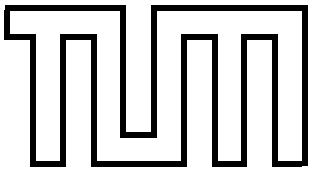


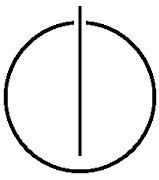
FAKULTÄT FÜR INFORMATIK

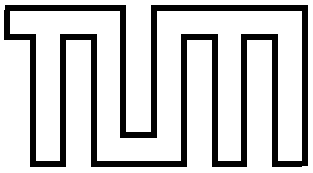
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Wirtschaftsinformatik

# Nutzungsszenarien introspektiver Modelle

Julian Sommerfeldt





FAKULTÄT FÜR INFORMATIK

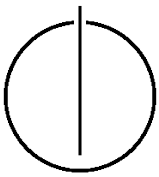
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Wirtschaftsinformatik

# Nutzungsszenarien introspektiver Modelle

# Usecases of intropective models


| | |
|---|---|
| Autor: | Julian Sommerfeldt |
| Themensteller: | Prof. Dr.rer.nat. Florian Matthes |
| Betreuer: | Dr. Thomas Büchner, Christian Neubert |
| Datum: | 10.08.2010 |

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

Baierbrunn, den 10.08.2010 .....

(Julian Sommerfeldt)

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

## 1.1 Modelle und Probleme

Je größer eine Software wird, desto komplexer werden auch die Zusammenhänge innerhalb des Systems und desto schwieriger ist es, den Überblick zu behalten. Hierzu dienen unterschiedliche Arten von Modellen, die sich auf das Wesentliche einer bestimmten Fragestellung konzentrieren, indem die Sachverhalte abstrahiert werden. Mit Hilfe dieser Modelle ist es besser möglich, die Arbeitsweise und den Aufbau eines Systems nachzuvollziehen und auf dieser Basis Entscheidungen zu treffen. Aufgrund dessen finden beispielsweise UML-Klassendiagramme bereits heute sehr verbreitet Anwendung und haben sich gerade in der Kommunikation zwischen den unterschiedlichen Stakeholdern bewährt.

Ein Nachteil von Modellen ist, dass deren Erstellung und Wartung relativ aufwendig, ihr Nutzen aber nicht direkt messbar ist. Dies führt in der Softwareentwicklung häufig zur Vernachlässigung der Pflege. Da sich ein System ständig weiterentwickelt, führt das dann automatisch zu Inkonsistenz zwischen Code und Modell. Dies kann bis zur Unbrauchbarkeit führen beziehungsweise eine gefährliche Fehlerquelle darstellen, da die Modelle während der Entwicklung oftmals als Informationsgrundlage herangezogen werden.

Dieser Problematik entgegen wirken kann die modellgetriebene Softwareentwicklung, indem sie die Modelle zu First-Class-Objekten macht. Das heißt, dass sie Teil der Software sind und nicht mehr wie in der modellbasierten Entwicklung parallel gepflegt werden müssen. Die starke Integration von Code und Modell führt automatisch zu konsistenten Modellen, die immer aktuell sind.

Diesen Ansatz verfolgt das Open Source Projekt Tricia, das am Lehrstuhl Software Engineering for Business Information Systems an der Technischen Universität München entwickelt wird. Es dient als Software für Onlinezusammenarbeit und zur Wissens- und Datenverwaltung. Tricia baut auf einem sogenannten introspektiven Framework auf. Durch die Introspektion wird die Extraktion eines Modells aus Implementierungsartefakten ermöglicht.

## 1.2 Ziele und Aufbau der Arbeit

Diese Arbeit beschreibt Möglichkeiten, die, durch Introspektion gewonnenen, Modelle sinnvoll zu nutzen. Hierzu werden drei mögliche Verwendungen und deren Realisierungen vorgestellt. Um für die nachfolgenden Abschnitte Begriffe und Zusammenhänge zu klären, werden die Grundlagen in Kapitel 2 gelegt. Kapitel 3 beschreibt dann die Erweiterung der Tricia-Dokumentation, die durch Javadoc realisiert wird. Hierzu wird der Zugang dazu über ein Wiki ermöglicht. Da Datenmodelle in Tricia eine zentrale Rolle spielen, werden diese in Kapitel 4 behandelt. In Kapitel 5 wird daraufhin die Visualisierung dieser Datenmodelle vorgestellt. Kapitel 6 erklärt das sogenannte Round Trip Engineering und die praktische Umsetzung dazu. Eine Zusammenfassung über die Arbeit gibt dann das Schlusskapitel 7.

# Kapitel 2

# Grundlagen

Dieses Kapitel beschreibt einige grundlegende Begriffe, deren Verständnis für die späteren Ausführungen essentiell sind. 2.1 und 2.2 erklären Modelle im Kontext dieser Arbeit beziehungsweise modellgetriebener Softwareentwicklung, die die Basis für die automatische Modellgenerierung darstellt. Anschließend wird in 2.3 erläutert, wie Round Trip Engineering funktioniert, und wie es in den Prozess der Softwareentwicklung passt. In 2.4 wird Tricia vorgestellt, und beschrieben, wie die Introspektion funktioniert. Abschließend wird in 2.5 die Motivation für und das Vorgehen innerhalb dieser Arbeit begründet.

## 2.1 Modell

Als Modell wird im Allgemeinen ein Abbild der Wirklichkeit bezeichnet. Als solches sind Modelle in vielen Bereichen vertreten und existieren in unterschiedlichster Ausprägung, wobei diese Arbeit sich mit Modellen aus der Informatik und Softwareentwicklung beschäftigt. Allen Modellformen gemeinsam sind die drei wesentlichen Eigenschaften eines Modells nach Herbert Stachowiak [10]:

- ***Abbildung*** Das Modell beschreibt einen Gegenstand der Realität. In der Softwareentwicklung handelt es sich hierbei häufig um Implementierungsartefakte.
- ***Verkürzung*** Da ein Modell der Abstraktion und der Verbesserung der Verständlichkeit dient, ist es notwendig, dass es sich auf die wesentlichen Bestandteile des Zugrundeliegenden konzentriert. So ist es beispielsweise häufig sinnvoll, Implementierungsdetails auszublenden.
- ***Pragmatismus*** Ein Modell wird nicht zum Selbstzweck erstellt, man muss sich immer die Fragen *Für wen?*, *Warum?* und *Wozu?* stellen.

Modelle können durch diese Eigenschaften helfen, komplexe Zusammenhänge zu abstrahieren und somit einfacher zu verstehende Sichten auf die Modelle bereitstellen. Ein weit verbreitetes

Beispiel hierfür sind UML-Klassendiagramme [13], die in der Informatik als ein Standard für die Darstellung objektorientierter Systeme angesehen werden kann.

### 2.1.1 Metamodell

Wenn Modelle andere Modell repräsentieren, dann spricht man von Metamodellen. Diese Metamodelle sind also den anderen übergeordnet und werden von der darunter liegenden Ebene instanziert, wodurch das Schema vorgegeben wird. Die Object Management Group [11] hat mit der Meta Object Facility [12] eine Grundlage für Metamodelle geschaffen. Abbildung 2.1 zeigt die vier Metaebenen an Hand eines Buches. Das UML-Klassendiagramm [13] ist eine Instanz der Ebene M3 und beschreibt die Form der Klassen innerhalb eines Modells der Ebene M1. Das Buch instanziert Class und verfügt über ein Attribut title. Reale Instanzen können dann auf Ebene M0 aus der Klassendefinition erzeugt werden.

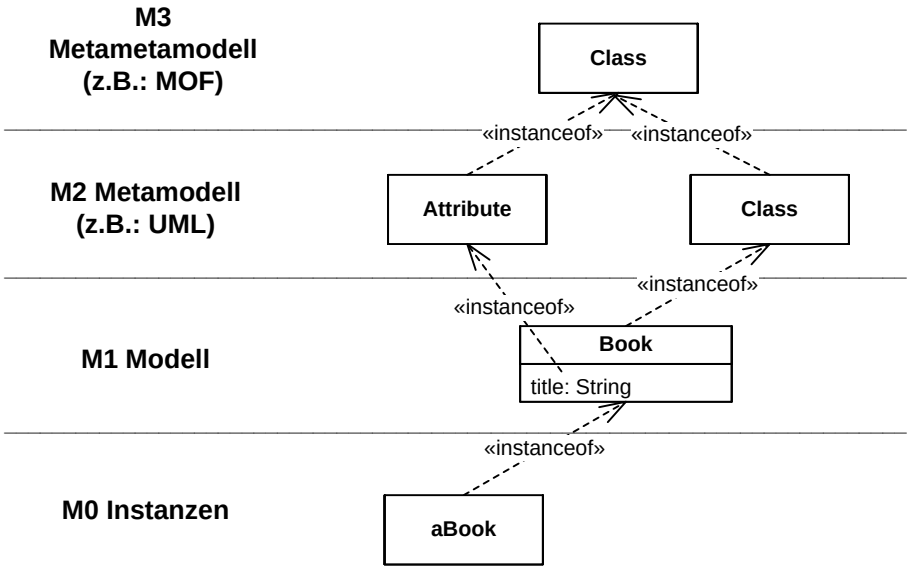



Abbildung 2.1: Metaebenen der OMG

## 2.2 Modellgetriebene Softwareentwicklung

Bisher findet der Einsatz von Modellen in der Softwareentwicklung primär modellbasiert statt. Das heißt, Modelle werden parallel zum Code entwickelt, verwaltet und gepflegt. Entwickler vernachlässigen diese Modelle häufig, da sie nicht immer direkten Nutzen daraus ziehen können. Es handelt sich also eher um ein langfristiges Engagement, da diese Modelle beispielsweise auch für Dokumentationszwecke benutzt werden können. Diese Vernachlässigung der Modellwartung im Entwicklungsprozess führt zwangsläufig zu inkonsistenten Modellen.

Da Modelle aber als zentrale Informationsquellen dienen, hat diese Problematik schwerwiegende Folgen. So können beispielsweise veraltete Versionen falsche Entscheidungen hervorrufen,

wenn diese als Grundlage dienen. Hier setzt die modellgetriebene Softwareentwicklung, engl. Model-driven software development (MDSD), an, bei der Modelle keine Nebenobjekte sondern *First-Class-Objects* und somit Teil der Software sind. Sie werden direkt integriert und können dadurch keine Inkonsistenz mehr aufweisen [21]. Nach [3] hat MDSD im Wesentlichen die Aufgaben, die Verwaltung des Lebenszyklus von Software einerseits zu vereinfachen und andererseits zu formalisieren, wodurch Automatismen ermöglicht werden. Entsprechend diesem Ansatz können noch zwei weitere Forderungen für Modelle aufgestellt werden:

- ***Konsistenz*** Zur Vermeidung von fehlerhaften Informationen ist die Konsistenz zwischen Code und Modell unbedingt zu gewährleisten.
- ***Automatismus*** Durch das Formalisieren und Standardisieren sollte es möglich sein, gewisse Automatismen zu realisieren. Für diese Arbeit zentral ist dabei die Frage der automatischen Modellgenerierung, also der Introspektion.

Wenn bei der Softwareentwicklung und der Modellverwendung auf diese Aspekte Rücksicht gelegt wird, kann dadurch die Software- und Dokumentationsqualität erhöht und die Komplexität reduziert werden. Der Vorgang der Modellgenerierung aus Codefragmenten wird im Allgemeinen Reverse Engineering [23] genannt, wobei folgend von Introspektion gesprochen wird.

## 2.3 Round Trip Engineering

Der Nutzen eines Modells richtet sich stark nach dessen Aktualität und vor allem Konsistenz mit dem zugrunde liegenden Code. Round Trip Engineering ist eine Möglichkeit, diese Konsistenz umfassend zu gewährleisten. Nach [20] stellt es im Wesentlichen die Kombination aus Forward Engineering [22] und Reverse Engineering und einer Komponente, die die Rahmenbedingungen festlegt, dar.

Das bisher weiter verbreitete Reverse Engineering ist das Erzeugen von Modellen aus Programmcode, bei uns Introspektion genannt. Das Gegenstück Forward Engineering schließt den Kreis, indem es aus einem Modell beziehungsweise einer Sicht auf ein Modell Code generiert. Bei beiden Transformationen ist die Einbeziehung des Metamodells (siehe 4.2) und des Metametamodells [5, S. 82ff] von Tricia wichtig, da dadurch die Rahmenbedingungen bestimmt werden. Ohne dieses Bindeglied wären die beiden Engineerings ungesteuert und es wäre nicht definiert, welche Ergebnisse sie liefern. Insofern gibt das Metamodell beispielsweise die Formatierung gewisser Codefragmente vor. Andersherum wird dadurch festgelegt, wie das introspektive Modell, das aus dem Code gewonnen wird, aufgebaut ist. Somit bilden diese Informationen die Basis für das Round Trip Engineering und sind ein essentieller Faktor für die Erreichung der Konsistenz. Der zyklische Prozess des Round Trip Engineerings ist schematisch in Abbildung 2.2 nachvollziehbar.

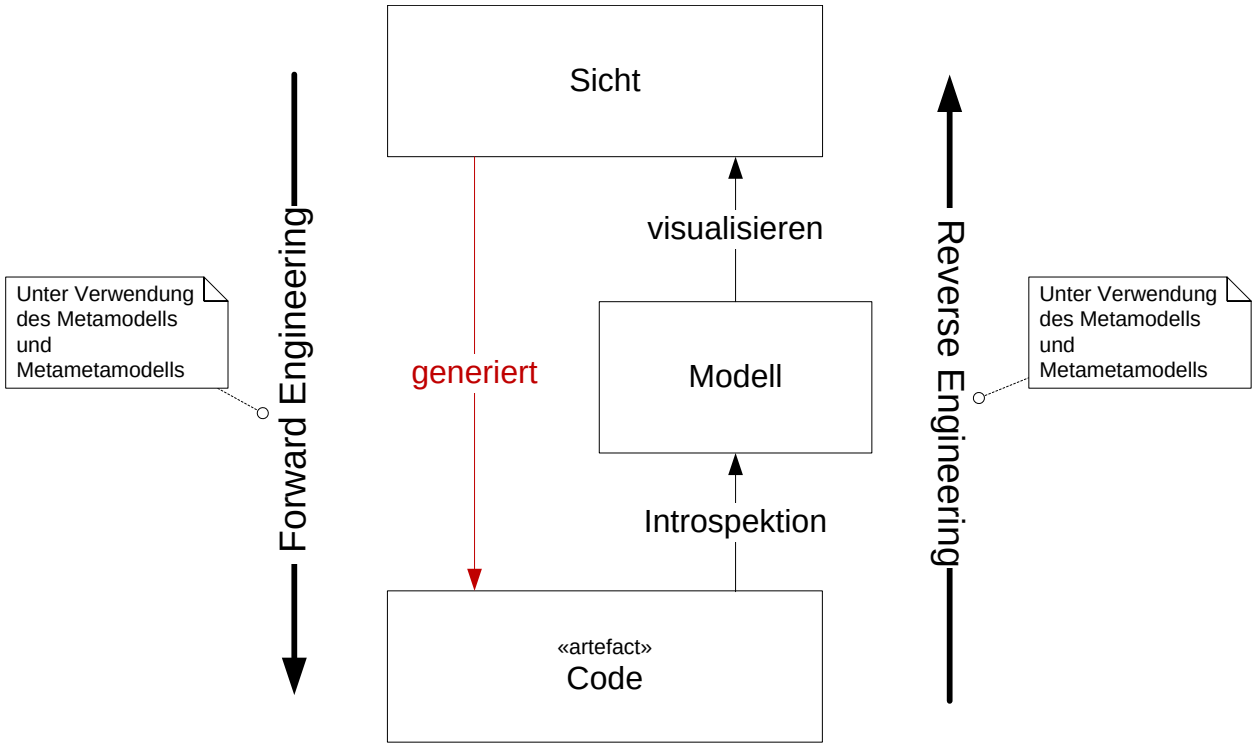



Abbildung 2.2: Round Trip Engineering

Auch beim Round Trip Engineering ist die Motivation bedingt durch steigende Komplexität der Programmarchitekturen. Es wird versucht, den Erstellungs- und Änderungsvorgang von Code zu vereinfachen. Erreicht wird dies durch Abstraktion und somit einer Reduzierung der Komplexität. Dies erleichtert, genau wie die Modelle an sich, den Zugang zum Verständnis und Umgang mit den vorliegenden Strukturen. Die Realisierung des Round Trip Engineerings wird in Kapitel 6 besprochen.

## 2.4 Tricia

Tricia ist eine Open Source Web Collaboration and Knowledge Management Software [2], die an der Technischen Universität München am Lehrstuhl Software Engineering for Business Information Systems [1] entwickelt wird. Es handelt sich um eine Internetplattform, die unterschiedliche sozial orientierte Systeme vereinigt und somit umfassende Zusammenarbeit ermöglicht. So beinhaltet unter anderem Wikis, Blogs und File Sharing. Darüber hinaus kann es aufgrund des Personenmanagements auch als Social Network angesehen werden.

