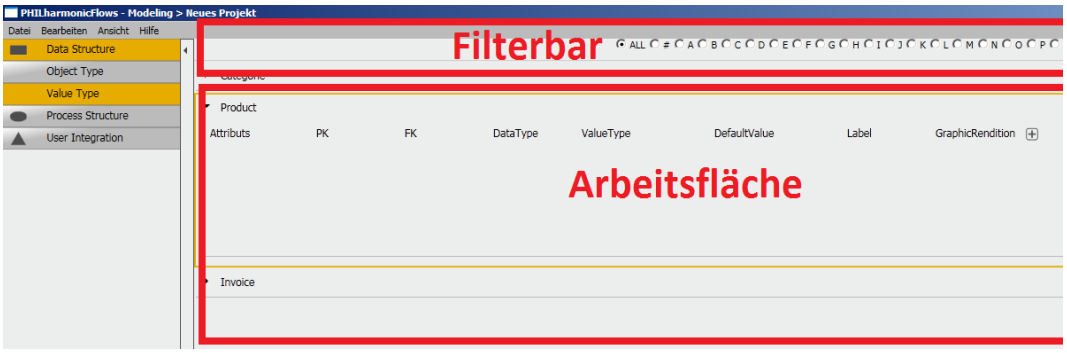


Abbildung 4.6: Werte-Typ-Browser

Die `Filterbar` filtert die Werte-Typen entweder nach den Anfangsbuchstaben oder nach einem komplexeren String. Ob es sich bei der komplexeren Suche um einen reinen String-Vergleich oder möglicherweise einer Expression handeln soll, ist offen. Die Abb. 4.7 zeigt die Radiobuttons für die Suche nach Anfangsbuchstaben, das Textfeld und den Such-Button für die komplexe Suche. Daneben existiert ein Button zum Erstellen eines neuen Werte-Typs.

Abbildung 4.7: Filterbar

Im Akkordeon ist der ausgewählte Werte-Typ aufgeklappt und orange umrandet (Abb. 4.8), während die anderen geschlossen sind. Die Übersichtstabelle ist aus dem Detailview entnommen worden und besitzt dieselben Eigenschaften und Arbeitsfelder. Ein Schwerpunkt dieser Arbeit ist die Implementierung der Werte-Typen. Dazu gehört auch, dass in der Modellierungsumgebung Instanzen dieser Typen angelegt werden sollen.

Anforderungen

Die abgeleiteten Anforderungen sind:

- Anpassung der Spaltenelemente der Liste
- Optionen für Bearbeiten und Löschen

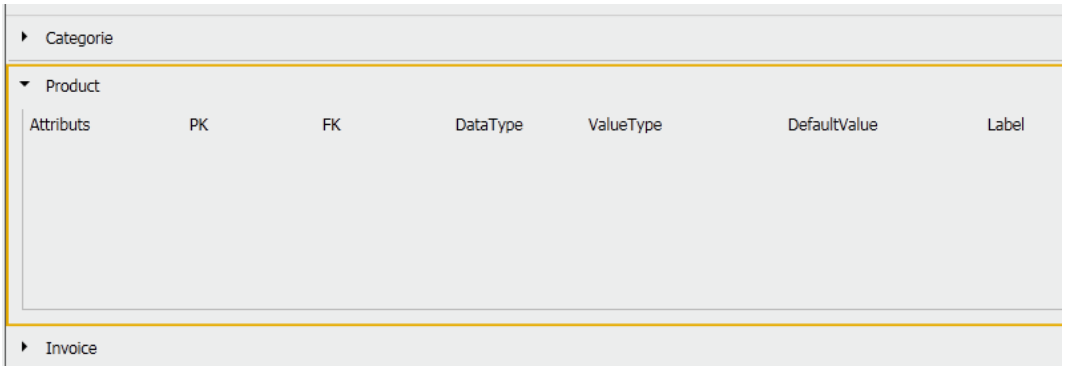


Abbildung 4.8: Screenshot der Werte-Type Ansicht

- Hinzufügen einer Instanzverwaltung

4.3.5 Strukturkompass

Der Strukturkompass ist eine Erweiterung der Arbeitsfläche und dient zum Auswählen eines Objekt-Typs (vgl. Abb. 4.9). Wenn ein Objekt-Typ ausgewählt wird, ändert sich die Arbeitsfläche. Im Fall der Mikro-Prozess-Arbeitsfläche wird der Mikro-Prozess des ausgewählten Objekt-Typen angezeigt. In der übernommenen Implementierung ist der Strukturkompass für die Mikro-Prozess Arbeitsfläche eingepasst. Da der Strukturkompass in verschiedenen weiteren Arbeitsflächen und Dialogen (z.B. Makro-Prozess Arbeitsfläche oder Benutzerintegration) zum Einsatz kommt, muss dieser erst modularisiert werden (d.h. die spezielle Einpassung muss durch eine allgemeine Schnittstelle ersetzt werden). Zusätzlich dazu sind weitere grafische Anzeigen wünschenswert, (z.B. das infärben der Kanten oder Knoten). Daneben werden wegen eines kleinen Fehlers Werte-Typen im Strukturkompass angezeigt. Sie sollen nicht angezeigt werden und müssen deshalb herausgefiltert werden.

Anforderungen

Die abgeleiteten Anforderungen sind:

- Filtern der Werte-Typen aus den Objekt-Typen
- Modularisieren
- Hinzufügen weiterer grafischer Anzeigen

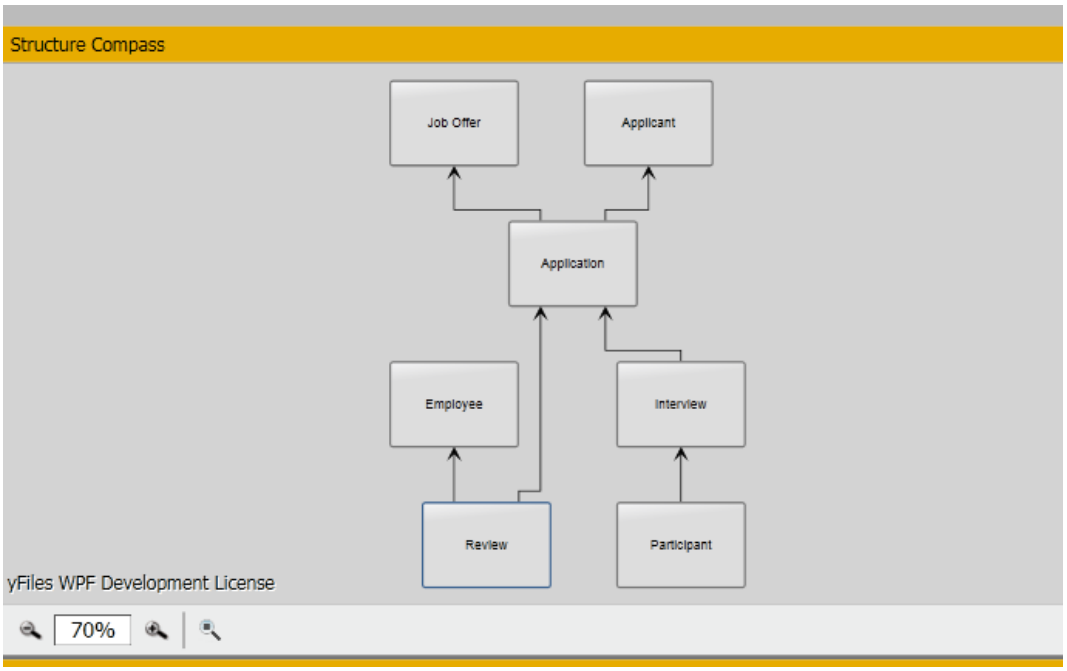


Abbildung 4.9: Strukturkompass

4.3.6 Mikro-Prozess-Zeichenfläche

Die Mikro-Prozess-Zeichenfläche ist das Kernelement der Mikro-Prozess Arbeitsfläche und dient zur Darstellung und Manipulationen des Mikro-Prozess-Graphen (vgl. Abb. 4.10).

Grafisch ist die Zeichenfläche vollständig implementiert. Bei der Bedienung sind noch kleinere Änderungen erwünscht:

- Die Werte (z.B. Zustandsname) können nicht direkt im Graphen geändert werden, sondern werden über Dialogfenster bearbeitet.
- Verschiedene Komponenten (z.B. Benutzerintegration) sind nur Mock-Ups, die es zu implementieren gilt.
- Mit der Implementation der Werte-Typen muss das Werte-Schritt-Dialogfenster angepasst werden, damit es die Werte-Typen unterstützt. Abb. 4.11 zeigt das aktuelle Dialogfenster.
- Das Klassenmodell im Hintergrund muss durch das Linq-Klassenmodell ersetzt werden (siehe Kapitel 7).

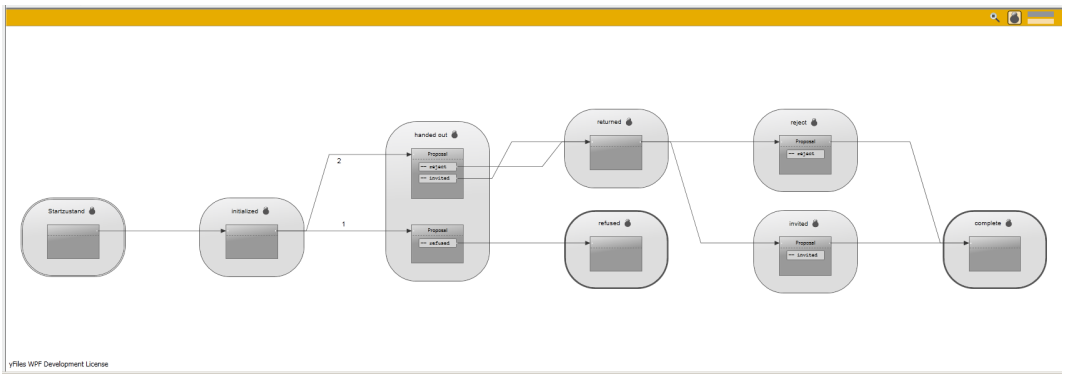


Abbildung 4.10: Mikro-Prozess-Zeichenfläche

Anforderungen

Die abgeleiteten Anforderungen sind:

- Direkte Bearbeitung der Werte im Graphen
- Implementieren der Benutzerintegration
- Anpassung der Dialogfenster
- Beheben kleinerer Fehler
- Ersetzen des verwendeten Klassenmodells durch die Linq-Version

4.3.7 Makro-Prozess-Arbeitsfläche

Die Erstellung des Makro-Prozesses ist bisher nicht implementiert. Ein Schwerpunkt dieser Arbeit ist die grafische und technische Implementierung der Makro-Prozesse. Dies wird in Kapitel 6 genauer behandelt.

Anforderungen

Die abgeleiteten Anforderungen sind:

- Erstellen der Zeichenfläche
- Erstellen der Navigation für Makro-Prozesse

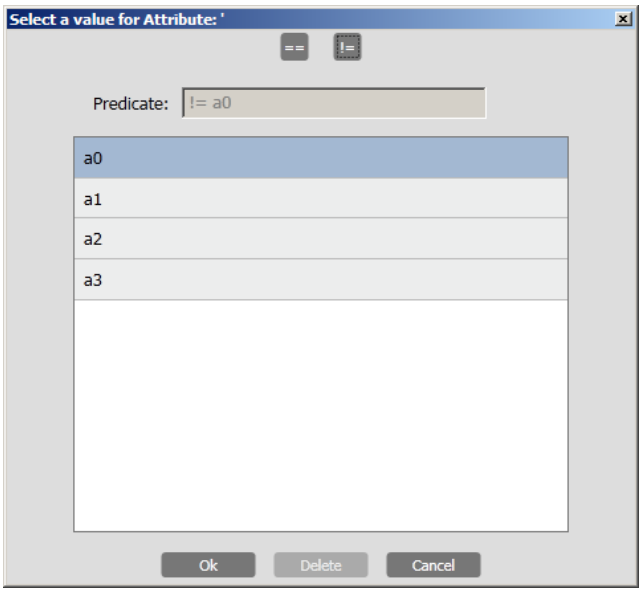


Abbildung 4.11: Aktuelles Werte-Schritt-Dialogfenster

- Erstellen der technischen Grundlagen für die Makro-Prozesse
- Erstellen der Algorithmen für die Strukturprüfung

4.3.8 Benutzerintegration

Im übernommenen Prototyp sind die notwendigen Arbeitsflächen nicht implementiert worden. Ein Schwerpunkt dieser Arbeit ist die grafische und technische Implementierung der Benutzerintegration. Dies wird in Kapitel 7 genauer behandelt.

Anforderungen

Die abgeleiteten Anforderungen sind:

- Erstellen der Zeichenfläche der Rechtematrix
- Erstellen der Zeichenfläche der Rollenverwaltung
- Erstellen der Verwaltung für Benutzer-Instanzen
- Erstellen der technischen Grundlagen für die Rechteverwaltung

- Erstellen der Algorithmen für die Strukturprüfung

4.4 Laufzeitumgebung

Die Laufzeitumgebung ermöglicht bisher die Verwaltung und Ausführung der Mikro-Prozesse über ein Webfrontend. Die Struktur der Laufzeitumgebung wurde in der Diplomarbeit [23] wie folgt beschrieben:

> Die Grundlage bilden der Laufzeit-Datenbankmanager (`RuntimeDBManager`) und der Typ-Datenbankmanager (`TypeDBManager`), welche mit der Typ-Datenbank und mit der Laufzeit-Datenbank des Prozessmodells kommunizieren und die Daten verwalten. Auf die Typ-Datenbank wird nur lesend zugegriffen, um Typ-Informationen zu einzelnen Instanzen abzurufen. Es folgen der Prozess-Manager (`Process-Manager`), der Formular-Manager (`FormManager`), der Rechte-Manager (`PermissionManager`), der Aktivitäts-Manager (`Activity-Manager`) und der Laufzeit-Manager (`RuntimeManager`). Sie sind für die Verwaltung von Prozess-Instanzen, die Erzeugung von Formularen, die Benutzerintegration und Rechteverwaltung, die Verwaltung von Aktionen auf Instanzen und für die Daten, die zur Laufzeit anfallen, verantwortlich. Im Modell-Manager (`ModelManager`) wird das aktuell geladene Prozessmodell verwaltet.

Die Abb. 4.12 zeigt die Hierarchie der Manager.

Im folgenden Kapitel werden die Komponenten der Laufzeitumgebung angesprochen, die während der Arbeit angepasst wurden.

4.4.1 Deployment

Unter Deployment wird der Vorgang der Übertragung eines Modells in die Laufzeitumgebung verstanden, um es ausführbar zu machen. Diese Funktion wird manuell in der Entwicklungsumgebung aufgerufen. Wünschenswert ist eine Nutzung in der Modellierungsumgebung oder der Laufzeitumgebung.

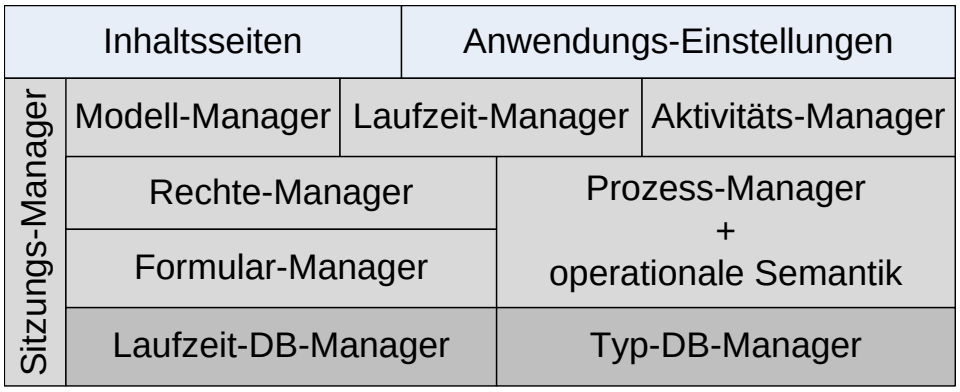


Abbildung 4.12: Komponenten der Laufzeitumgebung (nach [23])

Dabei wird eine neue Datenbank erzeugt, welche die Laufzeitdaten für ein Modell beinhaltet. In dieser Datenbank werden für jeden Objekt-Typ eine eigene Tabelle generiert. Diese Tabelle enthält für jedes Attribut eine Spalte mit dessen Namen. Als Datentyp der Spalte wird immer char(n) verwendet und bei Bedarf in der Laufzeitumgebung in den entsprechenden Originaldatentyp geparst, und zusätzlich werden Spalten für die Mikro-Prozess-Daten angelegt. Für die Mikro-Schritte wurde ein anderes System entwickelt. Alle Mikro-Schritte werden in einer Tabelle zusammengefasst, und mit einer Referenz zu ihrer Objekt-Instanz ausgestattet, um die Zuordnung zu einem Mikro-Prozess nicht zu verlieren. Die Abb. 4.13 zeigt eine Datenbankstruktur nach dem Deployen.

