- Diplomarbeit -

Entwicklung einer kollaborativen Erweiterung für GMF-Editoren auf Basis modellgetriebener und webbasierter Technologien

Martin Flügge

Hochschule für Technik und Wirtschaft
Fachbereich 4 – Wirtschaftswissenschaften II
Studiengang Angewandte Informatik

Gutachter:
Prof. Dr. Elke Naumann
Prof. Dr. Hermann Heßling

Berlin, den 9. Mai 2009

Abstrakt

GMF, das Graphical Modeling Framework, ermöglicht es mit Hilfe von modellgetriebenen Technologien grafische Editoren zu erzeugen. Diese Editoren können auf vielfältigste Weise genutzt werden. Es fehlt ihnen aber die Möglichkeit den Editor kollaborativ zu verwenden. Um diese Lücke zu schließen, wurde *Dawn*, eine kollaborative, webbasierte Erweiterung für GMF-Editoren, entwickelt. Dawn bietet verteilten Zugriff auf GMF-Diagramme und stellt auch Verfahren für die Erkennung und Vermeidung von Konflikten bereit. Neben der Kommunikationsinfrastruktur bietet das entwickelte System ein Konzept zur Darstellung von GMF-Diagrammen in einem Browser an. Diese Funktionalität kann dazu genutzt werden, auch auf mobilen Endgeräten GMF-Diagramme anzeigen zu können. Da mobile Endgeräte nicht immer in stabilen Netzwerken agieren, ist Dawn auch ohne Netzwerkverbindung lauffähig. Um das System vor unbefugtem Zugriff zu schützen, wurde außerdem ein generisches Nutzerkonzept entwickelt, welches auch ohne Netzwerkverbindung weiter genutzt werden kann.

GMF, the Graphical Modeling Framework, offers methods to create graphical editors with the help of model-driven techniques. Those editors can be used in various ways. Yet, no technology has been developed which allows the collaborative usage of them. To provide such a platform *Dawn*, a collaborative, web based extension for GMF-editors, had been developed. Dawn provides shared editing for GMF-diagrams as well as methods to detect and avoid conflicts. Aside the communication infrastructure the system allows viewing GMF-diagrams in web-browsers. This functionality can be used to view such diagrams on mobile devices. Because mobile devices sometimes are used in not reliable network environments *Dawn* can operate without network connection. To avoid unauthorized access to the system, a generic user concept had been developed which can also operate without connection to the Dawn server.

Inhaltsverzeichnis

1 Einleitung

„Ein Bild sagt mehr als tausend Worte“. Dieser Satz hat nicht nur eine gemeingültige Bedeutung, sondern hat in den letzten Jahren für die Softwareentwicklung immer mehr an Bedeutung gewonnen. Immer kürzere Entwicklungszyklen haben es unabdingbar gemacht, Software unter Verwendung von grafischen Hilfsmitteln zu entwickeln. Die Vorteile von Modellierungssprachen, wie der *Unified Modeling Language* (*UML*), liegen auf der Hand – die starke Visualisierung ermöglicht es, einen besseren Gesamtüberblick über das zu entwickelnde System zu erhalten und so Design-Fehler schon im Vorfeld zu erkennen.

Zusätzlich ermöglichen grafische Systembeschreibungen auch eine genauere Kommunikation zwischen den Entwicklern und dem Kunden. Basierend auf den grafischen Modellen kann in der Regel losgelöst von konkreten Implementierungen kommuniziert, und so die konzeptuelle Umsetzung in den Vordergrund gerückt werden.

Grafische Systembeschreibungen sind zudem programmiersprachenunabhängig, wodurch Programme in der Regel auf andere Programmiersprachen umgeschrieben werden können, ohne das konzeptuelle Design ändern zu müssen. Dies ist besonders interessant, wenn man für eine bestehende Business-Logik unterschiedliche Frontends anbieten möchte. Nicht zuletzt ermöglicht der grafische Überblick über das System auch neuen Mitarbeitern schneller in die Thematik einzusteigen.

1.1 Motivation

All diese Vorteile haben in den letzten Jahren sowohl auf dem kommerziellen Sektor, als auch im Open-Source Bereich dazu geführt, dass eine Vielzahl unterschiedlicher Software für die Entwicklungsunterstützung erstellt wurde.

Dabei hat sich gerade im Bereich der Java-Entwicklung die integrierte Entwicklungsumgebung *Eclipse* als ideale Plattform für die Bereitstellung grafischer Modellierer erwiesen. Ein mächtiger Funktionsumfang und eine intuitive Bedienung, in Verbindung mit einem flexiblen Plugin-Konzept, haben Eclipse zu einer der beliebtesten Entwicklungsumgebungen gemacht [vgl. Wunderlich 2006, S. 20]. Verschiedene Hersteller haben kommerzielle oder kostenlose Plugins im Bereich Software-Modellierung für Eclipse entwickelt.

Das *Graphical Modeling Framework* (*GMF*), ein Projekt der Eclipse Foundation, beschäftigt sich mit dem generativen Erstellen von grafischen Editoren. Hierbei ist es mittels modellgetriebener Softwareentwicklung möglich, grafische Editoren für die verschiedensten Anwendungsgebiete zu erstellen. *GMF* stellt sowohl einen Generator zum Erstellen der Editoren, als auch eine Laufzeitumgebung zur Benutzung dieser zur Verfügung.

Um allerdings den Modellbestand eines Editors mit einem anderen Entwickler austauschen zu können, müssen die Daten entweder über Export/Import-Mechanismen oder über Versionsverwaltungsprogramme verwaltet werden. Dabei wäre es im Zuge der weltweiten Globalisierung wesentlich sinnvoller eine Plattform zur Verfügung zu stellen, welche es ermöglicht Modelle zeitgleich von verschiedensten Orten der Welt zu bearbeiten.

So werden im Bereich „Office Applications“ schon unterschiedlichste Plattformen angeboten mit deren Hilfe kollaboratives Arbeiten einfach und komfortabel bewerkstelligt werden kann. Bekanntester Vertreter dieser Plattformen ist Google mit Google Docs[1], einer webbasierten Anwendung zum Bearbeiten von Textdokumenten, Tabellenkalkulationen oder Präsentationen.

Würde man dieses Konzept auf die von GMF erstellten Editoren übertragen, ergäben sich vielfältige neue Anwendungsgebiete. Entwickler könnten zum Beispiel direkt mit dem Kunden über das zu entstehende System kommunizieren, wobei dem Kunden über die *HTTP*-basierte (*Hypertext Transfer Protocol*) Applikation ein einfacher und komfortabler Zugang zu dem Projekt gegeben wäre. Ebenso könnten Anwender verschiedener Standorte direkt miteinander an der grafischen Modellierung arbeiten.

1.2 Zielsetzung

Im Rahmen dieser Arbeit soll ein Framework entstehen, welches es ermöglicht GMF-basierte Editoren mit Hilfe von geeigneten Netzwerktechnologien zu verbinden. Hierbei soll ein generischer Ansatz geschaffen werden, der es erlaubt auch schon im Einsatz befindliche Editoren um die zu erstellenden Funktionalitäten zu erweitern. Das zu entwickelnde System soll eine Erweiterung zu dem bestehenden *Graphical Modeling Framework* darstellen. Die Kommunikationsschnittstellen und -komponenten sollen ebenfalls, basierend auf detaillierten Modellbeschreibungen generiert werden können.

[1] https://www.google.com

Das System wird folglicherweise aus einer Laufzeitkomponente, welche eine Kommunikations- und Synchronisationsplattform zu Verfügung stellt, und einer generativen Komponente bestehen, die für neu zu entwickelnde bzw. schon existierende Editoren die entsprechenden Kommunikationsadapter erstellen kann. Über eine einheitliche Datenbasis sollen die Projektinformationen jederzeit für alle Entwickler zur Verfügung stehen, sodass sich der Entwicklerkreis dynamisch vergrößern und verkleinern kann, ohne dass Projektinformationen verloren gehen.

GMF-Editoren besitzen von Natur aus einen entscheidenden Nachteil. Da sie in der Eclipse-Umgebung laufen, benötigen sie ein gewisses Maß an Ressourcen und sind auf eine installierte Java Runtime angewiesen. Nicht jeder Netzwerkteilnehmer erfüllt diese Anforderungen. Um die verteilten Editoren aber für eine Vielzahl von Endgeräten anbieten zu können, soll das Framework zusätzlich über eine Web-Viewer Komponente verfügen, mit deren Hilfe die Bearbeitung grafischer Modelle auch mittels eines Browsers verfolgt werden kann. So ist es denkbar das System auf mobilen Endgeräten einzusetzen, die über einen internetfähigen Browser, nicht aber über Eclipse, verfügen. Die Arbeit soll hierbei die Grundlagen dafür schaffen, dass später Diagramme auch browserbasiert editiert werden können.

Zusätzlich soll ein generisches Rollen-Konzept die Rechte von einzelnen Benutzern des Systems verwalten können. Die Funktionalitäten des Frameworks werden prototypisch anhand eines verteilten Diagramms modelliert. Grundlage dieses Prototyps stellt ein vereinfachtes UML-Klassendiagramm dar, welches durch leichte Modifikationen den Funktionsumfang des Systems demonstrieren kann. Hierbei wird es nicht nur Inhalt der prototypischen Darstellung sein die Funktionsweise des Systems zu verdeutlichen, sondern auch den generativen Aspekt, also die Einfachheit der Erzeugung der kollaborativen Erweiterung zu veranschaulichen.

1.3 Aufbau der Arbeit

Im Zuge der Arbeit sollen zunächst die Grundlagen modellbasierter Entwicklungsansätze wie *Model Driven Software Development* (*MDSD*) untersucht und beschrieben werden. Dazu werden essentiellen Begriffe wie *Modell*, *Plattform* oder *Generator* erläutert und die *MDSD* mit ähnlichen Ansätzen verglichen. Im Anschluss werden Begriffe aus dem Umfeld der *Unified Modeling Language* (*UML*) erläutert. Der darauf folgende Abschnitt behandelt die *OSGi Service Platform*, welche die Grundlage des Eclipse Frameworks bildet. Dieses Wissen bereitet die Grundlage für den Aufbau von Eclipse und die in dieser Arbeit verwendeten Eclipse Frameworks *PDE* (*Plugin Development Environment*), *EMF* (*Eclipse Modeling Framework*) und *GMF* (*Graphical*

Modeling Framework), welche im Anschluss erläutert werden. Das Grundlagen-Kapitel schließt mit einer Betrachtung weiterer in der Arbeit genutzter Web- und Persistenztechnologien.

Um den Aufgabenbereich definieren zu können und einsetzbare Technologien zu betrachten, werden im Kapitel *Anforderungsanalyse und -definition* die Anforderungen an das zu entwickelnde System analysiert. Dies geschieht unter dem Fokus einen Anforderungskatalog zu erstellen, welcher als Leitfaden für Entwicklung des Konzeptes und die Implementierung dienen soll. Das umfasst die Betrachtung bestehender Netzwerk- und Generator-Frameworks und deren Begutachtung hinsichtlich der Einsetzbarkeit innerhalb der Arbeit.

Das Kapitel *Konzeptentwicklung und Systemdesign* entwickelt anhand der vorher festgelegten Anforderungen an das System die konzeptuellen Grundlagen für die Umsetzung. Es werden Konzepte für die Kommunikation und Synchronisation der Datenbestände der einzelnen Editoren erarbeitet. Da in verteilten Systemen, in denen mehrere Nutzer zeitgleichen Schreibzugriff auf Ressourcen haben, das Auftreten von Konflikten nie ausgeschlossen werden kann, werden anschließend Mechanismen entwickelt, um Konflikten begegnen zu können. Dies beinhaltet sowohl die Erkennung, Beseitigung, als auch deren Vermeidung. Im Anschluss wird erläutert, wie es dem System ermöglicht wird, auch ohne Verbindung zum Server weiter genutzt werden zu können. Dieser Betrachtung folgt die Entwicklung eines generischen Nutzer-Konzeptes, welches es einfach ermöglicht zusätzliche Einschränkungen in das System zu integrieren. Um die Daten auch vor Ausfall einzelner Systemkomponenten zu schützen, wird im Weiteren ein Persistenzkonzept erstellt. Da das System die verteilten Diagramme auch innerhalb eines Browsers darstellen können soll, wird anschließend konzipiert, wie die Elemente der Diagramme auch mit Hilfe von Web-Technologien dargestellt werden können. Um sich in den modellgetriebenen Entwicklungsprozess von *GMF* zu integrieren, wird zusätzlich ein Ansatz entworfen, wie die entwickelten Komponenten generativ erzeugt werden können. Dieser basiert unter anderem auf dem im vorhergehenden Abschnitt beschriebenen prototypischen Diagramm.

Die Beschreibung der *Implementierung* des Konzeptes behandelt im anschließenden Themenkomplex ausgewählte Umsetzungen. Hierbei werden vor allem jene Aspekte aufgegriffen, welche besonders schwierig oder interessant waren.

Testspezifische Auswertungen erfolgen im Kapitel *Tests*. Zusätzlich wird in diesem Kapitel eine Betrachtung der Performance des Systems vorgenommen.

In der *Ergebnisbetrachtung* erfolgt eine Auswertung hinsichtlich der Umsetzung der definierten Ziele. Dabei werden die Ergebnisse kritisch betrachtet und ihr Nutzen herausgestellt. Zusätzlich werden die in dieser Arbeit gewonnenen Erfahrungen resümiert und ein Ausblick auf die zukünftige Entwicklung des Systems gegeben.

1.4 Konventionen der Arbeit

Kursiv	**Für Abkürzungen und Eigennamen**
`Courier New`	Klassenbezeichner und Quellecode spezifische Bezeichner innerhalb des Fließtextes
`Courier New` und **`Courier New Fett`**	Für Quelltexte und Listing
"*Times New Roman, kursiv*"	Zitate

2 Grundlagen

Das folgende Kapitel erläutert die für diese Arbeit notwendigen Grundlagen. Um einen Einstieg in die Thematik zu erhalten, werden am Anfang die Begriffe *Modell*, *Meta-Modell* und *Meta-Meta-Modell* und ihre Zusammenhänge beschrieben. Im Anschluss geht die Arbeit auf die wichtigsten Aspekte der modellgetriebenen Softwareentwicklung (*Model Driven Software Development, MDSD*) ein und erläutert diese. Neben der Beschreibung des grundlegenden Vokabulars der *MDSD*, werden auch anverwandte Ansätze wie die generative Programmierung und die *Model Driven Architektur* (*MDA*) behandelt. Als formale Meta-Modellierungssprache für modellgetriebene Softwareentwicklung und als Basis für den zu entwickelnden Prototyp, erfolgt im anschließenden Kapitel ein Einblick in die Prinzipien der *Unified Modeling Language* (*UML*).

Der sich anschließende Abschnitt beschreibt die OSGi Service Platform Spezifikation und ihre enge Verknüpfung mit der *Eclipse IDE* (*Integrated Development Environment*). Da das Eclipse Projekt weit mehr als nur eine Arbeitsumgebung für Java-Entwicklung zur Verfügung stellt, werden im Anschluss die Grundlagen dieses Projektes näher beleuchtet. Hierbei soll ein Bewusstsein dafür geschaffen werden, wie umfangreich und vielseitig die Einsatzgebiete des Eclipse Projektes sind. Dies geht einher mit der Betrachtung der für diese Arbeit notwendigen Eclipse Frameworks *EMF* (*Eclipse Modeling Framework*) und *GMF* (*Graphical Modeling Framework*), sowie weiterer anverwandter Rahmenwerke.

Die abschließenden Kapitel beschäftigen sich mit den Grundlagen der innerhalb der Arbeit verwendeten Web- und Persistenztechnologien. Dies umfasst sowohl die Einführung in Java-basierte Web-Anwendungen, als auch aktuelle Techniken wie Webservices und Ajax.

2.1 Modelle

„*A model captures a view of a physical system. It is an abstraction of the physical system, with a certain purpose. […] Thus the model completely describes those aspects of the physical system that are relevant to the purpose of the model, at the appropriate level of detail*“ [Rupp et al. 2007, S.36].

Dieser Definition folgend stellen Modelle eine abstrakte Darstellung eines Systems dar, welche in Detaillierungsgrad und Ausprägung von der eigentlichen Zielbestimmung des

Modells abhängt. Hierbei versuchen Modelle die Komplexität vorhandener Systeme in einem Maße zu reduzieren, dass bezüglich einer konkreten Fragestellung die Aussage des Systems erhalten bleibt, für den Nutzer des Modells die Bearbeitung der Fragestellung aber wesentlich erleichtert wird.

Vereinfacht gesagt filtern Modelle Eigenschaften von komplexen Systemen hinsichtlich eines Problems und stellen diese in geeigneter Form dar. Es erfolgt also eine Reduktion der Komplexität auf die für die Problemstellung notwendigen Systemeigenschaften.

Modelle finden sich in vielen wissenschaftlichen Disziplinen. So dienen zum Beispiel in der Physik Atommodelle zur abstrakten Darstellung atomarer Vorgänge. In der diagnostischen Medizin finden Krankheitsmodelle Verwendung zur Analyse von Krankheiten, in der Architektur werden Skizzen zur vereinfachten Darstellung von Bauten genutzt und in der Wirtschaft dienen Geschäftsprozessmodelle zur Visualisierung von kundenorientierten Unternehmensvorgängen.

Diese Beispiele sollen verdeutlichen, dass Modelle nicht an eine bestimmte Form gebunden sind. Sie können sowohl in textueller, grafischer, plastischer, gedanklicher als auch kombinierter Form vorliegen.

2.2 Formale Modelle

Modelle finden also in verschiedensten wissenschaftlichen Disziplinen ihre Anwendung. Doch nicht jedes Modell lässt sich für die Softwareentwicklung nutzen. Um innerhalb von modellgetriebener Softwareentwicklung anwendbar zu sein, muss ein Modell einen Aspekt vollständig beschreiben. Ein Modell, welches diese Anforderung erfüllt, nennt sich *formal* [vgl. Stahl et al. 2007, S.11 ff.].

Die wichtigste Eigenschaft von formalen Modellen ist ihre Determiniertheit. Dies bedeutet, dass es hinsichtlich Syntax und Semantik eindeutige Regeln geben muss, welche die Auswertung des Modells immer zu demselben Ergebnis führen lassen. Diese Eindeutigkeit wird zum einen durch die Festlegung der Symbolik des Modells erreicht, die beschreibt welche Symbole innerhalb des Modells vorkommen dürfen bzw. müssen, und welche nicht. Mit Hilfe dieser Syntax wird es ermöglicht zu entscheiden ob ein Modell syntaktisch korrekt ist oder Fehler aufweist. Zum anderen müssen semantische Regeln für die Interpretation des Modell existieren, die es erlauben aus einem formalen Modell die modellspezifischen Informationen eindeutig reproduzierbar zu extrahieren [vgl. Gruhn et al. 2006, S65 ff.].

Diese Regeln werden genutzt, um den Modellinhalt derart zu spezifizieren, dass er mit Hilfe von Transformatoren in Quellcode überführt werden kann. Dieser Quellcode muss nicht, kann aber lauffähig sein.

Formale Modelle werden durch *formale Sprachen* beschrieben. Diese weisen ebenfalls die Eigenschaft der Determiniertheit auf. Hierbei gibt es verschiedene Ansätze. Die universellen Sprachen versuchen einen Sprachumfang zu erzeugen, mit dem sich nahezu alle Sachverhalte abbilden lassen. Die UML [vgl. 2.6] stellt Regeln und Elemente zur Verfügung, um eine Vielzahl von Systemen modellieren zu können. Universelle Sprachen haben allerdings den Nachteil, dass sie auf Grund ihrer Komplexität in der Regel schwer zu erlernen sind. Meist wird auch für die Modellierung eines konkreten Systems nicht der volle Funktionsumfang benötigt. Aus diesem Grund ist es oft auch sinnvoll Sprachen zu nutzen, die sich nur auf die für das System nötigen Parameter beschränken. Diese Sprachen stellen nur den benötigten Sprachumfang für eine Problemdomäne zur Verfügung. Sie werden domänenspezifische Sprachen [Domain Specific Language, DSL, vgl. 2.5.1] genannt.

2.3 Meta-Modelle

Modelle werden also mit Hilfe von geeigneten Sprachen modelliert. Diese Sprachen besitzen eine vordefinierte Syntax und Semantik. Ihnen liegt folglich ein Modell zugrunde. Dieses Modell definiert, welche Elemente und Zusammenhänge zwischen den Bestandteilen der Sprache zulässig sind und wie diese interpretiert werden. Derartige modellbeschreibende Modelle werden Meta-Modelle genannt. Die UML stellt zum Bespiel ein Meta-Modell zur Verfügung, mit deren Hilfe Modelle von Systemen erstellt werden können.

So wie Modelle ein System beschreiben, beschreiben Meta-Modelle die Sprache, mit der ein Modell erstellt werden kann. Das Meta-Modell definiert also den Sprachumfang der Meta-Sprache. Systeme werden durch Modelle beschrieben, Modelle durch Meta-Modelle [vgl. Abbildung 1].

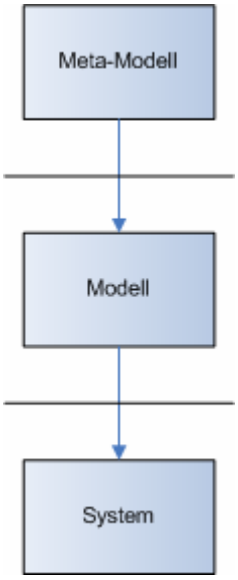

Abbildung 1 - Beziehung System-Modell-Metamodell

Das Meta-Modell spezifiziert hierbei die *abstrakte* Syntax einer modellbeschreibenden Sprache. Diese definiert die Strukturen, die zum Erstellen einer *konkreten* Syntax genutzt werden können. So spezifiziert zum Beispiel die abstrakte Syntax der UML, dass Klassendiagramme über Klassen- und Interfacebeschreibungen verfügen und welche Elemente in ihnen erlaubt sind.

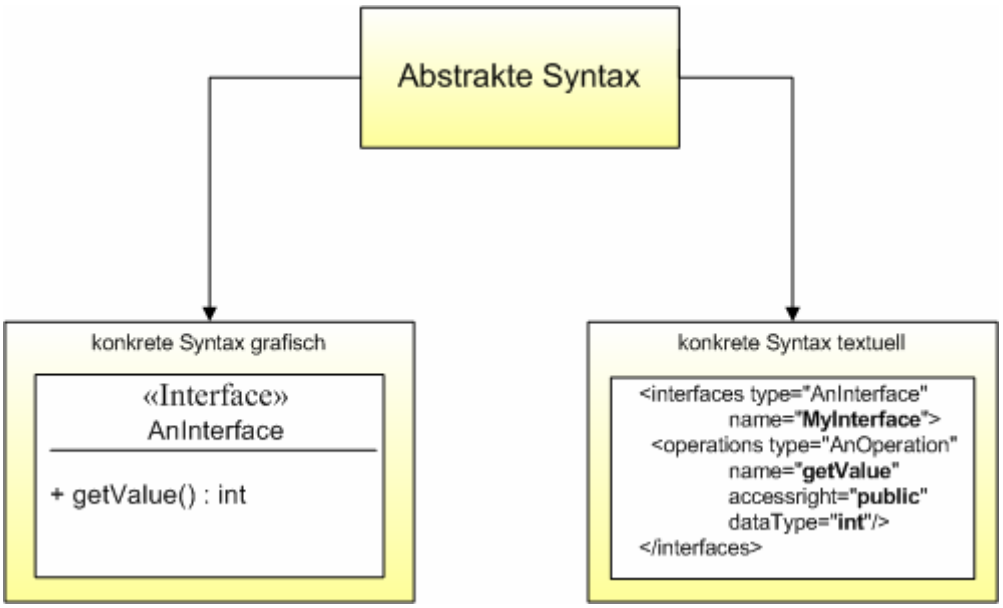

Abbildung 2 - Beispiel abstrakte/konkrete Syntax

Aus der abstrakten Syntax kann dann die konkrete Syntax derart abgeleitet werden, dass beispielsweise Interfaces immer durch ein Rechteck mit dem Schlüssel *«Interface»* dargestellt werden. Eine abstrakte Syntax kann allerdings durch mehrere konkrete Syntaxen implementiert werden. So muss in dem vorhergehenden Beispiel ein Interface

nicht unbedingt durch eine grafische Repräsentation symbolisiert werden, sondern kann auch als Text, zum Beispiel als XML, dargestellt werden. Abbildung 2 veranschaulicht dieses Beispiel.

Eine abstrakte Syntax kann demzufolge durch verschiedene konkrete Syntaxen, grafisch oder textuell, dargestellt werden [vgl. Gruhn et al. 2006, S.68].

2.4 Meta²-Modell

Meta-Modelle können ihrerseits wieder formal beschrieben werden. Hierbei wird zum Beispiel definiert, welche Elemente Meta-Modelle enthalten sollen oder welchen Regeln sie folgen. Diese Beschreibung wird in Form eines Meta-Meta-Modells (auch Meta²-Modelle) festgehalten. Meta-Meta-Modelle werden also genutzt, um Meta-Modelle zu erzeugen.

Ein einfaches Beispiel soll den Zusammenhang der Modellebenen verdeutlichen. In einem Softwaresystem sei die Klasse *Person* erstellt worden. Diese Klasse besitzt die Eigenschaften *Vorname* und *Nachname*. Wird diese Klasse instanziiert, entsteht ein konkretes Objekt, dem spezifische Eigenschaften zugewiesen werden können. Diese Klasse kann zum Beispiel mit Hilfe eines Klassendiagramms (*Modell*) beschrieben werden. Alle Elemente, welche in diesem Klassendiagramm genutzt werden können, sind von der UML umschrieben und spezifiziert (*Meta-Modell*). Es ist beispielsweise nicht möglich innerhalb eines UML-Klassendiagramms einen roten Kreis als Symbolisierung einer Klasse zu nutzen, da dieses Element nicht Bestandteil innerhalb der Klassendiagramm-Spezifikation der UML ist. Um nun seinerseits die UML beschreiben zu können, bedarf es eines stärker abstrahierten Modells – dem Meta-Meta-Modell.

An einem weiteren Beispiel sollen die Zusammenhänge zwischen den unterschiedlichen Begriffen veranschaulicht werden. Ein Stadtplan beschreibt den Verlauf von Straßen, Wegen, Kreuzungen und anderen in der Topographie eines Ortes vorkommenden Elementen. Er bildet also real existierende Dinge in einer vordefinierten Form ab. Der Stadtplan ist also das Modell der Realität, nämlich von Straßen und Orten. Jeder Stadtplan besitzt eine Legende, mit deren Hilfe es möglich ist einzelne Straßen und Gelände voneinander zu unterscheiden. So können beispielsweise Hauptstraßen als gelbe und Autobahnen als rote Linien dargestellt werden. Waldflächen können sich von Parkanlagen durch ein Baum-Symbol abgrenzen. Mit Hilfe dieser Informationen ist es möglich viele verschiedene Stadtpläne von den unterschiedlichsten Orten zu erstellen. Alle basieren sie aber auf derselben Legende und können auf dieselbe Weise

interpretiert werden. Ihnen liegt also das gleiche Meta-Modell zu Grunde. Bekannterweise gibt es aber verschiedene Hersteller von Stadtplänen. In der Regel haben diese Hersteller gemeinsam, dass sie unterschiedliche Legenden verwenden, um ihre Stadtpläne darzustellen. Wo die einen Hauptstraßen als gelbe Linien darstellen, nutzen andere wiederum orange-gestrichelte Strecken. Sie verwenden also unterschiedliche Meta-Modelle, um ihre Stadtpläne zu erzeugen. Gemeinsam ist aber allen, dass sie bestimmte Sachverhalte immer darstellen. So wird es wohl keinen Stadtplan geben, in dem keine Hauptstraßen, Wege oder Parkanlagen verzeichnet sind. Alle Pläne nutzen also für die Beschreibung ihrer Legenden (Meta-Modell) eine gemeinsame Basis – ein Meta-Meta-Modell [vgl. Abbildung 3].

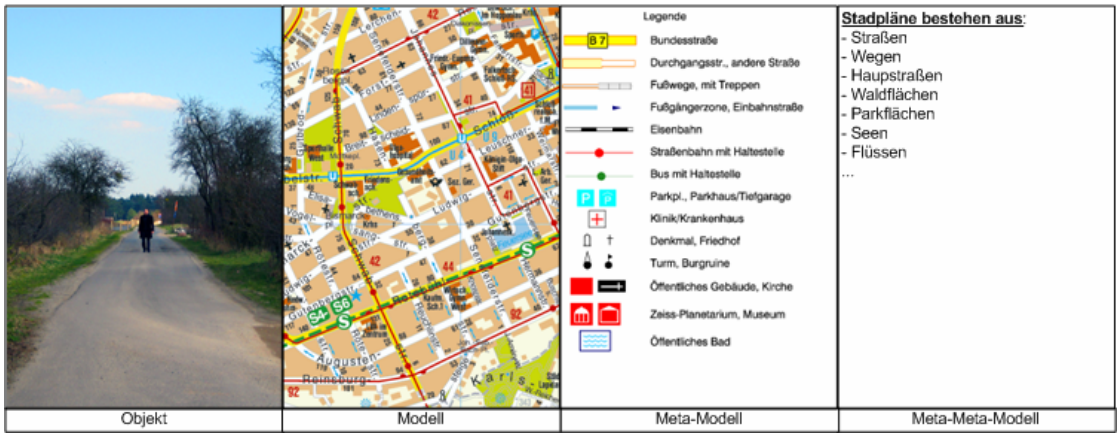

Abbildung 3 - Beispiel Stadtplan

Dieses Beispiel soll verdeutlichen, dass es darauf ankommt, von welchem Blickwinkel aus ein Modell betrachtet wird. Aus dem Blickwinkel der Objekte ist der Stadtplan das Modell und die Legende das Meta-Modell. Wird aber der Stadtplan betrachtet, so ist die Legende nur das dem Stadtplan zugrunde liegende Modell.

2.5 MDSD – Model Driven Software Development

Model Driven Software Development (*MDSD*) ist ein Konzept aus der Software-Entwicklung. Nach [Stahl et al. 2007, S.11] definiert sie sich wie folgt:

„Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen.“

Grundlage der MDSD bilden also formale Modelle [vgl. 2.2], aus denen mittels geeigneter Werkzeuge Quellcode erzeugt wird. Dieser Quellcode muss nicht zwangsläufig direkt zu einem lauffähigen Programm führen. Ziel der MDSD ist es aber

Modelle derart zu spezifizieren, dass manuelle Änderungen am Quellcode nicht mehr benötigt werden. Um aus dem Modell Quellcode zu erzeugen, werden Generatoren benutzt [vgl. 2.5.3.4]. Dieser Vorgang nennt sich *Model-to-Code Transformation* (*M2C*) oder auch *Model-To-Text Transformation* (*M2T*) [vgl. 2.5.3.4].

Neben der Abkürzung MDSD haben sich auch noch weitere Begriffe für modellgetriebene Softwareentwicklung etabliert. So finden sich ebenfalls die Abkürzungen *MDD* (*Model Driven Development*) und *MDSE* (*Model Driven Software Engineering*), welche synonym zu modellgetriebener Softwareentwicklung genutzt werden können. Innerhalb dieser Arbeit wird ausschließlich die Abkürzung MDSD verwendet. Die nachfolgenden Kapitel beschäftigen sich mit den grundlegenden Begriffen der *MDSD*.

2.5.1 Domänen und domänenspezifische Sprachen

Domänen bezeichnen ein abgeschlossenes Wissens- oder Arbeitsgebiet. Sie dienen innerhalb der MDSD zur Abgrenzung von anderen Bereichen. Die Domäne bildet die Grundlage für das zu entwickelnde Meta-Modell, welches die Konzepte der Domäne erfasst und spezifiziert. Diese Abgrenzung ist notwendig, um das den Modellen zugrunde liegende Meta-Modell so übersichtlich und verständlich wie möglich zu halten. Domänen können sowohl fachlicher als auch technischer Natur sein, wobei die Unterscheidung anhand der Art erfolgt, wie die Domäne betrachtet wird [vgl. Stahl et al. 2007, S.28].

Domänenspezifische Sprachen (*Domain Specific Language, DSL*) sind auf eine bestimmte Domäne zugeschnitten. Sie basieren auf der in dem Meta-Modell der Domäne spezifizierten abstrakten Syntax und statischen Semantik und setzen diese durch konkrete Implementierungen um. DSLs können sowohl grafischer, textueller als auch hybrider Natur sein.

Domänenspezifische Sprachen stehen im Gegensatz zu allgemeinen Sprachen wie der UML. Ihr Ziel ist es, nur den für die Lösung eines Problems notwendigen Umfang zu implementieren. Sie sind deshalb in der Regel nicht auf eine andere Domäne übertragbar. Ein weiterer Vorteil liegt darin, dass sie von Domänenspezialisten leichter erlernbar sind als universelle Sprachen, da sich das Vokabular der Syntax an dem Fachvokabular der Domäne orientiert und die Semantik der DSL versucht die Sachverhalte der Domäne wirklichkeitsgetreu wiederzugeben. Sie können als Programmiersprachen der Domäne angesehen werden, was es ihnen erlaubt wesentlich einfacher Programme zu erstellen, da sie weniger abstrakt und mehr auf das Problem

zugeschnitten sind als weit verbreitete Hochsprachen wie Java, C++ oder C# [vgl. Gruhn et al. 2006, S.70 ff.]. Ein Entity-Relationship-Diagramm ist beispielsweise eine grafische DSL für die Modellierung von Datenbanken.

2.5.2 Plattformen

Plattformen stellen innerhalb der *MDSD* das Fundament für die Ausführung des generierten Codes dar. Plattformen können sowohl Betriebssysteme (Windows, Unix, Linux) als auch Programmiersprachen (Java, C#) oder Frameworks (Spring, JSF, Struts), aber auch bestimmte Hardware sein. Oft stellt sich die Plattform als Kombination dieser einzelnen Komponenten dar [vgl. Gruhn et al. 2006, S.26]. Eine mögliche Plattform wäre zum Beispiel eine Java Virtual Machine und darauf aufbauend das Spring Framework in Kombination mit Hibernate für die Persistenz der Anwendung.

2.5.3 Modelle der MDSD

Die modellgetriebene Softwareentwicklung basiert auf der Nutzung von Modellen. Innerhalb der MDSD werden verschiedene Modelle genutzt. Die folgenden Kapitel beschreiben die wichtigsten in der MDSD genutzten Modelle und ihre Einsatzgebiete. Die folgenden Modelle wurden innerhalb der MDA (Modell Driven Architekture), einem Spezialfalls der MDSD entwickelt.

2.5.3.1 CIM – Computation Independent Model

Das *Computation Independent Model* (*CIM*) bietet eine abstrakte Sicht auf das System, welche losgelöst von jeglicher Implementierung ist. Es beschreibt eine Domäne mit dem in ihr üblichen Vokabular und wird deshalb auch als Domänenmodell bezeichnet. Da es unabhängig von jeglicher Umsetzung, allein die Sachverhalte der Domäne beschreibt, ist es einfach von den Anwendern der Domäne zu verstehen. Es kann also zur Kommunikation mit dem Kunden oder Endanwender eines Systems genutzt werden [vgl. Gruhn et al. 2006, S.27].

2.5.3.2 PIM – Platform Independent Model

Das *Platform Independent Model* (*PIM*) beschreibt die funktionellen und strukturellen Aspekte eines Systems [vgl. Gruhn et al. 2006, S.128]. Dabei ist es unabhängig von der

späteren konkreten Implementierung. Das PIM ist nicht an die sprachliche Umsetzung gebunden, also ob ein System in Java, C++ oder C implementiert wird.

Plattformunabhängige Modelle können mit Hilfe von Modellierungssprachen, wie der UML dargestellt werden und enthalten neben den statischen Darstellungen des Systems auch dynamische Aspekte, welche zum Beispiel mit Hilfe von Aktivitätsdiagrammen dargestellt werden [vgl. Gruhn et al. 2006, S.27].

2.5.3.3 PSM – Platform Specific Model

Plattformspezifische Modelle (*Platform Specific Modell, PSM*) stellen eine Erweiterung des PIM hinsichtlich implementierungsrelevanter Details dar. Hierbei werden die plattformunabhängigen Definitionen des PIM um plattformspezifische Informationen, wie konkrete Datentypen, erweitert. Dabei ist entscheidend, dass die Zielplattform (z.B. Java, C++, RMI) bekannt ist, denn nur so kann das Modell richtig erstellt werden. Auf Basis eines PIM können mehrere plattformspezifische Modelle erstellt werden, die unterschiedliche Plattformen repräsentieren. Da das PSM alle plattformspezifischen Informationen besitzt, wird es genutzt, um den Quellcode der Anwendung zu generieren [vgl. Gruhn et al. 2006, S.27].

2.5.3.4 Modell-Transformationen

Modelle können durch Transformationen in andere Zustände überführt werden. So kann beispielsweise ein PIM durch Anreicherung von Informationen in ein PSM umgewandelt werden. Ist, wie in diesem Fall, das Ergebnis einer Transformation ein anderes Modell, so spricht man von einer *Model-to-Model-Transformation* (*M2M*). Hierbei wird das Ursprungsmodell, welches die Grundlage für die Transformation bietet, als Quellmodell bezeichnet. Das Ergebnis der Transformation wird Zielmodell genannt [vgl. Stahl et al. 2007, S.33]. Ein Quellmodell kann dabei mehrere Zielmodelle haben. Plattformunabhängige Modelle können zum Beispiel in Modelle für unterschiedliche Plattformen transformiert werden. Denkbar wäre die Umwandlung des Domänenmodells in ein Windows- und ein Linux-repräsentierendes Modell, welches jeweils betriebssystemspezifische Erweiterungen enthält. .

Aus Modellen kann aber auch Quellcode für eine bestimmte Plattform erzeugt werden. Diesen Vorgang nennt man *Model-to-Code-Transformation* (*M2C*). Wie bei der *M2M* kann die *Model-to-Code-Transformation* mehrere Ergebnisse haben, zum Beispiel in Form von Quellcode für unterschiedliche Programmiersprachen. Jedes Ergebnis ist

dabei aber plattformabhängig. Die *Model-to-Code-Transformation* wird auch als *Generierung* bezeichnet [vgl. Stahl et al. 2007, S.33]

2.5.4 Generatoren

Um ein Modell in einen anderen Zustand zu überführen, also zu transformieren, und so, um spezifische Informationen zu erweitern, werden Generatoren benutzt. Generatoren stellen ein Stück Software dar, welche die Abbildung von einem Modell auf ein anderes realisieren. Hierbei kann das erzeugte Zielmodell entweder eine Erweiterung der Quelle oder direkt Quellcode sein [vgl. Abbildung 4]. Der Einsatz von Generatoren hat den Vorteil, dass das Ergebnis jederzeit reproduzierbar ist. Vereinfacht gesagt – Generatoren machen keine Flüchtigkeitsfehler.

Abbildung 4 - Transformation von PIM bis zum Quellcode

Neben selbst implementierten Generatoren bietet sich auch die Nutzung von Generator-Frameworks an, welche mit Hilfe von Templates einfach und effizient Modelle transformieren können [vgl. Stahl et al. 2007, S.12 ff.].

Generatoren basieren immer auf einer Plattform, da sie das Quellmodell um die plattformspezifischen Details erweitern. Andernfalls wäre das Quellmodell gleich dem Zielmodell, was den Vorgang der Transformation überflüssig machen würde.

2.5.5 Vor- und Nachteile der MDSD

Die Vorteile modellgetriebener Softwareentwicklung sind klar erkennbar. Durch die unterschiedlichen, modellbasierten Abstraktionsstufen, können die an der Systemerstellung beteiligten Parteien auf unterschiedlicher Ebene miteinander kommunizieren. So kann mit dem Kunden auf CIM-Ebene (Domänen-Modell) das Projekt besprochen werden. Durch die Abwesenheit von implementierungsspezifischen Details, wird der Kunde in die Lage versetzt, sich ganz auf die Fachlogik konzentrieren zu können. Zusätzlich gestattet diese Abstraktion den Domänenexperten das ihnen vertraute Vokabular zur Beschreibung des Systems zu benutzen, da das PIM auf der entwickelten domänenspezifischen Sprache aufbaut.

Da das erstellte Generat immer auf einem höheren Modell aufbaut, können Änderungen schnell und einfach vorgenommen werden. Ändert man beispielsweise den Namen eines Bezeichners innerhalb des Domänenmodells, so hat dies direkten Einfluss auf die nachfolgenden Modelle (PSM, Quellcode). Weil diese Transformation mittels Generatoren vorgenommen wird, reduziert sich auch die Anzahl der Fehler durch Änderungen. Dem obigen Beispiel folgend genügt es, meist Tool-gestützt, die nachfolgenden Modelle neu zu generieren. Manuelle Änderungen werden so in der Regel nicht mehr benötigt. Dies erzeugt, im Zusammenhang mit der Reduktion von Fehlern, auch eine zeitliche und somit finanzielle Ersparnis bei der Erstellung von Softwaresystemen. Beliefert ein Unternehmen beispielsweise mehrere Kunden mit derselben Software, die nur durch kundenspezifische Änderungen (Logos, Layout, kleinere Anpassungen) ergänzt wird, so könnten diese in jeweils einem Modell für jeden Kunden erstellt werden. Hierbei werden in den Modellen nur die kundenspezifischen Parameter verändert. Obwohl der genutzte Generator sich nicht ändert, kann trotzdem eine an den Kunden angepasste Software erstellt werden, ohne eine Zeile Quelltext ändern zu müssen.

Ein weiterer Vorteil im Zusammenhang mit der Verwendung der MDSD kann sich durch eine Reduzierung der zu versionierenden Dateien ergeben. Da der Quellcode aus einem Modell generiert wird, müssen lediglich das Modell und der Generator versioniert werden. Dies kann bei großen Softwareprojekten zu einer erheblichen Einsparung führen.

Der Umgang mit modellgetriebener Softwareentwicklung zwingt allerdings auch oft zum Umdenken in der angewendeten Programmierpraxis. Nicht mehr der Quellcode steht im Vordergrund, sondern das diesen generierende Modell. Dies hat zum Vorteil, dass Modelle nicht mehr alleine zum Zwecke der Dokumentation genutzt werden, sondern integraler Bestandteil des Softwareerstellungsprozesses sind. Allerdings erfordert solch ein Vorgehen auch, dass der komplette Prozess auf das modellgetriebene Paradigma umgestellt werden muss.

2.5.6 Anverwandte Disziplinen

In den folgenden Kapiteln, sollen zum besseren Verständnis, verwandte Technologien zur modellgetriebenen Softwareentwicklung aufgezeigt und Unterschiede beleuchtet werden.

2.5.6.1 Generative Programmierung

Der Grundgedanke der *Generativen Programmierung* liegt in der automatischen Erzeugung von Quellcode mit Hilfe eines Generators. Im Gegensatz zur MDSD strebt die generative Programmierung allerdings nach kompletter Automatisierung des Generierungsvorganges. Das bedeutet, dass das Generat, also der Programmcode, möglichst ohne zusätzliche Änderungen lauffähig ist. Aus einem Problemraum, der die formalen Anforderungen für ein System spezifiziert, wird mit Hilfe von Konfigurationswissen der Lösungsraum generiert, bei dem es sich um das komponentenbasierte Endprodukt handelt [vgl. Stahl et al. 2007, S. 38]. Nach Stahl stellt die *Generative Programmierung* einen Spezialfall der MDSD dar.

2.5.6.2 MDA – Model Driven Architecture

Die *Model Driven Architecture* (*MDA*) ist ein weiteres Vorgehensmodell zur Erzeugung modellbasierter Softwaresysteme. Es wurde von der *Object Management Group* (*OMG*[2]) entwickelt [vgl. Gruhn et al. 2006, S.20]. Sie stellt ebenfalls im engeren Sinne eine Spezialisierung der *MDSD* dar, da die *MDA* das *Meta Object Facility* [*MOF*, vgl. 2.6.2] als einziges Meta-Meta-Modell zulässt. Dadurch lassen sich nur DSLs erstellen, die auf diesen Standard aufsetzen. Basis dieser Erweiterung sollten nach MDA UML-Profile sein, was zur Konsequenz hat, dass auch die erstellten domänenspezifischen Sprachen UML-konform bzw. -abhängig sind. So wie die abstrakte Syntax durch UML-Profile beschränkt wird, legt sich die MDA im Gegensatz zur MDSD auch bei der Grundlage für die statische Semantik fest. Diese basiert auf der *Object Constraint Language* [*OCL*[3]].

Wie bei der Syntax und bei der Semantik, gibt die OMG auch bei der Transformation von Modellen die Technologie vor. Hierbei wird *Query View Transformation* (*QVT*) als Sprache für die Model-to-Model-Transformation genutzt, welche Bestandteil der OMG MOF Spezifikation ist.

Der große Unterschied zwischen MDSD und MDA besteht also in der eingeschränkten Freiheit, die auf Grund der Vorgaben und Spezifikationen seitens der OMG vorhanden ist, bei der Wahl der einzusetzenden Technologien. Andererseits bietet die MDA aber auch einen bereits vordefinierten Workflow und standardisierte Werkzeuge für den modellgetriebenen Softwareprozess.

---

[2] http://www.omg.org

[3] Mit Hilfe der OCL können Bedingungen (*Constraints*) für das Modell erstellt werden, welche einzelne Bereiche des Modells einschränken oder aber eingrenzen [vgl. Rupp et al. 2006, S. 106 ff.].

2.6 UML – Unified Modeling Language

Die *Unified Modeling Language* (*UML*) ist eine Sprache zur Modellierung, Dokumentation und Visualisierung von Softwaresystemen. Sie entstand Ende der 90er Jahre aus der *Unified Method* (*UM*) von James Rumbaugh und Grady Booch und der Analyse- und Entwurfsmethodik *OOSE* (*Object-oriented Software Engineering*) von Ivar Jacobson, als Versuch die damals unterschiedlichsten Entwurfsmethoden zu vereinheitlichen und einen gemeinsamen Standard zu schaffen. Bis zur Version 1.0 entwickelten die Konzepte ausschließlich ihre drei Gründer, welche später auch als „die drei Amigos“ bezeichnet wurden. In folgenden Versionen verstärkte sich der Einfluss führender Unternehmen immer mehr, und im November 1997 wurde die UML von der *Object Management Group* (*OMG*) zum Standard erhoben, welche seitdem als Herausgeber des Standards fungiert. Anfangs noch als zu komplex und schwer erlernbar eingestuft, verbesserte sich die UML mit der Version 2 im Jahre 2005 und passte sich den Veränderungen in der Softwareentwicklung an. Im November 2007 wurde die aktuellste Version 2.1.2 verabschiedet [vgl. Rupp et al. 2007, S.12 ff.].

Die UML ist eine Sprache zur Modellierung von Systemen der realen Welt. Sie dient also zur Beschreibung und Erstellung von Modellen und der Modellierung von Systemen. Ihre Anwendungsgebiete reichen von einfacher System-Dokumentation bis hin zur Erstellung von formalen Modellen, mit deren Hilfe Quellcode erzeugt werden kann.

2.6.1 Diagrammwelten

Die Version 2 der UML umfasst insgesamt 13 verschiedene Diagrammtypen. Sechs davon werden zur statischen, strukturierten Darstellung von Systemen verwendet. Man nennt sie *Strukturdiagramme*. Die anderen sieben Diagrammtypen beschreiben das Verhalten von Systemen und werden konsequenterweise *Verhaltensdiagramme* genannt. Sie unterteilen sich zusätzlich in *Interaktionsdiagramme*, welche den Nachrichtenaustausch zwischen Objekten darstellen, und *nicht interaktive Verhaltensdiagramme*. Die nachfolgende Tabelle gibt einen Überblick über die Diagrammwelt der UML 2.

		Diagramm	Einsatzbereich
Strukturdiagramme		**Klassendiagramm**	Klassendiagramme beschreiben, aus welchen Klassen das System modelliert werden kann und in welchen Beziehungen die Klassen zueinander stehen.
		Paketdiagramm	Paketdiagramme dienen dem Strukturieren von Projekten, indem sie einzelne Einheiten in Pakete gliedern.
		Objektdiagramm	Objektdiagramme zeigen einen aktuellen Zustand von Objekten in einem System. Sie zeigen die instanziierten Klassen mit ihren aktuellen Parametern und Beziehungen zueinander.
		Kompositionsstruktur-diagramm	Kompositionsstrukturdiagramme stellen das Innenleben von Systemteilen, zum Beispiel Klassen, dar.
		Komponenten-diagramm	Komponentendiagramme zeigen die Komponenten des Systems und ihre Schnittstellen.
		Verteilungsdiagramm	Mit Verteilungsdiagrammen wird die Verteilung der einzelnen Softwarekomponenten auf Hardwarekomponenten dargestellt. Zusätzlich werden die Kommunikationsknoten zwischen den einzelnen Strukturelementen sichtbar gemacht.
Verhaltensdiagramme	Nicht interaktive Diagramme	**Use-Case-Diagramm**	Use-Case-Diagramme oder auch Anwendungsfalldiagramme zeigen die Funktionsweise des Systems aus Sicht von Benutzern oder externen Systemen an. Es stellt die Beziehungen zwischen Benutzern und den einzelnen Use-Cases dar.
		Aktivitätsdiagramm	Aktivitätsdiagramme werden in der Regel zur Ablaufdarstellung von Use-Cases benutzt. Es kann aber für alle flussorientierten Abläufe innerhalb des Systems genutzt werden. Sie können so stark detailliert werden, dass in vielen Fällen Programmcode aus ihnen generiert werden kann. Neben normalen Programmflüssen können mit ihnen auch nebenläufige Programmabläufe dargestellt werden.
		Zustandsdiagramm	Gibt einen Überblick über die möglichen Zustände eines Teilsystems, zum Beispiel eines Objektes. Einzelnen Zuständen können Verhalten zugeordnet werden, jeweils, wenn ein Objekt einen Zustand erreicht, ihn verlässt oder sich in ihm befindet.
	Interaktionsdiagramme	**Sequenzdiagramm**	Sequenzdiagramme stellen zeitlich den Nachrichtenaustausch zwischen Ausprägungen, zum Beispiel zwischen Objekten oder Methoden dar. Es können sowohl synchrone als auch nicht-synchrone zeitliche Abläufe dargestellt werden.
		Kommunikations-diagramm	Kommunikationsdiagramme stellen die Informationen dar, welche von Kommunikationspartnern ausgetauscht werden. Im Gegensatz zum Sequenzdiagramm steht der Gesamtüberblick über die Nachrichten und nicht die zeitliche Abfolge im Vordergrund.
		Zeitverlaufsdiagramm	Zeitverlaufsdiagramme geben ein exaktes Bild vom zeitlichen Ablauf von Funktionen und Zuständen des Systems. Sie werden benutzt, wenn eine präzise Planung des zeitlichen Verhaltens notwendig ist.
		Interaktionsübersichts-diagramm	Interaktionsübersichtsdiagramme modellieren einen Gesamtüberblick über die Interaktionen des Systems. Sie fassen somit alle anderen Aktivitätsdiagramme auf einer höheren Ebene zusammen.

Tabelle 1 - Die 13 Diagramme der UML 2
[vgl. Gruhn et al. 2006]

2.6.2 MOF – Meta Object Facility

Die *Meta Object Facility* (*MOF*) Spezifikation der *OMG* ist ein Framework für die Definition und Manipulation von Daten und Metadaten. Gleichzeitig ist es das Meta-Meta-Modell der OMG. So baut zum Beispiel die UML auf dem MOF Standard auf. Es dient also zum Spezifizieren von Meta-Modellen bzw. deren Modellierungssprachen. Dabei ist es vollständig plattformunabhängig. Das *Meta Object Facility* (*MOF*) beinhaltet ein Meta-Ebenenmodell der OMG. Es greift die in den Kapitel 2.1 bis 2.4 dargestellten Zusammenhänge zwischen Modellen auf und stellt diese in einem Architekturmodell dar. Das MOF ist dabei ein Teil des UML Standards, welcher sich mit der Modellierung von Klassen beschäftigt [vgl. OMG 2006, S.5 ff].

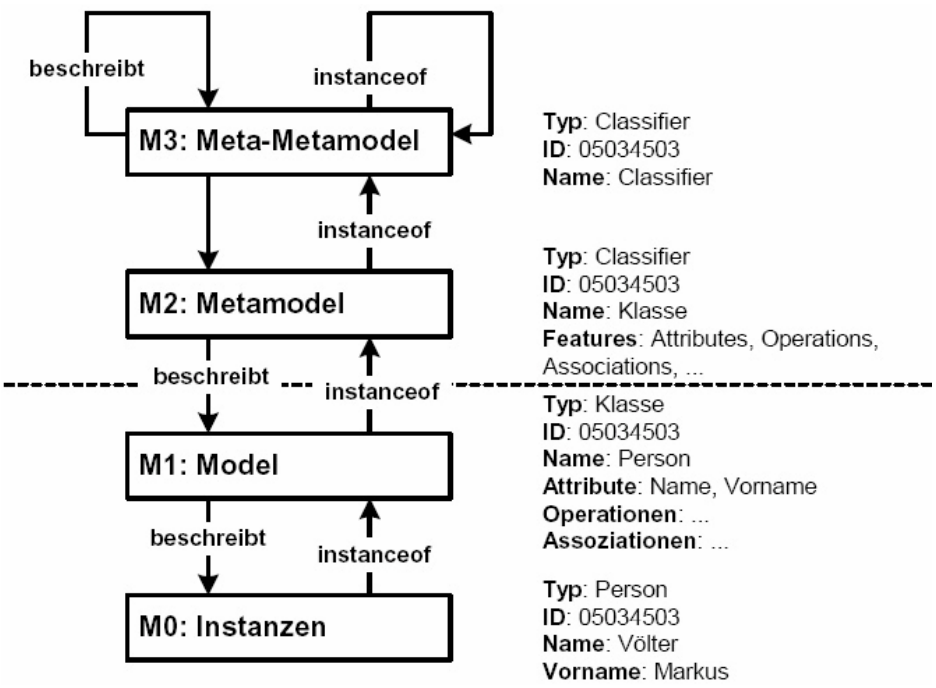

Abbildung 5 - MOF Architekturmodell
[Quelle: Stahl et al. 2007, S. 62]

Abbildung 5 illustriert das Architekturmodell, welches sich in vier Ebenen gliedert. Dabei befindet sich MOF selbst auf der obersten Ebene (M3), dem Meta-Meta-Model. Die darunter liegende Ebene (M2) spezifiziert das Meta-Modell, in welches sich unter anderem der UML Standard als Modellierungssprache einreiht. Die Modellebene (M1) besteht aus Klassen oder ihrer Repräsentation als UML Modell. Auf der untersten Ebene (M0), dem System, befinden sich konkret instanziierte Klassen, also die Objekte, welche direkt im System agieren.

Auf Grund seiner Komplexität teilt sich das MOF in zwei verschiedene Teile. Das *CMOF* (*Complete MOF*) bildet den kompletten Umfang der Spezifikation ab. Um für einfachere Sachverhalte nicht das komplette Modell nutzen zu müssen, gibt es zusätzlich noch eine vereinfachte Version, das *Essential MOF* (*EMOF*). Dabei ist EMOF stark an objekt-orientierte Sachverhalte angelehnt [vgl. Gruhn et al. 2006; S. 89 ff].

Zusätzlich zur Modellebenen-Repräsentation beinhaltet das MOF auch die Spezifikation eines auf XML basierten Austauschformats zur Darstellung von MOF-kompatiblen Datenstrukturen – das *XML Metadata Interchange* [*XMI*, vgl. 2.8.4.5].

2.7 OSGi Service Platform

Die *OSGi* (*Open Services Gateway Initiative*) Service Plattform ist eine Java-basierte, hardwareunabhängige Softwareplattform für die dynamische Nutzung und Verwaltung von Softwarekomponenten (*Bundles*) und Diensten (*Services*). Genauer gesagt handelt es sich bei der OSGi um eine Spezifikation durch die OSGi Alliance[4], welche seit 1999 entwickelt wird [vgl. Wütherich et al. 2008, S. 12].

Die Spezifikation erlaubt es Bundles und Services zur Laufzeit zu installieren/deinstallieren bzw. zu starten und zu stoppen, also ohne dass die Plattform gestoppt werden muss. Diese Flexibilität hat maßgeblich zum Erfolg von OSGi beigetragen.

Ziel dieser Plattform ist es, die Abhängigkeiten zwischen einzelnen Komponenten eines Softwaresystems zu reduzieren. Die Grundlage der Service Plattform bildet dabei das OSGi-Framework, welches die dynamische Verwaltung von Bundles und Services übernimmt.