### Introspektion

Programmiertechnisch ist Tricia pluginbasiert und lässt sich deshalb sehr gut erweitern. Aufgrund extra für Tricia entwickelter Eclipse-Plugins [25] wird für Entwickler eine sehr gute Eclipse-Integration geboten. Unter anderem wird dadurch die bereits angesprochene Introspek-

tion ermöglicht. Man kann automatisch generische Modelle aus dem Triciacode erzeugen, wobei es das Interaktionsmodell, das Berechtigungsmodell und das Datenmodell gibt. Die vorliegende Arbeit beschäftigt sich im Wesentlichen mit dem Datenmodell, das auch als ein Herzstück von Tricia angesehen werden kann. Für diese Arbeit ist es ausreichend, den Vorgang der Introspektion wie in Darstellung 2.3 zu verstehen, wobei er in [5] auch näher nachvollziehbar ist.

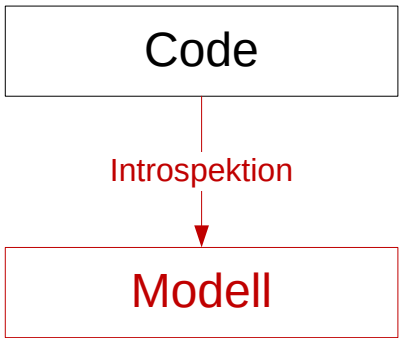



Abbildung 2.3: Introspektion

## 2.5 Motivation und Vorgehen

Nachdem jederzeit introspektive generische Modelle gewonnen werden können, stellt sich die Frage der Nutzung. Welche Stakeholder haben also wann an welcher Verwendung des Modells Interesse? Wie lassen sich die Informationen effizient verarbeiten und welche Nutzungsszenarien sind dafür denkbar?

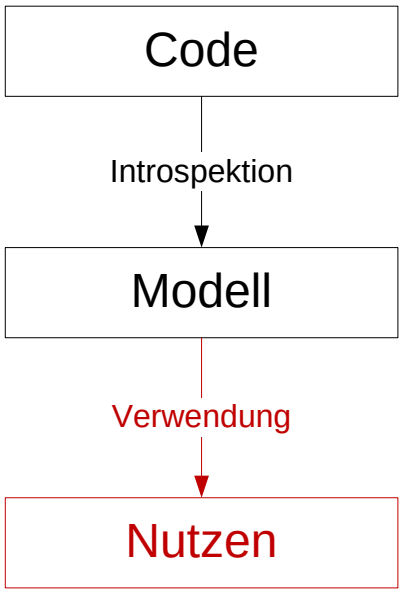



Abbildung 2.4: Modellverwendung

Hierzu gliedert sich diese Arbeit im Wesentlichen in drei Teile. Kapitel 3 beschreibt die Erweiterung der, durch Javadoc realisierten, Dokumentation. Die Kapitel 5 und 6 beschäftigen sich mit dem von der Introspektion bereitgestellten Datenmodell. Ersteres beschreibt die Visualisierung solcher Modelle. Zweiteres geht auf die Realisierung des in 2.3 erklärten Round Trip Engineerings ein.

# Kapitel 3

# Javadoc-Import

Die Dokumentation von Tricia wird durch Javadoc realisiert. Zu Beginn dieses Kapitels wird in 3.1 erklärt, was Javadoc auszeichnet und warum es als Dokumentationsmittel gewählt wurde. Welche Probleme es beim Zugriff auf die Dokumentationsinformationen gibt und weshalb es notwendig ist, die Erreichbarkeit zu verbessern wird in 3.1.2 begründet. Die erhöhte Verfügbarkeit wird durch die Bereitstellung der Dokumentation in einem Wiki geschaffen. Die Implementierung dieses Ansatzes wird in 3.2 beschrieben, worauf 3.3 das Ergebnis vorstellt.

## 3.1 Javadoc

Javadoc ist eine Software, die aus Java-Quellcode und Kommentaren Dokumentationen erstellen kann [18]. Es zeichnet sich besonders durch die hohe Integration in den Code aus und weist dadurch eine relativ hohe Konsistenz auf. Durch die einfache Handhabung für Entwickler hat Javadoc einen sehr hohen Bekanntheitsgrad erreicht und wird heutzutage weit verbreitet genutzt. Diese Vorteile haben in Tricia zur Wahl von Javadoc als Dokumentationsmittel geführt.

### 3.1.1 Javadoc als introspektives Modell

Da sich diese Arbeit mit introspektiven Modellen befasst, wird hier begründet, weshalb Javadoc ebenfalls als solches angesehen werden kann. Hierzu müssen die in 2.1 und 2.2 angeführten Forderungen erfüllt werden.

- ***Abbildung*** Javadoc wird dazu benutzt, Implementierungsartefakte näher zu beschreiben.
- ***Verkürzung*** Durch die Kommentare wird es möglich, auf, für den Modellnutzer, irrelevante Inhalte - wie Implementierungsdetails - zu verzichten.
- ***Pragmatismus*** Die Verfasser der Javadoc-Kommentare werden von den drei oben beschriebenen Fragen *Für wen?*, *Warum?* und *Wozu?* geleitet.

- ***Konsistenz*** Die bestehende *Integration*, also die Tatsache, dass die Dokumentationsquelle Teil des Codes darstellt, ist ein zentraler Aspekt, um hohe Konsistenz gewährleisten zu können.
- ***Introspektion*** Die Integration und die Funktionsweise des Tools ermöglichen die Extraktion der Modellinformationen aus dem Quellcode.

### 3.1.2 Probleme und Lösungsansätze

Aufgrund der hohen Integration in den Quellcode, ist es bisher nur möglich, die Kommentare direkt dort zu bearbeiten. Bei der Erstellung der Dokumentation ist dann der gesamte Code notwendig, da aus diesem und den Kommentaren die komplette Dokumentation generiert wird. Genauso ist die Verwendung beschränkt darauf, dass Entwickler die fertige Dokumentation verteilen, was vor allem bei Änderungen schnell zu inkonsistenten unterschiedlichen Versionen führt. Deshalb ist es wünschenswert, eine zentrale Instanz zu verwalten, so dass diese immer konsistent und gut verfügbar ist. Für diese Anwendung eignet sich ein Wiki, dass die einzelnen Dokumentationsinhalte in Wikiseiten verwaltet. Darüber hinaus wird auch ermöglicht, dass Benutzer des Wikis einzelne Seiten editieren.

## 3.2 Implementierung

Eine zentrale Instanz des Wikis, in dem die Dokumentation gespeichert wird, sollte immer konsistent und somit aktuell sein. Um das zu erreichen, muss die Aktualisierung automatisiert werden. Das heißt, so bald sich etwas am Code oder den Javadoc-Kommentaren ändert, muss die Dokumentation auf den neuesten Stand gebracht werden. Wie das realisiert wird, ist in Abbildung 3.1 dargestellt. Folgend werden die einzelnen Schritte erklärt.

**1** Der Entwickler arbeitet am Triciacode und verändert diesen und die zugehörigen Javadoc-Kommentare in der Datei. Um die Änderungen zu sichern und allen Mitentwicklern zu Verfügung zu stellen, werden diese in ein Online-Repository geschrieben. In diesem Fall handelt es sich um BitBucket [7].

**2 & 3** Ein Continous-Integration-Server, hier Hudson [8], dessen Funktionsweise in 3.2.1 näher erklärt wird, bemerkt die Veränderung und führt einen CheckOut durch. Dieser kann dann einen build durchführen, Tests durchlaufen und Benachrichtigungen über den Erfolg oder Misserfolg versenden.

**4** Nach dem CheckOut wird ein Programm, das den Import in das Dokumentationswiki ausführt, aufgerufen. Der eigentliche Import wird in 3.2.2 beschrieben.

**5** Die Grundfunktionen eines Wikis erlauben nun jedem Benutzer die Wikipages einzusehen beziehungsweise, differenziert authorisierbar, zu editieren. Dadurch besteht ein einfacher, hoch verfügbarer und konsistenter Zugang zur Dokumentation.

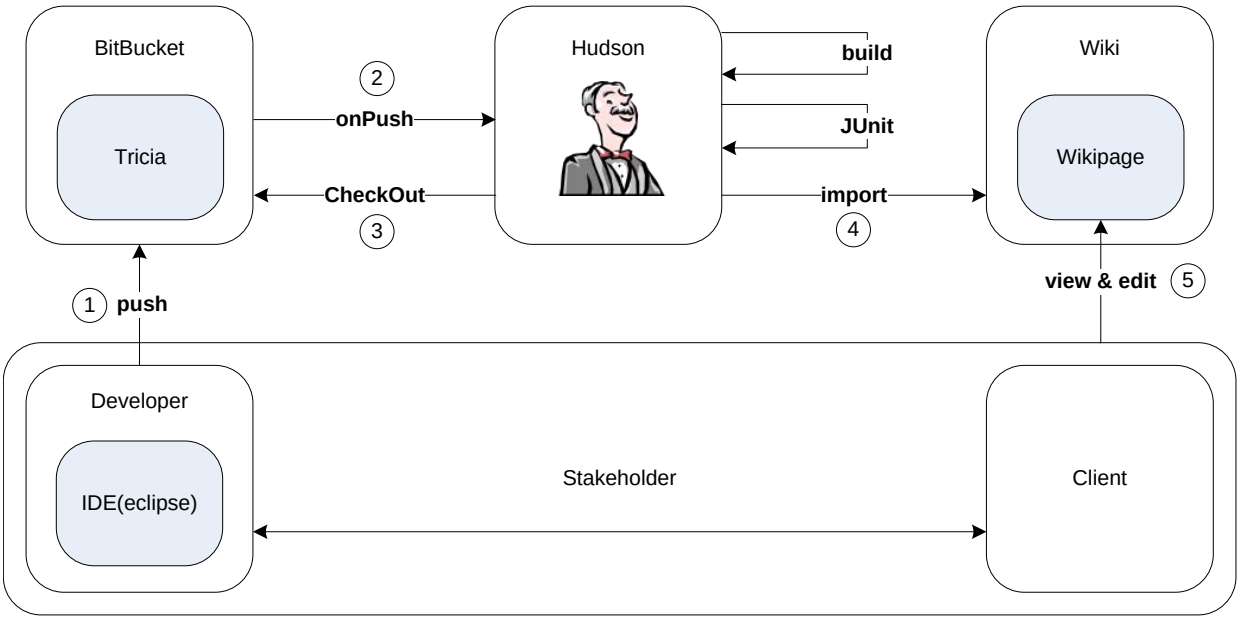



Abbildung 3.1: Automatismus im Javadoc-Import

### 3.2.1 Hudson

Als sogenannter Continous-Integration-Server ist der Hudson [8] für das regelmäßige Bilden und Testen einer Anwendung verantwortlich. In diesem Fall stößt er nach dem CheckOut der Triciadateien eine `build.xml` für das Programm Apache Ant [24] an. Dieses *Build-Managment-Tool* ermöglicht die automatisierte Ausführung bestimmter Befehle. Hier wird ein Javadoc-Befehl angestoßen, dem ein spezielles Doclet übergeben wird, das für den Import in das Wiki zuständig ist. Die Funktionsweise des Doclets wird in 3.2.2 erklärt.

### 3.2.2 Das Doclet und der Server

Wenn Javadoc ohne spezielles Doclet aufgerufen wird, wird das Standarddoclet benutzt, welches eine HTML-Dokumentation erstellt. Innerhalb eines selbst definierten Doclets besteht die Möglichkeit, über ein `RootDoc`-Objekt auf alle enthaltene Klassen, Methoden und Kommentare zuzugreifen. Diese Informationen können dann beliebig weiterverwendet werden. So werden die Inhalte erst an das für das Wiki benötigte HTML [32] angepasst und dann über HTTP-Requests [33] importiert.

Da Verlinkungen zu anderen Klassen oder Bildern in Javadoc in Form von Tags angegeben werden, müssen diese entsprechend durch HTML-Links ersetzt werden, bevor sie in das Wiki geschrieben werden können. So wird beispielsweise aus

```
{@link WhatIsTriciaDoc}
```

folgendes:

```
<a href="de-infoasset-platform-documentation-whatistriciadoc">
     WhatIsTriciaDoc</a>
```

Wie man sieht, müssen ebenfalls die Packetpfade, in denen sich die jeweilige Klasse befindet, mit einbezogen werden, damit exakte Zuweisungen garantieren werden können. Diese Information kann über die Java-Imports bezogen werden. Des Weiteren ist es notwendig, die Dateien, wie Bilder, korrekt zu verlinken. Hierzu wird beispielhaft

```
<img src="{@docRoot}/images/asset-big-picture.png"/>
```

durch

```
<img src="/file/attachments/wikis/javadoc-import-wiki/
de-infoasset-platform-documentation-assetbigpicturedoc/asset-big-picture.png"/>
```

ersetzt.

Die so angepassten Inhalte werden dann über ein HTTP-Request an den Server übermittelt. Dafür wurde ein Tricia-Plugin entwickelt, welches die Anfragen annimmt und die übermittelten Daten in das vorher angelegte Wiki schreibt. Um zu verhindern, dass editierte Wikiseiten überschrieben werden, wird ein zeitbasierter Vergleich durchgeführt und bei Abweichungen ein Administrator benachrichtigt, der die Änderungen überprüfen kann.

Im letzten Schritt werden - ebenfalls über HTTP-Requests - die Dateien in die entsprechenden Ordner hochgeladen um die Dokumentation zu vervollständigen.

## 3.3 Ergebnis

Eine Seite des Wikis ist in Abbildung 3.2 zu sehen. Oben in der „Positionsangabe“ kann man die portierte Packet-Struktur erkennen (1). Des Weiteren sieht man die unterschiedlichen übernommenen Elemente, wie Links zu anderen Klassen und Methoden (2) und die importierten Bilder (3). Die vom Wiki bereitgestellte Personalisierung, wie sie oben rechts erkennbar ist (4), erlaubt, bestimmten Benutzern Schreibrechte zu geben, so dass nur authorisierte Entwickler Änderungen vornehmen dürfen. Zu finden ist diese Dokumentation auf der Internetseite von [2].

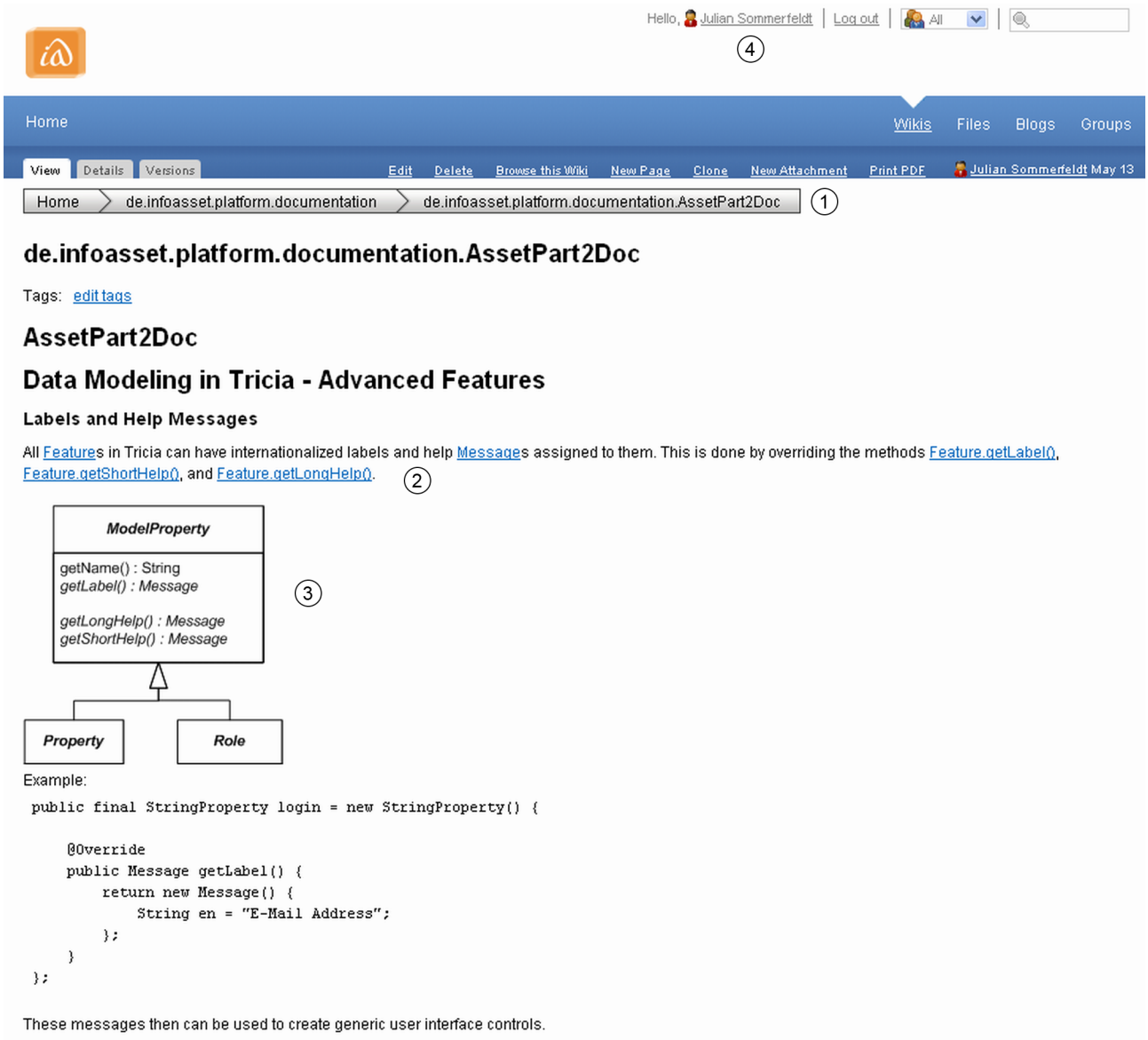



Abbildung 3.2: Javadoc-Import in Tricia

### 3.3.1 Ausblick

Wie oben angesprochen soll diese Umsetzung auch dazu dienen, dass die Dokumentationsinhalte verändert werden können. Damit aber zwischen Quellcode und dem Wiki Konsistenz herrscht, ist es notwendig, dass im Wiki editierte Daten auch zurückgeschrieben werden. Dies ist eine für die Zukunft geplante Erweiterung, die in dieser Arbeit aber nicht realisiert wird.