Der Algorithmus zum Erstellen der Datenbank und Tabellen nutzt die vorhandenen Linq-

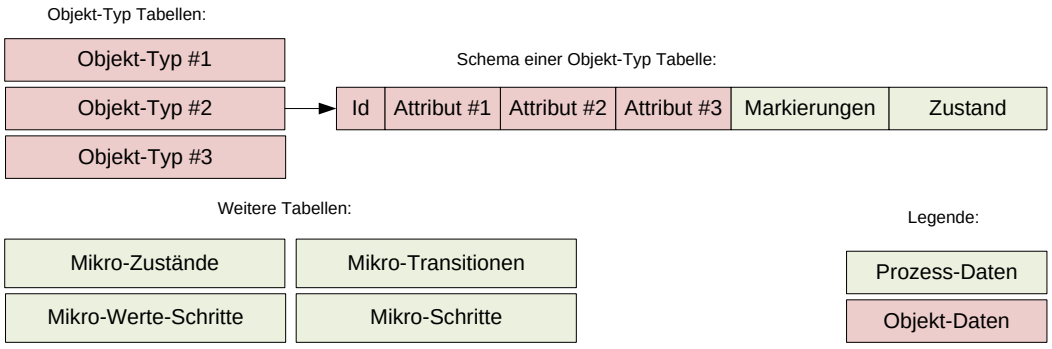


Abbildung 4.13: Tabellenstruktur der Laufzeit-Datenbank (nach [23])

Klassen, um die Datenstruktur des Modells zu laden. Danach baut der Algorithmus einen SQL-String zusammen, der die benötigten Tabellen erstellt. Der Algorithmus prüft nicht, welche Tabellen schon vorhanden sind (d.h. bei Änderungen am Modell werden die Tabellen gelöscht und anschließend neu erstellt).

Anforderungen

Die abgeleiteten Anforderungen sind:

- Die Funktion 'Deployment' für den Nutzer aktivieren
- Anpassen an Linq-Klassenmodell
- Anpassen des 'Deployment'-Algorithmus für eine Schema-Evolution

4.4.2 Zugriff zur Modell-Datenbank

Der Zugriff auf die Datenbank wird über die Klasse `TypeDBManager` abgewickelt. Dieser erstellt SQL-Anfragen an die Datenbank und speichert die Ergebnisse in einfachen Klassen ab. Diese enthalten nur die Werte und Fremdschlüssel der Typen. Ein einfaches Navigieren über Referenzen wie bei Linq-Klassen ist nicht möglich, sondern muss über Schlüssel/Wert Datenstrukturen navigiert werden.
Die Klassen der Laufzeitumgebung sind unabhängig von der aktuellen Tabellenstruktur der Modell-Datenbank. Ein Nachteil ist, dass bei Änderungen an der Tabellenstruktur der Modell-Datenbank die SQL-Anfragen von Hand geändert werden müssen.

Anforderungen

Die abgeleiteten Anforderungen sind:

- Anpassen an Linq-Klassenmodell
- Nutzung der Linq-Technologie

4.5 Zusammenfassung

Zusammengefasst ergeben sich folgende Anforderungen:

- Anbindung der Datenbank
  - Voll funktionsfähiges Abspeichern der Modelle
  - Verbindung der Prototypen
  - Beheben der Workarounds
  - Beheben der Linq-Fehler
  - Vereinheitlichung der Klassenmodelle
  - Einbindung der Datenbank
- Objekt-Typ-Browser
  - Direkte Bearbeitung der Kardinalitäten und der Name der Relationen im Graphen
  - Verbesserung der Layout-Algorithmus des Graphen
  - Verbesserung der Layout-Algorithmus der Labels
  - Anpassung der Spaltenelemente der Liste
  - Optionen für Bearbeiten und Löschen
  - Hinzufügen der Verweise auf Werte-Objekt-Typen
- Werte-Typ-Browser
  - Anpassung der Spaltenelemente der Liste
  - Optionen für Bearbeiten und Löschen
  - Hinzufügen einer Instanzverwaltung
- Mikro-Prozess-Arbeitsfläche
  - Direkte Bearbeitung der Werte im Graphen
  - Implementieren der Benutzerintegration
  - Anpassung der Dialogfenster

  - Beheben kleinerer Bugs
  - Ersetzen des verwendeten Klassenmodells durch die Linq-Version
- Makro-Prozess-Arbeitsfläche
  - Erstellen der Zeichenfläche
  - Erstellen der Navigation für Makro-Prozesse
  - Erstellen der technischen Grundlagen für die Makro-Prozesse
  - Erstellen der Algorithmen für die Strukturprüfung
- Benutzerintegration
  - Erstellen der Zeichenfläche der Rechtematrix
  - Erstellen der Zeichenfläche der Rollenverwaltung
  - Erstellen der Verwaltung für Benutzer-Instanzen
  - Erstellen der technischen Grundlagen für die Rechteverwaltung
  - Erstellen der Algorithmen für die Strukturprüfung

5 Integration der Prototypen

Nach der Klärung des Stands der Prototypen wird nach Lösungswegen für die Anforderungen gesucht. Dieses Kapitel behandelt die Lösungsfindung zur Integration der Laufzeit- und Modellierungsumgebung. Die Integration ist abgeschlossen, sobald beide Prototypen Modelle aus der Datenbank laden und in die Datenbank speichern. Dazu gehört auch die Modellierung, das Deployment und das Ausführung von Modellen. Dies ist in den Prototypen soweit vollständig implementiert, einzig die Verbindung zur Datenbank und die Anpassung der Prototypen fehlen.
Zunächst wird die Anbindung der Modellierungsumgebung besprochen, anschliessend die Anbindung der Laufzeitumgebung.
Abb. 5.1 zeigt eine kurze Übersicht über das Kapitel.

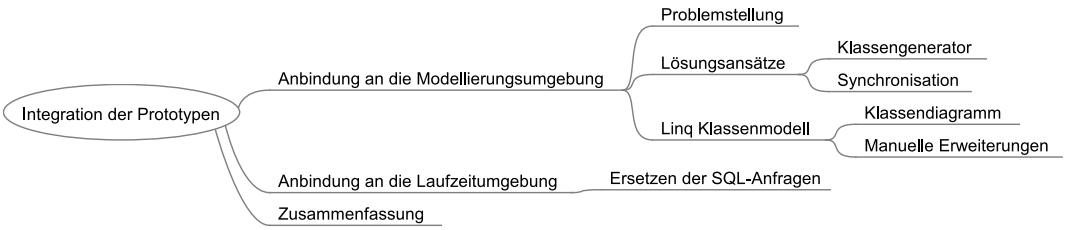


Abbildung 5.1: Übersicht über das Kapitel

5.1 Anbindung der Modellierungsumgebung an die Datenbank

Wie in Kapitel 4.2 erklärt, wird die Verbindung zur Datenbank über die Linq Technologie realisiert. Mit dem Klassengenerator von Linq wurden Klassen generiert, welche die Tabellen der Datenbank symbolisieren. Diese Klassen waren jedoch wegen Fehler im Linq defekt und die Funktionalität wurde soweit eingeschränkt, dass nur die Datenstruktur abgespeichert wurde (siehe Kapitel 4.2). Die Mikro-Prozesse, die Makro-Prozesse oder die

Benutzerintegration waren somit nicht funktionsfähig. Damit dies für den Entwickler deutlich wurde, waren die Methoden durch Exceptions abgesichert (vgl. 5.1).

```
1 [LinqAssociation(Name = "Attribute_AttributPermission", Storage =
     "attributPermissions", ThisKey = "AttributeId", OtherKey = "
     AttributeId")]
2 public EntitySet<AttributPermission> AttributPermissions{
3         get { throw new Exception("Unsupported Method");}
4         set { throw new Exception("Unsupported Method");}
5 }
```

Listing 5.1: Linq-Code für den Objekt-Typ

Nachfolgend werden zunächst die verschiedenen Lösungsansätze erfasst und gegeneinander abgewogen.
Im Anschluss wird die Umsetzung der Lösung vorgestellt.

5.1.1 Problemstellung

Das Problem lässt sich in ein Verbindungsproblem und ein Synchronisierungsproblem aufteilen.

Das Verbindungsproblem befasst sich mit der Frage, wie die Daten aus der Datenbank gelesen und in einem Klassenmodell im Hauptspeicher gehalten werden. Im Hinblick auf eine mögliche Veränderung der Struktur des PHILharmonicFlows-Konzeptes und der daraus folgenden Änderung der Datenstruktur, muss sich eine solche Änderung auf die Datenbank und das Klassenmodell übertragen lassen. Hierbei ist der Einsatz von automatischen Werkzeugen von Vorteil, da sie bei Änderungen dem Entwickler Arbeit ersparen.
Ein weiterer Punkt ist die einfache Nutzung der Klassen für die Entwickler. Wenn die Klassen mit Referenzen auf andere Klassen ausgestattet sind, kann der Entwickler einfacher durch das Modell navigieren, als wenn diese Klassen nur Primärschlüssel besitzen. Bei einer reinen Primärschlüssel-Navigation ist jedoch problematisch, dass neue Objekte noch keinen Wert für den Primärschlüssel besitzen, da sie noch nicht in die Datenbank übertragen wurden.

Das Synchronisationsproblem befasst sich mit der Frage, wie die Datenstruktur auf Veränderungen reagiert und diese Veränderung in die Datenbank gelangen. Da die Datenstruktur des PHILharmonicFlows-Konzeptes sehr verwoben ist, müssen Veränderungen an vielen Stellen im Modell erfolgen. Zum Beispiel betrifft das Löschen eines Objekt-Typ-Attributs sowohl andere Objekt-Typen, als auch Mikro-Schritte oder die Benutzerintegration.
Weiter ist die Reihenfolge der Änderungen zu beachten. Wenn ein neuer Objekt-Typ erstellt und geändert wird, bevor er in die Datenbank übertragen wurde, muss zuerst der Objekt-Typ in die Tabelle eingefügt werden, bevor er geändert werden kann.

Eine Lösung muss diese Fragen zufriedenstellend beantworten und in überschaubarer Zeit umsetzbar sein.

5.1.2 Lösungsansätze

Wegen der Vorteile der Linq-to-SQL Technologie (z.B. automatische Generierung von SQL-Anfragen) ist beschlossen worden, diese weiterhin zu benutzen. Linq übernimmt die Kommunikation zwischen Klassenmodell und Datenbank. Hierfür gibt es drei verschiedene Lösungsansätze:

a) Manuelles Erweitern der vorhanden Klassen.

b) Neugenerierung der Klassen mit Hilfe des Generators

c) Neugestaltung der Datenbank mit modellgetriebenem Ansatz

Das Synchronisationsproblem kann von Linq nicht gelöst werden, sondern erfordert ein Konzept des Entwicklers. Grundsätzlich dürfen die Daten in der Datenbank nur geändert werden, wenn das Modell gespeichert wird. Das Hauptproblem dabei ist, dass drei Modellebenen existieren:

- Datenbank
- Linq-Klassen
- yFiles Graphen

Linq übernimmt die Synchronisierung zwischen Datenbank und Linq-Klassen. Die Synchronisierung zwischen Linq-Klassen und Graphen muss der Entwickler übernehmen. Um dies umzusetzen, stehen mehrere Lösungsansätze zur Verfügung:

1) Löschen des Modells in der Datenbank und neue Klassen erzeugen
2) Linq-Klassen und Graphen vergleichen und Klassen anpassen
3) Während der Modifizierung des Graphen auch die Linq-Klassen anpassen

Klassengeneration

Ein guter Lösungsweg soll einfach und zugleich flexibel genug sein, um auf zukünftige Veränderungen reagieren zu können. Dabei sollen Einarbeitungszeit und Fehleranfälligkeit für nachfolgende Entwickler gering sein. Die Abb. 5.2 zeigt die drei Ansätze.

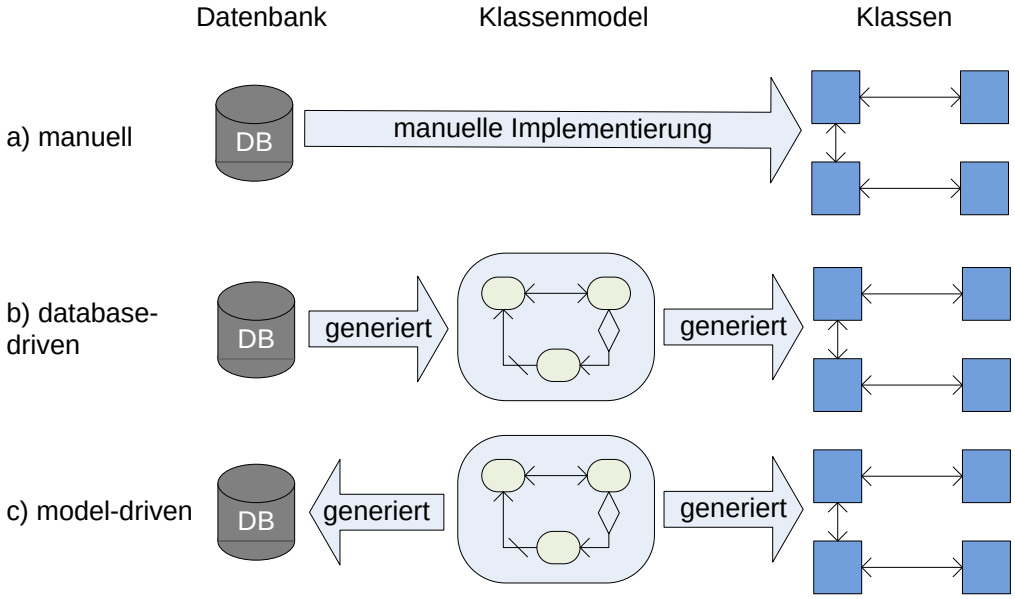


Abbildung 5.2: Drei Ansätze

a) Manuelle Erweiterung bedeutet, dass alle Änderungen von Hand erledigt werden müssen. Dies hat den Vorteil, dass keine Generatoren konfiguriert werden müssen, oder auf Problemfälle für die Generatoren geachtet werden muss. Dadurch ist eine flexible Anpassung möglich, da ein Programmierer bei Problemen Lösungswege erarbeiten muss, was ein Script nicht beherrscht. Zudem können Architektur-Muster oder objektorientierte Eigenschaften leichter umgesetzt werden.
Nachteilig ist, dass der Entwickler sich in Linq einarbeiten muss, bevor er Änderungen am

Code machen kann. Dies bedeutet auch, dass menschliche Fehler in den Code gelangen können.

Fazit: Der manuelle Lösungsweg ist nicht einfach, dafür aber flexibel bei Klassenstruktur oder Problemen. Zukünftige Veränderungen sind nur mit Aufwand umsetzbar. Einarbeitungszeit und Fehleranfälligkeit ist hoch.

b) Neugenerierung der Klassen mit Hilfe des database-driven Ansatzes bedeutet, dass die Veränderungen nur auf der Datenbankseite gemacht werden müssen. Dafür ist auf die Möglichkeiten des Klassengenerators zu achten und Zeit in die Konfiguration des Generators zu investieren. Zudem gehen objektorientierte Eigenschaften (z.B. Vererbung) verloren.
Ein Vorteil für nachfolgende Entwickler ist, dass der Klassengenerator nur bei Problemen modifiziert werden muss. Bei Änderungen der Datenstruktur muss nur die Datenbank geändert werden, was nur Einarbeitungszeit in die Vorschriften des Klassengenerators und in die Datenbank benötigt. Dies reduziert die möglichen menschlichen Fehler.

Zusammenfassend lässt sich sagen, dass der Lösungsweg des database-driven Ansatzes einfach, jedoch nicht flexibel ist. Die Einarbeitungszeit und die Fehleranfälligkeit ist jedoch niedrig.

c) Neugestaltung der Datenbank und des Klassenmodells mit Hilfe des model-driven Ansatzes bedeutet, dass sowohl die Datenbank, als auch die Klassenstruktur geändert werden. Der Entwickler muss sein Konzept in ein Klassendiagramm übertragen. Das Erstellen eines Klassendiagramms ist bei einer Softwareentwicklung nach einem Entwicklungsprozess (z.B. Wasserfall [14]) kaum zusätzlicher Aufwand, da dort im Normalfall ein Klassendiagramm vor der Implementierung gefordert wird. Es muss jedoch darauf geachtet werden, wie der Generator das Klassendiagramm umsetzt, damit die gewünschte Struktur in der Datenbank und in den Klassen generiert wird. Ähnlich wie beim database-driven Ansatz muss Zeit für die Konfiguration und Erweiterung des Generators investiert werden. Dafür lassen sich Eigenschaften der Objektorientierung modellieren.
Vorteil für die nachfolgenden Entwickler ist hier, dass der Klassengenerator nur bei Problemen geändert werden muss. Bei Änderungen muss nur ein Klassendiagramm geändert

werden, das leichter zu gestalten, jedoch auch umfangreicher als eine Datenbank ist.

Fazit: Der Lösungsweg des model-driven Ansatzes ist einfacher und flexibler als der database-driven Ansatz. Dafür ist die Einarbeitungszeit etwas höher, aber die Fehleranfälligkeit ist niedrig.