2.7.1 Einsatzgebiete

Historisch betrachtet stammt OSGi aus dem Bereich der Gebäudeautomatisierung. Dort diente und dient es, in Form eines Internet Residential Gateways[5], als Basis für die Vernetzung von einzelnen Systemen untereinander und als Gateway zum Internet. Dadurch können hausinterne Anlagen über das Internet ferngesteuert werden. In seiner Funktion als Residential Gateway stellt OSGi die Schnittstelle von einem externen

[4] http://www.osgi.org – vor 2004: Open Services Gateway Initiative
[5] Internet Residential Gateway dient zur Vernetzung von internen Geräten mit dem Internet als externes Medium [vgl. O'Driscoll 2008, S, 175 ff.]

Netzwerk zu den inneren Netzwerken des Gebäudes dar [vgl. Wütherich et al. 2008, S.14]. Verschiedene Technologien, so zum Beispiel UPnP[6], beruhen auf OSGi. Weiterhin ist es möglich automatisierte Anlagen (Licht, Heizung, Sicherheitsanlagen etc.) zu aktualisieren, herunterzufahren oder zu installieren, ohne das gesamte Gebäudesystem abschalten oder neu starten zu müssen [vgl. Jung 2002, S. 454 ff.].

Diesen Vorteil haben auch andere Industriezweige erkannt, in denen sich OSGi immer mehr verbreitet. So hat es bereits in den Sektor der mobilen Endgeräte Einzug gehalten, indem es als Basis für Aktualisierungen von Smartphones genutzt wird. Auch im Bereich der Telematik findet eine OSGi-Plattform Anwendung. So verwenden zum Beispiel BMW und Volvo OSGi-Implementierungen für verschiedenen Produkte [vgl. OSGi 2009].

Besonders stark hat OSGi sich auch im Bereich der *Rich Client Plattformen* (*RCP*) und Desktop Anwendungen durchgesetzt. Nicht zuletzt das vollständige Aufsetzten der Eclipse IDE seit der Version 3.0 hat zu diesem Erfolg geführt. Mittlerweile wechseln immer mehr Softwaresysteme und Frameworks auf OSGi. So ist es ab Spring 2.5 möglich, Spring in OSGi-konformen Kontexten auszuführen [Zeitner et al. 2008, S.32 ff.]. Auch Anbieter von Application Servern ändern ihre Plattformen. Red Hat nutzt es derzeit innerhalb des JBoss Servers und auch Glassfish setzt auf OSGi [vgl. SUN 2008].

2.7.2 OSGi-Framework

Das OSGi-Framework bildet die Grundlage einer OSGi-Implementierung, indem es einen Container für die Verwaltung und Nutzung von Services und Bundles zur Verfügung stellt. Das Framework gliedert sich in mehrere Schichten [vgl. Abbildung 6].

Aufbauend auf der Hardware und dem Betriebssystem, bildet das *Execution Environment* den elementaren Ausgangspunkt für OSGi. Hierbei wurde das System so spezifiziert, dass OSGi auf unterschiedlichen Java-Plattformen zur Anwendung kommen kann. Ausführungsumgebungen sind die unterschiedlichen Java-Plattformen (*Java Runtime Environment, JRE*). Neben der Virtuellen Maschine der *Java Standard Edition* (*Java SE*) und der *Java Enterprise Edition* (*Java EE*) kann auch die *Java Micro Edition* (*Java ME*) innerhalb von eingebetteten Systemen genutzt werden.

[6] Universal Plug and Play – herstellerübergreifendes Kommunikations-Protokoll zur Steuerung von Geräten basierend auf IP [vgl. Schemberg et al. 2006, S.116]

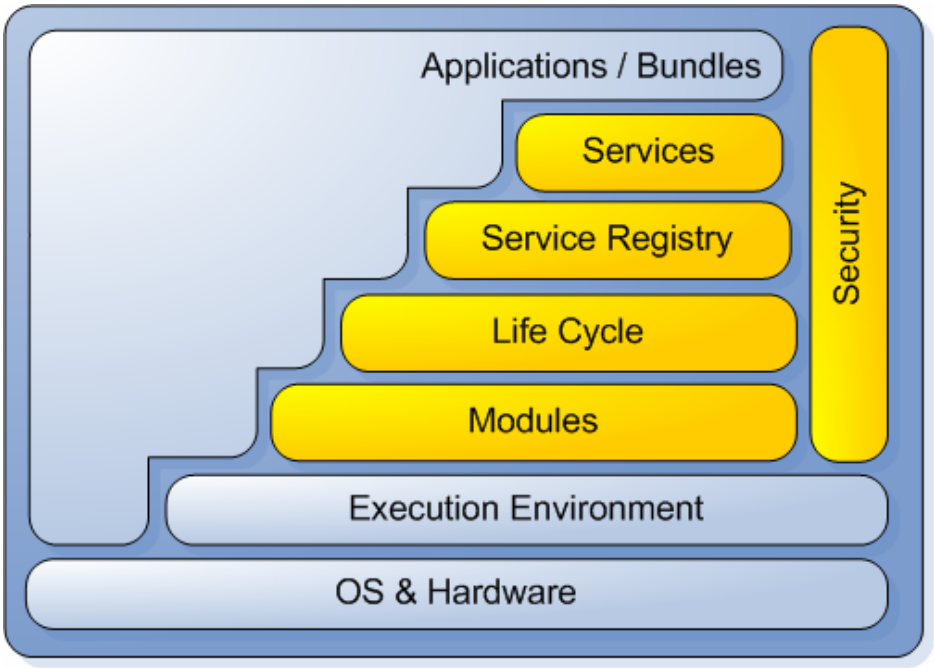

Abbildung 6 - Architektur der OSGi-Frameworks
[Quelle: vgl. Wütherich et al. 2008, S.19]

Die *Modulschicht*, als unterste Ebene im OSGi-Framework, stellt die Modularisierung der Plattform zur Verfügung. Integraler Bestandteil dieser Sicht sind die Bundles. Aufgabe der Modulschicht ist es, die Abhängigkeiten zwischen den einzelnen Bundles herzustellen, indem sie die in der Konfiguration des Bundles spezifizierten Importe und Exporte verwaltet und zur Verfügung stellt. Die Modulschicht sorgt also für die Verknüpfung der sonst lose gekoppelten Komponenten.

Das *Life-Cycle-Management* übernimmt die Verwaltung des Lebenszyklus eines Bundles. Im Gegensatz zu der statischen Verwaltung im Module-Layer, verwaltet der Life-Cycle-Layer die dynamischen Aspekte von Bundles. Diese Schicht erlaubt es Bundles zu installieren und zu starten oder auch wieder aus der Plattform zu entfernen. Hierbei bedient sich das Framework eines *Management Agents*. Da in der Spezifikation nur die Schnittstellen und Funktionen des Management Agents, nicht aber seine Implementierung definiert ist, können sie in verschiedenen Formen existieren. Je nachdem, wie er von einer konkreten Implementierung umgesetzt wird, kann er sowohl textueller als auch grafischer Natur sein. So gibt es reine Konsolen-Anwendungen [vgl. Abbildung 7], mit denen sich Bundles per Befehl installieren lassen, aber auch komplexe grafische Benutzerschnittstellen, welche den Bundle-Lebenszyklus verwalten können.

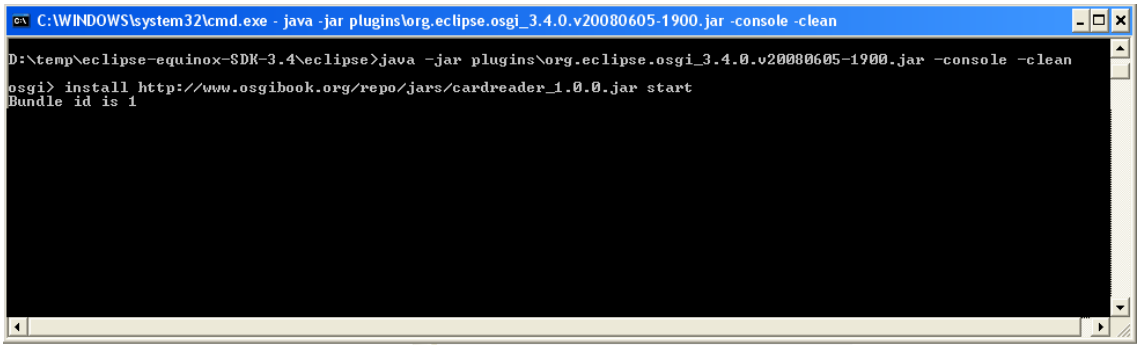

Abbildung 7 - einfacher OSGi-Management Agent

Neben der Modulschicht und der Lebenszyklus-Steuerung, ermöglicht der OSGi-Container noch das Bereitstellen von globalen Diensten. Diese Dienste werden an einer einheitlichen Registrierungsinstanz (*Service Registry*) zur Laufzeit an- bzw. abgemeldet und stehen systemweit allen Bundles zur Verfügung. Dabei entsprechen die Dienste einfachen Java-Objekten, welche durch die Service Registry unter einem eindeutigen Namen registriert werden. Da Dienste auch zur Laufzeit entfernt werden können, stellt das Framework drei Komponenten zur Verwaltung zur Verfügung – *Service Listener*, *Service Tracker* und *Declarative Services*. Service Listener reagieren auf Änderungen am Service (Service Event wird ausgelöst) und können andere Komponenten über diese Änderungen informieren. Service Tracker kapseln den Zugriff auf die Service Registry [vgl. Funke 2009, S.12ff.]. Es sind Java-Klassen, die den programmatischen Umgang mit dynamischen Diensten steuern. Mit Hilfe des Declarative Services ist es möglich Abhängigkeiten deklarativ zu beschreiben [vgl. Wütherich et al. 2008, S.114 ff.].

Um die Ausführungsrechte von Bundles steuern zu können, implementiert ein OSGi-Container eine vertikale *Sicherheitsschicht*. Über diesen Mechanismus ist es möglich gezielt einzelnen Bundles Rechte zuzuweisen bzw. ihnen diese zu entziehen.

2.7.3 Bundles

Bundles sind eigenständige Komponenten, welche innerhalb des OSGi-Frameworks lauffähig sind. Sie können, müssen aber nicht, ihre Funktionalität anderen Bundles mittels Schnittstellen zur Verfügung stellen. Der Zugriff auf diese Schnittstellen wird dabei vom OSGi-Framework zur Verfügung gestellt, wodurch lose Kopplung der einzelnen Bundles untereinander erlaubt und somit die Wiederverwendbarkeit der einzelnen Komponenten erhöht wird. Wie bereits beschrieben übernimmt das Framework die Registrierung bzw. das Abmelden der einzelnen Bundles. Damit dies bewerkstelligt werden kann, muss jedes Bundle über eine Activator-Klasse verfügen, welche das Interface `org.osgi.framework.BundleActivator` implementiert. Diese

muss die Methoden **`start(BundleContext)`** und **`stop(BundleContext)`** umsetzen, damit über den *Bundle-Context* eine Verbindung zwischen OSGi-Framework und Bundle hergestellt werden kann.

Die in Bundles enthaltenen Klassen und Funktionalitäten sind nicht automatisch für alle anderen Bundles zugängig. Um die Klassen und Ressourcen für andere zur Verfügung zu stellen, müssen die einzelnen Komponenten exportiert werden. Dadurch kann der Zugriff auf ein Bundle explizit gesteuert werden.

Technisch gesehen handelt es sich bei einem Bundle um nichts anderes als ein normales Java-Archiv (jar), welches um OSGi-spezifische Informationen erweitert wird. Diese Informationen werden innerhalb einer Manifest-Datei (Manifest.MF) deklariert, welche sich innerhalb des META-INF Ordners eines jeden Bundles befindet. In ihr werden alle Bundle-spezifischen Informationen wie zum Beispiel die Version, die Activator-Klasse oder die zur Verfügung gestellten Dienste deklariert.

Bundles können auch durch diese deklarativen Beschreibung konfiguriert werden. Mit Hilfe der Informationen ist es dem OSGi-Framework möglich die Verknüpfungen (Import/Export) zwischen den einzelnen Bundles herzustellen. Listing 1 stellt beispielhaft eine Manifest Datei dar. In ihr können die Manifest-Versionen festgelegt werden, um Konflikte zwischen unterschiedlichen Generationen zu vermeiden.

```
1   Manifest-Version: 1.0
2   Bundle-ManifestVersion: 2
3   Bundle-Name: Extension Plug-in
4   Bundle-SymbolicName:
5   org.mftech.diagram.uml.class.diagram.extension;singleton:=true
6   Bundle-Version: 1.0.0
7   Bundle-Activator: classdiagram.diagram.extension.Activator
8   Require-Bundle: org.mftech.diagram.uml.class.diagram;bundle-
9   version="1.0.0",
10  Import-Package: org.eclipse.emf.transaction,
11                  org.eclipse.gmf.runtime.notation
```

Listing 1 - OSGi-Manifest

Neben dieser Information wird auch der Name festgelegt, unter dem das Bundle bekannt gemacht wird. Eine der wichtigsten Einstellungen ist die Angabe des Activators (Zeile 7). Diese Klasse wird vom OSGi-Container aufgerufen und für die Steuerung des Lebenszyklus eines Bundles genutzt. Zusätzlich können noch Angaben über die Abhängigkeiten zu anderen Bundles (Zeile 8) oder benötigte Pakete erfolgen (Zeilen 10-11). Die Konfigurationsmöglichkeiten eines Bundles sind weitaus größer, als in dem Listing dargestellt [vgl. Seeberger 2009].

Wie in Kapitel 2.7.1 beschrieben, verwaltet der OSGi-Container den Lebenszyklus eines Bundles. Prinzipiell kann ein Bundle zur Laufzeit installiert oder deinstalliert werden. Voraussetzung für eine erfolgreiche Installation ist eine korrekte Manifest-Datei, also eine gültige Deklaration der Bundle-Beschreibungen. Ist die Installation abgeschlossen, so wird das Bundle im Container verankert und kann genutzt werden. Nach der Installation versucht der Container alle im Manifest deklarierten Abhängigkeiten aufzulösen. Gelingt dies, so wird das Bundle in den Status *resolved* gesetzt [vgl. Abbildung 8]. Von diesem Status aus kann ein Bundle gestartet werden.

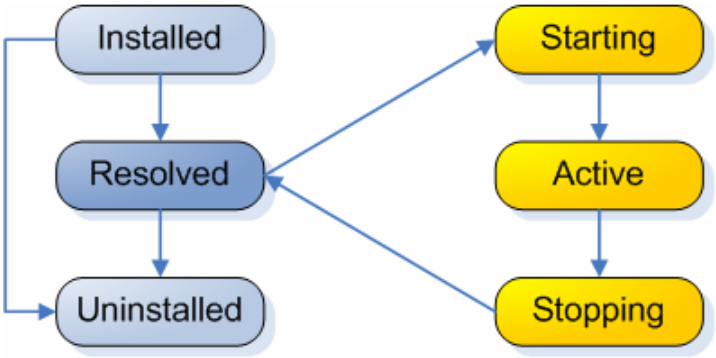

Abbildung 8 - Bundle Lebenszyklus
[Quelle: vgl. Wütherich et al. 2008, S.23]

Hierzu wird der in der Manifest-Datei definierte Activator aufgerufen, genauer gesagt dessen Start-Methode ausgeführt. Wird dieser erfolgreich beendet, so ist das Bundle aktiv und kann genutzt werden. Die Activator-Schnittstelle wird ebenfalls genutzt, um ein Bundle wieder zu stoppen, wodurch es automatisch in den Zustand *resolved* zurück versetzt wird.

2.7.4 Services

Services dienen neben Bundles ebenfalls zur Trennung der einzelnen Komponenten. Sie sind Dienste, welche systemweit allen Bundles zur Verfügung gestellt werden können. Prinzipiell handelt es sich um Java-Objekte, welche ihre Methoden über Interfaces bereitstellen. Somit ist es möglich, dass verschiedene Implementierungen für die Bundles transparent implementiert werden können. Dienste können an einer globalen Systemeinheit (*Service Registry*) registriert und abgefragt werden. Jedes Bundle erhält somit Zugriff auf Objekte, welche von anderen Bundles oder der OSGi Service Platform global zur Verfügung gestellt werden. Durch diese schnittstellenbasierte Kommunikation wird eine lose Kopplung der einzelnen Komponenten erreicht [vgl. Wütherich et al. 2008, S.23].

Neben nutzerimplementierten Services, stellt das Framework Basisdienste (*OSGi Standard Services*) zur Verfügung, welche in den Bundles genutzt werden können. Ziel dieser Dienste ist es, eine einheitliche Grundlage für die Nutzung häufig verwendeter Funktionen innerhalb des Frameworks zur Verfügung zu stellen. So können Bundles zum Beispiel einen LOG-Service für einheitliche Ausgaben nutzen oder über den Meta-Type *Service* Metainformationen über ihre Dienste zur Verfügung stellen, die zur Laufzeit von anderen Bundles angefragt werden können. Eine Übersicht über die in den unterschiedlichen Spezifikationen angebotenen Dienste findet sich in [Wütherich et al. 2008, S.25 ff.].

2.7.5 OSGi-Implementierungen

Die OSGi Alliance definiert für den OSGi-Standard das nötige API (*Application Programming Interface*) und die Testfälle für eine konkrete Implementierung des Standards. Zusätzlich stellt sie eine Referenzimplementierung zur Verfügung, welche aber nicht für den Produktiveinsatz, sondern als Vorlage für eigene Implementierungen gedacht ist. Basierend auf diesen Spezifikationen, wurden verschiedene kommerzielle oder freie Implementierungen erstellt. Eine dieser Implementierungen ist *Equinox* der Eclipse Foundation, welche unter anderem als Grundlage für die Eclipse IDE dient. Aber auch die Apache Foundation bietet unter dem Namen *Apache Felix* eine OSGi-Implementierung an [vgl. Haiges 2006, S.35]. In der Regel bestehen die OSGi-Implementierungen aus dem OSGi-Framework und zusätzlichen Services.

Die unten aufgeführte Tabelle gibt einen groben Überblick über derzeitige Open-Source OSGi-Implementierungen.

Name	**Hersteller**
Apache Felix	Apache Software Foundation
Equinox	Eclipse Foundation
Knopflerfish	MakeWave[7]
Prosyst mBedded Server Equinox Edition	Prosyst[8]
Oscar OSGi	OW2 Consortium[9]

Tabelle 2 - Open-Source OSGi-Implementierungen

Da OSGi grundlegend eine komponentenbasierte Architektur nutzt, werden Abhängigkeiten zwischen einzelnen Systemteilen auf ein Minimum reduziert.

[7] http://www.makewave.com/site.en/
[8] http://www.prosyst.com/products/osgi_se_equi_ed.html
[9] http://forge.ow2.org/projects/oscar/

Zusätzlich wird ein hohes Maß an Wiederverwendbarkeit der Komponenten realisiert. Weil OSGi es erlaubt Bundles zur Laufzeit zu installieren und zu deinstallieren bzw. zu starten und zu stoppen, können einzelne Systemteile flexibel ausgetauscht werden, ohne den Gesamtablauf des Systems empfindlich zu stören. Durch diese Modularisierung kann auch die Entwicklung wesentlich besser aufgeteilt werden. Einzelne Teams können sich auf die Entwicklung von Kernfunktionalitäten konzentrieren, aus denen dann „höherwertigere“ Bundles zusammengebaut werden können. Durch die Nutzung von Services bietet die Plattform bereits erprobte Komponenten, die genutzt werden können ohne oft verwendete Teile, wie zum Beispiel Logging-Funktionalitäten, neu implementieren zu müssen.

Weil Bundles auch in verschiedenen Versionen ausgeliefert werden können, ist es möglich auch Systemaktualisierungen effektiv durchzuführen, was eine hohe Wartbarkeit, Skalierbarkeit und Flexibilität des auf OSGi-basierten Systems erlaubt.

OSGi ist an die erhöhte Ressourcennutzung von Java gebunden, da es auf eine *Java Virtual Machine* angewiesen ist. Aus diesem Grund sind OSGi-Implementierungen auch in der Programmiersprache Java realisiert.

2.7.6 OSGi und SOA

In der Fachwelt tritt zuweilen die Behauptung auf, OSGi sei die Umsetzung einer s*erviceorientierten Architektur* (*SOA*) innerhalb der Virtuellen Maschine. Oftmals wird diese These allerdings stark diskutiert [vgl. Voelter Blog 2007].

Der Begriff einer serviceorientierten Architektur hat sich leider nicht in einer einheitlichen Definition durchgesetzt. Im Allgemeinen wird unter einer *SOA* aber eine auf Diensten basierende Architektur verstanden, in der die Dienste ihre Funktionen durch definierte Schnittstellen anderen Diensten zur Verfügung stellen. [Melzer et al. 2008] liefert hierzu folgende Definition:

„Serviceorientierte Architekturen, kurz SOA, sind das abstrakte Konzept einer Software-Architektur, in deren Zentrum das Anbieten, Suchen und Nutzen von Diensten über ein Netzwerk steht.“

Anwendungen in einer SOA werden durch die Orchestrierung von Diensten zusammengebaut und Dienste können ebenfalls aus solchen bestehen. Ein wichtiger Aspekt einer SOA ist die lose Bindung zwischen den einzelnen Diensten, da die Kommunikation ausschließlich über die Schnittstellen erfolgt. Hierbei können in der Praxis verschiedene Technologien (Webservices, CORBA, REST etc.) genutzt werden,

wobei sich zur Zeit am stärksten Webservices für die Implementierung einer SOA durchgesetzt haben.

Das entscheidende Argument gegen die Behauptung, dass OSGi SOA sei, ist in dem Unterschied zwischen einer Software-Komponente und einem Service in Bezug auf SOA zu finden. Der Begriff der Komponente wurde auf der *European Conference of Object-Oriented Programming* (*ECOOP*) im Jahr 1996 wie folgt definiert.

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties" [Szyperski et al. 2002, S.41].

Aufbauend auf dieser Definition versteht sich eine Komponente als Software, welche über Schnittstellenspezifikationen ihre Funktionen zur Verfügung stellt. Dabei kann sie unabhängig von anderen Komponenten verwendet werden und kann in diesem Zusammenhang von Anwendungen Dritter genutzt werden. Ferner ist die konkrete Implementierung einer Komponente nicht von Bedeutung für die sie nutzende Instanz, da über die Schnittstellen auf die Komponente zugegriffen wird. Dies ermöglicht eine hohe Wiederverwendbarkeit und Austauschbarkeit von Komponenten.

Alle diese Eigenschaften vereint auch ein Service im Kontext von SOA auf sich. Ebenso wie bei der Software-Komponente, spielt die konkrete Implementierung eines Dienstes keine Rolle, solange die Vorgaben der Schnittstelle erfüllt sind. Daneben werden Services auch als wiederverwendbare Konstrukte genutzt, um eine gesamte Anwendung zu erstellen, deren einzelne Elemente wiederverwendbar und austauschbar sind. Im Gegensatz zur Komponente, ist ein Service aber nicht kontextabhängig. Software-Komponenten sind programmiersprachenabhängig und aus diesem Grund wesentlich stärker technologieorientiert. Services hingegen nutzen ein interoperables Austauschformat, wie zum Beispiel XML, weshalb die Programmiersprache für die Implementierung der Service-Schnittstelle nicht von Bedeutung ist. Diese Aussage wird durch folgende Definition gestützt:

„Unter einer SOA versteht man eine Systemarchitektur, die vielfältige, verschiedene und eventuell inkompatible Methoden oder Applikationen als wieder verwendbare und offen zugreifbare Dienste repräsentiert und dadurch eine plattform- und sprachunabhängige Nutzung und Wiederverwendung ermöglicht" [vgl. Melzer et al. 2008, S.13].

OSGi, basierend auf dem Komponentenmodell, bietet mit den Bundles und Services ebenfalls schnittstellenorientierte Softwareteile, also Komponenten ähnlich einer SOA an. Durch die explizite Bindung an Java können die einzelnen Komponenten von OSGi

allerdings nicht losgelöst von der Virtuellen Maschine genutzt werden. Weder ein Bundle noch ein OSGi-Service können also als Dienst bezüglich einer SOA betrachtet werden. Beide Begriffe und Definition können zwar als stark verwandt, aber nicht als identisch angesehen werden. Somit stellt OSGi auch keine Implementierung einer serviceorientierten Architektur dar.

2.8 Eclipse

Basis der Implementierung dieser Arbeit bildet das Eclipse Framework. Allseits bekannt ist, dass Eclipse eine integrierte Entwicklungsumgebung (*Integrated Development Environment, IDE*) für die Entwicklung unterschiedlichster Software ist. Doch Eclipse ist weitaus mehr. Die folgenden Kaptitel sollen ein Bewusstsein für die Vielfältigkeit von Eclipse schaffen.

Eclipse basierte ursprünglich auf *Visual Age for Java* von IBM. 2001 wurden die Quellen offen gelegt, und Eclipse zu einen Open-Source Projekt [vgl. Künneth 2008, S.13]. Daraus entwickelte sich eine leistungsfähige IDE, welche sich durch ein flexibles Plugin-Konzept modular erweitern lässt. Durch dieses Konzept ließen sich verschiedenste Erweiterungen für Eclipse erstellen. So ist Eclipse nicht nur eine IDE für die Entwicklung von Java, sondern lässt sich auch für viele andere Sprachen nutzen. Es bietet zum Beispiel das CDT-Projekt[10], eine Adaptierung für die Entwicklung von C/C++ Anwendungen.

Eclipse ist aber mehr als nur eine IDE. Es ist auch eine Open-Source Community, welche sich mit der Erstellung von Open-Source Anwendungen und Frameworks im Bereich der Softwareentwicklung beschäftigt. Mittlerweile besteht das *Eclipse Project* aus 11 Hauptprojekten mit insgesamt 49 offiziellen Unterprojekten und weiteren 48 Projekten im Inkubations-Status[11] [vgl. Eclipse 2009a]. Das aktuelle Eclipse Release *Ganymede* vereinigt 23 aufeinander abgestimmte Projekte, welche Frameworks für unterschiedlichste Sprachen und Technologien bereitstellen [vgl. Teufel 2008, S. 13]. Jedes dieser Projekte besteht wiederum aus einer Vielzahl von Subprojekten, was Eclipse zu einer äußerst flexiblen Plattform macht. Tabelle 3 gibt einen Überblick über gängige Eclipse Projekte und ihre Anwendungsgebieten.

Seit 2004 übernimmt die Eclipse Foundation die Organisation und Verwaltung der vielen Projekte. Sie unterstützt die einzelnen Projekte und Team-Leiter, koordiniert die Aufgaben und organisierte Veranstaltungen.

---

[10] http://www.eclipse.org/cdt/
[11] Projekte im Incubator werden zu vollständigen Open-Source Projekten entwickelt.

Name	Anwendungsgebiet
BIRT	Business Intelligence and Reporting Tools; Erstellung von Reports; http://www.eclipse.org/birt
Buckminster	Framework zum automatischen Bauen und Deployen von Projekten; http://www.eclipse.org/buckminster
CDT	C/C++ Development Tooling; IDE für C/C++; http://www.eclipse.org/cdt
DLTK	Dynamic Languages Toolkit; Toolkit für dynamische Sprachen wie Tcl, Ruby oder Python; http://www.eclipse.org/dltk
DSDP DD	Device Software Development Platform – Debugging Devide; Debugging Framework von Eclipse; http://www.eclipse.org/dsdp/dd/
DSDP TM	Device Software Development Platform – Target Management; Remote Verwaltung von Objekten; http://www.eclipse.org/dsdp/tm/
DTP	Data Tools Platform; http://www.eclipse.org/datatools/
ECF	Eclipse Communication Framework; http://www.eclipse.org/ecf/
Eclipse Project	Hauptprojekt von Eclipse; http://www.eclipse.org/eclipse/
EMF	Eclipse Modeling Framework; Basis für modellgetriebene Softwareentwicklung; http://www.eclipse.org/emf/
EMFT	Eclipse Modeling Framework Technologies; bietet Tools zum Verarbeiten von Modellen; http://www.eclipse.org/modeling/emft
EPP	Eclipse Package Projekt; http://www.eclipse.org/epp
GEF	Graphical Editor Framework; Entwicklung von grafischen Editoren http://www.eclipse.org/gef/
GMF	Graphical Modeling Framework; Entwicklung von grafischen Editoren basieren auf EMF; http://www.eclipse.org/gmf/
MDT	Model Development Tools; u.a. Support für BPMN, UML oder OCL; http://www.eclipse.org/modeling/mdt/
M2M	Unterstützt Model-to-Model Transformationen; http://www.eclipse.org/m2m/
M2T	unterstützt Model-to-Text Transformationen; http://www.eclipse.org/modeling/m2t/
MyLyn	Zur Verwaltung von Aufgaben in einem Software Projekt; http://www.eclipse.org/mylyn/
RAP	Rich Ajax Platform, Implementierung von RCP webbasiert. http://www.eclipse.org/rap/
STP	SOA Tools Platform Projekt; unterstützt SOA-orientiertes Entwickeln; http://www.eclipse.org/stp/
Subversive	Versionsmanagement auf Basis von Subversion; http://www.eclipse.org/subversive/
TPTP	Testumgebung für Eclipse; http://www.eclipse.org/tptp/
WTP	Web Tools Plattform; Unterstützung bei der Entwicklung von webbasierten Anwendungen; http://www.eclipse.org/webtools/

Tabelle 3 - Überblick über die Eclipse Projekte im Ganymede Release [Quelle: vgl. Teufel 2008, S.14]

2.8.1 Die Plattform – Equinox

Eclipse ist eine dynamisch erweiterbare Anwendung. Grundlage dieses dynamischen Konzeptes bilden Plugins, welche sich einfach in die Plattform integrieren und wieder aus ihr entfernen lassen. Dabei nutzte Eclipse anfangs ein proprietäres System, welches einige Probleme aufzeigte [vgl. Daum 2007, S.11]. Mit dem Release von Eclipse 3.0 wurde auch das Plugin-System komplett neu überarbeitet. Die Plattform von Eclipse wurde in Form einer OSGi-Implementierung neu gestaltet und mit dem Namen *Equinox* versehen. Im Oktober 2005 spendete die Firma IBM, welche sowohl Mitglied in der Eclipse Foundation, als auch der OSGi Alliance ist, eine umfangreiche Codebasis basierend auf dem OSGi Release 4 [vgl. Haiges 2006, S.35]. Dies machte Equinox zu einer stark standard-konformen OSGi-Implementierung.

Zwar basiert Equinox auf Eclipse, kann aber komplett losgelöst von diesem als eigenständige OSGi-Implementierung genutzt werden. Dabei ist es mit einer Größe von 4,7 Megabytes (Release 3.4[12]) sehr schlank gehalten.

Aufbauend auf dieser Plattform, werden alle Projekte innerhalb von Eclipse realisiert. Neben den Standardservices bietet es auch die im OSGi-Container genutzten Bundles. Da OSGi erst ab Eclipse-Release 3.0 genutzt wurde, und vorher schon der Name *Plugin* für die austauschbaren Komponenten innerhalb von Eclipse Verwendung fand, ist diese Namensgebung beibehalten worden. Ein Plugin innerhalb von Eclipse ist also nichts anderes als ein OSGi-Bundle. Basierend auf dieser Design-Entscheidung, stehen der Eclipse-Plattform alle Vorteile eines OSGi-Containers zur Verfügung. Plugins können dynamisch zur Laufzeit installiert und entfernt werden. Globale Services können einheitlich genutzt werden.

Für die Verwaltung des Containers setzt Equinox eine einfache Konsole als Management Agent ein, welche mit wenigen Befehlen auskommt [vgl. Abbildung 7].

Wie aus Abbildung 9 ersichtlich, basiert jede Entwicklung von Eclipse auf Equinox als unterste Schicht im Eclipse-Architektur-Modell. Das bedeutet, dass jede auf dieser Plattform basierende Software auch ihre Vorteile, zum Beispiel die des Plugin-Konzepts, nutzen kann.

[12] http://download.eclipse.org/eclipse/equinox/

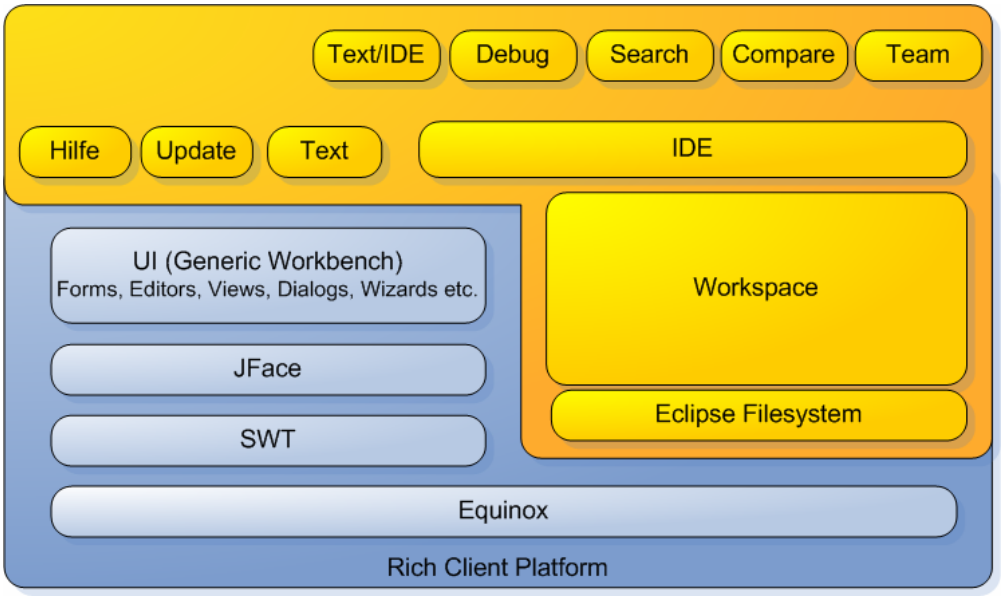

Abbildung 9 - Architektur von Eclipse
[Quelle: vgl. Daum 2007, S.12]

2.8.2 RCP – Rich Client Platform

Mit Eclipse 3.0 wurde neben der OSGi-basierten Plattform auch das Konzept der *Rich Client Platform* eingeführt. War es vorher nur möglich über Plugins die Eclipse IDE zu erweitern und zu verändern, so können nun eigenständige Anwendungen unabhängig von der IDE erstellt werden. Die Anwendungen bauen einzig auf Equinox auf und können beliebig gestaltet werden. Genau genommen ist die Eclipse Entwicklungsumgebung auch nichts anderes als eine Anwendung, welche auf der Rich Client Plattform aufbaut. Dabei stehen jeder Rich Client Anwendung die Vorteile zur Verfügung, wie sie auch aus der Eclipse IDE bekannt sind, was unter anderem eine flexible UI-Gestaltung, vordefinierte Wizards und einen ausgefeilten Update-Mechanismus beinhaltet [vgl. Daum 2007, S.11 ff.].

2.8.3 PDE – Plugin Development Environment

Eines der Projekte, welches auf der Eclipse Plattform Equinox aufbaut, ist die Plugin Entwicklungsumgebung (*Plugin Development Environment, PDE*). PDE ist eins von fünf Sub-Projekten des Eclipse-Projektes und stellt nützliche Tools und User-Interfaces für die Bearbeitung von Plugins zur Verfügung und ist somit essentieller Bestandteil einer jeden Eclipse Entwicklung. Da Plugins nahezu identisch mit OSGi-Bundles sind [vgl. 2.8.1 und 2.8.3.1], kann PDE auch außerhalb von Eclipse für die Entwicklung von OSGi-Anwendungen genutzt werden.

Die Plugin Entwicklungsumgebung unterteilt sich in drei Teile – *PDE Build*, *PDE API Tools* und *PDE UI*. Die ersten beiden Projekte unterstützen den Entwickler beim Build-Prozess und bieten hilfreiche Funktionen, um beispielsweise die Versionen der Plugins zu verwalten. Das PDE UI wiederum bietet eine Schaltzentrale für die komplette Verwaltung der Plugin-Entwicklung [vgl. Abbildung 10].

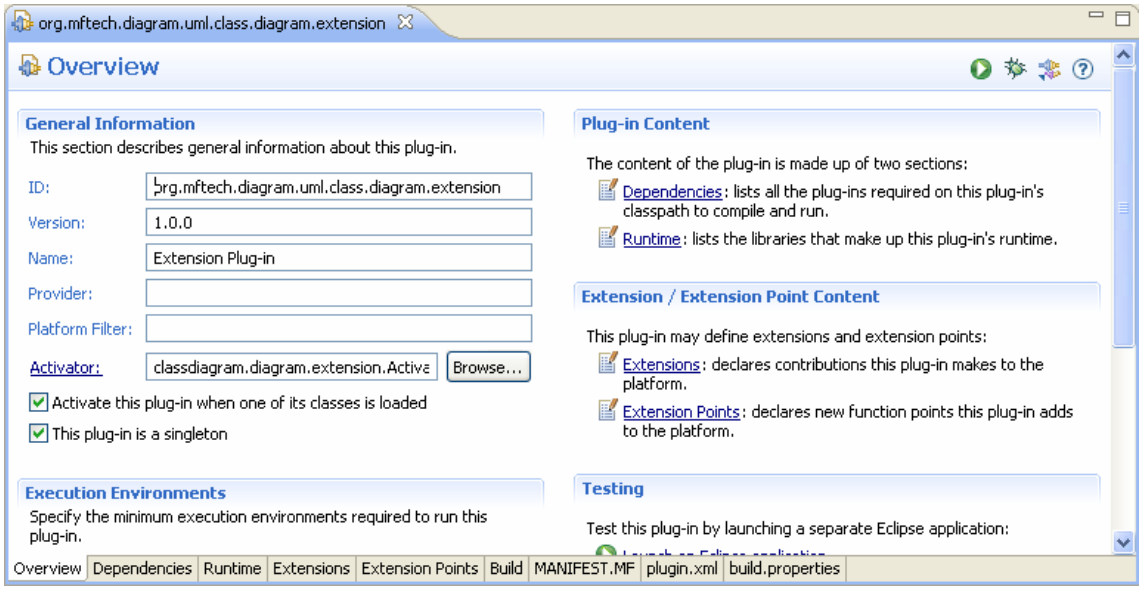

Abbildung 10 - PDE User Interface

Hierbei stellt es die Informationen aus den unterschiedlichen Konfigurationsdateien eines Plugins aufbereitet dar und erlaubt auf einfache Weise deren Manipulation. Dazu bietet es verschiedene Editoren an, mit denen auch direkt die Dateien im Text-Format bearbeitet werden können. Im Folgenden sollen die für die Arbeit notwendigen Begriffe im Rahmen der Plugin-Entwicklung beschrieben und erläutert werden.

2.8.3.1 Plugins

Wie bereits angedeutet, handelt es sich bei Plugins um OSGi-Bundles, welche auf der Equinox Plattform aufbauen. Wie Bundles sind Plugins ebenfalls Java-Archive, die um Meta-Informationen (Manifest.MF) erweitert werden. Zusätzlich zu dem Manifest können weitere Meta-Informationen für das Plugin innerhalb einer speziellen Datei, der plugin.xml, eingestellt werden. Sie beschreibt deklarativ, welche Erweiterungen [Extensions, vgl. 2.8.3.2] das Plugin nutzt und welche Erweiterungen [Extension Points, vgl. 2.8.3.3] sie anderen Plugins zur Verfügung stellt. Im Gegensatz zum Manifest.MF ist das Vorhandensein der plugin.xml optional.

Plugins können, wie Bundles, eine Activator-Klasse besitzen, welche die Methoden zur Steuerung des Lebenszyklus des Plugins bereitstellt. In der Regel erbt diese von der

abstrakten Basisklasse **`Plugin`**, welche das Interface **`org.osgi.framework.BundleActivtor`** implementiert und somit eine Implementierung der OSGi-Spezifikation ist.

Wie bereits beschrieben bauen sämtliche Anwendungen in Eclipse auf Plugins auf. Auch die Eclipse IDE besteht im Grunde genommen aus der Equinox Plattform und einer großen Anzahl von Plugins, welche die Funktionen der Entwicklungsumgebung bereitstellen. Dabei kann eine Anwendung beliebig erweitert werden, indem zusätzliche Plugins hinzugefügt werden. Diese Erweiterungen werden mit Hilfe von Extensions und Extension Points realisiert.

2.8.3.2 Extensions

Die Eclipse Plattform bietet bereits eine Vielzahl von vorgefertigten Elementen, um eine Anwendung erstellen zu können. So hat man bei der Erstellung seiner eigenen Software eine umfangreiche Auswahl aus Editoren, Sichten (*Views*), Wizards und anderen Dialog-Elementen zur Verfügung. Oftmals eignen sich diese aber nicht in ihrer eigentlichen Form, sondern müssen um einige Punkte erweitert werden. Möchte man beispielsweise einen eigenen Editor in die IDE einbringen, so kann man vorhandene Extension Points benutzen, um den Editor in das System *einzuklinken*. Der selbst implementierte Editor wird als Erweiterung mit dem Erweiterungspunkt verknüpft.

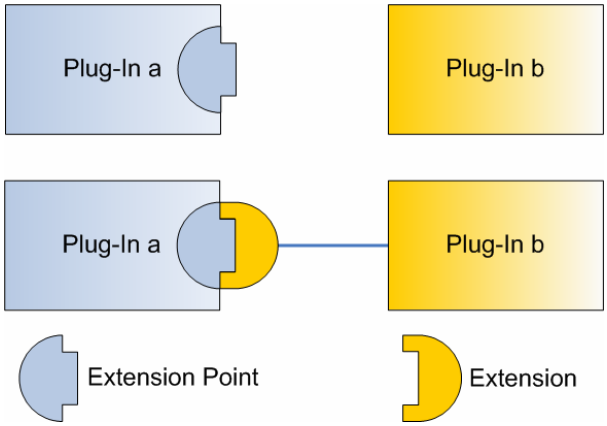

Abbildung 11 - Plugin Mechanismus

In diesem Zusammenhang lässt sich die Analogie zu einer Steckdose und einem Stecker verwenden. Der Editor entspricht dem Stecker (*Extension*), welcher mit einer passenden Steckdose (*Extension Point)* verbunden werden kann [vgl. Abbildung 11].

Im Ergebnis kann ein neuer Editor in das System eingebaut werden, welcher alle Eigenschaften von bestehen Editoren nutzt. So ist es in Eclipse beispielsweise möglich bestimmte Dateiendungen mit einem Editor so zu verbinden, dass sich bei Auswahl der

Datei der richtige Editor öffnet. Diese und andere Informationen können deklarativ in der Extension beschrieben werden, sodass dieses Verhalten nicht eigenständig programmiert werden muss.

Dabei ist zu beachten, dass Erweiterungen nicht allein auf UI-Elemente beschränkt sind und in jeglicher Form innerhalb einer Software genutzt werden können.

2.8.3.3 Extension Points

Jedes Plugin besitzt die Möglichkeit eigene Erweiterungspunkte zu definieren. Erweiterungspunkte können genutzt werden, um das Verhalten eines Plugins zu verändern [vgl. Widder 2004, S. 77]. Dieses Prinzip liefert also die Möglichkeit das eigene Plugin so generisch zu halten, dass andere Entwickler flexible Erweiterungen vornehmen können. Dabei stellt sich aber die Frage, worin sich der Extension Point Mechanismus von der einfachen Vererbung unterscheidet. Auch mit Vererbung kann das Verhalten von Klassen modifiziert werden, indem eine Unterklasse die entsprechenden Methoden überschreibt. Abbildung 12 stellt ein Szenario dar, indem *a* und *b* zwei Plugins sind, die jeweils die *Klassen A* und *B* enthalten. Dabei soll *Plugin b* das *Plugin a* erweitern. Im Fall der Vererbung kann *B* als Komponente alle Informationen des *Plugins a* nutzen. Soll nun *Klasse B* (z.B. ein neuer Editor) genutzt werden, so muss das *Plugin b* die aktive Komponente sein. Die bedingt, dass *a* in diesem Fall sich nicht aktiv verhält.

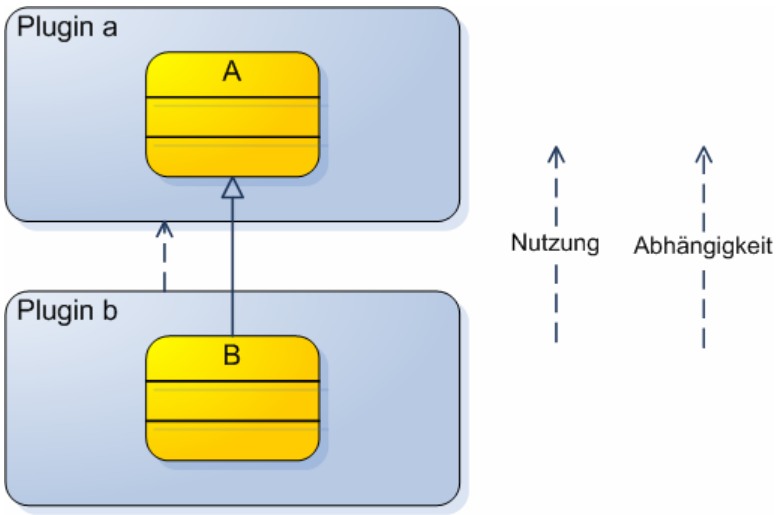

Abbildung 12 - Nutzungs- und Abhängigkeitsrichtung bei Vererbung [Quelle: vgl. Hennig et al 2008, S.19]

Soll nun aber *a* die aktive Komponente sein, die über zusätzliche Informationen von *b* erweitert wird, so ist dies mit dem Vererbungsmodell nicht zu realisieren. *Klasse B* kann

alle Informationen der Basisklasse nutzen. Eine umgekehrte Nutzung ist allerdings nicht möglich. Ebenfalls entsteht durch das Vererbungskonzept eine starke Abhängigkeit zwischen den beiden Komponenten. Diese Abhängigkeit kann zwar durch die Nutzung von Interfaces reduziert werden, bestehen bleibt allerdings das Problem, dass *Plugin a* nicht als eigenständige Komponente dynamisch durch die Informationen ergänzt werden kann.

Dieses Problem kann durch die Nutzung von Extension Points gelöst werden. Hierbei kann *a* als eigenständige Komponente laufen, zum Beispiel als lauffähiger Editor. Bei Bedarf kann, sogar zur Laufzeit, das Verhalten von *a* geändert werden, indem es die Methoden der *Klasse B* benutzt. Dabei dreht sich also die Nutzungsrichtung um [vgl. Abbildung 13].

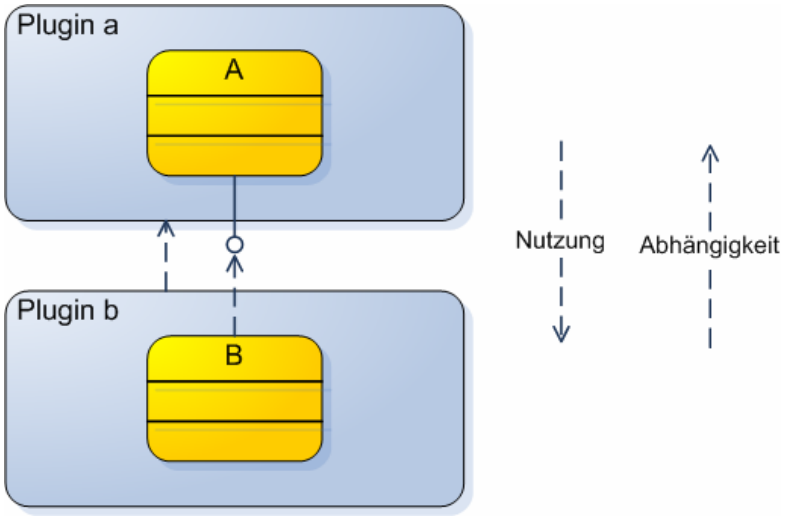

Abbildung 13 - Nutzungs- und Abhängigkeitsrichtung bei Extension Points
[Quelle: vgl. Hennig et al. 2008, S.19]

Ermöglicht wird dies, indem *Plugin a* die Schnittstellen definiert, an denen es sich erweitern lässt. Diese Schnittstellen (*Extension Point*) wird deklarativ in der plugin.xml beschrieben. Sie verweist auf ihre Definition, welche in wird in Form einer XML-Schema Beschreibung erstellt und im Plugin hinterlegt wird. *Plugin b* kann diese benutzen, um sich am *Plugin a* anzumelden [vgl. Hennig et al 2008, S. 19 ff.]. Zur Laufzeit kann *Plugin a* eine globale Registrierung (*Extension-Registry*) abfragen, ob sich Erweiterungen angemeldet haben. Die Extension-Registry wird innerhalb von Equinox als OSGi-Service zur Verfügung gestellt. Da die Beschreibung von Extension Point und Extension rein deklarativ erfolgen und die genutzten Methoden anhand von Schnittstellendefinitionen vereinbart werden, sind die Komponenten a und b vollständig von einander entkoppelt, was dem der OSGi zugrunde liegenden Komponentenmodell entspricht.

2.8.3.4 Fragmente

Neben der Deklaration der Abhängigkeit zu anderen Plugins und dem Extension Point Mechanismus, gibt es noch eine dritte Möglichkeit um Plugins zu verändern. Fragmente entstammen dem OSGi-Standard, wo sie auch als OSGi-Fragmente bezeichnet werden. Sie sind ähnlich einem Plugin aufgebaut. Allerdings fehlt ihnen eine Activator-Klasse. Sie sind also nicht eigenständig lauffähig. Vielmehr erweitern Fragmente bestehende Plugins. Sie sind deshalb immer auf ein Host-Plugin angewiesen, in dessen Kontext alle sich innerhalb eines Fragments befindlichen Ressourcen existieren. Wird ein Fragment mit einem Host-Plugins verbunden, so verhält es sich, als wären alle Inhalte bereits in dem Plugin gewesen. Beide werden zur Laufzeit zu einer Einheit verknüpft [vgl. Sippel 2008, S. 257]. Dadurch können beliebige Inhalte hinzugefügt werden, ohne das Plugin zu verändern. Eine verbreitete Anwendung von Fragmenten besteht in der Verteilung von Sprachpaketen. Informationen für bestimmte Sprachen können im Nachhinein ergänzt werden, falls zum Beispiel bei Auslieferung die Übersetzung in einer bestimmten Sprache noch nicht vorlag. Der Inhalt von Fragmenten ist allerdings nicht allein auf Sprach-Dateien begrenzt, sondern erstreckt sich auf alle Inhalte, die auch in einem regulären Plugin zur Verfügung stehen. Innerhalb von Eclipse besitzen Fragmente äquivalent zur plugin.xml eine Konfigurationsdatei mit der Abhängigkeiten geregelt werden können. Zur Unterscheidung nennt sich diese Datei *fragment.xml*.

2.8.3.5 Ansichten, Editoren und Perspektiven

In jeder *RCP* (*Rich Client Platform*), also auch in der Eclipse IDE, spielen drei Konzepte eine wichtige Rolle – *Ansichten*, *Editoren* und *Perspektiven*. Ansichten (*Views*) und Editoren sind Bestandteile einer jeden *Workbench*-Seite, also dem Fenster der RCP. Editoren und Ansichten ähneln sich stark. Beide können genutzt werden, um Daten zu manipulieren. Dabei kommen aber Editoren verstärkt zum Einsatz wenn Daten direkt editiert werden sollen, beispielsweise um Texte, Quellcodes, oder grafische Modelle zu bearbeiten. Views hingegen dienen meist vorrangig zur Anzeige von Daten, zum Beispiel um die Paket-Struktur eines Projektes darzustellen. Sie können aber auch Daten manipulieren. Perspektiven bieten eine vordefinierte Anordnung von Editoren und Views, die unter einem bestimmten Namen hinterlegt ist. Wird in eine Perspektive gewechselt, so werden genau die Views und Editoren angezeigt, die typisch für diese Perspektive sind [vgl. Sippel 2008; S.172 ff.].

2.8.4 EMF – Eclipse Modeling Framework

Wie in Kapitel 2.8 beschrieben, bietet Eclipse eine Vielzahl von Projekten zur Unterstützung bei der Erstellung von Software. Eines dieser Projekte ist das *Eclipse Modeling Framework* (*EMF*), welches sich intensiv mit der modellgetriebenen Softwareentwicklung auseinander setzt.

Dabei unterteilt sich EMF in drei Bestandteile - dem *EMF.Core*, dem *EMF.Edit* und dem *EMF.Codegen*.

2.8.4.1 EMF.Core – Ecore

Der *EMF.Core* enthält ein simples Meta-Modell, um andere Meta-Modelle zu spezifizieren. Es stellt also, in Bezug auf das zu erstellende System, ein Meta-Meta Modell bereit[vgl. 2.4]. EMF kann somit als Meta-Model-Modellierer betrachtet werden. Dieses simple Modell wird *Ecore* genannt. Interessanterweise ist das Ecore ebenfalls ein EMF-Modell, es ist also sein eigenes Meta-Modell.

Grundlegend kann das Ecore auf vier Elemente reduziert werden, *EClass*, *EAttribute*, *EReference* und *EDataType*, deren Zusammenhang in Abbildung 14 dargestellt ist. Diese Namen weisen nicht grundlos eine starke Ähnlichkeit zu den Bezeichnern der UML-Klassendiagramm-Spezifikation auf. Das EMF-Core ist eine vereinfachte Form des Klassendiagramm Modells, welches auf dem MOF-Standard basiert. Um genau zu sein, kann *Ecore* als Implementierung von EMOF betrachtet werden [vgl. 2.6.2].

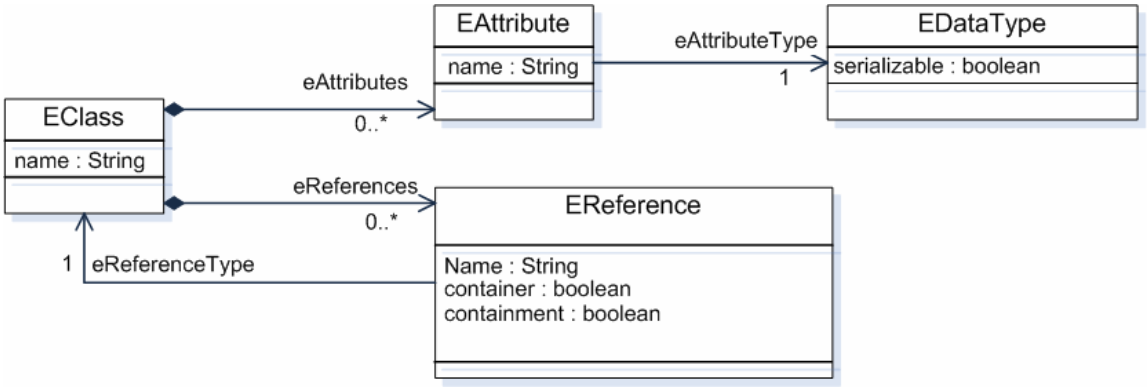

Abbildung 14 – vereinfachtes EMF Ecore Modell
[Quelle: vgl. Merks et al. 2009]

EClasses repräsentieren die Modell-Klassen. Sie beinhalten Attribute (*EAttribute*) und Referenzen (*EReference*). EAttributes sind vergleichbar mit den Attributen aus einem Klassendiagramm. Sie bestehen aus einem Namen und einem Typ (*EDataType*). Im

Gegensatz dazu stellen EReferences eine Assoziation zwischen Klassen dar. EDataTypes werden genutzt, um den Datentyp eines EAttributes darzustellen. Hierbei kann der EDataType sowohl ein primitiver Datentyp (int, double, etc), eine beliebige Java-Klasse, als auch eine in dem Modell definierte Klasse sein.

Auf Basis dieses EMF Meta-Meta-Modells können eigene Meta-Modelle erstellt werden, welche ebenfalls als Ecore Modelle vorliegen. Die Erstellung eines Ecore Modells kann aus verschiedenen Quellen erfolgen. So bietet EMF einen Import aus Java Annotations, UML oder einer XML-Schema Beschreibung an. Zusätzlich besitzt EMF noch einen baumbasierten Editor, mit dem komfortabel ein Meta-Modell erstellt werden kann. In den Begrifflichkeiten der MDSD spiegelt das eigene Ecore das *PIM* (*Platform Independent Model*) wieder.

2.8.4.2 Generator Modell

Basierend auf dem Ecore kann EMF ein generatorspezifisches Modell (*Gen-Model*) erstellen. Dieses Modell dient der Erstellung dreier verschiedener Projekte, als Basis für die weitere Entwicklung des Systems. So kann EMF basierend auf der *Gen-Model* Beschreibung die Java-Klassen erstellen, die denen in dem Modell beschriebenen Klassen entsprechen. Zusätzlich werden Hilfsklassen, wie beispielsweise eine Factory für die Erzeugung von Objekten generiert. Ebenso kann über EMF Quellcode zur Manipulation des Datenmodells (EMF.Edit) und ein lauffähiger Editor erzeugt werden. Das *Gen-Model* dient hierbei als Domänenmodell, um den Generator zu steuern. Im Umfeld der *MDSD* stellt das Gen-Model das *PSM* dar, welches benutzt wird, um den plattformspezifischen Code zu generieren.