Die Realisierung dieses Reimports wäre relativ aufwendig. So ist es beispielsweise notwendig, die unterschiedlichen Quellen zu synchronisieren, da ein Mehrbenutzerbetrieb möglich ist. Eine weitere Fragestellung betrifft die Konsistenz zwischen Code und Dokumentation, wenn diese auf der Wikiseite bearbeitet wird. Da dem Schreibenden der Code nicht direkt zur Verfügung steht, kann es vorkommen, dass inhaltlich falsche Kommentare verfasst werden, die nicht zum Code selber passen.

# Kapitel 4

# Datenmodellierung mit Tricia

Dieses Kapitel beschreibt, wie in Tricia Daten modelliert werden. Für die Modellierung der Datenobjekte, die in Tricia Assets genannt werden, dient das Metamodell als Basis und bestimmt dadurch deren Aufbau. Einleitend wird in 4.1 eine Instanzierung des Metamodells erläutert, um ein Anwendungsbeispiel für ein Datenmodell zu skizzieren. Anschließend werden die Elemente des Metamodells in 4.2 genauer beschrieben. Diese unterteilen sich in Entities, Properties, Roles, Mixins, Validators und Changelistener. Das Verständnis der Datenmodelle und des Metamodells ist eine Grundlage für die Datenmodellvisualisierung in Kapitel 5 und das Round Trip Engineering in Kapitel 6.

## 4.1 Ein exemplarisches Datenmodell

Das Metamodell wird benutzt, um den Aufbau der Datenmodelle zu bestimmen. Datenmodelle sind Instanzierungen des Metamodells und richten sich in ihrer Struktur somit genau nach dem dort festgelegten Schema. Einführend sei das Beispiel im UML-Klassendiagramm 4.1 gegeben, welches die Entitäten Wiki und WikiPage und deren Beziehungen untereinander abbildet. Anhand dieses Beispiels wird danach das Metamodell in 4.2 genau erläutert.

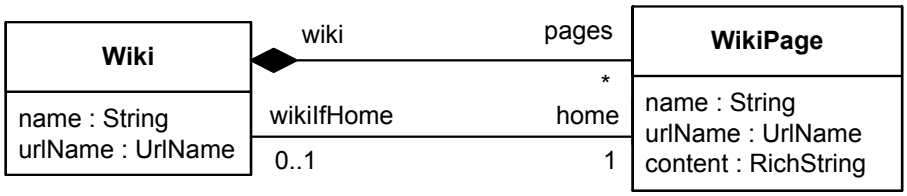



Abbildung 4.1: Instanzierung des Metamodells: Wiki-WikiPage

Ein Wiki verfügt über zwei Eigenschaften, *name* und *urlName*, die wiederum jeweils einen Typ, nämlich *String* und *UrlName*, haben. Es bestehen zwei Relationen mit WikiPage, die durch

Rollen, Mengenangaben und Verbindungsart näher spezifiziert werden. So hat ein Wiki genau eine *home*-WikiPage und beliebig viele *pages*. Die zweite Verbindungsart ist existenzabhängig, was bedeutet, dass die pages eines Wikis ohne das Wiki selber nicht existieren, da sie Teil des Wikis sind. Auch die WikiPage wird durch drei Eigenschaften - *name*, *urlName* und *content* - mit jeweiligen Typen genauer spezifiziert.

Zusätzlich ist in Abbildung 4.2 eine textuelle Sicht auf das Datenmodell gegeben. Diese enthält weitere Informationen, die für die in 4.2 folgenden allgemeineren Erklärungen des Metamodells von Relevanz sind.

```
entity Wiki
 label = (en : "Wiki")
 mandatoryMixins
  Linkable
  Searchable
 features
  name : StringProperty
   maxLength = 255
   isIndexed = false
   isPersistent = true
   label = (en : "Name")
   validate
    MinimalLengthValidator(length = 1)
  urlName : UrlNameProperty
   maxLength = 255
   isIndexed = false
   isPersistent = true
   label = (en : "Name in URL")
  pages : ManyRole (WikiPage)
   oppositeRole = wiki : OneRole
   isCascadeDelete = true
   isPersistent = true
  home : OneRole (WikiPage)
   oppositeRole = wikiIfHome : OneRole
   isCascadeDelete = false
   isOwner = true
   isPersistent = true
   label = (en : "Home Page")
```

```
entity WikiPage
 label = (en : "Wiki Page",de : "Wikiseite")
 mandatoryMixins
  Commentable
  Linkable
  Taggable
  Searchable
 features
  name : StringProperty
   maxLength = 255
   isIndexed = false
   isPersistent = true
   label = (en : "Name")
   validate
    MinimalLengthValidator(length = 1)
   onChange
    updateUrlName (
     WikiPage.name,
     WikiPage.urlName
    )
  urlName : UrlNameProperty
   maxLength = 255
   isIndexed = false
   isPersistent = true
   label = (en : "Name in URL")
  content : RichStringProperty
   maxLength = 16777216
   isIndexed = false
   isPersistent = true
   label = (en : "Content")
  wiki : OneRole (Wiki)
   oppositeRole = pages : ManyRole
   isCascadeDelete = false
   isOwner = false
   isPersistent = true
   label = (en : "Wiki")
   validate
    NotNullOneValidator
  wikiIfHome : OneRole (Wiki)
   oppositeRole = home : OneRole
   isCascadeDelete = false
   isOwner = false
   isPersistent = true
```

Abbildung 4.2: Textuelle Sicht auf das Datenmodell Wiki-WikiPage

## 4.2 Das Metamodell

Alle Elemente, die dieses eben beschriebene Datenmodell ausmachen, werden durch das Metamodell bestimmt. Das UML-Klassendiagramm 4.3 zeigt die Zusammenhänge zwischen den einzelnen Komponenten, welche im Einzelnen in den nächsten Abschnitten erläutert werden. Dabei wird immer ein Bezug zum oben erläuterten Beispiel hergestellt um die Funktionsweise zu verdeutlichen.

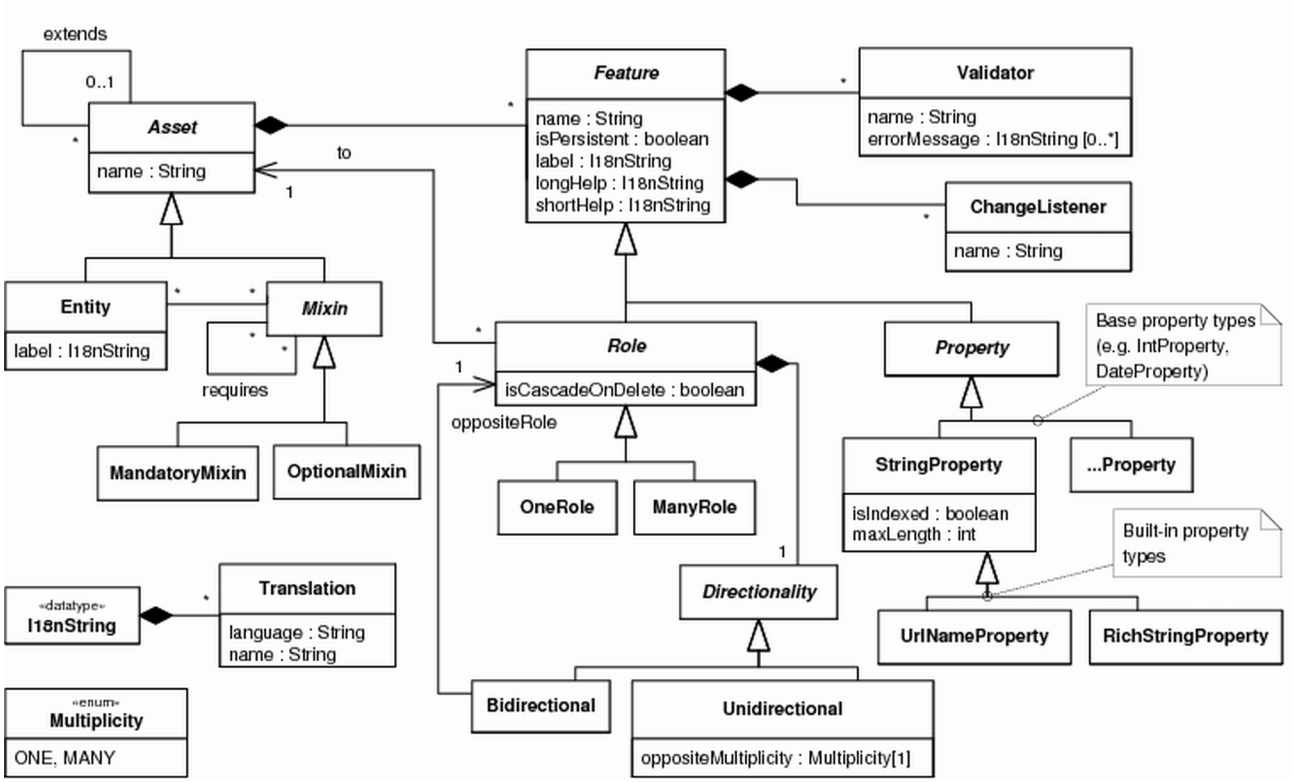



Abbildung 4.3: Das Metamodell von Tricia

### 4.2.1 Entities, Properties und Roles

Entities, Properties und Roles bilden den Kern des Metamodells und werden durch die anderen Komponenten erweitert.

**Entities** In Tricia werden Objekte durch die Klasse Entity repräsentiert, welche über einen Namen, der zur Identifikation dient, und ein internationalisiertes *label*, das bei Endbenutzersichten angezeigt wird, verfügt. Im Beispiel sind Wiki und WikiPage Entities, wobei WikiPage über ein englisches („Wiki Page“) und ein deutsches („Wikiseite“) label beschrieben wird.

**Properties** Eigenschaften beschreiben Assets näher und werden über den Typ Property definiert. Tricia bietet gewisse Basisproperties, wie *BooleanProperty*, *IntProperty*, *StringProperty*,

*DomainValueProperty*, *DateProperty* und *TimestampProperty* an. Darauf aufbauend werden durch den Tricia Kern weitere Typen, die für den Webeinsatz notwendig sind, definiert:

Die *RichStringProperty* ist speziell sinnvoll für HTML, da hier beispielsweise gesichert wird, dass keine gefährlichen Scripts eingebaut werden können. Die *UrlNameProperty* wird dafür benutzt, sinnvolle URLs zu erstellen, die möglichst dem Objektnamen ähneln sollten. Die *PasswordProperty* verschlüsselt Inhalte, so dass diese niemals offen angezeigt werden. Durch die *IdProperty* werden Entitäten eindeutig bestimmt.

Das Wiki im Beispiel enthält zwei dieser Properties, nämlich eine StringProperty name und UrlNameProperty urlName. Ebenso enthält eine WikiPage die drei Properties name, urlName und content.

**Roles** Beziehungen zwischen Assets werden durch die Klasse Role bestimmt, welche wiederum über weitere Eigenschaften verfügt. Einerseits wird die Kardinalität durch die Klassen OneRole und ManyRole definiert. Andererseits ist die Richtung entweder unidirektional oder bidirektional. Im ersten Fall muss die Multiplizität des Gegenparts in *oppositeMultiplicity* angegeben werden. Bei der bidirektionalen Verbindung wird auf die gegenüberliegende Role mit *oppositeRole* verwiesen. Mit *isCascadeOnDelete* wird angegeben, ob es sich um eine existenzabhängige Beziehung handelt. Soll also beim Löschen des einen Assets auch das über diese Komposition verbundene Asset gelöscht werden.

Bei Wiki und WikiPage bestehen zwei bidirektionale Beziehungen, wobei wiki-pages existenzabhängig ist, was durch die gefüllte Raute symbolisiert ist.

**Features** Da Properties und Roles gewisse Eigenschaften teilen, werden sie unter der abstrakten Klasse Feature zusammengefasst. Ein Feature hat einen Namen, ein Label und eine *longHelp* und *shortHelp*, die Hilfetexte in unterschiedlichen Sprachen beinhalten. Des Weiteren bestimmt *isPersistent*, ob das Feature in einer Datenbank persistiert werden soll. Nicht persistente Features können beispielsweise aus Daten zur Laufzeit generiert werden und nur temporäre Gültigkeit oder Notwendigkeit haben. Darüber hinaus verfügt ein Feature über Validatoren und Changelistener, die im nächsten Abschnitt erklärt werden.

### 4.2.2 Validators und Changelistener

**Validators** Um Eingaben von Benutzern zu überprüfen, kann ein Feature mit Validatoren konfiguriert werden. Diese Validatoren sind notwendig, um Bedingungen festzulegen, die die Datenintegrität gewährleisten. Eigenschaften sind einerseits ein Name und andererseits Fehlermeldungen, die dem Endbenutzer bei falschen Eingaben angezeigt werden.

Tricia bietet einige Validatoren, wie den *EmailValidator* oder den *MinimalLengthValidator*, an, mit denen beispielweise StringProperties getestet werden können. Aber auch Roles können mit Validatoren versehen werden, wie im Beispiel die Beziehung von WikiPage zu Wiki: Die

wiki-Komposition darf niemals leer sein, eine WikiPage braucht immer ein Wiki. Deshalb ist diese Role mit einem *NotNullOneValidator* ausgestattet.

**Changelistener** Sollten sich Featurewerte ändern ist es teilweise notwendig, im gleichen Zug weitere Änderungen vorzunehmen. Dies wird über Changelistener realisiert, die über einen Namen verfügen und zu einem gewissen Feature zugeordnet sind.

Das Beispiel beinhaltet in WikiPage den *updateUrlName* Changelistener, der die URL einer neuen Wikiseite dem Standard anpasst, wenn der Name, aber keine spezielle URL, angegeben wird.

### 4.2.3 Entities und Mixins

Die einzige Möglichkeit, die bisher durch das Metamodell bereitgestellt wird, Wiederverwendung zu realisieren ist die der Vererbung. So kann eine Entity von einer anderen Entity erben und somit alle Features übernehmen und erweitern. Durch die von Java vorgegebene Einzelvererbung wird verhindert, dass eine Entity Teile mehrerer anderer Entities übernehmen kann. Genau dies ist allerdings häufig der Fall, weshalb es notwendig ist, den Aspekt der Wiederverwendung durch ein anderes Konzept mächtiger zu gestalten.

Hierfür bietet Tricia sogenannte Mixins an, die bestimmten Entities zugewiesen werden können, wobei deren Anzahl nicht begrenzt ist. Das Mixin *Commentable* beispielsweise ist mit dem Asset *Comment* verbunden, wodurch eine WikiPage die Features der Entity Comment übernehmen kann, aber keine Unterklasse von dieser ist. So lassen sich beliebig viele unterschiedliche Mixins einer Entity zuweisen. Bei solchen fest zugewiesenen Mixins, wird von *MandatoryMixins* gesprochen. Dem gegenüber stehen die *OptionalMixins*, die den Entities während der Laufzeit zugewiesen werden können. Als Beispiel sei hier das Mixin *CalendarItem* angeführt, das anzeigt, dass eine WikiPage ein zeitliches Event darstellt. Bei beiden Mixinarten besteht die Möglichkeit, die einzelnen Mixins untereinander zu verknüpfen. So muss eine Entity mit einem *Searchable*-Mixin ebenfalls *Linkable* sein, damit innerhalb eines Suchergebnisses darauf verwiesen werden kann.