Zusammenfassung

Die Tabelle 5.1 zeigt die Zusammenfassung der drei verschiedenen Ansätze:

Ansatz	Einfachheit	Flexibilität	Einarbeitung	Änderungen	Fehler
a) Manuell	-	++	–	–	–
b) Database-driven	+	0	+	+	+
c) Model-driven	+	+	0	++	+

Tabelle 5.1: Zusammenfassung der Klassengeneration

Der manuelle Ansatz schneidet am schlechtesten ab, da die Flexibilität den Aufwand nicht rechtfertigt. Database-driven Ansatz und model-driven Ansatz sind nahezu gleichwertig, jedoch der model-driven Ansatz bedeutet, dass alle Daten in der Datenbank verloren gehen, oder nur mit Mühe in die neue Datenbank migriert werden können. Aus diesem Grund wird der database-driven Ansatz bevorzugt.

Synchronisation

Ein guter Lösungsweg soll einfach umzusetzen sein und dennoch die Funktionalität von yFiles unterstützen. Wichtige Funktionen sind:

- Erstellen von Objekten
- Modifizieren von Objekten
- Löschen von Objekten
- Undo-/Redo-Funktion

Zwei wichtige Grundkriterien die jeder Ansatz erfüllen muss:

- Das korrekte Abspeichern des Modells

- Das Wiederherstellen des alten Models, falls beim Abspeichern ein Fehler auftrat

Ansätze, die dies nicht erfüllen, werden hier nicht aufgezeigt.

Zusätzlich dazu müssen die Lösungsansätze einen aktuellen Zustand des Modells liefern, auf dem verschiedene Algorithmen (z.B. Strukturanalyse/Validierung des Mikro-Prozesses) arbeiten können. Diese sollten ohne Aufwand die Struktur lesen können.

1) Ersetzen der Datenbankdaten ist der einfachste Ansatz zum Abspeichern der Daten. Beim Speichervorgang werden zunächst das alte Modell aus der Datenbank gelöscht und danach neue Linq-Klassen erzeugt, die in die Datenbank übertragen werden. Die neuen Linq-Klassen werden durch die Graphen von yFiles und durch Hilfsklassen erstellt, welche Daten beinhalten, die nicht inmittelbar zum Graphen gehören. Dies bedeutet, dass die Linq-SQL Klassen nur zum Laden und Speichern des Modelles benötigt werden.
Dieser Ansatz hat den Vorteil, dass die Linq-Bugs beim Synchronisieren mit der Datenbank umgangen werden (siehe Kapitel 3.5).
Während des Modellierens besteht jedoch der Nachteil, dass alle Abfragen und Algorithmen auf yFiles-Graphen arbeiten müssen. Dies bedeutet, dass keine Navigation zwischen den Klassen existiert und an vielen Stellen Klassen gecastet werden, was einen Wechsel der Klasse aufwendig macht.

Eine Alternative zu den Graphen ist, eine Kopie der geladenen Linq-Klassen zu erstellen und alle Veränderungen in dieser Kopie zu synchronisieren. Beim Speichern wird das alte Modell gelöscht und die Kopie in die Datenbank übertragen. Dies verhindert die Nachteile während der Modellierung, dafür ist das Synchronisieren schwieriger. Bei beiden Fällen sind viele Datenbank-Operationen nötig, um ein Modell zu speichern, auch wenn nichts daran geändert wurde.

2) Vergleichen der Datenbank mit dem veränderten Modell ist ein Ansatz, um die Menge von Operationen zu minimieren, die bei reinem Ersetzen entstehen würden. Ähnlich zu dem ersten Ansatz kann das zu modifizierende Modell als Graph mit Hilfsklassen oder als Kopie der Linq-Klassen implementiert werden. Beim Speichern werden die Unterschiede der Kopie zur Datenbank erfasst. Diese Unterschiede müssen in Modifizierungsschritte der Linq-Klassen umgewandelt werden. Wichtig dabei ist, die Reihenfolge so zu wählen, dass

keine Fehler geworfen werden (z.B. wegen fehlender Fremdschlüssel).
Dieser Ansatz hat den Vorteil, dass die Linq-Bugs beim Synchronisieren mit der Datenbank dadurch umgangen werden (siehe Kapitel 3.5).
Während der Modellierung bestehen dieselben Probleme, wie beim ersten Ansatz.

Bei diesem Lösungsweg liegt der Problembereich auf der Logik des Vergleiches und der Reihenfolge der Modifizierungen.

3) Linq bietet die Möglichkeit, die Linq-Klassen mit der Datenbank zu synchronisieren. Dies erspart dem Entwickler einigen Aufwand, jedoch bei Linq-Bugs hat er das Problem, dass er den Bug nur umgehen und nicht lösen kann.
Während der Modellierung bestehen dieselben Probleme wie bei den anderen Ansätzen.

Beim Linq-Ansatz besteht das Problem bei der Synchronisierung zwischen Graphen und Linq-Klassen.

Zusammenfassung
Die Tabelle 5.2 zeigt die Zusammenfassung der Schwierigkeiten der verschiedenen Ansätze: Die Ansätze, das Modell mit Hilfe des Graphen zu verwalten, verringert den Aufwand

Ansatz /Aufwand:	Algorithmen	Synchronisation Datenbank <-> Klassen	Synchronisation Modell <-> Klassen
1) Ersetzen (Graph)	-	0	0
1) Ersetzen (Klassen)	+	+	-
2) Vergleichen (Graph)	-	-	0
2) Vergleichen (Klassen)	+	-	-
3) Linq	+	(+)	-

Tabelle 5.2: Zusammenfassung der Schwierigkeiten

während des Modellierens, erschwert jedoch die Übertragung in die Datenbank und erhöht den Aufwand bei den Algorithmen. Zudem ist es bei den Makro-Prozessen notwendig, auf die Struktur der Mikro-Prozesse zuzugreifen. Deswegen ist eine einfache Navigation innerhalb der Klassen vorteilhafter. Aus diesem Grund werden die synchron gehaltenen Linq-

Klassen bevorzugt.
Der Ansatz des Ersetzens ist nicht gut, da zu viele Datenbank-Operationen nötig sind. In einem Szenario mit einem Internet-Server und großen Modellen wird der Nachteil noch größer. Bei der Auswahl zwischen manuellem Vergleichen und dem automatischem Vergleich von Linq, ist Linq im Vorteil, da keine Entwicklungszeit in die Logik der Übertragung in die Datenbank benötigt wird. Falls sich die Datenbankstruktur ändert, ist eine Modifikation des generierten Codes nicht notwendig. Der Nachteil durch die Linq-Fehler kann nicht abgeschätzt werden, da diese nur bei besonderen Fällen auftreten und Linq regelmäßig von Microsoft aktualisiert wird.

5.1.3 Linq Klassenmodell

Der gewählte Lösungsweg sollte zuerst der database-driven Ansatz sein, in dem Linq die Übertragung zur Datenbank übernimmt. Der Linq-Klassen-Generator konnte jedoch kein lauffähiges Klassen-Modell generieren. Einerseits wurden vom Generator Fremdschlüssel nicht richtig übernommen, was zu Problemen beim Speichern führte, und andererseits waren in der Datenbank einige unnötige Tabellen, die Linq zu integrieren versuchte, was Fehler erzeugte. Zudem traten Linq-Bugs bezüglich der Primärschlüssel auf. Nach erfolgloser Lösungssuche wurde dieser Lösungsweg aufgegeben.

Aus diesem Grund wurde der nun folgende Lösungsweg mit dem model-driven Ansatz gewählt. Es bestand die Möglichkeit, ein Klassendiagramm aus der bestehenden Datenbank zu generieren, jedoch wurde wegen der Probleme beim database-driven Ansatz dies abgelehnt und ein neues Modell wurde generiert. Dazu wird aus den Arbeiten des Konzeptes ([11],[10]) und der ursprünglichen Datenbank als Vorlage ein Klassendiagramm erstellt.

Klassendiagramm

Die Abb. 5.3 zeigt einen Auszug aus dem Klassendiagramm der neuen Datenbank. Beim Erstellen des Klassendiagramms werden die Primärschlüssel nicht mehr ID genannt, sondern jetzt als <Klasse>Id benannt. Andere Klassen, die auf diese Klasse referenzieren, besitzen den Fremdschlüssel <Klasse>Id. Die Abb. 5.4 zeigt eine solche gleiche Benennung. Die Tabelle *MicroValueStep* besitzt den Fremdschlüssel *MicroStepId*, der auf den Primärschlüssel *MicroStepId* der Tabelle *MicroStep* verweist. Bei manchen Abfragen vertauschte

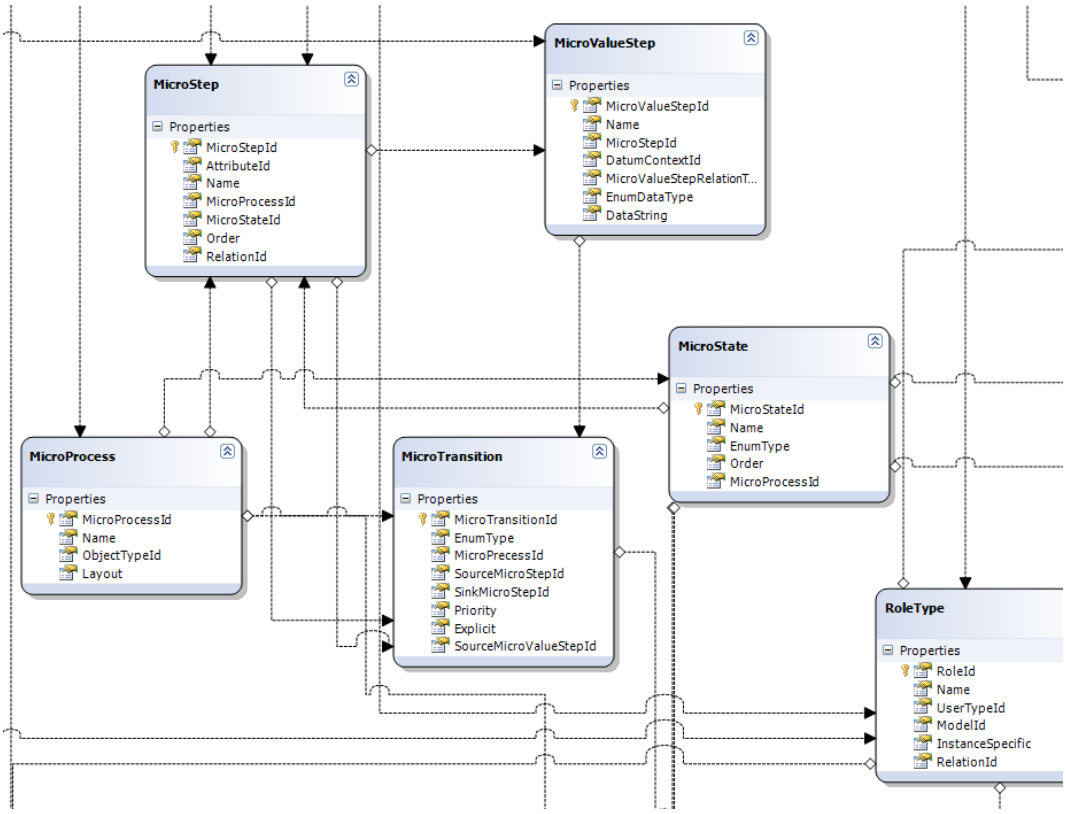


Abbildung 5.3: Auszug aus dem neuen Modell

das Linq-Framework die Namen der Schlüssel. Dies ist bei einer Gleichbenennung von Primär- und Fremdschlüssel kein Problem.

Die Enums sind nicht in das Klassendiagramm eingearbeitet, da sie beim Klassengenerator zu Problemen geführt haben. Um die Enums dennoch für den Entwickler zugänglich zu machen, sind sie als private Int-Attribute implementiert und werden manuell in die Enums umgecastet.

Die Rechte und Pflichten sind als vererbte Klassen modelliert (vgl. Abb. 5.5). Diese werden von Linq in einer einzigen Tabelle gespeichert und zur Unterscheidung mit einem VererbungsTyp-Feld ausgestattet. Diese Modellierung wurde so realisiert, um die Rechte in einer Liste zusammenzufassen, und nahe an der ursprünglichen Datenbank zu sein. Eine Alternative ist, die Rechte in separaten Tabellen zu speichern.

Aus diesem Klassendiagramm werden die Linq-Klassen generiert und die Datenbank geändert. Die Änderung erfolgt über ein separates Visual Studio Projekt. In diesem wird die

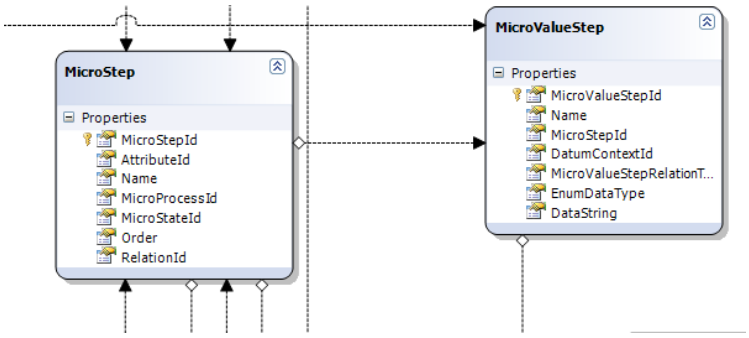


Abbildung 5.4: Gleichbenennung von Primär- und Fremdschlüssel

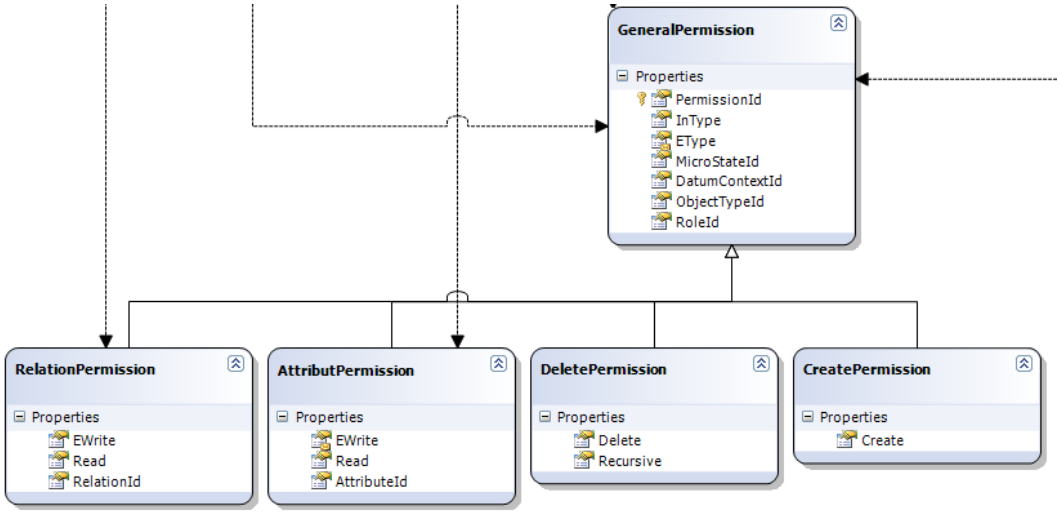


Abbildung 5.5: Vererbung von Rechten

aktuelle Datenbank mit dem neu generierten Modell verglichen und entsprechende Änderungsoperationen erstellt. Dabei ist zu beachten, dass das Ändern der Datenbank folgende Einschränkungen besitzt:

- Löschen einer Tabelle im Modell, löscht nicht die entsprechende Tabelle in der Datenbank .
- Löschen einer Tabelle im Modell, löscht nicht die Fremdschlüssel von und zu anderen Tabellen.
- Löschen einer Spalte ist nur möglich, falls keine Daten oder nur null-Werte in dieser enthalten sind.

- Ändern einer Spalte ist nur möglich, falls keine Daten oder nur null-Werte in dieser enthalten sind.
- Hinzufügen einer Spalte ist nur möglich, falls die Tabelle noch keine Werte enthält, andernfalls muss ein Default-Wert angegeben sein.
- Ändern der Primär- und Fremdschlüssel ist nur möglich, falls die Datenbank nach der Änderung konsistent ist.

Diese Einschränkungen dienen zur Sicherung der Konsistenz der Datenbank und zur Vermeidung von ungewolltem Datenverlust. Dies führt dazu, dass die Datenbank nicht aktualisiert wird oder dass das Speichern Fehler wirft. Ein klassisches Beispiel ist, dass nach dem Löschen einer Tabelle aus dem Modell diese noch in der Datenbank ist und Fremdschlüssel zu anderen Tabellen besitzt. Somit verhindert die Einschränkung, dass Zeilen in der Tabelle gelöscht werden können, da dadurch die Fremdschlüssel verletzt würden. In solchen Fällen muss die Datenbank manuell geändert werden.