2.8.4.3 EMF.Edit und EMF.Codegen

EMF.Edit ist ein EMF-Framework, welches Methoden zur Bearbeitung von EMF-Modellen bereitstellt. Durch *EMF.Edit* wird es Entwicklern ermöglicht eigene Editoren für EMF-Modelle zu programmieren. Hierbei können generische Klassen für die Bereitstellung der Daten (*ViewProvider*) oder ein Command-Framework für die Manipulation der Daten genutzt werden [vgl. auch 2.8.6.1].

EMF.Codegen ermöglicht das Generieren aller nötigen Klassen, um später eigene Editoren erstellen zu können. Hierzu zählen sowohl die Darstellung des EMF-Modells in Java, als auch generische Zugriffsklassen. Zusätzlich kann *EMF-Codegen* bereits einen einfachen baum-basierten Editor für das entwickelte EMF-Modell erzeugen [vgl. Merks et al. 2009].

2.8.4.4 Resource und ResourceSet

Zum Speichern der EMF Informationen wird das *EMF Persistence API* zur Verfügung gestellt. Die beiden wichtigsten Komponenten darin sind die **`Resource`** und das **`ResourceSet`**. Ressourcen stellen dabei eine Art Persistenz-Container für alle Objekte dar. Objekte können einfach einer Ressource hinzugefügt werden. Dabei werden ebenfalls alle ihre Kind-Objekte mit hinzugefügt. Ressourcen bieten zwei Methoden für die Persistierung - **`save()`** und **`load()`**. Um die Implementierung des Datenformats kümmert sich die Ressource. Die von EMF verwendete Ressource speichert alle Daten im XMI-Format und nutzt dazu eine XML-Helper-Klasse. Ressourcen sind aber austauschbar, sodass die XMI-Ressourcen durch andere Implementierungen ersetzt werden können.

ResourceSets sind eine Sammlung von Ressourcen innerhalb eines Kontextes. GMF-Editoren nutzen beispielsweise standardmäßig zwei Ressourcen, eine für die Persistierung des Modells und eine für die View. ResourceSets verwalten alle an ihnen angemeldeten Ressourcen [vgl. Merks et al. 2009].

2.8.4.5 EMF und XMI

Standardmäßig serialisiert EMF das Ecore-Model eines Systems in XMI. *XMI* (*XML Metadata Interchange*) ist ein XML-basiertes Austauschformat der OMG. Ein Vorteil von XMI ist seine Interoperabilität bezüglich MOF-konformer Modelle. Andere Programme können also die von EMF erzeugten Modelle laden und verarbeiten. Ein weiterer wichtiger Anwendungsfall von XMI ist die Nutzung als Basis für einen Code-Generator. So können Generator-Frameworks, wie zum Beispiel *openArchitectureWare* [*oAW*, vgl. 3.2.2], XMI-Modelle als Grundlage für die M2M- oder M2C-Transformation benutzen. Dadurch wird der Generierungsprozess unabhängig davon, mit welchen Tool das Modell erstellt wurde [vgl. Stahl et al. 2007, S. 388 ff.]. Neben dem Meta-Modell für das System wird auch das eigentliche Modell nach XMI serialisiert.

2.8.4.6 Dynamic EMF

EMF bietet neben der Serialisierung von Modellen und Meta-Modellen noch eine weitere für diese Arbeit bedeutsame Funktionalität. Mit Hilfe von *Dynamic EMF* ist es möglich ein Modell programmatisch zu ändern, ohne die eigentliche Implementierung zu kennen. Basis hierfür bilden die beiden Klassen *EObject* und *EClass*. Dabei setzt EMF ein System ähnlich dem der Java Reflections ein. Jedes von EMF generierte

Objekt erbt automatisch von der Basisklasse EObject und bekommt damit eine Reihe nützlicher Methoden zur Verfügung gestellt, mit denen sein Modell bearbeitet werden kann. So können mit den Methoden **`eGet()`** und **`eSet()`** die Daten des Klasse manipuliert werden. Über die Methode **`eClass()`** kann ein Java-Objekt Informationen über seine Klassendefinition erhalten. Mit Hilfe dieser Informationen können die Manipulationen vorgenommen werden, ohne das zugrunde liegende Meta-Modell zu kennen**.** Listing 2 zeigt wie von einem EObject (oldObject) alle Attributswerte in ein anderes EObject (newObject) kopiert werden, ohne dass die konkrete Implementierung beider Objekte bekannt ist. Hierbei stellt das Listing nur ein vereinfachtes Beispiel dar, in dem Typprüfungen außer Acht gelassen wurden.

```
1   for (EAttribute attribute: oldObject.eClass().getEAllAttributes())
2   {
3     newObject.eSet(attribute, oldObject.eGet(attribute));
4   }
5
```

Listing 2 - Dynamic EMF

2.8.5 Exkurs – SWT, JFace und Draw2D

Da die nachfolgenden Eclipse-Projekte durch die Nutzung grafischer Frameworks geprägt sind, soll an dieser Stelle ein Überblick über die bestehenden grafischen Bibliotheken geschaffen werden.

In der Java-Welt gibt es verschiedene Bibliotheken, welche es erlauben, grafische User-Interfaces zu erstellen. Das *Abstract Window Toolkit* (*AWT*) war dabei die erste Implementierung. AWT als schwergewichtiges Toolkit bezeichnet, was bedeutet, dass es zur Darstellung der grafischen Elemente die Funktionen benutzt, die direkt vom Betriebssystem angeboten werden. Da Java allerdings plattformunabhängig ist, konnte die AWT Implementierung nur aus dem kleinsten gemeinsamen Nenner der GUI-Elemente aller unterstützten Betriebssysteme bestehen. Auch führte dieser Ansatz dazu, dass die Oberfläche auf unterschiedlichen Systemen anders aussah. Diese Nachteile und weitere Einschränkungen von AWT führten dazu, dass mit Java 1.2 eine neue grafische Bibliothek in den Standard integriert wurde – *Swing*. In Swing wurden alle Komponenten selbst implementiert. Es verfolgt also einen leichtgewichtigen Ansatz, was die GUI unabhängig von den vom Betriebssystem bereitgestellten Elementen macht. Dies hat allerdings den Nachteil, dass das Zeichnen von Elementen sehr ressourcenlastig ist, wodurch auf Swing basierende Anwendungen oft als sehr „schwerfällig“ empfunden werden. Nicht zuletzt diese Tatsache ist mit dafür

verantwortlich, dass Java als langsam bezeichnet wird. Als Reaktion auf diese beiden Entwicklungen erstand 2001 das *Standard Widget Toolkit* (*SWT*) von IBM. Ebenso wie AWT nutzt SWT die nativen GUI Elemente (*Widgets*) des Betriebssystems. Allerdings kann es auch, wenn diese nicht vorhanden sind, einzelne Widgets emulieren. Zusätzlich zu üblichen Widgets bietet SWT auch eine Vielzahl weiterer nützlicher Elemente, wie Baum- oder Kalender-Editoren an [vgl. Sippel 2008, S.69]. Die gesamte Eclipse IDE ist aus SWT aufgebaut.

Da das Erstellen von komplexen GUIs mit sämtlichen Ereignisbehandlungen unter Umständen recht aufwendig werden kann und komplexere Widgets (komplexe Dialoge und Wizards) immer wieder neu erstellt werden mussten, wurde eine zusätzliche Bibliothek geschaffen, die Teile von *SWT* kapselt und einfachere Implementierungen erzeugen kann. *JFace* bildet also eine Implementierungsschicht, welche direkt auf SWT aufsetzt. Es vereinfacht die Entwicklung von grafischen Oberflächen wesentlich und erlaubt es übersichtlicheren Code zu erstellen [vgl. Sippel 2008, S.81 ff.].

Neben dem JFace Toolkit zum Erstellen höherwertigerer Dialoge entwickelte sich eine weitere grafische Bibliothek – das *Draw2D*. Ursprünglich aus dem GEF-Projekt [vgl. 2.8.6] entstanden, bietet Draw2D die Möglichkeit Elemente auf die grafische Oberfläche zu malen. So können beliebige Elemente durch die Kombination von Grundformen (Linien, Rechtecke, Kreise etc.) erstellt werden. Ein wichtiger Bestandteil von *JFace* sind die Viewer. Sie erlauben eine einfache Trennung von grafischer Repräsentation und dem Datenmodell. Sie nutzen dafür spezielle Klassen, die ihnen den anzuzeigenden Inhalt des Modells aufbereitet liefern (*ContentProvider*). Durch diese Technik können unterschiedliche Viewer (*TreeViewer*, *TableViewer*), auf dasselbe Datenmodell zugreifen, es aber unterschiedlich darstellen.

2.8.6 GEF – Graphical Editing Framework

Ein Weiteres der 23 Hauptprojekte von Eclipse ist das *Graphical Editing Framework* (*GEF*). GEF erlaubt die einfache Entwicklung von grafischen Editoren. Dabei basiert GEF auf dem *Model-View-Contro*l Design Pattern (*MVC*), was eine Trennung von grafischer Repräsentation und zugrunde liegendem Modell ermöglicht. Eine bekannte Implementierung von GEF ist der Eclipse Visual Editor [vgl. Daum 2007, S. 353].

Grundlage für die Umsetzung des MVC Pattern bilden in GEF die EditParts, welche die Funktion von Controllern übernehmen. EditParts sind Klassen, welche die grafische Repräsentation eines Objektes erzeugen. Sie stellen also die Definition einer Grafik mit

Hilfe von Draw2D dar. Diese Darstellungen erben immer von der Basisklasse `Figure` [vgl. Mammana 2008].

Neben der Erzeugung der View stellen die EditParts die Verbindung zum Modell her. In GEF kann das Model frei definiert werden und liegt in Form von einfachen Java-Beans vor.

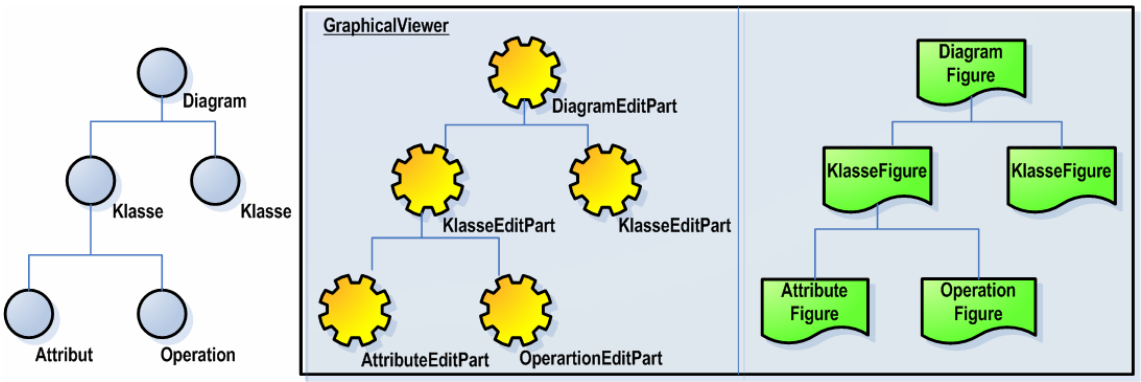

Abbildung 15 - Darstellung von MVC in GEF

GEF ist sehr fein granular. Das bedeutet, dass für die Verwaltung einzelner Objekte jeweils ein eigener EditPart und eine eigene Figur existieren. Besteht zum Beispiel das Datenmodell *Klassen* aus Attributen und Operationen, so können für diese eigene Figuren und EditParts implementiert werden.

Abbildung 15 stellt dieses Beispiel dar. Um die Zeichenelemente zu visualisieren, wird wie in JFace ein Viewer, eine Implementierung des Interfaces GraphicalViewer, genutzt. Dieser Viewer besitzt eine *EditPartFactory* welche die EditParts, und somit auch die in den EditParts enthaltenen Figures, für das Modell erstellt [vgl. Eclipse 2009].

2.8.6.1 Requests, Commands und EditPolicies

Zur Manipulation des Modells nutzt GEF das *Eclipse Command Framework*, welches auch EMF zugrunde liegt. Innerhalb des Command Frameworks werden alle Änderungen mit Hilfe von Kommandos (*Commands*) ausgeführt. Diese werden auf einem *CommandStack* hinterlegt und dort zur Ausführung gebracht. Dieser *CommandStack* ist mit eine globalen Instanz, der *EditDomain* verbunden. Mit Hilfe des CommandStacks liegt eine Historie der abgearbeiteten Kommandos vor. Anhand dieser Information können Undo- und Redo-Operationen ausgeführt werden. Die Verwaltung der EditPart-spezifischen Kommandos wird von so genannten *EditPolicies* vorgenommen. EditPolicies reagieren auf *Requests*. Um beispielsweise das Attribut

einer Klasse zu ändern, muss ein entsprechender Request an den EditPart gestellt werden, der für die Verwaltung der entsprechenden Klasse verantwortlich ist. Dieser leitet die Anfrage an die entsprechende EditPolicy weiter. Die EditPolicy entscheidet daraufhin, welches Kommando, basierend auf dem Request, ausgeführt werden soll, und liefert dieses an den EditPart zurück. Dieser wiederum liefert das Kommando an die aufrufende Instanz zurück, welche dieses dann zur Ausführung bringen kann. Das Sequenzdiagramm in Abbildung 16 verdeutlicht diesen Sachverhalt. EditPolicies können, auch zur Laufzeit, mit EditParts verknüpft oder von diesen entfernt werden. Durch dieses Prinzip kann das System sehr flexibel gestaltet werden [vgl. Daum 2007, S.358 ff.].

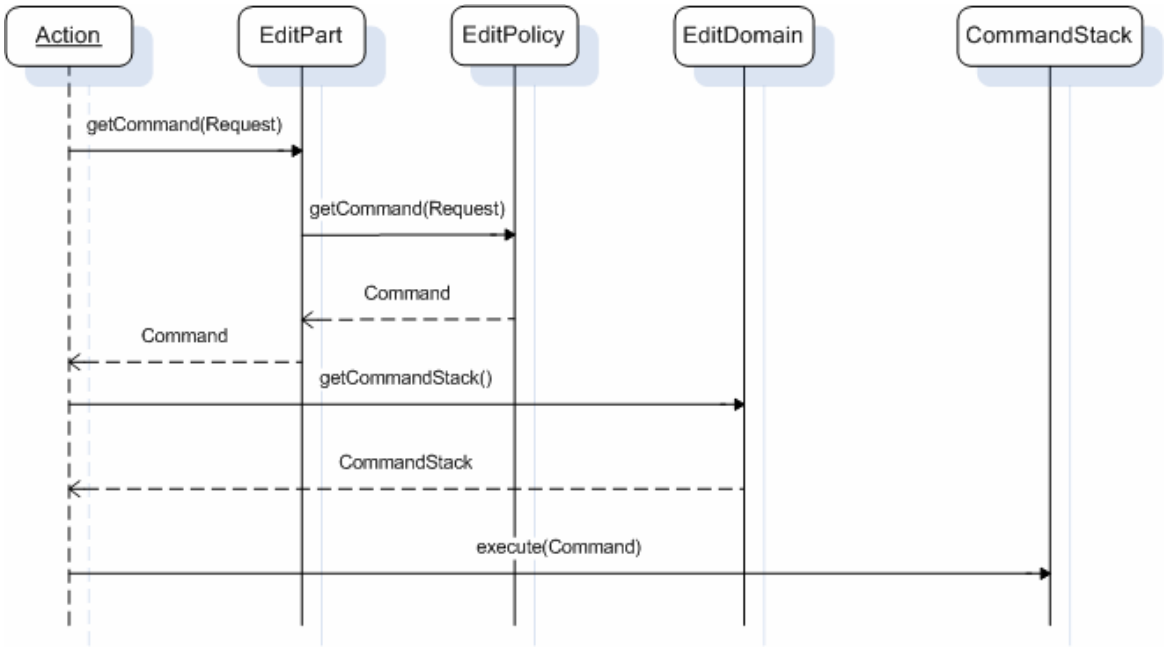

Abbildung 16 - Ausführen eines Kommandos

2.8.7 GMF – Graphical Modeling Framework

Das *Graphical Modeling Framework* (*GMF*) bildet die Symbiose zwischen EMF und GEF. Hierbei erlaubt es anhand eines einfachen vordefinierten Workflows grafische Editoren zu erstellen, welche auf einem EMF-Modell aufbauen. EMF liefert also die Bearbeitung des Daten-Modells, während GEF Methoden zur Behandlung der grafischen Repräsentation besteuert. Dabei können sehr schnell und flexibel lauffähige Editoren erstellt werden. In den folgenden Kapiteln werden die für die Arbeit notwendigen Begrifflichkeiten im Umfeld von GMF erläutert.

2.8.7.1 GMF-Tooling

Mit Hilfe von GMF kann, ohne eine Zeile Code zu programmieren, ein grafischer Editor erzeugt werden. Hierbei werden modellgetriebene Technologien verwendet. Grundlage eines jeden GMF-Projektes bildet ein EMF-Modell.

GMF teilt sich in zwei wesentliche Komponenten - *Tooling* und *Runtime*. Die Tooling Komponente erlaubt durch Modelle das Aussehen und das Verhalten des grafischen Editors zu verändern und über einen Generator einen Editor zu erzeugen. Das *GMF-Graph Model* definiert das Aussehen der grafischen Elemente. Hier kann definiert werden, welche Knoten und welche Kanten der Editor darstellen soll und wie diese geschaffen sind. Dabei reicht die Vielfalt von einfachen Rechtecken, bis hin zu sehr komplexen Gebilden. Das Aussehen eines jeden Elements wird über eine Baumstruktur erzeugt, mit deren Hilfe das Aussehen fein gegliedert werden kann [vgl. Gronback 2009, S.503 ff.].

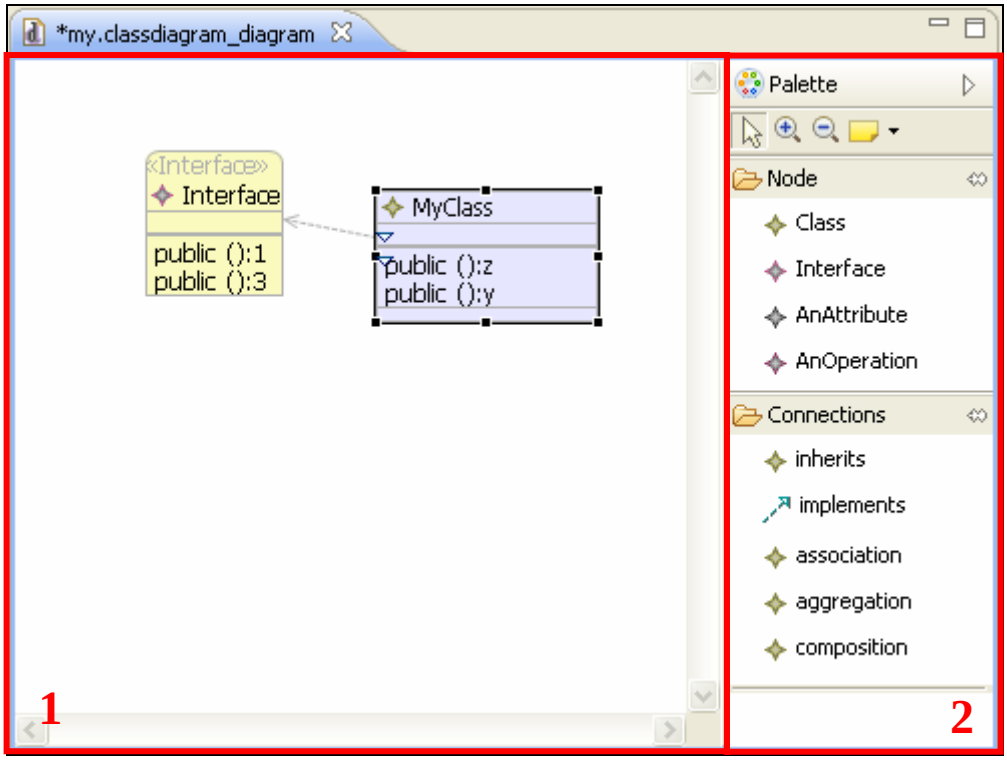

Abbildung 17 - Einfacher mit GMF erzeugter Editor

Ein weiteres Modell, das *GMF-Tooling-Model*, definiert den Aufbau der Werkzeugleiste [vgl. Abbildung 17, Nr. 2]. Auch hier kann anhand einer baumbasierten Struktur das Aussehen und Verhalten des Editors konfiguriert werden. Mit diesem

Modell kann der Entwickler bestimmen, welche Elemente in der Werkzeugleiste vorkommen und dementsprechend auch auf der Zeichenfläche gezeichnet werden können. Zusätzlich bietet er Funktionen zur Gestaltung der Menüleiste, wie zum Beispiel das Gruppieren von Elementen [vgl. Gronback 2009, S.518 ff.].

Aufbauend auf dem Domänen-Modell (*ecore*), der grafischen Definition (*gmfgraph*) und der Tooling-Definition (*gmftool*) werden im GMF-Mapping-Modell diese Informationen zu einer Einheit verwoben. Das Mapping Modell ordnet hierbei allen anzuzeigenden Elementen ein Domänen-Objekt, ein Aussehen und einen Eintrag in der Werkzeugleiste zu [vgl. Schuster 2008, S. 43].

Als Eingabe für den GMF-Generator, der aus der Spezifikation den Editor erzeugt, dient ein weiteres Model – das *GMF-GenModel*. Es wird aus dem GMF-Mapping-Modell erzeugt.

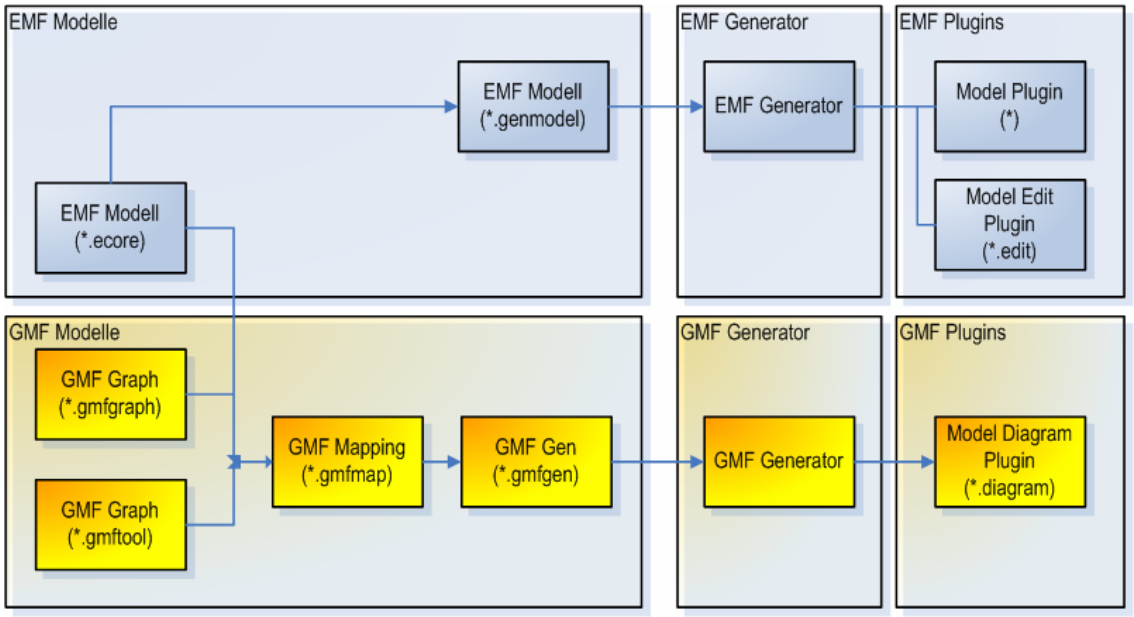

Abbildung 18 - GMF-Modelle

In Abbildung 18 wird der Zusammenhang zwischen EMF und GMF noch einmal grafisch dargestellt. GMF setzt dabei auf ein bereits vorhandenes EMF Modell auf. Während EMF für die Generierung des Model-Codes und des Edit-Codes zuständig ist [vgl. 2.8.4.2], erzeugt GMF den Quellcode für den Editor und baut dabei auf die bereits erstellen Plugins auf. Im Ergebnis entsteht ein Editor basierend auf dem bestehenden Domänenmodell, der im weiteren Schritt um weitere Features ergänzt werden kann.

2.8.7.2 GMF-Runtime

Die zweite Komponente des GMF-Frameworks ist die GMF-Laufzeitumgebung (*GMF Runtime*). Zentraler Bestandteil der GMF-Architektur sind die EditParts [vgl. 2.8.6] des GEF-Frameworks. Da GMF auf GEF aufbaut, wird auch hier strikt das Modell von der View getrennt. Im GMF-Sprachgebrauch wird das Domänenmodell als semantisches Modell (*Semantic Model*) und das grafische Modell als Notationsmodell (*Notational Model*) bezeichnet. Das *Notational Model* liefert eine Referenz auf das eigentliche Domänen-Modell und bietet darüber hinaus modellspezifische Informationen für die grafische Darstellung. Hierzu zählen unter anderem die Position, die Größe oder verknüpfte semantische Elemente. Die Objekte innerhalb des Notational Models, die zur Speicherung der View-Informationen dienen, werden als *View* bezeichnet. Zu beachten ist hier, dass es sich bei View-Objekten nicht um die View im Sinne des MVC-Pattern handelt, sondern ausschließlich um eine Modell-Komponente, welche für die Persistenz der Layoutinformationen der einzelnen Grafiken verantwortlich ist. Die eigentliche View wird im GMF-Modell durch auf Draw2D basierende Figure-Klassen repräsentiert [vgl. Abbildung 19]. Hierin besteht auch der wesentliche Unterschied zur GEF Architektur. Während GEF beliebige Domänenmodelle verwenden kann, besitzt GMF ein festgelegtes Modell, welches aus dem semantischen und dem notationellen Modell zusammensetzt ist. Auf Grund dieser Tatsache wurde auch die EditPart-Schnittstelle in GMF um zusätzliche Funktionen erweitert. Alle GMF-Editparts implementieren das Interface `IGraphicalEditpart`, welches unter Anderem zusätzliche Methoden zum Zugriff auch das Semantic Model definiert. Weiterhin besitzen alle EditParts eine Referenz auf die anzuzeigende *Figure*. Das Aussehen der Figur wird dabei mittels Draw2D realisiert und bei der Code-Generierung des GMF-Editors mit erzeugt. In der Regel liegt die *Figure*-Klasse als Inner-Class in jedem EditPart vor.

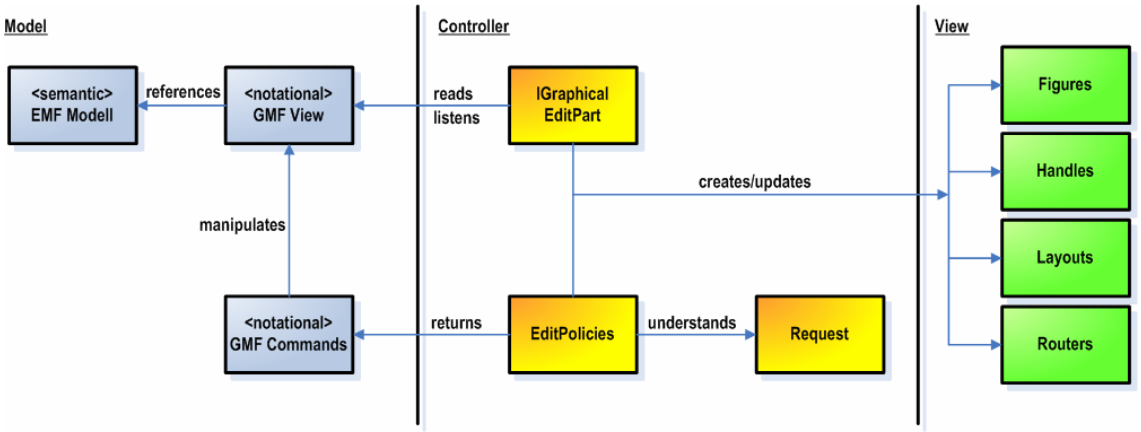

Abbildung 19 - GMF-Architektur
[Quelle: vgl. GMF 2009b]

Neben den EditParts wurde auch das Command-Framework um weitere für GMF benötigte Commands erweitert. So gibt es bereits vorgefertigte Kommandos für immer wieder benötigte Operationen, wie zum Beispiel für das Setzen von Werten (**`SetValueCommand`**) oder das Erstellen von Elementen (**`CreateElementCommand`**). Zur Manipulation des Modellbestandes nutzt GMF das aus GEF bekannte System bestehend aus EditPolicies, Requests und Commands [vgl. Gronback 2009, S.562 ff.].

2.8.7.3 Figure Deskriptoren

GMF nutzt *Figure Deskriptoren*, um innerhalb des GMF-Graph-Modells das Aussehen der einzelnen Diagrammelemente zu definieren. Für jedes Element wird ein Figuren Deskriptor erstellt. Diese werden in einer *FigureGallery* zusammengefasst.

Zur Darstellung von Knoten stehen verschiedene Grundformen zur Verfügung. Rechtecke (**`Rectangle`**), abgerundete Rechtecke (**`RoundedRectangle`**) und Ellipsen (**`Ellipse`**) können direkt verwendet werden. Für kompliziertere Grundformen kann die Klasse **`Polygon`** verwendet werden, um das Aussehen des Knotens beliebig zu gestalten.

Um Referenzen zwischen Knoten, also Kanten darzustellen, werden in GMF **`PolyLine Connections`** genutzt. PolyLine Connections können Referenzen auf *Decoration Figure* Deskriptoren enthalten, welche ebenfalls in der FigureGallery definiert sind. GMF unterscheidet dabei zwei verschiedene Arten von Dekorationen **`PolyLine Decorations`** und **`Polygon Decorations`**. Erstere dienen ausschließlich der Darstellung von linienorientierten Dekorationen. Mit ihrer Hilfe können einfache Pfeile oder Objekte ohne Hintergrund erstellt werden. **`Polygon Decorations`** dagegen sind immer ausgefüllte Objekte. Das bedeutet, dass der Bereich, der von den Punkten einer Polygonbeschreibung umspannt wird, mit der in der Dekoration angegebenen Farbe ausgefüllt wird. Sie können zum Beispiel genutzt werden, um Darstellungen wie ausgefüllte Rauten zu erstellen. Um das Aussehen der Dekoratoren zu verändern, können **`Template Points`** benutzt werden. Ein **`Template Point`** definiert einen Punkt im zweidimensionalen Raum, relativ zum Ende der PolyLine. Über die Angabe mehrerer **`Template Points`** kann so die Beschreibung komplexer Objekte hinterlegt werden [vgl. GMF 2009a].

Kanten und Knoten können mit Beschriftungen, den so genannten **`Labels`**, versehen werden. Neben diesen kann das Aussehen von Kanten und Knoten auch über Parameter verändert werden, mit denen sich Hintergrund- und Vordergrundfarbe oder die Liniendicke der Elemente einstellen lassen.

2.8.7.4 ElementTypes

Jede Figur innerhalb der *FigureGallery* kann mehrfach genutzt werden. So enthalten beispielsweise in einem Klassendiagramm sowohl die Klassen als auch die Schnittstellen eine Figur zum Anzeigen von Operationen. Diese wird aber nicht mehrfach definiert, sondern es gibt nur eine Figure, welche innerhalb des GMF-Mapping-Modells entweder dem Klassen- oder Schnittstellen-Modell zugeordnet wird. Dadurch wird die Darstellung (GMF-Graph) vom Datenmodell (semantic und notational Modell) abstrahiert. Erst auf der Ebene des Mappings wird eine Zuordnung getroffen. Damit das GMF-Framework definieren kann welche Figure welchem View zugeordnet wird, gibt es die *ElementTypes*. ElementTypes werden an einer globalen *ElementTypeRegistry* angemeldet und können über eine eindeutige ID abgefragt werden. Dieser Identifikator besteht aus dem Namen der Klasse des Datenmodells und einem Wert für die Unterscheidung der Zuordnung. Für das oben aufgeführte Beispiel würden in GMF zwei ElementTypes definiert werden. Einer für die Methoden innerhalb von Klassen (**`org.mftech.diagram.uml.class.diagram.AnOperation_2002`**) und einer für Methoden in Schnittstellen (**`org.mftech.diagram.uml.class.diagram.AnOperation_2004`**).

2.8.7.5 Vor- und Nachteile von GMF

Das GMF-Framework erlaubt es schnell und effizient Rohformen von grafischen Editoren zu erzeugen, welche für viele Anwendungszwecke bereits vollkommen ausreichend sind. Bei Bedarf kann aufgrund des generischen Rahmenkonzeptes von GMF der Editor beliebig erweitert werden. Im Vergleich zu GEF ist es aber an ein festes Modell gekoppelt. Soll also ein Domänenmodell genutzt werden, welches nicht auf EMF beruht, muss es nach EMF konvertiert oder GEF als Framework für die Erstellung des grafischen Editors genutzt werden.

2.9 Erweiterte Grundlagen

Nach der Einführung in die wichtigsten Konzepte der modellgetriebenen Softwareentwicklung und der Welt von Eclipse werden im folgenden weitere Begriffe und Frameworks erläutert, welche für die umzusetzenden Ziele der Arbeit benötigt werden.

2.9.1 Web Services

Unter *Web Services* werden Software-Komponenten verstanden, welche ihre Funktionalitäten über standardisierte Schnittstellen webbasiert zur Verfügung stellen. Sie sind über einen *Uniform Resource Identifier* (*URI*) eindeutig referenzierbar. Die Spezifikation der Schnittstellen erfolgt dabei in XML und ist damit plattformunabhängig und interoperabel einsetzbar [vgl. Burghardt 2004]. Zur Beschreibung von Schnittstellen kann die XML-basierte *Web Service Description Language* (*WSDL*) genutzt werden. Die bekanntesten Web Service Implementierungen, *XML-RPC* (*XML Remote Procedure Call*), *SOAP* (ehemals *Simple Object Access Protocol*) und *REST* (*REpresentational State Transfer*), setzen dabei auf das HTTP-Protokoll auf.

SOAP ist ein durch die W3C[13] standardisiertes Protokoll und stand ursprünglich für *Simple Object Access Protocol*. Da es sich aber schnell zu einer sehr komplexen Spezifikation entwickelte und auch nicht für den Zugriff auf Objekte gedacht war, wurde dieses Akronym abgelegt. Gelegentlich findet aber der Begriff SOA-Protokoll (wobei SOA in diesem Fall für Serviceorientierte Architektur steht) Anwendung für SOAP. SOAP kann benutzt werden, um einen *entfernten Prozeduraufruf* (*Remote Procedure Call, RCP*[14]) auszuführen. Dabei nutzt es eine festgelegte XML-basierte Syntax, um die Parameter und Rückgabewerte auszutauschen. SOAP kann dabei auf unterschiedliche Übertragungsprotokolle aufsetzen [vgl. Melzer et al., S. 76 ff].

Das von Dave Winer 1998 entwickelte XML-RPC kann als vereinfachte Version von SOAP betrachtet werden. Im Gegensatz zu SOAP ist es nicht durch das W3C standardisiert und nutzt ausschließlich HTTP als Übertragungsprotokoll. Es ist zudem auch nicht so flexibel einsetzbar [vgl. Melzer et al. 2008, S100 ff.].

Anders als SOAP und XML-RPC stellt der *REpresentational State Transfer* (*REST*) kein Protokoll, sondern einen Architekturstil dar, der zur Umsetzung von Web Services genutzt werden kann. REST nutzt neben der HTTP-Operation *GET* zum Anzeigen von Informationen auch die Operationen *POST*, *PUT* und *DELETE* zur Datenmanipulation. REST spezifiziert nur, wie die Operationen ausgeführt werden. Den Inhalt der Kommunikation kann der Entwickler selbst bestimmen [vgl. Melzer et al. 2008, S103 ff.].

---

[13] World Wide Web Consortium; http://www.w3.org/

[14] Remote Procedure Call, Aufrufen einer Funktion über Systemgrenzen hinweg. Es ist dabei für den Aufrufer transparent, wo die Ausführung stattfindet und kann auhc auf einem anderen Rechner oder System erfolgen [vgl. Tanenbaum 2003, S526 ff.]

2.9.2 Apache Axis

„Axis ist ein in der Sprache Java geschriebener SOAP-Prozessor, der sich als Servlet in die Tomcat Servlet Engine integrieren lässt“ [Melzer et al. 2008, S. 172].

Axis ermöglicht es den Umgang mit dem XML-Dialekt von SOAP vollkommen zu vermeiden, indem es die gesamte Kommunikation hinter Java-Implementierungen versteckt. Es kann sogar basierend auf einer WSDL die Zugriffsklassen für einen Web-Service automatisch generieren. Dabei genügt es innerhalb eines *Web Service Deployment Descriptors* (*WSDD*) den Namen einer Schnittstelle zu definieren. Axis ist anhand dieser Information in der Lage alle Methoden dieser Schnittstelle in eine entsprechende WSDL umzuwandeln und ihre Implementierung als Web Service anzubieten. Es integriert sich dafür als Servlet innerhalb eines Web-Containers [vgl. Melzer et al. 2008, S. 173 ff.].

2.9.3 Hibernate

Hibernate[15] ist ein objektrelationaler Mapper für Java, welcher es ermöglicht objektorientierte Sachverhalte auf relationale Tabellen umzusetzen. Hibernate ermöglicht es ohne direkte Nutzung von *SQL* (*Structured Query Language*) Java-Objekte in einer Datenbank zu persistieren. Dabei können Einfüge- und Änderungsoperationen als auch Transaktionen direkt über Java-Code abgewickelt werden. Die Zuordnung zwischen den Klassen und den Tabellen der Datenbank kann dabei entweder über XML-Dateien oder Java Annotations erfolgen. Um Daten abzufragen, nutzt Hibernate neben SQL auch *HQL* (*Hibernate Query Language*), eine SQL-angelehnte Sprache, die aber einen stärkeren Fokus auf die objektorientierten Aspekte des Systems legt [vgl. Oates et al. 2008, S.23 ff.].

2.9.4 Ajax

Unter dem Begriff *Ajax* (*Asynchronous JavaScript and XML*) werden zusammenfassend JavaScript-Technologien verstanden, mit denen es möglich ist eine Kommunikation zwischen einem Web-Server und dem Browser aufzubauen, ohne den gesamten Inhalt einer Webseite neu laden zu müssen. Interessanterweise kann Ajax sowohl synchron, als auch asynchron ausgeführt werden. Grundlage von Ajax bildet ein *XMLHttpRequest*, welcher eine *HTTP*-Anfrage im Hintergrund ausführen und somit Daten von Server laden kann [vgl. Wenz 2008]. In Kombination mit dem *Document Object Model*

[15] https://www.hibernate.org/

(*DOM*[16]) ermöglicht Ajax es so dynamisch Teile einer Webseite neu zu laden oder auszutauschen.

2.10 Zusammenfassung

In den vorangegangenen Kapiteln wurden die für das Verständnis der Arbeit notwendigen Grundlagen gelegt. Systeme können mit Hilfe von Modellen grafisch, textuell oder hybrid dargestellt werden wodurch der Überblick über die Materie erleichtert wird. Modelle basieren auf Meta-Modellen, welche den Rahmen für ein Modell vorgeben. Meta-Modelle können ebenfalls auf einem Meta-Modell basieren, dem Meta-Meta-Modell. Ein bekanntes Meta-Meta Modell ist das *MOF*, welches die Basis für das UML-Meta Modell bildet.

Der Kern von Eclipse basiert auf *OSGi*, einer flexiblen, komponentenbasierten Softwareplattform. Die in Eclipse vorliegende OSGi Implementierung *Equinox* bietet dabei die Möglichkeit vielfältige, modulare Anwendungen auf Basis einer *Rich Client Platform* zu erstellen. Die bekannte Eclipse-IDE ist ebenfalls eine RCP. RCPs können durch die *Plugin Development Environment* erstellt und erweitert werden.

Das *Eclipse Modeling Framework* bietet Werkzeuge zur Erstellung von Meta Modellen, zur Generierung von Quellcode und zur Bearbeitung von Modellen basierend auf den Meta-Modellen an. Mit Hilfe des *Graphical Modeling Frameworks* können die Modelle nicht nur in einer baumbasierten Ansicht, sondern auch in einem graphischen Editor bearbeitet werden. Hierbei nutzt das *GMF* neben *EMF* auch das *GEF*, einem Framework zur Erstellung von Editoren.

Abschließend wurden mit einer Einführung in *Web Services* und dem Apache Framework *Axis* die Basis für die web-basierte Kommunikation des Systems gelegt. Zusätzlich das Framework *Hibernate* illustriert, mit dem die Persistenzschicht des Systems implementiert werden soll.

[16] DOM – Technologie, welche es ermöglicht ein HTML- oder XML-Dokument in Form eines Baumes darstellbar und zugreifbar zu machen [*vgl. Wenz 2008S.249 ff]*.

3 Anforderungsanalyse und -definition

Innerhalb der Arbeit soll eine generische, modellgetriebene Netzwerkkomponente entwickelt werden, mit der es möglich ist GMF-basierte Editoren über ein Netzwerk so zu verbinden, dass den Anwendern ein kollaboratives Arbeiten am grafischen Modellbestand ermöglicht wird. Hierbei sind unterschiedlichste Aspekte, sowohl in Hinsicht auf softwaretechnische Probleme, als auch in Bezug auf die Nutzerfreundlichkeit, bei der Verwendung der Schnittstelle und der modellgetriebenen Erzeugung der dafür notwendigen Erweiterungen, relevant.

Im Folgenden sollen die Anforderungen an das Projekt analysiert und infolgedessen spezifiziert werden. Dabei werden mögliche Technologien betrachtet und auf ihre Verwendbarkeit innerhalb des Projektes hin untersucht.

In Vorbereitung auf die Entwicklung des Konzeptes werden in diesem Kapitel die funktionalen, systemtechnischen Anforderungen an das System erstellt und festgehalten. Diese Spezifikation resultiert in dem in Anhang A festgehaltenen Anforderungskatalog.

Eingangs werden verschiedene Ansätze diskutiert und bezüglich ihrer Nützlichkeit und Umsetzbarkeit im Rahmen des Projektes analysiert. Dies geht einher mit der Definition der für diese Arbeit notwendigen Systemfestlegungen.

3.1 Betrachtung Eclipse-basierter Netzwerk-Frameworks

Das Eclipse Projekt stellt zwei Frameworks zur Verfügung, um netzwerkbasiert Informationen austauschen zu können. In den folgenden Kapiteln sollen beide betrachtet und für eine mögliche Nutzung im Projekt evaluiert werden.

3.1.1 CDO – Connected Distributed Objects

Das Eclipse Project bietet mit *Connected Distributed Objects* (*CDO*[17]) ein Framework, um EMF-basierte Diagramme kollaborativ über ein gemeinsames Repository nutzen zu können. Es ist ein Sub-Projekt des EMFT-Projektes (*Eclipse Modeling Framework Technology*). Ziel von CDO ist es, das kollaborative Arbeiten an baumbasierten EMF-Projekten zu unterstützen und zu vereinfachen. Dabei setzt CDO für die

[17] http://wiki.eclipse.org/CDO

Kommunikation auf Net4J[18], einer Client-Server-Plattform zur Erstellung und Verwendung von Netzwerkprotokollen. Da GMF-Modelle für die Datenhaltung EMF-Modelle benutzen (semantic Model) und die in GMF verwendeten View-Elemente (notational Model) ebenfalls ein EMF Modell darstellen, wäre ein kollaborativer GMF-Editor auf Basis von CDO prinzipiell möglich. Allerdings lassen sich, basierend auf der System-Architektur von CDO, nicht alle Anforderungen des zu entwickelnden Systems umsetzten. Zum einen ist es für CDO-Projekte erforderlich das bestehende *PSM* durch ein CDO-basiertes Modell zu ersetzen, welchen Einfluss auf die Generierung des Java-Codes nimmt. So werden unter anderem die generierten Klassen um ein ID-Attribut erweitert um die eindeutige Identifikation von Objekten im System zu gewährleisten [vgl. Stepper 2008]. Da die zu erstellende Plattform zum Ziel hat, so wenige Änderungen wie möglich an vorhandenem Code bzw. Framework auszuführen, um auch schon bestehende Editoren ohne Änderung integrieren zu können, würde die Nutzung von CDO verhindern, dass dieses Ziel vollständig erreicht werden könnte. Wäre dies aber die einzigste Einschränkung, läge die Änderung des *semantischen* Modells noch im vertretbaren Rahmen. Um allerdings die View-spezifischen Klassen des *notational* Models (Diagram, Node, Edge, etc.), welche integrierter Bestandteil von GMF sind, an dieses System anzupassen, müsste ein ungleich höherer Aufwand betrieben werden.

Ein weiteres Problem, welches gegen die Nutzung von CDO als Kommunikationsplattform spricht, liegt im Kommunikations-Protokoll selbst. CDO nutzt für die Verbindung zwischen Server und Client ein eigenes, binäres Protokoll, welches mit Hilfe des Kommunikations-Frameworks *Net4J* implementiert wurde [vgl. Stepper 2009]. Dadurch wird eine Interoperation mit anderen Systemen stark erschwert, beziehungsweise gänzlich verhindert. Um Clients außerhalb des Java Umfelds an das System anschließen zu können, bedürfte es einer eigenständigen Implementierung, die in der Lage ist mit Net4J zu kommunizieren und das Protokoll von CDO zu verstehen.

Net4J bietet den großen Vorteil, dass es für die Anwendungsschicht die zugrunde liegende Kommunikationsimplementierung verbirgt. So können auf Net4J basierende Systeme, also auch CDO, über verschiedene Kommunikationswege arbeiten. CDO bietet zum einen den direkten Zugriff über TCP, aber auch einen Wrapper-Modus für die Verwendung von HTTP an. Somit könnten Clients aus beliebigen Netzwerken heraus auf den Server zugreifen. Allerdings werden die Daten dem Client nur mittels Lazy-Loading zur Verfügung gestellt. Dadurch werden nur die Objekte an den Client übertragen, die er auch tatsächlich gerade benutzt beziehungsweise benutzen möchte.

[18] http://wiki.eclipse.org/Net4j

Dieses Vorgehen hat zum einen den Vorteil, dass das System eine hohe Performance gewährleisten kann, da die Kommunikation auf ein Minimum reduziert wird. Zum anderen wäre bei sehr großen Modellen eine komplette Übertragung aller Daten mit den heutigen technischen Möglichkeiten undenkbar. So ist CDO für Modellbestände in Größenordnungen von mehreren Gigabytes ausgelegt, was bei einer initialen Übertragung mehrere Stunden in Anspruch nehmen würde. Leider verhindert dieser Mechanismus aber auch, dass kleinere Projekte unabhängig vom Server arbeiten können. Bricht die Verbindung zum Server ab, kann der Client nicht mehr an seinem Modell arbeiten.

CDO ist ein leistungsstarkes und performantes Kommunikations-Framework, welches auf die Bedürfnisse von EMF-Modellen zugeschnitten ist. Allerdings erlaubt es auf Grund der System-Architektur nicht die in den Zielen betrachteten Anforderungen umzusetzen.

3.1.2 ECF – Eclipse Communication Framework

Neben CDO beschäftigt sich ein weiteres Eclipse Projekt damit, Funktionalitäten zur Kommunikation über vorgefertigte Kommunikationskanäle zur Verfügung zu stellen. Dabei baut *ECF* (*Eclipse Communication Framework*) zurzeit auf bestehende Kommunikationsnetzwerke auf. Es bietet Provider für *XMPP* (*Extensible Messaging and Presence Protocol*, *Jabber*), *JMS* (*Java Message Service*) oder *IRC* (*Internet Relay Chat*). Über einen generischen Adapter können auch eigene Implementierungen umgesetzt werden. Die Kommunikation baut auf dem von Instant Messengern bekannten *Message* Prinzip. ECF ermöglicht es innerhalb von RCP-Anwendungen die oben genannten Protokolle einzubinden und so Messenger Funktionalitäten umzusetzen [vgl. Haischt 2006].

ECF ist ein interessantes Framework. Allerdings waren zum Zeitpunkt, an dem diese Arbeit begann, die Unterstützung für Real-Time Shared Editing noch nicht ausgereift und Veränderungen unterworfen [vgl. ECF 2008]. Aus diesem Grund wurde ECF nicht verwendet. Deshalb wird für die Kommunikation ein eigenes System entwickelt werden.

3.2 Generator Frameworks

Das System wird sich in den generativen Entwicklungsprozess des GMF-Frameworks einreihen, dass bedeutet, dass sämtliche Erweiterungen und Quellcodes ebenfalls über

zugrunde liegende Modelle in ausführbaren Programmcode überführt werden. Im Bereich der modellgetriebenen Softwareentwicklung gibt es verschiedene Frameworks, welche Transformationen sowohl von Model zu Modell, als auch von Modell zu Code unterstützten. Im Folgenden werden verschiedene dieser Frameworks zur Generierung von Text aus Modellen (M2T) verglichen und eine Entscheidung für das am geeignetsten in Bezug auf diese Arbeit getroffen.

3.2.1 JET – Java Emitter Templates

Java Emitter Templates (*JET*) ist ein Sub-Projekt des Eclipse M2T (Model-to-Text) Projektes, welches Code aus einem bestehenden Modell erzeugt. Dabei nutzt es eine Syntax, welche stark an *Java Server Pages* (*JSP*) angelehnt ist, um die Transformationsvorschriften zu erstellen. Wie JSP ist JET also eine templatebasierte Sprache. Es enthält einen statischen Anteil, welches 1:1 in den generierten Text übernommen wird und einen dynamischen Anteil, um den Text mit Informationen aus dem zu transformierenden Modell anzureichern. JET nutzt *XPath* (*XML Path Language*[19]), um das Modell einzulesen und einzelnen Teile aus ihm abzufragen. Aus diesem Grund müssen alle Modelle, die mit JET verarbeitet werden, im XML-Format vorliegen. JET ist sehr einfach zu erlernen, besonders wenn Vorkenntnisse im Umgang mit Java Server Pages vorliegen [vgl. Schönböck 2006]. Bis zur Version 2.0 nutzte auch das Graphical Modeling Framework JET für die Transformation der Modelle in den Editor-Code.

3.2.2 XPand – oAW

XPand ist die M2T-Sprache des o*penArchitectureWare Frameworks* (*oAW*). Das Framework wurde ursprünglich von der „b+m Informatik AG“ entwickelt und später als Open-Source Projekt zur Verfügung gestellt. Den Einstiegspunkt in die Generierung bildet dabei der oAW-Workflow, welcher in einer XML-basierten Sprache den Ablauf beschreibt. Hierbei lädt der Workflow die einzelnen für die Transformation benötigten Modelle. Für die Verarbeitung der Modelle werden, wie in JET, Templates genutzt. Diese Templates werden in einer eigenen, aber leicht verständlichen Sprache spezifiziert, welche einfache Konstrukte wie Schleifen oder bedingte Abfragen ermöglicht. Ein großer Vorteil von XPand ist es, dass komplexere Ausdrücke direkt an Java-Implementierungen weitergeleitet werden können. Hierzu wird mittels Extension-Dateien eine Verknüpfung zwischen den Sprachmitteln von XPand und eigenen Java-

[19] XPath – Abfragesprache für XML-Dokument; entwickelt vom W3C

Klassen geschaffen. Dies ermöglicht eine einfache und flexible Erweiterung des Sprachumfanges [vgl. Schnepel 2008, S.10]. Ein weiteres Argument für die Nutzung von XPand ist die Tatsache, dass ab Version 2.0 das GMF auf oAW für die Transformationen setzt. Um konform mit der Entwicklung innerhalb des Graphical Modeling Frameworks zu sein, werden XPand und oAW im Rahmen dieser Arbeit für die Model-to-Text-Transformationen genutzt.

3.3 Web Frameworks

Für die Entwicklung der Web-Komponenten des Systems ist es bei dem zu erwartenden Komplexitätsgrad unumgänglich ein Framework einzusetzen, welches die Erstellung der Anwendung erleichtert. Aus der Sicht des Autors ist dies sogar bei kleineren Anwendungen ratsam, da durch die Nutzung von Frameworks die Anwendungen leichter auf neue Anforderungen reagieren können. Da die Eclipse-Entwicklung auf Java basiert, ist es nur konsequent ein Java-basiertes Web-Framework zu wählen. Der Markt bietet dabei viele hervorragende Lösungen an. Die Apache Software Foundation offeriert mit *Struts 1.x*[20] ein sehr weit verbreitetes Framework, welches derzeit in der überarbeiteten Version 2.0 versucht den Erfolg des Vorgängers zu erreichen. Die stärkste Konkurrenz bekommt Struts durch die *Java Server Faces* (*JSF*[21]) Implementierung von Sun. Die Apache Foundation bietet allerdings ebenfalls mit *Shale*[22] und *MyFaces*[23] zwei Frameworks an, welche auf der *Java Server Faces* Spezifikation von SUN beruhen. Neben den genannten Frameworks könnten ebenfalls noch *Apache Tapestry*[24], *Spring MVC*[25] oder *Stripes*[26] als mögliche Grundlage für die Implementierung genutzt werden. Alle diese Frameworks besitzen je nach Anwendungszweck Vor- und Nachteile gegenüber den anderen Produkten. Ziel dieser Arbeit ist es aber nicht die einzelnen Eigenschaften dieser Projekte zu vergleichen und im Detail zu analysieren. Um trotzdem einen Überblick über die Bedeutung der einzelnen Projekte zu bekommen, wurde - inspiriert durch die Analyse von Matt Raible aus dem Jahr 2007 [vgl. Raible 2007] - eine Analyse über die Verbreitung der einzelnen Frameworks durchgeführt. Dafür kamen drei verschiedene Indikatoren zum Einsatz. Zum einen wurde das Bedürfnis des Marktes an diese Frameworks untersucht, indem

[20] http://struts.apache.org/
[21] http://java.sun.com/javaee/javaserverfaces/
[22] http://shale.apache.org/
[23] http://myfaces.apache.org/
[24] http://tapestry.apache.org/
[25] http://www.springsource.org/
[26] http://www.stripesframework.org

die Suchergebnisse von Stellenbörsen miteinander verglichen wurden. Zusätzlich wurde die angebotene Fachliteratur mengenmäßig verglichen.

Für die Analyse der Marktverbreitung sind die Online-Portale von *Monster.de*[27], *JobScout24*[28] und *Stepstone*[29] nach Jobangeboten durchsucht worden, welche Kenntnisse in den einzelnen Frameworks voraussetzen. Dabei zeigte sich ein starker Trend zu Struts und JSF. Die anderen Frameworks werden wesentlich weniger stark von der Industrie nachgefragt. In Abbildung 20 sind die Ergebnisse in einer Übersicht dargestellt

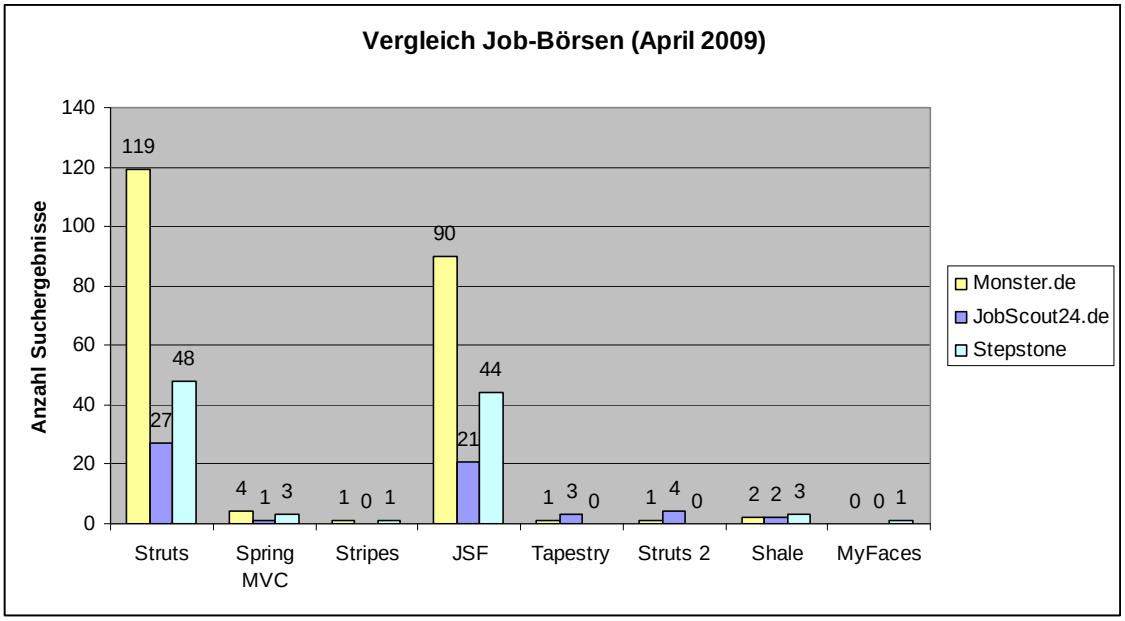

Abbildung 20 - Webframeworks - Vergleich Jobbörsen

Um zusätzlich die Nachfrage zu untersuchen und die angebotenen Kompetenzen der Fachkräfte beurteilen zu können, wurde auch auf der Kontakt-Plattform *XING*[30] nach Angeboten und Nachfragen gesucht. Da die Untersuchungen nicht mit dem Premium-Account von XING ausgeführt wurden, waren die maximalen Suchergebnisse auf 200 beschränkt. Dadurch lässt sich kein direkter Vergleich zwischen JSF und Struts herstellen. Es zeigt sich aber, dass diese beiden Frameworks trotzdem am stärksten nachgefragt werden und auch zu den Kompetenzen der meisten Personen gehören.