# Kapitel 5

# Datenmodellvisualisierung

Im vorangegangenen Kapitel 4 wurde beschrieben, wie sich die Struktur von Datenmodellen durch das Tricia Metamodell bestimmt. Sobald sich mehrere Assets mit Beziehungen untereinander in einem Datenmodell befinden, wird dieses häufig schnell komplex. Um diese Komplexität zu reduzieren, ist es notwendig, Datenmodelle so darzustellen, dass diese mit intuitivem Verständnis nachvollzogen werden können. Hierfür eignet sich eine dem UML-Klassendiagramm ähnliche Visualisierung. Der Vergleich der textuellen und der graphischen Sicht auf das in 4.1 beschriebene Beispiel zeigt deutlich, wie viel klarer die Informationen bei der Datenmodellvisualisierung aufbereitet sind.

Zu Beginn dieses Kapitel werden in 5.1 für diese Art der Darstellung Nutzungsszenarien besprochen, in denen beschrieben wird, wie unterschiedliche Stakeholder davon profitieren können und weshalb es sinnvoll ist, ein Programm zur automatischen Datenmodellvisualisierung zu entwickeln. Anschließend werden in 5.2 unterschiedliche Visualisierungsmöglichkeiten diskutiert beziehungsweise ausgewählt. Aus den Nutzungsszenarien ergeben sich dann in 5.3 die funktionalen Anforderungen, die das Programm erfüllen muss. Zur Umsetzung dieses Systems werden daraufhin in 5.4 technische Grundsatzentscheidungen getroffen, die die Implementierung stark beeinflussen. Nach dem Abschnitt 5.5, der die genaue Realisierung beschreibt, wird das fertige Programm in 5.6 vorgestellt.

## 5.1 Nutzungsszenarien

An der Nutzung von Visualisierungen von Datenmodellen können viele Stakeholder auf unterschiedliche Weise profitieren. Angefangen beim Softwarearchitekt, der bei seinem Architekturentwurf die Möglichkeit hat, bereits vorhandene Strukturen besser zu analysieren indem er sich den Aufbau graphisch darstellen lassen kann. So kann er beispielsweise bei Erweiterungen existenter Systeme die Datenarchitektur den Gegebenheiten anpassen.

Der Entwickler ist wohl der, der am meisten Vorteile aus einem solchen Programm ziehen kann. Er kann zum Beispiel aus seinem Programmcode ein visualisiertes Datenmodell erstellen und mit den Vorgaben des Architekten vergleichen und somit auf Konsistenz mit dem Entwurf

prüfen. Da sich die Modelle in der ständigen Weiterentwicklung befinden, wird es häufig auch nützlich sein, wenn er eine gewisse Sicht auf mehrere Assets speichern und später wieder abrufen kann. So ist immer ein bestimmter Interessensbereich aktuell visualisierbar. Wichtig hierbei ist auch, dass man je nach Zweck Aspekte ein- oder ausblenden kann. Wenn beispielsweise gerade nur Bedarf an Rolleninformationen besteht, ist die Anzeige von Properties oder Mixins eher störend, da sie vom Fokus ablenken.

Zwischen allen beteiligten Parteien, wie Softwarearchitekt und Entwickler aber auch Projektmanager und Kunde, wird die Kommunikation gefördert, da es leichter ist, Sachverhalte an Hand von Diagrammen, die durch die Datenmodellvisualisierung bereitgestellt werden, zu erklären. So sollten durch eine Exportmöglichkeit in ein Bildformat auch Publikationen oder Präsentationen unterstützt werden können.

## 5.2 Visualisierungsalternativen

Vor allem um Beziehungen zwischen den Assets herausstellen zu können, bietet sich eine graphische Sicht an. Deshalb ist gerade dies ein Aspekt, der in der Wahl der Art der Visualisierung eine entscheidende Rolle spielen sollte. Für die Selektion aus den Visualisierungsalternativen mussten unterschiedliche Notationen für Graphen evaluiert werden, wobei im folgenden auf die *Pin*-Notation und die dem *UML-Klassendiagramm* ähnliche Notation eingegangen wird.

### 5.2.1 UML-Notation

Die Unified Modeling Language [13] wurde von der Object Management Group [11] als ein Standard für unterschiedlichste Modellierungsaufgaben festgelegt. Eines der bekanntesten Modelle ist das Klassendiagramm, welches bereits weite Verbreitung und Zuspruch erfahren hat. Ein Beispiel für ein einfaches Klassendiagramm ist die in 4.1 bereits besprochene Art der Darstellung (Abb. 5.1).

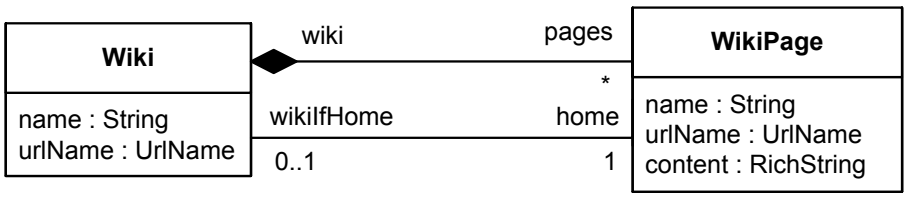



Abbildung 5.1: Wiki-WikiPage UML-Notation

Die Attribute der Klassen sind direkt abzulesen und die Verbindungen verfügen über Rollenbezeichnungen und Angaben zur Multiplizität. Es ist möglich, mit wenig Vorwissen Diagramme dieser Notation intuitiv nachzuvollziehen.

### 5.2.2 Pinnotation

In der Pinnotation werden die Rollen innerhalb der Klassen aufgeführt und auch dort näher beschrieben, wodurch die Klassen in ihrer Höhe je nach Menge der Relationen massiv anwachsen. Die Verbindungen führen von einem Pin, wie zum Beispiel den *pages* in Wiki, zum genauen Gegenstück, *wiki* in WikiPage. Durch die mangelnde Abgrenzung der Informationen der Beziehungen von denen der Properties beispielsweise fällt die Zuordnung schwerer.


Abbildung 5.2: Wiki-WikiPage Pinnotation

### 5.2.3 Fazit Visualisierung

Aufgrund der klareren Darstellung der UML-Notation und ihrer weiten Verbreitung wurde diese für die Datenmodellvisualisierungen gewählt. Dies erleichtert wiederum die Kommunikation zwischen unterschiedlichen Parteien, da es sich um einen Standard handelt, der nicht neu eingeführt werden muss.

## 5.3 Funktionale Anforderungen

Einige funktionale Anforderungen, die das Programm zur Datenmodellvisualisierung erfüllen muss, ergeben sich durch das Metamodell und andere durch die Nutzungsszenarien. Das Metamodell stellt Entities, Properties, Mixins und Roles in den Mittelpunkt, weshalb auch in der Visualisierung der Fokus darauf gelegt wird. Die Notwendigkeiten, die sich daraus ergeben werden in den kommenden Abschnitten 5.3.1 und 5.3.2 erläutert. Die Abschnitte 5.3.3 und 5.3.4 gehen anschließend auf die Funktionen ein, die durch die Nutzungsszenarien bestimmt werden.

### 5.3.1 Entities

Die Entities werden neben den Relationen durch Properties und Mixins näher beschrieben. Aufgrund der Wahl für die UML-Notation (siehe 5.2) sollten diese wie im Klassendiagramm innerhalb einer rechteckigen Form stichpunktartig aufgeführt werden. Es muss des Weiteren möglich sein, die Entities über Linien zu verbinden um die Relationen darstellen zu können.

### 5.3.2 Relationen

Die Relationen, die durch beschriftete Verbindungen realisiert werden, müssen flexibel gestaltbar sein, damit die unterschiedlichen Arten der Beziehungen symbolisiert werden können. Diese verschiedenen Darstellungsarten stellen vor allem in technischer Hinsicht eine Herausforderung dar. Die möglichen Verbindungstypen werden in den folgenden Paragraphen an Hand von Beispielen beschrieben.

**Bidirektionale** Assoziationen, wie zwischen Wiki und WikiPage in Abbildung 5.3, stellen die klassischste Art dar. Ein Wiki hat immer eine Homepage, aber eine WikiPage muss keine Homepage eines Wikis sein. Auf beiden Seiten befinden sich Rollenbezeichnungen mit zugehörigen Multiplizitäten.

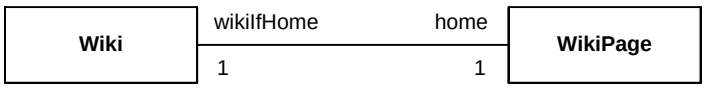



Abbildung 5.3: Bidirektionale Relation zwischen Wiki und WikiPage

**Unidirektionale** wie sie auch zwischen Person und Group im Diagramm 5.4 zu finden ist, stehen dem gegenüber. Hierbei ist die Kommunikation einseitig und eine Person wird durch Gruppen benachrichtigt, was durch den Pfeil symbolisiert wird.

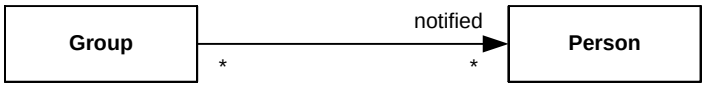



Abbildung 5.4: Unidirektionale Relation zwischen Group und Person

**Cascade-On-Delete** sind spezielle Verbindungen, beispielhaft vertreten durch die Group-Membership Relation. Die Existenz der Mitgliedschaft ist demnach an die Existenz der Gruppe gebunden. Ohne entsprechende Gruppe macht es natürlich auch keinen Sinn, eine Zugehörigkeit zu dieser zu führen. Das Zeichen dafür ist eine gefüllte Raute auf Seiten der „beinhaltenden“ Entität.

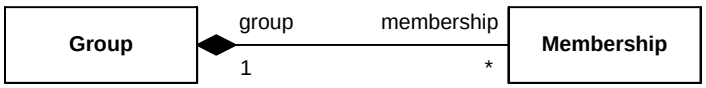



Abbildung 5.5: Cascade-On-Delete Relation zwischen Group und Membership

**Vererbung** beschreibt eine weitere besondere Art der Assoziation. Im Diagramm 5.6 durch das ungefüllte Dreieck und die Verbindungen zwischen Principal-Person und Principal-Group präsent, beschreibt sie die Tatsache, dass Person und Group Unterklassen von Principal sind.

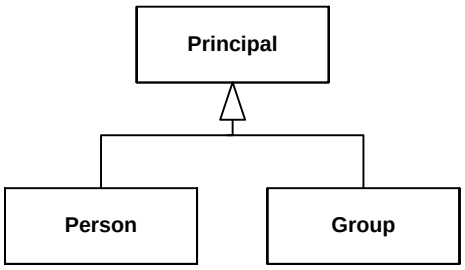



Abbildung 5.6: Vererbung: Person und Group von Principal

### 5.3.3 Detaillierungsgrad

Es muss möglich sein, die Darstellung an individuelle Bedürfnisse anzupassen, indem man unterschiedliche Teile differenziert ausblenden kann. Je nach Bedarf sollte Fokus auf bestimmte Faktoren gelegt und die für bestimmte Zwecke unnötigen Informationen vernachlässigt werden können. So muss es machbar sein, zum Beispiel alle Mixins und Properties nicht mehr anzuzeigen, wodurch die Zusammenhänge zwischen den Entities, also die Roles, in den Mittelpunkt rücken.

### 5.3.4 Exportfunktionen

**Bild** Um die generierten Diagramme auch außerhalb des Programms verwenden zu können, sollten diese als Bild (jpg) abgespeichert werden können und somit für Weiterverwendungen, wie der Publikation in Texten oder für Präsentationen, zur Verfügung stehen.

**Graph** Da Layoutalgorithmen auch heute noch nicht zur Perfektion ausgereift sind und auch häufig spezielle Anordnungen der Entitäten in den Visualisierungen gewünscht sind, wird die Option benötigt, diese per Hand anzuordnen. Die Auswahl der benötigten Entitäten und deren Positionen müssten allerdings bei erneuter Abfrage wiederholt angegeben werden. Um diesen Vorgang zu automatisieren müssen die Informationen abgespeichert und zu einem beliebigen Zeitpunkt wiederhergestellt werden können. Hierzu müssen die Entitäten neu geladen werden, damit sie trotzdem aktuell und konsistent mit dem zugrunde liegenden Code sind.

## 5.4 Technische Realisierungsmöglichkeiten

Für die technische Realisierung der Datenmodellvisualisierung bieten sich viele unterschiedliche Möglichkeiten an, aus denen die für den späteren Anwendungsbereich passendsten herausgefiltert werden müssen. Ein Beispiel hierfür die ist Wahl zwischen einer webbasierten Applikation und einem Desktopsystem. Beide Seiten dieser grundlegenden Entscheidungen weisen Vor- und Nachteile auf, die, auch im Hinblick auf die Implementierung, abgewägt werden müssen. In diesem Abschnitt werden drei dieser Entscheidungen vorgestellt und begründet.

### 5.4.1 IMF - Introspective Modeling Framework

Das IMF ist eine Sammlung von Eclipse-Plugins, die unter anderem die Introspektion des Tricia-codes und die komplette Integration in die Eclipse IDE [25] ermöglicht. Dementsprechend ist die Verwendung des IMF notwendig, um ein introspektives Modell zu gewinnen. Im *Assets*-Plugin können dann direkt Sichten auf das Modell erstellt werden. Es existiert bereits eine Baumsicht, die in Abbildung 5.7 zu sehen ist. Hierbei handelt es sich aber eher um eine technische Sicht, in der vor allem die Roles nicht ausreichend herausgestellt werden. Weitere Informationen zur Funktionsweise des Introspective Modeling Frameworks werden in [5] beschrieben.

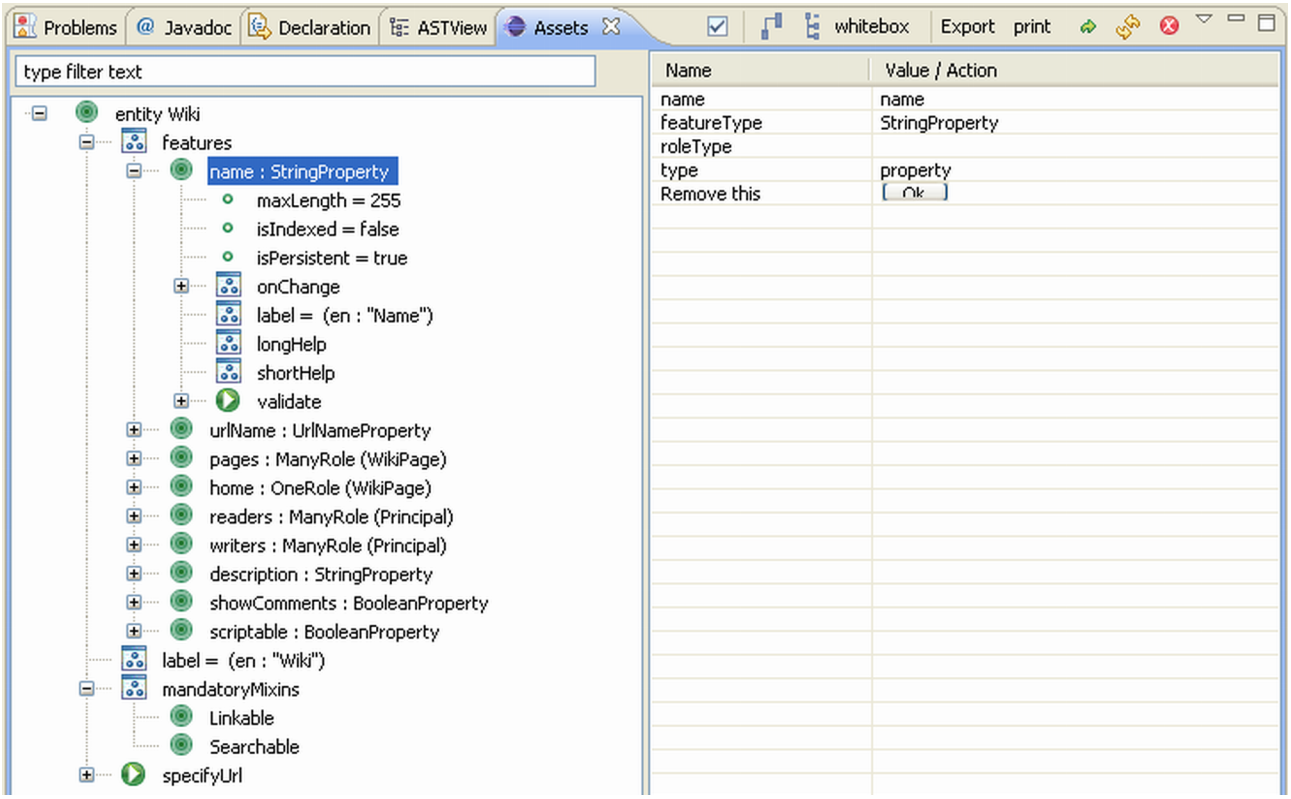



Abbildung 5.7: Modell-Baumansicht Wiki

### 5.4.2 Web vs. Desktop

Gerade in den heutigen Zeiten orientiert sich die Softwareentwicklung, aufgrund der neuen Möglichkeiten die sich dadurch ergeben, immer mehr in Richtung *Webanwendungen* weg von den klassischen *Desktopanwendungen*.