Manuelle Erweiterungen der Klassen

Linq generiert die gesamte notwendige Logik, die im Prototypen gewünscht ist. Für das Manipulieren von Objekten ist der Entwickler selbst zuständig. Im Falle von PHILharmonicFlows ist dies eine komplexe Angelegenheit, da z.B. das Löschen unter Umständen auf große Teile des Modells Auswirkungen hat. Die Abb. 5.6 zeigt eine vereinfachte Darstellung der Datenbank und die Fremdschlüssel.

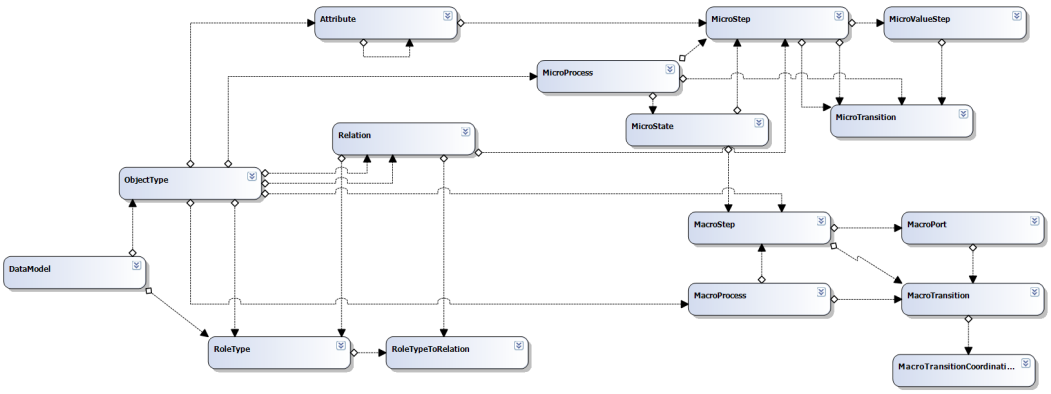


Abbildung 5.6: Vereinfachtes Darstellung der Datenbank

Die Abb. 5.7 zeigt die abstrakte Darstellung der Struktur der Modellierungsumgebung und deren Abhängigkeiten.

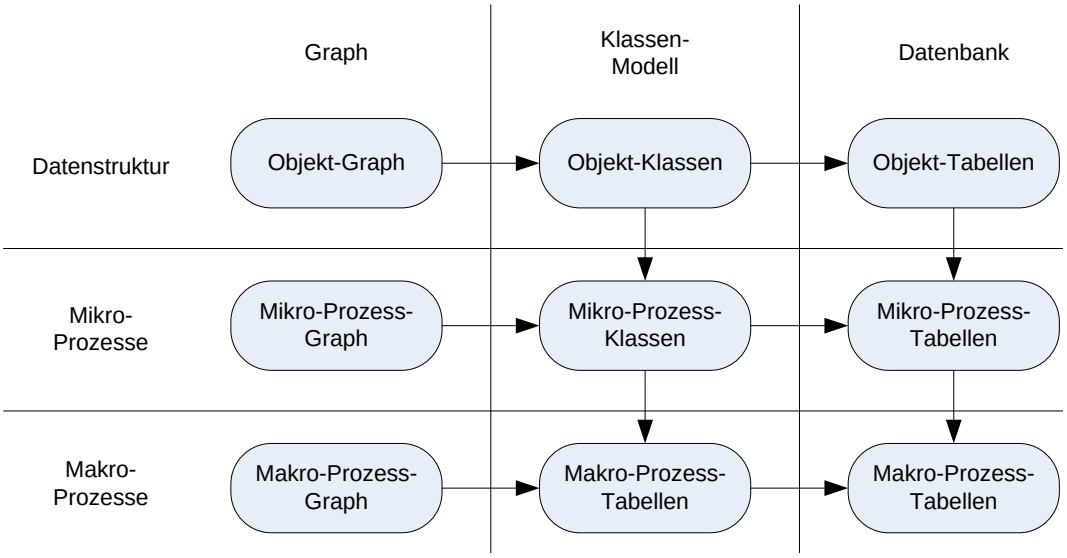


Abbildung 5.7: Schema der Abhänigkeiten

In der Vertikalen existieren die Ebenen Datenstruktur, Mikro-Prozesse und Makro-Prozesse. In der Horizontalen gibt es die Felder Graph, Klassen und Datenbank. Die Pfeile zeigen die Auswirkungen auf die anderen Elemente. Hier ist deutlich zu sehen, dass die höheren Ebenen Auswirkungen auf die darunterliegenden Ebenen haben. Je weiter oben eine Änderung stattfindet, desto mehr Elemente sind betroffen. Wird ein Objekt-Typ gelöscht, wird auch der entsprechende Mikro-Prozess gelöscht. Dies ist nicht weiter problematisch. Interessanter ist die Frage: Was passiert mit dem Makro-Prozess? Dieser besitzt Makro-Schritte, die auf gelöschte Mikro-Schritte referenzieren.
Grundsätzlich gilt, dass bei Veränderungen so wenig wie möglich an den anderen Elementen geändert wird und ggf. Null-Referenzen während der Modellierung erlaubt sind. Dies bedeutet, dass alle Algorithmen und Visualisierungen mit Null-Referenzen funktionieren. Im Folgenden werden die Auswirkungen von Veränderungen der Datenstruktur und der Mikro-Prozessstruktur besprochen. Dabei werden nur die Typen angesprochen, die Auswirkungen auf andere Ebenen haben oder umfangreiche Veränderungen bewirken. Die Veränderungen für die Makro-Prozesse werden im Kapitel 2.3.2 über die Makro-Prozesse separat behandelt.

Das Löschen eines Attributs betrifft einerseits den Objekt-Typ, andererseits auch Mikro-Schritte. Dabei ist wiederum zu beachten, dass das Attribut für die Mikro-Werte-Schritte notwendig ist. Um die Struktur zu gewährleisten, werden die Werte-Schritte und die Referenz zum Attribut im Mikro-Schritt gelöscht. Mikro-Transitionen der Werte-Schritte werden auf den Mikro-Schritt umgeleitet. Nur bei der Vergabe der Prioritäten kann eine ”Korrektheit per Konstruktion” nicht gewährleistet werden. Diese Prioritäten werden nicht geändert, sondern bei Validierung wird eine Fehlermeldung geworfen, da die Prioritäten nur vom Modellierer bestimmt werden können.

Das Ändern eines Attributs führt zu einem Problem mit den Mikro-Werte-Schritten, falls der Datentyp des Attributs geändert wird und die Werte in den Werte-Schritten dazu nicht kompatibel sind. In diesem Fall wird nichts geändert, sondern nur eine Fehlermeldung bei der Validierung geworfen.

Das Löschen eines Zustandes löscht nicht die Mikro-Schritte im Mikro-Prozess. Diese können einem neuen oder einem vorhanden Mikro-Zustand zugewiesen oder manuell gelöscht werden.

Der Zustandswechsel eines Mikro-Schrittes betrifft die Art der ein- und ausgehenden Mikro-Transitionen. Externe explizite Mikro-Transitionen werden zu internen impliziten Mikro-Transitionen und interne implizite Mikro-Transitionen werden zu externen impliziten Mikro-Transitionen umgewandelt. Falls durch die Verschiebung ein Attribut im Zustand doppelt verwendet wird, ändert sich nichts und bei der Überprüfung wird ein Fehler ausgegeben.

Die Änderung des Ziels oder der Quelle einer Mikro-Transition betrifft die Art der Mikro-Transition, gleich wie beim Zustandswechsel eines Mikro-Schrittes. Die Priorität der Mikro-Transition wird nicht geändert und liefert bei der Überprüfung ggf. eine Fehlermeldung, falls gleiche Prioritäten existieren.

Die Destroy-Methode existiert in jedem Typ. Genau wie in der Objektorientierung vorgesehen, räumt diese den Typ auf, d.h. sie löscht die Referenzen, die untergeordnete Elemente und sich selbst aus der Datenbank. Bei Änderungen an der Klassenstruktur müssen

diese Methoden manuell angepasst werden.
Im Regelfall wird das Entfernen der Objekte aus der Datenbank ohne Probleme von Linq umgesetzt, jedoch gab es beim Löschen von Mikro-Werte-Schritten das Problem, dass in der Datenbank erst die Referenzen des Werte-Schrittes auf Null gesetzt, und dann dieser gelöscht wurde. Diese Reihenfolge führt zu einem SQL-Fehler, da der Zwischenstand mit einem Null-Fremdschlüssel nicht erlaubt ist. Um diesen Fehler zu lösen, ist ein 'Deleted'-Bit in die Klasse eingefügt, das der Oberfläche und den Algorithmen signalisiert, dass dieses Objekt zum Löschen ansteht und nicht angezeigt bzw. ignoriert werden soll.

5.2 Anbindung der Laufzeitumgebung an die Datenbank

Die Anbindung der Laufzeitumgebung stellt andere Anforderungen als die Modellierungsumgebung. Während im Modellierer eine Interaktion zwischen Datenbank, Klassenmodell und Graphen gefordert ist, benötigt die Laufzeitumgebung nur einen lesenden Zugriff.
Die Werte aus der Datenbank werden mittels SQL-Strings und Arrays abgefragt und in einfache Klassen übertragen. Diese werden dann für die Anzeige und die semantische Operation genutzt. Aus diesem Grund sind diese Klassen in der Laufzeitumgebung vernetzt. Eine Ersetzung der Klassen durch die Linq-Klassen ist deswegen zeitaufwändig und komplex, und würde für die Laufzeitumgebung keine signifikante Verbesserung bieten. Eine Ersetzung würde zwar ein Klassenmodell weniger bieten, jedoch ist nach aktuellem Stand der Dinge die Mikro-Prozessausführung abgeschlossen und deswegen keine Veränderung des Modelles zu erwarten.
Aus diesem Grund beschränkt sich die Integration auf die Ersetzung der SQL-Anfragen durch die Linq-Technologie.

5.2.1 Ersetzen der SQL-Anfragen durch Linq

In dem `TypeDBManager` der Laufzeitumgebung werden die SQL-Abfragen nach dem Muster aus Listing 5.2 getätigt, indem SQL-Strings aufgebaut werden und die Antworten zeilenweise gelesen werden. Jede Änderung an der Datenbank muss in diesen Methoden von Hand gemacht werden. Der Entwickler kann Fehler oder vergessene Änderungen in den Abfragen erst zur Laufzeit feststellen.

```
1  const string query = "SELECT TOP 1 AttributeId FROM Attribute
       WHERE Name = @name AND ObjectTypeId = @oid";
2
3  var con = new SqlConnection(Connectionstring);
4  var cmd = new SqlCommand(query, con);
5  cmd.Parameters.AddWithValue("@name", name);
6  cmd.Parameters.AddWithValue("@oid", oid);
7  con.Open();
8
9  var id = ConvInt(cmd.ExecuteScalar());
10 con.Close();
11 return id;
```

Listing 5.2: SQL-Abfrage der AttributId

Mit Linq wird nur auf die Linq-Klassen zugegriffen und deren Werte in die einfachen Klassen übertragen. Das Listing 5.3 macht dasselbe wie das Listing 5.2, nur mit Linq-Technologie und nicht über SQL-Abfragen.

```
1  return _context.Attributes.Where(v => v.Name == name && v.
       ObjectTypeId == oid).First().AttributeId;
```

Listing 5.3: Linq-Abfrage für AttributeId

5.3 Zusammenfassung

Das Ergebnis der Integration ist die Verbindung von Modellierungsumgebung und Laufzeitumgebung. Anders als anfangs geplant, ist dies über eine Neugestaltung der Datenbank geschehen. Dafür ist die Gestaltung der Datenbank und des Klassenmodells nun eine modell-getriebene Modellierung. Dank der Möglichkeiten der Linq-Technologie ist die Umgestaltung und Erweiterung der Datenbank und des Klassenmodells flexibler und objektorientierter. Gleichzeitig sind auch Grenzen der Linq-Technologie aufgezeigt worden. Die Lücke zwischen Linq und yFiles-Graphen musste von Hand geschlossen und Lösungswege dafür

gefunden werden.

Die Integration ist nicht abgeschlossen, da keine Rechteverwaltung in der Laufzeitumgebung existiert. Dies ist nicht anders lösbar, da die Rechtegestaltung (Klassen, Methoden und Logik) zu diesem Zeitpunkt nicht geklärt wurde.

6 Implementation der Makro-Prozesse im Modellierer

Nachdem die Datenbankverbindung zum Prototyp hergestellt ist, kann im nächsten Schritt die Implementierung von neuen Komponenten beginnen. Im diesem Kapitel wird die Implementierung der Makro-Prozesse behandelt.
Zunächst werden nur die grafischen Erweiterungen und Veränderungen in der Modellierungsumgebung besprochen, danach folgen die technischen Details. Die Abb. 6.1 zeigt eine kurze Übersicht des Kapitels.
Die Makro-Prozesse dienen zur Steuerung und Synchronisation der Mikro-Prozesse. Dies

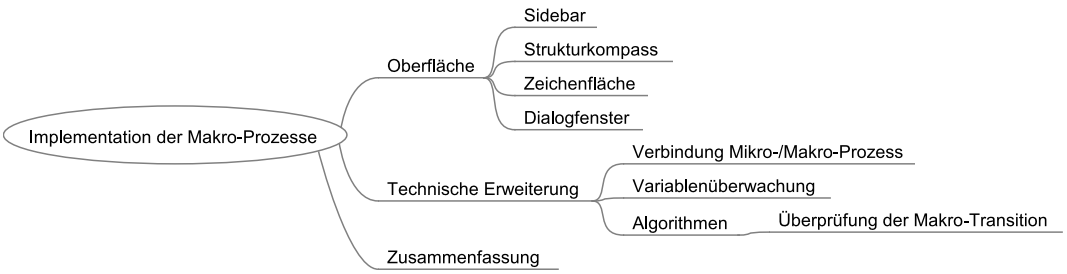


Abbildung 6.1: Übersicht über das Kapitel

hat zur Folge, dass in der Oberfläche sowohl der Makro-Prozess, als auch der Mikro-Prozess dargestellt werden müssen. Das Usability-Konzept von Nicole Wagner [27] sieht eine integrierte Sicht aus Mikro-Prozessen und Makro-Prozess vor. Diese Sicht basiert auf einem älteren Konzeptstand und wurde im Zuge einer fachlichen Anpassung durch eine einfachere Sicht ersetzt. Die Anpassung basiert auf den grafischen Darstellungen der Makro-Prozessen in der Modellierungsgrundlage von Vera Künzle [11].

6.1 Oberfläche

Die Abb. 6.2 zeigt die Aufteilung der Oberfläche in die drei wichtigsten Komponenten: *Sidebar*, *Strukturkompass* und *Zeichenfläche*. Der *Strukturkompass* und die *Zeichenfläche* werden zusammen als *Arbeitsfläche* bezeichnet. Im Folgenden werden die drei Komponenten besprochen.

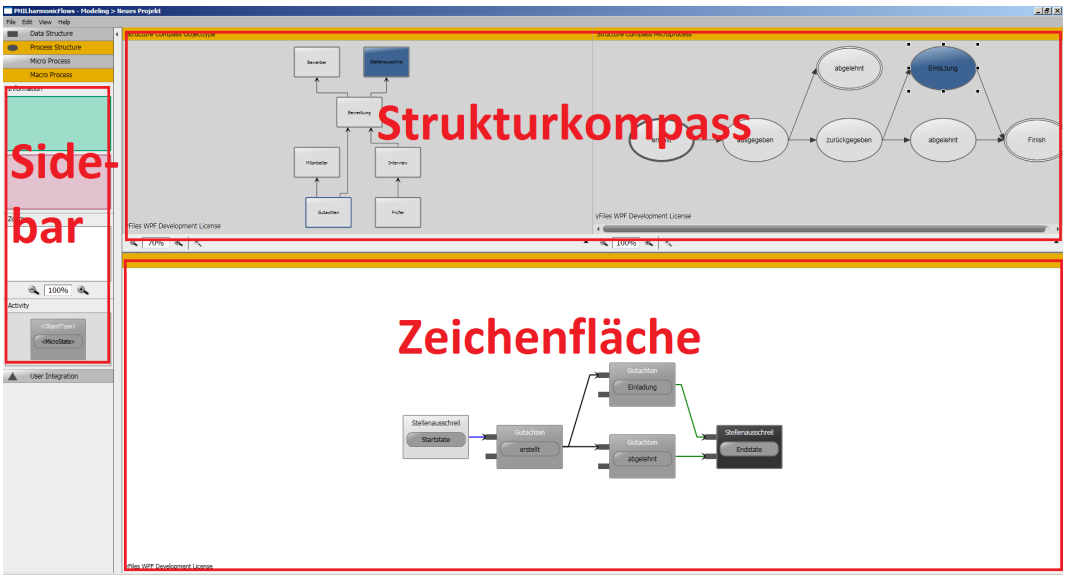
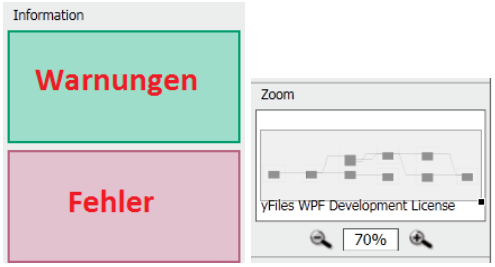


Abbildung 6.2: Aufteilung der Oberfläche

6.1.1 Sidebar

Die *Sidebar* ermöglicht dem Benutzer, die Makro-Prozess-Elemente in die *Zeichenfläche* zu ziehen. Dies ist dasselbe System, wie es in der Mikro-Prozess *Arbeitsfläche* eingesetzt wird. Aktuell werden nur Makro-Prozess-Schritte angeboten (vgl. Abb. 6.3). Daneben exis-

Abbildung 6.3: Aktivitäten



(a) Informations-Box (b) Miniaturansicht

Abbildung 6.4: Elemente der Sidebar

tiert noch eine Informations-Box für die Anzeige von Fehlern und Warnungen (siehe Abb. 6.4(a)). Für eine bessere Übersicht über den Makro-Prozess gibt es eine Miniaturansicht (siehe Abb. 6.4(b)).