[27] *www.monster.de*
[28] *www.jobscout24.de*
[29] *www.stepstone.de*
[30] www.xing.de

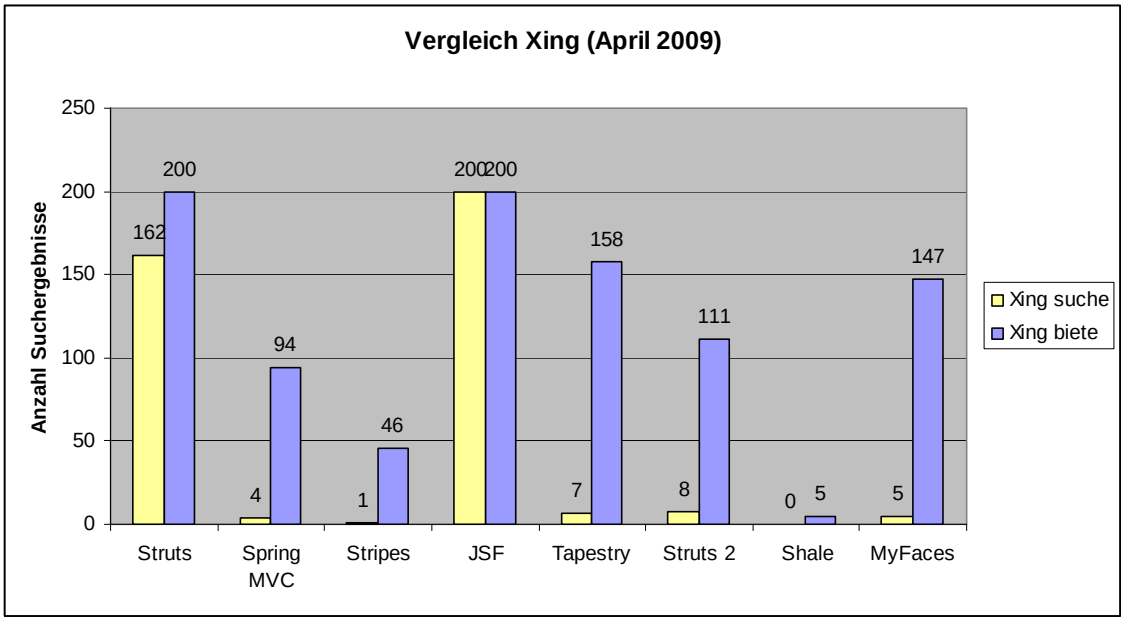

Abbildung 21 - Webframeworks - Vergleich Xing

Die anderen Frameworks scheinen wenig nachgefragt zu werden, wenngleich auch *Spring MVC*, *Tapestry*, *Struts* und *MyFaces* von einer größeren Anzahl von Entwicklern angeboten wird [vgl. Abbildung 21].

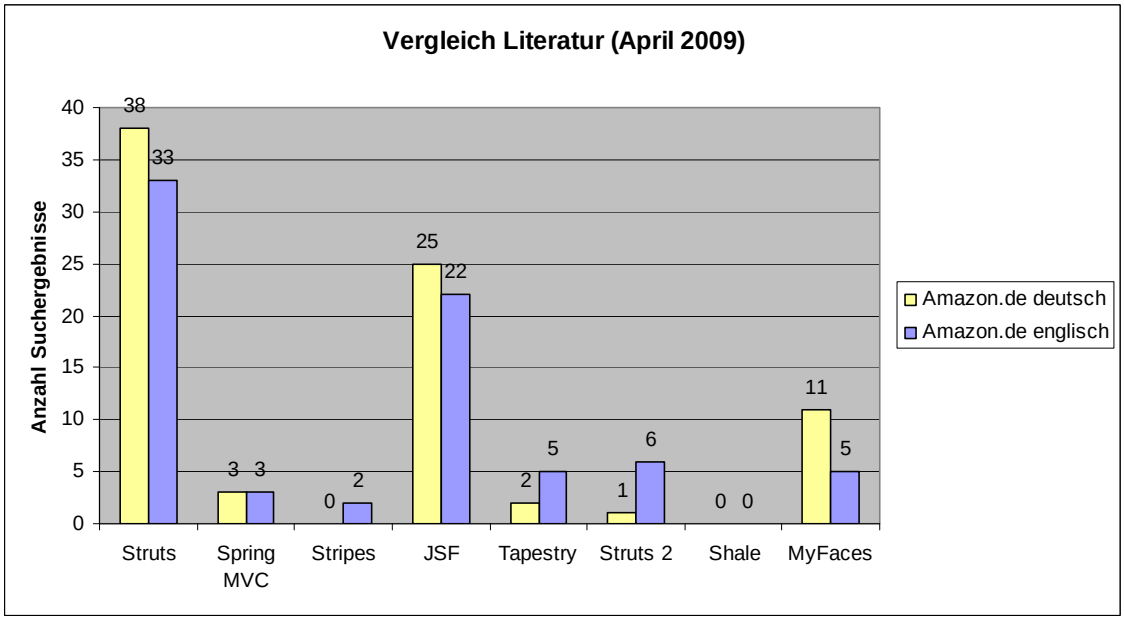

Abbildung 22 - Webframeworks - Vergleich Literatur

Die Analyse der Fachliteratur zeigt keine Abweichung zu den vorher ausgewerteten Ergebnissen. Auch hier lag das Struts Framework leicht vor JSF [vgl. Abbildung 22]. Betrachtet wurden hierbei die Suchergebnisse auf der Buch-Plattform *Amazon*[31]. Damit

[31] www.amazon.de

auch Neuerscheinungen in Englisch mit berücksichtigt werden, wurde die Suche sowohl für deutsch- als auch englischsprachige Werke ausgeführt.

Die eingesetzten Methoden erheben keinen Anspruch auf Vollständigkeit oder können eine genaue Aussage zu einem absoluten Favoriten treffen. Dazu wäre eine wesentlich intensivere Analyse erforderlich, als es im Rahmen dieser Arbeit möglich gewesen wäre. Sie zeigt aber eine gute Tendenz über die Verbreitung der einzelnen Produkte. Demnach finden *Struts* und *JSF* die meiste Marktverbreitung.

Für die Umsetzung der Web-Komponenten wurde auf Grund der stärksten Verbreitung das Apache Framework *Struts* gewählt, um den Entwicklungsprozess zu unterstützen. Da das zu entwickelnde System aber ein generatives Framework ist, welches dem Nutzer ein möglichst flexibles Umfeld für Erweiterungen bereitstellen möchte, ist mit dieser Wahl keine absolute Festlegung getroffen. Vielmehr stellt *Struts* nur den Rahmen für die prototypische Entwicklung. Angedacht ist es dem Entwickler in späteren Versionen die Wahl des Web-Frameworks zu überlassen. So könnten auch *JSF*, *Tapestry* oder andere Produkte ausgewählt werden, um die Web-Komponenten des Systems zu realisieren. Diese Entscheidung trifft dann der Entwickler zu dem Zeitpunkt, an dem der Server generiert wird.

3.4 Anwendungsszenarien

Hauptanwendung des Systems ist die kollaborative Nutzung von GMF-Editoren. Dabei ist es letztendlich dem Endanwender überlassen, in welcher Weise dies genutzt wird. Dies ist es auch abhängig von der eigentlichen Zielbestimmung des entworfenen Editors. Denkbare Szenarien könnten zum Beispiel auf dem Gebiet des verteilten Software-Engineerings oder des Softwareentwurfs liegen, bei dem Entwickler an verschiedenen Standorten zusammenarbeiten. Durch die vom System angebotene Web-Viewer Komponente, welche eine echtzeitbasierte Betrachtung der Arbeit mittels Webbrowser ermöglicht [vgl. 3.6.3], können Kunden oder externe Berater unabhängig von der lokalen Softwareumgebung, vorausgesetzt es ist ein Zugang zum Internet und ein Webbrowser vorhanden, das Projektgeschehen verfolgen.

Dies bietet sich besonders für Firmen an, deren Haupteinnahmequelle im Consulting besteht und deren Experten dementsprechend häufig auf Geschäftsreisen sind. Selbst ohne installiertes Eclipse ist es möglich sich an der Projektarbeit zu beteiligen. Durch das zur Verfügung gestellte Webinterface können dabei sogar Geräte eingesetzt werden, welche nicht für den Einsatz von Eclipse geeignet sind. So ist die Betrachtung des grafischen Modells auch auf mobilen Geräten möglich, wenn diese über einen

geeigneten Netzwerkzugang verfügen. Infolgedessen ist die Betrachtung des Modellbestandes unabhängig von Ort und Zeit.

Neben der kollaborativen grafischen Entwicklung ist aber noch eine Vielzahl weiterer Anwendungsbereiche möglich. Im Managementbereich können in Echtzeit Geschäftsprozesse, Workflows oder Organigramme modelliert werden. Dabei ist es sogar denkbar, dass direkt in den Ablauf des Workflows über einen entfernten Rechner eingegriffen werden kann. Prinzipiell sind also die Anwendungsmöglichkeiten so vielfältig wie die der generierten Editoren.

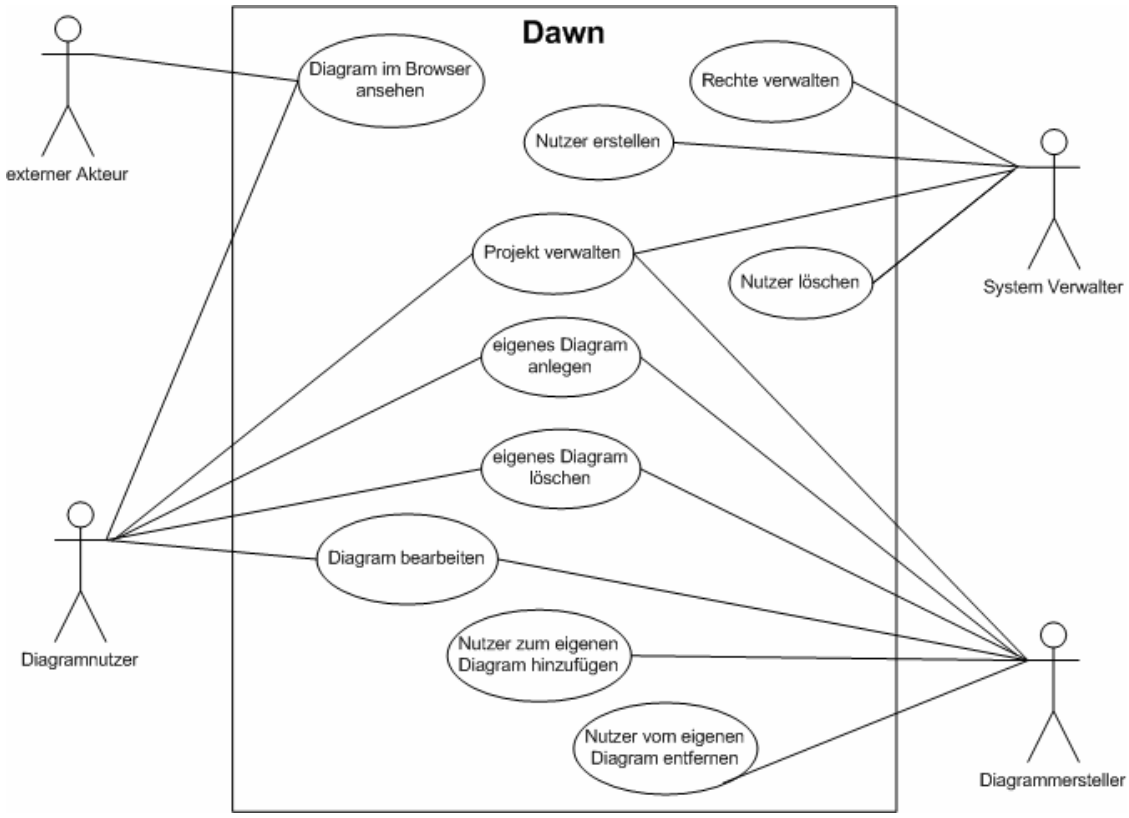

Abbildung 23 - Anwendungsfall: Überblick

Im System lassen sich vier Hauptakteure identifizieren – der Diagramnutzer, der Diagrammersteller, der externe Akteur und der System Verwalter. Die Rechte des externen Nutzers werden allein auf das Betrachten des Web-Viewers reduziert. Sie können also lediglich den aktuellen Projektzustand jederzeit im Browser betrachten, aber keinerlei Änderungen durchführen. Der Diagramnutzer kann ebenfalls das Diagramm im Browser betrachten. Zusätzlich verfügt er aber über alle Rechte, um das Diagramm innerhalb von Eclipse über den GMF-Editor bearbeiten, anlegen oder löschen zu können. Neben diesen Aktionen kann der Nutzer, der ein Diagram erstellt hat zusätzlich noch weitere Nutzer dem Diagram hinzufügen, beziehungsweise diese wieder entfernen. Der Systemverwalter kann alle Operationen der drei vorangehenden

Nutzer ausführen. Weiterhin besitzt er aber die Rechte zur Verwaltung des Systems. Hierzu gehören das Anlegen und Löschen von Nutzern und das Verwalten der Rechte des Systems. Diese beschriebenen Rollen können noch durch ein detailliertes Rechtesystem eingeschränkt werden. In Abbildung 23 werden diese Zusammenhänge dargestellt. Um die Übersichtlichkeit nicht zu verlieren, wurde der Systemverwalter nicht mit allen Anwendungsfällen verknüpft. Der Systemverwalter besitzt aber die volle Rechte im System und darf alle Aktionen ausführen.

3.5 System Architektur

Das System soll als Server-Client Architektur konzipiert werden. In Anbetracht der Tatsache, dass Funktionalitäten wie Rechteverwaltung und Web-Komponenten angeboten werden sollen, wäre es wenig sinnvoll, eine Peer-to-Peer Architektur zu verwenden, in denen jeder Editor gleichzeitig als Server und Client agiert. Würden beispielsweise alle Clients abgeschaltet, wäre auch das Web-Frontend nicht mehr zugänglich. Aus diesem Grund wird das System über eine steuernde Instanz verfügen, an der sich die Klienten anmelden können. Aufbauend auf diesem Prinzip muss der Editor um eine Netzwerksschnittstelle erweitert werden, die die Verbindung zu dem Server aufnehmen und eine Synchronisation mit dem Datenbestand des Servers durchführen kann. Neben dieser Einheit muss der GMF-Editor zusätzlich noch um Konfigurations- und Administrationselemente erweitert werden. Dies umfasst sowohl Dialoge und Wizards um neue Projekte anlegen zu können, als auch GUI-Elemente, welche u.a. den Verbindungszustand zum Server anzeigen können.

Abbildung 24 zeigt eine vereinfachte Systemdarstellung der zu entwickelnden Komponenten (gelb dargestellt).

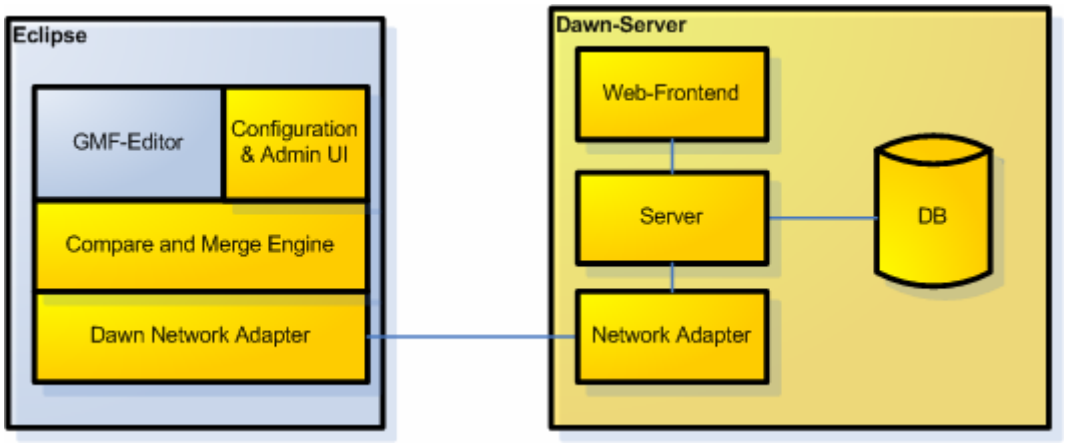

Abbildung 24 - vereinfachte Systemdarstellung

3.6 Systemanforderungen

Aufbauend auf dem Architekturmodell sollen im Folgenden die funktionalen Aspekte des Systems analysiert und diesbezüglich Grundlagen für die Designentscheidungen getroffen werden. Die Ergebnisse dieser Analyse werden in dem in Anhang beschrieben Anforderungskatalog zusammengefasst [vgl. Anhang A]. Um den Umfang der Arbeit begrenzen und abgrenzen zu können, werden ebenfalls die für die Arbeit nicht notwendigen aber wünschenswerten Kriterien aufgelistet. Ebenso werden Abgrenzkriterien bestimmt, welche nicht im Design- und Implementierungsumfang enthalten sind.

3.6.1 Kommunikation und Firewall-Transparenz

Ziel des Systems ist es jederzeit schnell und flexibel einsetzbar zu sein. Für den Anwender soll es keinen Unterschied machen, ob es in einem lokalen Netzwerk (*Local Area Network, LAN*) oder im Internet genutzt wird. Aus diesem Grund muss das System möglichst resistent gegenüber Firewall-Konfigurationen sein, die den Übergang von einem Netzwerk zu einem anderen sichern. Befände sich zum Beispiel ein Client in einem Firmennetzwerk, der Server aber im Internet, so könnte eine mögliche Firewall-Einstellung die Kommunikation über einen proprietären Port verhindern. Aus diesem Grund soll das System mindestens über das HTTP-Protokoll (*Hypertext Transfer Protocol*) kommunizieren können. Dieses wird in der Regel nicht durch Firewalls blockiert, sodass die Kommunikation jederzeit aufgebaut werden kann. Dementsprechend bietet es sich an, bereits vorhandene Kommunikations-Technologien zu nutzen, die über HTTP kommunizieren wie das *Simple Object Access Protocol* (*SOAP*), *XML-RPC* oder *Representational State Transfer* (*REST*). Diese bieten gegenüber einer eigenen Lösung den Vorteil, dass sie ausgetestet sind und stabil laufen. Dies hat allerdings zur Konsequenz, dass der Client keine permanente Verbindung zum Server aufnehmen kann. Das daraus resultierende Polling[32] des Clients hat zum Nachteil, dass der Server Änderungen nicht sofort allen Systemteilnehmern mitteilen kann. Möchte ein Client seinen Systemzustand aktualisieren, so muss er diesen beim Server erfragen. Dadurch erhöht sich auch die Netzlast. Es müssen also geeignete Mechanismen entwickelt werden, damit die Netzlast so klein wie möglich gehalten wird, also nur die wirklich benötigten Daten übertragen werden.

[32] Polling – zyklisches Abfragen eines Systemzustandes

3.6.2 Ausfallresistenz, Mobilität und Netzwerkunabhängigkeit

Neben der Erreichbarkeit des Servers, auch durch Firewall-Systeme hindurch, soll sich das System auch resistent gegenüber dem Wegfall der Serverkomponente verhalten. Für den Nutzer soll eine Trennung vom System, zum Beispiel durch Ausfall des Servers oder des Netzwerkes, ohne große Probleme verlaufen. Im idealen Fall bekommt der Nutzer den Wegfall anderer System-Komponenten gar nicht mit. Arbeitet er gerade an einem Editor, welcher über das zu entwickelnde System mit anderen Editoren verbunden ist, und bricht die Netzwerkverbindung ab, so soll diese unbemerkt vom Nutzer wieder aufgebaut und der Modellbestand synchronisiert werden können. Dies ist besonders wichtig, wenn der Editor in einem mobilen Umfeld genutzt wird, bei dem die Netzwerkverbindung nicht immer kontinuierlich aufrechterhalten werden kann. Bei Verbindungen über *WLAN* (*Wireless Local Area Network*), *GPRS* (*General Packet Radio Service*) oder *UMTS* (*Universal Mobile Telecommunication System*) kann es immer systembedingt zu kurzzeitigen Ausfällen kommen, beispielsweise wenn aufgrund einer längeren Tunnelfahrt die Verbindung zur Basisstation abbricht.

Neben systemabhängigen kurzen Ausfallzeiten soll ein Editor auch in der Lage sein komplett ohne Netzwerkverbindung arbeiten zu können. Hierbei sollen zum Beispiel Anwendungsszenarien in Betracht gezogen werden, bei denen der Nutzer nicht in der Lage ist seinen Client mit einem Netzwerk zu verbinden. Um in einem solchen Fall Ausfallzeiten in der Entwicklung zu verhindern, muss das System auch in der Lage sein ohne Verbindung zum Server lauffähig zu sein. In diesem Fall müssen alle Änderungen protokolliert und mit dem Server synchronisiert werden, sobald der Nutzer wieder mit dem Netzwerk verbunden ist.

3.6.3 Web-Viewer und Systemverwaltung

Wie in Kapitel 1.2 beschrieben, soll das System über eine Web-Oberfläche den Modellbestand auch unabhängig von Eclipse darstellen können. Hierzu muss eine Möglichkeit geschaffen werden, um die grafischen Elemente des GMF-Editors als Web-Element darstellen zu können. Dies ist besonders schwierig, da das Aussehen des GMF-Editors von den Endanwendern bestimmt wird, also im Vorhinein nicht bekannt ist. Es müssen also über ein generisches, adaptives Verfahren die spezifizierten View-Elemente umgewandelt werden. Da das System sowohl den Server als auch den Netzwerkadapter mittels modellgetriebener Techniken erzeugt [vgl. 3.6.6], bietet es sich an, auch die Web-Elemente in diesem Vorgang zu generieren. Dabei kann auf das gmfgraph-Modell zugegriffen werden, welches alle Informationen für das Aussehen der

View-Elemente besitzt. Eine generative Erzeugung hat gegen eine dynamische Generierung der einzelnen Views zur Laufzeit den Vorteil, dass das System wesentlich performanter agieren kann. Würden alle Elemente des grafischen Modells zur Laufzeit erzeugt und dargestellt werden, so könnte es bei sehr großen Modellbeständen zur erheblichen Verzögerungen bei der Anzeige im Browser kommen.

Neben der Web-Viewer-Komponente, die das grafische Modell darstellt, soll aber auch mittels Web-Frontend die Verwaltung der Projekte auf dem Server und der Nutzer vorgenommen werden können. Ein Web-Frontend für die Serververwaltung hat den Vorteil, dass administrative Tätigkeiten losgelöst von Eclipse erfolgen können. Es ist also von jedem Ort mit Netzwerkzugang möglich Projekteinstellungen zu verändern, Nutzer zu entfernen oder Rechte Einzelner zu beschränken bzw. zu erweitern.

3.6.4 Synchronisation und Ressourcenverwaltung

Über die Serverkomponente sollen die einzelnen Eclipse-Editoren miteinander kommunizieren können. Änderungen am Modellbestand eines Clients, zum Beispiel das Erstellen eines neuen Knotens, sollen sich unmittelbar auf alle anderen Nutzer übertragen. Dies erfordert, dass das System eine Möglichkeit besitzt, um potentielle Konflikte zu erkennen. Es muss also möglich sein, dass wenn mehrere Nutzer gleichzeitig dasselbe Objekt bearbeitet haben, das System diesen Konflikt erkennt und verhindert, dass das Diagramm in einem nicht konsistenten Zustand gerät. Hierfür muss eine Möglichkeit geschaffen werden die Objekte des Modellbestandes eindeutig identifizieren zu können. Aufgetretene Konflikte müssen für den Nutzer eindeutig erkennbar visualisiert werden. Ferner muss es eine Möglichkeit geben diese Konflikte durch den Benutzer wieder auflösen zu lassen. Im Allgemeinen kann dabei die Änderung der in den Konflikt geratenen Partei nur verworfen werden. Das System soll darüber hinaus aber auch die Möglichkeit besitzen Änderungen der anderen Partei zurückzusetzen und seinen eigenen Modellbestand als gültig zu erklären. Dieses Vorgehen hat den Vorteil, dass Konflikte wesentlich flexibler gelöst werden können. Allerdings bedarf dieses Vorgehen einer eindeutigen Kommunikation zwischen den Projektteilnehmern oder ein geregeltes Nutzermanagement, um zu verhindern, dass Nutzer gegenseitig ihren Projektbestand zurücksetzen [vgl. auch 3.6.5].

Neben der Erkennung von Konflikten soll das System auch über eine Möglichkeit verfügen Konflikte zu vermeiden. Dies kann durch eine exklusive Schreibberechtigung auf einen Teil des Modellbestandes erfolgen. Vorteile dieses expliziten Lockings ist es, dass der Modellbestand auch nicht durch versehentliche Operationen geändert werden kann.

3.6.5 User-Management und Sicherheit

Da der Anwendungszweck abhängig von den zugrunde liegenden Editoren ist, kann es auch geschehen, dass das System in Projekten Einsatz findet, in denen der Zugriff auf diese einzelnen grafischen Modelle geschützt sein muss. Folglicherweise muss ein Nutzermanagement zur Verfügung gestellt werden, welches neben den Zugriffsrestriktionen auch eine flexible Handhabung der Rechte der einzelnen Benutzer ermöglicht. Da der Quellcode erweiterbar sein soll, um spätere Anwendungen einfach zu ermöglichen, muss hierbei ein stark generisches Nutzerkonzept entwickelt werden, welches sich einfach an die Besonderheiten der einzelnen Editoren anpassen lässt. Möchte ein Entwickler beispielsweise das Recht zum Löschen eines bestimmten Kanten-Typs steuern können, muss diese Erweiterung einfach in sein Projekt integrierbar sein.

Durch die starke Fokussierung der Wirtschaft auf externe Berater ist es dabei von Wichtigkeit die Informationsvielfalt beziehungsweise die Rechte der an dem verteilten Projekt arbeitenden Fachkräfte einschränken zu können. Jedes Projekt muss folglich über einen Administrator verfügen, der den Zugang zu dem Modellbestand regelt.

3.6.6 Generierung der Komponenten

GMF generiert den Quellcode für den Editor des Anwenders aus Modellspezifikationen. Das System soll ebenfalls mit Hilfe von Modellen die Netzwerkerweiterung generieren können. Über geeignete Modelle soll dem Endanwender hierbei die Möglichkeit gegeben werden Einfluss auf den generierten Code zu nehmen. Hierbei soll sich die Codegenerierung nahtlos in den Entwicklungsprozess für GMF-Editoren eingliedern, also für GMF-Entwickler intuitiv anwendbar sein.

3.6.7 Anforderungen an das prototypische Diagramm

Im Rahmen der Arbeit soll zur Demonstration der umgesetzten Konzepte ein vereinfachtes Diagramm erstellt werden. Dieses Diagramm wird in Anlehnung an ein Klassendiagramm der UML modelliert. Die grundlegenden Elemente von Klassendiagrammen sind in der Regel bekannt oder intuitiv erlernbar. Trotzdem besitzen Klassendiagramme durch Operationen, Attribute und verschiedene Verbindungstypen ein reichhaltiges Repertoire an Elementen, um die Möglichkeiten des Systems demonstrieren zu können. Das Diagramm soll aber nicht alle Funktionen eines Klassendiagramms nachstellen, sondern nur in Anlehnung an ein solches kreiert

werden. Dabei können Konzepte innerhalb des Diagramms von denen eines UML-Klassendiagramms abweichen.

3.6.8 Systemumgebung

Das System wird mit Hilfe von Java in der Version 1.5 implementiert. Basis für die Entwicklungen bilden die zur Beginn der Arbeit aktuellen Versionen von EMF und GMF (EMF 2.4.0; GMF 2.1.0). Als weiterer Rahmen wird die Eclipse IDE 3.4.0 im Ganymede Release genutzt. Die Entwicklungen finden ausschließlich unter Windows XP SP2 statt. Dies ist auch das Betriebssystem unter dem die Anwendung getestet ist, wenngleich auch das System unter anderen Betriebssystemen, welche die Java Virtual Machine anbieten, lauffähig sein sollte. Als Web-Container wird Apache Tomcat 6.0.18 in Verbindung mit Axis 1.3 für die SOAP-basierte Kommunikation genutzt. Die Persistierung erfolgt in MySQL 3.21 als Datenbankbetriebssystem.

3.7 Wunschkriterien

Neben den bereits spezifizierten und umzusetzenden Kriterien sind in dieser Arbeit eine Vielzahl weitere Entwicklungen denkbar. Um den Zeitrahmen begrenzen zu können, werden diese im Folgenden als Wunschkriterien aufgelistet. Diese sollen dann umgesetzt werden, wenn der zeitliche Fortschritt der Arbeit dies zulässt. Zusätzlich können sie als Anregungen für zukünftige Erweiterungen dienen.

Damit eine übersichtliche Versionierung der einzelnen grafischen Modellbestände vorgenommen werden kann, wäre es denkbar eine Historie über den Zustand des Modells anzufertigen. Durch eine solche Verwaltung könnte die Entwicklung eines Projektes einfach nachvollzogen werden.

Im Rahmen dieser Arbeit werden nur Ressourcen betrachtet, die EMF und GMF in einer Datei speichern [vgl. 3.8]. Möglich ist allerdings die Betrachtung einer getrennten Speicherung beider Modelle.

Zusätzlich könnte in das System noch eine Chat-Funktionalität integriert werden, über welche die Nutzer direkt kommunizieren und Informationen zum aktuellen Entwicklungsstand austauschen könnten. Eine Auflistung aller Wunschkriterien findet sich unter [Anhang A]

3.8 Abgrenzungen

Das System wird ausschließlich für GMF-basierte Editoren entwickelt und konzipiert. Dies schließt nicht aus, dass die entwickelten Komponenten so generisch gestaltet werden, dass sie auch in anderen Zusammenhängen nutzbar sind. Fokus der Arbeit liegt allerdings allein auf der Netzwerkerweiterung für GMF-Editoren. Zudem wird die Implementierung ausschließlich für GMF-Editoren entwickelt, welche innerhalb der Eclipse-IDE laufen. GMF-basierte Editoren innerhalb von eigenen RCP-Anwendungen werden nicht berücksichtigt. Auch hier ist es möglich und auch wahrscheinlich, dass die Ergebnisse ebenfalls für RCP-Anwendungen einsetzbar sind.

Der Einfachheit halber werden innerhalb der Arbeit nur Editoren betrachtet, die das EMF und das GMF-Modell in einer Ressource speichern. Weiterhin wird die gesamte Entwicklung ausschließlich auf *Java SE* (*Standard Edition*) in der Version 1.5 und nachfolgenden Versionen ausgeführt. Sowohl kleine Versionen als auch die *Micro Edition* (*ME*) oder die *Enterprise Edition* (*EE*) der Virtuellen Maschine von Java werden nicht als Entwicklungsgrundlage genutzt. Eine Auflistung aller Abgrenzungskriterien findet sich unter [Anhang A]

3.9 Anforderungskatalog

Die in den vorangegangen Kapiteln erfassten Anforderungen sind in einem übersichtlichen Anforderungskatalog zusammenfasst und in einzelne Aufgaben aufgeschlüsselt worden [vgl. Anhang A]. Anhand dieses Katalogs kann eine sorgfältige Planung des Projektes und des zeitlichen Umfangs vorgenommen werden, indem für die einzelnen Aufgaben die Aufwände abgeschätzt und in die Zeitplanung übernommen werden. Während der Entwicklung dient dieser Anforderungskatalog als Grundlage und Leitfaden für die Konzeptionierungen und Umsetzungen.

4 Konzeptentwicklung und Systemdesign

Nach der Analyse der Systemanforderungen legt das folgende Kapitel die konzeptuellen Grundlagen für die Implementierung. Es beschäftigt sich mit der Entwicklung von Lösungsansätzen der in Kapitel 3 geforderten Anforderungen an das System. Hierbei werden die Konzepte erarbeitet, welche die Implementierung und Umsetzung des Systems ermöglichen.

Basis jeder modellgetriebenen Entwicklung ist die Erstellung eines Prototyps, welcher als Muster für die Entwicklung der Generatoren fungiert. Aus diesem Grund werden zuerst alle Systemspezifika erläutert, die sich mit der Entwicklung und Konzeptionierung des Prototyps beschäftigen.

Um einen Gesamtüberblick über das System zu geben, wird in Kapitel 4.1 die Systemarchitektur und die in ihr enthaltenen Komponenten erläutert. Ausgehend von dieser Basis erfolgt im anschließenden Kapitel die Beschreibung der wesentlichen Konzepte der Kommunikation. Dies beinhaltet sowohl die Entwicklung des Übertragungsverfahrens, als auch ein Entwurf für die Beschaffenheit der zu übertragenden Daten. Nachdem festgelegt ist, wie Client und Server Informationen austauschen können, werden Mechanismen entwickelt, um den Datenbestand auf dem Server und dem Client mit den gesendeten Informationen abgleichen zu können. Dies umfasst das Erkennen und Zusammenfügen von Unterschieden. Dabei können systembedingt Konflikte auftreten. Die Erkennung, Behandlung und Vermeidung dieser Konflikte wird in Kapitel 4.4 thematisiert. Nachdem in diesen Kapiteln die Grundlage für die weitere Entwicklung des Systems geschaffen wurde, wird in Kapitel 4.5 das in 3.6.2 geforderte Feature diskutiert. Hierbei werden Probleme der Netzwerkunabhängigkeit und Lösungsansätze aufgezeigt.

Im Anschluss verlagert sich der Fokus stärker auf den Server. Es werden die Persistierung der Daten, die Verwaltung des Systems und die Darstellung der Diagramme mittels Web-Viewer thematisiert. Ferner werden die Technologien und Konzepte für die Rechteverwaltung des Systems entwickelt und dargelegt.

Nachdem das System soweit konzeptioniert wurde, dass es durch eine Implementierung in einen lauffähigen Zustand überführt werden kann, wird in Kapitel 4.11 die modellgetriebene Generierung der einzelnen Komponenten veranschaulicht. Dies beinhaltet sowohl die Beschreibung der zu generierenden Komponenten, als auch des

sich darauf stützenden Datenmodells. Abbildung 25 gibt einen Überblick über die einzelnen Aufgaben, welche im Rahmen des Projektes zu bewältigen sind.

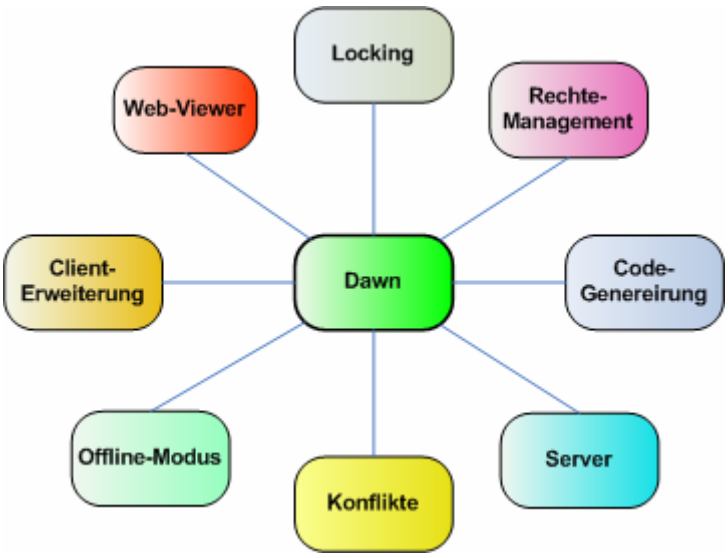

Abbildung 25 - Überblick über die einzelnen Sub-Projekte

Das System wurde, in Analogie zu sonnenbasierten Ereignissen (Eclipse = Sonnenfinsternis), auf den Namen *Dawn* (Dämmerung) getauft. Im Folgenden werden die Begriffe *System* und *Dawn* synonym verwendet.

4.1 Systemarchitektur

In Kapitel 3.5 wurden die Gründe für die Umsetzung als Server/Client-System analysiert und diskutiert. Um die Architektur von Dawn entwickeln zu können, müssen mehrere Komponenten betrachtet werden. Zum einen bildet ein generierter GMF-Editor die Grundlage der clientseitigen Entwicklung. Dieser muss über eine Erweiterung in die Lage versetzt werden, eine Verbindung zu dem Dawn-Server aufnehmen zu können. Gleichzeitig muss diese Erweiterung die Synchronisation der Datenbestände vornehmen. Zum anderen ist es erforderlich, dass serverseitig ebenfalls Synchronisierungen vorgenommen und die Daten persistiert werden.

Das in Abbildung 26 dargestellte Verteilungsdiagramm erweitert den groben Systemüberblick aus Abbildung 24. Dabei agieren die einzelnen Komponenten ausschließlich über Schnittstellen, sind also folglich lose gekoppelt und austauschbar.

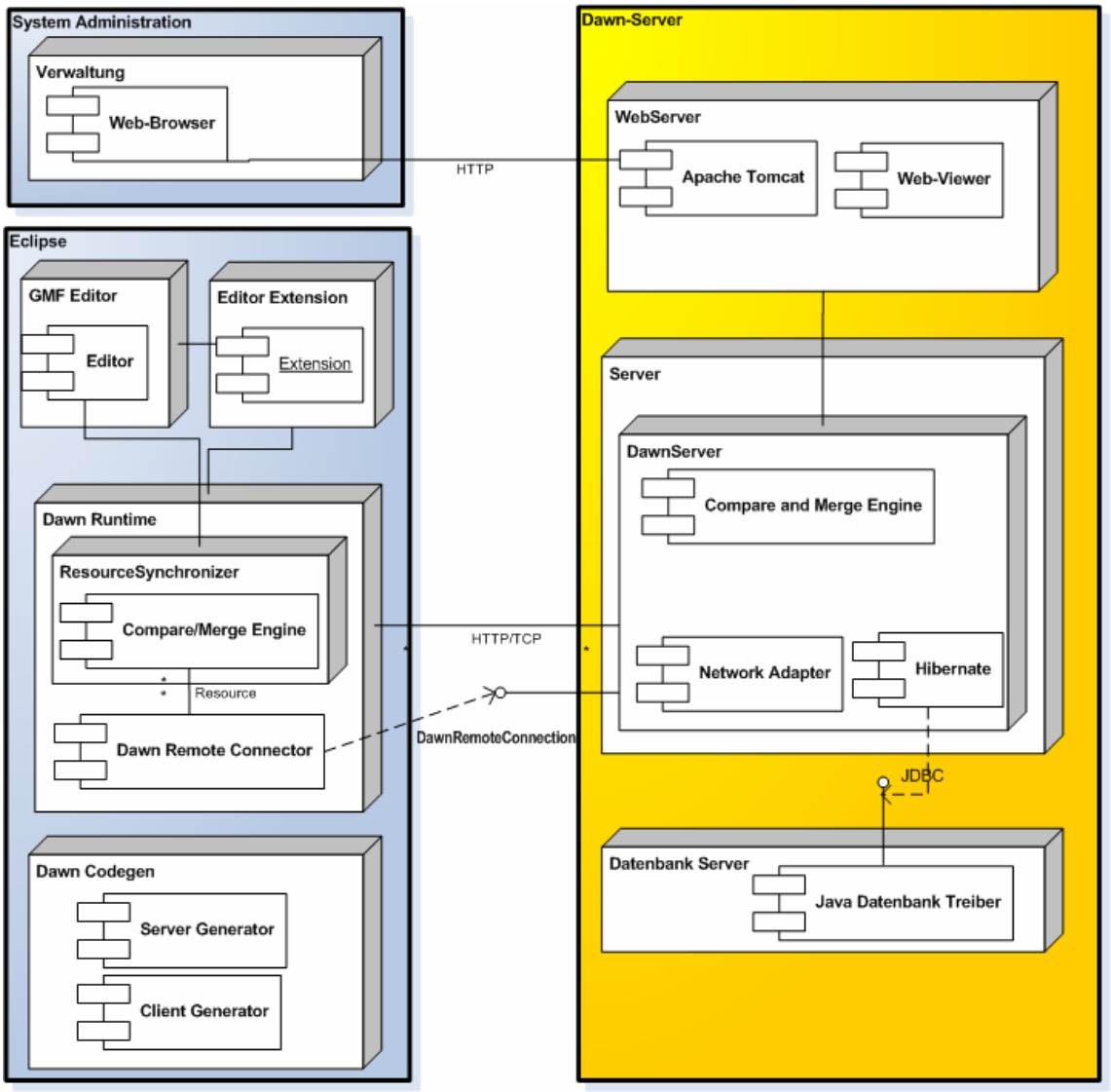

Abbildung 26 - Verteilungsdiagramm des Systems

Auf der Client-Seite wird der vorhandene Editor durch zwei Komponenten erweitert. Einerseits durch die *Dawn-Runtime*, einer globalen Schnittstelle, welche für alle Editoren eine Ablaufumgebung für die Netzwerkverwaltung und Kommunikation zur Verfügung stellt, und andererseits durch zusätzliche editorspezifische Erweiterungen, wie Wizards und Menüs, mit denen die Verwaltung von Dawn möglich ist. Die Runtime besteht unter anderem aus einer Verbindungsschnittstelle zum Server und einer Synchronisationskomponente, welche für das Zusammenfügen der Ressourcen verantwortlich ist. Hierbei werden die erkannten Änderungen im Editor zur Ansicht gebracht. Dies umfasst auch das Erkennen und Markieren von Konflikten. Eine ausführliche Beschreibung der Synchronisations- und Konflikterkennungsmechanismen erfolgt in den Kapiteln 4.3 und 4.4.

Die clientseitige Editor-Erweiterung wird grundlegend genutzt, um die spezifischen Besonderheiten jedes GMF-Editors an die Dawn-Runtime zu übertragen und diese mit den Eigenschaften des Editors zu initialisieren. Dabei nimmt die Erweiterung keinerlei Änderungen am bestehenden Editor vor, sondern ergänzt diesen nur um Informationen. Aus diesem Grund kann der GMF-Editor auch jederzeit losgelöst von Dawn genutzt werden. Im Kapitel 4.9 wird der Zusammenhang zwischen dem Editor-Plugin, der Erweiterung und der Dawn-Runtime detailliert dargestellt.

Der Dawn-Server stellt verschiedene Adapter für die Kommunikation mit der clientseitigen Netzwerkschnittstelle (**`DawnRemoteConnector`**) zur Verfügung. Clients können also unterschiedliche Netzwerktechnologien und -protokolle (z.B. RMI, SOAP, REST) benutzen. Alle diese Schnittstellen müssen allerdings das Interface **`DawnRemoteConnection`** implementieren, welches alle Methoden zur Kommunikation mit dem Server zur Verfügung stellt. Das **`DawnRemoteConnection`** Interface beschreibt dabei die Informationen, die zwischen dem Server und dem Client ausgetauscht werden. Basierend auf dieser Spezifikation können beliebige Implementierungen vorgenommen werden. Hierbei ist zu beachten, dass der Dawn-Server auch eigenständig ohne Web-Kontext lauffähig ist. In diesem Fall verliert er aber die Möglichkeit der Administrierung durch eine Web-Oberfläche. In Kapitel 4.2 wird dieser Kommunikationsmechanismus ausführlich beschrieben. Der Server verfügt ebenfalls über eine Komponente, welche es ermöglicht Änderungen am Modellbestand mit dem Serverbestand zu synchronisieren. Änderungen werden über das objektrelationale Mapping-Framework *Hibernate* in eine Datenbank persistiert. Vorteil der Nutzung von Hibernate ist die einfache Handhabung im objektorientieren Kontext und eine einfache Austauschbarkeit der Datenbank. Somit ist es jederzeit möglich das Datenbankbetriebssystem zu wechseln, ohne in das Architekturmodell eingreifen zu müssen. Der Dawn-Server wird in den Kontext eines Web-Servers integriert, über den die Kommunikation mit webbasierten Protokollen zur Verfügung gestellt wird.

Neben diesen Protokollen kann mittels einer webbasierter Oberfläche der Server auch administriert werden. Hierbei greift der Servlet-Kontext direkt auf die vom Server bereitgestellten Schnittstellen zu. Ebenfalls über diese Schnittstellen kann der Webserver die Diagramminformationen in einem Browser darstellen. Im Kapitel 4.8 werden die spezifischen Einzelheiten der Web-Komponenten näher erläutert.

Der Server verwaltet alle Diagramm-Informationen in einer eigenen Einheit, die sich *Projekt* nennt. Projekte fügen jedem Diagramm serverseitig zusätzlich Meta-Informationen hinzu. So sind diese für die Betrachtung der Nutzer und Rechte und

ebenfalls für die serverseitige Speicherung der Diagramminformationen zuständig [vgl. Abbildung 27].

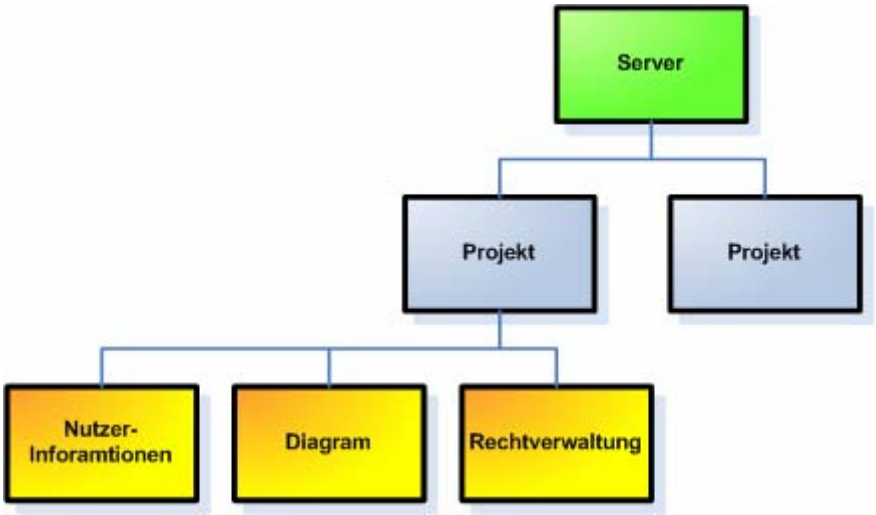

Abbildung 27 - Projekte

In der derzeit spezifizierten Version besteht eine eins-zu-eins Beziehung zwischen Diagramm und Projekt. Später kann aber durchaus ein Projekt für die Verwaltung mehrerer kontextzusammenhängender Diagramme genutzt werden. Ein Projekt stellt also den serverseitigen Container für die gesamte Behandlung von Diagrammen dar.

Neben den Client- und Server-Komponenten beinhaltet Dawn noch ein weiteres System, das *Dawn Codegen*, welches für die Generierung der anderen Komponenten verantwortlich es. Zusätzlich zur eigentlichen Generierung stellt es einen Editor zur Bearbeitung des Generator-Models und diverse Menü-Punkte für eine flexible Generierung zur Verfügung. Die Details des *Dawn Codegen* werden in den Kapiteln 4.11 und 5.9 thematisiert.

4.2 Kommunikationsstruktur

Wie in Kapitel 3.6.2 dargelegt, ist es ein Ziel von Dawn unabhängig von Firewall-Restriktionen agieren zu können. Hierzu muss ein webbasiertes Protokoll genutzt werden. Dabei stünden gängige Protokolle wie SOAP oder REST zur Verfügung. Alternativ könnte auch eine eigene Implementierung in Erwägung gezogen werden. SOAP und XML-RPC bieten den Vorteil, dass sie stark durch gängige Frameworks (z.B. Apache Axis) unterstützt werden. Sind nur einfache Datentypen zu übertragen, kann die gesamte Zugriffsschicht generiert werden. REST kann gegenüber SOAP einen Geschwindigkeitsvorteil bieten, da SOAP teilweise stark aufgebläht ist. Wiederum spezifiziert REST ausschließlich die Übertragungsmethode, lässt aber die

implementierungs-spezifischen Details des Protokolls offen. Dem gegenüber stehen performantere, nicht webbasierte Architekturen wie Java RMI. Um eine flexible Struktur zu schaffen, in welcher der Nutzer nicht auf eine einzige Technologie beschränkt ist, wurde ein generischer Ansatz für die Kommunikationsstruktur entwickelt – die Netzwerkadapter.

4.2.1 Netzwerkadapter

Dawn unterstützt unterschiedliche Protokolle. So kann das System über SOAP oder REST, aber auch über nicht webbasierte Technologien wie RMI oder gar eigene Entwicklungen kommunizieren. Grundlage hierfür bilden Netzwerkadapter, die die Kommunikation von der Verarbeitungsschicht abstrahieren. Jede Netzwerkadapter-Implementierung muss das Interface `DawnRemoteConnection` implementieren. Dadurch wird sichergestellt, dass die eigentliche Implementierung hinter der Schnittstelle versteckt werden kann. Diese Schnittstelle repräsentiert eine spezielle Verbindung zwischen Client und Server. Für den Client ist es also transparent, welche Implementierung er nutzt, also über welchen Kommunikationskanal die Daten übertragen werden. Abbildung 28 stellt den Aufbau der Kommunikationsschicht dar. Die für das Transportmedium notwendigen Komponenten sind grün dargestellt.

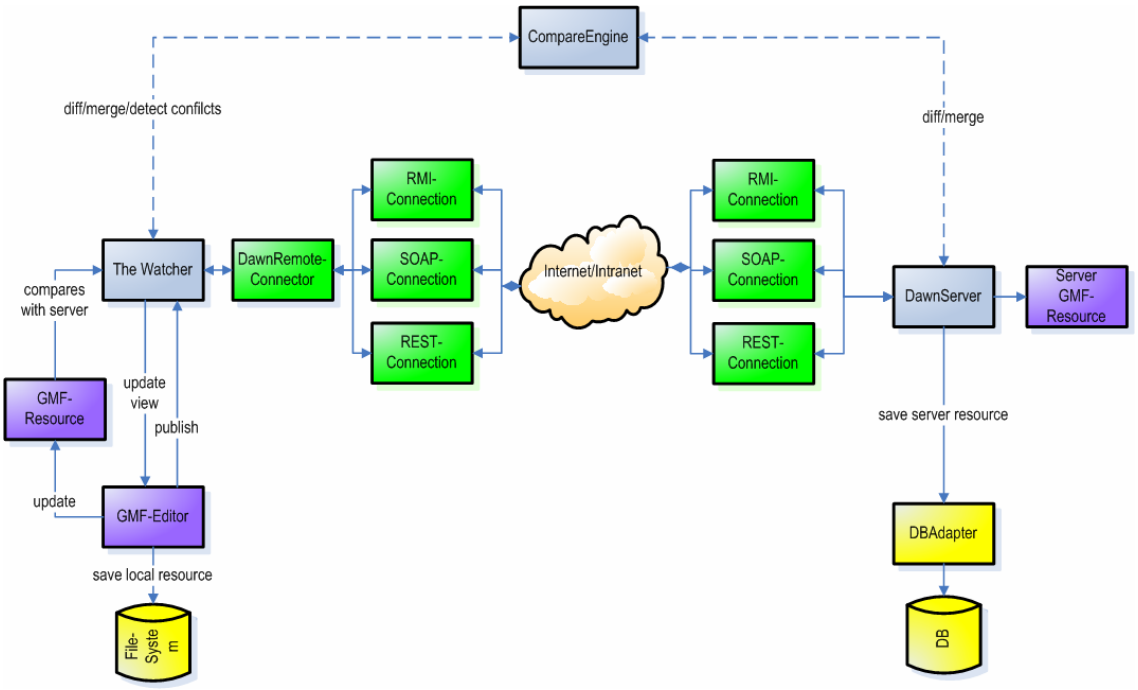

Abbildung 28 - System Design

Auf der Clientseite instanziiert die Komponente **`DawnRemoteConnector`** eine spezifische Remote-Connection Instanz. Welche Implementierung genutzt wird, kann über System-Parameter festgelegt zur Laufzeit verändert werden. Im Sinne des Fabrik-Entwurfsmusters, bei dem mit Hilfe einer Fabrik-Klasse ein konkretes Objekt erschaffen wird [vgl. Daum 2007, S.194 ff.], übernimmt der DawnRemoteConnector die Aufgabe der erzeugenden Fabrik. Da er aber noch zusätzliche Funktionen, wie die Prüfung der Netzwerkverbindung übernimmt, ist seine Namensgebung leicht abweichend von den für Fabriken üblichen Konventionen. Möchte der Client mit dem Server kommunizieren, stellt er eine Anfrage an den DawnRemoteConnector, welcher die passende Remote-Connection Implementierung liefert, also den aktuellen Kommunikationskanal zur Verfügung stellt. Durch dieses Konzept wird es dem Client ermöglicht, zur Laufzeit auf eine andere Netzwerktechnologie zu wechseln. So könnte aus Performancegründen der Nutzer in einer offenen Umgebung auf RMI als Transportmedium setzen, sobald sein Client sich aber in ein restriktiveres Netzwerk bewegt, könnte er auf eine webbasierte Lösung (z.B. SOAP) wechseln, um so Beschränkungen zu umgehen. Dawn stellt in der prototypischen Implementierung Remote-Connections für SOAP und RMI zur Verfügung. Ferner ist es aber auch möglich eigene Protokolle zu implementieren.

Um zur Laufzeit die Remote-Connection clientseitig wechseln zu können, muss der Server alle zur Verfügung stehenden Adapter parallel anbieten. Der Server hält also alle Kommunikationskanäle zur Laufzeit offen. Dieses Design erlaubt es flexibel unterschiedlichste Kommunikationswege für die Clients zu ermöglichen. Dadurch können die Nutzer zeitgleich über die unterschiedlichsten Wege miteinander kommunizieren.

Basierend auf dem Verstecken des Kommunikationsmediums hinter dem Remote-Connection Interface müssen allerdings alle Implementierungen auf den kleinsten gemeinsamen Nenner reduziert werden. Dies bedingt, dass alle Connections über Polling arbeiten, damit die Informationen vom Server bezogen werden können. Ein Kommunikationsaufbau vom Server zum Client ist somit, auch wenn von der genutzten Technologie unterstützt, nicht möglich.

4.2.2 Austausch der Ressource

In Kapitel 2.8.4.5 wurde beschrieben, dass EMF- und GMF-Editoren ihr Datenmodell in Form von XMI abspeichern. Vorteil von XMI ist, dass es ein interoperables Austauschformat ist, mit dem auch andere Editoren fernab von Eclipse umgehen können. Dawn versteht sich zwar als kollaborative Erweiterung für GMF-Editoren, lässt

aber nicht die Vision einer viel stärker verteilten Architektur außer Acht. Aus diesem Grund wurde auch für die Übertragung des Modellbestandes eine nach XMI serialisierte Form gewählt. Dadurch kann es in späteren Entwicklungsstadien möglich sein, andere Editoren außerhalb der Eclipse-Welt mit in das System zu integrieren. Ein weiterer Grund für die Nutzung von XMI besteht in der Tatsache, dass es in XML spezifiziert, also textbasiert ist. Dies erlaubt eine einfache Integration in textbasierte Übertragungsmedien wie SOAP.

Kern der Modellübertragung bildet die *Transport-Ressource*. Sie ist eine Variante der in GMF genutzten GMF-Ressource. Um Daten vom Client zum Server oder umgekehrt zu übertragen, wird eine leere Transportressource erzeugt. Alle Änderungen, die publiziert werden sollen, werden in diese Ressource integriert. Es entsteht also eine nach allen Änderungen gefilterte Sicht auf den Modellbestand. Diese Ressource wird daraufhin nach XMI serialisiert. Hierbei werden die bereits von EMF bereitgestellten Klassen und Methoden genutzt. Dies hat den Vorteil, dass auf ein bereits getestetes und stabiles System zurückgegriffen werden kann und der Aufwand für eine eigene Implementierung vermieden wird.