**Webanwendungen** bieten durch die immer höher verfügbare und bessere IT-Infrastruktur fast überall auf der Welt eine sehr hohe Erreichbarkeit. So kann man beispielsweise von jedem Rechner mit Internetanbindung auf die Applikation zugreifen und diese nutzen, zumeist sogar ohne zusätzliche Software, da die Programme in einem Browser ausgeführt werden können. Diese Art der Anwendung bietet eine teilweise notwendige Flexibilität, auf die in bestimmten Bereichen aufgrund der, mit der Globalisierung einhergehenden, Effekte kaum verzichtet werden kann. Immer mehr Unternehmen richten sich heute schon und in Zukunft weiter in diese Richtung aus, um die positiven Faktoren nutzen zu können.

**Desktopanwendungen** stehen dem gegenüber mit der zu Beginn meist aufwendigeren Installation und Einrichtung von Software und den damit verbundenen Einschränkung auf diese Systeme. Da sich der Anwenderkreis hier allerdings weitestgehend auf die Entwickler, die ohnehin größere Anwendungen auf ihrem Computer benötigen, beschränkt, fällt dieser Faktor weniger ins Gewicht als bei anderen Applikationen.

Ein weiteres Argument für die Entscheidung für eine Desktopanwendung ist die Implementierung. Rich-Client-Programme bieten vor allem in Bezug auf die Visualisierung wesentliche Vorteile. Die Draw2D Realisierungsmöglichkeiten innerhalb eines Browsers sind im Verhältnis zu den unbeschränkten eines Desktopsystems sehr eingrenzend. Als technische Kernfrage muss dieser Faktor also besonders beachtet werden. Darüber hinaus wird die Integration in die Eclipse IDE [25] ermöglicht, die bereits das IMF, welches näher in 5.4.1 beschrieben wurde, bietet und für die Tricia Entwicklung empfohlen wird. Diese enge Verbindung ermöglicht eine hohe Modell-Code Integration. Dadurch wird es möglich andere Anwendungen, wie das Round Trip Engineering in Kapitel 6, zu realisieren.

**Eclipse RAP** strebt eine Zwischenlösung an, indem es eine Möglichkeit zur Verfügung stellt, Rich-Client Anwendungen, die für das Eclipse OSGI-Framework erstellt wurden, im Browser auszuführen. Allerdings wird derzeit noch keinerlei Draw2D Unterstützung angeboten, was bei der Datenmodellvisualisierung ein KO-Kriterium ist. (Eclipse Rich Ajax Plattform [28])

**Fazit** Da der wesentliche Vorteil einer Webapplikation, die hohe Verfügbarkeit, bei diesem Anwenderkreis keine all zu große Rolle spielt, kann das Desktopsystem bestechen. Die besseren Alternativen der Implementierung der Visualisierung und die gute Integrationsmöglichkeit in die Eclipse-Plattform machen es sinnvoller sich für eine Desktopanwendung (Eclipse-Plugin) zu entscheiden und somit eine umfassendere und flexiblere Applikation schaffen zu können.

### 5.4.3 GEF vs. NVL

Da die Visualisierung als Kern dieser Anwendung angesehen werden kann, ist die Frage nach der Bibliothek, mit der die graphische Sicht erstellt wird, zentral. Beachtet werden müssen hierbei besonders die Möglichkeiten, auf die Anforderungen eingehen zu können und die Verwendungsmethodik der Schnittstellen.

**Graphical Editing Framework** Motiviert durch die Tatsache, dass es sich bei dem Programm um ein Eclipse-Plugin handelt, bietet sich das Eclipse Graphical Editing Framework [26] durch die relativ nahtlose Integrationsmöglichkeit in Eclipse an. Diese Fähigkeit ist insbesondere durch die Realisierung in SWT (Standard Widget Toolkit [30]), dem Eclipse-Äquivalent von AWT (Abstract Window Toolkit [17]) beziehungsweise Swing [19], zu erklären. Da es sich um ein Editing-Framework handelt, wird hier besonders auf die Möglichkeit, graphische Elemente zu bearbeiten, Wert gelegt. Um diese relativ aufwendige Aufgabe erfüllen zu können, arbeitet GEF mit dem Model-View-Controller Prinzip [6] und wird dadurch verhältnismäßig komplex in der Verwendung. Die Datenmodellvisualisierung benötigt diese Funktion allerdings nicht. Die Darstellung von Elementen reicht aus, weshalb GEF für dieses System als überladen angesehen werden kann.

**Netbeans Visual Library** Eine andere Visualisierungsbibliothek stellt die ebenfalls freie und auf Java-Entwickler fokussierte IDE Netbeans [14] zur Verfügung. Auf Basis der *Netbeans Visual Library* [16] wurden unter anderem bereits die Netbeans UML Features [15] implementiert, welche in ihrer Darstellungsform und Anwendung der hier gewünschten Visualisierung relativ nahe kommen. Mit wenigen Aufrufen und Modifizierungen besteht zügig die Möglichkeit, UML ähnliche Visualisierungen zu realisieren. Die Distanz zwischen dem von der NVL benutzten Swing und SWT lässt sich durch die SWT/AWT Bridge [29] kompensieren und schließt somit dieses Problem der Inkompatibilität als KO-Kriterium aus.

**Fazit** Die angenehme und passende Verwendungsweise entschied die Fragestellung der Visualisierungsbibliothek zu Gunsten der Netbeans Visual Library.

## 5.5 Implementierung

In diesem Abschnitt wird beschrieben, wie die Datenmodellvisualisierung implementiert ist. Wie bereits in 5.4.2 erläutert, wird als Basis Eclipse verwendet und ein Plugin dafür realisiert. Genauer gesagt, wird das Introspective Modeling Framework (5.4.1) erweitert, wodurch dessen Funktionen und Schnittstellen benutzt werden können. Im Diagramm 5.8 kann diese Anpassung nachvollzogen werden.

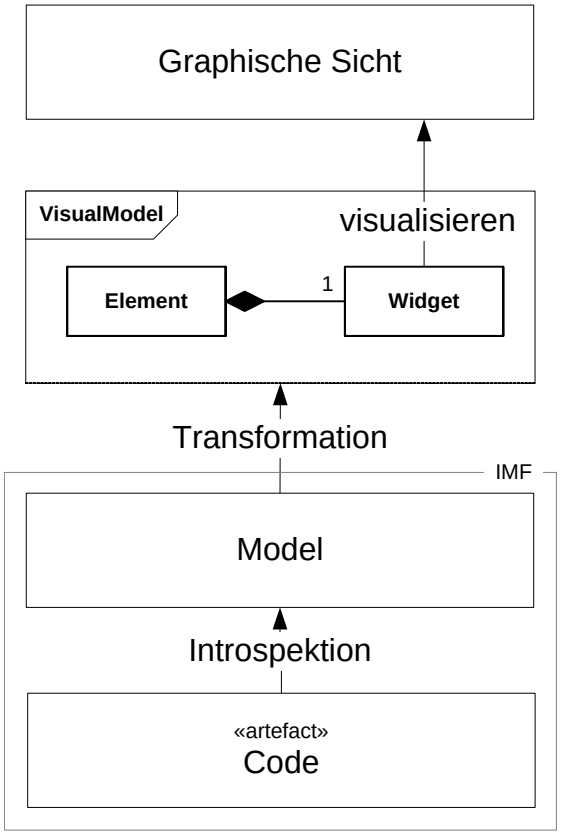



Abbildung 5.8: Implementierung der Datenmodellvisualisierung

Durch Introspektion stellt das IMF ein völlig generisches Modell der Codeartefakte zur Verfügung. Um den Fokus auf die für die Visualisierung wichtigen Informationen des Modells zu legen, wurde das VisualModel eingeführt. Es stellt eine Brücke zwischen dem generischen Modell und der Visualisierung dar. Das introspektive Modell wird in das VisualModel transformiert, indem Elemente gebildet werden, die Teile des Metamodells (siehe 4.2), wie beispielsweise Properties, repräsentieren. Jedes Element verfügt über ein graphisches Widget der Netbeans Visual Library, durch deren Visualisierung eine graphische Sicht gebildet wird.

Die Funktionsweise und einzelnen Elemente des VisualModel werden in 5.5.1 weiter beschrieben. Anschließend wird in 5.5.2 die Realisierung der in 5.3.4 gestellten Forderungen nach Exportfunktionen besprochen.

### 5.5.1 VisualModel

Diese Passage beschreibt das VisualModel, durch das eine logische Trennung zwischen dem introspektiven Modell und der Visualisierung geschaffen wird. Zuerst wird die Architektur und dann der Vorgang der Visualisierung besprochen.

**Die Architektur** Das VisualModel verfügt für jedes graphisch darzustellendes Element des Metamodells über ein Äquivalent. Zur visuellen Repräsentation besitzen diese VisualModel-Elemente ein Widget der Netbeans Visual Library. Das VisualModel an sich beinhaltet Entities und Relations. Eine Entity enthält weitere Mixins oder Properties, die von der abstrakten Klasse Pin erben, da sie in ihrer Darstellung ähnlich sind. Eine Relation wird durch eine SourceRole und eine TargetRole näher spezifiziert. Darüber hinaus verfügt die Relation über eine Entity als source und eine Entity als target.

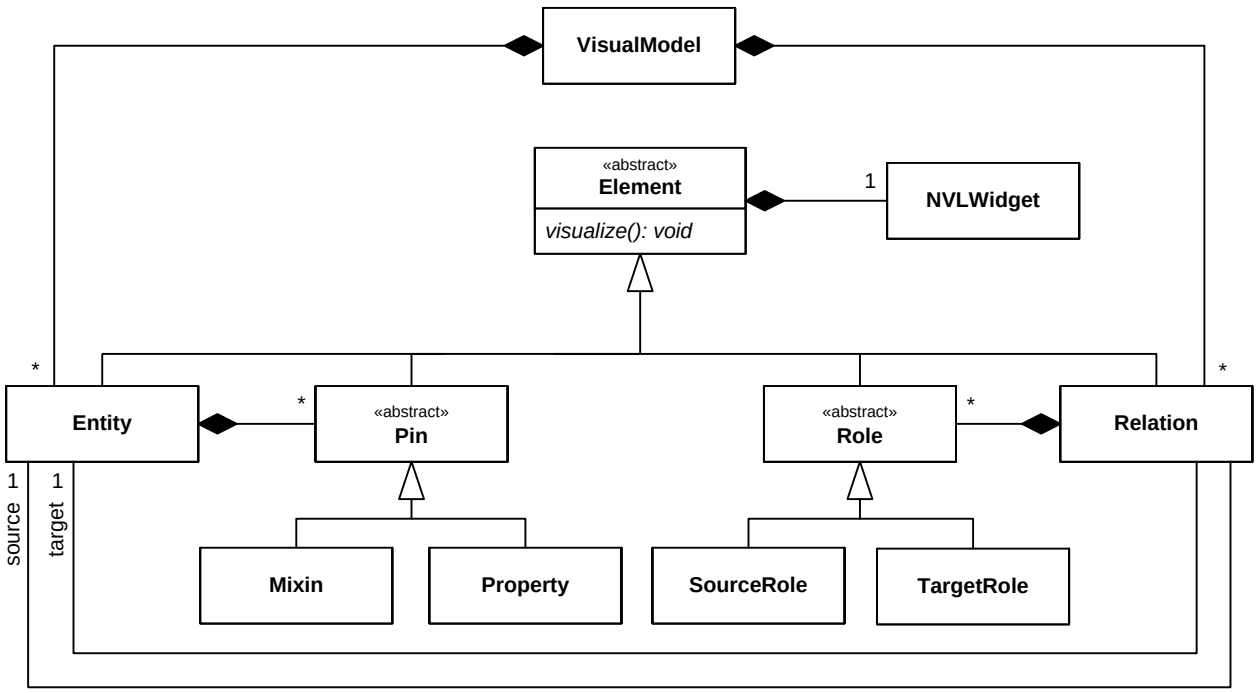



Abbildung 5.9: Assetsview: Programmiermodell

**Visualisierung** Um die Aufgabe der Visualisierung zu vereinfachen, wird diese an die einzelnen Elemente des VisualModels verteilt. Hierzu verfügt jedes Element über sein eigenes Widget und muss die abstrakte Methode `visualize()` implementieren, innerhalb der die graphische Gestaltung bestimmt wird. Dadurch wird die große Aufgabe der Visualisierung auf kleine Verantwortlichkeiten heruntergebrochen und die Verteilung der Zuständigkeiten reduziert die Komplexität der Implementierung der Diagrammdarstellung.

### 5.5.2 UMLView

Die Klasse UMLView aus Abbildung 5.10 steuert die Datenmodellvisualisierung. Durch sie wird das VisualModel gefüllt und die Visualisierung angestoßen. Für die in 5.3.4 geforderten Exportfunktionen werden die drei Methoden `saveAsJPG()`, `saveGraphModel()` und `loadGraphModel()` zur Verfügung gestellt.

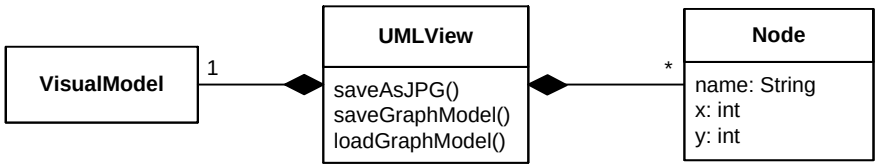



Abbildung 5.10: UMLView

Mit `saveAsJPG()` kann das visualisierte Diagramm als Bild abgespeichert und somit auch außerhalb des Programms verwendet werden.

Die Methode `saveGraphModel()` liest enthaltene Entities und deren Positionen aus dem VisualModel. Anschließend werden diese Informationen in ein `Node` je Entity transformiert. Diese Nodes werden dann mit Hilfe von XStream [35] serialisiert und in eine XML-Datei [34] geschrieben. Nachfolgend ist beispielhaft ein kleiner Ausschnitt aus solch einer Datei gegeben.

```
<Node>
        <name>de.infoasset.wiki.assets.Wiki</name>
        <x>100</x>
        <y>100</y>
</Node>
<Node>
        <name>de.infoasset.wiki.assets.WikiPage</name>
        <x>500</x>
        <y>100</y>
</Node>
```

Durch `loadGraphModel()` kann diese XML-Datei zu jedem Zeitpunkt eingelesen werden, wobei ein introspektives Modell, nur an Hand der abgespeicherten Namen der Entities, neu geladen wird. Dadurch wird immer Aktualität und Konsistenz mit der Datenbasis garantiert. Anschließend wird das Modell wieder in das VisualModel transformiert und visualisiert und die Positionen der Entities werden angepasst.

## 5.6 Ergebnis

Dieser Abschnitt zeigt und erklärt das fertige Programm an einem Beispiel. Als Input für dieses introspektive Modell wurden die vier Entitäten *Group*, *Person*, *Principal* und *Membership* gewählt. Die Abbildung 5.11 zeigt die automatisch generierte Datenmodellvisualisierung.

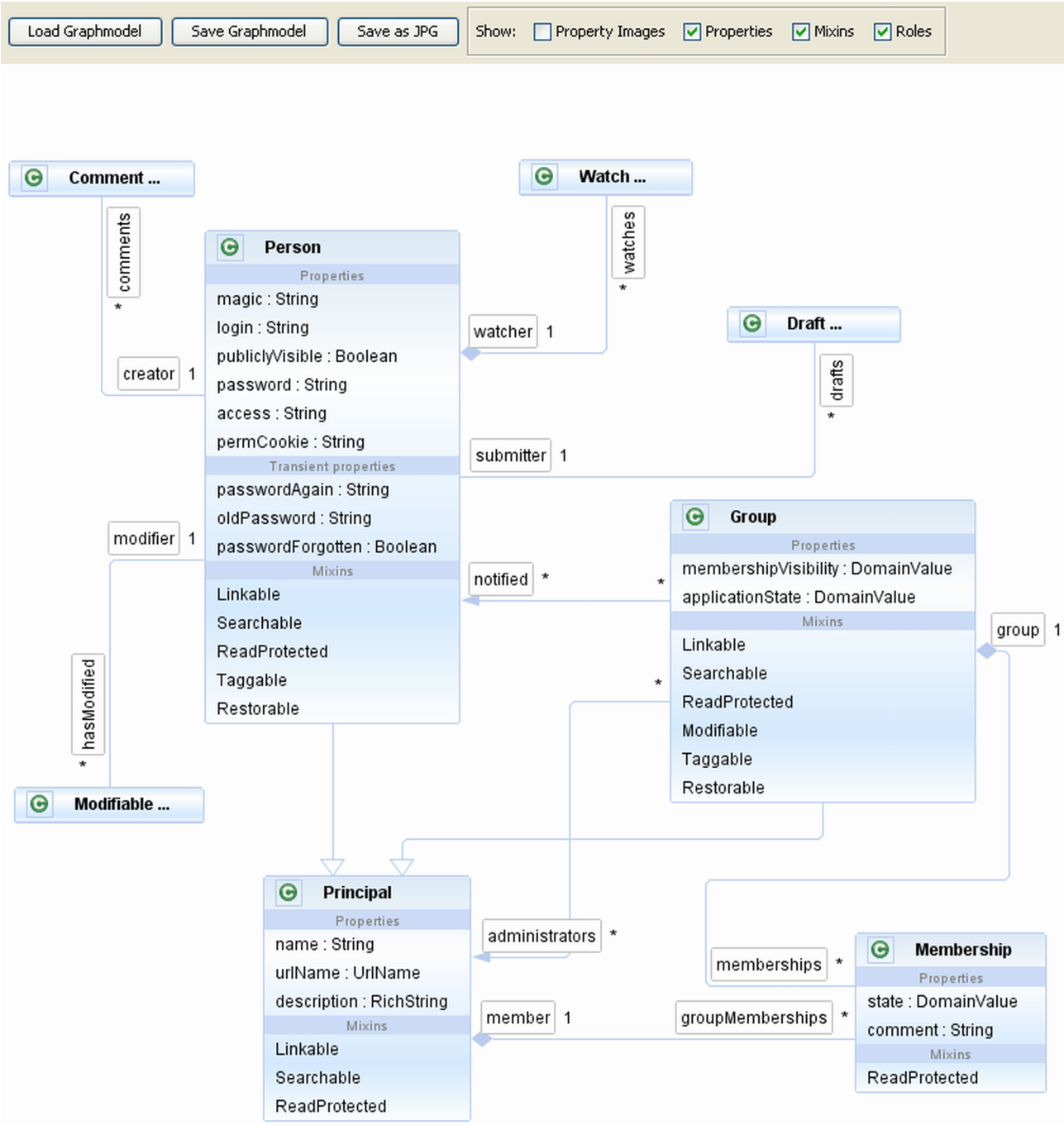



Abbildung 5.11: Ergebnis der Datenmodellvisualisierung

Zu erkennen ist die dem UML-Klassendiagramm ähnliche Darstellung der Entitäten mit den Rollen untereinander. Innerhalb der Entitäten werden die Properties und Mixins abgebildet. Die Entitäten, die mit drei abschließenden Punkten gekennzeichnet sind, ergeben sich automatisch durch Rollen zu Entitäten, die nicht im introspektiven Modell enthalten sind.