6.1.2 Strukturkompass

Der *Strukturkompass* dient der Navigation zwischen den verschiedenen Mikro- und Makro-Prozessen. Anders als bei der Mikro-Prozess-Arbeitsfläche existieren in dieser *Arbeits-*

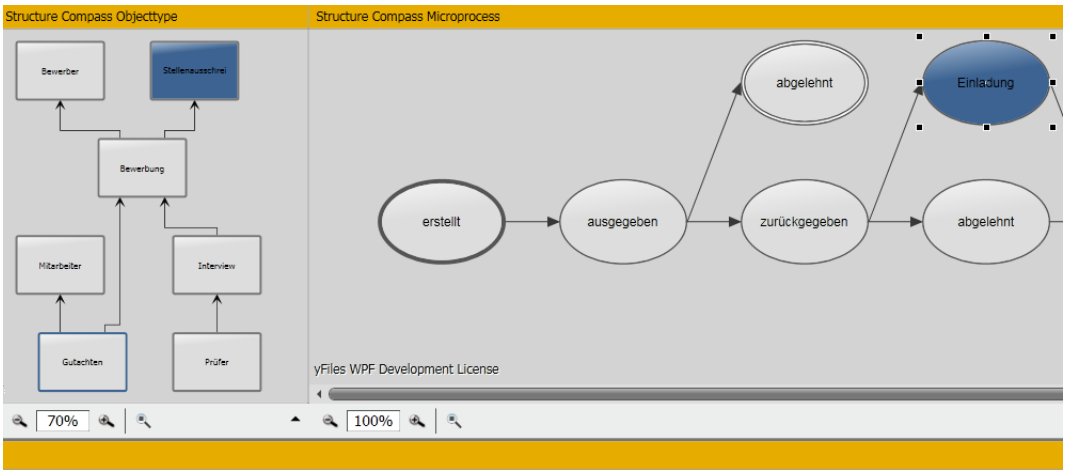


Abbildung 6.5: links: Objekt-Typen, rechts: Mikro-Prozess

fläche zwei *Strukturkompasse*. Im linken Strukturkompass, dem sogenannten `Structur Compass Datamodel` (Strukturkompass des Datenmodells), sind die Objekt-Typen dargestellt, während im rechten, dem `Structur Compass Microprocess` (*Strukturkompass*

des Mikro-Prozesses), der Mikro-Prozess des ausgewählten Objekt-Typs zu sehen ist (vgl. Abb. 6.5).

Strukturkompass des Datenmodells ermöglicht das Auswählen des Makro-Prozesses, der in der *Zeichenfläche* dargestellt wird, sowie des Mikro-Prozesses, der im rechten Strukturkompass angezeigt wird. Der Makro-Prozess wird durch Doppelklick des entsprechenden Objekt-Typs ausgewählt. Die Auswahl wird durch die blaue Füllung des Objekt-Typs visuell angezeigt. Die *Zeichenfläche* erstellt den entsprechenden Graphen zum Makro-Prozess. Der Mikro-Prozess wird durch Klick auf den Objekt-Typ ausgewählt und durch die blaue Umrandung des Objekt-Typs visuell dargestellt. Der rechte *Strukturkompass* erstellt den entsprechenden Graphen des Mikro-Prozesses.

Die Klasse des *Strukturkompasses* wurde zunächst kopiert, um das Original nicht zu verändern. Dies war notwendig, da die Veränderungen sehr umfangreich sind und auf diese Weise keine Fehler in die bestehende Mikro-Prozess-*Arbeitsfläche* gelangen. Nach dem Kopieren wurde der neue *Strukturkompass* so modifiziert, dass bei einem Klick der Objekt-Typ blau umrandet wird, während bei einem doppelten Klick der Objekt-Typ blau eingefärbt wird. Zusätzlich besitzt der *Strukturkompass* zwei neue Schnittstellen, die diese Klick-Events für andere Komponenten anbieten (vgl. Listing 6.1).

```
1 public delegate void ObjectTypeClickedHandler(ObjectType o);
2 public event ObjectTypeClickedHandler ObjectTypeClicked;
3
4 public delegate void ObjectTypeDoubleClickedHandler(ObjectType o);
5 public event ObjectTypeDoubleClickedHandler
    ObjectTypeDoubleClicked;
```

Listing 6.1: Zwei Schnittstellen

Der Style für die Objekt-Typen musste um die zwei Darstellungsmöglichkeiten erweitert werden. Dabei trat der Fehler auf, dass die Darstellung im *Strukturkompass* nicht immer aktualisiert wurde. Dies wurde behoben, indem nach jeder Veränderung der *Strukturkompass* gezwungen wurde, den Graphen neu darzustellen.

Strukturkompass des Mikro-Prozesses zeigt den ausgewählten Mikro-Prozess. Die Darstellung ist aus der Zustandsübersicht der Mikro-Prozess-*Zeichenfläche* übernommen. Die Zustände kann der Benutzer in die *Zeichenfläche* ziehen und dort einen Makro-Schritt daraus erstellen.

Dieser *Strukturkompass* ist eine Kopie des ursprünglichen *Strukturkompasses*. Der Grund ist, dass die Veränderungen der Darstellung von Objekt-Typen zu Mikro-Prozessen zu komplex ist, um sie sinnvoll in einer Komponente zu vereinen.
Ein schwieriges Problem war das Ziehen (DragAndDrop) vom *Strukturkompass* zur *Zeichenfläche*, da yFiles dies nicht unterstützt. Dies wurde gelöst, in dem das Verschieben von Knoten genutzt wurde. Falls der Benutzer einen Knoten verschieben möchte, wird dieser unbeweglich und ein Drag-Event wird erstellt. Das Listing 6.2 zeigt das Erstellen des Drag-Events. Dies nutzt den DragDrop-Handler von WPF. Der Handler erhält nicht den Linq-Klasse, sondern den yFiles-Knoten. Die Darstellungskomponente `graphControl` zeigt deswegen während des Ziehens einen Knoten an. Dadurch erhält der Benutzer ein besseres visuelles Feedback als mit der Linq-Klasse.

```
1 public void InitializeDrag(IInputModeContext inputModeContext){
2         // Drag-Container erstellen
3         DataObject dao = new DataObject();
4         // Konten in den Container setzen
5         dao.SetData(typeof(INode), _node);
6         //Drag&Drop-Event starten
7         DragDrop.DoDragDrop(_graphControl, dao, DragDropEffects.
              Link | DragDropEffects.Copy | DragDropEffects.Move);
8 }
```

Listing 6.2: Erstellen des Drag-Event

Die Unbeweglichkeit verhindert, dass sich der Sichtbereich nicht in Richtung Mauszeiger verschiebt, wenn dieser in die Nähe des Randes kommt. Durch diesen Trick kann der Benutzer nun den Knoten aus dem *Strukturkompass* in die *Zeichenfläche* ziehen, ohne die Darstellung des Strukturkompasses zu verändern.
Beim Loslassen in der Arbeitsfläche wird ein Drop-Event ausgelöst und yFiles erstellt einen Knoten an der entsprechenden Position.

6.1.3 Zeichenfläche

Die *Zeichenfläche* ist der Kern der Arbeitsfläche. Sie dient zur Darstellung und Manipulation des Makro-Prozesses. Wie bei der Mikro-Prozess-Zeichenfläche werden Makro-Schritte per 'Drag&Drop' in der Zeichenfläche erstellt. Wenn der Knoten vom *Strukturkompass* gezogen ist, so wird der Makro-Schritt automatisch mit den notwendigen Werten (Objekt-Typ und Zustand) erstellt. Andernfalls bekommt dieser Null-Werte (vgl. Abb. 6.6). Die Werte können jederzeit über eine *Combobox* geändert werden. Diese zeigt nur die gültigen Objekt-Typen bzw. Zustände an, die in dem vorliegenden Graphen gültig sind.
Anders als bei den Mikro-Prozessen kann hier der Modellierer die Start- und End-Makro-Schritte nicht selber Modellieren, sondern werden vom System verwaltet (siehe Kapitel 6.2.1). Die Makro-Schritte werden mit Kanten verbunden. Dabei überprüft das System, ob

Abbildung 6.6: links vom Strukturkompass; rechts von der Sidebar

eine Verbindung gültig ist (siehe Kapitel 6.2.3). Ist sie ungültig, bietet das System keine Andockpunkte (vgl. Abb. 6.7). Nachdem eine Verbindung erstellt ist, prüft das System, welche

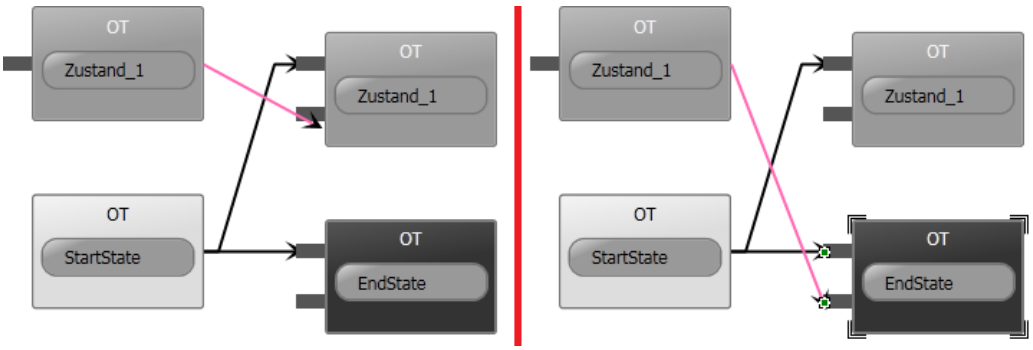


Abbildung 6.7: links ohne und rechts mit Andockpunkten

Art von Transition vorliegt (Top-Down, etc.) und färbt diese entsprechend der Art ein (vgl. Tabelle 6.1). Durch einen Doppelklick auf die Kante wird ein Dialogfenster geöffnet, um die Synchronisationsbedingung zu bearbeiten.

Typ	Farbe
Top-Down	Blau
Bottom-Up	Grün
Transverse	Rot
Self	Schwarz
Invalid	Pink

Tabelle 6.2: Farben der Transition-Arten

6.1.4 Dialogfenster

Das aufwändigste Dialogfenster für die Synchronisationsbedingungen ist das der Aggregation. Hier hat der Benutzer die Aufgabe, ein komplexes Prädikat in dem Textfeld anzugeben (vgl. Abb. 6.8). Um ihn dabei zu unterstützen, sind in der oberen Hälfte des Fenster einige Knöpfe angebracht, die an der Curser-Position im Textfeld entsprechende Zeichen einfügen. Sobald der Nutzer die Knöpfe `Check` oder `Ok` benutzt, prüft das Dialogfenster, ob das

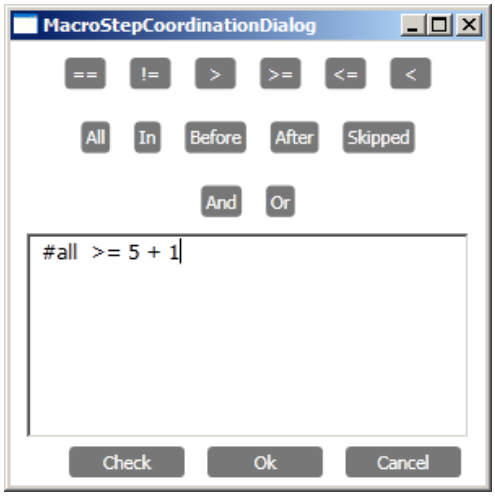


Abbildung 6.8: Dialogfenster für die Aggregation

eingegebene Prädikat gültig ist. Dafür wird ein Expression-Parser eingesetzt, der aus dem String einen Expression-Baum generiert, welcher zur Laufzeit ausgewertet werden kann. Dieser überprüft nur die syntaktische Korrektheit, nicht die semantische des Prädikats.

6.2 Technische Erweiterungen

In diesem Abschnitt werden die technischen Erweiterungen der Software vorgestellt, die für den Benutzer nicht sichtbar sind. Dabei werden nur diejenigen betrachtet, die aufwändig oder komplex sind. Einfachere Änderungen (z.B. das Zerstören von Objekten oder das Konvertieren von Werten) werden nicht behandelt, um den Rahmen der Arbeit nicht zu sprengen.

6.2.1 Verbindung Mikro-/Makro-Prozess

Wie in Kapitel 2.3.2 beschrieben, sind die Makro-Prozesse eines Objekt-Typs mit den Mikro-Prozessen verknüpft. Der Makro-Prozess beginnt mit einem Start-Makro-Schritt, der einen Start-Zustand des Mikro-Prozesses referenziert, und endet mit einem End-Makro-Schritt, der einen End-Zustand des Mikro-Prozesses referenziert. Um den Benutzer zu entlasten, werden diese Start- und End-Makro-Schritte im Makro-Prozess automatisch erstellt und angepasst, wenn die Start- und End-Zustände des Mikro-Prozesses geändert werden. Dabei gibt es folgende Fälle:

- Start-Zustand wird erstellt
- Start-Zustand wird gelöscht
- End-Zustand wird erstellt
- End-Zustand wird gelöscht

Erstellen und Löschen bedeutet jedoch nicht das Erstellen oder Löschen von Zuständen, sondern das Setzen von Flags für Start- oder End-Zustände. Bei jedem Fall ist die Anzahl der Zustände zu beachten: bei keinem, einem oder mehreren Zuständen verhält sich die Anpassung jeweils anders. Die Anpassungen können ungültige Makro-Prozesse hervorrufen. Diese werden in den Informations-Box angezeigt und müssen vom Benutzer aufgelöst werden.
Um die Anpassungen zu verdeutlichen, wird dies an einem minimalen Prozess vorgeführt. Als Ausgangssituation dient Abb. 6.9. Jede Abbildung eines Falles ist das Ergebnis der Änderung des Mikro-Prozesses der Ausgangssituation.

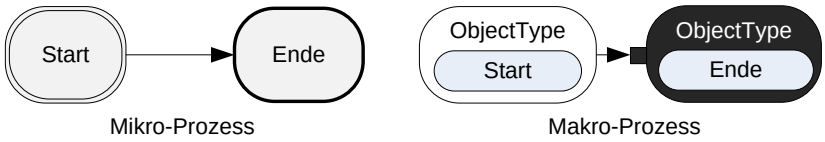


Abbildung 6.9: Ausgangssituation

Fall: Start-Zustand wird erstellt

Falls kein Start-Zustand existiert, beginnt der Makro-Prozess mit einem Start-Schritt, der keinen Zustand referenziert. Sobald der Benutzer einen Start-Zustand erstellt, verweist der Start-Makro-Schritt auf diesen Zustand.
Die Modellierungsumgebung erlaubt das Modellieren von mehreren Start-Zuständen, auch wenn dies semantisch inkorrekt ist. Falls nun der Benutzer weitere Start-Zustände erstellt, ändert sich am Makro-Prozess nichts. Der Start-Makro-Schritt referenziert weiterhin den ersten Start-Zustand.

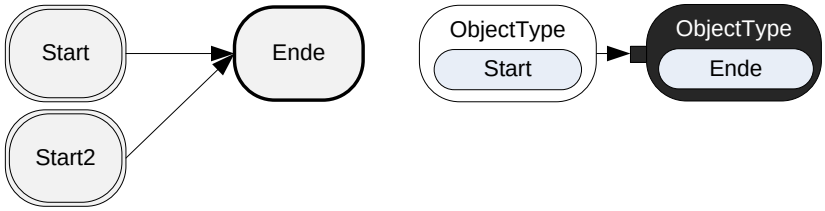


Abbildung 6.10: Zwei Start-Zustände

Fall: Start-Zustand wird gelöscht

Existiert nur ein Start-Zustand und dieser wird gelöscht, so wird die Referenz des Start-Makro-Schritts auf Null gesetzt.
Falls mehrere Start-Zustände existieren, und der referenzierte Start-Zustand gelöscht wird, sucht das System einen anderen Start-Zustand und zeigt auf diesen im Start-Makro-Schritt.
Wenn ein nicht referenzierter Start-Zustand gelöscht wird, passiert im Makro-Prozess nichts.