Ein weiterer Vorteil dieser rein textbasierten Kommunikation ist es, dass externe Komponenten, in diesem Fall das EMF-Framework, die Serialisierung übernimmt. Um mit Remote-Objekt Technologien Java Objekte transportieren zu können, müssen diese die Schnittstelle `Serializable` implementieren. Die Ressourcen in EMF und GMF implementieren dieses Interface allerdings nicht. Diese Serialisierung kann zwar auch von erbenden Klassen implementiert werden, so dass eine Möglichkeit bestünde im Nachhinein die Funktionalität nach zu implementieren, allerdings müssen alle Attribute aus der Basisklasse „per Hand“ serialisiert werden, was ein unwesentlich höherer Aufwand ist, als die genutzte Variante. Zum anderen hat die textuelle Übertragung der Ressource den Vorteil, dass das Datenmaterial bei Bedarf noch komprimiert werden kann. Über dieses Verfahren können beispielsweise viele Daten übertragen werden. Durch den Einsatz von auf Text zugeschnittene Komprimierungsverfahren, wie beispielsweise *LZW*[33], kann eine sehr hohe Komprimierung erreicht werden.

4.2.3 Watcher

Die wichtigste Schnittstelle zwischen der Übertragung der Daten und der Synchronisierung des Modellbestandes ist eine Systemkomponente, welche für den regelmäßigen Abruf der Daten sorgt. Diese Komponente wird von jeder

[33] *Lempel-Ziv-Welch*, verlustfreies Komprimierungsverfahren, welches eine Code-Tabelle für die Komprimierung nutzt [vgl. Henning 2003, S.40]

Editorerweiterung separat zur Verfügung gestellt und muss das Interface **`Watcher`** implementieren. Die Watcher-Implementierung sorgt dafür, dass editorspezifische Objekte und Informationen an die Dawn-Runtime übergeben werden können. In einem vorgegebenen Intervall synchronisiert er den Modellbestand mit dem Server, indem er den aktuellen Modellbestand des Servers zum Client überträgt und diese an die für die Synchronisation zuständigen Komponenten (u.a. **`ResourceSynchronizer`**) weitergibt. Die Einordnung des Watchers in das Dawn-Architekturmodell wurde in Abbildung 28 dargestellt.

Neben der Aufgabe die Daten vom Server zu holen, ist es auch Aufgabe des Watchers das System über den Zustand der Verbindung zum Server zu informieren. Um dies zu ermöglichen greift er auf den DawnRemoteConnector zu, welcher ihm die Information liefert, ob der Server gerade erreichbar ist oder nicht. Kann der Server nicht kontaktiert werden, informiert der Watcher das System über diesen Zustand, damit zum Beispiel im Editor eine Warnmeldung erscheinen kann. Zusätzlich versucht der Watcher weiterhin eine Verbindung aufzubauen. Sobald diese wieder vorhanden ist, informiert er die restlichen Systemteile darüber und beginnt wieder mit der Synchronisierung des Datenbestandes.

4.3 Synchronisation

Um den Datenbestand des Servers mit dem des Clients konsistent zu halten, müssen beide Systeme synchronisieren. Hierbei werden Änderungen des Clients an den Server übertragen und Änderungen des Servers an den Client. Verantwortlich für das Auslösen einer Synchronisation ist die jeweilige Watcher-Instanz der einzelnen Editoren [vgl. 4.2.3]. Nachdem die Synchronisation erfolgreich ausgeführt wurde, müssen Server und Client über denselben Modellbestand verfügen. Die Synchronisation entspricht folglich dem Zusammenführen von Ressourcen, wobei immer ein Akteur die *Master*-Rolle und einer die *Slave*-Rolle einnimmt [vgl. Abbildung 29]. Der *Slave* übernimmt automatisch, außer bei Auftreten von Konflikten [vgl. 4.4], die Änderungen des *Masters*. Je nachdem, welcher Akteur gerade Daten sendet, entscheidet darüber, welche Komponente die Rolle des *Masters* und welche die des *Slaves* einnimmt. Sendet der Client seinen Modellbestand an den Server, so übernimmt er die Rolle des Masters. Der Server aktualisiert daraufhin alle Änderungen. Dieser Vorgang des Übertragens von Daten an den Server wird im Kontext dieser Arbeit als *Publish* oder *Veröffentlichen* bezeichnet. Zur Vermeidung inkonsistenter Zustände, muss das Publizieren eine *atomare* Aktion sein. Am Ende des Publizierens befinden sich der publizierende Client und der Server im synchronen Zustand, sofern keine Konflikte aufgetreten sind.

Im Gegensatz dazu, wird das Übertragen der Daten vom Server an den Client als *Update* oder *Aktualisieren* bezeichnet. Hierbei übernimmt der Server die *Master*-Rolle. Alle Änderungen werden an den Client übertragen. Abbildung 29 verdeutlicht den Unterschied zwischen *Publish* und *Update*.

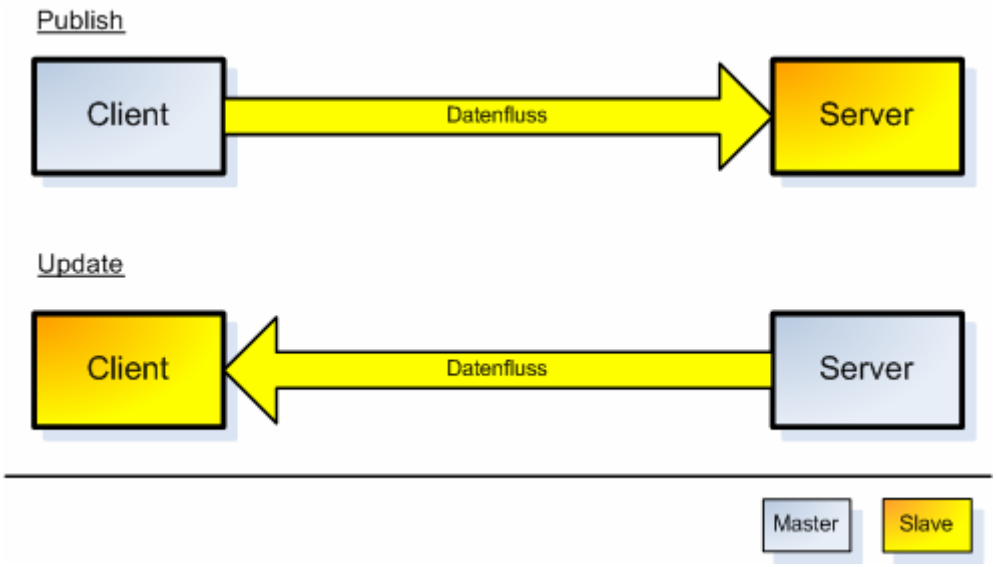

Abbildung 29 - Publish/Update

Da Dawn als lose gekoppeltes System spezifiziert wurde, bestimmt der Client durch sein Polling-Intervall, wann eine Synchronisierung stattfindet. Mit jedem Intervall, wenn eine Verbindung zum Server hergestellt werden kann, wird ein *Update* ausgeführt. Mit Hilfe dieser Informationen kann der Client Änderungen und Konflikte erkennen und ggf. entsprechende Maßnahmen einleiten. Um unnötige Netzlast zu vermeiden, wird das *Publish* immer nur ausgeführt, wenn der Client innerhalb des Editors speichert. Dies bewirkt ein Übertragen der Änderungen, welche der Nutzer seit dem letzten Speichervorgang getätigt hat, an den Server.

Während der Synchronisation werden auf dem Slave Einfüge-, Änderungs- oder Löschoperationen, basierend auf den erkannten Änderungen, durchgeführt. Diese Unterteilung in die drei Grund-Operationen mag trivial erscheinen, ist aber für die Erkennung und Behandlung von Konflikten von elementarer Bedeutung, da sie Konflikte von unterschiedlicher Komplexität erzeugen und entsprechend unterschiedlich behandelt werden müssen.

4.3.1 Datenübertragung

Der offensichtlichste Ansatz, den Datenbestand der Clients mit dem Server zu synchronisieren, besteht darin die jeweiligen kompletten Datenbestände auszutauschen.

Der Client sendet also seinen vollständigen Datenbestand an den Server. Dieser gleicht das Modell ab und stellt die Änderungen den anderen Clients zur Verfügung. Dieser Ansatz erweist sich sehr schnell als unpraktisch, sobald mehr als ein Client mit dem Server kommuniziert. In Abbildung 30 wird verdeutlicht, zu welchem Problem es beim Löschen von Daten kommen kann.

Im ersten Schritt erstellt *Client 1* ein Objekt *A*. Dieses wird über das Speichern des Modells zum Server übertragen. Der Server gleicht seinen Modellbestand ab, indem er das Objekt seinem Modell hinzufügt. Im nächsten Schritt aktualisiert *Client 2* seine Daten, indem er sich den kompletten Datenbestand vom Server holt und mit seinem lokalen abgleicht. Alle Clients verfügen nun über denselben Datenbestand. Im dritten Schritt erstellt *Client 1* ein weiteres Objekt – *B*. Dieses wird ebenfalls zum Server geschickt und der Server-Ressource hinzugefügt. Bevor *Client 2* aber mit dem Modellbestand des Servers abgleicht, publiziert er seinen Datenbestand (Schritt 4). Der Server erhält nun nur das *Objekt A* von *Client 2* und muss davon ausgehen, dass es gelöscht wurde. Er entfernt also das Objekt aus seiner Ressource.

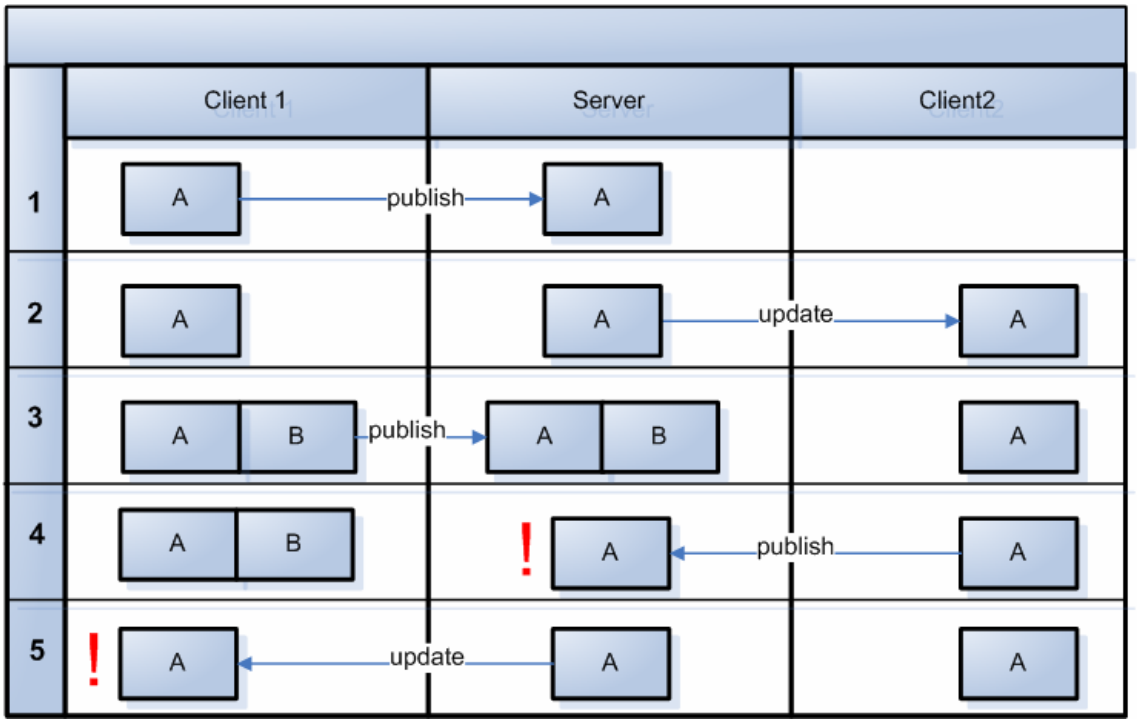

Abbildung 30 - Full-Sync-Delete Anomalie

Aktualisiert nun *Client 1* seine Daten, bevor er diese speichert, so bekommt er vom Server nur *Objekt A* geschickt, und muss davon ausgehen, dass *B* gelöscht wurde. Infolgedessen entfernt er das Objekt ebenfalls aus seinem Datenbestand. Dieses Problem wurde *Full-Sync-Delete Anomalie* genannt, da die Löschung von Objekten aus

dem Modellbestand nur auftritt, wenn der komplette Datenbestand zur Synchronisation genutzt wird.

Es ist offensichtlich, dass diese Vorgehensweise nicht sinnvoll ist, da *Objekt B* zu keinem Zeitpunkt von einem Client gelöscht wurde. Hier versagt die einfache Vergleichsstrategie, da zwar neu hinzugefügte und geänderte Objekte als solche erkannt werden, die Abwesenheit von Objekten aber nicht als Indikator für eine Löschung verwendet werden kann. Zwar minimiert eine Reduktion des Polling-Intervalls die Gefahr des Auftretens, kann diese aber nicht effektiv verhindern. Auch das direkte *Update* vor dem *Publish* kann das Risiko dieser Anomalie nur reduzieren, da genau in dem Zeitraum zwischen beiden Operationen ein zweiter Client seine Daten publizieren könnte. Dieses Verfahren wäre nur wirksam, wenn *Update* und *Publish* auf dem Server zusammen als atomare Operation ablaufen würde. Da das System aber als lose gekoppelt definiert wurde, bei dem Clients nicht permanent mit dem Server verbunden sind, würde durch dieses Vorgehen das Entstehen von Deadlocks[34] provozieren. Würde die Ressource für den Update/Publish-Vorgang gelockt und würde ein Client nach dem *Update* seine Netzwerkverbindung verlieren, wäre die komplette Ressource blockiert, unter Umständen für mehrere Stunden oder Tage.

Um diese Problematiken wirksam zu umgehen, müssen gelöschte Objekte separat übertragen werden. Im optimalen Fall dürfen also nicht die kompletten Datenbestände, sondern nur die aktuelleren Änderungen bezüglichen Einfüge-, Änderungs- und Löschoperationen an den Server geschickt werden. Dies bewirkt, einhergehend mit der wirksamen Anomalie-Vermeidung, eine Steigerung der Performance. Auf das Performance-Verhalten des Systems wird detaillierter in 6.3 eingegangen.

4.3.2 Operationen

In dem System müssen Einfüge-, Änderungs- und Löschoperationen automatisch erkannt und verarbeitet werden können.

Einfügeoperationen stellen dabei die einfachste Operation dar. Sobald der Slave feststellt, dass sich innerhalb der Transport-Ressource Objekte befinden, die nicht lokal vorhanden sind, wird eine Einfügeoperation ausgelöst. Hierbei wird das Vorhandensein von Objekten anhand eines eindeutigen Identifikators geprüft [vgl. 4.3.3].

Für die Behandlung von *Änderungen* muss das System mehrere Parameter betrachten, um diese eindeutig identifizieren zu können. Wird bei der Überprüfung des eindeutigen

[34] Deadlock – Zustand in dem mehrere Prozesse auf die Freigabe gegenseitig reservierter Ressourcen warten und sich dadurch vollständig blockieren [vgl. Tanenbaum 2001, S.36ff].

Identifikators festgestellt, dass ein Objekt sowohl in der Slave-Ressource, als auch in der Master-Ressource vorhanden ist, wird das Objekt auf Unterschiede geprüft. Da das System generisch für alle Objekt-Typen konzipiert ist, muss ein rekursiver Algorithmus alle Attribute und Kind-Objekte überprüfen [vgl. 4.3.3]. Da die Identifikation von Attributen und Kindern ebenfalls anhand eines Identifikators erfolgt, müssen Listen-Objekte in Master und Slave nicht gleich sortiert sein.

Auf Grund der Konfliktbehandlung [vgl. 4.4.4] kann das System davon ausgehen, dass Änderungen während eines *Updates* immer korrekt sind. Sie werden also direkt auf dem Server übernommen.

Änderungen an Objekten, die seit dem letzten *Publishen* durch einen anderen Client geändert, aber nicht lokal modifiziert wurden, sollen übernommen werden. Lokale Änderungen müssen natürlich von der automatischen Aktualisierung ausgenommen werden, da sonst Objekte, die durch einen Nutzer geändert, aber noch nicht publiziert wurden, überschrieben würden. Ebenso muss eine detaillierte Überprüfung der Änderungen erfolgen, um gegebenenfalls Konflikte feststellen zu können. Eine Änderung ist die aufwendigste Operation, da sie zu der größten Anzahl an Konflikten führt. Wie Änderungen ausgeführt werden ohne den Modellbestand in einen inkonsistenten Zustand zu überführen beschreibt Kapitel 4.4.

Löschoperationen ähneln in ihrer Behandlung den Einfüge-Operationen. Sobald innerhalb eines *Publish*-Vorgangs die Abwesenheit eines Objektes in der Ressource des Masters erkannt wird, wird das Objekt vom Server entfernt. Hierbei wird ebenfalls von der Richtigkeit der Client-Informationen ausgegangen. Lösch-Operationen auf dem Client, also während des *Updates* müssen gesondert behandelt werden, um zu vermeiden, dass lokal geänderte Objekte gelöscht werden [vgl. 4.4].

4.3.3 Compare Engine

Damit exakt erkannt werden kann, dass sich ein Objekt geändert hat, beziehungsweise eingefügt oder entfernt wurde, bedarf es eines eindeutigen Identifikators. XMI-Ressourcen können zusätzlich für jedes Objekt mit einer eindeutigen Kennzeichnung, der XMI-ID, versehen werden. GMF-Editoren sind per default so eingestellt, dass sie immer XMI-IDs benutzen. Aus diesem Grund werden diese IDs auch zur Erkennung von Veränderungen im Modellbestand genutzt. Eclipse besitzt ein Projekt, welches es ermöglicht Ressourcen zu vergleichen und Unterschiede zu erkennen. Dieses Projekt heißt *EMF Compare*. Zum Zeitpunkt dieser Arbeit konnte EMF Compare seine Vergleichsoperationen allerdings nicht auf Basis der XMI-ID durchführen, sondern

nutzte für die Vergleiche lediglich die Referenzen der Objekte [vgl. EMFT 2008]. Dies führte dazu, dass geänderte Objekte nicht geändert wurden, sondern entfernt und durch ein neues Objekt ersetzt wurden. Dieses Verhalten ist allerdings inakzeptabel, wenn zum Beispiel Objekte mit Referenzen auf andere Objekte verglichen werden sollen, wie zum Beispiel Kanten, die mit unterschiedlichen Knoten verknüpft sind. Werden diese gelöscht und neu hinzugefügt, können unschöne Seiteneffekte auftreten. Beispielsweise kann es passieren, dass plötzlich die Kanten nicht mehr mit den Knoten verbunden sind. Aus diesem Grund und um mehr Einfluss auf das Verhalten der Vergleichskomponente haben zu können, wurde entschieden eine eigene Implementierung zu erstellen. Diese Compare Engine (*ResourceDiffHelper*) erhält zwei Ressourcen-Objekte und kann mit Hilfe von *Dynamic EMF* herausfinden, welche Änderungen stattgefunden haben. Dabei werden die Änderungen anhand der in Kapitel 4.3.2 beschriebenen Vorgehensweisen erkannt. Diese Modifikationen werden an die aufrufende Instanz zurückgegeben, damit diese verarbeitet werden können. Die Resultate werden in drei separaten Listen für neu eingefügte, geänderte und gelöschte Objekte gespeichert.

4.3.4 Merge Engine

Nachdem die Unterschiede zwischen dem lokalen und dem serverseitigen Modell herausgefunden wurden, müssen diese Änderungen entweder auf dem Server in das Server-Modell oder clientseitig in das Client-Modell integriert werden. Auf dem Server wird hierzu eine Systemkomponente genutzt, welche mit Hilfe von *Dynamic EMF* [vgl. 2.8.4.6] die Modelle zusammenführen kann. Vorteil an dieser Verfahrensweise ist, dass der Server weitgehend generisch arbeiten kann. Über die XMI-ID kann der Server einzelnen Objekte identifizieren und diese dann ohne Kenntnis der konkreten Implementierung aktualisieren.

Innerhalb von Eclipse lässt sich die auf dem Server verwendete Komponente, der **`ResourceHelper`**, nicht ohne weiteres benutzen. Zum einen verbietet das GMF-Framework direkte Manipulationen am Datenmodell und zum anderen könnten clientseitig über den Server erstellte Operationen nicht rückgängig gemacht werden. Um für spätere Erweiterungen des Systems gewappnet zu sein, muss das System diese Funktionalität beherrschen. Dadurch kann es in späteren System-Versionen möglich sein, dass es administrativen Nutzern erlaubt ist, Operationen von anderen Nutzern remote rückgängig zu machen. Damit Änderungen am Modellbestand des GMF-Editors vorgenommen werden können, muss das *EMF-Command-Framework* genutzt werden. Hierbei werden für die Änderung des Datenbestands spezifische Requests erzeugt, und diese an den zu verändernden EditPart gestellt. Soll beispielsweise ein neuer Knoten

erzeugt werden, wird ein **`CreateShapeRequest`** an den EditPart gestellt, welcher die Verwaltung des Elements übernimmt, auf welchem der Knoten gezeichnet werden soll. Dies ist in der Regel das Diagramm-Wurzelelement. Dadurch wird bewirkt, wie in Kapitel 2.8.6.1 beschrieben, dass der EditPart diese Anfrage an eine entsprechende EditPolicy weiterleitet, welche das Kommando erzeugt, um den neuen Knoten zu erstellen. Zur Trennung dieser Funktionalität vom restlichen Code, übernimmt eine eigene Implementierung (**`DawnChangeHelper`**) die Steuerung der in Dawn spezifischen Kommandos.

Der **`DawnChangeHelper`** nutzt dabei wie der **`ResourceHelper`** generische Aktionen. Hierbei offenbart sich die Stärke des von GMF verwendeten Kommando-Mechanismus. Um einen neuen Knoten zu erstellen, benötigt der **`DawnChangeHelper`** keine Kenntnis über die Art des Knotens. Ob es sich dabei um eine Implementierung einer Klassen-View, einer Interface-View oder irgendeines anderen Knotens handelt, ist für ihn vollkommen transparent. Die einzige Information, welche eine Beziehung zum zu erstellenden Element herstellt, ist der *ElementType* [vgl. 2.8.7.4], welcher für die Erstellung des Requests benötigt wird. Alle Operationen die erforderlich sind, werden, um letztendlich den Knoten auf dem Graphen zu erzeugen, innerhalb des Kommandos realisiert, welches von den EditPolicies geliefert wird. Der Erzeuger des Requests muss diese Vorgänge nicht kennen. Da der **`DawnChangeHelper`** keinerlei Abhängigkeiten zu anderen Systemen benötigt, kann er als statische Komponente entworfen werden, um von unterschiedlichen Komponenten aus den Zugriff auf die Datenmanipulationsoperationen bereitstellen zu können.

4.3.5 Schutz lokaler Änderungen

Neben der Problematik, dass versehentlich Objekte auf dem Server gelöscht werden, muss das System auch die lokalen Änderungen der Nutzer schützen. Dies betrifft sowohl die aktuellen, als auch die bereits bearbeiteten Objekte. Wird ein Objekt, auf welches ein lokaler Nutzer gerade einen Fokus gesetzt hat, serverseitig gelöscht, so würde, ohne jeglichen Schutz-Mechanismus, dieses Objekt bei der nächsten Aktualisierung (Update) aus dem Client-Diagramm entfernt werden. Die Änderungen des Nutzers wären verloren. Ebenso verhält es sich mit serverseitigen Änderungen. Hier würden plötzlich globale Änderungen an dem Objekt vorgenommen werden, welches gerade editiert wird. Das System muss folglich in der Lage sein, selektierte und lokal geänderte Objekte zu identifizieren und von globalen Änderungen zu unterscheiden. Lokal eingefügte, geänderte und gelöschte Objekte müssen von der globalen Aktualisierung ausgenommen werden, bis die Änderungen publiziert wurden. Leider

begünstigt diese verzögerte Aktualisierung die Wahrscheinlichkeit des Auftretens von Konflikten, da je länger das Abgleichen mit dem Gesamtsystem verhindert wird, die Wahrscheinlichkeit steigt, dass andere Nutzer Änderungen am selben Objekt vornehmen. Die folgenden Kapitel beschreiben, wie es zu diesen Konflikten kommen kann und wie sie behandelt werden können.

4.4 Konflikte

Konflikte können in allen Systemen auftreten, in denen mehrere Benutzer schreibenden Zugriff auf eine Ressource haben. Grundsätzlich werden zwei verschiedene Arten von Konflikten unterschieden. *Lese/Schreib-Konflikte* treten dann auf, wenn eine Komponente lesenden Zugriff auf das System hat und eine andere Komponente zeitgleich eine Schreib-Operation ausführt. *Schreib/Schreib-Konflikte* entstehen, wenn beide Komponenten zeitgleich Zugriff auf die Ressource haben [vgl. Tanenbaum et al. 2003, S.322 ff.].

Hierbei ist die Beschaffenheit des Systems unabhängig von dem grundlegenden Potential der Konfliktbildung. So können sie zum Beispiel innerhalb von Programmen auftreten, wenn mehrere parallele Threads auf einen Speicher zugreifen, wenn zeitgleich Nutzer dieselbe Datenbank-Ressource nutzen oder auch, wenn Clients eines verteilten Systems Ressourcen ändern, ohne von Änderungen anderer Teilnehmer Kenntnis zu bekommen.

Da Dawn ein System ist, welches auf verteilten Ressourcen aufbaut und Änderungen nur in unregelmäßigen Abständen (Publish/Update) kommuniziert werden, erhöht sich die Gefahr von inkonsistenten Zuständen und Konflikten. Erschwert wird diese Problematik dadurch, dass der Server auf Grund der losen Kopplung keine Möglichkeit besitzt Änderungen an der Server-Ressource sofort an alle Clients zu kommunizieren. Er ist darauf angewiesen, in regelmäßigen Abständen den aktuellen Datenbestand abzugleichen. Da ein Nutzer aber auch über einen längeren Zeitraum offline arbeiten können soll, müssen auch langläufige Konflikt-Potentiale erkannt und behandelt werden [vgl. 4.5].

Die in den verschiedenen Systemen auftretenden Konflikte können, wenn auch ähnlich, von unterschiedlicher Natur sein. Darum müssen Konflikte für jedes verteilte System gesondert untersucht und analysiert werden. Besonders die Konfliktbehebung kann sich dabei stark unterscheiden. Aus diesem Grund beschäftigt sich das folgende Kapitel mit den in Dawn vorkommenden Konflikt-Situationen, ihren Konsequenzen und Lösungs- bzw. Vermeidungsansätzen. Dabei werden zuerst die möglichen Konflikte und

konflikterzeugenden Situationen beschrieben, um anschließend Konzepte für die systemgesteuerte Erkennung dieser zu entwickeln. Daraufhin wird erläutert, wie erkannte Konflikte vom Nutzer behoben werden können. Anschließend werden Verfahren beschrieben, die es ermöglichen unter bestimmten Umständen Konflikte im System gänzlich zu vermeiden.

4.4.1 Konflikte – Client Seite

Konflikte können auf der Client-Seite auftreten, sobald der Nutzer eine Operation ausführt. Aus diesem Grund muss die Erkennung von Konflikten so schnell wie möglich erfolgen um zu verhindern, dass der Nutzer weiter mit einem nicht validen Modellzustand arbeitet und sich das Problem so ausweitet. Bevor Konflikte wirksam erkannt beziehungsweise vermieden werden können, muss ein System zuerst hinsichtlich möglicher Konfliktsituationen analysiert werden, um herauszufinden in welchen Situationen welche Konflikte auftreten können.

4.4.1.1 Lokal- und Remote-Änderungskonflikt

Die erste Art von Konfliktsituation kann bei Änderungen von Objekten auftreten. Ein *Lokal- und Remote-Änderungskonflikt* entsteht immer dann, wenn zwei Nutzer gleichzeitig ein Objekt ändern. Abbildung 31 verdeutlich die Aktionen, die zum Entstehen dieses Konfliktes führen. Ausgangsbasis bildet ein synchronisierter Systemzustand, in dem alle Systemteile über denselben Modellbestand verfügen (Schritt 1). Im zweiten Schritt ändern zwei Nutzer ein Objekt – Client 1 versetzt das Objekt in *Zustand B* und Client 2 in *Zustand C*. Dieser Zustand ist noch nicht kritisch, da noch keine Übertragung an den Server stattgefunden hat. Im dritten Schritt publiziert Client 2 seine Änderung. Der Server synchronisiert die Ressource mit diesen Informationen.

Der *Objektzustand C* wird als global gültig erklärt. An dieser Stelle ist ohne Gegenmaßnahmen das Entstehen eines Konfliktes nicht mehr zu vermeiden. Würde Client 1, wie in Schritt 4 dargestellt, seine Objektänderung publizieren, würden die Änderungen von Client 1 überschrieben werden. Das Szenario entspricht einer klassischen *Race Condition*[35], da Client 1 seine Daten publiziert ohne Kenntnis von den Änderungen anderer Clients zu erhalten.

[35] Wettlaufsituationen in parallelen Systemen, die zu schwer auffindbaren Programmierfehlern führen können [vgl. Tanenbaum 2001, S.100ff.].

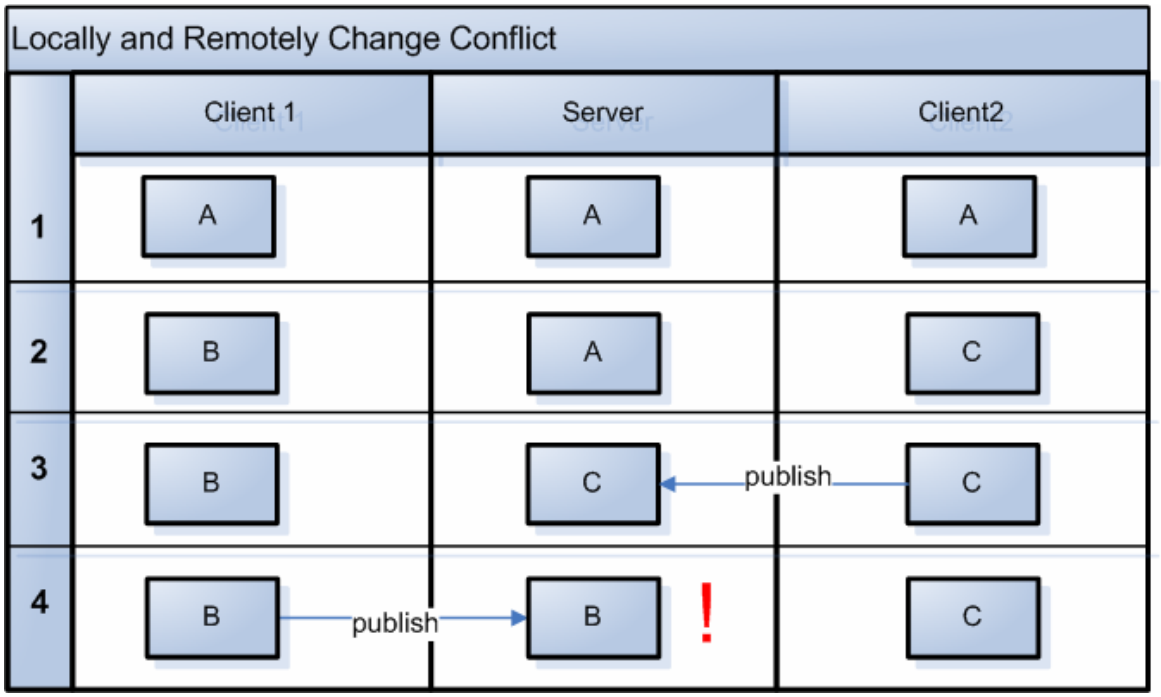

Abbildung 31 - Lokal- und Remote-Änderungskonflikt

4.4.1.2 Lokaler Löschkonflikt

Neben Konflikten durch Änderungen von Objekten, können Konflikte auch durch das Löschen von Objekten entstehen. Hierbei ist zu unterscheiden ob ein Nutzer selbst das Objekt gelöscht hat (lokale Löschung), oder die Löschung von einem anderen Nutzer durchgeführt wurde (entfernte Löschung). Lokale Lösch-Konflikte treten auf, wenn ein Nutzer ein Element löscht, ein anderer Nutzer aber bereits Änderungen an diesem Objekt vorgenommen hat.

In Abbildung 32 führt Client 2 (entfernter Nutzer) eine Änderung an einem Objekt durch und verändert es so, dass es sich im *Zustand B* befindet.

Zeitgleich entfernt Client 1 dieses Objekt lokal aus seinem Modellbestand. Client 2 auf dem entfernten System publiziert aber seine Änderungen zuerst, ohne dass Client 1 über diese Änderung informiert wird. Die Änderungen könnte auch gar nicht in seinem Modellbestand angezeigt werden, da das Objekt bereits gelöscht wurde. Würde Client 1 nun die Löschung publizieren, gingen die Änderung des anderen Nutzers verloren.

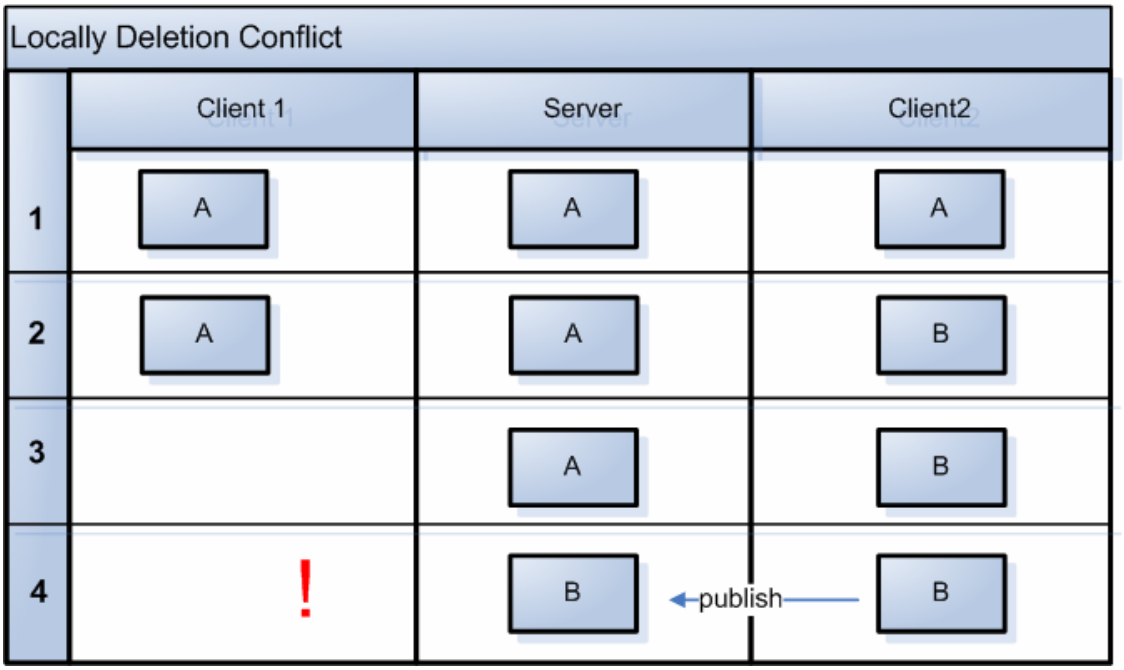

Abbildung 32 - Lokaler Löschkonflikt

4.4.1.3 Entfernter Löschkonflikt

Ein entfernter Löschkonflikt entsteht, ähnlich wie der lokale Löschkonflikt, wenn ein Nutzer ein Element löscht, an dem ein anderer Nutzer bereits Änderungen vorgenommen hat. Der Unterschied zum lokalen Löschkonflikt besteht darin, dass diese Löschung bei dem ändernden Nutzer noch nicht publiziert wurde. Er arbeitet folglich auf einem bereits gelöschten Objekt [vgl. Abbildung 33]. Um zu verhindern, dass das Objekt beim nächsten Veröffentlichen einfach wieder erstellt wird, muss der Nutzer über diesen Konflikt informiert werden, damit er ihn manuell beseitigen kann.

Prinzipiell stellt sich die Situation sehr ähnlich der unter 4.4.1.2 beschriebenen dar. Im Fall des entfernten Löschkonfliktes publiziert allerdings der Nutzer, welcher die Löschung ausgeführt hat zuerst seine Modellbestandsänderung. Würde nun Client 1 von Schritt 4 ausgehend seine Änderungen am Objekt publizieren, würde die Löschung von Client 2 rückgängig gemacht werden.

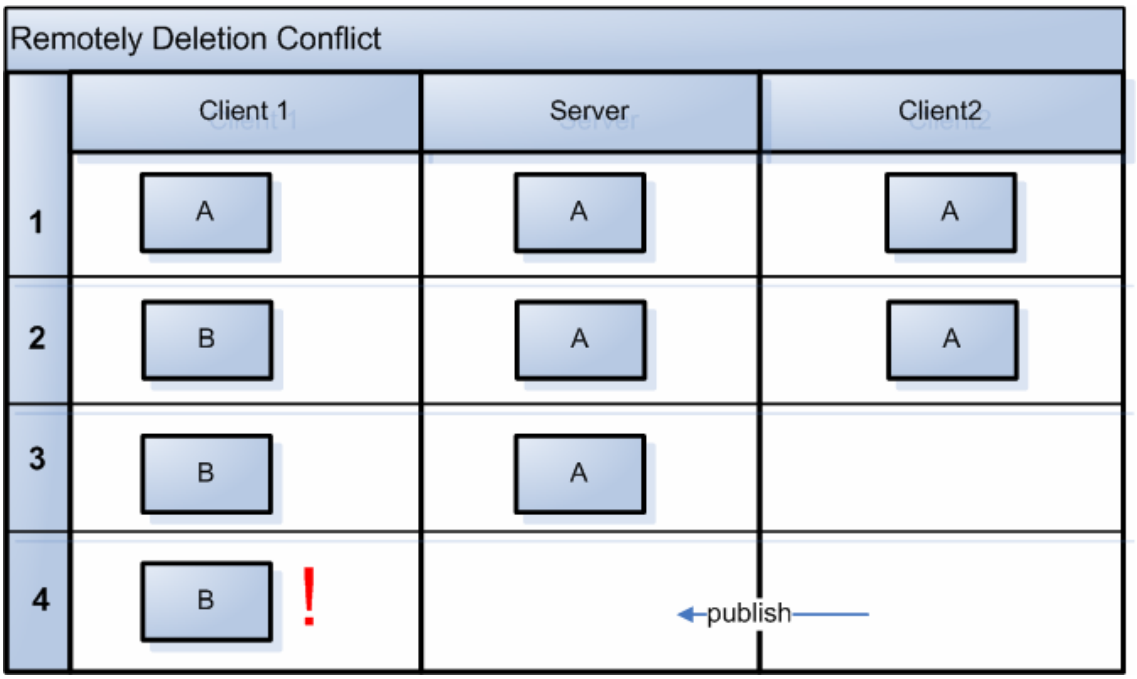

Abbildung 33 - Entfernter Löschkonflikt

4.4.2 Konflikte – Serverseite

Aber auch, wenn auf dem Client kein Konflikt erkannt wird, und infolgedessen ein für den Client valider Modellbestand an den Server gesendet wird, kann nicht ausgeschlossen werden, dass Konflikte auftreten. Dies liegt an der Latenzzeit, welche die Übertragung der Daten benötigt. Folglicherweise muss auch auf dem Server eine Validierung der Daten stattfinden, bevor diese persistiert werden.

Die clientseitige Konflikterkennung kann einen Großteil von Konflikten schon während der Nutzerinteraktion erkennen und das System vor invaliden Zuständen schützen. Alle Konflikte können allerdings nicht verhindert werden, da Änderungen anderer Nutzer nur gesichert auf dem Server erkannt werden können. Der Server darf also nicht allein darauf vertrauen, dass die Daten der Clients gültig sind, sondern muss den Zustand der Modelle hinsichtlich ihrer Aktualität überprüfen, um fehlerhafte Änderungen zu vermeiden. Abbildung 34 verdeutlicht einen solchen Fall. Im ersten Schritt besitzt nur der Server ein Objekt im *Zustand A*. Im zweiten Schritt beziehen zwei am System angemeldete Clients den Modellbestand, ohne voneinander Kenntnis zu haben. Sie verfügen nun beide über das Objekt im *Zustand A*. Alle lokalen Änderungen werden auf diesem Stand aufgebaut. In Schritt 3 ändern beide Clients das Objekt, zum Beispiel indem sie seine Position innerhalb des Diagramms verändern. Beide Clients führen aber unterschiedliche Modifikationen durch. Nutzer 1 setzt das Objekt in den *Zustand B*,

während Nutzer 2 das Objekt in *Zustand C* verschiebt. Nun publiziert Nutzer 1 seine Daten und verändert den Modellbestand auf dem Server (Schritt 4). Nutzer 2, der von diesen Änderungen nichts mitbekommen hat, publiziert seinen Datenbestand in Schritt 5 ebenfalls an den Server. Da Client 2 über die Änderung in *Zustand B* nicht informiert wurde, wird nun aber die Änderung von Nutzer 1 überschrieben.

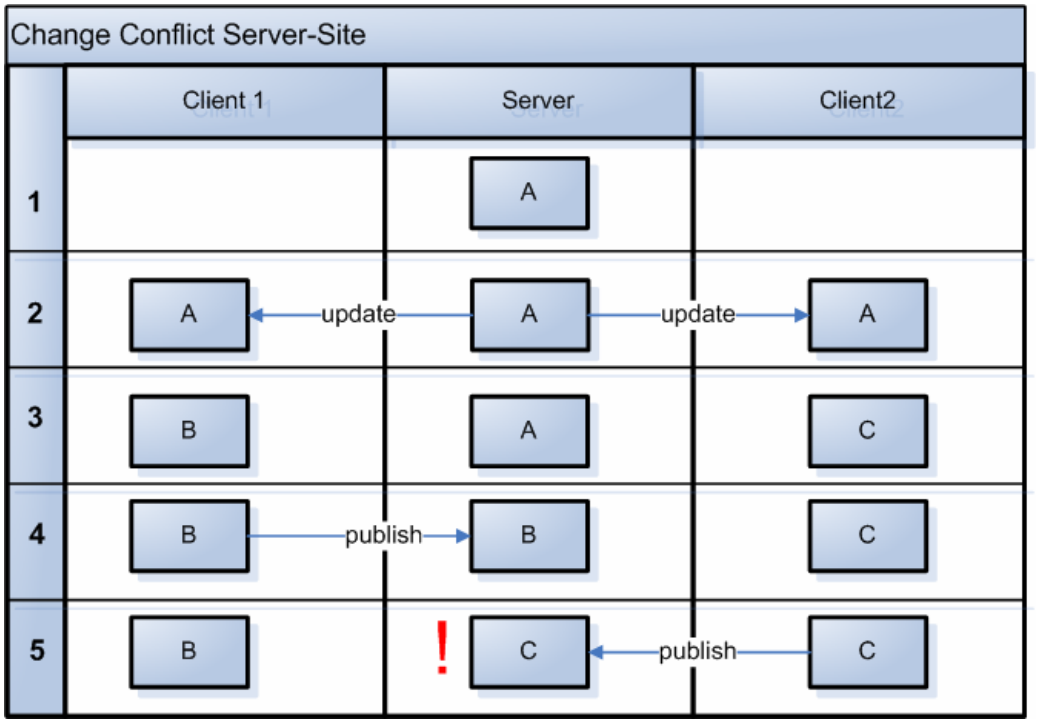

Abbildung 34 - Änderungskonflikte serverseitig

Es ist offensichtlich, dass eine clientseitige Behandlung dieses Problem nicht lösen kann. Selbst wenn Nutzer 2 sich vor dem Publizieren den aktuellen Modellbestand des Servers holen würde, bestünde doch ein kleines Zeitfenster zwischen dem *Update* und dem *Publish* in dem andere Clients den globalen Modellbestand verändern können. Der Server muss also in der Lage sein dieses Problem zu lösen. Hierbei bieten sich drei Lösungsansätze an. Zum einen könnte der Server sich den letzten Informationstand eines jeden Client merken. Fordert also ein Client ein *Update* an, so speichert der Server eine Kopie der ausgelieferten Daten. Da der Client ebenfalls über Konflikterkennungs- und Behandlungsverfahren verfügt, kann der Server davon ausgehen, dass die Daten beim Client nicht in einen Konflikt treten. Führt nun der Nutzer eine Publizierung der Daten durch, kann der Server den alten Stand mit seinen aktuellen Daten und den neuen Daten der Clients vergleichen. Unterscheiden sich alle drei Objekte, so muss das Objekt zwischenzeitlich von einem anderen Nutzer geändert worden sein. Die Aktualisierung

muss also verhindert werden. Es setzt allerdings einen erhöhten Verwaltungsaufwand auf dem Server voraus.

Der zweite Ansatz zur Lösung des Problems besteht im Einführen von Versionsnummern für Objekte. Abbildung 35 veranschaulicht das Verfahren. Im ersten Schritt verfügt der Server über ein Objekt im *Zustand A*. Zusätzlich zu den reinen Objektinformationen besitzt das Objekt aber eine Versionsnummer mit der initialen Version 1. Diese Versionsnummer wird während des *Updates* in Schritt 2 zusammen mit den Objektinformationen an die Clients geliefert. Die Clients besitzen nun einen eindeutigen Identifikator, über den sich der Modellbestand, auf dem sie ihre Änderungen durchführen, identifizieren lässt. Nutzer 1 und Nutzer 2 ändern wie im vorherigen Fall nun die Objekte. Im vierten Schritt publiziert Nutzer 1 seinen aktuellen Modellbestand. Zusätzlich zu den Änderungen übermittelt er aber die Versionsnummer, auf der seine Änderungen beruhen. Der Server kann nun die Versionsnummer mit der seines aktuellen Objektes vergleichen und erkennen, dass keine Versionsunterschiede vorliegen.

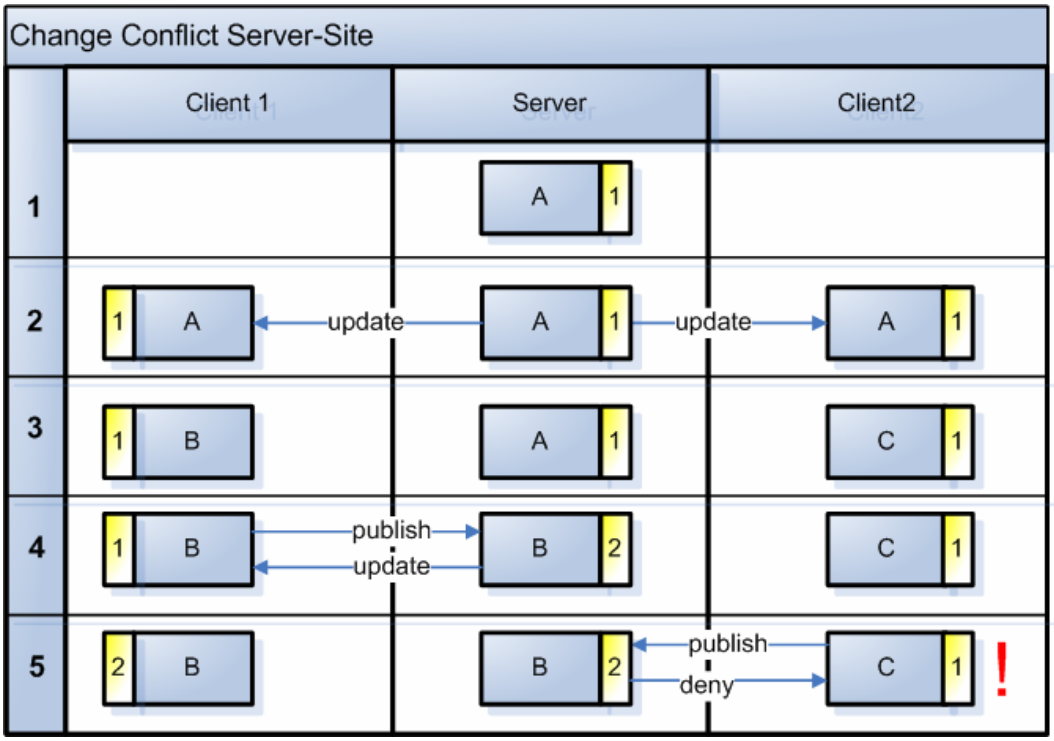

Abbildung 35 - Änderungskonflikt serverseitig - Lösung

Die Änderung von Nutzer 1 ist also eine direkte Folge seines aktuellen Bestandes. Der Server kann das Objekt in den neuen Zustand setzen. Zusätzlich erhöht er, da nun eine Veränderung an dem Objekt stattgefunden hat, die Version und informiert Nutzer 1 über diesen Wert (Schritt 4). Nutzer 2 möchte nun auch seine Änderungen publizieren, hat aber, wie zuvor, keine Notiz von dem neuen Objektzustand auf dem Server erhalten. Er

publiziert den *Zustand C* des Objektes und die Versionsnummer, auf der seine Änderungen beruhen. Der Server vergleicht daraufhin wieder die Versionsnummern, stellt aber fest, dass die Änderungen der Clients auf einer veralteten Version beruhen. Er aktualisiert das Objekt nicht, sondern informiert den Client, dass ein Änderungskonflikt stattgefunden hat. Client 2 kann daraufhin das Objekt in den Zustand *conflicted* setzen und dem Nutzer die weiteren Konfliktbehandlungen überlassen.

Die dritte Möglichkeit diesem Änderungskonflikt zu begegnen ist ein hybrider Ansatz aus beiden bereits vorgestellten Methoden. Da der Client zur lokalen Änderungs- und Konflikterkennung seinen alten Versionsbestand vorhält, kann diese Information zur Konfliktbehebung genutzt werden [vgl. 4.4.3]. Anstatt eine Versionsnummer einzuführen, welche auf dem Server gespeichert werden muss, kann der Client, zusätzlich zu seiner aktuellen Änderung, den alten Modellbestand an den Server übermitteln. Da es sich bei dem alten Modellbestand des Clients systembedingt um die letzte gültige Version des Servers handelt, kann der Server nun wie im ersten Ansatz beschrieben herausfinden, ob eine Änderung zu dem globalen Objekt stattgefunden hat und gegebenenfalls die Änderungsanforderung zurückweisen. Die erste und letzte Methode kommt ohne die Einführung zusätzlicher Variablen aus, indem sie über Vergleiche Änderungen am Modellbestand herausfindet. Die erste Methode setzt aber eine zusätzlich Speicherung auf dem Server voraus, wohingegen der letzte Ansatz zusätzlich Informationen überträgt. Der versionsnummernbasierte Ansatz speichert beziehungsweise überträgt nur wenige Informationen, verlangt aber, dass zu den Objekten zusätzliche Daten erfasst werden müssen. Dies kann in Systemen, an den keine Änderungen an bestehenden Klassen und Komponenten vorgenommen werden können oder sollen, unter Umständen kompliziert implementiert werden.

In Dawn wird der zweite Ansatz implementiert, da dieser die einfachste und performanteste Variante darstellt. Hierbei werden zu jedem Objekt einer Ressource zusätzlich eine Zuordnung zwischen aktueller Version und XMI-ID gespeichert wird. Diese wird bei der Aktualisierung übertragen und kann so den Server über die Versionen der Objekte informieren. Besteht ein Konflikt, so kann der Server den Client über alle Objekte informieren, die nicht aktualisiert werden konnten. Daraufhin kann der Client die entsprechenden Maßnahmen zur Anzeige und Beseitigung der Konflikte einleiten.

4.4.3 Konflikterkennung

Die in den obigen Kapiteln aufgeführten Konflikte müssen systemseitig zuverlässig erkannt werden, bevor sie behandelt werden können.

Grundlage für die Erkennung der Konflikte bildet das Ergebnis der Compare-Engine wie im Kapitel 4.3.3 beschrieben. Dieser Vergleich basiert auf den Unterschieden zwischen der lokalen Ressource (`LocalResource`), welche alle lokalen Änderungen am Modellbestand enthält und den globalen Änderungen der Modellbestandes, welche vom Server geschickt wurden (`ServerResource`). Allerdings reicht diese Information allein nicht aus, da zusätzlich noch der alte Modellbestand des Servers vorhanden sein muss, um alle Konflikte zu erkennen. Aus diesem Grund wird bei jeder erfolgreichen Aktualisierung der Modellbestand lokal in einer weiteren Ressource gespeichert – der *LastResource*. Mit Hilfe der LastResource können zum Beispiel lokale und entfernte Änderungskonflikte erkannt werden. Unterscheidet sich ein Objekt sowohl in der LocalResource als auch in der Last- und der ServerResource, so bedeutet dies, dass sich ein Objekt sowohl lokal als auch global geändert hat.

Neben der Funktion der Erkennung von Konflikten dient die LastResource auch als Vergleichsgrundlage, um lokale Änderungen zu identifizieren und diese bis zur nächsten Synchronisation zu registrieren. Der in Abbildung 36 abgebildete Algorithmus stellt dar, wie anhand der Objektvergleiche lokal analysiert wird, ob ein Konflikt aufgetreten ist, oder nicht.

Um Konflikte rechtzeitig erkennen zu können, wird diese Überprüfung mit jeder Aktualisierung durchgeführt. Diese Überprüfung wird ebenfalls durchgeführt, bevor der Client Daten an den Server publizieren möchte. Jedem Publish geht also ein Update voraus.

Sobald ein Konflikt im System identifiziert wird, muss verhindert werden, dass die Daten an den Server übertragen werden beziehungsweise, dass der Konflikt in den globalen Modellbestand übernommen wird. Aus diesem Grund wird das Publizieren von Änderungen an den Server so lange unterbunden, bis alle Konflikte im lokalen System behoben sind. Dem Nutzer werden dabei alle Konflikte angezeigt und er kann für jeden Fall entscheiden, wie er den Konflikt lösen möchte.

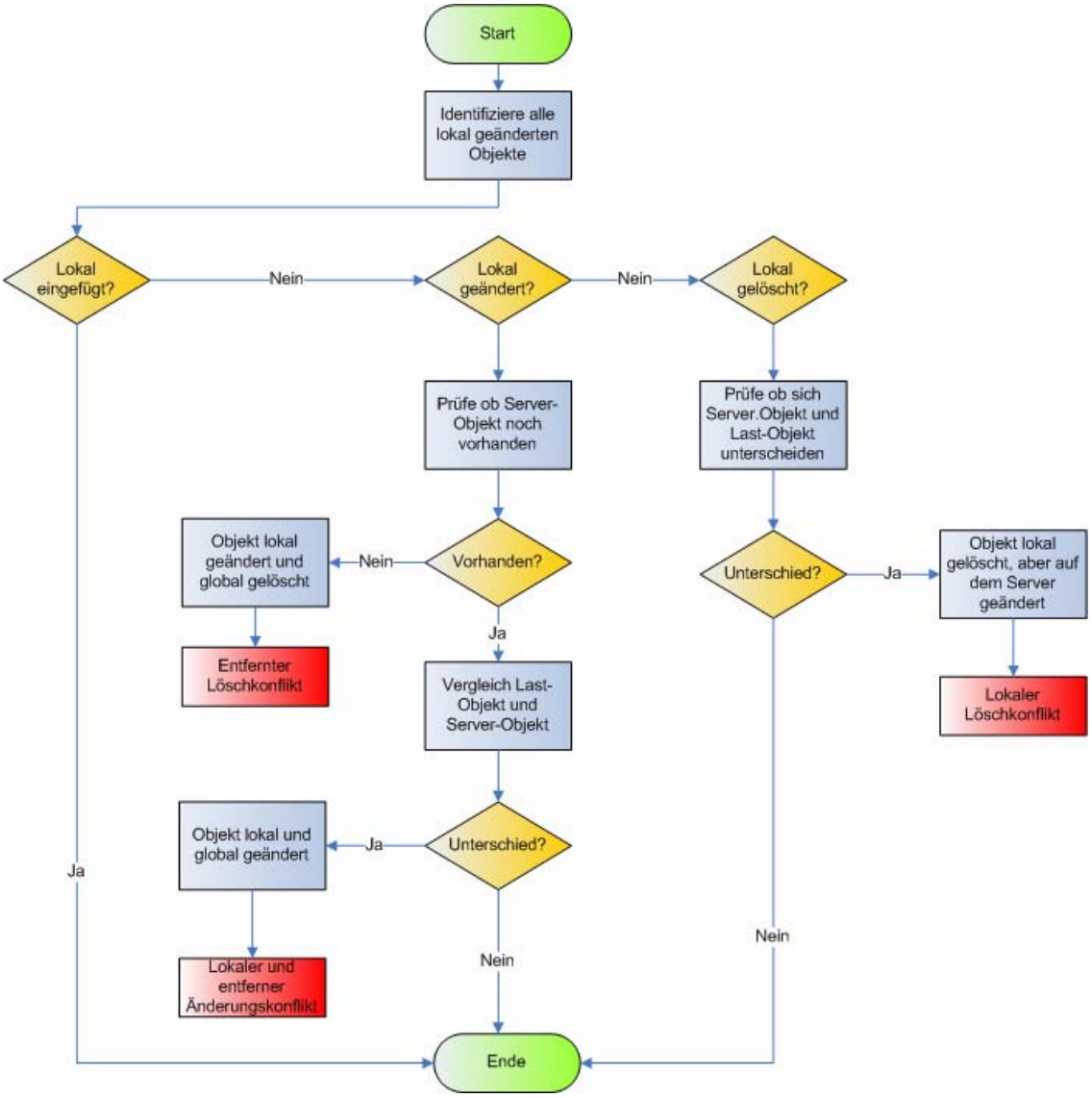

Abbildung 36 - Algorithmus zur Konflikterkennung

4.4.4 Konfliktbeseitigung

Eine einfache Herangehensweise an das Lösen von Konflikten ist die lokale Änderung zu verwerfen und die Änderung des Serverstandes zu akzeptieren. Diese Variante ist sehr leicht zu implementieren, hat aber den Nachteil, dass komplexe Änderungen unter Umständen verloren gehen. Besonders durch die lose Kopplung zwischen Server und Client kann es dazu kommen, dass aufwendige Änderungen verloren gehen, da der Server den Client nicht sofort über Konflikte informieren kann und darauf angewiesen ist, dass der Client sich aktualisiert. Zum anderen ist Dawn auch als System spezifiziert, welches offline arbeiten kann. Zwischen einzelnen Aktualisierungszyklen kann also sehr viel Zeit vergehen, unter Umständen Stunden oder Tage. Wird in dieser Zeit stark an dem grafischen Modell gearbeitet, wäre ein Zurücksetzen dieser Arbeit im

Konfliktfall nicht akzeptabel. Aus diesem Grund wird es in Dawn ermöglicht, dass der Nutzer selbst entscheiden kann, ob er die Version des Servers, also die letzte Änderung, akzeptiert oder sein Modell als valide einstuft. Diese Entscheidung kann der Nutzer für jedes in Konflikt getretene Objekt eigenständig treffen. Er ist also nicht daran gebunden sein gesamtes Modell zu publizieren oder zu verwerfen, sondern kann selektiv die Konflikte auflösen.