Die Erfüllung der Anforderung von 5.3.2, dass alle Verbindungsarten dargestellt werden können, kann im Diagramm an Hand der Relationen nachvollzogen werden. Die Leiste im oberen Bereich bietet die Bedienung für den anpassbaren Detaillierungsgrad ebenso wie die Exportfunktionen.

Das Plugin ist in Eclipse als View aufrufbar, innerhalb derer dann bestimmte Klassen, die Entitäten beinhalten, dem introspektiven Modell hinzugefügt werden können. Hier besteht auch die Möglichkeit, zwischen unterschiedlichen Sichten zu wechseln. So kann von der graphischen Darstellung auch zu der bereits in 5.4.1 beschriebenen Baumsicht geschaltet werden. Dadurch hat man je nach Interessensfokus die passende Sicht zur Verfügung.

# Kapitel 6

# Round Trip Engineering

Das bereits in Kapitel 2.3 beschriebene Round Trip Engineering bietet die Möglichkeit, den Weg der Introspektion zurück zu gehen. Das heißt, dass Codeartefakte automatisch aus einer Sicht heraus generiert werden können. Das Anlegen und Bearbeiten von Entitäten und deren Properties und Roles ist aufgrund der notwendigen Kompatibilität zum Metamodell (siehe Kapitel 4) häufig relativ aufwendig. Die Codeerzeugung stellt eine Abstraktion vom Code dar und reduziert somit den Aufwand des manuellen Programmierens.

Dieses Kapitel beschreibt deshalb, wie das Round Trip Engineering mit Hilfe von Dialogen, in denen Entitäten bearbeitet werden können, realisiert wurde. Zunächst werden in 6.1 die Problemstellungen, die beim Round Trip Engineering auftreten, erläutert. In 6.2 werden dann Entwürfe für die Dialoge an Hand von Mockups vorgestellt. Anschließend wird in 6.3 die Implementierung, auch mit Blick auf die in 6.1 dargelegten Herausforderungen, erläutert. 6.4 gibt abschließend eine Ergebnisübersicht und beschreibt, wie die Realisierung genutzt werden kann.

## 6.1 Problemstellungen

Das Round Trip Engineering bringt einige Problemstellungen mit sich, von denen folgend drei besprochen werden. Die Lösungen dieser Herausforderungen werden dann im Implementierungsabschnitt 6.3 vorgestellt.

#### Metamodell-Kompatibilität

Metamodell-Kompatibilität ist notwendig, um weiterhin Introspektion zu ermöglichen. So müssen bestimmte Codefragmente dem vorgegebenen Schema entsprechen. Als Beispiel sei hier eine Eigenschaft einer Property, die maximale Länge eines Textes, gegeben:

```
public int getMaxLength(){
        return 10;
}
```

Diese Eigenschaft wird dementsprechend nicht über normale Attribute, sondern über sogenannte Value-Methoden festgelegt. Nur wenn diese Kompatibilität ausnahmslos gewährleistet wird, können später erneut introspektive Modelle erstellt werden. Deshalb ist die Einbeziehung des Metamodells zwingend erforderlich.

### Synchronität

Beim Round Trip Engineering ergeben sich drei parallele Zustände: Der Code, die Sicht und das Modell, dass durch die Dialoge bearbeitet wird. Nach Durchführung der Änderungen am Modell und der Codegenerierung müssen alle drei synchronisiert und konsistent sein.

### Codegenerierung

Sichten auf das introspektive Modell sind grundsätzlich Abstraktionen vom Code. Nur dadurch kann eine Reduzierung der Komplexität erreicht werden. Aufgrund dieser Abstraktion ist nicht der gesamte Code introspektiv. Nachfolgend sieht man als Beispiel den ChangeListener *updateUrlName* der Entity WikiPage, der bereits in 4.2.2 beschrieben wurde.

```
public final StringProperty name = new StringProperty(){
    final ChangeListener updateUrlName = new InstantChangeListener(){
        @Override
        public void change(Diff diff){

            UrlNameProperty.updateUrlName(urlNameQuery(), urlName, get());

        }
    };
}
```

Der rot markierte Inhalt der Methode *change* ist nicht im introspektiven Modell enthalten. Aufgrund dieser fehlenden Informationen könnte beispielsweise bei einer Änderung des Namens der Property *name* nicht der gesamte Code neu generiert werden, sondern nur der, der auch im introspektiven Modell ist. So muss exakt nur der Name geändert werden und die anderen Codefragmente werden nicht berührt. Es sind also feingranulare Änderungen notwendig.

## 6.2 Entwurf

Da das Anlegen und Bearbeiten der Entitäten, im Verhältnis zum manuellen Programmieren, vereinfacht werden soll, stellt die Usability des Programms für das Round Trip Engineering einen wichtigen Aspekt dar. Deshalb wurden im Vorhinein Mockups mit Hilfe von Balsamiq Mockup [4] entworfen, die sich optisch an die Eclipse-Umgebung anlehnen.

### Entity Mockup

Der Mockup 6.1 zeigt einen Dialog mit der Entity WikiPage und gibt eine Übersicht über die Properties, Roles und Mixins. Es soll von hier aus möglich sein, neue Inhalte hinzuzufügen, zu entfernen beziehungsweise zu editieren und den Namen zu ändern.

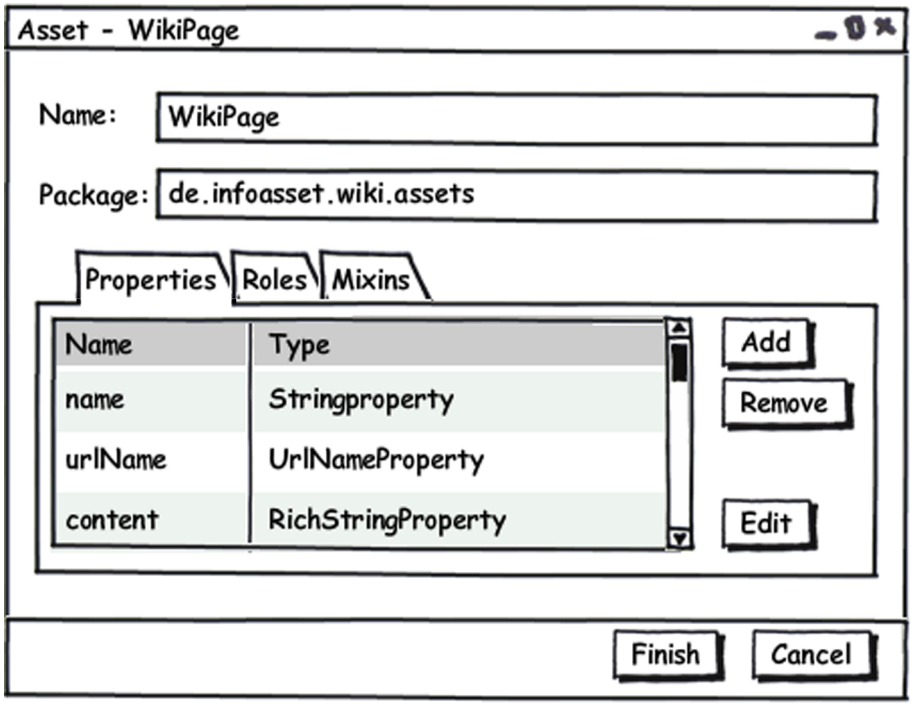



Abbildung 6.1: Round Trip Engineering: Entity Mockup

### Property & Role Mockups

Über den Button „Edit“ bei den Properties und Roles öffnet sich je ein weiterer Dialog 6.2 und 6.3, die eine detailliertere Übersicht über die Featureinformationen liefern. Hier ist es auch möglich bestimmte Eigenschaften, wie die maximale Länge über das direkte Ändern des *maxLength*-Wertes innerhalb der Tabelle, anzupassen. Genauso können über die beiden anderen Reiter ChangeListener und Validatoren angelegt oder entfernt werden.

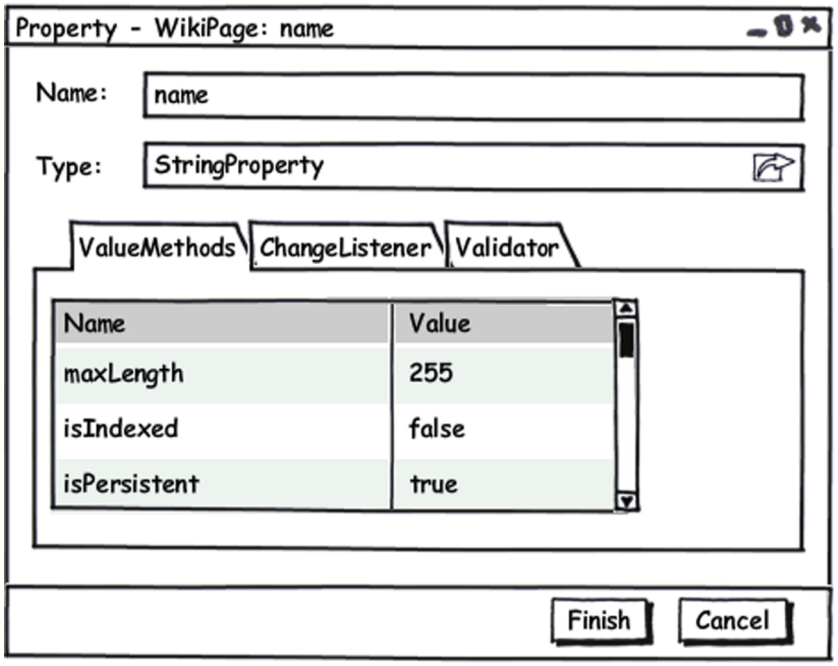



Abbildung 6.2: Round Trip Engineering: Property Mockup

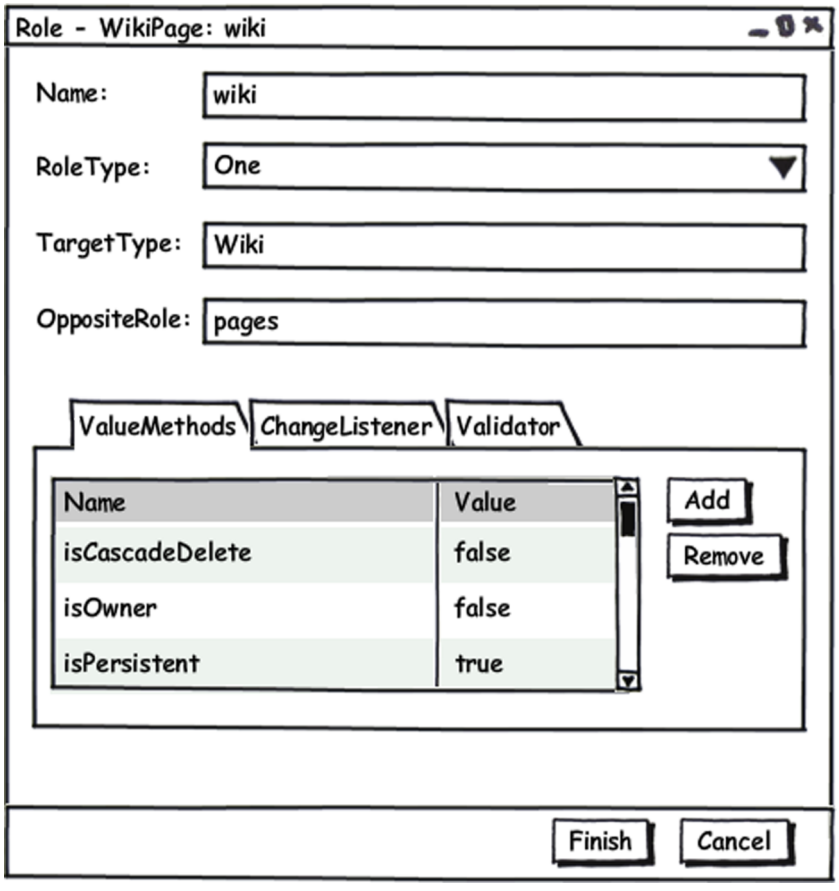



Abbildung 6.3: Round Trip Engineering: Role Mockup

## 6.3 Implementierung

Die Realisierung des Round Trip Engineerings, über die Abbildung 6.4 eine Übersicht zeichnet, baut auf dem introspektiven Modell auf, das aus Codeartefakten gewonnen wird. Dieses generische Modell wird in ein RefactoringModel transformiert. Das RefactoringModel, das in 6.3.2 beschrieben wird, legt Fokus auf die für das Refactoring, also die Umgestaltung des Codes, und die Codegenerierung wichtigen Aspekte. Anschließend ist es möglich, Refactoringdialoge, für die in 6.2 Mockups erstellt wurden, bereitzustellen, innerhalb derer Entitäten bearbeitet werden können. Daraufhin werden die Änderungen, wie in 6.3.3 beschrieben, gesammelt und mit Hilfe des Abstract Syntax Tree (AST) in den Code generiert. Für den Aufbau des RefactoringModels ist das Verständnis des AST essentiell, weshalb dieser in 6.3.1 zu Beginn erläutert wird.

Um das in 6.1 angesprochene Problem der Synchronität zu lösen, wird dann das introspektive Modell neu geladen und Sichten darauf aktualisiert. Somit herrscht übergreifende Konsistenz zwischen Code, Sicht und Modell.

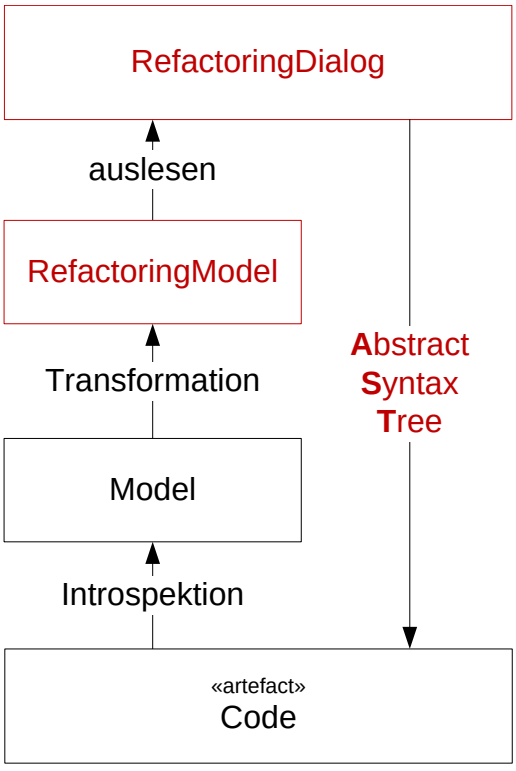



Abbildung 6.4: Round Trip Engineering Implementierung: Übersicht

### 6.3.1 Abstract Syntax Tree

Der Abstract Syntax Tree, der durch das Eclipse JDT Core [27] bereitgestellt wird, stellt einen feingranularen Baum zur Verfügung, der eine Java-Klasse respräsentiert. Die Unterteilung des

Baums richtet sich nach der Syntax der Sprache Java, wodurch alle in einer Klasse enthaltenen Elemente abgebildet werden.