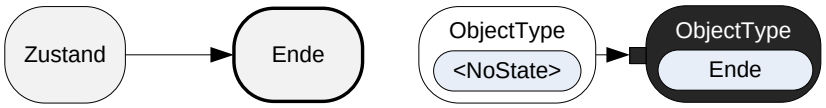


Abbildung 6.11: Ohne Start-Zustand

Fall: End-Zustand wird erstellt

Sobald der Benutzer einen End-Zustand erstellt, wird geprüft, ob ein entsprechender Makro-Schritt im Makro-Prozess existiert. Wenn ja, wird dieser in einen End-Makro-Schritt konvertiert. Falls nicht, wird ein neuer End-Makro-Schritt erstellt und dieser mit dem Start-Makro-Schritt verbunden.

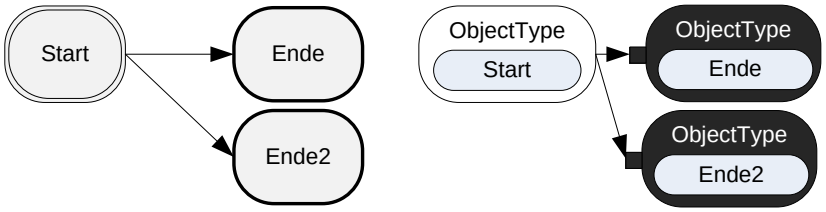


Abbildung 6.12: Zwei End-Zustände

Fall: End-Zustand wird gelöscht

Wird der End-Zustand gelöscht, wird sein entsprechender End-Makro-Schritt im Makro-Prozess in einen normalen Makro-Schritt konvertiert.

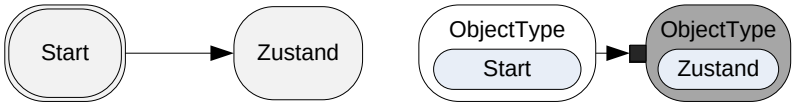


Abbildung 6.13: Ohne End-Zustand

6.2.2 Variablenüberwachung

Linq bietet für jede Variable eine `On<VariableName>Changing` Methode an (vgl. Listing 6.3). Diese Methode kann mittels 'override' überschrieben und so auf eine Änderung der

Daten reagiert werden. Die Methode wird nur aufgerufen, wenn der neue Wert vom aktuelle Wert abweicht. Der Aufruf erfolgt bevor der neue Wert zugewiesen wird. Dies kann wichtig sein, um den alten Wert an anderer Stelle zu löschen (z.B. aus der Datenbank zu entfernen).

```
1 partial void OnETypeChanging(int value){
2         var oldmtc = this.MacroTransitionCoordination;
3         switch ((MacroTransitionType)value){
4                 //newmtc = neuer MacroTransitionCoordination
5         }
6         //delete old and set new
7         if(oldmtc != null) oldmtc.Destroy();
8         this.MacroTransitionCoordination = newmtc;
9 }
```

Listing 6.3: Ändern der Transitionsbedingung, wenn der Transitionstyp geändert wird

Eine Alternative sind die von Linq implementierten `INotifyPropertyChanging` und `INotifyPropertyChanged` Interfaces. Diese übergeben einen String mit dem Namen der Variable, die geändert wird bzw. geändert werden soll. Die Interfaces können auch außerhalb der Linq-Klassen genutzt werden (z.B. von GUI-Elementen).

Eingesetzt wird diese die Überwachung für die Art der Makro-Transition. Ändert sich dieser Wert, löscht das System die dazugehörigen `MacroTransitionCoordination`-Klassen der Makro-Transition und erstellt eine neue Klasse entsprechend der neuen Art der Makro-Transition.

6.2.3 Algorithmen und Konsistenz

Die Algorithmen sind für die Überprüfung der Konsistenz des Makro-Prozesses wichtig. Diese werden in sogenannten `<Type>ConsistencyManagers` zusammengefasst. Es existieren Manager für *Objekt-Typ*, *Makro-Prozess* und *Mikro-Prozess*. Der hierfür wichtige Manager ist der `MacroProcessConsistencyManager`(vgl. Abb. 6.14). Die Methoden lassen sich in folgende Gruppen aufteilen:

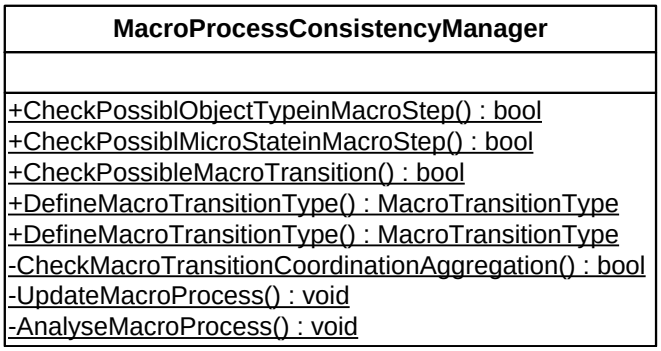


Abbildung 6.14: Klassendiagramm des MakroProcessConsistenceManager

- Define: Bestimmt den Wert einer Struktur (z.B. Den Typ einer Transition) und gibt diesen zurück.
- Check: Bestimmt ob eine Struktur gültig wäre, wenn die Struktur geändert würde (z.B. neue Makro-Transition in einem Makro-Prozess). Dabei wird die Struktur nicht geändert.
- Analyse: Überprüft ein Element (z.B. den Makro-Prozess) auf inkonsistente Strukturen (z.B. fehlende Makro-Transitionen). Die Ergebnisse werden in einer Liste gespeichert, dass Fehlertyp und beteiligte Elemente beinhaltet.
- Update: Überprüft ein Element (z.B. den Makro-Prozess) und löst Inkonsistenzen selbstständig auf, falls dafür Lösungen implementiert sind (z.B. Verbindung Mikro-/Makro-Prozess).

Alle Methoden sind so gestaltet, dass sie nur die Eingabeparameter benötigen, d.h. sie können in der Laufzeitumgebung und in der Modellierungsumgebung genutzt werden. Des Weiteren sind die Methoden auf Null-Referenzen ausgelegt, die während der Modellierung auftreten können (z.B. fehlender Zustand oder Null-Werte im Makro-Schritt).
Im Folgenden werden die häufig genutzten Algorithmen vorgestellt.

Überprüfung der Makro-Transition

Die Methode `CheckPossibleMacroTransition` prüft ob eine Makro-Transition zwischen zwei Makro-Schritten erstellt werden darf. Diese Methode kommt sowohl bei der Modellierung der Kanten des Makro-Prozesses, als auch bei der allgemeinen Überprüfung vor. Da

sie im ersten Fall auf eine Mausbewegung reagiert, muss gewährleistet werden, dass sie eine schnelle Antwortzeit von unter einer Sekunde hat, da sonst der Benutzer Verzögerungen während der Benutzung bemerken würde.
Die Überprüfung besteht aus folgenden Schritten:

- Bestimmung des Transitionstyps
- Überprüfung der Pfade

Zur Bestimmung des Transitionstyps wird die Methode `DefineMacroTransitionType (sourceOt, sinkOt)` der `ObjectTypeConsistencyManager`-Klassen genutzt.

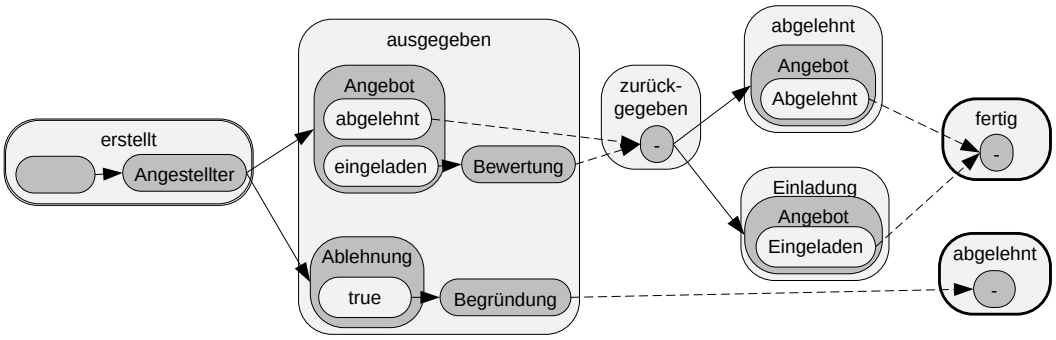


Abbildung 6.15: Beispiel eines Mikro-Prozess

Als Pfad wird ein Weg von einem Knoten zu einem anderen in einem Graphen definiert. In Abb. 6.15 gibt es einen Pfad von `erstellt` zu `Einladung` über `ausgegeben` und `zurückgegeben`. Jedoch von `fertig` existiert kein Pfad zu anderen Knoten. Die Pfad-Regel im Makro-Prozess besagt, dass wenn ein Pfad zwischen zwei Makro-Schritten desselben Objekt-Typs existiert, auch ein Pfad zwischen deren Zuständen existieren muss. In Abb. 6.16 besteht zwischen `Gutachten-erstellt` und `Gutachten-abgelehnt` im Makro-Prozess ein Pfad. Im Mikro-Prozess besteht zwischen diesen Zuständen ebenfalls ein Pfad (vgl. Abb. 6.15).

Zusammenfassend bedeutet dies: Falls eine Transition zwischen zwei Makro-Schritten erstellt wird, muss erstens die Pfadregel für diese zwei Makro-Schritte geprüft werden, und zweitens die Vorgänger des ersten Makro-Schrittes mit den Nachfolgern des zweiten Makro-Schrittes geprüft werden.

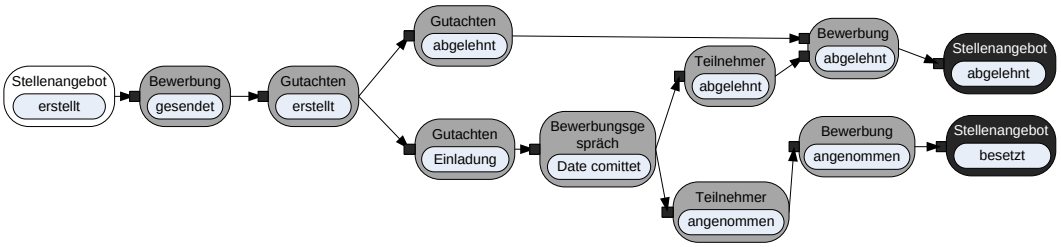


Abbildung 6.16: Beispiel eines Makro-Prozess-Typs

Die Pfadüberprüfung in den Mikro-Prozessen erledigt die Methode `CheckPath(ancestor, successor)` der `MicroProcessConsistencyManager`-Klasse. Sie speichert nicht die Struktur des Mikro-Prozesses und baut die Struktur bei jedem Aufruf neu auf. Eine Optimierung dieser Methode würde die Dauer des Algorithmus verbessern. Bei Tests mit mehr als 20 Schritten traten keine merkbaren Verzögerungen auf.

6.3 Zusammenfassung

Das Ergebnis der Implementation ist eine Modellierungsmöglichkeit für die Makro-Prozesse und verschiedene Algorithmen für die Überprüfung der Konsistenz der Prozess-Modells. In der Oberfläche sind noch einige Workarounds (z.B. DragAndDrop beim Strukturkompass) enthalten, die mit mehr Zeit und Aufwand besser gemacht werden können, aber für einen Prototyp durchaus vertretbar sind.

7 Implementation der Benutzerintergration in der Modellierungsumgebung

Die Benutzerintegration ist eine wichtige Komponente für die Prozessausführung, welche die Benutzerrechte in einem Prozess steuert.
Zunächst werden nur die grafischen Erweiterungen und Veränderungen in der Modellierungsumgebung besprochen, danach die technischen Details. Die Abb. 7.1 zeigt eine kurze Übersicht des Kapitels.

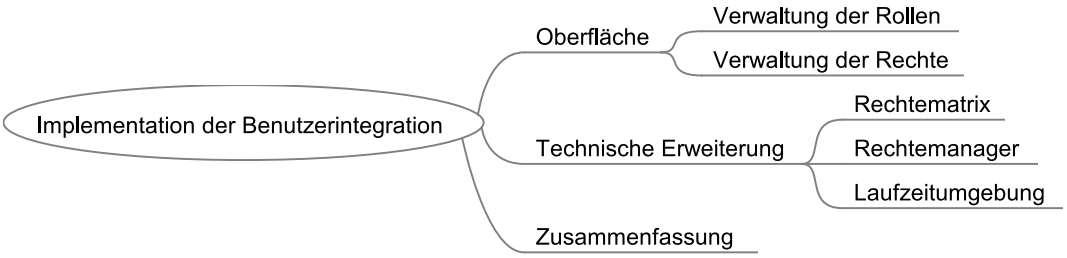


Abbildung 7.1: Übersicht über das Kapitel

7.1 Oberfläche

Die Oberflächengestaltung basiert auf dem Design-Entwurf von Nicole Wagner [27]. Die Benutzerintegration ist in die Verwaltung der Rollen und der Rechte für die Rollen eingeteilt. Daneben wurde die Verwaltung der Verantwortlichkeiten in die Modellierung der Mikro-Prozesse integriert.

7.1.1 Verwaltung der Rollen

Die Verwaltung der Rollen ist ähnlich der Werte-Typen aufgebaut (Abb. 7.2). Jeder Benutzer-Typ besitzt eine eigene Zeile, die aufgeklappt werden kann. Darin befindet sich eine Tabelle mit den zugewiesenen Rollen. Die Rollen können mit den entsprechenden Icons gelöscht

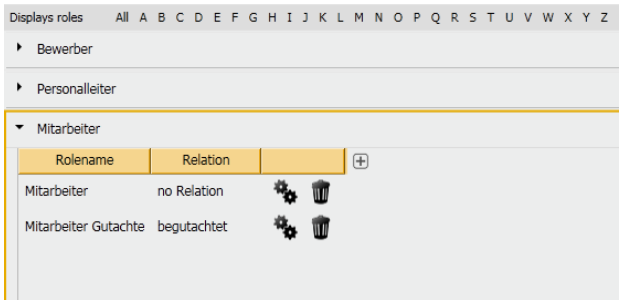


Abbildung 7.2: Verwaltung der Rollen

und editiert werden. Wie bei den Wertetypen können neue Rollen über das Plus-Symbol erstellt werden. Dazu öffnet sich ein kompaktes Dialogfenster mit den benötigen Feldern.

7.1.2 Verwaltung der Rechte

Die Verwaltung der Rechte geschieht in einer sogenannten Rechte-Matrix (Abb. 7.3).

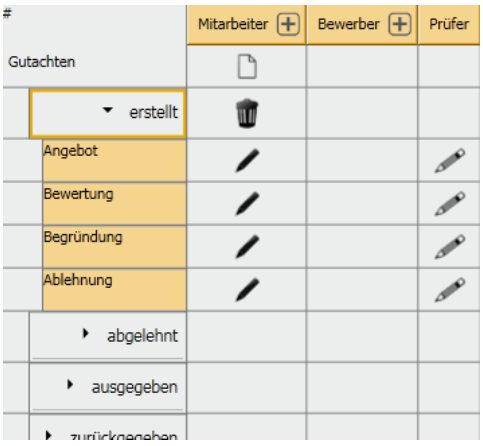


Abbildung 7.3: Verwaltung der Rechte

Name	Icon
Erstellen	
Löschen	
Lesen	
Optionales Schreiben	
Gezwungenes Schreiben	

Tabelle 7.2: Legende

Nach Austesten verschiedener Darstellungen fiel die Wahl auf das ursprüngliche Design, bei dem die Spalten den Rollen entsprechen, und die Zeilen den Objekt-Typen. Die Rechte werden wie folgt angezeigt (vgl. Abb. 7.4):
Die Zellen zeigen das Erstell-Recht an. Die Objekt-Typen lassen sich um deren Zustände

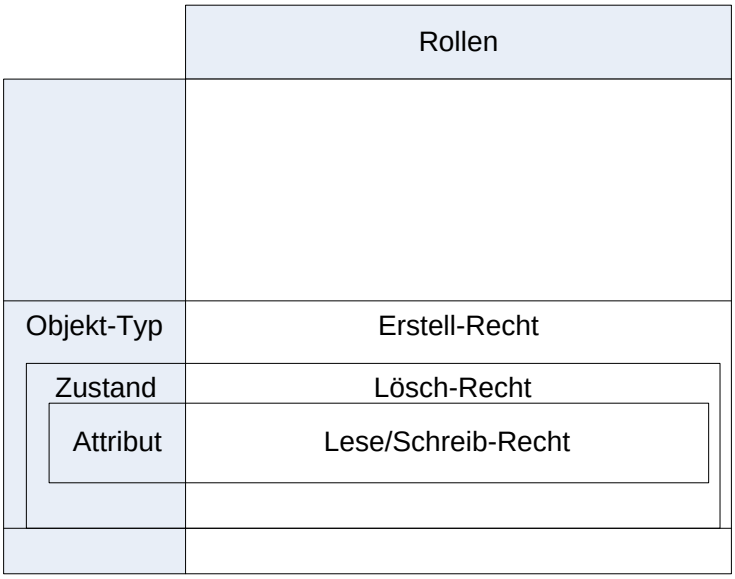


Abbildung 7.4: Schema der Rechte-Matrix

erweitern, diese Zellen zeigen dann die Löschen-Rechte an. Die Zustände können wiederum um die Attribute erweitert werden. Deren Zellen beinhalten die Lese/Schreib-Rechte. Statt jeweils ein Symbol für Lese- und Schreib-Recht anzuzeigen, werden diese kombiniert. Das Schreib-Recht beinhaltet nun das Lese-Recht, deswegen wird nur zwischen den verschiedenen Rechten durchgeschaltet (kein Recht -> Lese-Recht -> optionales Schreib-Recht -> gezwungenes Schreib-Recht -> kein Recht -> ...).
Die in dem Zustand genutzten Attribute werden mit einer fetten Schrift versehen, damit der Modellierer weiss, welche Rechte wichtig sind.