Dieses flexible System ermöglicht es zwar individuell den Modelbestand zu bereinigen, kann aber auch zu Problemen führen, wenn die Projektteilnehmer sich nicht über Modeländerungen verständigen. Löst beispielsweise ein Nutzer einen Konflikt, indem er sein Objekt als gültig erklärt, so gerät ein anderer Nutzer, der gerade an diesem Objekt arbeitet, automatisch in einen Konflikt. Löst dieser Nutzer ebenfalls den Konflikt durch Publizieren seiner Änderung kann es zu einem *Ping-Pong* Effekt kommen, bei dem sich die Nutzer gegenseitig in einen Konflikt setzen. Aus diesem Grund ist es unerlässlich, dass Konflikte mit höchster Sorgfalt und möglichst durch Rücksprache mit anderen Projektteilnehmern gelöst werden. Da mit steigender Teilnehmerzahl allerdings in manchen Projekten nicht immer eine ordentliche Kommunikation möglich ist, stellt Dawn mit Hilfe eines flexiblen Rechte-Systems die Möglichkeit zur Verfügung, Konfliktbehebungen zu beschränken [vgl. 4.6]. So könnten, basierend auf dem Rechte-Management, Beschränkungen erlassen werden, die es bestimmten Nutzern untersagen den eigenen Modellbestand als gültig zu erklären. Diese Nutzer könnten dann nur den globalen Status des Modells akzeptieren. Abbildung 37 verdeutlicht den Algorithmus, der zur Identifikation und Behandlung der Konflikte entworfen wurde.

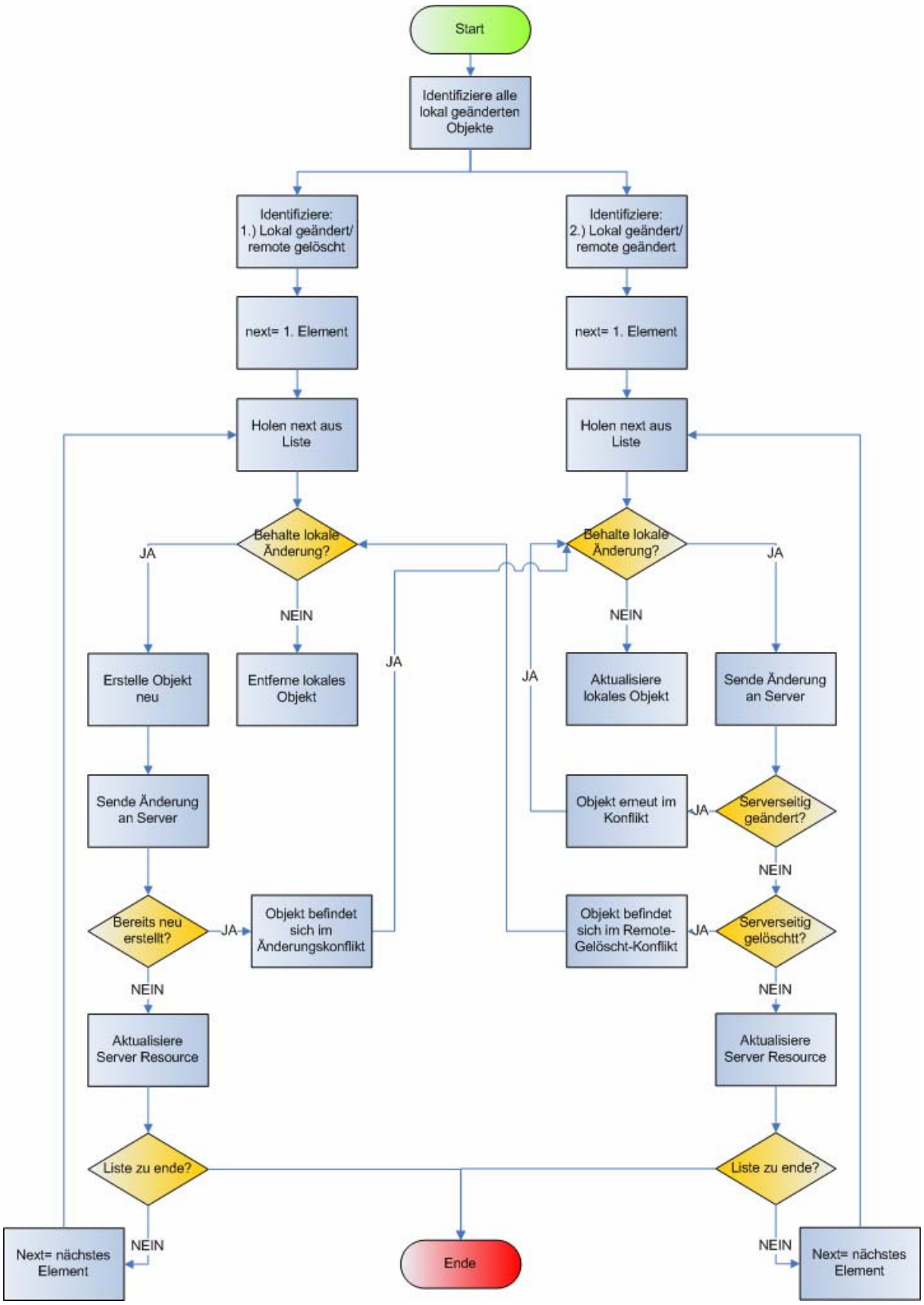

Abbildung 37 - Algorithmus zur Konfliktbehebung lokal geänderter Objekte

4.4.5 Konfliktvermeidung

Die in den vorherigen Kapiteln beschriebenen Verfahren zur Konflikterkennung nutzen das Prinzip des optimistischen Lockings, bei dem prinzipiell alle Operationen erlaubt sind. Erst wenn eine Schreiboperation ausgeführt wird, also die Ressourcen synchronisiert werden, findet eine Überprüfung auf Konflikte statt [vgl. Tanenbaum et al. 2003, S. 323ff.]. Im Fehlerfall wird die Operation nicht ausgeführt, um den Datenbestand valide zu halten. Optimistische Locking Verfahren können allerdings das Auftreten von Konflikten nicht generell verhindern, sondern diese nur erkennen und unterbinden, dass die Daten korrumpiert werden. Die Lösung des Konflikts muss entweder manuell durch den Nutzer oder automatisch erfolgen. Damit Konflikte definitiv vermieden werden können, muss ausgeschlossen werden, dass mehrere Nutzer gleichzeitig Schreibberechtigungen auf ein Objekt haben. Kapitel 4.4 legt bereits dar, dass Konflikte nur auftreten können, wenn mehrere Nutzer zeitgleich Schreibrechte auf ein Objekt haben. Hierbei ist der Umkehrschluss zulässig, nämlich dass Konflikte nicht auftreten können, sobald nur ein einziger Nutzer Schreibrechte auf ein Objekt besitzt. Diese Tatsache wird in Dawn genutzt, um Objekte vor Konflikten zu schützen. Über das Nutzerinterface können ausgewählte Kanten und Knoten selektiert und einer Liste von gesperrten Objekten hinzugefügt werden. Diese Liste wird an den Server publiziert, welcher die globale Verwaltung aller gesperrten Objekte übernimmt. Diese Liste wird an alle anderen Clients publiziert. Diese sperren die Objekte lokal und verhindern, dass auf ihnen Schreiboperationen ausgeführt werden können. Wie dieses Konzept in der konkreten Implementierung umgesetzt wird, beschreibt Kapitel 5.4.3.

Locking in Diagrammen ist ein sehr komplexer Vorgang. Es genügt nicht einfach nur ein Objekt zu locken, sondern es müssen sämtliche Referenzen zu diesem Objekt geprüft werden. Wird beispielsweise ein Lock auf eine Kante gesetzt, so müssen alle mit dieser Kante verbundenen Objekte mit gelockt werden, da in GMF-Editoren beim Löschen eines Knotens auch die entsprechenden Kanten verschwinden. Deshalb müssen verbundene Objekte mit reserviert werden, was zu komplexen Problemen führen kann, wenn andere Nutzer diese Objekte bereits reserviert haben. Um mehr Zeit in die Lösung der anderen Probleme investieren zu können, bereitet Dawn zwar die Basis für dieses komplexe Locking, setzt es aber nur für Knoten direkt um und belässt das Locking von Kanten vorerst in einem experimentellen Status.

4.5 Netzwerkunabhängigkeit

Der in Dawn angedachte Offline-Modus versetzt die Nutzer in die Lage unabhängig von einer Verbindung zum Server arbeiten zu können. Dadurch können mobile Clients ihre Arbeit fortsetzen, auch wenn sie temporär keinen Netzwerkzugang haben. Allerdings entsteht durch die Abwesenheit der Kommunikation mit dem Server eine Reihe von Problemen. Da der Kern der clientseitigen Konflikterkennung auf dem Dreier-Vergleich von Local-, Last- und Server-Ressource beruht [vgl. 4.4.3], müssen alle diese Ressourcen vorhanden sein. Bricht im normalen Betrieb die Netzwerkverbindung weg, so ist die LastResource immer noch im Speicher der Anwendung enthalten. Wird die Verbindung wieder hergestellt, können Änderungen synchronisiert und Konflikte, die zwischenzeitlich entstanden sind, erkannt und behoben werden. Schließt der Nutzer allerdings nach Abbruch der Verbindung die Eclipse-Instanz, so befindet sich beim Neustart keine LastResource im Speicher. Diese kann auch nicht über den Server angefordert werden wenn keine Verbindung besteht. Änderungen können also nicht zuverlässig erkannt werden. Durch einen simplen Eingriff in das System kann dieses Problem allerdings leicht behoben werden. Die LastResource wird als zusätzliche Ressource zur GMF-Ressource behandelt und ebenfalls im File-System abgelegt. Bei jeder Änderung werden also zwei Ressourcen – der aktuelle Stand und der letztbekannte Stand vom Server - abgelegt. Bricht die Verbindung zum Server ab, der Client arbeitet also im Offline-Modus, wird nur noch die aktuelle Ressource gespeichert. Wird die Clientinstanz geschlossen und wieder geöffnet, so wird neben der lokalen Ressource auch der letzte Serverstand in den Datenspeicher geladen, sodass sobald die Verbindung zum Server wieder herstellt ist, der Vergleich der drei Ressourcen erfolgen kann und mögliche Konflikte detektiert werden können.

Ein weiteres Problem im Offline-Modus entsteht bezüglich des in Kapitel 4.4.5 entworfenen Locking-Verfahrens. Die Information, welche Objekte gelockt sind, muss ebenfalls nach einem Neustart von Eclipse wieder zur Verfügung stehen. Andernfalls könnte ein Nutzer plötzlich Objekte editieren, die serverseitig gelockt sind. Aus diesem Grund muss die Liste aller gelockten Objekte ebenfalls mit persistiert werden. Zusätzlich müssen sofort beim Start des GMF-Editors, wenn eine Verbindung zum Server hergestellt werden kann, die Informationen über aktuell gelockte Objekte eingeholt werden. Hat ein Nutzer Objekte gelockt, während ein anderer Nutzer offline gearbeitet und so keine Kenntnis vom Locken der Objekte erlangt hat, so müssen diese automatisch gelockt werden, wenn sie noch nicht verändert wurden. Andernfalls muss ein Konflikte ausgelöst werden, wenn dieses Objekt bereits lokal geändert wurde. Das Objekt darf allerdings solange nicht aus dem Konflikt gelöst werden, solange es durch

einen anderen Nutzer gelockt ist. Hierbei müssen sich die beiden Parteien über eine adäquate Konfliktlösung verständigen. Eine andere Möglichkeit diesem Problem zu begegnen, wäre das komplette Überschreiben aller Änderungen von gelockten Objekten, was aber unter Umständen einen erheblichen Datenverlust zur Folge hätte. Abbildung 38 stellt den Vorgang bei Wiederaufnahme der Verbindung grafisch dar.

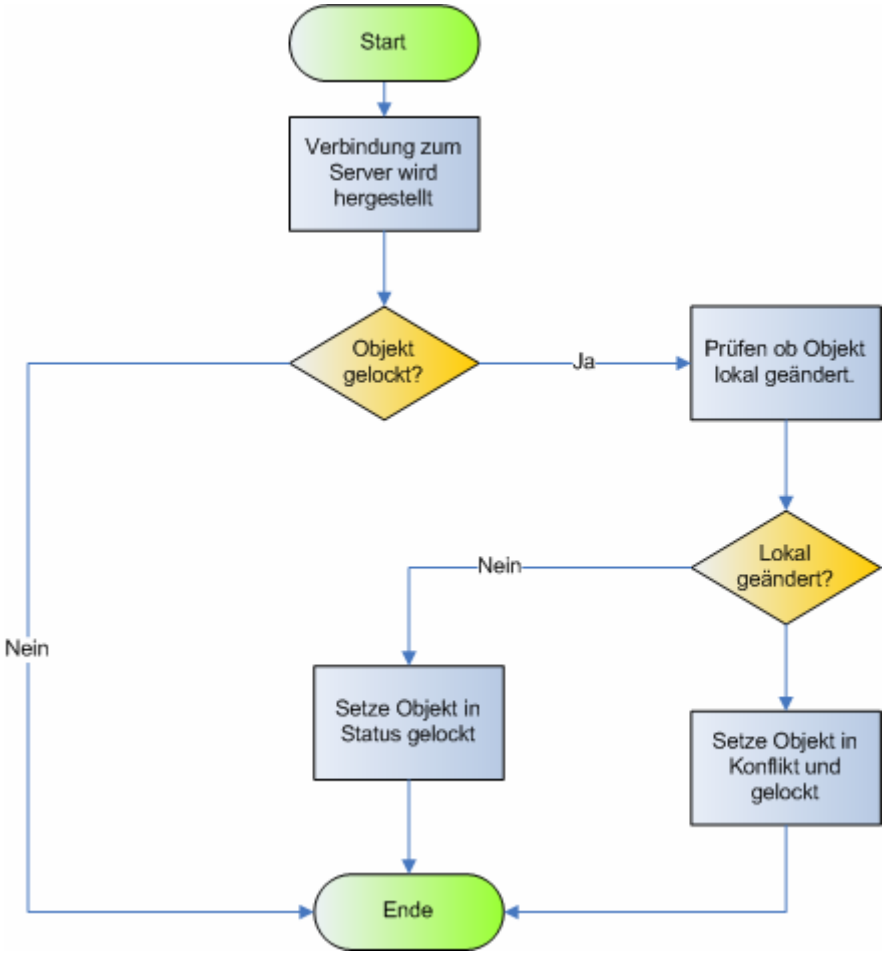

Abbildung 38 - Flussdiagramm: Locking im Offline-Modus

Neben den Problemen bezüglich der Konflikterkennung und dem clientseitigen Locken von Objekten muss der Offline-Modus in der Lage sein das in Kapitel 4.6 behandelte Rechtesystem auch ohne Verbindung zum Server umsetzen zu können. Auch hierbei müssen die Informationen einen Neustart der IDE überstehen. Der Editor muss also, auch wenn er im Offline-Modus gestartet wird, sämtliche Rechte, die der Nutzer zu dem Zeitpunkt hatte, als die Netzwerkverbindung noch bestand, simulieren können, damit Nutzer nicht plötzlich in vollem Funktionsumfang das Modell editieren können. Alle drei Probleme (Konflikte, Locking und Rechte) an ihren spezifischen Stellen im Quellcode anpassen zu wollen, wäre vom Aufwand her und softwarearchitektonisch nicht tragbar. Aus diesem Grund wurde eine Komponente konzipiert, welche ausschließlich für die Verwaltung des Systems im verbindungslosen Betrieb zuständig

ist. Diese Komponente, der *Offline-Server*, hält alle Daten des Online-Servers vor und kann diesen bei Verbindungsausfall simulieren. Genauer gesagt ist es für den Editor vollkommen unerheblich, ob eine Netzwerkverbindung besteht, da der Offline-Server sämtliche Anfragen an den Server simulieren kann. Hierzu synchronisiert sich der Offline-Server bei jedem Polling-Intervall mit dem Online-Server und persistiert alle notwendigen Informationen, damit die Informationen auch einen lokalen Systemausfall überstehen. Startet die IDE neu, so prüft der Offline-Server ob eine Netzwerkverbindung besteht. Kann eine Verbindung aufgebaut werden, so werden die Daten abgeglichen, andernfalls werden alle Informationen aus dem File-System geladen und der Offline-Server mit diesen initialisiert.

Technisch wird die Transparenz für den Editor durch das in Kapitel 4.2.1 entworfene Adapter-Konzept realisiert. Neben den RemoteConnection-Instanzen, welche die Verbindung für spezifische Protokolle aufbauen, wird eine weitere Instanz erzeugt, um eine virtuelle Verbindung zum Offline-Server herzustellen. Diese RemoteConnection implementiert ebenfalls die Schnittstelle **`DawnRemoteConnection`**, kann also genau wie die protokollspezifischen Verbindungen genutzt werden. Allerdings baut diese Connection nicht über ein Protokoll eine Verbindung zu einem realen Server auf, sondern startet lokal den Offline-Server und übernimmt dessen Verwaltung.

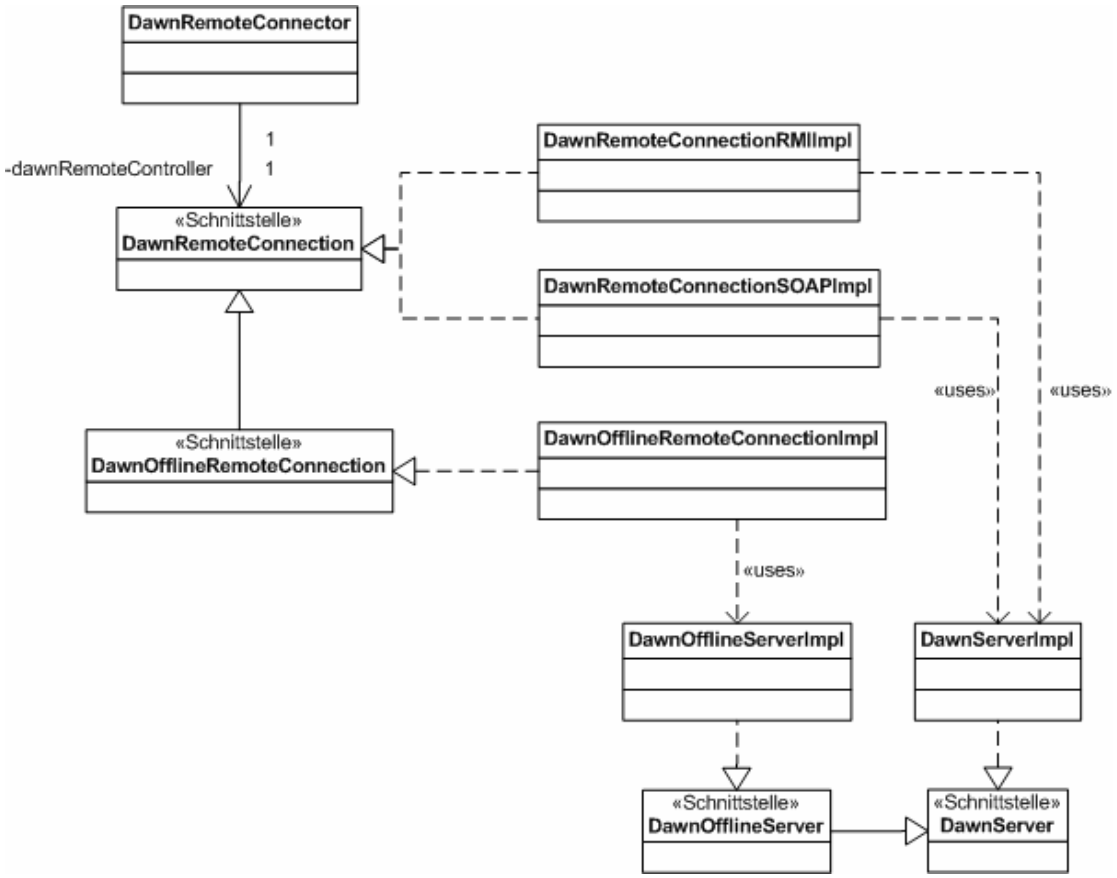

Abbildung 39 – Klassendiagramm: Adapter mit Offline-Klassen

Abbildung 39 zeigt die Integration der Offline-Komponenten in das bestehende Schema. Die Schnittstelle **`DawnRemoteConnection`** wird durch die Schnittstelle **`DawnOfflineRemoteConnection`** erweitert. Die Implementierung des DawnRemoteControllers greift nicht, wie die anderen Implementierungen, auf den realen Server zu, sondern auf die lokal laufende Instanz. Die lokale Implementierung des Servers (**`DawnOfflineServerImpl`**) implementiert ein erweitertes **`DawnServer`** Interface. Dies geschieht, da diese Schnittstelle zusätzliche Methoden zum lokalen Speichern und Laden der persistierten Daten zur Verfügung stellen muss, welche allerdings ausschließlich vom **`DawnRemoteConnector`** genutzt werden.

4.6 Authentifizierung und Autorisierung

Um die Diagramme vor unbefugtem Zugriff schützen und die Bearbeitungsrechte einzelner Nutzer einschränken zu können, muss Dawn über ein System zur Authentifizierung und Autorisierung verfügen. Zusätzlich sollen die Rechte auf die Anwendungsfälle der einzelnen Editoren abgestimmt werden. Dawn soll allerdings auch in sicheren Umgebungen eingesetzt werden können, in denen keine Autorisierung benötigt wird. Um diesen Anforderungen gerecht zu werden, wurde für Dawn ein generisches, rollenbasiertes Nutzer-Management Konzept entwickelt.

4.6.1 Rollen und Operationen

Das in Abbildung 23 dargestellte Anwendungsfalldiagramm zeigt vier Rollen. Einen Betrachter, der ausschließlich Lese-Rechte besitzt, einen Projekt-Nutzer, einen Projekt Administrator und einen Systemverwalter. Sie besitzen unterschiedliche Rechte und Sichten in dem System. Da Dawn aber eine Erweiterung von Editoren ist, deren Zielbestimmung vom Entwickler festgelegt wird, verbietet sich von vornherein eine statische Zuordnung von Rollen. Die Vielfalt kann hierbei von Beschränkungen für jede einzelne Operation bis hin dazu reichen, dass der Endanwender gänzlich auf Authentifizierung und Autorisierung verzichten möchte. All diese Anwendungsfälle müssen von Dawn abgedeckt werden. Zusätzliche Erweiterungen, wie zum Beispiel explizite Rechte für das Löschen von Kanten, sollen später einfach in das System eingebracht werden können. Folglicherweise muss bei der Entwicklung des Rechtesystems ein generischer Ansatz genutzt werden.

Basis dieses generischen Nutzer- und Rechtesystems stellt hierbei ein Rollenkonzept dar, welches anhand von Rollen-Rechten die Zugriffrechte eines Nutzers innerhalb

eines Projektes darstellt. Im Kern kann einem Nutzer in einem Projekt nur eine Rolle zugeordnet werden. Er kann sich aber innerhalb verschiedener Projekte in unterschiedlichen Rollen befinden, also unterschiedlich Rechte haben.

Jede Rolle kann mit einer beliebigen Anzahl an Operationen verknüpft werden. Operationen stellen hierbei ausführbare Aktionen innerhalb des Systems dar. In der prototypischen Implementierung werden drei Operationen definiert – *insert*, *update*, *delete* – jeweils für das Einfügen, Änderung und Löschen von Objekten. Späteren Entwicklungen steht es frei diese Liste zu erweitern, zum Beispiel um das bereits angesprochene explizite Recht zum Löschen von Kanten. Operationen sind dabei so elementar, dass sie nur zwei Zustände kennen – ein oder aus. Diese Zuordnung zwischen Rolle und Operation ist in einer Liste verwaltet (**`Map<int, boolean>`**, vgl. Abbildung 40). Durch diese Zuordnung kann eine Rolle zu jeder Zeit um weitere Operationen erweitert (**`addOperation(int, boolean)`**) und einfach von der Rolle abgefragt werden (**`boolean isAllowed(int)`**).

4.6.2 UserManager Konzept

Um die Verwaltung der Nutzer und ihrer Rollen von den Projekten zu entkoppeln, wurden die nötigen Funktionalitäten in eine eigene Komponente, den *UserManager*, ausgegliedert. Der Nutzer-Manager sorgt also für die Zuordnung zwischen den Rollen und den Nutzern des Systems. Komponenten, wie Projekte und der Dawn-Server, delegieren ihre Nutzerverwaltung an eine jeweilige UserManager-Implementierung. Diese erweitern immer das Interface **`UserManager`** [vgl. Abbildung 40]. Somit wird gewährleistet, dass grundlegende Management-Funktionen zur Verfügung gestellt werden.

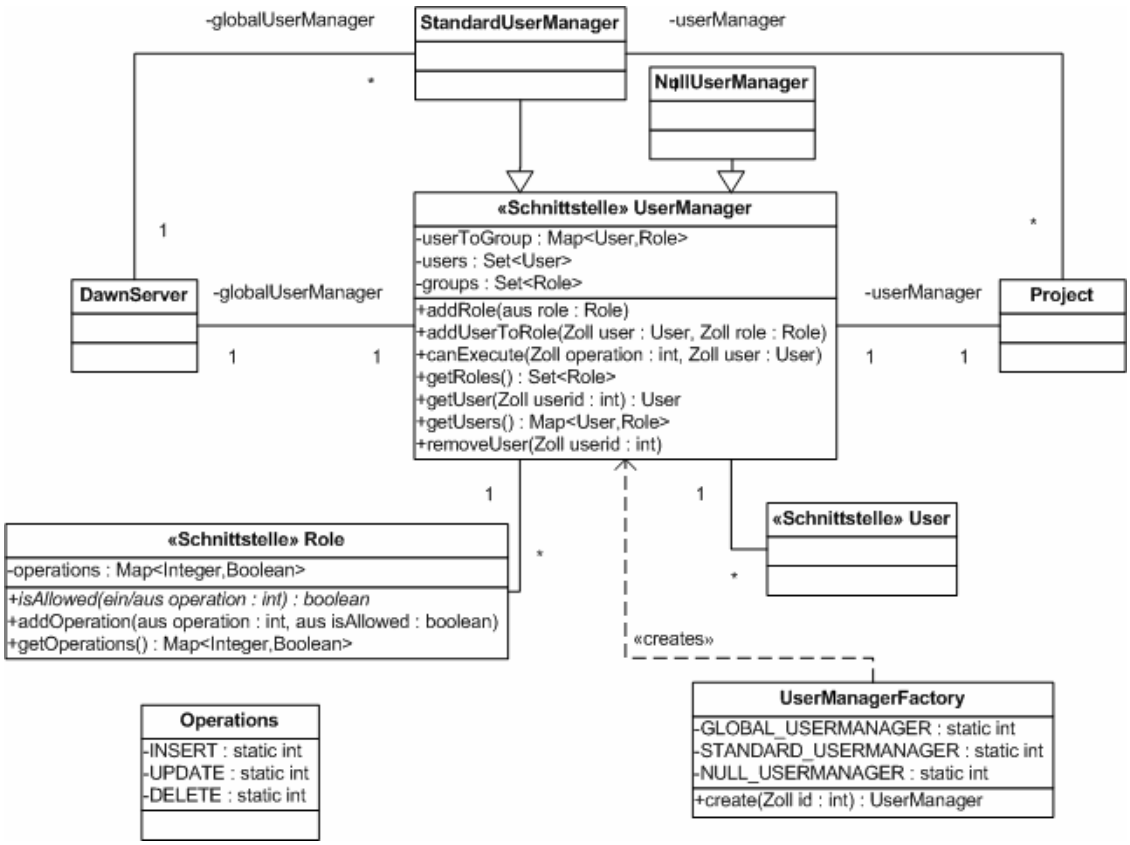

Abbildung 40 - Rechtesystem in Dawn

Der UserManager verwaltet eine Liste von Nutzern, eine Liste von Rollen und die Zuordnung zwischen beiden. Basis für den Zugriff auf die Rechte der Nutzer stellt die Methode **`boolean canExecute(int, User)`** dar, welche als Parameter den Identifikator einer Operation und eine Referenz auf einen Nutzer übergeben bekommt. Werden die Rollen um weitere Operationen erweitert, können diese ohne Änderung des UserManagers abgefragt werden. Dawn-Programmierer können an dieser Stelle das UserManager-Interface erweitern, um eigene Methoden zur Verfügung zu stellen. In der einfachen Implementierung nutzt jede Komponente den **`StandardUserManager`** als Implementierung der UserManager-Schnittstelle. Diese wird mit den oben aufgeführten Rollen initialisiert. Hierbei übernimmt eine Fabrik die Erzeugung der UserManager-Implementierung für das System, damit sich die einzelnen Komponenten nicht um die korrekte Einrichtung der Rechte kümmern müssen.

Vorteil dieser Implementierung ist die Entkopplung der Projekte von der Nutzerverwaltung. Für jedes einzelne Projekt ist es transparent, welcher UserManager sich um die Rechte des Projektes kümmert, da es lediglich auf die entsprechende Instanz referenziert. Damit kann auch die Zuweisung der User-Manager zu den Projekten flexibel erfolgen. Beispielsweise kann jedes Projekt über seinen eigenen User-Manager verfügen [vgl. Abbildung 41, Fall 1]. Dadurch können die Rechte autark

vom restlichen System verwaltet werden. Es ist aber ebenso möglich für einzelne Gruppen von Projekten einen gemeinsamen UserManager zur Verfügung zu stellen [vgl. Abbildung 41, Fall 2]. Änderungen an dessen Rechten und Nutzern hätten somit direkte Auswirkung auf alle Projekte. Dieser Gedanke kann bis zu dem Punkt weitergeführt werden, dass alle Projekte einen gemeinsamen Nutzermanager referenzieren, der die gesamte Verwaltung übernimmt [vgl. Abbildung 41, Fall 3].

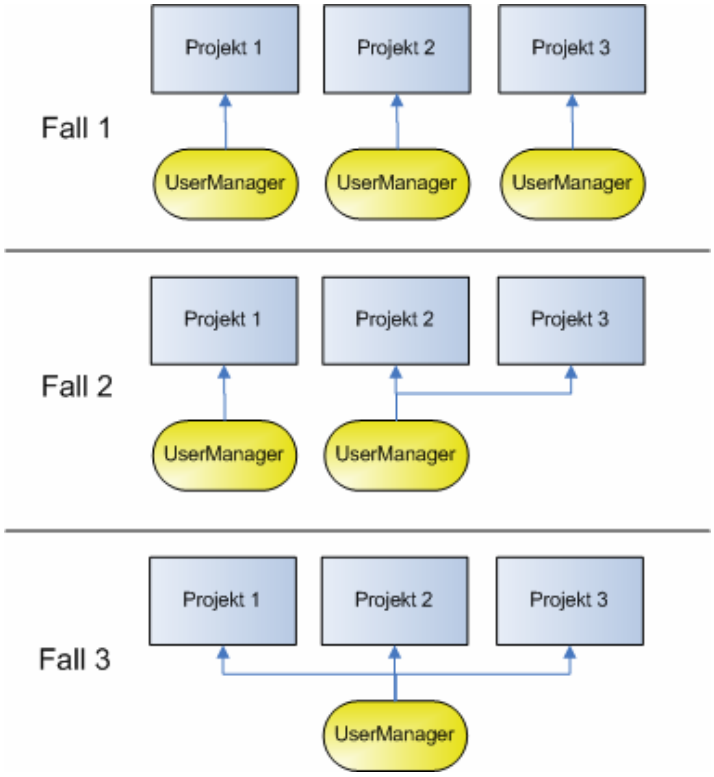

Abbildung 41 - Unterschiedliche Referenzierung von UserManagern

4.6.3 GlobalUserManager und NullUserManager

Nutzer und Gruppen werden global vom Dawn-Server verwaltet und bei Bedarf als Referenzen den einzelnen Projekten übergeben. Hierbei wurde das für die Projekte entwickelte UserManager-System auch für die globalen Information genutzt, um die Funktionalitäten an eine eigenständige Komponente zu delegieren. Der Dawnserver nutzt hierfür einen *StandardUserManager*, der im Serverkontext *als GlobalUserManager* bezeichnet wird. Dieser verhält sich wie der UserManager, der den Projekten zugeordnet wird, allerdings werden seine Rechte auf Ebene des Servers *interpretiert*. Das bedeutet, dass serverspezifische Rechte, wie das Erstellen von Projekten oder das Administrieren des Servers von einem globalen UserManager verwaltet werden, auf den nur der Server Zugriff hat.

Dawn bietet natürlich auch die Möglichkeit losgelöst von dem Rechtesystem zu arbeiten. Beispielsweise, wenn das System in einer sicheren Umgebung zum Einsatz kommt, in dem keine Authentifizierung benötigt wird. Aus diesem Grund wird eine besondere Implementierung des UserManager Interfaces erzeugt – der **`NullUserManager`**. Dieser quittiert jede Rechte-Anfrage mit dem Wert *true*. Dadurch werden alle Rechte im System freigeschaltet. Es sind also egal, wer eine Anfrage tätigt, alle Operationen erlaubt. Der Einsatz des NullUserManagers hat den Vorteil, dass keine Änderungen am System vorgenommen werden müssen. Für die Projekte ist die Anfrage transparent, da allein der UserManager über die Rechte entscheidet.

4.7 Datenbank und Persistenz

Die Persistenz von Daten in Dawn geschieht zweiseitig. Alle Informationen zu den Projekten und den Konfigurationen des Servers werden serverseitig in einer Datenbank persistiert. Daneben bleibt die Speicherung der Modellinformationen auf dem Client unberührt. Dieser verhält sich weitestgehend wie der normale GMF-Editor.

Auf dem Server werden alle Daten mit Hilfe eines *Objektrelationalen-Mappers* (*ORM*) in einer Datenbank hinterlegt. Der Vorteil hierbei liegt in dem hohen Abstraktionsniveau, welches objektrelationale Mapper bei der Speicherung von Daten mitbringen. Zum einen kann der Software-Architekt sich allein auf die Entwicklung der objektorientierten Struktur konzentrieren und muss nicht auf die konzeptuelle Konsistenz zwischen objektorientiertem Modell und dem Datenbankendesign achten. Zum anderen kann mit Hilfe von OR-Mappern eine Abstraktion von der genutzten Datenbank erreicht werden, was Datenbankbetriebssysteme einfach austauschbar macht. Der in dieser Arbeit verwendete objektrelationale Mapper Hibernate unterstützt beide Features. Das Entity Relationship Diagramm in Abbildung 42 zeigt alle zu speichernden Daten an. Auf die Konfiguration und Nutzung von Hibernate wird in Kapitel 5.5 eingegangen.

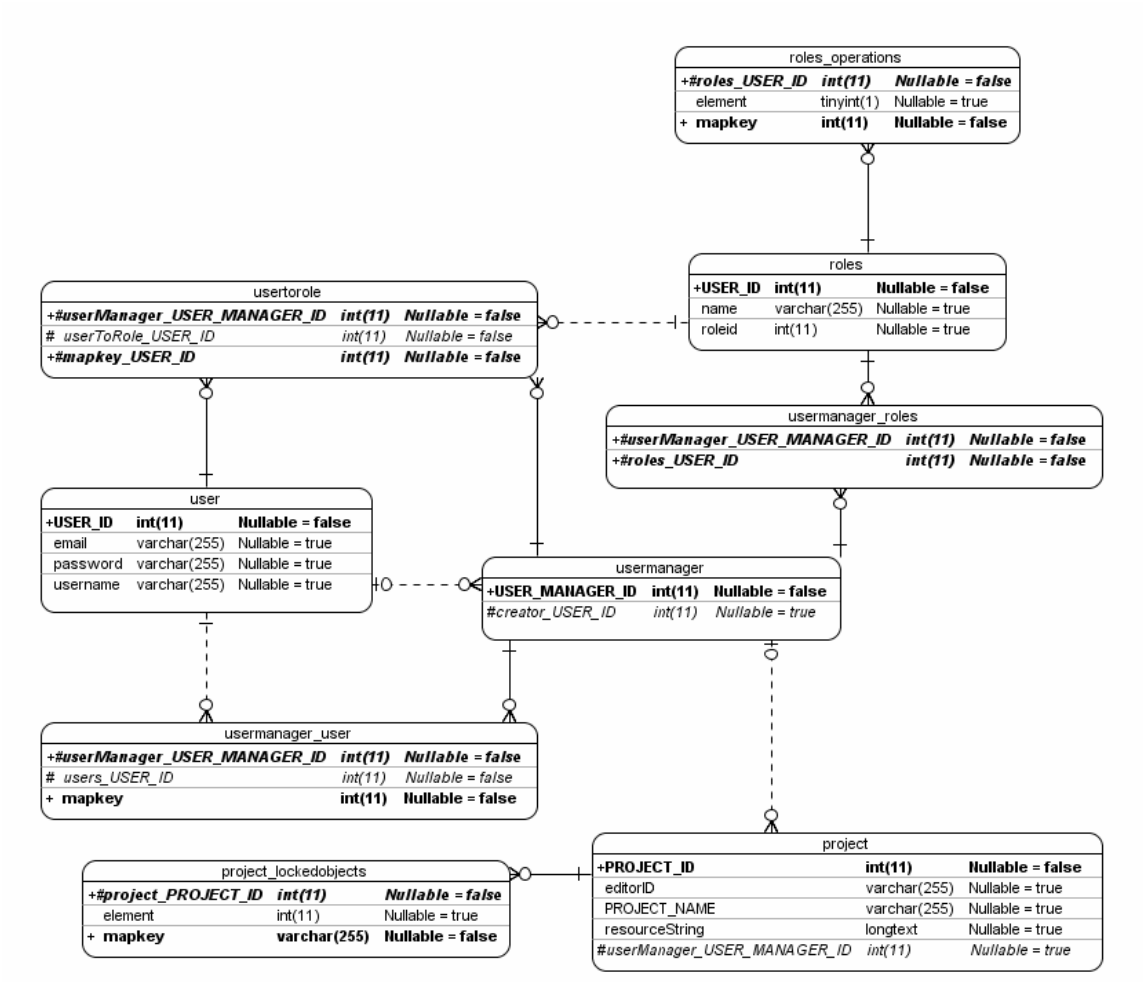

Abbildung 42 - ERM Diagramm des Persistenzmodells

Neben Hibernate als Framework für die objektrelationale Umsetzung der Java-Objekte auf relationale Tabellen, wird *MySQL* als Datenbankbetriebssystem zum Einsatz kommen. MySQL ist aber durch den Einsatz von Hibernate jeder Zeit austauschbar. Um das Mapping vorzunehmen, gibt es in Hibernate zwei verschiedene Möglichkeiten. Zum einen können XML-Konfigurationsdateien geschrieben werden, in denen das Mapping vermerkt ist. Diese Variante hat den Vorteil, dass diese Zuordnung unabhängig vom kompilierten Bytecode ist. Soll im laufenden System eine einfache Änderung durchgeführt werden, kann dies in den XML-Konfigurationsdateien geschehen. Allerdings ist das Schreiben dieser Dateien mit steigender Objekt-Anzahl extrem zeitaufwändig. Zur Vereinfachung bietet sich die Nutzung von Frameworks wie beispielsweise *XDoclet*[36] an, welches über Annotations-ähnliche Vermerke in den Kommentaren der Klassen die XML-Dateien erzeugen kann. Eine weitere Methode, um das Mapping zu realisieren, ist das Nutzen von Java-Annotations. Vorteil dieser Variante ist es, dass keine zusätzlichen Dateien genutzt werden müssen. Die Mapping-

[36] http://xdoclet.sourceforge.net/

Informationen sind in den Klassen und somit auch im Byte-Code hinterlegt. Ein entscheidender Vorteil gegenüber der Nutzung von XDoclet ist es, dass Hibernate-Annotations automatisch immer konform zum Hibernate-Framework sind, da diese von dessen Entwicklern verwaltet werden. XDoclet dagegen wird von anderen Entwicklern vorangetrieben, weshalb Änderungen in Hibernate nicht immer sofort in XDoclet umgesetzt werden. Aus diesem Grund wurden alle zu persistierenden Klassen mit Hibernate-Annotations versehen.

Wie bereits erwähnt wird an der Speicherungsart der Ressource auf dem Client keine Änderung vorgenommen. Es werden aber zusätzliche Meta-Informationen gespeichert, um das System auch im Offline-Modus [vgl. 4.5] korrekt betreiben zu können. Hierzu zählen der zuletzt bekannte Stand des Server-Modells (LastResource) und die Rechte-Informationen. Diese Informationen werden clientseitig in einem Meta-Informationen-Verzeichnis abgelegt, welches für jedes Diagramm erstellt wird. Ein Vorteil dieser redundanten Speicherung ist zum einen, dass das Diagramm jederzeit komplett losgelöst von Dawn arbeiten kann. Möchte der Nutzer das Diagramm nicht mehr kollaborativ betreiben, kann er es mit dem normalen GMF-Editor für das Diagramm öffnen und gewohnt weiter arbeiten. Ein anderer Vorteil ist, dass die Daten redundant vorliegen. Im Fall eins kompletten Verlustes aller Daten auf dem Server könnte über geeignete Algorithmen in Kombination mit manuellen Aktionen das Modell aus den Beständen der Clients rekonstruiert werden.

4.8 Webkomponenten

Dawn bietet zwei webbasierte Schnittstellen für die Benutzer an. Zum einen eine Konfigurationsoberfläche, mit deren Hilfe Projekte, Nutzer und Rechte verwaltet werden können, und zum anderen den Web-Viewer, der es erlaubt die Bearbeitung des Diagramms unabhängig von einer installierten Eclipse-Instanz verfolgen zu können.

4.8.1 Web-Viewer

Bei dem Web-Viewer handelt ist sich, wie in Kapitel 3.6.3 beschrieben, um eine webbasierte Repräsentation eines GMF-Diagramms. Im Rahmen dieser Arbeit wird lediglich eine nicht-editierbare Version entwickelt. Allerdings geschieht dies unter dem Fokus, dass später dieses Modell zu einem webbasierten Editor erweitert werden kann. Folglich muss diese spätere Ausbaustufe bei der Konzeptionierung, soweit möglich, mit betrachtet werden.

4.8.1.1 Darstellung der Diagramme

Für eine einfache Ansicht eines GMF-Diagramms im Browser hätte die Export-Funktion von GMF-Editoren genutzt werden können. Diese basiert allerdings darauf, dass vom Diagramm entweder ein Screenshot oder ein SVG-Export[37] erzeugt wird, welcher dann in eine HTML-Seite eingebettet wird. Zumindest für die erstere Variante hätte eine eigene grafische Eclipse-Instanz auf dem Server gestartet werden müssen. Ein weitaus größerer Nachteil dieses Ansatzes ist es aber, dass er keine Möglichkeit bietet später einen webbasierten Editor zu erstellen, da die einzelnen Elemente in einem Screenshot weder skaliert noch verschoben werden können. Aus diesem Grund kam nur die Entwicklung einer eigenständigen Lösung in Betracht.

Um GMF-Elemente in einem Browser darstellen zu können, muss eine grafische Entsprechung der einzelnen Views erstellt werden, wie sie der GMF-Entwickler spezifiziert hat. Da jeder GMF-Editor auf Java basiert, könnte ein Ansatz versuchen den Editor innerhalb eines Java-Applets im Browser ablaufen zu lassen. Allerdings würde, aufgrund der starken Abhängigkeiten zur Eclipse-Umgebung, das Applet schnell zu groß werden, um es einfach nutzen zu können. Dadurch würde die Performance stark reduziert und einhergehend damit ebenso die Akzeptanz der Nutzer. Außerdem setzt der Einsatz eines Java-Applets die Installation einer Java-Virtual-Maschine in der passenden Version auf dem Client voraus, was gerade für den Einsatz des Web-Viewers nicht der Fall sein soll, da dieser u.a. im Hinblick auf minimal konfigurierte Clients entwickelt wird. Die Nutzung von Java-Applets scheidet folglicherweise aus.

Ähnliche Argumente lassen sich gegen eine Implementierung mittels Flash anbringen. Auch hier muss der Browser in der Lage sein, die flashbasierten Animationen anzeigen zu können, was in jedem Fall voraussetzt, dass der Webbrowser das Flash-Plugin installieren und ausführen kann. Daneben besteht auch die Anforderung, dass die Web-Repräsentationen generisch aus dem GMF-Modell generiert werden können [vgl. 4.11]. Das proprietäre Format von Flash ist dabei denkbar ungünstig geeignet, um die Views aus einem Modell zu generieren.

Diese Überlegungen resultieren in die Nutzung von HTML in Verbindung mit JavaScript und CSS (*Cascading Style Sheets*) als Grundlage für den Web-Viewer. HTML und JavaScript sind textbasierte Formate, lassen sich also sehr leicht generieren. Außerdem bieten sie die nötige Performance und Systemunabhängigkeit. Durch *Ajax (Asynchronous JavaScript and XML)* als Kommunikationstechnologie ist es außerdem möglich mit einem Backend zu kommunizieren, was die spätere Erweiterung der

[37] Scalable Vector Graphics; XML basierte Anwendung zur Darstellung von Vector-Grafiken [vgl. Henning 2003, S. 451]

Implementierung zu einem Web-Editor ermöglicht. Einziger Nachteil von JavaScript ist die Teilweise nicht standardkonforme Implementierung in den einzelnen Browsern und die dadurch bedingte Erhöhung des Implementierungsaufwandes. Da einige Browser eigene Interpretationen oder Implementierungen von JavaScript nutzen (zum Beispiel der Internet Explorer mit JScript), kann es dazu kommen, dass bestimmte Systemteile für die einzelnen Browser angepasst werden müssen. Um dieses Problem im Rahmen dieser Arbeit zu vermeiden und den Fokus auf die reine Entwicklung legen zu können, werden alle Konzepte ausschließlich auf dem Firefox in der Version 3.x entwickelt.

Zur Darstellung der verschiedenen Kanten und Knoten, müssen diese Elemente mit denen in HTML zur Verfügung gestellten Mitteln gezeichnet werden. Leider sind diese Möglichkeiten sehr begrenzt, können aber durch die Nutzung von CSS durchaus erweitert werden, was aber meist mit einem erhöhtem Aufwand und einem nicht geringen Maß an Kreativität einhergehen muss. Auch muss für die spätere Nutzung als Editor mit bedacht werden, dass die einzelnen Elemente sich auf der browserbasierten Zeichenfläche verschieben lassen müssen. All diese Anforderungen selbst entwickeln und umsetzen zu wollen, würde den Rahmen dieser Arbeit sprengen. Nach eingehender Recherche fand sich ein Framework, welches einen großen Teil der geforderten Anforderungen bereits umsetzt. *OpenJacob*[38] ist ein freies Framework zur Implementierung von Rich Internet Applications[39]. Ein Subprojekt von OpenJacob ist *OpenJacob Draw2D*, welches versucht Draw2D, die Zeichen-API von Eclipse [vgl. 2.8.5], mit Hilfe von Web-Technologien im Browser zu realisieren. Dabei können bereits komplexere Knoten und Kanten gezeichnet werden, was das API zu einer optimalen Grundlage für die eigene Entwicklung macht.

Für jedes View-Element eines GMF-Editors wird eine passende JavaScript-Klasse implementiert, welche dieselben Eigenschaften bezüglich der Darstellung aufweist wie die Java-Klasse innerhalb von Eclipse.

Kern der Darstellung im Web wird ein einfaches Servlet bilden, welches die Umwandlung des im Dawn-Server vorgehaltenen Modells in eine Web-Repräsentation vornimmt. Dabei greift es über den Server auf das entsprechende Projekt zu und liest die aktuellen Informationen aus. Das Servlet nutzt also auf die in dem View-Modell hinterlegt Informationen für die grafische Darstellung. Durch diese Informationen können für jeden Knoten und jeder Kante darstellungsspezifische Informationen wie Position, Größe, Hintergrundfarbe etc. ausgelesen werden. Zusätzlich muss ermittelt werden, welcher genaue Knoten- bzw. Kantentyp gerade vorliegt, damit die passende

[38] http://www.openjacob.org/
[39] Rich Internat Apllications stellen das webbasierte Gegenstück zu RCP dar. Es wird versucht komplett eigenständige Anwendungen im Browser zu realisieren.

JavaScript-Klasse geladen werden kann. Diese wird mit den Darstellungsinformationen instanziiert und zur Anzeige gebracht. Das Servlet übernimmt dabei die Generierung des JavaScript-Quellcodes. Im Ergebnis entsteht ein webbasiertes Abbild des aktuellen Zustandes des GMF-Modells.

Damit eines Zuordnung zwischen dem View-Modell und den JavaScript-Figures vorgenommen werden kann, werden die in Kapitel 2.8.7 beschriebenen ElementTypes genutzt. Diese ElementTypes sind die Grundlage, um die JavaScript-Klassen ihren View-Elementen zuordnen zu können. Diese erfolgt in einer Meta-Datei (*config.xml*), welche jedem Diagramm-Typ innerhalb des Servers zugewiesen ist. Listing 3 zeigt eine vereinfachte config.xml, in der der Element-Type *2001* auf das JavaScript-View-Element **`AnAttributeLabelFigure`** gemapped wird. Zusätzlich werden noch die für die Anzeige in der View genutzten Attribute in dem Mapping vermerkt. Dies wird benötigt, da JavaScript im Gegensatz zur GMF-Implementierung keine Informationen darüber besitzt, welche Attribute es anzeigen soll. Innerhalb von GMF wird über die EditParts eindeutig definiert, welche Attribute innerhalb der View angezeigt werden sollen. Der EditPart hält die Referenz auf die anzuzeigenden Figure-Elemente. Auf dem Server fehlt allerdings diese Zuordnung, da der Server keine EditParts, sondern nur das reine Modell (semantic und notational) zur Verfügung hat. Aus diesem Grund werden die Attribute, welche in den JavaScript-Figuren angezeigt werden sollen in der Konfigurationsdatei vermerkt. Hierbei handelt es sich um die Namen der Attribute des EMF-Objektes, welches mit der View verknüpft ist. Die Werte dieses Attributs können mit Hilfe von Dynamic EMF ausgelesen und zur Anzeige gebracht werden. Neben den Attributen enthält diese Datei auch das *View-Pattern* für die Darstellung von Attributen. Dieses View-Pattern wird im GMF-Modell genutzt, um Einfluss darauf zu nehmen, in welcher Reihenfolge Attribute angezeigt werden sollen. Die zusätzliche Angabe von Trennzeichen ermöglicht eine flexible Darstellung der Informationen.

```
1   <?xml version="1.0" encoding="ISO-8859-1"?>
2   <figureMappings>
3   <figureMapping type="2001">
4   <javaScriptClass
5   name="org.mftech.diagram.uml.clazz.AnAttributeLabelFigure" />
6               <viewAttribute name="accessright" />
7               <viewAttribute name="dataType" />
8               <viewAttribute name="name" />
9               <viewPattern name="{0} {2}:{1}" />
10        </figureMapping>
11  </figureMappings>
```

Listing 3 - Mapping von ElementType zur JavaScript-View

Im aufgeführten Listing wird zuerst das erste Attribut, dann das dritte und zuletzt das zweite Attribut angezeigt. Hierbei werden die letzten beiden Attribute durch einen Doppelpunkt voneinander getrennt [Listing 3, Zeile 9]. Damit die Web-Darstellung und die Darstellung im Eclipse identisch sind, verarbeitet das Diagramm-Servlet zusätzlich diese Information und zeigt die Ausgabe richtig formatiert an.

In Analogie zu dem in Abbildung 19 dargestellten Architektur-Modell von GMF, zeigt Abbildung 43 das Architekturmodell des Web-Viewers. Hierbei übernimmt das Diagramm-Servlet die Funktion des Controllers. Mit den Konfigurationsinformationen, welche es aus der config.xml bezieht, kann es die einzelnen JavaScript-Figures instanziieren. Dabei greift es auf das auf dem Server persistierte Datenmodell, bestehend aus den GMF-Views und dem EMF-Modell, zu. Das Diagramm-Servlet verbindet also das GMF-Datenmodell mit der Web-Darstellung.

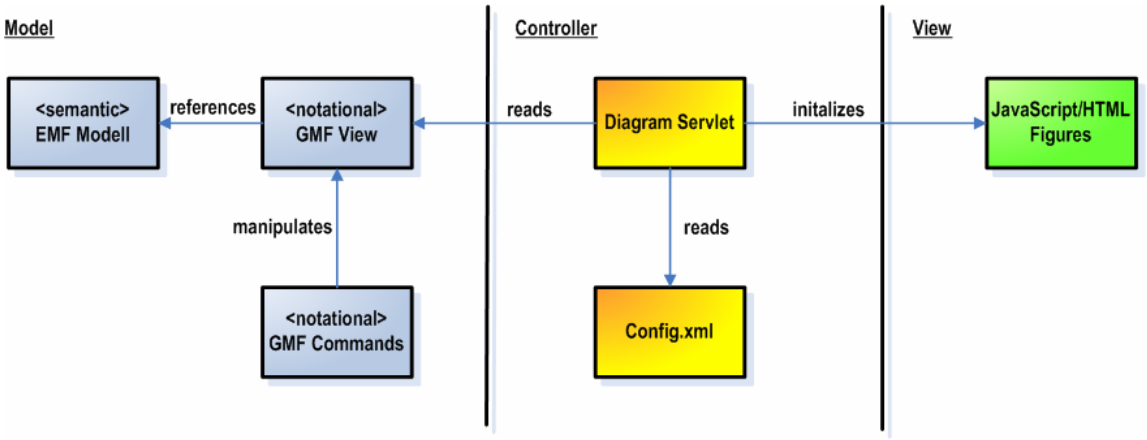

Abbildung 43 - Web-Viewer Architektur

Da es viel zu umständlich und gegen das Konzept der modellgetriebenen Softwareentwicklung wäre, die einzelnen JavaScript-Klassen per Hand zu implementieren, können diese basierend auf dem GMF-Modell automatisch generiert werden. In Kapitel 4.11 wird erläutert, wie diese Generierung vonstatten geht.

4.8.1.2 Kommunikation

Die Darstellung des Diagramms im Browsers wird durch den Aufruf des Diagramm-Servlets ermöglicht. Dadurch wird der aktuelle Zustand des Diagramms an den Client übertragen. Damit der Nutzer aber ohne eigene Aktionen jeder Zeit Änderungen betrachten kann, muss in regelmäßigen Abständen die Ansicht aktualisiert werden. Der einfachste Ansatz bestünde darin einen Meta-Refresh in die HTML-Seite einzubetten. Dieser kann so gestaltet werden, dass die Seite sich nach einem vordefinierten Zeitintervall erneut lädt. Die Änderungen würden somit sichtbar. Allerdings werden

dabei immer alle Informationen übertragen, egal ob sich Änderungen ergeben haben, oder nicht. Dies resultiert sowohl in einer erhöhten Last auf dem Server, der die Daten generieren muss, als auch in einer erhöhten Netzlast.