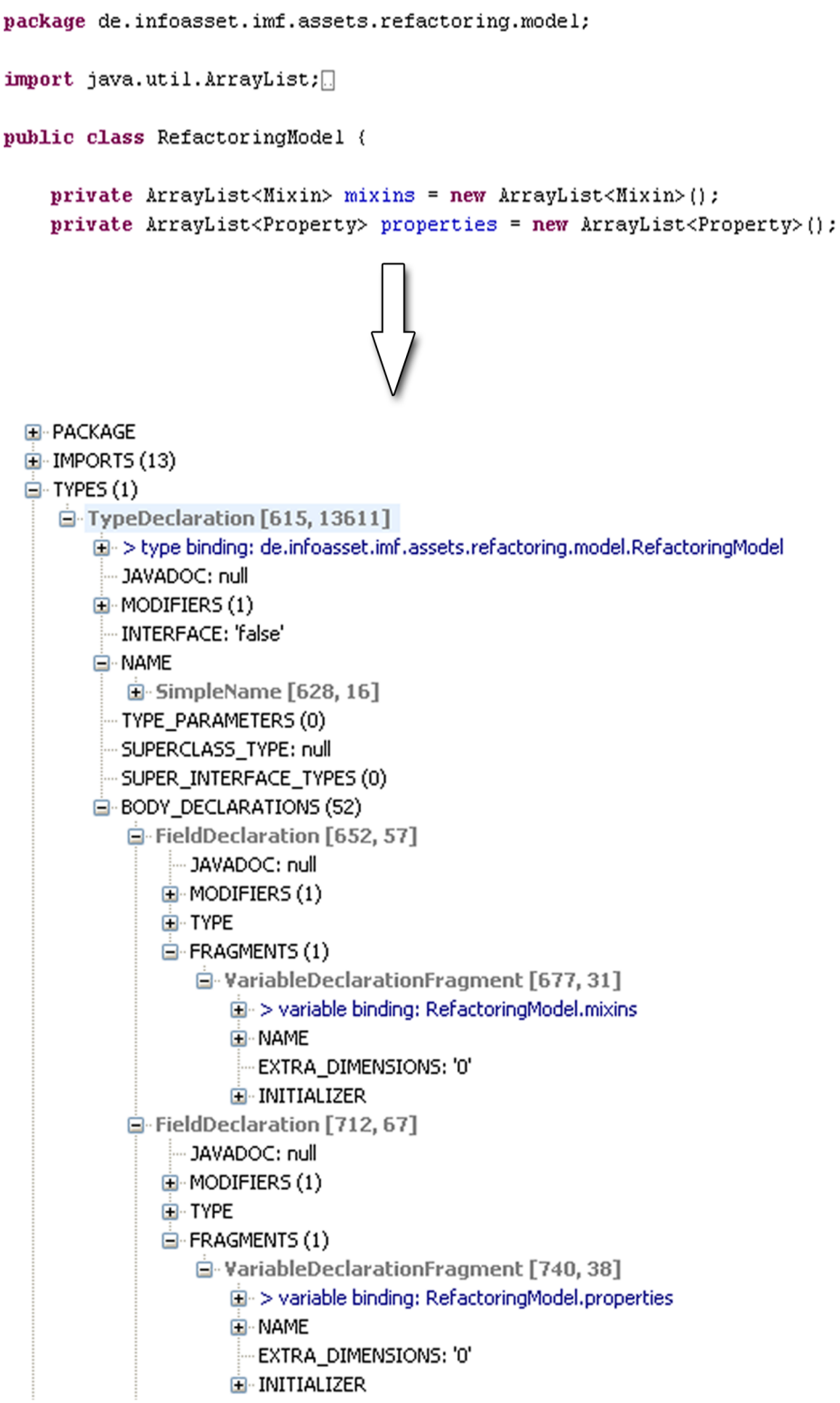



Abbildung 6.5: ASTView in Eclipse

Abbildung 6.5 zeigt ein Codefragment und den zugehörigen AST-Ausschnitt in der ASTView von Eclipse. Dort kann nachvollzogen werden, wie jedes syntaktische Element im AST repräsentiert wird und wie viele Informationen dadurch entstehen. Über diese exakte Struktur ist es möglich, über Schlüsselwörter wie `TypeDeclaration` oder `FieldDeclaration`, auf einzelne Elemente zuzugreifen.

Die AST API bietet auch die Möglichkeit, den Inhalt einzelner Elemente zu verändern oder komplett neue Teilbäume in den AST einzufügen. Dadurch wird die Codegenerierung realisiert und das Problem der feingranularen Änderungen, welches in 6.1 behandelt wurde, gelöst. Um der ebenfalls in 6.1 angesprochene Herausforderung der Metamodell-Kompatibilität gerecht zu werden, ist es essentiell, dass bei der Generierung des Codes das Metamodell mit einbezogen wird.

### 6.3.2 RefactoringModel

Das RefactoringModel, welches als Brücke zwischen dem generischen introspektiven Modell und den Refactoringdialogen fungiert, ist in dem UML-Klassendiagramm 6.6 dargestellt. Jedes Element des Modells zeichnet sich vor allem durch den ASTNode aus, über den es verfügt. Der ASTNode repräsentiert den, das jeweilige Element betreffende, Teilbaum des gesamten AST. Durch diese Zuordnung ist immer vom Element aus ein direkter Zugriff auf den zugrunde liegenden Code möglich.

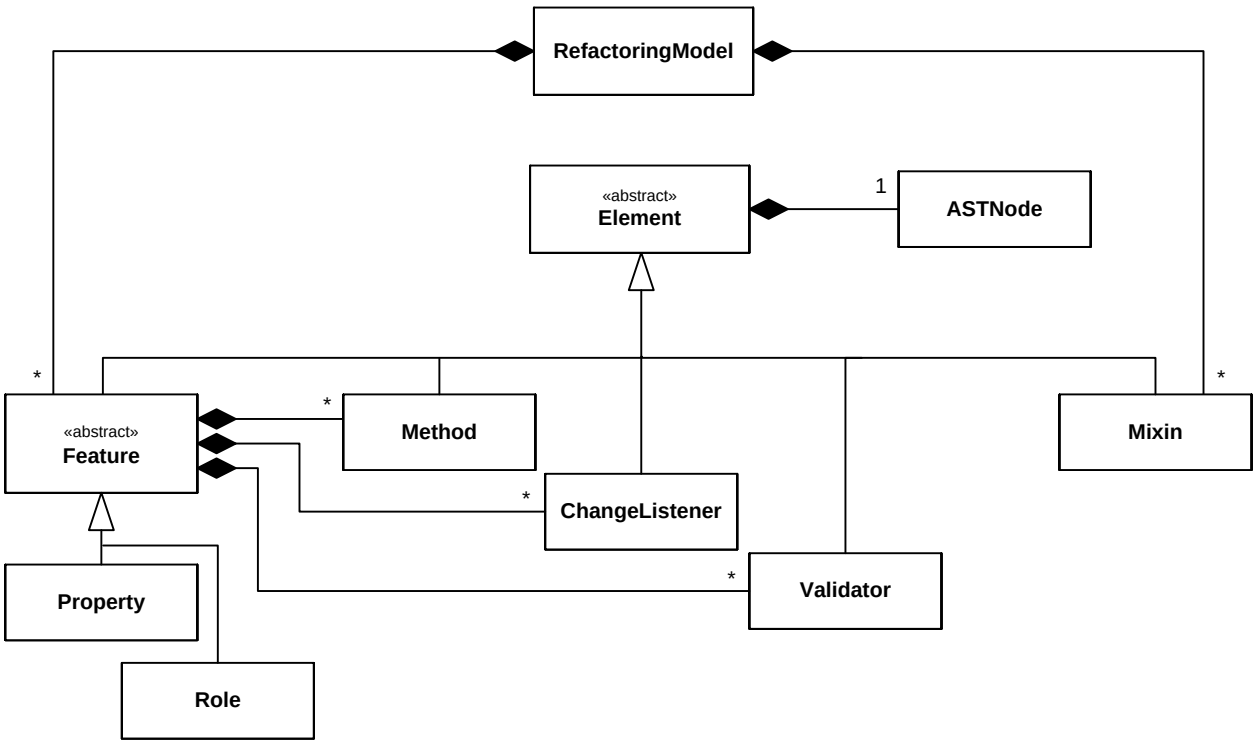



Abbildung 6.6: Round Trip Engineering: RefactoringModel

Alle Bestandteile des Modells sind also Unterklassen der abstrakten *Element*-Klasse. Properties und Roles erben wiederum von Feature, welches ebenfalls abstrakt ist und beliebig viele Methods (Methoden, die Eigenschaften eines Features repräsentieren), ChangeListener und Validators beinhaltet. Das Modell wird neben den Features durch Mixins komplementiert. Hier sei zum Vergleich auf das Metamodell in Abbildung 4.3 verwiesen.

### 6.3.3 LTK Refactoring Framework

Eclipse benutzt intern für Refactoringaufgaben, wie das Renaming, das Language Toolkit Refactoring Framework [9]. Das Framework bietet gute Schnittstellen und hilfreiche integrierte Funktionen. Die von Eclipse bekannte Möglichkeit, sich Refactoringänderungen in einer Preview, die im Ergebnisteil 6.4 abgebildet ist, anzeigen zu lassen, wird genauso unterstützt wie die Integration in die Eclipse-History. Die Benutzung der Schnittstelle besteht im Wesentlichen darin, dass Unterklassen der drei in Abbildung 6.7 gezeigten Klassen erstellt werden.

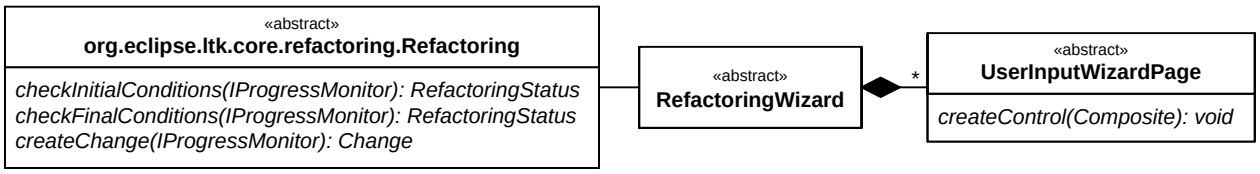



Abbildung 6.7: Refactoring LTK Schnittstelle

Ein Refactoring ist mit einem RefactoringWizard, also dem RefactoringDialog, verbunden, welcher wiederum über mehrere UserInputWizardPages verfügt. Durch das Subclassing und die Implementierung der abgebildeten abstrakten Methoden wird automatisch ein Dialog erzeugt, der von der Funktionsweise und der Optik zu Eclipse passt.

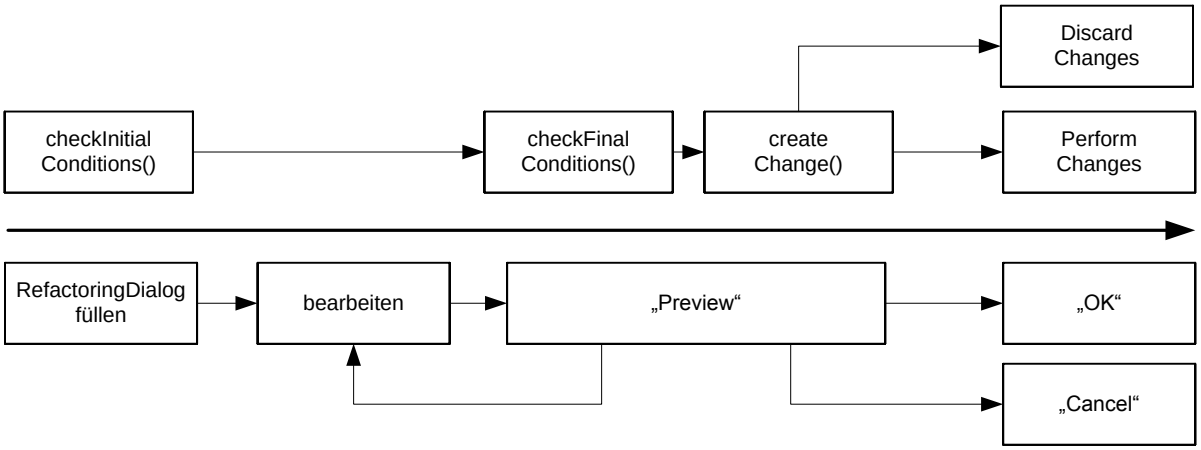



Abbildung 6.8: Refactoring Ablauf

Die zeitliche Gegenüberstellung der Methoden des Refactorings und der Benutzerinteraktionen in Abbildung 6.8 zeigt, wie die Methoden arbeiten. Wenn der RefactoringDialog aufgerufen wird, wird in der Methode `checkInitialConditions` das RefactoringModel gefüllt und der Dialog angezeigt. Nachdem Bearbeitungen durchgeführt wurden, kann eine Preview angezeigt werden. Dazu werden in `checkFinalConditions` die Änderungen gesammelt und in `createChange` zu einem Changeobjekt zusammengefasst. Je nachdem, ob die Codeänderugen dem Gewünschten entsprechen, können diese dann in den Code geschrieben oder verworfen werden. Ein Anwendungsbeispiel ist in [31] genau beschrieben und kann dort nachvollzogen werden.

## 6.4 Ergebnis

Dieser Abschnitt stellt die Realisierung der Round Trip Engineerings vor. Der Dialog, mit deren Hilfe Entitäten bearbeitet werden, ist in Abbildung 6.9 zu sehen.

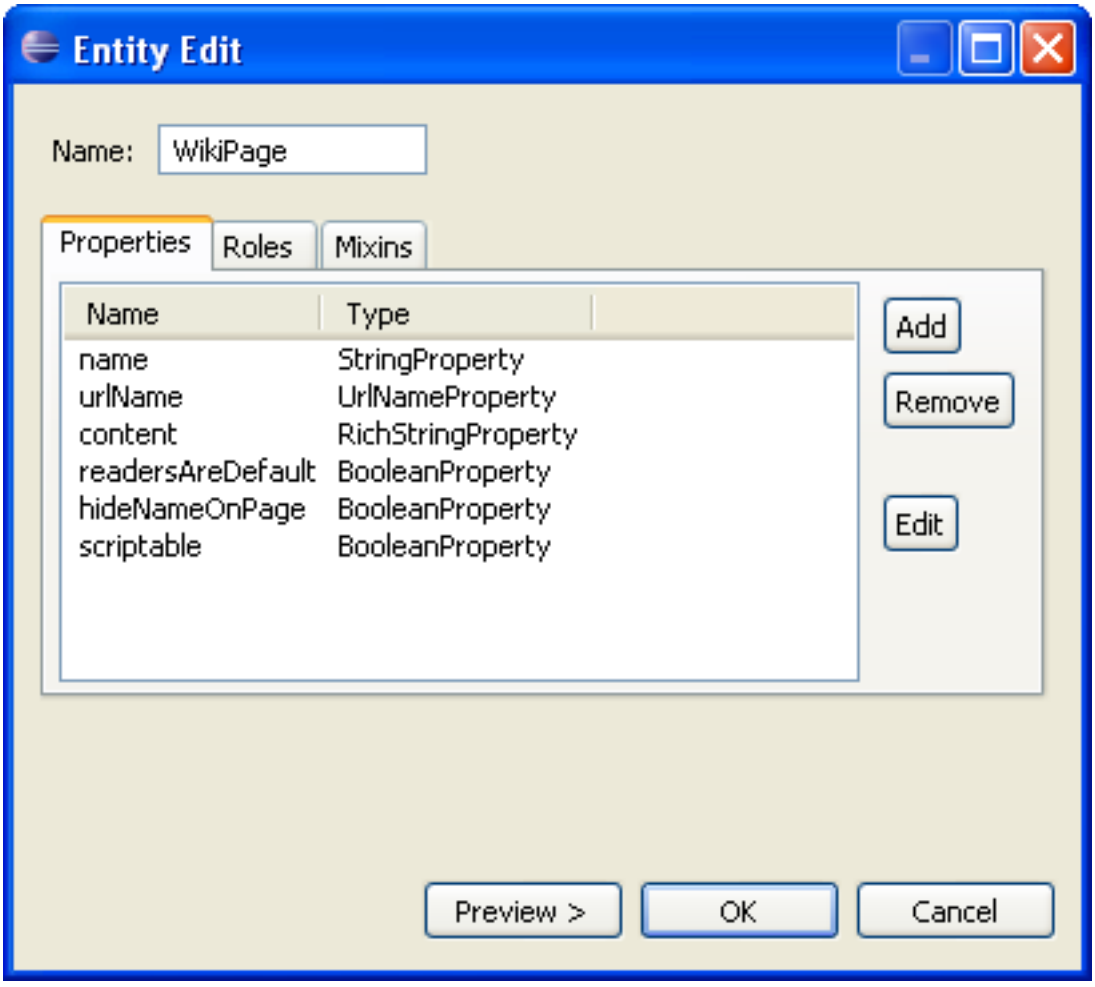



Abbildung 6.9: Round Trip Engineering Dialog: WikiPage

Hier wurde als Beispiel WikiPage gewählt, deren Properties tabellarisch aufgelistet sind. Die Roles und Mixins sind über die anderen Reiter erreichbar. Es können Features entfernt, hinzugefügt oder bearbeitet werden. Über den „Edit“-Button gelangt man bei den Roles zum Dialog 6.10.