Die Rechte-Matrix zeigt zusätzlich die Verantwortlichkeiten an (z.B. das Zustands-Privileg gibt einer Rolle alle Rechte für einen Zustand). Diese von den Verantwortlichkeiten kommenden Rechte können vom Modellierer nicht weggenommen werden. Beim Erstellen der Matrix sind immer zwei Überprüfungen pro Zelle notwendig, für das Recht selbst und für die Verantwortlichkeiten, die dieses Recht geben.

7.1.3 Dialogfenster für instanzspezifische Rechte

Ein instanzspezifische Recht definiert Rechte, die eine Rolle für ein bestimmten Objekt-Typ und dessen Attribute besitzt. Bestimmt wird dieser Objekt-Typ über einen Pfad. Kernelement dieses Dialogfensters ist es, diesen Pfad zu bestimmen. Die Abb. 7.5 zeigt ein solches Beispiel.

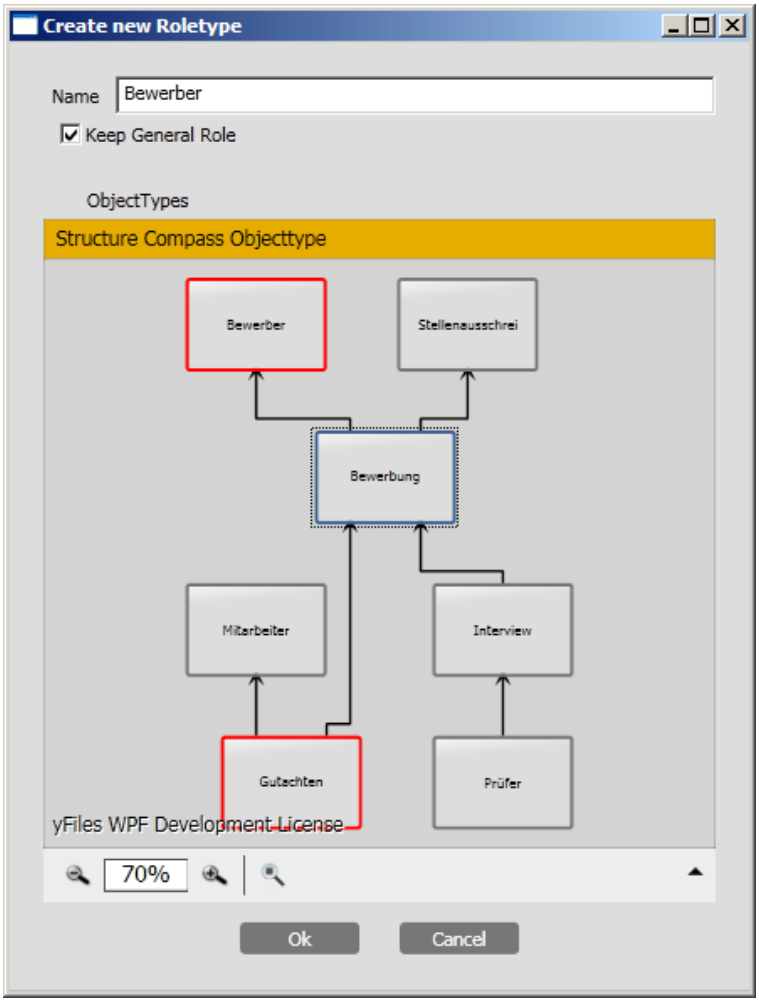


Abbildung 7.5: Dialogfenster für spezifische Rollen

Ausgehend von `Mitarbeiter` wurde ein Pfad über `Bewerbung` zu `Personlalleiter` gewählt. Wird auf eine Relation oder ein Objekt-Typ geklickt, dass am Ende des Pfades ist, so wird der Pfad damit erweitert. Ist die Relation oder der Objekt-Typ bereits Element des

Pfades, so wird der Pfad gekürzt, bis das Element nicht mehr im Pfad ist. Würde auf den Objekt-Typ `Bewerbung` geklickt, so würden `Bewerbung` und `Personalleiter` aus dem Pfad entfernt.
Start- und End-Punkt sind Rot umrandet, damit diese deutlich erkennbar sind. Ist die Rolle eine Relations-Rollen-Typ, so ist die Relation auch rot eingefärbt. Die Elemente des Pfades zwischen Start und End-Punkt werden blau eingefärbt.

Die Anzeige erfolgt über die Variante des Strukturkompasses vom Makro-Prozess. Dieser enthält die nötigen Schnittstellen, um Knoten und Kanten rot bzw. blau zu farben, und um auf die Klick-Events zu reagieren. Die Steuerung und Verarbeitung der Events erfolgt im Dialogfenster.

7.2 Technische Erweiterung

In diesem Kapitel werden die notwendigen technischen Erweiterungen besprochen. Dies umfasst die Modifikation von Oberflächenelementen und die Modellierung der Klassenstruktur. Daneben müssen die wichtigen `Destroy`-Methoden und eine Rechte-Verwaltung und -Überprüfung implementiert werden.

7.2.1 Rechte-Matrix

Die Matrix ist eine Modifikation einer einfachen `Grid`-Komponente, die dynamisch für jede Rolle eine Spalte generiert und die Zeilen nacheinander aufbaut und hinzufügt. Eine elegantere Lösung wäre es, dies über eine `DataGrid` und diverse `DataTemplates` zu erstellen. Dies ist jedoch schwieriger zu Debuggen, da WPF die Templates dynamisch im Hintergrund erzeugt und diese dadurch während der Laufzeit nur schwer zugänglich sind.

Beim Erstellen der Zeile muss auf drei Fälle geachtet werden: Es existiert eine Verantwortlichkeit, nur ein Recht oder nichts. So ergibt sich je nach Fall ein anderes Verhalten. Bei Verantwortlichkeiten müssen die Events abgeschaltet werden. Bei ”Nichts” muss erst ein Objekt erstellt werden. Die Logik für die Bestimmung des Falles wird in den Rechtemanager ausgelagert.

7.2.2 Rechte-Manager

Die Rechte besitzen Referenzen zu einem Objekt-Typ, einer Rollen, einem Zustand und manche zu einem Attribut. Deswegen kann von diesen Objekten eine Rechteabfrage stattfinden. Um dies zu vereinheitlichen, werden alle Rechteabfragen über einen Rechte-Manager (`RightManager`-Klasse) abgewickelt (vgl. Abb. 7.6), die ähnlich den verschiedenen `ConsistencyManagern` dafür Analyse- und Check-Methoden besitzt.

Die Methoden des Rechte-Managers sind so gestaltet, dass sie von der Modellierungs-

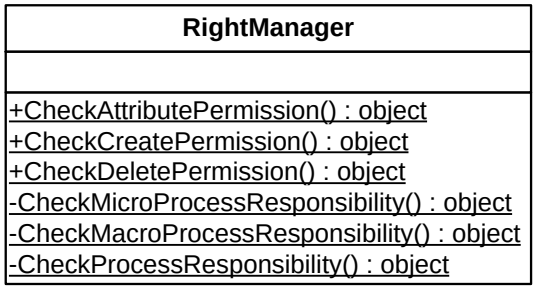


Abbildung 7.6: Klassendiagramm der RightManager-Klasse

umgebung als auch von der Laufzeitumgebung genutzt werden können. Dazu wurde jede Methode so überladen, dass sie die verschiedenen Eingabeparameter verarbeiten kann.

Der Rechte-Manager beachtet auch Verantwortlichkeiten. Diese Verantwortlichkeiten sind Generalbevollmächtigungen für ein Element im Prozess (z.B. Zustand). Dies bedeutet, dass bei einer Rechteüberprüfung auch Verantwortlichkeit berücksichtigt werden müssen. Diese sind in einer Hierarchie angeordnet, wobei die höheren Elemente die unteren enthalten. Die Abb. 7.7 zeigt die Hierarchie der Rechte und der Verantwortlichkeit. Der Rechte-Manager sucht zuerst in der Hierarchie nach einer Verantwortlichkeit. Falls eine Verantwortlichkeit gefunden wurde, so wird `true` zurück gegeben. Andernfalls wird nach dem entsprechenden Recht gesucht. Falls ein Recht gefunden wurde, wird es zurückgegeben. Andernfalls wird ein `null`-Objekt zurückgegeben. Dieser variabler Rückgabetyp hat den Nachteil, das der Entwickler eine Typ-Überprüfung machen muss und wissen muss, welcher Typ zurück gegeben wird. Eine Alternative dazu wäre es, Rückgabe-Container zu benutzen. Diese bedeuten jedoch zusätzlichen Wartungsaufwand und deswegen wurde darauf verzichtet.

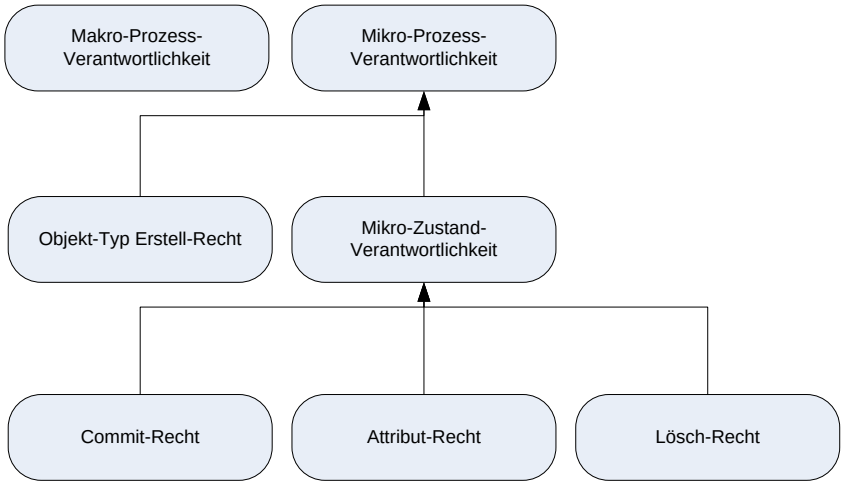


Abbildung 7.7: Rechte und Verantwortlichkeit Hierarchie

7.2.3 Laufzeitumgebung

Der Rechtemanager der Laufzeitumgebung benutzt eine fest programmierte Rechteverwaltung (d.h. die Verknüpfung zwischen die Rollen-Instanzen und den Benutzer-Instanzen sind manuell in die Datenbank eingetragen worden und müssen für neue Benutzer-Instanzen manuell geändert werden). Dies war wegen fehlender Rechteverwaltung nicht anders lösbar. Der Rechtemanager wird im Zuge einer studentischen Arbeit auf den neuen Rechtemanager angepasst.

7.3 Zusammenfassung

Die Benutzerintegration ist sowohl grafisch als auch technisch anspruchsvoll. Technisch mussten die Rechte und Verantwortlichkeiten in das Linq-Klassenmodell eingepflegt werden, und eine `RightManager`-Klasse für die Verwaltung und Überprüfung erstellt werden. Die Veränderung der Laufzeitumgebung ist auf eine nachfolgende studentische Arbeit verschoben worden, da diese erst implementiert werden kann, wenn die Benutzerintegration im Modellierungsumgebung vollständig ist.
Die grafische Seite ist deutlich anspruchsvoller, da viele Strukturen dynamisch sind. Die `RechteMatrix` hat dynamische Spalten und Zeilen, wobei die Zeilen zusätzlich noch ausklappbar sind. Die Zellen bekommen ihren Inhalt aus verschieden Quellen (Rechte oder

Verantwortlichkeiten) und verhalten sich deswegen unterschiedlich.

Das Ergebnis ist eine grafische Oberfläche nahe am Designentwurf, sowie strukturiert erweiterbare Klassen. Am Ende der Arbeit wurde ersichtlich, dass der Designentwurf noch Optimierungspotenzial besitzt.

8 Zusammenfassung und Ausblick

In diesem Kapitel wird eine kurze Zusammenfassung der Arbeit sowie ein Ausblick über die möglichen Arbeitsfelder gegeben.

8.1 Zusammenfassung

Nach einer schwierigen und langen Einarbeitungszeit in das PHILharmonicFlows-Konzept und die zwei Prototypen, war es die Aufgabe, beide Prototypen mit der Datenbank zu verbinden. Dazu mussten erst verschiedene Lösungswege evaluiert werden (vgl. Kapitel 5). Der beste Lösungsweg wurde dann umgesetzt. Darauf aufbauend wurden für den lauffähigen Prototyp wichtige Komponenten implementiert: Werte-Typen (vgl. Kapitel 4.3.4) und Benutzerintegration (vgl. Kapitel 7). Diese sind nach den gegebenen Usability-Designs erstellt und mit allen notwendigen Interaktionen (Erstellen, Löschen und Editieren) implementiert worden. Unterbrochen wurde die Implementation für Tests und Debuggen der Prototypen, um Fehler zu entfernen.
Danach wurde mit der Programmierung der Makro-Prozesse und der Benutzerintegration in die Modellierungsumgebung begonnen (vgl. Kapitel 6). Die Implementierung umfasste sowohl den grafischen, als auch den technischen Aspekt des Usability-Konzeptes. Um den Benutzer zu entlasten, wurden automatische Anpassungen integriert, die auf Veränderungen in den Makro- und Mikro-Prozessen reagieren und diese entsprechend modifizieren.

8.2 Offene Arbeitsfelder

Das PHILharmonicFlows ist ein weitreichendes Konzept bei dem die Prototypen einen großen Teil des Konzepts realisieren. Einige noch offene Arbeitsfelder für die Modellierungsumgebung sind:

- DataContext
- Weitere Makro-Prozess-Ansichten
- Schema-Evolution der Modelle in der Laufzeitumgebung
- Vereinheitlichung der Konsistenzprüfung
- Vereinheitlichung der Oberflächengestaltung
- Ausstatten der grafischen Elemente mit ToolTips
- Undo-/Redo-Funktion
- Mehrbenutzerbetrieb

Die meisten Arbeitsfelder befassen sich mit der Oberfläche des Prototyps und können in studentischen Arbeiten erledigt werden. Die Schema-Evolution und der Mehrbenutzerbetrieb erfordern ein Konzept, das erst erarbeitet werden müsste und sich deswegen gut für Bachelor-/Master-/Diplomarbeiten eignet. Für die Implementierung der Undo-/Redo-Funktion ist möglicherweise ein Wechsel der Technologie notwendig, da Linq dies nicht unterstützt. Alternativen wie SDO besitzen eine Verlaufs-Funktion, welche die Undo-/Redo-Funktion nutzen könnten.
In der Laufzeitumgebung existieren weitere offene Arbeitsfelder, die in der Diplomarbeit [23] besprochen werden.

8.3 Ausblick

Die Prototypen von PHILharmonicFlows bieten ein gutes Grundgerüst und vermitteln einen ersten Eindruck für das Konzept. Was noch fehlt, sind verschiedene Features, beispielsweise ToolTips, Copy&Paste, Undo/Redo, etc..., die man heute als grundlegend erachtet. Um dies zu klären, ist eine genaue Analyse der zur Zeit verfügbaren Prozess-Management-Systeme notwendig.

Ein im Konzept nicht beachteter Aspekt ist die Ausnahmebehandlung bei Rechten. Beispielsweise sollte der Wegfall eines Benutzers mit offenen Prozessen noch geklärt und Lösungen gefunden werden. Dazu könnte ein ”Administrator” mit entsprechenden Befugnissen in die Rechteverwaltung eingebunden werden, der bei solchen Ausnahmefällen ein-

greifen kann.

Ein weiterer Aspekt ist die Einbindung einer Service-Orientierten-Architektur (SOA). In der aktuellen Implementierung der Laufzeitumgebung werden Oberfläche und Logik von einem Programm verwaltet. Mit SOA können Schnittstellen angeboten werden, die andere Programme nutzen können. Beispiele hierfür wären Überwachungsprogramme und Smartphone-Programme. Außerdem könnte damit die Ausführungslogik von der Oberfläche getrennt werden.

Die Datenbank-Technologie Linq sollte auch neu evaluiert werden. Microsoft hat den Support für Linq zu Gunsten ihres neuen ”Entity Framework” zurückgefahren [7]. Linq wird deswegen vermutlich neue Technologien nicht mehr unterstützen und es ist fraglich ob alte Fehler noch behoben werden. Für den derzeitigen Einsatz reicht der Funktionsumfang von Linq aus.