Um die Last für den Server und das Netzwerk zu reduzieren, darf der Client also nur den Inhalt der Seite aktualisieren, wenn sich die Daten geändert haben. Dawn nutzt hierfür einen Ajax-Request, um die HTML-Seite über Modellbestandsänderungen zu informieren. Jedes Projekt verfügt hierzu über eine Versionsnummer. Wird die Seite initial geladen, so wird diese Versionsnummer mit übertragen. Findet nun eine Änderung auf dem Server statt, wird diese Nummer inkrementiert. Über einen periodischen Ajax-Request erfragt der Browser die aktuelle Versionsnummer und kann über einen Vergleich mit dem lokalen Identifikator herausfinden, ob sich das Diagramm geändert hat. Daraufhin löst er einen Refresh der Seite aus, wodurch der neue Modellbestand an den Browser übertragen wird.

4.8.2 Web-Frontend

Das Konfigurations-UI wird als klassische Weboberfläche implementiert. Hierbei müssen Methoden für die Verwaltung von Nutzern, die Verwaltung der Rechte und die Möglichkeit der Bearbeitung von Projekten geschaffen werden. Dabei baut die Oberfläche auf dem Web-Framework *Struts* in der Version 1.3 auf. Struts hat den Vorteil, dass es sehr weit verbreitet ist [vgl. 3.3] und eine einfache Methode zur Umsetzung des MVC-Patterns im Web bietet. Zusätzlich stellt Struts mit dem *Tiles Framework* eine praktische Template-Architektur zur Verfügung, um die Web-Seiten modular aufbauen zu können. Das Web-Frontend nutzt ebenfalls das Rechte-Konzept des Dawn-Servers. Nicht alle Bereiche dürfen von allen Nutzern eingesehen werden. So ist das Löschen oder das Editieren von Nutzern ausschließlich dem globalen Administrator vorbehalten. Projektinterne Änderungen können allerdings von den Administratoren der einzelnen Projekte ausgeführt werden. Abbildung 44 stellt das Konzept der Web-GUI in einer Sitemap dar. Hierbei sind Bereiche, die nur vom Server-Administrator genutzt werden können rot dargestellt. Die Administrationsseiten, die auch von den Projekt-Managern benutzt werden können sind gelblich hinterlegt. Alle anderen Seiten können von allen Nutzern ausgeführt werden (blau).

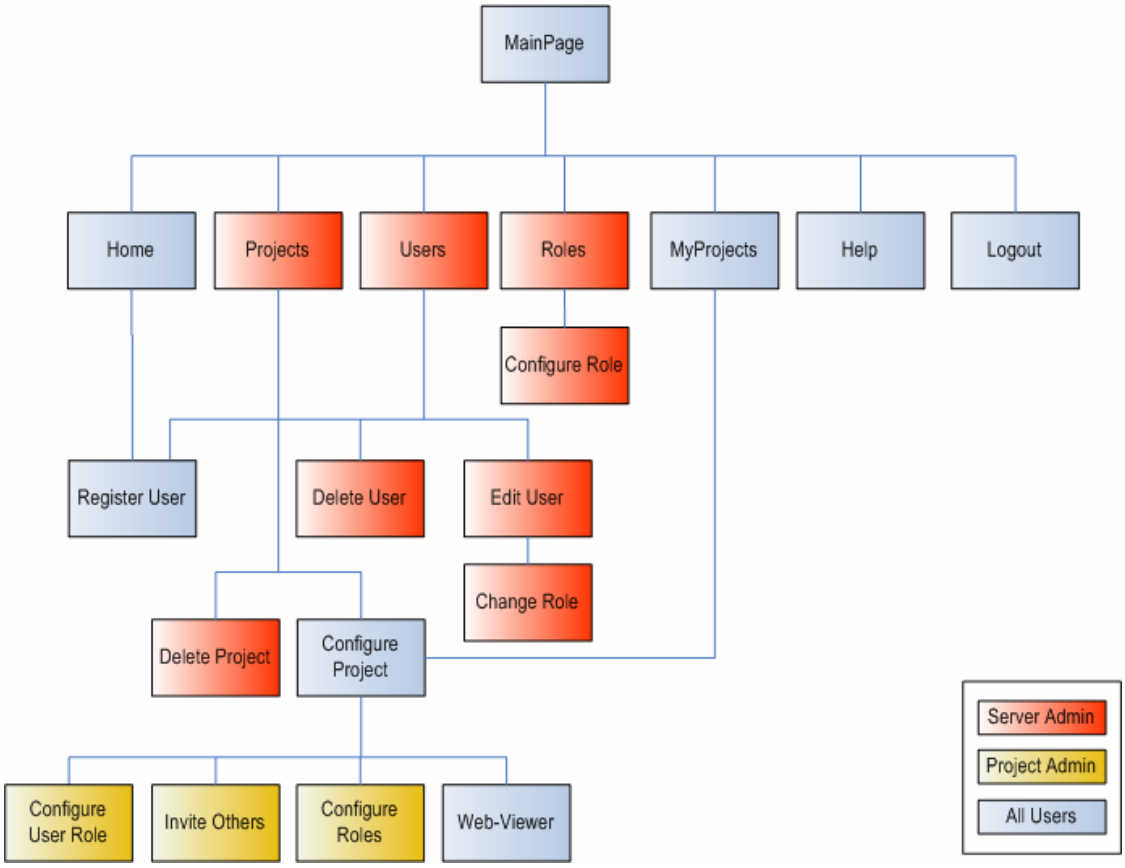

Abbildung 44 - Sitemap - Web-GUI

Die Hauptseite der Verwaltungs-UI bietet verschiedene Hauptmenüs an. Das Menü *Projects* kann ausschließlich vom Administrator genutzt werden. Es liefert eine Liste von alle auf dem Server verfügbaren Projekten und gibt zusätzlich die Möglichkeit die Projekte zu löschen. Über den *Configure Projects* Menüpunkt kann der Administrator den Nutzern des Projekts eine Rolle zuweisen (*Configure User Roles*), neue Nutzer zu einem Projekt einladen (*Invite Others*) oder die Rechte einzelner Rollen des UserManagers verwalten (*Configure Role*s). Zusätzlich enthält es einen Verweis zu dem Web-Viewer, mit dem das Projekt webbasiert betrachtet werden kann. Das Menü *MyProjects* bietet, bis auf die Löschfunktionalitäten, dieselben Funktionen für die Projekte eines Nutzers an. Die Menüs *Configure User Role*, *Invite Others* und *Configure Roles* können allerdings nur genutzt werden, wenn der Nutzer auch Administrator des Projektes ist.

Neben der Projektverwaltung ist es dem Administrator des Servers erlaubt alle Nutzer aufzulisten (*Users*) und Änderungen an ihren Rechten vorzunehmen. Zusätzlich kann er über das Menü *Roles* die globalen Rechte für den Zugriff auf den Server verändern.

4.9 Dawn-Runtime und Editor-Erweiterung

Die *Dawn-Runtime* ist der elementare Kern der Client-Seite in Dawn. Sie kapselt alle Methoden, welche die Editoren zur Kommunikation mit dem Server benötigen. Daneben stellt sie Menüs und Dialoge zur Verfügung, um das System einfach und komfortabel verwalten zu können. Die Dawn-Runtime wird als eigenes Plugin realisiert, welches im Kontext von Eclipse läuft. Die Dawn-Erweiterung hingegen wird als Fragment [vgl. 2.8.3.4] implementiert. Dadurch werden alle zusätzlichen Erweiterungen im Kontext des Host-Plugins, also des GMF-Editors ausgeführt. Dies hat den Vorteil, dass am Host-Plugin keine Änderungen vorgenommen werden müssen, um Klassen erweitern zu können. Würde die Dawn-Erweiterung als Plugin implementiert, so müsste das GMF-Editor-Plugin alle zu erweiternden Klassen exportieren, also öffentlich zugängig machen, was Änderungen am GMF-Editor Plugin bedeuten würde.

Neben den Wizards erweitert die Editor-Erweiterung auch den bestehenden Editor, ohne diesen zu verändern. Dies ist notwendig, um beispielsweise das Save-Event für die Synchronisation des Modellbestandes nutzen zu können.

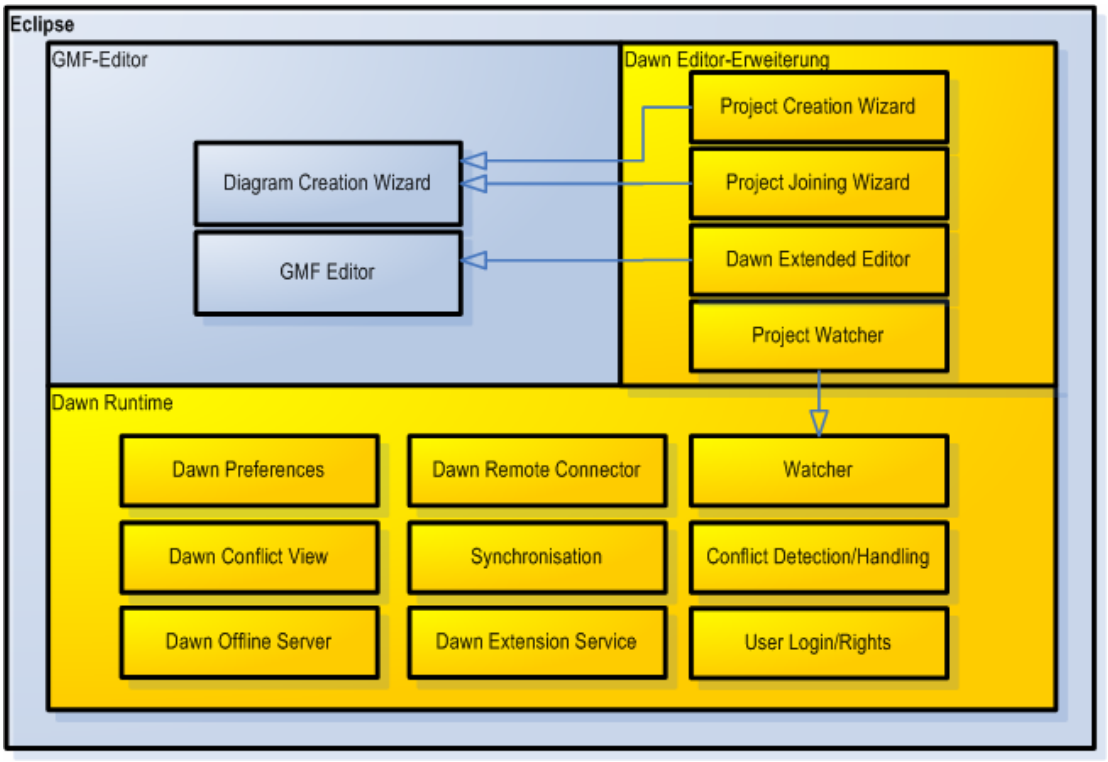

Abbildung 45 - Architektur des Clients

Aus Abbildung 45 wird ersichtlich, welche Teile des GMF-Editors durch die Editor-Erweiterung erweitert werden. Jeder GMF-Editor erhält seine für ihn spezifische Erweiterung. Alle diese Erweiterungen greifen auf die Dawn-Runtime zu, um die Funktionalitäten von Dawn auszuführen. Dabei bietet die Dawn-Runtime die

beschriebenen Mechanismen für den Kommunikationsaufbau, die Konfliktbehandlung, das Locking oder den Offline-Modus an. Zusätzlich stellt sie Konfigurationen zur Verfügung, mit denen globale Einstellung für alle Editoren, wie zum Beispiel den Übertragungsmodus oder die Verbindungseinstellungen zum Server eingestellt werden können.

4.9.1 Wizards

Die Editor-Erweiterung stellt zwei Wizards für Dawn zur Verfügung. Der erste Wizard kann genutzt werden, um Dawn-basierende Diagramme zu erstellen. Der zweite wird genutzt, um sich mit einem Dawn-Projekt zu verbinden. Dabei bauen beide Wizards auf dem vom GMF-Editor bereits zur Verfügung gestellten Wizard für das Anlegen des Diagramms auf und erweitern diesen um zusätzliche Informationen. Dieses Vorgehen ist sehr sinnvoll, da so auch bereits bestehende Änderungen am Editor mit übernommen werden. Hat beispielsweise ein Entwickler in den Creation-Wizard des Editors zusätzliche Funktionen eingebaut, so sind diese auch über die Dawn-Wizards nutzbar.

4.9.2 Preferences

Über die Preferences der Dawn-Runtime können verschiedene Parameter verändert werden, die das Gesamtverhalten des Systems beeinflussen. Dabei stehen verbindungsrelevante Parameter, als auch nutzerspezifische Einstellung zur Verfügung. Die nachfolgende Tabelle erklärt die einzelnen Parameter.

Name	**Bedeutung**
Server URL	URL des Servers
Server Port	Port des Webservers
Project Name	Name des Web-Projektes
Publish Mode	Die Übertragungsmethode • On Interval = Publish bei jedem Interval • On Save = Publish nur beim Speichern
Connection Mode	Die Verbindungsmethode, beziehungsweise das Übertragungsprotokoll. Zur Zeit SOAP oder RMI

Tabelle 4 - Preferences

4.9.3 Dawn Conflict View

Nicht alle Konflikte können direkt über das Diagramm gelöst beziehungsweise angezeigt werden. Lokale Lösch-Konflikte [vgl. 4.4.1.2] beziehen sich auf Objekte, die lokal gelöscht, aber von einem anderen Clients geändert wurden. Da sich diese Objekte

nicht mehr in dem Diagramm befinden, können sie auch dort nicht mehr angezeigt werden. Mit Hilfe der *Dawn Conflict View* wird es trotzdem ermöglicht auch diese Konflikte zu verwalten. Dazu wird eine spezielle *View* entworfen, die es ermöglicht mit Hilfe einer tabellarischen Ansicht alle Konflikte anzuzeigen. In Kapitel 5.4.2 wird beschrieben, wie diese View mit den Mitteln von Eclipse umgesetzt wurde.

4.10 Konzept des prototypischen Diagramms

Die Funktionalitäten sollen anhand eines stark vereinfachten Klassendiagramms verdeutlicht werden. Dabei geht es nicht darum, sich an dem Funktionsumfang des UML-Diagramms zu orientieren, sondern vielmehr dessen Elemente zur Veranschaulichung zu nutzen. Das bedeutet, dass das entwickelte Diagramm sich aufgrund des beschränkten Umfanges nur bedingt als Modellierungsdiagramm eignet. Die Analogie zum Klassendiagramm wurde gewählt, da die Kenntnis über grundlegende Elemente weitgehend vorhanden ist. Außerdem bietet es eine Vielzahl grafischer Elemente an, die sich gut dazu eignen komplexere Graphen übersichtlich darzustellen.

Das prototypische Diagramm wird über Klassen und Schnittstellen verfügen, um zu zeigen, dass unterschiedliche Knotentypen gehandhabt werden können. Hierbei sollen abweichend vom UML-Standard die Kanten der Schnittstellen rund dargestellt werden, um die Anzeige von runden Ecken im Browser zu demonstrieren. Die Klassen und Schnittstellen werden sich farblich unterscheiden. Beide Knotentypen können aber Parameter und Operationen definieren, welche einen Namen und Typ, und bei Letzterem eine Parameterliste besitzen.

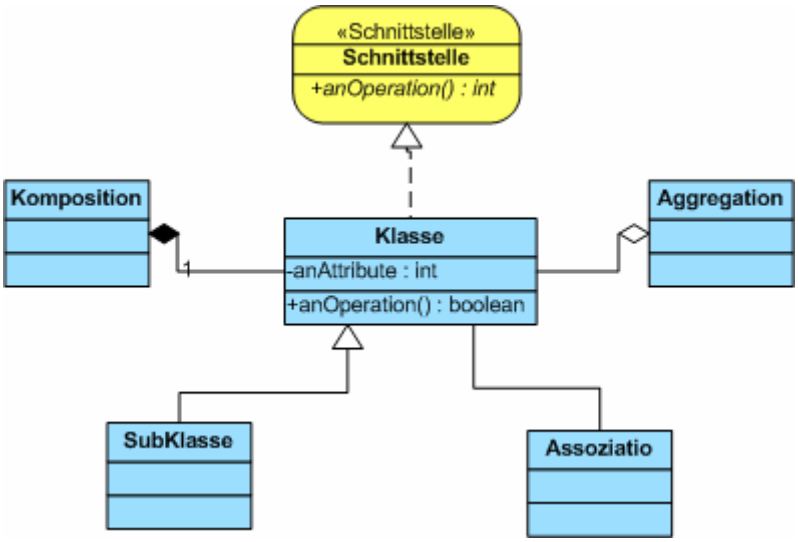

Abbildung 46 - prototypisches Klassendiagramm

Zusätzlich werden die fünf klassischen Konnektoren des UML-Diagramms – *Generalisierung*, *Realisierung*, *Assoziation*, *Aggregation*, *Komposition* – genutzt, um unterschiedliche Kantentypen darzustellen. Zur Vereinfachung werden diese nicht über Label für die Anzeige von Stereotypen, Multiplizitäten oder sonstigen verbindungsverfeinernden Informationen verfügen. Die Navigation wird ebenfalls nicht im Nachhinein veränderbar sein, sondern sich an der Erstellungsrichtung des Konnektors orientieren. Abbildung 46 zeigt einen Entwurf der möglichen grafischen Umsetzung.

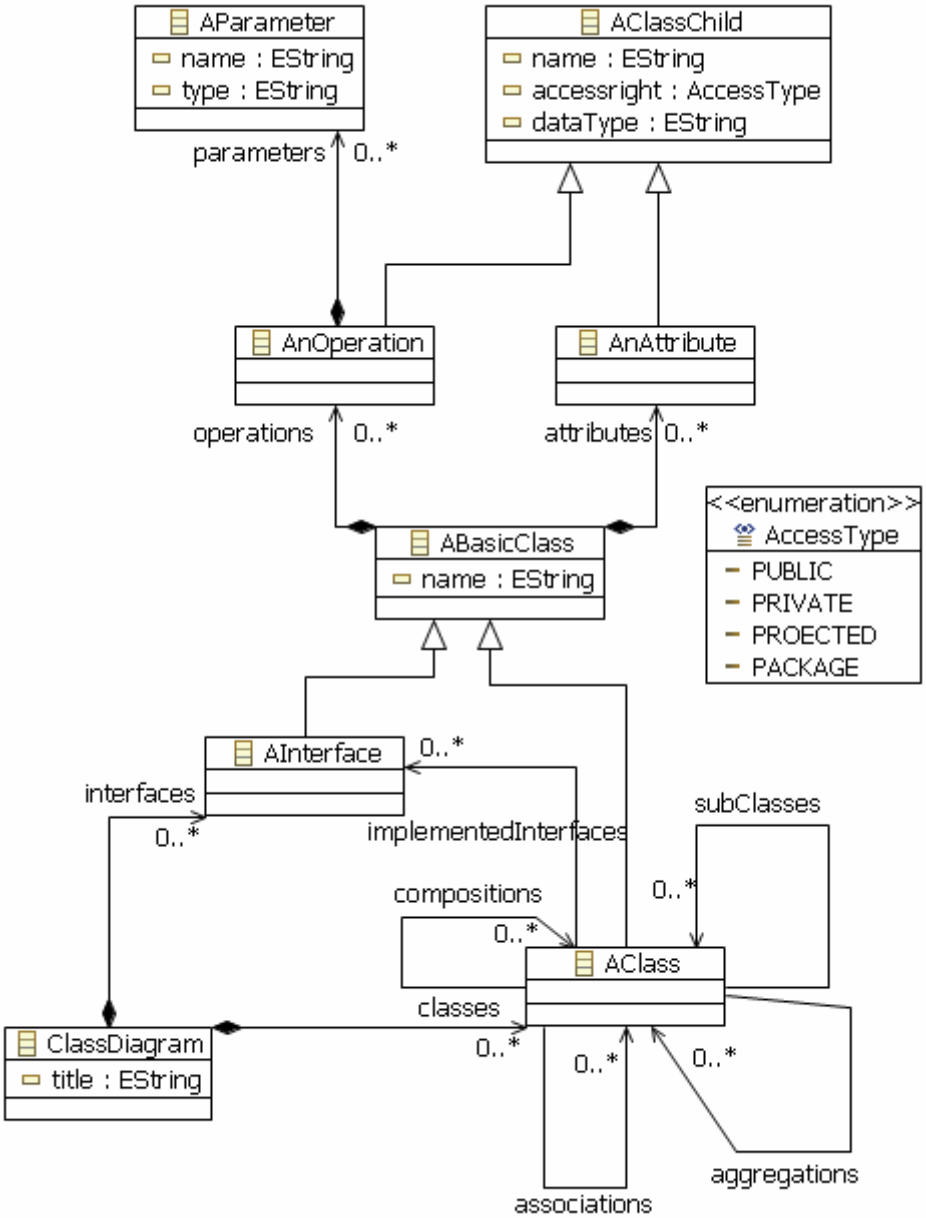

Abbildung 47 - Dawn Prototyp Ecore-Modell

Da jeder GMF-Editor auf einem Ecore-Modell basiert, musste ein Datenmodell für den Editor entworfen werden, das die Elemente wie Klasse, Interface, Assoziation etc. bestimmt. Da die Bezeichner des Modells in englischer Sprache verfasst sind, Elemente wie `interface` oder `class` aber reservierte Schlüsselwörter in Java sind, wurden alle Bezeichner mit dem Prefix „A“ oder „An“ versehen. Abbildung 47 zeigt Ecore-Modell für den Prototyp.

4.11 Code-Generierung

Die vorherigen Kapitel legten den konzeptuellen Grundstein für die Erstellung eines Prototyps für das gesamte Dawn-System. In der Regel ist dies einer der ersten Schritte im modellgetriebenen Softwareentwicklungsprozess. Auf dem Prototyp aufbauend können, so noch nicht geschehen, Modelle entwickelt werden, welche durch Generatoren transformiert werden können.

4.11.1 Dawn-GenModel

Grundlage jeder Model-to-Text Transformation ist ein Modell. Da nicht alle Informationen, die zur Generierung des Dawn-Codes benötigt werden, in den GMF-Modellen vorhanden sind, nutzt Dawn ein eigenständiges Modell für die Transformation. Dieses Modell enthält Informationen, die benötigt werden, um beispielsweise den Namen der Web-Applikation, den Serverport oder die Datenbankparameter definieren zu können. Dieses Modell wird *Dawn-GenModel* genannt und gibt dem Nutzer die Möglichkeit auf die Code-Generierung Einfluss zu nehmen. Das ihm zugrunde liegende Meta-Modell wird als Ecore-Model implementiert. Dies hat zum einen den Vorteil, dass es mit Hilfe des Ecore-Editors sehr leicht erstellbar ist. Zum anderen ist es damit auch konform zum EMF- und GMF-Entwicklungsprozess. Eine weitere Stärke des Einsatzes von EMF für die Spezifikation des Dawn-Metamodells ist die automatische Erzeugung eines baumbasierten Editors für die Bearbeitung eines entsprechenden Modells. Somit muss der Anwender das Modell nicht im XML-Format innerhalb eines Editors bearbeiten, sondern kann komfortabel ein angepasstes Nutzer-Interface verwenden. Abbildung 48 gibt einen Überblick über die in dem Meta-Modell spezifizierten Elemente.

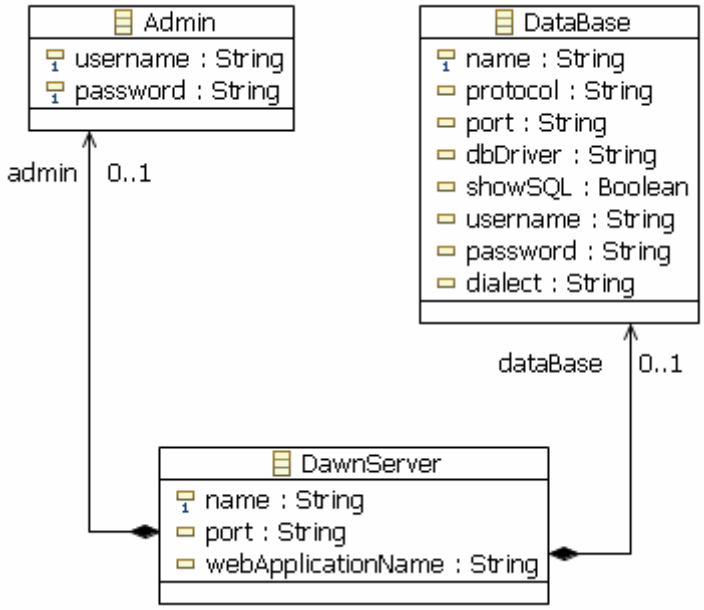

Abbildung 48 - Dawn-GenModel Ecore

4.11.2 Generierung der Komponenten

Dawn besteht aus Sicht der Generierung betrachtet aus drei Elementen – der Editor-Erweiterung, dem Server und den JavaScript-Klassen als Basis für den Web-Viewer. Alle diese Komponenten können unabhängig voneinander generiert werden.

Die Generierung des *Servers* erfolgt als hybrides Generat, welches zu einem Teil aus fertigem Code entsteht und zum anderen Teil durch modellgesteuerte Informationen ergänzt wird. Diese Vorgehensweise wird genutzt, da der Server-Code extrem umfangreich ist. Eine vollständige Generierung aller Klassen, Konfigurationsdateien, Java Server Pages und sonstiger Dateien wäre nicht praktikabel, besonders weil ein großer Teil dieser Informationen keine Änderungen durch die Transformation erfährt. Allerdings soll dem Nutzer der komplette Server-Code zur Verfügung gestellt werden, damit dieser gegebenenfalls gewünschte Änderungen ausführen kann. Änderungen durch die Transformation erfahren zum Beispiel die Hibernate Konfigurationsdatei, welche mit den datenbankspezifischen Informationen ergänzt wird, und Konfigurationsdateien für den Server. Um die sofortige Arbeit mit dem Server zu ermöglichen, wird innerhalb der Eclipse IDE ein *Sysdeo Tomcat-Projekt* erstellt und der Code in dieses integriert. Mit dem Tomcat-Plugin für Eclipse besteht die Möglichkeit einen Tomcat-Server direkt aus Eclipse heraus zu steuern, Projekte zu deployen oder diese als Web-Archive (*.war) zu exportieren. Da sich die Arbeit mit diesem Plugin bewährt hat, wird dem Nutzer diese Integration bereits bei der Generierung zur

Verfügung gestellt, damit der Server direkt benutzt werden kann. Natürlich verhindert die Nutzung des Tomcat-Plugins nicht, dass der Server auch in anderen Application-Servern genutzt werden kann.

In den Server werden die JavaScript-View-Repräsentationen der einzelnen GMF-Views generiert. Diese Generierung kann unabhängig von der Erstellung des Servers laufen. Wird der Server über das oben erwähnte Tomcat-Plugin gestartet, so kann sogar zur Laufzeit der Code für den Web-Viewer ausgetauscht werden. Für die Generierung der JavaScript-Klassen greift Dawn auf das GMF-GenModel [vgl. 2.8.7.1] zu. Da das GenModel das GMF-Graph Model referenziert, welches alle Informationen zu den View-Deskriptoren enthält, können sämtliche Informationen bezogen werden, welche benötigt werden, um die JavaScript-Views zu erzeugen. Zusätzlich liefert das GenModel Informationen, die es ermöglichen eine Zuordnung zu den Views zu gestalten. Da das GMF-Graph Modell lediglich das Aussehen der Views definiert, lässt sich in ihm aber noch keine Zuordnung zu dem Datenmodell herstellen. Eine View kann beispielsweise mit mehreren EMF-Objekten verknüpft sein. Erst nachdem über das Mapping-Modell eine Zuordnung zwischen Modell, View und Tooling getroffen wurde, kann eine View eindeutig bestimmt werden. Dies geschieht über die *ElementTypes*, welche genau eine View beschreiben. Die Generierung der *JavaScript-Figures* erzeugt also nicht nur allein die Figuren, sondern auch die Konfigurationsdatei, welche vom *Diagramm-Servlet* genutzt wird, um die Knoten und Kanten richtig darstellen zu können [vgl. 4.8.1].

Die Netzwerkerweiterung für den GMF-Editor wird ebenfalls generiert. Diese wird als Fragment-Erweiterung zu dem bestehenden Editor-Plugin erzeugt. Als Basis für die Generierung wird ebenfalls das GMF-GenModel genutzt, da es alle notwendigen Informationen und Referenzen beinhaltet. Um die Menge der zu erstellenden Dateien möglichst gering zu halten, wurde ein Großteil der Funktionalität in die Dawn-Runtime ausgelagert. Dies ermöglicht es wenige einfache Klassen zu erstellen. Somit bleibt der Quellcode für den Anwender übersichtlich und leicht verständlich, da komplexere Vorgänge in der Runtime versteckt werden. Neben der Erstellung aller Java-Klassen werden auch die für das Fragment notwendigen Dateien (Manifest.MF, fragment.xml, build.properties) erzeugt. Das Fragment wird automatisch mit dem Host-Plugin verbunden, sodass es sofort ausgeliefert werden kann. Der Nutzer muss keine weiteren Änderungen vornehmen und kann die Erweiterung sofort nutzen.

4.12 Zusammenfassung

In diesem Kapitel wurde das Konzept von Dawn entwickelt. Eingangs wurde erläutert, dass Dawn aus vier wesentlichen Komponenten besteht. Clientseitig erweitert das System den bestehenden GMF-Editor durch die *Editor-Extension*. Diese greift auf die *Dawn-Runtime* zu, welche Basisoperationen für die Kommunikation, das Synchronisieren, das Locking und die Konfliktbehandlung bereitstellt. Der *Dawn-Server* dient zur Verwaltung der Daten und ihrer Persistierung. Sowohl der Dawn-Server als auch die Client-Erweiterung können über das *Dawn-Codegen* modellgetrieben generiert werden [vgl. 4.1].

Dawn nutzt ein flexibles Kommunikations-Adapter-Konzept, um die Übertragungstechnologie für die Daten austauschen zu können. Dieser Wechsel kann sogar zur Laufzeit erfolgen [vgl. 4.2].

Zur Datensynchronisation setzt Dawn eine eigene Compare- und Merge-Engine ein, die Unterschiede in den Modellbeständen lokalisieren und die Modelle zusammenführen kann [vgl. 4.3]. In Dawn können drei verschiedene Konflikte auftreten – Lokal- und Remote-Änderungskonflikte, lokale Löschkonflikte und entfernte Löschkonflikte. Diese werden mit Hilfe der Speicherung des letzen Serverzustandes identifiziert werden.

Um die Konflikte zu beseitigen, nutzt Dawn ein offenes System, was es dem Nutzer erlaubt den Konflikt sowohl durch Annehmen, als auch durch Verweigerung der Änderungen des anderen Nutzers zu lösen. Damit Konflikte ganz ausgeschlossen werden können, kann das System bestimmte Bereiche eines Diagramms blockieren [vgl. 4.4].

Dawn unterstützt das Bearbeiten des Diagramms, auch wenn keine Verbindung zum Server besteht. Es nutzt hierzu eine Offline-Server Instanz, die sich bei bestehender Verbindung mit dem Dawn-Server synchronisiert und im Bedarfsfall den Server simulieren kann [vgl. 4.5].

Über ein generisches Rechte- und Rollenkonzept ist das System so flexibel, dass spätere Anforderungen hinsichtlich der Vergabe von Rechten einfach implementiert werden können. Im Zentrum dieses Konzeptes steht die Komponente *UserManager*, an die alle nutzerspezifischen Anfragen weitergeleitet werden. Durch spezielle Implementierungen dieser Schnittstelle können die Fähigkeiten individuell gestaltet werden, sodass es auch möglich ist das Nutzermanagement abzuschalten ohne Einfluss auf andere Komponenten nehmen zu müssen [vgl. 4.6].

Der Dawn-Server nutzt zur Persistierung der Daten *Hibernate* als objektrelationalen Mapper. Zusätzlich stellt er zwei Web-Komponenten zur Verfügung – eine Konfigurationsoberfläche und einen Web-Viewer. Eine webbasierte Konfigurationsschnittstelle erlaubt es Manipulationen an den einzelnen Projekten vorzunehmen, Nutzern Rechte zuzuweisen beziehungsweise zu entziehen oder die Rechte der Rollen zu verändern [vgl. 4.8.2].

Um die GMF-Diagramme auch in einem Browser anzeigen zu können, wurde ein Konzept entworfen, dass die Figuren mit Hilfe von HTML und JavaScript angezeigt werden können. Dabei übernimmt das **`DiagramServlet`** die Rolle des Controllers und kann mit Hilfe generierter Konfigurationsinformationen eine Zuordnung zwischen den Views und dem persistierten Datenmodell herstellen [vgl. 4.8.1].

Der Server, sowie die JavaScript-Views und die Editor-Erweitung können mittels eines modellgetriebenen Prozesses erzeugt werden. Dabei basiert die Generierung auf dem Dawn-GenModel. Dieses greift bei der Generierung auf die bestehenden GMF-Modelle zu und kann so deren Einstellung in die Erweiterungen einfließen lassen[vgl. 4.11].

5 Implementierung

Die folgenden Kapitel beschreiben ausgewählte Teilschritte während der Implementierung des Konzepts. Hierbei werden insbesondere die Aspekte beschrieben, die sich während der Umsetzung des Konzeptes als besonders schwierig oder interessant herausgestellt haben. Nicht alle Implementierungen erfahren eine detaillierte Erklärung, sondern nur jene, die von besonderem Interesse sind. Eingangs werden dabei die wesentlichen Aspekte zum Starten und Initialisieren der Kommunikation und Synchronisation erläutert. Danach werden implementierungsspezifische Besonderheiten der Client-Seite, besonders die Eigenheiten der Editorerweiterung und der Dawn-Runtime beschrieben. Im Anschluss daran wird auf die Aspekte der Server-Seite eingegangen, welche die Datenhaltung, die Web-Verwaltung und den Web-Viewer umfassen. Abschließend wird die Generierung des Quelltextes mit Hilfe des o*penArchitectureWare* Frameworks behandelt.

5.1 Kommunikation und Synchronisation

In den Kapiteln 4.2 und 4.3 wurden die Konzepte für die Kommunikation und Synchronisation in Dawn erläutert. Nachfolgend werden einige interessante Aspekte der Implementierung beschrieben.

5.1.1 Initialisierung der Kommunikation

Wie in Kapitel 4.2.3 beschrieben, wird beim Start des Editors der *Watcher*-Thread erzeugt, welcher die Daten mit dem Server synchronisiert. Zeitgleich wird, wenn der Nutzer noch nicht lokal authentifiziert ist, ein Login-Dialog angezeigt, um den Nutzer am System zu registrieren und seine Rechte mit dem Server abzugleichen.

Beim Start von Eclipse werden alle Plugins gestartet und alle bereits offenen Editoren initialisiert. Hat also ein Benutzer Eclipse geschlossen und alle Diagramme, an denen er in der aktuellen Sitzung gearbeitet hat, offen gelassen (was dem Normalfall entspricht), so werden diese sofort beim nächsten Start erzeugt. Dies hat zur Folge, dass während die Eclipse-Instanz startet der Vorgang unterbrochen wird und über dem Startbild ein Login-Dialog angezeigt wird. Folgerichtig muss das Starten der Wächter-Threads solange verzögert werden, bis die Eclipse GUI vollständig geladen wurde. Zwar bietet Eclipse verschiedene Methoden an, welche den Zustand der Workbench anzeigen (z.B.

`PlatformUI.isWorkbenchRunnig()`), diese liefern allerdings schon während des Start-Vorganges den Wert *true*, da die Workbench weit vor dem finalen Anzeigen der GUI initialisiert wird. Das verzögerte Starten der Wächter-Instanz um einen festen Zeitwert, kann ebenfalls nicht als akzeptable Lösung genutzt werden. Einerseits kann die Startdauer von Eclipse abhängig von den installierten Plugins und der aktuellen Auslastung des Systems unterschiedlich lange dauern. Andererseits werden Editoren auch innerhalb einer bereits gestarteten Eclipse-Instanz geöffnet. Im letzten Fall muss die Verbindung zum Server sofort aufgenommen werden und darf nicht erst zeitverzögert erfolgen.

Mit Hilfe eines kleinen Tricks kann das Problem allerdings dennoch behoben werden. Der Unterschied zwischen dem gestarteten und dem startenden Eclipse ist, dass verschiedene GUI-Komponenten erst nach dem Start initialisiert sind. So ist zum Beispiel die *Menu-Bar* nur dann initialisiert, wenn die Eclipse-Workbench sichtbar ist. Mit Hilfe dieser Information kann also das Starten von Komponenten solange verzögert werden, bis die Menu-Bar vom System erzeugt wurde. Listing 4 zeigt die Quelltext-Passage, mit der das zeitverzögerte Starten der Komponenten realisiert wird.

```
1   while (Display.getCurrent().getActiveShell().getMenuBar() == null)
2   {
3     System.out.println("Eclipse has not finished starting...wait");
4     try
5     {
6        Thread.sleep(200);
7     }
8     catch (InterruptedException e)
9     {
10       e.printStackTrace();
11     }
12  }
```

Listing 4 - Zeitverzögertes Starten der Dawn Komponenten

5.1.2 RemoteConnections und RemoteConnector

In Dawn wurden zwei Kommunikationsprotokolle implementiert, RMI und SOAP, um zu demonstrieren, dass die Nutzung unterschiedlicher Adapter möglich ist. Die SOAP-Schnittstelle wurde mit Hilfe des Apache-Frameworks *Axis* implementiert. Axis bietet mit dem Kommandozeilenprogramm *WSDL2Java* eine Möglichkeit automatisch aus einer Web-Service Definition (*.wsdl) den Quellcode des entsprechenden Stubs zu erzeugen. Problematisch bei dieser Generierung war allerdings, dass Axis als einzigen Listentyp Arrays kennt. Das Generat besaß folglich nicht die in der **`DawnRemoteConnection`** definierten Listen (**`List<>`**), sondern ausschließlich Arrays

vom Typ Object (**`Object[]`**). Um die eigentliche Funktionalität aber hinter dem RemoteConnection-Interface verbergen zu können, wurde eine WrapperKlasse geschrieben, welche die Methoden des generierten SOAP-Stubs kapseln und gegebenenfalls die Object-Arrays in die vom Interface erwarteten Listen konvertiert.

Der RemoteConnector kann zur Laufzeit das Kommunikationsprotokoll wechseln. Dazu wird jedes Mal, wenn eine Komponente eine RemoteConnection anfordert, der über die Eigenschaften gesetzte Wert (**`currentType`**) geprüft. Anhand dessen wird eine Verbindung über das angeforderte Protokoll aufgebaut. Listing 5 zeigt den Ausschnitt aus dem **`DawnRemoteConnector`**.

```
1   if (currentType == RMI)
2   {
3   mfInstance = (DawnRemoteConnectionRMI) Naming.lookup("rmi://" +
4   servername + "/" + serverContextname);
5   }
6   else if (currentType == SOAP)
7   {
8    String serverAdress = "http://" + serverName + ":" + serverport +
9   "/" + serverContextname + "/services/DawnCommunicationService";
10   mfInstance = new DawnRemoteConnectionSOAP(serverAdress);
11  }
```

Listing 5 - Auswahl des Kommunikationsprotokolls

Die einzelnen serverseitigen Adapter greifen alle direkt auf die Singleton[40]-Instanz des Servers zu. Um zu verhindern, dass gleichzeitiger Zugriff auf die Daten zu Inkonsistenzen führt, sind alle Methoden des Servers gegen parallele, mehrfache Nutzung geschützt. Dies wurde umgesetzt, indem die kritischen Bereiche der einzelnen Methoden mit *synchronized*-Blöcken versehen wurden. Diese Bereiche werden von der Java Virtual Machine geschützt ausgeführt.

5.2 Umsetzung des prototypischen GMF-Editors

Der an ein Klassendiagramm angelehnte GMF-Editor wurde basierend auf dem spezifizierten EMF-Modell [vgl. 4.10] und der grafischen Vorlage [vgl. Abbildung 46] implementiert. Dabei wurden dem GMF-Entwicklungsprozess entsprechend erst die grafischen Repräsentationen (Figures) und die Tooling-Palette definiert und diese daraufhin mit dem Datenmodell verknüpft. Über die modellgetriebenen Mechanismen von EMF und GMF wurden daraufhin die Datenhaltungsklassen, das Edit-Plugin und

[40] Singleton – Entwurfsmuster, bei dem sichergestellt wird, dass von einer Klasse nur eine Instanz erzeugt werden kann [vgl. Schmidt 2007, S.102]

das Diagramm-Plugin generiert [vgl. 2.8.7.1]. Der prototypische Editor diente als Grundlage für die gesamte Entwicklung. Die grundlegenden Funktionen des prototypischen Diagramms in Verbindung mit Dawn können in [Flügge 2009f] betrachtet werden.

5.3 Editor-Erweiterung

Die Clientseite von Dawn teilt sich bezüglich des Laufzeitverhaltens in zwei Teile – der Editor-Erweiterung und der Dawn-Runtime [vgl. 4.1]. Die Editor-Erweiterung stellt dabei alle Informationen bereit, die nicht generisch von der Runtime abgedeckt werden können. Diese sind neben speziellen Klassen auch der erweiterte Editor und die Wizards zur Erstellung der Dawn-Diagramme. Die folgenden Kapitel beschreiben wie diese Komponenten umgesetzt wurden und wie sie die Dawn-Runtime unterstützen.

5.3.1 ElementTypeHelper und ResourceSet

Die Dawn-Runtime übernimmt einen großen Teil aller Aktionen innerhalb der Clientseite von Dawn. Allerdings benötigt sie einige wenige Informationen von den einzelnen Editoren, um editorspezifische Operationen ausführen zu können. Dazu gehört ein *ResourceSet*, welches die Verwaltung der Ressourcen eines Editors übernimmt [vgl. 2.8.4.4]. Jedes ResourceSet eines GMF-Editors muss sowohl das Meta-Modell von GMF, also auch sein eigenes registrieren.

```
1   public class ClassdiagramResourceSet {
2         public static ResourceSet getResourceSet() {
3         Resource.Factory.Registry reg =
4   Resource.Factory.Registry.INSTANCE;
5               Map m = reg.getExtensionToFactoryMap();
6               m.put("*", new XMIResourceFactoryImpl());
7
8               ResourceSet rsSet = new ResourceSetImpl();
9               rsSet.setResourceFactoryRegistry(reg);
10        rsSet.getPackageRegistry()
11          .put("http://www.eclipse.org/gmf/runtime/1.0.1/notation",
12                          NotationPackage.eINSTANCE);
13        rsSet.getPackageRegistry()
14          .put("http://www.Classdiagrameditor.de",
15                          ClassdiagramPackage.eINSTANCE);
16              return rsSet;
17        }
18  }
```

Listing 6 - editorspezifisches ResourceSet

In Listing 6 wird in den Zeilen 12-17 diese Registrierung vorgenommen. Jeder Editor benötigt dieses ResourceSet, um es der Dawn-Runtime zur Laufzeit übergeben zu können.

Neben dem ResourceSet benötigt jeder Editor eine Klasse, welche es ermöglicht eine Zuordnung zwischen einem View-Element und seinem ElementType zu treffen. Leider bietet das GMF-Framework keine Möglichkeit direkt an diese Information zu kommen, weshalb eine eigene Implementierung erstellt werden musste. Zwar werden die ElementTypes in der ElementTypeRegistry eines Editors verwaltet, können dort aber nur über einen statischen String abgefragt werden. Das generische Mapping zu einer View kann nicht erfolgen. Die Komponente, der **`ElementTypeHelper`**, erstellt die Zuordnung, indem er die *VisualID* einer View mit der eines passenden EditParts vergleicht, um den entsprechenden ElementType zu finden. Listing 7 zeigt einen Ausschnitt des **`ElementTypeHelpers`** für den prototypischen Klassendiagramm-Editor.

```
1   public IElementType getElementType(View view)
2   {
3     int visualId = ClassdiagramVisualIDRegistry.getVisualID(view);
4     switch (visualId)
5     {
6         case ClassDiagramEditPart.VISUAL_ID :
7            return ClassdiagramElementTypes.ClassDiagram_79;
8         case AnAttributeEditPart.VISUAL_ID :
9           return ClassdiagramElementTypes.AnAttribute_2001;
10  .
11  .
12  .
13    }
14    return null;
15  }
```

Listing 7 - getElementType(View)

Damit die einzelnen Implementierungen einfach von den Anwendern erweitert werden können, arbeitet die Dawn-Runtime nicht direkt mit diesen Implementierungen, sondern mit Fabriken, welche die Erzeugung des ResourceSets und des ElementTypeHelpers übernehmen. Welchen Mechanismus die Dawn-Runtime nutzt, um zur Laufzeit die dynamischen Informationen der einzelnen Editoren zu erhalten, wird in Kapitel 5.4.4 beschrieben. Kapitel 5.9 beschäftigt sich mit den generativen Aspekten dieser beiden Komponenten.

5.3.2 DawnExtendedEditor

Kern eines jeden GMF-Editors ist eine Klasse, welche die Basis-Klasse *DocumentDiagramEditor* erweitert. Diese Editor-Klasse verwaltet nicht nur Lade- und

Speichervorgänge, sondern initialisiert auch beim Start des Editors das grafische Modell mit den Daten aus den Ressourcen. Um ebenfalls die Dawn-spezifischen Parameter beim Start des Editors initialisieren zu können, muss die Editor-Klasse des GMF-Projektes erweitert werden. Zu den spezifischen Parametern gehören unter anderem das Prüfen der Server-Verbindung, ggf. das Laden der Offline-Informationen, das Starten der Watcher-Instanz oder das Abfragen des Logins. Damit der Editor um diese Funktionalitäten erweitert werden kann, besitzt jedes Kommunikationsfragment die Klasse **`DawnExtendedEditor`**, welche die Editoren-Klasse des jeweiligen GMF-Diagramm-Plugins beerbt.

Die erste Erweiterung, die in den DawnExtendedEditor implementiert wurde, war das Überschreiben der Speicher-Routine. Neben dem Persistieren der Daten im File-System prüft der Editor zusätzlich bei jedem Speichervorgang ob eine Netzwerkverbindung besteht und ob Konflikte aufgetreten sind. Somit kann direkt das Publizieren der Daten verhindert werden. Ist kein Konflikt vorhanden und eine Verbindung zum Server möglich, wird innerhalb der Save-Methoden das Publizieren der Daten veranlasst.

Neben der Save-Methode wird der Editor um seine grafischen Methoden erweitert. Jeder GMF-Editor ist für die Erstellung seiner Tooling-Palette, also der Werkzeugleiste des jeweiligen Editors, verantwortlich. In GMF wird der entsprechende Quellcode aus den Informationen des Tooling-Modells generiert. Dawn erweitert diesen Code um einen zusätzlichen Menüpunkt, der zur Anzeige von Dawn-spezifischen Informationen genutzt wird.

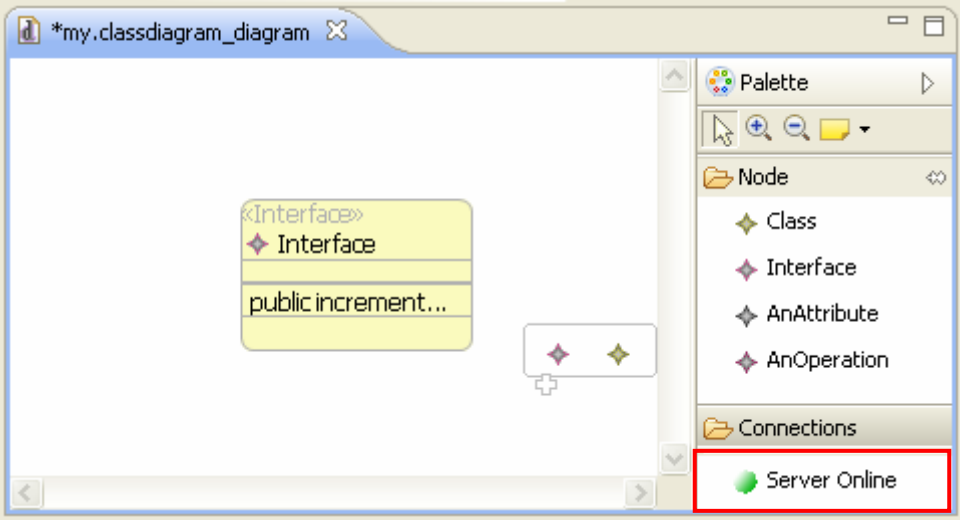

Abbildung 49 - Dawn Information

Um allerdings bei Erweiterung bzw. Regenerierung des GMF-Editors oder bei Erweiterungen seitens des GMF-Editor-Entwicklers keine Konflikte zu erzeugen, wird dieser Menüpunkt immer an die letzte Stelle in der Menüleiste gesetzt [vgl. Abbildung

49]. Bei der Erstellung der GUI greift der **DawnExtendedEditor** dabei auf die RootPalette zu und fügt die entsprechenden Einträge hinzu. Hierzu überschreibt er die Methode zur Erstellung der RootPalette (**PaletteRoot createPaletteRoot (PaletteRoot existingPaletteRoot)**) des eigentlichen Editors.

Um die Anzeige der Serververbindung ändern zu können, stellt der DawnExtendedEditor eine öffentliche Methode bereit (**void showServerOnlineMode (boolean)**). Diese Methode nutzt die Watcher-Implementierung für den Fall, dass die Verbindung zum Server nicht hergestellt werden kann, um den Editor zu veranlassen das Symbol in dem GUI zu verändern.

Alle externen Komponenten, welche Änderungen am erweiterten Editor vornehmen wollen, bekommen in der Regel vom System aber nur eine **DocumentDiagramEditor** Schnittstelle geliefert, hinter welcher der eigentliche Editor versteckt wird. Diese verfügt natürlich nicht über die Methoden, die der erweiterte Editor bereitstellet, um die Änderungen im Dawn-System vornehmen zu können. Damit die Dawn-Runtime aber mit diesen Methoden arbeiten kann, müssen alle erweiterten Editoren das Interface **DawnDiagramEditorInterface** implementieren. Über dieses Interface werden die Methoden zur Verfügung gestellt, die zur Verwaltung der Remote-Features notwendig sind. Somit ist es der Dawn-Runtime möglich den DocumentDiagramEditor auf die richtige Schnittstelle zu casten und die Methoden, wie das Setzen des Server-Status oder den Zugriff auf die Watcher-Instanz, zu realisieren [vgl. Listing 8].

```
1   public interface DawnDiagramEditorInterface
2   {
3         public Watcher getWatcher();
4         public  void setOwnPaletteVisible(boolean visible);
5         public String getContributorID();
6         void showServerOnlineMode(boolean online);
7   }
```

Listing 8 - DawnDiagramEditorInterface

Sämtliche Konzepte für die Veränderung des eigentlichen Editors basieren ausschließlich auf dessen Erweiterung. Somit ist es ohne Probleme möglich den GMF-Editor losgelöst von Dawn zu betreiben.

5.3.3 Wizards

GMF-Diagramme werden mit Hilfe von Wizards erzeugt. In Kapitel 4.9.1 wurde bereits erläutert, dass Dawn-Diagramme ebenfalls durch Wizards erzeugt werden. Hierbei werden diese benötigt, um ein Projekt anzulegen und um einem Projekt beizutreten.

Optisch unterscheidet sich der Dialog zum Anlegen eines Dawn-Projektes nicht von dem zum Erstellen des GMF-Diagramms [vgl. Abbildung 50, links]. Der einzigste Unterschied, den der Nutzer bemerkt, ist der Login-Dialog, welcher erscheint, wenn ein Projekt auf dem Server angelegt werden soll. Zusätzlich führt der Server eine Überprüfung durch, ob ein Projekt unter diesem Namen schon vorhanden ist. In diesem Fall wird eine Fehlermeldung generiert und der Nutzer muss einen anderen Namen wählen.

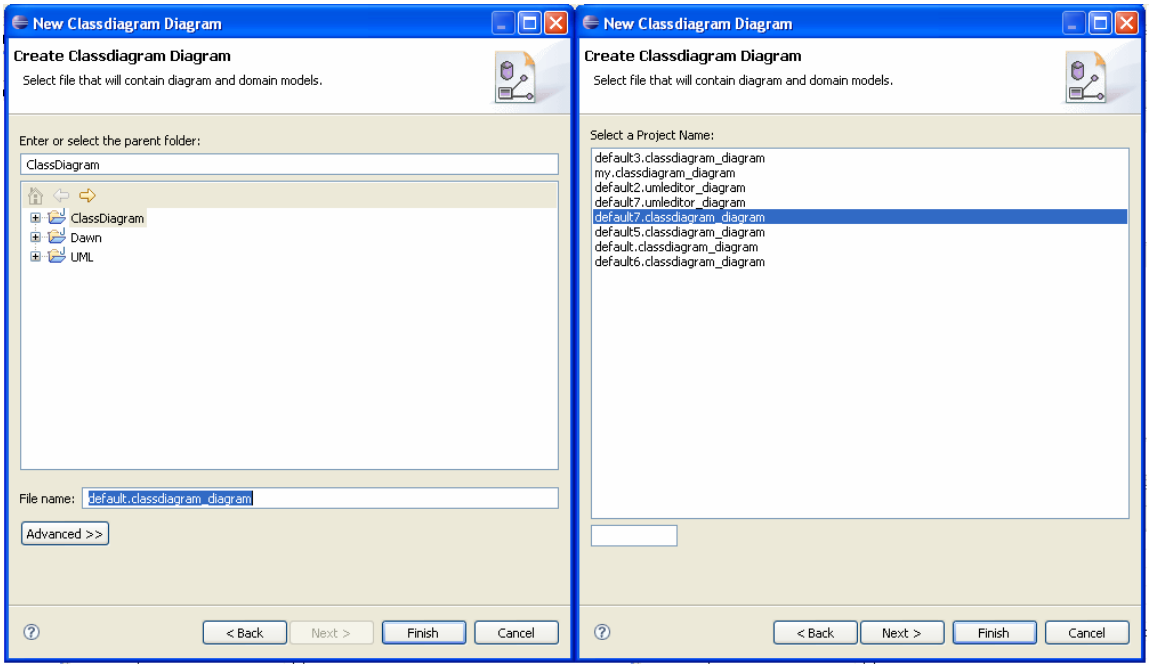

Abbildung 50 - Wizards zum Erstellen und Beitreten von Projekten

Der Wizard zum Beitreten zu bestehenden Projekten ist ebenfalls ähnlich aufgebaut [vgl. Abbildung 50, rechts], erlaubt dem Nutzer aber nicht einen eigenen Namen für das Projekt anzugeben. Stattdessen wird ihm eine Liste aller Projekte angezeigt, zu denen er Zugriff hat. Dies bedingt, dass er von dem Verwalter des Projekts zur Liste der Projektnutzer hinzugefügt wurde. Durch diesen Mechanismus können keine unautorisierten Nutzer Zugang zu einem Projekt erhalten.

5.4 Dawn-Runtime

Die Dawn-Runtime ist die zentrale Komponente, um die Funktionalitäten von Dawn auf der Clientseite umzusetzen. In den folgenden Kapiteln wird auf einige interessante Aspekte ihrer Implementierung eingegangen.

5.4.1 Konfliktbehandlung und -behebung

Damit Konflikte gelöst werden können, müssen sie als erstes dem Nutzer angezeigt werden. Dawn umrandet hierzu alle in Konflikt stehenden Objekte mit einem farbigen Rahmen. Um die Unterschiede der Konflikte für den Nutzer sofort erkenntlich zu machen, werden lokale und globale Änderungskonflikte *blau* und entfernte Löschkonflikte *rot* markiert. Technisch wird diese Markierung über ein Ereignis realisiert, welches der **`ResourceSynchronizer`** an den **`DawnChangeHelper`** schickt, sobald ein Konflikt auftritt. Dieser erstellt dann mit Hilfe eines passenden Kommandos die Umrandung für das Element.

Zum Auflösen des Konfliktes können geänderte oder entfernt-gelöschte Objekte direkt im Diagramm über einen Rechtsklick bearbeitet werden. Damit dieser Menüpunkt auf den Elementen eines Diagramms realisiert werden kann, muss das Fragment über zusätzliche Erweiterungen (*Extensions*) verfügen. Hierfür benötigt man eine neue Extension für den Extension Point **`org.eclipse.ui.popupMenus`**. Dieser kann genutzt werden, um Popup-Menüs auf bestimmten Objekten zu erstellen und zu gestalten. Ihm wird eine **`objectContribution`** hinzugefügt, welche die Verknüpfung des Menüs mit einem bestimmten Objekttyp angibt. In diesem Fall soll das Popupmenü nur angezeigt werden, wenn das markierte Objekt auch ein Element des Diagramms, also ein EditPart ist.