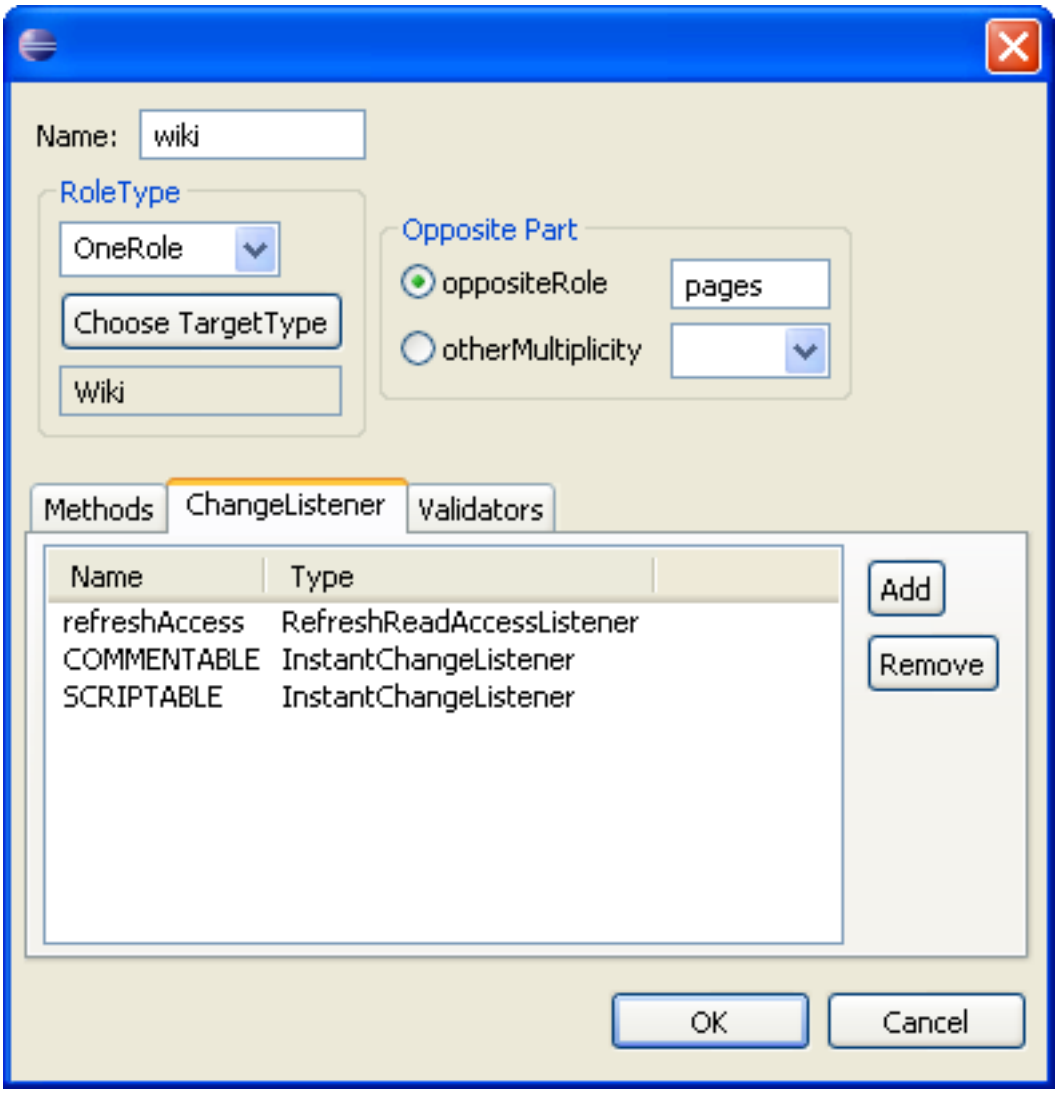



Abbildung 6.10: Round Trip Engineering Dialog: Role

Hier besteht die Möglichkeit, alle Informationen, über die sich die Role spezifiziert, zu editieren oder zu erweitern. Die Gestaltung der Dialoge wurde wieder stark vom Metamodell (4.2) beeinflusst, da Kompatibilität dazu erforderlich ist, um weiterhin Introspektion zu ermöglichen.

Nach den Änderungen besteht die Möglichkeit, sich die Codechanges in einer Vorschau anzusehen, um sie nochmals zu überprüfen. Im Beispiel im Preview 6.11 wurde die maximale Länge einer Property geändert, wodurch sie vom Standardwert abgewichen ist, weshalb eine neue Value-Methode eingefügt werden muss. Die Property urlName wurde vollkommen entfernt.

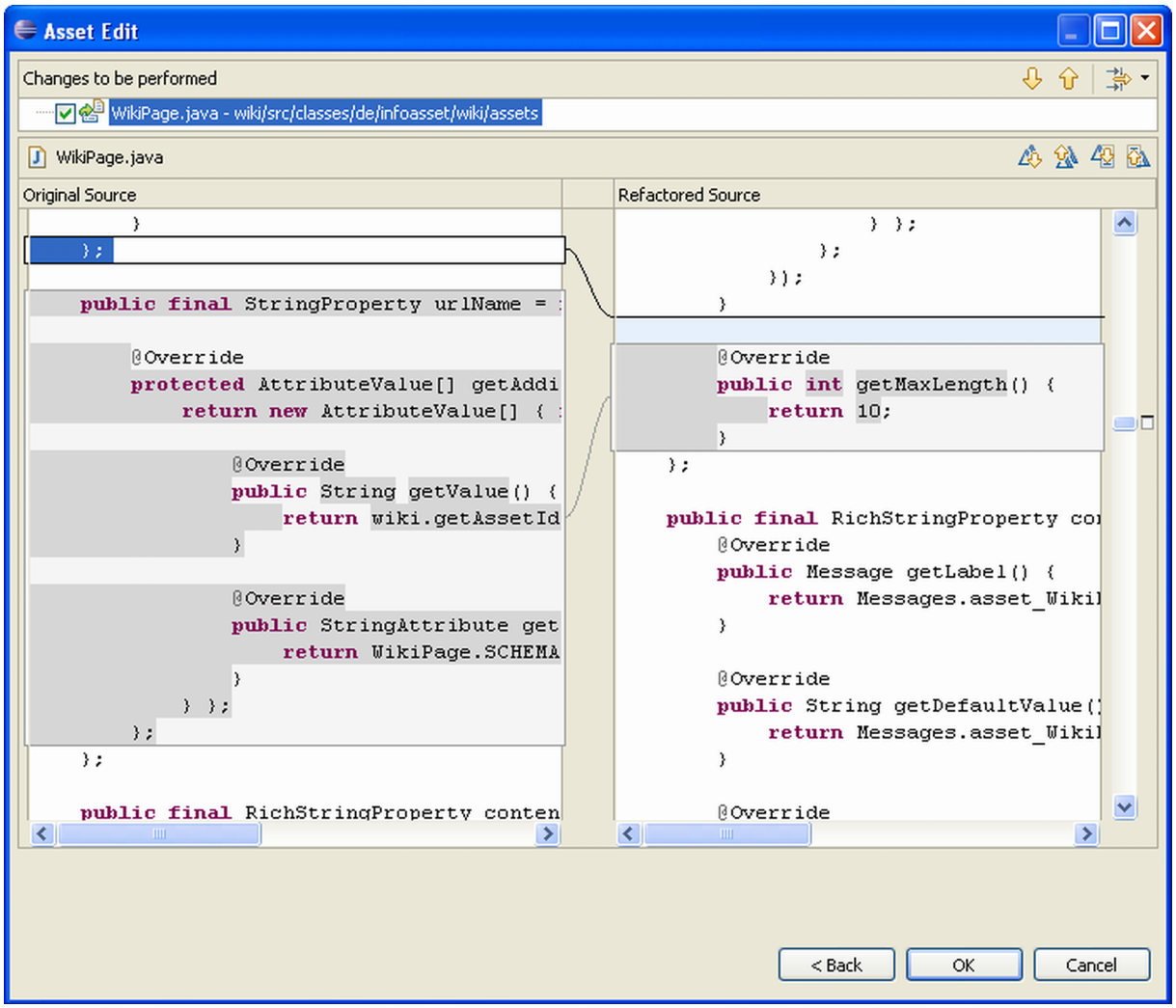



Abbildung 6.11: Round Trip Engineering Dialog: Preview

# Kapitel 7

# Schluss

Durch die modellgetriebene Softwareentwicklung wird eine hohe Modell-Code Integration erreicht. Somit werden Modelle zu wichtigen Bestandteilen der Software, wodurch sie ihrer wichtigen Rolle im Softwareengineering gerecht werden können. Durch die von Tricia bereitgestellte Introspektion stehen immer konsistente Modelle zur Verfügung, die automatisch generiert werden können. Die unterschiedlichen Weiterverwendungsmöglichkeiten der Modelle können wesentlichen Mehrnutzen produzieren.

Durch Abstraktion wird der Zugang zu Dokumentation und Struktur des Systems verbessert. Die Möglichkeit, auf die Dokumentation von Tricia in einem Wiki zuzugreifen, erleichtert den Einstieg in Tricia. Die Komplexität der Datenmodelle kann durch die visuelle Darstellung reduziert werden und erhöht dadurch das Verständnis. Durch das Round Trip Engineering wird die Erstellung und Änderung der Entitäten enorm vereinfacht.

Durch die ständige automatische Konsistenz, die bei diesen Modellen vorherrscht, wird das Problem der Modelle als Fehlerquelle, das in der Einleitung angesprochen wurde, direkt adressiert. Jegliche Nutzung stützt sich somit auf immer aktuelle Informationen, ohne dass eine ständige Wartung und Pflege notwendig ist. Dieser Ansatz wird in Zukunft wahrscheinlich weiter Verbreitung finden, da er in der Verwendung viele Vorteile mit sich bringt. Ein Problem stellt die hohe technologische Bindung an spezielle Systeme dar. In diesem Fall basiert Tricia und die in dieser Arbeit vorgestellten Erweiterungen auf Java und der Eclipse IDE. Man wird also gezwungen, mit den vorgegebenen Systemen zu arbeiten, wodurch die technologische Unabhängigkeit verloren geht.

# Abbildungsverzeichnis

# Literaturverzeichnis


[1] Software Engineering for Business Information Systems. Internetseite. Online erreichbar unter http://wwwmatthes.in.tum.de/wikis/sebis/home, aufgerufen am 3.August 2010.

[2] InfoAsset AG. InfoAsset Tricia - Open Source Web Collaboration and Knowledge Management Software. Internetseite. Online erreichbar unter http://infoasset.de, aufgerufen am 8. Mai 2010.

[3] Hailpern B. and Tarr P. Model-driven development: The good, the bad, and the ugly. 2006.

[4] Balsamiq Studios, LLC. Balsamiq Mockups. Internetseite. Online erreichbar unter http://www.balsamiq.com/, aufgerufen am 4. Juli 2010.

[5] Thomas Büchner. Introspektive modellgetriebene Softwareentwicklung. 2007. Dissertation.

[6] Bernhard Lahres and Gregor Raýman. Model View Controller. Internetseite. Online erreichbar unter http://openbook.galileocomputing.de/oo/oo_06_moduleundarchitektur_001.htm, aufgerufen am 25. Juli 2010.

[7] BitBucket. Internetseite. Online erreichbar unter http://bitbucket.org/, aufgerufen am 15. Mai 2010.

[8] Hudson Continous Integration Server. Internetseite. Online erreichbar unter http://hudson-ci.org/, aufgerufen am 15. Mai 2010.

[9] Leif Frenzel. The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs. Internetseite. Online erreichbar unter http://www.eclipse.org/articles/Article-LTK/ltk.html, aufgerufen am 25. Juni 2010.

[10] Wikipedia: Modell. Internetseite. Online erreichbar unter http://de.wikipedia.org/wiki/Modell, aufgerufen am 12. Mai 2010.

[11] Object Management Group. OMG - Homepage. Internetseite. Online erreichbar unter http://omg.org/, aufgerufen am 17. Juni 2010.

[12] Object Management Group. OMG's MetaObject Facility. Internetseite. Online erreichbar unter http://www.omg.org/mof/, aufgerufen am 2.August 2010.

[13] Object Management Group. UML - Unified Modeling Language. Internetseite. Online erreichbar unter http://uml.org/, aufgerufen am 16. Juni 2010.

[14] ORACLE Corporation. Netbeans IDE. Internetseite. Online erreichbar unter http://netbeans.org/, aufgerufen am 20. Juni 2010.

[15] ORACLE Corporation. Netbeans UML Features. Internetseite. Online erreichbar unter http://netbeans.org/features/uml/, aufgerufen am 20. Juni 2010.

[16] ORACLE Corporation. Netbeans Visual Library. Internetseite. Online erreichbar unter http://platform.netbeans.org/graph/, aufgerufen am 20. Juni 2010.

[17] ORACLE Sun Developer Network. Abstract Window Toolkit. Internetseite. Online erreichbar unter http://java.sun.com/products/jdk/awt/, aufgerufen am 20. Juni 2010.

[18] ORACLE Sun Developer Network. Javadoc. Internetseite. Online erreichbar unter http://java.sun.com/j2se/javadoc/, aufgerufen am 13. Mai 2010.

[19] ORACLE Sun Developer Network. Swing. Internetseite. Online erreichbar unter http://java.sun.com/javase/6/docs/technotes/guides/swing/index.html, aufgerufen am 20. Juni 2010.

[20] Shane Sendall and Jochen Küster. Taming Model Round-Trip Engineering. 2004.

[21] Thomas Stahl and Markus Völter. *Modellgetriebene Softwareentwicklung.* dpunkt.verlag, 2005.

[22] Stefan Eicker. Forward Engineering. Internetseite. Online erreichbar unter http://www.enzyklopaedie-der-wirtschaftsinformatik.de/wi-enzyklopaedie/lexikon/is-management/Integration-und-Migration-von-IT-Systemen/Software-Reengineering/Forward-Engineering/index.html/?searchterm=engineering, aufgerufen am 23. Juni 2010.

[23] Stefan Eicker and Reinhard Jung. Reverse Engineering. Internetseite. Online erreichbar unter http://www.enzyklopaedie-der-wirtschaftsinformatik.de/wi-enzyklopaedie/lexikon/is-management/Integration-und-Migration-von-IT-Systemen/Software-Reengineering/Reverse-Engineering/index.html/?searchterm=engineering, aufgerufen am 23. Juni 2010.

[24] The Apache Software Foundation. Apache Ant. Internetseite. Online erreichbar unter http://ant.apache.org/, aufgerufen am 16. Mai 2010.

[25] The Eclipse Foundation. Eclipse. Internetseite. Online erreichbar unter http://www.eclipse.org/, aufgerufen am 16. Juni 2010.

[26] The Eclipse Foundation. Eclipse Graphical Framework. Internetseite. Online erreichbar unter http://www.eclipse.org/gef/, aufgerufen am 20. Juni 2010.

[27] The Eclipse Foundation. Eclipse JDT Core. Internetseite. Online erreichbar unter http://www.eclipse.org/projects/project_summary.php?projectid=eclipse.jdt.core, aufgerufen am 20. Juni 2010.

[28] The Eclipse Foundation. Eclipse Rich Ajax Plattform. Internetseite. Online erreichbar unter http://www.eclipse.org/rap/, aufgerufen am 16. Juni 2010.

[29] The Eclipse Foundation. SWT/AWT Bridge. Internetseite. Online erreichbar unter http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/org/eclipse/swt/awt/package-summary.html, aufgerufen am 20. Juni 2010.

[30] The Eclipse Foundation. The Standard Widget Toolkit. Internetseite. Online erreichbar unter http://www.eclipse.org/swt/, aufgerufen am 20. Juni 2010.

[31] Tobias Widmer. Unleashing the Power of Refactoring. Internetseite. Online erreichbar unter http://www.eclipse.org/articles/article.php?file=Article-Unleashing-the-Power-of-Refactoring/index.html, aufgerufen am 25. Juni 2010.

[32] W3C. HTML 4.01 Specification. Internetseite. Online erreichbar unter http://www.w3.org/TR/html4/, aufgerufen am 08. August 2010.

[33] W3C. HTTP - Hypertext Transfer Protocol. Internetseite. Online erreichbar unter http://www.w3.org/Protocols/, aufgerufen am 08. August 2010.

[34] World Wide Web Consortium. XML - Extensible Markup Language. Internetseite. Online erreichbar unter http://www.w3.org/XML/, aufgerufen am 19. Juni 2010.

[35] XStream Committers. XStream. Internetseite. Online erreichbar unter http://xstream.codehaus.org/, aufgerufen am 22. Juni 2010.