A Testfälle

Die Testfälle sind sehr generisch, da nur die Funktionalität und nicht die Realitätsnähe im Vordergrund steht. Deshalb werden keine Werte genannt.

A.1 Testen der Linq-Verbindung

Die einzelnen Punkte der Testfälle werden durch ein Speichern und Laden des Projektes getestet.

A.1.1 Datenstruktur

Test	**Ergebnis**
Erstellen eines Projekts	Projekt in der Datenbank
Erstellen der Datenstruktur mit mehreren Objekt-Typen, Benutzer-Typen und Werte-Typen	Objekte gespeichert
Erstellen von Relation-Typen zwischen den Objekt-Typen	Objekte gespeichert
Ändern von Relation-Typ	Verschieben eines Relation-Typs ist nicht möglich. Werte können geändert werden
Testen des Datenebenen-Filters, indem unerlaubte Relation-Typen erstellt werden	Erstellen der Relation-Typen unmöglich
Erstellen von Attribut-Typen bei Objekt-Typen	Attribute wurden gespeichert
Erstellen von Attribut-Typen bei Werte-Typen	Attribut-Typ wurden erstellt

Test	Ergebnis
Zuweisen von Objekt-Typ-Attribut-Typen zu Werte-Typ Attributen	Zuweisung wurde übernommen
Erstellen von Instanzen der Werte-Typen	Instanzen wurden gespeichert
Erstellen von ungültigen Instanzen	Dialogfenster zeigte die ungültigen Einträge an
Ändern von Attributen der Werte-Typen	Dialogfenster erschien und Attribut wurde geändert
Löschen von Relation-Typen	erfolgreich
Löschen von Attribut-Typen	erfolgreich
Löschen von Objekt-Typen	erfolgreich und vollständig

A.1.2 Mikro-Prozess

Der Test setzt einen Objekt-Typ mit mindestens zwei Attributen voraus.

Test	Ergebnis
Auswählen des Objekt-Typs	Minimaler Mikro-Prozess wird angezeigt
Erstellen von Zustands-Typen	erfolgreich; nach dem Laden sind diese Vertikal angeordnet
Erstellen von Mikro-Schritt-Typen	erfolgreich; nach dem Laden sind diese Vertikal angeordnet
Erstellen der notwendigen Transition-Typen	erfolgreich; Graph ist topologisch sortiert
Ändern der Transition-Typen	erfolgreich
zuweisen von Attribut-Typen zu einem Mikro-Schritt-Typ	erfolgreich; wenn der Start-Schritt-Typ ein Attribut-Typ bekommt, bricht die Sortierung vorzeitig ab und alle Elemente werden vertikal angeordnet
Testen des Attributs-Filters	Attribut-Typ wird aus der Combobox herausgefiltert
Löschen von Mikro-Schritt-Typen	erfolgreich
Löschen von Zustands-Typen	erfolgreich bei gültigem Prozess; bei ungültigem Prozess wird ein SQL-Fehler geworfen, da der Fremdschlüssel verletzt ist

Test	Ergebnis
Entfernen von Start- und End-Flag	wird gespeichert

A.1.3 Makro-Prozess

Der Test setzt mindestens zwei Objekt-Typen und zwei Mikro-Prozesse mit mehreren Zuständen voraus.

Test	Ergebnis
Auswählen des Makro-Prozesses	der Minimale Makro-Prozess wird in der Zeichenfläche angezeigt
Ziehen eines Makro-Schritt-Typs aus der Sidebar in den Prozess	Leerer Makro-Schritt-Typ erstellt; speichern nicht möglich, da ungültiger Prozess
Zuweisen von Objekt-Typ und Attribut-Typ zu dem Leeren Makro-Schritt-Typ	erfolgreich; speicherbar
Auswählen eines Makro-Prozesses	erfolgreich
Ziehen eines Mikro-Zustand-Typs in die Zeichenfläche	Makro-Schritt-Typ hat entsprechende Werte
Erstellen der Transition-Typen	erfolgreich, wenn Pfad-Regel nicht verletzt wird
Testen des dynamischen Erstellens der Makro-Input-Typen	erfolgreich
Löschen von Transition-Typen	erfolgreich
Löschen von Makro-Schritt-Typen	erfolgreich
Entfernen des Start-Zustands-Typ	Zustand der Start-Schritt-Typs ist leer
Hinzufügen eines Start-Zustands-Typ	Start-Schritt-Typ verweist auf den Zustand-Typ
Hinzufügen eines End-Zustands-Typs	neuer End-Makro-Schritt-Typ, der nicht gelayoutet wurde
Entfernen des End-Zustands-Typs	Makro-Schritt-Typ wird zu einem Normalen Makro-Schritt-Typ
Löschen eines Makro-Schritt-Typs	erfolgreich

A.1.4 Rechte

Der Test setzt mindestens zwei Objekt-Typen, zwei Mikro-Prozesse mit mehreren Zuständen und zugewiesenen Attributen voraus.

Test	Ergebnis
Erstellen eines Benutzer-Typs	Benutzer-Typ ist erstellt
Überprüfen Rolle in der Rollenübersicht	Rolle wurde erstellt
Erstellen einer neuen Rolle mit Relation	erfolgreich; Liste wurde nicht automatisch aktualisiert
Editieren der Rolle	Dialogfenster öffnete aufgrund eines Rechtschreibfehler in der XAML nicht. Dialogfenster zeigte Compiler-Name statt Relationsname
Löschen der Rolle	erfolgreich
Wechseln zur Rechte-Matrix	Expandierte Teile lassen sich nicht einklappen, da Event nicht ausgelöst wurde
Hinzufügen und Entfernen von Rechten	Rechte wurden nicht richtig angezeigt, da die Filterung der Zustände nicht funktionierte
Erstellen einer Instanz-spezifischen Rolle	speicherung erfolgreich; Namen der alten Rolle und der neuen Rolle vertauscht
Löschen der Rolle	nicht möglich, da keine Option dafür

A.1.5 Zusammenfassung

Alle Tests lieferten die erwarteten Ergebnisse. Die GUI reagierte an manchen Stellen nicht wie gewünscht, was auf die Verwendung der falschen Events oder Rechtschreibfehler zurückzuführen war. Komplexe Fehler traten nicht auf. Alle Fehler wurden behoben.
Bei der Benutzung des Objekt-Typ-Browsers trat sporadisch ein Fehler auf, in dem ein Label eine negative Größe besaß. Eine vorhergehende Prüfung der Werte brachte nichts, was auf ein Problem mit parallel laufenden Threads der Frameworks (WPF und yFiles) hindeutet.

A.2 Deployment

Der Test setzt eine Datenstruktur und einen oder mehrere Mikro-Prozesse voraus.

Test	Ergebnis
Deployen des Projekts	Datenbank wurde generiert

A.3 Laufzeitumgebung

Die Laufzeitumgebung konnte nicht getestet werden, da die Rechteverwaltung fehlt. Geändert wurde nur die Datenanbindung, was zu keiner Veränderungen des Programmablaufes führt.

B Quelltexte

```
<Window x:Class="PHILharmonicFlows.gui.dialogs.ValueTypeDialoge"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
           presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ValueType Editor" SizeToContent="WidthAndHeight"
        ShowInTaskbar="False"
        WindowStartupLocation="CenterOwner"
        WindowStyle="ToolWindow"
        ResizeMode="NoResize"
        Background="{StaticResource BaseColor60Brush}"
        Loaded="OnLoaded">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="Auto" />
        </Grid.ColumnDefinitions>
        <Label      Grid.Row="0" Grid.Column="0">Name:</Label>
        <TextBox  Grid.Row="0" Grid.Column="1" x:Name="
           _valueTypeName" VerticalAlignment="Center"
           HorizontalAlignment="Stretch">Empty</TextBox>
```

```
23          <StackPanel Grid.Row="3" Grid.ColumnSpan="2" DockPanel.
               Dock="Bottom" Margin="0,10,0,0" Orientation="Horizontal
               " HorizontalAlignment="Center">
24              <Button Content="Ok" Height="23" Name="buttonOk" Width
                   ="75" Margin="0 0 5 0" IsDefault="True" Style="{
                   StaticResource DefaultButton}" Click="
                   OnOkButtonClicked"/>
25              <Button Content="Cancel" Height="23" Name="
                   buttonCancel" Width="75" Margin="5 0 0 0" IsCancel=
                   "True" Style="{StaticResource DefaultButton}"/>
26          </StackPanel>
27      </Grid>
28  </Window>
```

Listing B.1: XAML-Datei für ValueTypeDialog

```
1   using System.Windows;
2   using System.Windows.Input;
3   using PHILharmonicFlows.Database.Modell;
4
5   namespace PHILharmonicFlows.gui.dialogs
6   {
7       /// <summary>
8       /// Interaction logic for ValueTypeDialoge.xaml
9       /// </summary>
10      public partial class ValueTypeDialoge : Window
11      {
12          public ValueTypeDialoge()
13          {
14              InitializeComponent();
15          }
16
17          private void OnLoaded( object sender, RoutedEventArgs e )
18          {
19              Keyboard.Focus(_valueTypeName);
```

```
20          }
21
22          private void OnOkButtonClicked(object sender,
                RoutedEventArgs e)
23          {
24              var model = DatabaseManager.GetDatabaseManager().
                    GetCurrentModel();
25              model.AddNewValueType(_valueTypeName.Text);
26              DialogResult = true;
27          }
28      }
29  }
```

Listing B.2: XAML.CS-Datei für ValueTypeDialog

```
1  <ResourceDictionary xmlns="http://schemas.microsoft.com/winfx
       /2006/xaml/presentation"
2                      xmlns:x="http://schemas.microsoft.com/winfx
                           /2006/xaml" xmlns:controls="clr-
                           namespace:PHILharmonicFlows.controls">
3
4  <Style x:Key="DefaultButton" TargetType="Button">
5          <Setter Property="Margin" Value="0"/>
6          <Setter Property="Background" Value="{StaticResource
               BaseColor90Brush}"/>
7      <Setter Property="Foreground" Value="White"/>
8      <Setter Property="Template">
9          <Setter.Value>
10         <ControlTemplate TargetType="Button">
11                 <Border Name="border" BorderThickness="0" Margin="
                       2" BorderBrush="DarkGray" CornerRadius="3"
                       Background="{TemplateBinding Background}">
12                      <Grid>
```

```
13                          <ContentPresenter HorizontalAlignment="
                               Center" VerticalAlignment="Center" Name
                               ="content"/>
14                  </Grid>
15                  </Border>
16                  <ControlTemplate.Triggers>
17                          <Trigger Property="IsMouseOver" Value="
                               True">
18                          <Setter Property="Background"  Value="{
                               StaticResource NavigationColor70Brush}"
                               />
19                          <Setter Property="Foreground"  Value="
                               Black"/>
20                          <Setter Property="BitmapEffect" TargetName
                               ="border">
21                               <Setter.Value>
22                                       <OuterGlowBitmapEffect
                                            GlowColor="{
                                            StaticResource
                                            NavigationColor70}"
                                            GlowSize="4"/>
23                               </Setter.Value>
24                          </Setter>
25                  </Trigger>
26                  <Trigger Property="IsPressed" Value="True">
27                          <Setter Property="Background"  Value="{
                               StaticResource SelectionColor70Brush}"/
                               >
28                          <Setter Property="Foreground"  Value="
                               Black"/>
29                  </Trigger>
30                  <Trigger Property="IsEnabled" Value="False">
31                          <Setter Property="Opacity" Value="0.5" />
32                  </Trigger>
```

```
33                          </ControlTemplate.Triggers>
34          </ControlTemplate>
35          </Setter.Value>
36      </Setter>
37  </Style>
38  </ResourceDictionary>
```

Listing B.3: Style-Datei für einen Knopf

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] BECK, Hannes: *Implementierung einer Komponente zur Modellierung von Mikro-Prozessen in einem datenorientierten Prozess-Management-System*, University Ulm, Diplomarbeit, 2012

[2] BERNTSSON, Ola: *LinqBugs*. http://social.msdn.microsoft.com/forums/en-US/linqprojectgeneral/thread/0c494803-249c-4f5d-91c0-0d5ed90366d6/

[3] DADAM, Peter ; REICHERT, Manfred ; RINDERLE-MA, Stefanie ; GOESER, Kevin ; KREHER, Ulrich ; JURISCH, Martin: *Von ADEPT zur AristaFlow BPM Suite - Eine Vision wird Realität: "Correctness by Construction" und flexible, robuste Ausführung von Unternehmensprozessen*. Ulm : University of Ulm, Faculty of Electrical Engineering and Computer Science, 2009 (UIB-2009-02). http://dbis.eprints.uni-ulm.de/488/

[4] FOUNDATION, Eclipse: *Eclipse*. http://www.eclipse.org/org/

[5] FREUND, Jakob ; RÜCKER, Bernd ; HENNING, Thomas: *Praxishandbuch BPMN*. Carl Hanser Verlag, 2010

[6] GROUP, TGI: *Petri Nets World*. http://www.informatik.uni-hamburg.de/TGI/PetriNets/index.html

[7] HEIKNIEMI, Jouni: *Entity Framework – a sorry history, new release 5.0 and future openness*. http://www.redmond-recap.com/2012/08/27/entity-framework-a-sorry-history-new-release-5-0-and-future-openness/

[8] JSON: *Introducing JSON*. http://www.json.org/

[9] KÜHNEL, Andreas: *Visual C# 2010*. Galileo Computing, 2010

[10] KÜNZLE, Vera ; REICHERT, Manfred: PHILharmonicFlows: Towards a Framework for Object-aware Process Management. In: *Software Process: Improvement and Practice* Institute of Databases and Information Systems, University Ulm, 2009

[11] KÜNZLE, Vera ; REICHERT, Manfred: *PHILharmonic Flows MODELING*, University Ulm, Diss., Mai 2010

[12] KÜNZLE, Vera ; REICHERT, Manfred: *PHILharmonic Flows Runtime*, University Ulm, Diss., Mai 2010

[13] LOHRER, Matthias: *Einstieg in ASP.NET*. Galileo Computing, 2003

[14] LUDEWIG, Jochen ; LICHTER, Horst: *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt, 2010

[15] MICROSOFT: *Grundlagen zu Abfrageausdrücken (C#-Programmierhandbuch)*. `http://msdn.microsoft.com/de-de/library/bb384065%28v=vs.90%29.aspx`

[16] MICROSOFT: *Microsoft Common-Language-Runtime*. `http://msdn.microsoft.com/de-de/library/vstudio/8bs2ecf4.aspx`

[17] MICROSOFT: *Microsoft Workflow Fundation*. `http://msdn.microsoft.com/de-de/library/cc431274.aspx`

[18] MICROSOFT: *Silverlight*. `http://www.microsoft.com/web/page.aspx?templang=de-de&chunkfile=special/silverlight.html`

[19] PCOUPLAND: *Ajax Control Toolkit*. `http://ajaxcontroltoolkit.codeplex.com/`

[20] PHP: *PHP*. `http://www.php.net/`

[21] PRÖBSTLE, Andreas: *Technische Konzeption und Realisierung der Modellierungskomponente für ein datenorientiertes Prozess-Management-System*, Univerity Ulm, Diplomarbeit, 2011

[22] SCHEB, Christian: *Entwicklung eines Usability-Konzepts für die Laufzeitumgebung eines datenorientierten Prozess-Management-Systems*, University Ulm, Diplomarbeit, Nov 2010

[23] SCHULZ, Stefan: *Implementierung einer Komponente zur Ausführung von Mikro-Prozessen in einem datenorientierten Prozess-Management-System*, University Ulm, Diplomarbeit, Mar 2012

[24] SYCH, Oleg: *T4 Toolbox*. `http://t4toolbox.codeplex.com/`

[25] ULLENBOOM, Christian: *Java 7 - Mehr als eine Insel*. Galileo Computing, 2012

[26] WAGNER, Michael: *SQL,XML:2006 : Evaluierung der Standardkonformität ausgewählter Datenbanksysteme / Michael Wagner*. Diplomica-Verl., 2010

[27] WAGNER, Nicole: *Entwicklung eines Usability-Konzepts für die Modellierungsumgebung eines datenorientierten Prozess-Management-Systems*, University Ulm, Diplomarbeit, 2010

[28] WELIE.COM: *Accordion*. `http://www.welie.com/patterns/showPattern.php?patternID=accordion/`

[29] WENZ, Christian: *JavaScript und AJAX*. Galileo Computing, 2007

[30] XAMARIN: *Mono*. `http://www.mono-project.com/Main_Page`

[31] YWORKS: *yFiles for WPF*. `http://www.yworks.com/de/products_yfileswpf_about.html`

[32] YWORKS: *yFiles WPF Developer's Guide (UI Part)*. `http://docs.yworks.com/yfileswpf/Index.html?topic=dguide/yFilesDGLayout/analysis.html`

Name: Thomas Spindler Matrikelnummer: 616986

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Thomas Spindler