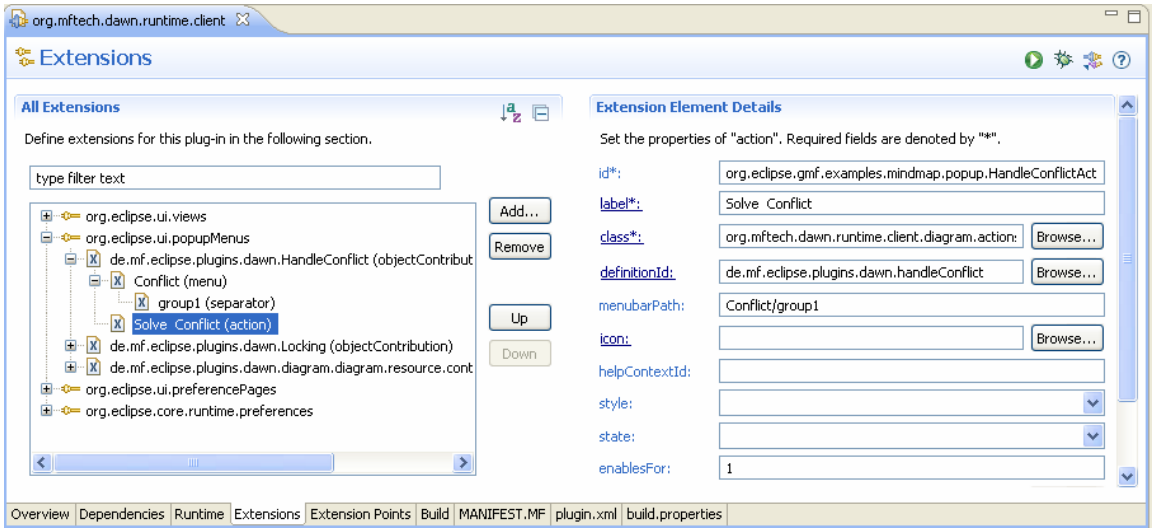

Abbildung 51 - Anlegen der Action für die Konfliktbehandlung

Zur Realisierung dieser Anforderung, wird dem Parameter **`objectClass`** der **`objectContribution`** der Wert **`org.eclipse.gef.EditPart`** hinzugefügt. Um über das Menü eine Aktion auslösen zu können, muss eine **`Action`** an die Erweiterung gebunden werden [vgl. Abbildung 51].

Die Action delegiert die Aufgabe an eine Komponente zur Lösung von Konflikten weiter – den **`SolveConflictHelper`**. Dieser kann anhand der selektierten View erkennen, welcher Konflikt gerade bei dem gewählten Objekt vorliegt. Dazu wurde der **`ResourceSynchronizer`** um die Methode **`getConflictType(String xmiID)`** erweitert, welche die Art des Konfliktes der View liefert. Daraufhin kann der **`SolveConflictHelper`** entsprechende Menüs für die Behandlung des Konfliktes anzeigen, um seine Beseitigung einzuleiten. Für den Nutzer ist dieser Vorgang allerdings transparent. Er erhält lediglich ein einziges Menü für die Konfliktbehebung [vgl. Abbildung 52]. Dawn kümmert sich automatisch darum die richtigen Konfliktbehandlungsroutinen zur Verfügung zu stellen.

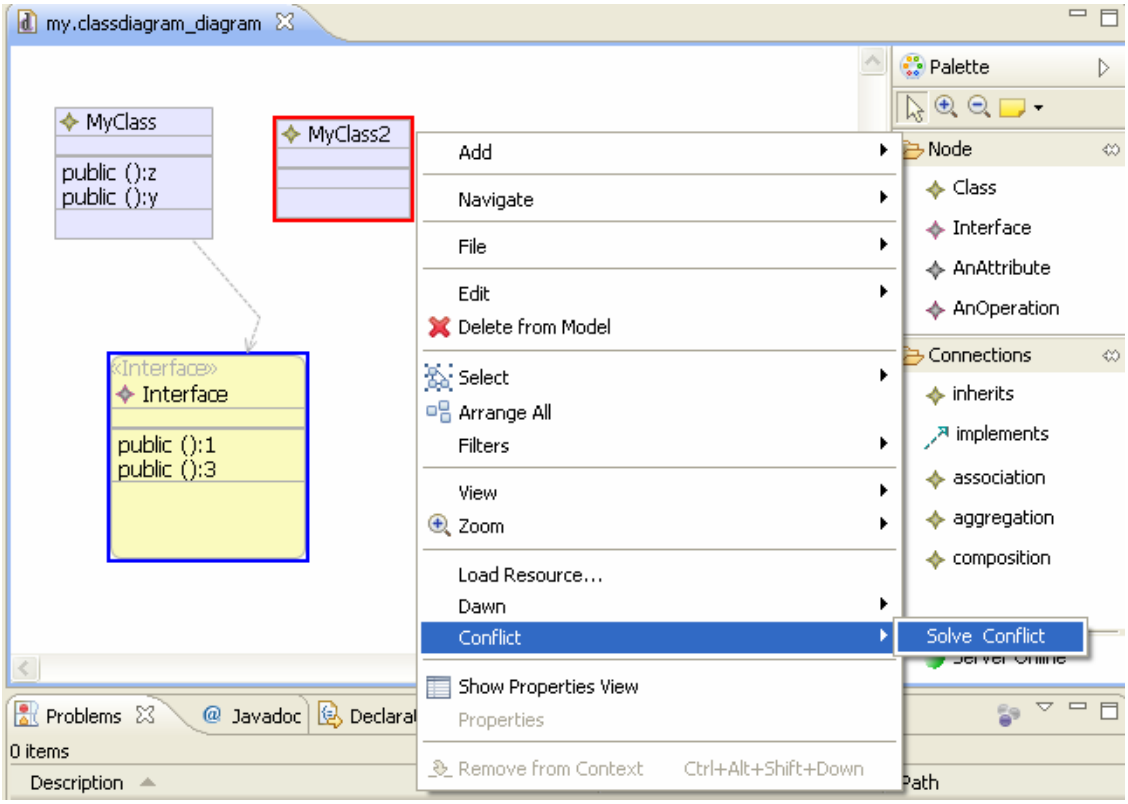

Abbildung 52 - Editor mit Konflikten

5.4.2 Dawn Conflict View

Nicht alle Konflikte können allerdings im grafischen Editor angezeigt werden. Objekte, welche lokal gelöscht wurden, müssen anderweitig als Konflikt vermerkt und gelöst

werden können. Hierzu bietet es sich an die von Eclipse bereitgestellten Views zu verwenden.

Zur Realisierung der *Dawn Conflict View*, muss eine View-Klasse implementiert werden, welche die abstrakte Basis-Klasse **`ViewPart`** erweitert. Über den Extension Point **`org.eclispe.ui.views`** kann daraufhin diese Implementierung innerhalb von Eclipse registriert werden. Dies äußert sich dadurch, dass die View über dasselbe Menü erreichbar ist, wie alle anderen Views innerhalb von Eclipse (Window → Show View → Others…)

Da die Conflict-View eine baumbasierte Repräsentation zur Darstellung der Konflikte benutzt, wird die JFace-Klasse **`TreeViewer`** als Grundlage für die Darstellung genutzt. Wie bei allen JFace-Elementen, liefern ContentProvider die Daten für die Anzeige. Dadurch wird eine strikte Trennung zwischen View und Modell erreicht. Außerdem bereiten die ContentProvider die Daten für ihre entsprechende View auf. So besitzen baumbasierte Ansichten andere ContentProvider-Implementierungen, als beispielsweise tabellarische Ansichten. Wie bei JFace-Viewern üblich wurde der ContentProvider als innere Klasse realisiert. Diese Implementierung kann über den aktuellen Editor auf den ResourceSynchronizer zugreifen und dort die aktuell in Konflikt stehenden Objekte abfragen. Nachdem der ContentProvider diese Informationen erhalten hat, kann er sie in das für die Conflict-View benötigte Format umwandeln und über den Viewer zur Anzeige bringen.

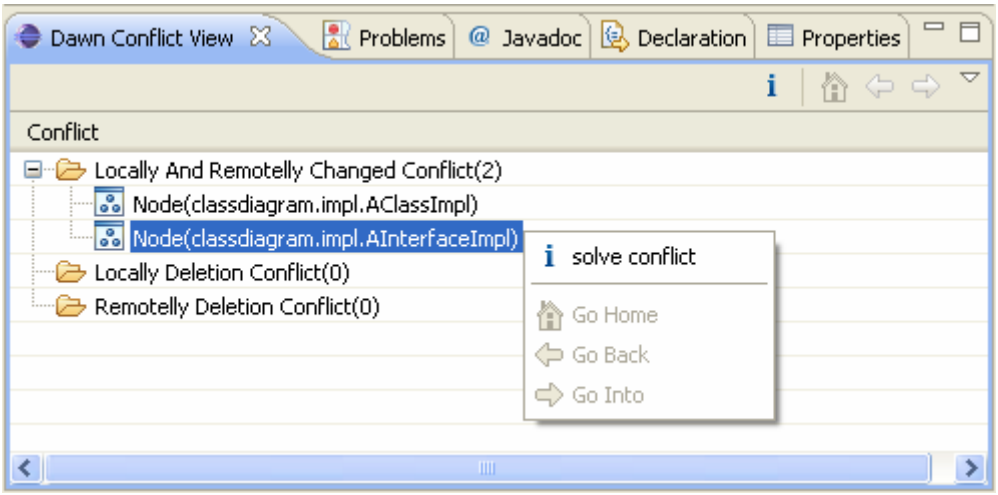

Abbildung 53 - Dawn Conflict View

In der View werden, sortiert nach den drei Konflikten, alle betroffenen Objekte angezeigt [vgl. Abbildung 53]. Für jedes Objekt steht das bereits beschriebene Kontext-Menü zur Verfügung, um den Konflikt aufzulösen.

Allerdings versorgt der ContentProvider die TreeView nur beim Start mit den Informationen. Damit die View auch im laufenden Betrieb die aktuellen Konflikte anzeigt, muss sobald sich eine Änderung in den Konflikten ergibt, diese sofort reflektiert werden können. Hierzu wird das *Observer*-Entwurfsmuster verwendet, bei dem sich eine Instanz, der *Observer*, an einem zu beobachtenden Objekt (*Observable*) registriert. Sobald im *Observable* eine Änderung auftritt, kann es alle Observer über seinen aktuellen Zustand informieren. Der Observer, in diesem Fall die *Dawn Conflict View*, muss dazu die Methode **`update`** der Schnittstelle Observer implementieren. Das Gegenstück bildet der ResourceSynchronizer, der die Konflikt-Objekte verwaltet. Er muss die Klasse **`Observable`** aus dem Paket **`java.util`** erweitern. Durch diese Erweiterung erhält er Methoden mit deren Hilfe sich die Observer verwalten und im Bedarfsfall informieren lassen. Das Nutzen der Conflict-View und das Beseitigen von Konflikten in Dawn werden in einem Screencast unter [Flügge 2009d] dargestellt.

5.4.3 Locking

Für die Umsetzung des Lockings wurde, wie bei dem Konflikt-Mechanismus, ein zusätzliches Popup-Menü erstellt, welches nur auf den Elementen eines Diagramms sichtbar ist. Wie bei den Konflikten wird ebenfalls der Erweiterungspunkt **`org.eclipse.ui.popupMenus`** genutzt, um das Menü zu erstellen. Der Nutzer hat für das gewählte Objekt die Möglichkeit es zu locken oder wieder zu entsperren. Es können auch mehrere Objekte markiert werden.

Wird ein Objekt gelockt, so wird diese Information an den Server übertragen und in einer Liste hinterlegt. Das Objekt wird lokal mit einer grünen Umrandung markiert, um dem Nutzer schnell einen Überblick darüber zu geben, welche Objekte exklusiv für ihn reserviert sind [vgl. Abbildung 54, rechts]. Da der Server, aufgrund der Architektur, diese Information nicht sofort an die Clients verteilen kann, muss er warten bis die Clients sich diese Information besorgen. Serverseitig werden alle Locks durch die jeweiligen Projekte verwaltet. Dazu werden lediglich die XMI-IDs der betroffenen Objekte in einer Liste vorgehalten.

Bei jedem Polling-Intervall wird als erstes vom Client diese Liste vom Server geladen. Alle Objekte, die nicht vom lokalen Nutzer gelocked wurden, werden durch eine gelbe Umrandung markiert und können nicht mehr editiert werden [vgl. Abbildung 54, links].

Die gelockten Objekte werden sofort eingefroren. Sobald der Nutzer, der den Lock auf ein Objekt gesetzt hat, eine Änderung publiziert, wird diese auf allen anderen Clients verbreitet, unabhängig davon ob bereits Änderungen an dem Objekt ausgeführt wurden.

Hierbei werden keine Konflikte erkannt. Es gilt die Regel, dass Locks über lokalen Änderungen stehen. In [Flügge 2009c] kann der Vorgang des Lockings visuell betrachtet werden.

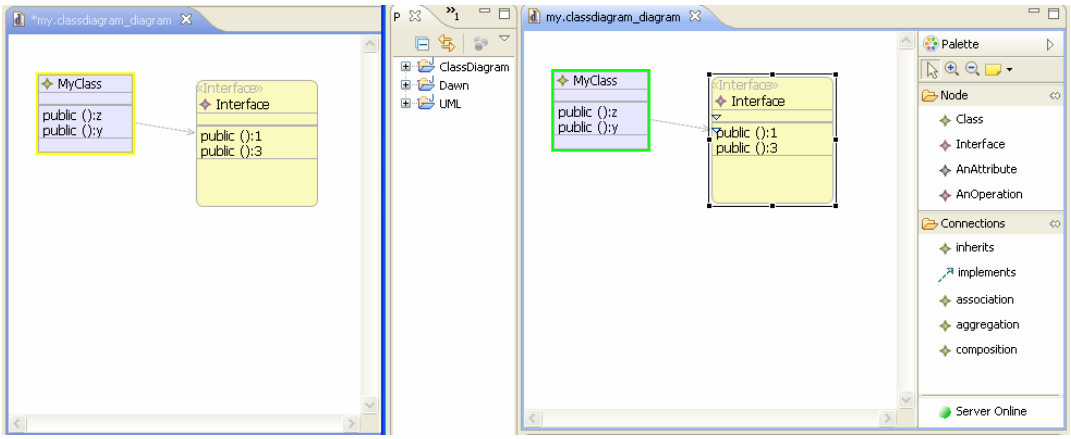

Abbildung 54 - Locken eines Objektes

5.4.4 Extenions Points und Dawn Extension Service

Die Auslagerung von globalen Komponenten wie den Preferences, den Konflikt-Komponenten und dem Locking-Mechanismus ermöglicht es den Quellcode an einer Stelle zentral zu verwalten. Zusätzlich wird verhindert, dass Systemteile zu oft registriert werden. Würden die Menüs für die Konfliktbehandlung in jedem Editor über den entsprechenden Extension Point angemeldet, so würde der unschöne Effekt entstehen, dass das Dawn-Menü so oft angezeigt würde, wie Editoren vorhanden sind. Das Auslagern bringt aber ein nicht unwesentliches Problem mit sich. Um beispielsweise einen Konflikt aufzulösen, wird der ElementType einer View benötigt, wenn sie restauriert werden soll. Dieser spezielle ElementType ist aber spezifisch für jeden Editor. Zwar stellt der generierte ElementTypeHelper dieses Mapping zur Verfügung, doch leider ist dies nicht in der Dawn-Runtime verfügbar, da sie ausschließlich globale Informationen verwaltet, die losgelöst von den konkreten Editoren ist. Das Ableiten von den entsprechenden Klassen würde wieder zu dem Problem führen, dass der spezifische Editor die ausführende Instanz ist, nicht aber die Runtime. Dieses Problem ähnelt der bereits in [2.8.3.3] exemplarisch dargestellten Problematik. Die Dawn-Runtime (in Analogie zur Komponente *a* aus 2.8.3.3) möchte als aktive Komponente Teile des erweiterten Editors nutzen. Zur Lösung dieses Problems gibt es nur zwei Möglichkeiten. Die erste Möglichkeit bestünde darin, dass der erweiterte Editor die benötigten Objekte, wie zum Beispiel den ElementTypeHelper, initialisiert und an eine statische Singleton-Komponente bindet, die auch von der Runtime aus zugänglich ist. Dies verursacht aber zum einen, dass die Komponenten

bereits instanziiert werden, obwohl sie noch nicht benötigt werden. Zum anderen ist auch der Umgang mit Singletons in Eclipse sparsam zu nutzen.

Eine weitere und wesentlich elegantere, und deshalb auch eingesetzte Lösung, ist die Nutzung von Extension Points zur Registrierung der einzelnen Systemteile.

Um von einem Plugin aus einen Extension Point zur Verfügung stellen zu können, wird eine XML-Schema Definition (*.exsd) erstellt, in der beschrieben wird, welche Parameter die Erweiterung anbieten soll und unter welchem Namen sie am Plugin gefunden werden kann. Hierfür stellt Eclipse einen Editor bereit, mit dem diese Einstellungen komfortabel vorgenommen werden können [vgl. Abbildung 55].

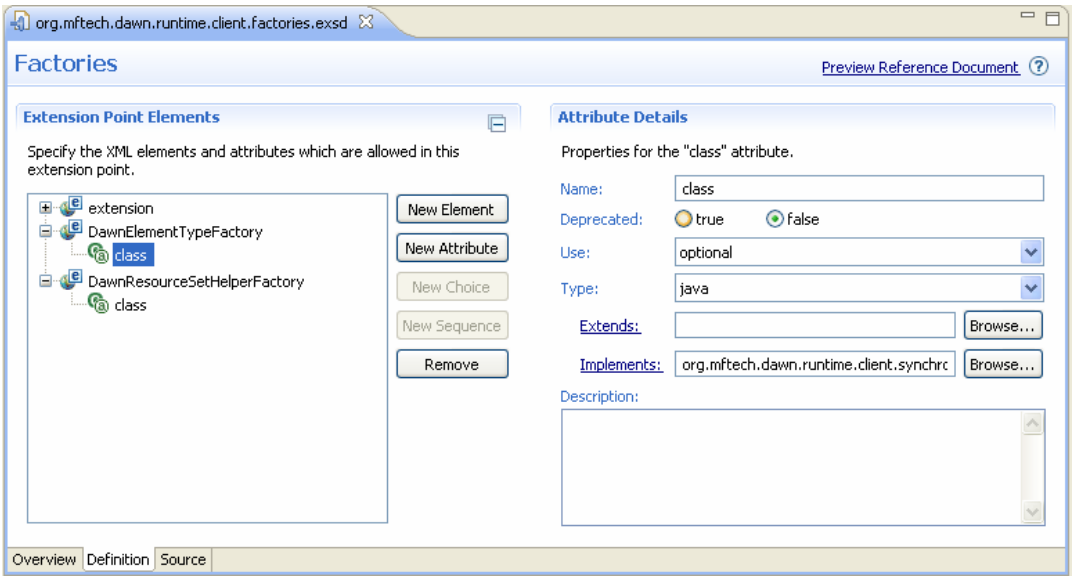

Abbildung 55 - Extension Point Schema Editor

Für die Dawn-Runtime wurden zwei Erweiterungspunkte definiert. Einer für die Nutzung des ElementTypeHelpers und einer für das ResourceSet des jeweiligen Editors. Um die Komponenten aber einfach trennen zu können und es dem clientseitigen Entwickler zu erlauben, einfach die konkreten Implementierungen auszutauschen, werden die Komponenten hinter Schnittstellen verborgen, deren konkrete Implementierung durch Fabriken erzeugt wird. An den Erweiterungspunkten werden die Fabriken der jeweiligen Komponente registriert.

Damit die Dawn-Runtime zur Laufzeit auf alle angemeldeten Erweiterung zugreifen kann, nutzt sie den *DawnExtensionService*. Diese Komponente kann auf die Extension-Registry von Eclipse zugreifen und alle Erweiterungen abfragen. In Listing 9 ist das Auslesen der **`ElementTypeHelperFactory`** dargestellt. In den Zeilen 8-9 wird eine Liste von Konfigurationselementen (**`IConfigurationElement`**) abgefragt, die unter dem für die Dawn-Runtime definierten Identifikator erreichbar sind. In Zeile 13 wird

überprüft, ob diese Erweiterung dem richtigen Typ entspricht. Damit können die angemeldeten Fabriken von der Dawn-Runtime abgefragt werden. Allerdings kann damit noch nicht die konkret benötigte Fabrik gefunden werden, da alle DawnElementTypeHelperFactories über diesen Filter sichtbar sind. Wird aber beispielsweise in Editor A das Konflikt-Menü genutzt, so muss auch dessen passende Fabrik gefunden werden. Zur Lösung dieses Problems, wurde an die Methode die *Plugin-ID* des aktuellen Editors übergeben. Diese entspricht dem Namensraum des zu findenden Konfigurationselements. In Zeile 16 kann dadurch auf den gleichen Namensraum geprüft und die richtige Erweiterung zurückgeliefert werden.

```
1   public DawnElementTypeHelperFactory
2   createDawnElementTypeHelperFactory(String pluginId)
3   {
4     if (dawnElementTypeHelperFactory == null)
5     {
6
7         IConfigurationElement[] config =
8   Platform.getExtensionRegistry()
9   .getConfigurationElementsFor(DAWN_FACTORY_ID);
10        for (IConfigurationElement e : config)
11        {
12           Object o = e.createExecutableExtension("class");
13           if (o instanceof DawnElementTypeHelperFactory)
14           {
15              if(e.getNamespaceIdentifier().equals(pluginId))
16                 return (DawnElementTypeHelperFactory) o;
17           }
18        }
19    }
20              return null;
21  }
```

Listing 9 - Dawn Extension Service

5.4.5 Preferences

Die Konfigurationseinstellungen für die Dawn-Editoren [vgl. 4.9.2] können über spezielle Seiten bearbeitet werden. Eclipse hält hierfür das Konzept der *Preferences* bereit. Diese können gemeinsam unter *Window* → *Preferences* in jeder Eclipse-Instanz aufgerufen werden. Das Nutzen des Eclipse-eigenen Präferenz-Mechanismus hat zwei entscheidende Vorteile. Zum einen wird die komplette Persistierung der Daten von Eclipse übernommen. Der Entwickler muss sich also keine Gedanken darüber machen wo und in welcher Form die Konfigurationsdaten abgespeichert werden. Jede Änderung wird automatisch übernommen.

Zum anderen müssen keine eigenen Dialoge für das Verwalten der Präferenzen implementiert werden. Dadurch können die Konfigurationseinstellungen auch sehr intuitiv angeboten werden. Jeder, der ein Grundverständnis der Eclipse IDE hat, kennt das Konzept der Preferences und wird dort als erstes nach Einstellungen der Plugins suchen.

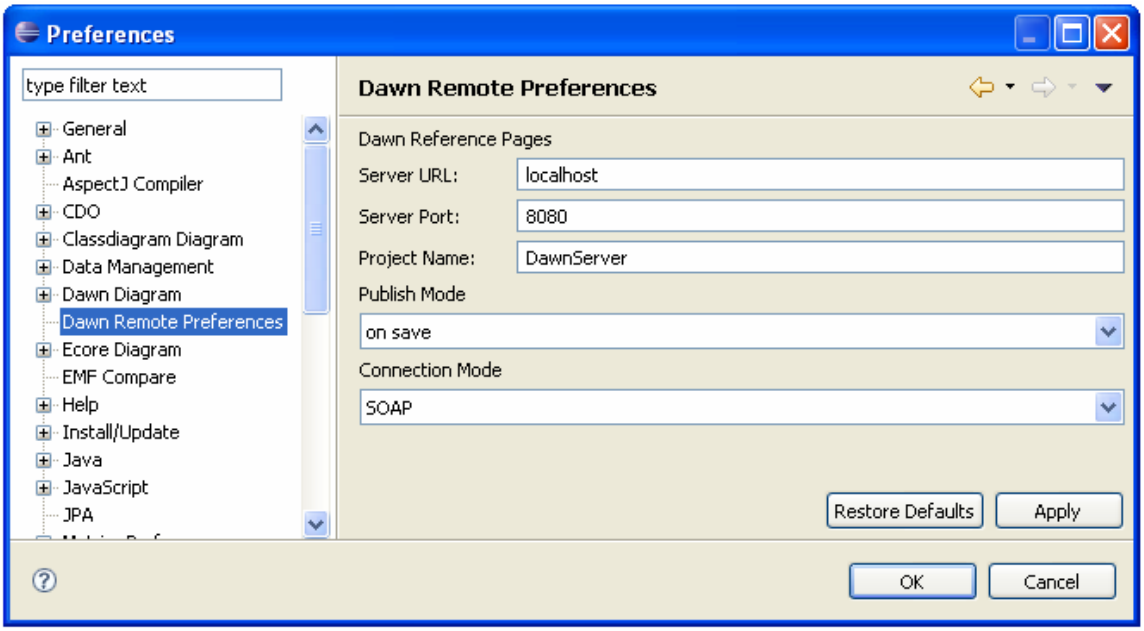

Abbildung 56 - Dawn Preferences

Um eigene *Preferences* in Eclipse integrieren zu können, bietet Eclipse zwei Extension Points (**`org.eclipse.ui.preferencePages`** und **`org.eclispe.core.runtime.preferences`**) an [vgl. Daum 2006, S.252]. Über den Erweiterungspunkt **`org.eclipse.ui.preferencePages`** kann eine eigene Präferenzseite in das System eingebunden werden. Dazu muss lediglich der Name, der in den Preferences angezeigt werden soll, und die Klasse, die die Anzeige der Präferenzen übernimmt, angegeben werden. Abhängig davon, welche Daten angezeigt werden, kann diese Klasse unterschiedliche Implementierungen erweitern. Da die Dawn-Preferences *Felder* zur Eingabe nutzen, wird die Klasse **`FieldEditorPreferencePage`** erweitert. Innerhalb der **`DawnRemotePreferencePage`** können SWT-Widgets genutzt werden, um die GUI im Aussehen anzupassen und die einzelnen Einstellungsmöglichkeiten grafisch darzustellen. Abbildung 56 zeigt die fertigen Dawn-Preferences.

Damit diese Einstellungen initialisiert und mit Default-Werten belegt werden können, muss der zweite Extension Point erweitert werden. Über die Erweiterung **`org.eclispe.core.runtime.preferences`** wird eine weitere Klasse in das System eingebunden, die die Initialisierung der Daten übernimmt. Diese Klasse (**`PreferenceInitializer`**) erweitert die abstrakte Basisklasse

`AbstractPreferenceInitializer`. Dadurch, dass die Implementierung über den Extension Point im System registriert ist, kann die Eclipse-IDE diese beim Start aufrufen und alle Dawn-Parameter mit Initialwerten belegen.

Die Preferences sind global verfügbar und werden von dem Dawn-Runtime Plugin zur Verfügung gestellt.

5.4.6 Offline-Server

Der Offline-Server kommt immer dann zum Einsatz, wenn der reale Server nicht mehr verfügbar ist. Um in einem solchen Fall den Client mit den aktuellsten Nutzerinformationen versorgen zu können, muss er regelmäßig die notwendigen Informationen mit dem Online-Server abgleichen. Ist er einmal instanziiert, so steht er für alle parallel laufenden Editoren zur Verfügung. Aus diesem Grund ist er auch in die Dawn-Runtime mit eingebettet. Allerdings reicht das Abgleichen mit dem sich im Hauptspeicher befindlichen Objekt nicht aus, um eine durchgängige Netzwerkunabhängigkeit zu erreichen. Wird die Eclipse-IDE geschlossen, müssen die Daten beim nächsten Neustart auch wieder zur Verfügung stehen, auch wenn keine Verbindung zum Online-Server hergestellt werden kann. Aus diesem Grund werden die Daten des Offline-Servers bei jeder Aktualisierung gespeichert. Da der Offline-Server global für alle Editoren zur Verfügung steht, werden seine Informationen nicht innerhalb eines bestimmten Objektes angelegt, sondern auf oberster Hierarchie im Wurzelverzeichnis des aktuellen *Workspace*. Für die Speicherung aller notwendigen Dateien wird ein eigenes Verzeichnis angelegt (**`.dawnmetadata`**), um die Informationen auch wieder sauber aus dem Workspace löschen zu können und um keine versehentlichen Konflikte mit anderen Dateien auszulösen. Dabei ist die Offline-Server-Implementierung selbst für das Laden und das Speichern verantwortlich. Die konkrete Implementierung bestimmt also, wie die Daten hinterlegt werden. Dadurch kann das Format bei Bedarf jederzeit durch eine andere Implementierung ersetzt werden. In der aktuellen Variante wird der Serialisierungs-Mechanismus von Java genutzt, damit ein Abbild des Offline-Servers gespeichert werden kann. Listing 10 zeigt, wie der Zugriff auf die zu speichernde Datei realisiert wird.

```
1   private String getSerializationFilePath()
2   {
3    String folderPAth=
4    ResourcesPlugin.getWorkspace().getRoot()
5     .getLocation()+"/.dawnmetadata";
6
7     File folder = new File(folderPAth);
8     if(!folder.exists())
9     {
10        folder.mkdir();
11    }
12     return folderPAth+"/dawnOfflineServer.ser";
13
14  }
```

Listing 10 - Zugriff auf das Workspace-Root

Der Offline-Modus bringt allerdings noch ein weiteres Problem mit sich. Ist der Online-Server verfügbar, werden die Locking-Informationen anhand der Nutzerkennung verwaltet. Startet nun aber der Client, ohne das der Server vorhanden ist, so wird auch kein Login-Dialog abgefragt, um den Nutzer nicht zu verwirren und ihn glauben zu lassen, dass der Server vorhanden wäre. Doch ohne, dass der Nutzer seine Kennung eingegeben hat, kann auch der Offline-Server zwar die gelockten Objekte senden, der Client aber nicht entscheiden, welche von ihm und welche von anderen Nutzern gelockt wurden. Infolgedessen würden alle gelockten Objekte lokal als von fremden Nutzern gesperrt markiert werden. Der Nutzer könnte diese nicht editieren, selbst wenn er sie für sich reserviert hat. Um dieses Problem zu beseitigen, speichert der Offline-Server neben den Projekt-Daten auch die zuletzt angemeldete Nutzerkennung mit ab. Wird der Server gestartet, so wird die Kennung geladen und in der lokalen Session hinterlegt. Dadurch kann der Nutzer, auch ohne Netzwerkverbindung, seine spezifischen Informationen vom Offline-Server erhalten. Dieser verweigert aber jegliche Anfragen zur Änderung des Lock-Status eines Objektes. Es ist also nicht möglich Objekte zu locken oder sie zu entsperren.

5.5 Persistenz

Hibernate abstrahiert den Zugriff auf das darunter liegende Datenbankbetriebssystem und bietet eine Schicht für das Umsetzen von objektorientieren Objekten auf relationale Tabellen. Um den Zugriff mittels Hibernate aber weiter zu abstrahieren und dadurch auch eine Unabhängigkeit von der objektrelationalen Implementierung zu bekommen, stellt Dawn eine Zwischenschicht zur Verfügung - den *DBManager*. Für jede Klasse die

in der Datenbank gespeichert werden soll, liefert der DBManager verschiedene, auf die Klassen zugeschnittene Funktionen an, um die Objekte laden, speichern oder löschen zu können [vgl. Abbildung 57].

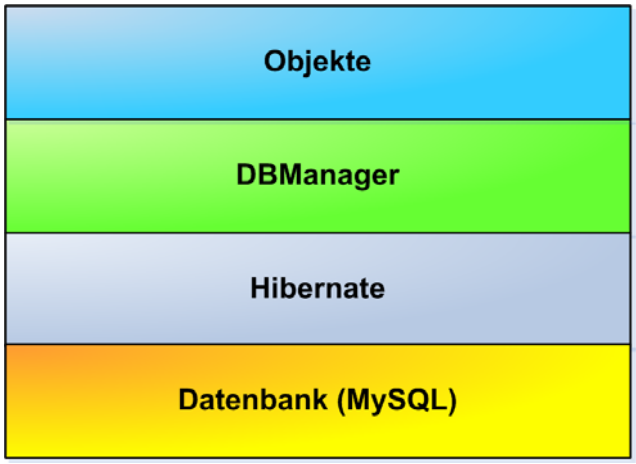

Abbildung 57 - DBManager

Um diese Methoden nicht von Hand generieren zu müssen, wurde eine einfache Generator-Klasse geschrieben, welche, mit den entsprechenden Klassen instanziiert, den DBManager erstellen kann. Dieser simple Generator nennt sich **`DBManangerCreator`**.

```
1   public class HibernateCreationUtil
2   {
3
4    private static final SessionFactory sessionFactory;
5
6     static
7     {
8       try
9       {
10        Configuration configuration = new AnnotationConfiguration();
11        Configuration
12         .setProperty("hibernate.hbm2ddl.auto", "create-drop");
13
14         sessionFactory =
15         configuration.configure().buildSessionFactory();
16      }
17      catch (Throwable ex)
18      { ... }
19  ...
20     }
21  }
```

Listing 11 – HibernateCreationUtil

Hibernate kann so konfiguriert werden, dass es nicht nur die Daten in die Datenbank überträgt, sondern diese auch erstellt. Dazu muss in der globalen Konfiguration das Attribut **`hibernate.hbm2ddl.auto`** auf den Wert **`create-drop`** gesetzt werden. Dies kann sowohl in der Konfigurations-XML als auch programmtechnisch geschehen.

Dawn bietet die Möglichkeit an, dass der Nutzer die Datenbank automatisch erstellt. Hierzu erkennt der Dawn-Server beim ersten Login über die Web-Schnittstelle, dass die Datenbank nicht vorhanden ist und ermöglicht dem Nutzer diese zu erstellen. Um dies programmtechnisch zu realisieren, wird bei dem Generierungsvorgang eine Klasse erstellt, welche die angegebenen Nutzerinformationen aus dem Dawn-GenModel enthält. Diese Klasse enthält Methoden, die über die Aktion ausgelöst die Datenbank initialisieren. Technisch wird diese durch eine spezielle Implementierung der Klasse **`HibernateUtil`** erreicht, welche den Zugriff auf die Datenbank für den Initialisierungsprozess so konfiguriert, dass alle Tabellen neu erstellt werden [vgl. Listing 11].

5.6 Web-Viewer

Kern-Komponente für das Anzeigen der Modell-Informationen des Servers in einem Browser ist das unter 4.8.1 beschriebene Diagramm-Servlet, welches die Modell-Informationen vom Dawn-Server geliefert bekommt und diese als HTML/JavaScript Figuren ausgibt [vgl. 4.8.1].

Das Problem bei der Anzeige der View ist, dass das Diagramm-Servlet lediglich die Modellinformationen, also die reinen Daten des Modells, sowie Modelldaten der Figuren vom Dawn-Server geliefert bekommt. Das Servlet hat also Kenntnis darüber, wo sich die Knoten und Kanten befinden, und welche Größe sie haben, aber keine weiteren Informationen über deren Aussehen. Auf der Clientseite übernimmt das Layout eine generierte Figure-Klasse, welche in den EditPart eingebettet ist. Auf der Serverseite werden JavaScript-Figuren für die Anzeige genutzt, welche in Dawn basierend auf dem GMF-GenModel generiert wurden [vgl.5.9.3].

Allein diese Modell-Informationen reichen allerdings nicht aus. Deshalb wurde im Konzept vorgesehen die fehlenden Informationen durch eine Meta-Datei (config.xml) zu ergänzen. Diese beinhaltet die Zuordnung zwischen einem View-Element und seiner passenden JavaScript-Repräsentation und weitere anzeigespezifische Daten. Listing 12 zeigt das der config.xml zugrunde liegende Schema.

Da der Server viele verschiedene Editoren verwalten kann und demzufolge verschiedene JavaScript-Pakete unterstützen muss, werden alle editoreigenen Web-Daten in einem Ordner zusammengefasst, der als Namen die Paketbezeichnung des Editors trägt. Der prototypische GMF-Editor würde beispielsweise in dem Verzeichnis **`org.mftech.diagram.uml.class.diagram`** abgelegt werden. Dieses Vorgehen dient nicht nur der Separierung der einzelnen Editoren, sondern wird auch als Identifikator

genutzt, damit das Diagramm-Servlet die richtigen Repräsentationen für den Editor anzeigen kann. Jedes dieser Verzeichnisse enthält einen Ordner für die Meta-Daten (*meta*), welches die config.xml enthält und in dem bei Bedarf weitere Meta-Daten untergebracht werden können. Zusätzlich befindet sich unter diesem Root-Verzeichnis das Verzeichnis *icons*, das die Bild-Dateien des GMF-Editors beinhaltet [vgl. Abbildung 58, rechts].

```
1   <?xml version="1.0" encoding="UTF-8"?>
2               <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3   elementFormDefault="qualified">
4                 <xs:element name="figureMappings">
5                   <xs:complexType>
6                     <xs:sequence>
7                       <xs:element maxOccurs="unbounded"
8   ref="figureMapping"/>
9                     </xs:sequence>
10                  </xs:complexType>
11                <xs:element>
12                <xs:element name="figureMapping">
13                  <xs:complexType>
14                    <xs:sequence>
15                      <xs:element ref="javaScriptClass"/>
16                      <xs:element minOccurs="0" maxOccurs="unbounded"
17  ref="viewAttribute"/>
18                      <xs:element minOccurs="0" ref="viewPattern"/>
19                    </xs:sequence>
20                    <xs:attribute name="type" use="required"
21  type="xs:integer"/>
22                  </xs:complexType>
23                <xs:element>
24                <xs:element name="javaScriptClass">
25                  <xs:complexType>
26                    <xs:attribute name="name" use="required"/>
27                  </xs:complexType>
28                <xs:element>
29                <xs:element name="viewAttribute">
30                  <xs:complexType>
31                    <xs:attribute name="name" use="required"
32  type="xs:NCName"/>
33                  </xs:complexType>
34                <xs:element>
35                <xs:element name="viewPattern">
36                  <xs:complexType>
37                    <xs:attribute name="name" use="required"/>
38                  </xs:complexType>
39                <xs:element>
40              <xs:schema>
```

Listing 12 - config.xml Schema

Die zu erstellenden JavaScript-Views unterteilen sich in vier Kategorien – Knoten, Kanten, Kanten-Dekoratoren und Label. Um die spätere Generierung und die Nutzung beziehungsweise die Erweiterung der JavaScript-View zu vereinfachen, wurde für jede

dieser Repräsentationen eine Basis-Klasse erschaffen, von der die konkreten Ausprägungen das grobe Aussehen erben und über Parameter erweitern können. Diese Basis-Klassen wurden ebenfalls in einem speziellen Ordner abgelegt (**`org.mftech.dawn.basic`**). Abbildung 58 zeigt wie die Informationen aus dem GMF-Modell (links) in Java-Script Klassen (rechts) umgewandelt wurden.

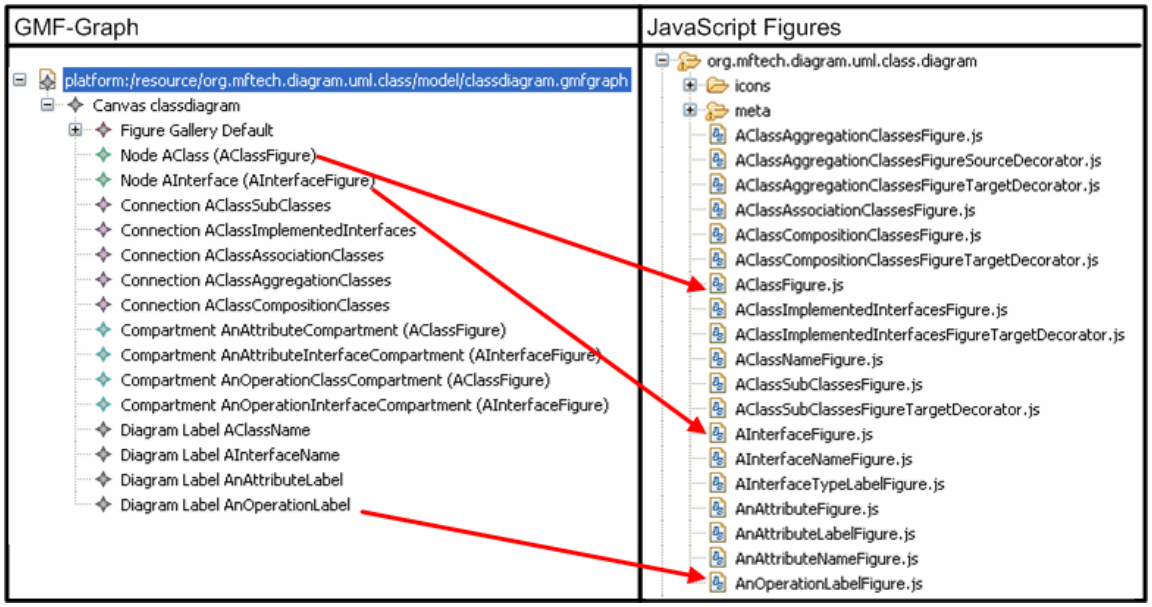

Abbildung 58 - Zuordnung GMF-Modell zu JavaScript-Views

Das Diagramm-Servlet generiert den kompletten JavaScript-Quelltext für die Anzeige eines bestimmten Diagramms. Eine Schwierigkeit bei der Erstellung des Quellcodes war es, die Variablenbezeichner dynamisch zu erstellen. Jeder Knoten und jede Kante musste einen eindeutigen Variablennamen besitzen, damit der Quelltext syntaktisch korrekt ist. Um dies zu bewerkstelligen, wurden die XMI-IDs, welche jede View eindeutig beschreibt, als Variablennamen verwendet. Leider können XMI-IDs syntaktische Eigenheiten aufweisen, die es verhindern, sie als gültige JavaScript-Variablen zu nutzen. Beispielsweise können XMI-IDs das Minus-Zeichen enthalten. Um diese Unzulänglichkeit auszubessern, wurden mittels eines Konverters die XMI-IDs in einen gültigen String umgewandelt. Zum besseren Verständnis der Funktionsweise des Diagramm-Servlets ist in Listing 13 ein einfaches Modell (semantic und notational), wie der Server es speichert, dargestellt. Listing 14 und Listing 15 zeigen den generierten JavaScript-Quelltext, wie er auf dem Client angezeigt wird, und die config.xml für die Zuordnungen. Der Übersicht halber wurden Deklarationen und sonstige nicht benötigte Inhalte weggelassen. Wichtige Aspekte sind in dem Listing fett dargestellt.

Zeile 1 [Listing 14] erstellt die Zeichenfläche, also das Diagramm. In den Zeilen 2-3 wird ein Knoten vom Typ **`AClassFigure`** instanziiert. Hierbei wird als

Variablenbezeichner die umgewandelte XMI-ID aus dem Notational-Modell verwendet [Listing 13, Zeile 20; Listing 14, Zeile 5]. Über den Type-Identifikator kann das Diagramm-Servlet mit Hilfe der config.xml [Listing 15, Zeilen 3-5] die Zuordnung zur JavaScript-Figure **`AClassFigure`** treffen. Da dieser Knoten ein Label enthält (Type 4002), wird dieser ebenfalls erzeugt und dem Knoten hinzugefügt [Listing 14, Zeile 5]. Der Wert, der in dem Label angezeigt werden soll, ist das Attribut **`name`**, welches sich im semantischen Modell der Klasse befindet [Listing 13, Zeile 6].

```
1   <!------- semantic model ------------->
2   <?xml version="1.0" encoding="UTF-8"?>
3   <classdiagram:ClassDiagram xmi:id="_xZLOYe-zEd2hpuKCIV2-UA">
4       <classes
5           xmi:type="classdiagram:AClass"
6           xmi:id="_pxDwwPLEEd2tPuftDUWv9g" name="Klasse" >
7           <operations xmi:type="classdiagram:AnOperation"
8                       xmi:id="_4Q4sER4hEd69o8NIsUctMg"
9                       name="foo"
10                      accessright="public"
11                      dataType="int"/>
12  </classes>
13  </classdiagram:ClassDiagram>
14
15  <!------- notational model ------------->
16  <notation:Diagram xmi:id="_xZLOYu-zEd2hpuKCIV2-UA"
17  type="Classdiagram" element="_xZLOYe-zEd2hpuKCIV2-UA"
18  name="default6.classdiagram_diagram" measurementUnit="Pixel">
19  <children xmi:type="notation:Node"
20            xmi:id="_pxDwwfLEEd2tPuftDUWv9g" type="1002"
21            element="_pxDwwPLEEd2tPuftDUWv9g">
22        <children xmi:type="notation:Node"
23                  xmi:id="_pxDwxPLEEd2tPuftDUWv9g" type="4002"
24                  element="_pxDwwPLEEd2tPuftDUWv9g"/>
25        <children xmi:type="notation:Node"
26                  xmi:id="_pxDwyPLEEd2tPuftDUWv9g"
27                  type="5004"
28                  element="_pxDwwPLEEd2tPuftDUWv9g">
29                  <children xmi:type="notation:Node"
30                            xmi:id="_4UTzgB4hEd69o8NIsUctMg"
31                            type="2004"
32                            element="_4Q4sER4hEd69o8NIsUctMg"/>
33        </children>
34        <layoutConstraint xmi:type="notation:Bounds"
35                          xmi:id="_pxDww_LEEd2tPuftDUWv9g"
36                          x="115" y="190" width="86" height="46"/>
37      </children>
38    </notation:Diagram>
39  </xmi:XMI>
```

Listing 13 - vereinfachtes Modell (semantic und notational)

Diese Information wird ebenfalls aus der config.xml [Listing 15, Zeile 13] gewonnen. Wie bereits beschrieben, wird der eigentliche Wert des Attributes mit Hilfe von

Dynamic EMF ermittelt. Die Layout-Informationen des Knoten werden aus dem Notational-Modell gewonnen [Listing 13, Zeile 36] und die JavaScript-Figure damit parametrisiert [Listing 14, Zeile 17-18].

Zusätzlich enthält dieser Knoten aber ein *Compartment*, welches ein Label für die Anzeige einer Operation nutzt. Von dieser Operation werden drei Attribute (*accessright*, *datatype* und *name*) angezeigt [Listing 15, Zeilen 18-20]. Hierbei muss das View-Pattern beachtet werden, welches der GMF-Entwickler auch für den GMF-Editor vorgesehen hat. Dieses wurde in die config.xml generiert und wird vom Diagramm-Servlet ausgewertet, um die Parameter in der richtigen Reihenfolge und durch die vorher definierten Trenn-Symbole darzustellen [Listing 14, Zeilen 12-13; Listing 15, Zeile 21]. Das Ergebnis der Quelltext-Generierung ist in Abbildung 59 zu sehen.

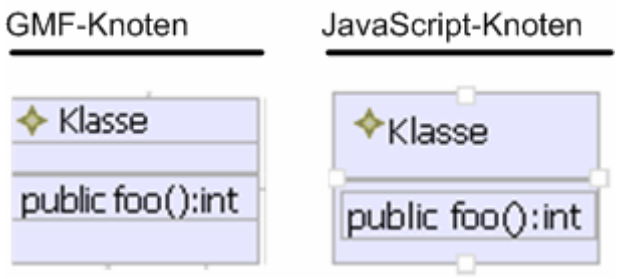

Abbildung 59 - GMF- und JavaScript-Knoten im Vergleich

Mit Hilfe des Web-Viewers können aber nicht nur einfache viereckige Knoten dargestellt werden, sondern auch verschiedenste Kanten, Knoten mit runden Ecken oder Konnektoren an den Kantenenden. Abbildung 60 stellt einen Vergleich zwischen einem Diagramm in dem prototypischen Editor (rechts) und der entsprechenden Darstellung im Browser (links) dar. In Kapitel 5.9.3 wird beschrieben wie mit Hilfe des GMF-Graph-Modells das Aussehen der JavaScript-Figuren beeinflusst werden kann.

```
1   var workflow  = new draw2d.Workflow("");
2   var _pxDwwfLEEd2tPuftDUWv9g = new
3   org.mftech.diagram.uml.clazz.AClassFigure();
4
5   var _pxDwxPLEEd2tPuftDUWv9g_0 = new
6   org.mftech.diagram.uml.clazz.AClassNameFigure("Klasse");
7   _pxDwwfLEEd2tPuftDUWv9g.addChild(_pxDwxPLEEd2tPuftDUWv9g_0);
8
9   var _pxDwyPLEEd2tPuftDUWv9g_2 = new
10  org.mftech.dawn.basic.DawnCompartmentFigure("");
11  var _4UTzgB4hEd69o8NIsUctMg_0 = new
12  org.mftech.diagram.uml.clazz.AnOperationLabelFigure("public foo()
13  : int");
14  _pxDwyPLEEd2tPuftDUWv9g_2.addChild(_4UTzgB4hEd69o8NIsUctMg_0);
15  _pxDwwfLEEd2tPuftDUWv9g.addChild(_pxDwyPLEEd2tPuftDUWv9g_2);
16
17  _pxDwwfLEEd2tPuftDUWv9g.setDimension(86, 46);
18  workflow.addFigure(_pxDwwfLEEd2tPuftDUWv9g,115,190);
```

Listing 14 - Ausschnitt - erzeugter Diagram Quelltext

```
1   <?xml version="1.0" encoding="ISO-8859-1"?>
2   <figureMappings>
3         <figureMapping type="1002">
4               <javaScriptClass
5   name="org.mftech.diagram.uml.clazz.AClassFigure" />
6         </figureMapping>
7
8         <figureMapping type="4002">
9               <javaScriptClass
10  name="org.mftech.diagram.uml.clazz.AClassNameFigure" />
11              <viewAttribute name="name" />
12              <viewPattern name="" />
13        </figureMapping>
14
15        <figureMapping type="2004">
16              <javaScriptClass
17  name="org.mftech.diagram.uml.clazz.AnOperationLabelFigure" />
18              <viewAttribute name="accessright" />
19              <viewAttribute name="dataType" />
20              <viewAttribute name="name" />
21              <viewPattern name="{0} {2}():{1}" />
22        </figureMapping>
23
24        <figureMapping type="5004">
25              <javaScriptClass
26  name="org.mftech.dawn.basic.DawnCompartmentFigure" />
27        </figureMapping>
28  </figureMappings>
```

Listing 15 - exemplarische config.xml

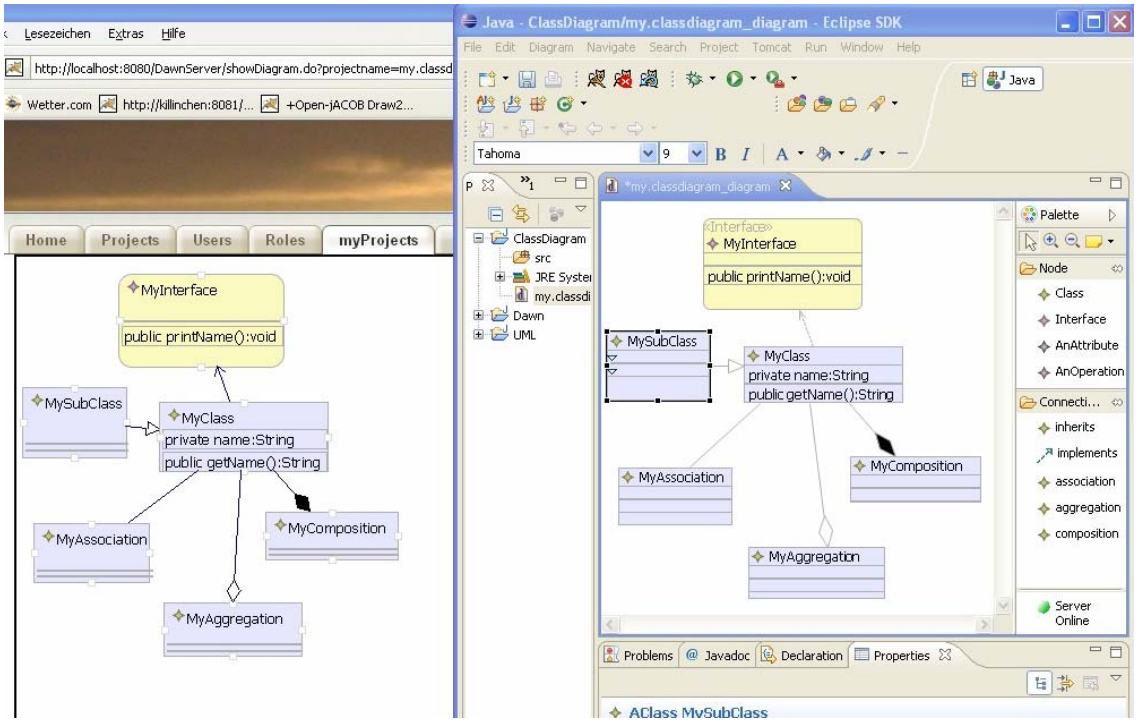

Abbildung 60 - Web-Viewer (links) und GMF-Editor (rechts)

Für die Ajax-basierte Kommunikation des Web-Viewers mit dem Backend [vgl. 4.8.1.2] wurde die freie JavaScript-Bibliothek *Prototype*[41] genutzt. Bedingt durch unterschiedliche *XMLHttpRequest* Implementierungen in den verschiedenen Browsern, müssten verschiedene Wege zum Aufbau der Kommunikation vorgesehen werden. *Prototype* kapselt diese Funktionalität bereits, was ein einfacheres und fehlerfreieres Programmieren ermöglicht.

Um die Darstellungsschicht von den dynamischen Elementen innerhalb des *Diagramm-Servlets* zu trennen, wurden die statischen Bestandteile des Servlets in eine *JSP* (*Java Server Page*) ausgelagert. Hierzu zählen sowohl die strukturgebenden Tags wie *<html>*, *<head>* oder *<body>*, als auch die Include-Anweisungen, die von allen Diagrammen genutzt werden. Die dynamischen Aspekte, also die Diagramm-spezifischen Inkludierungen und das eigentliche Diagramm, wurden gesondert generiert. Um diese ebenfalls in den Kontext der JSP einbetten zu können, wurde die dynamische Generierung in eigens dafür erstellte Tags ausgelagert. Hierzu wurde eine Tag-Bibliothek erstellt, über welche die Tags zugreifbar gemacht wurden. Listing 16 stellt auszugsweise die Web-Viewer-JSP und den Zugriff auch die dynamischen Aspekte dar.

[41] http://www.prototypejs.org/

```
1   <%@ taglib uri="http://www.mftech.org/dawn/tags" prefix="dawn"%>
2   <html xmlns="http://www.w3.org/1999/xhtml">
3   <head>
4   ...
5   <dawn:webviewerincludes
6   projectName="<%=request.getParameter("projectName")%>" />
7   ...
8   <script type="text/javascript">
9   <dawn:webviewerjs
10  projectName="<%=request.getParameter("projectName")%>" />
11  </script>
12  ...
```

Listing 16 - Web-Viewer JSP - Ausschnitt

5.7 Web Konfigurations-GUI und Projektverwaltung

Neben dem Web-Viewer unterstützt eine weitere webbasierte Komponente den Nutzer bei der Arbeit. Projekte lassen sich innerhalb von Dawn ausschließlich über das Web-Frontend konfigurieren [vgl. 4.8.2]. Diese Vorgehensweise hat zwei entscheidende Vorteile. Zum einen kann ein Benutzer Änderungen an einem Projekt vornehmen, ohne auf eine Eclipse-IDE angewiesen zu sein. Er kann also einfach und bequem von unterwegs einzelnen Benutzern Rechte entziehen, neue Benutzer einladen oder andere Einstellungen vornehmen. Zum anderen hat der Verzicht auf ein zusätzliches in Eclipse integriertes Menü den Vorteil, dass Änderungen nur an einer Komponente – dem Web-Frontend – vorgenommen werden müssen. So können Fehler schneller und zuverlässiger behoben und Änderungen implementiert werden, da die Arbeiten nur an einem Modul vorgenommen werden müssen.

Die Umsetzung wurde in Struts 1.3 ausgeführt. Hierbei dient Struts als Frontcontroller, um auf die Funktionalitäten des Servers zuzugreifen. Dabei kann es mit Hilfe seines *Validation Frameworks* falsche Eingaben im Vorfeld abfangen. Beispielsweise werden die Nutzerdaten auf Korrektheit validiert, um zu verhindern, dass eine syntaktisch falsche Email-Adresse oder kein Passwort eingegeben wurden. Durch den direkten Zugriff auf den Dawn-Server, werden sämtliche Aktionen, die in der Web-UI ausgeführt werden, sofort an den Server übertragen und gegebenenfalls von den Clients übernommen. Ändern sich die Rechte für einen Nutzer so hat diese Änderung sofortige Auswirkung. Abbildung 61 zeigt eine Ansicht des Verwaltungs-GUI im Browser.

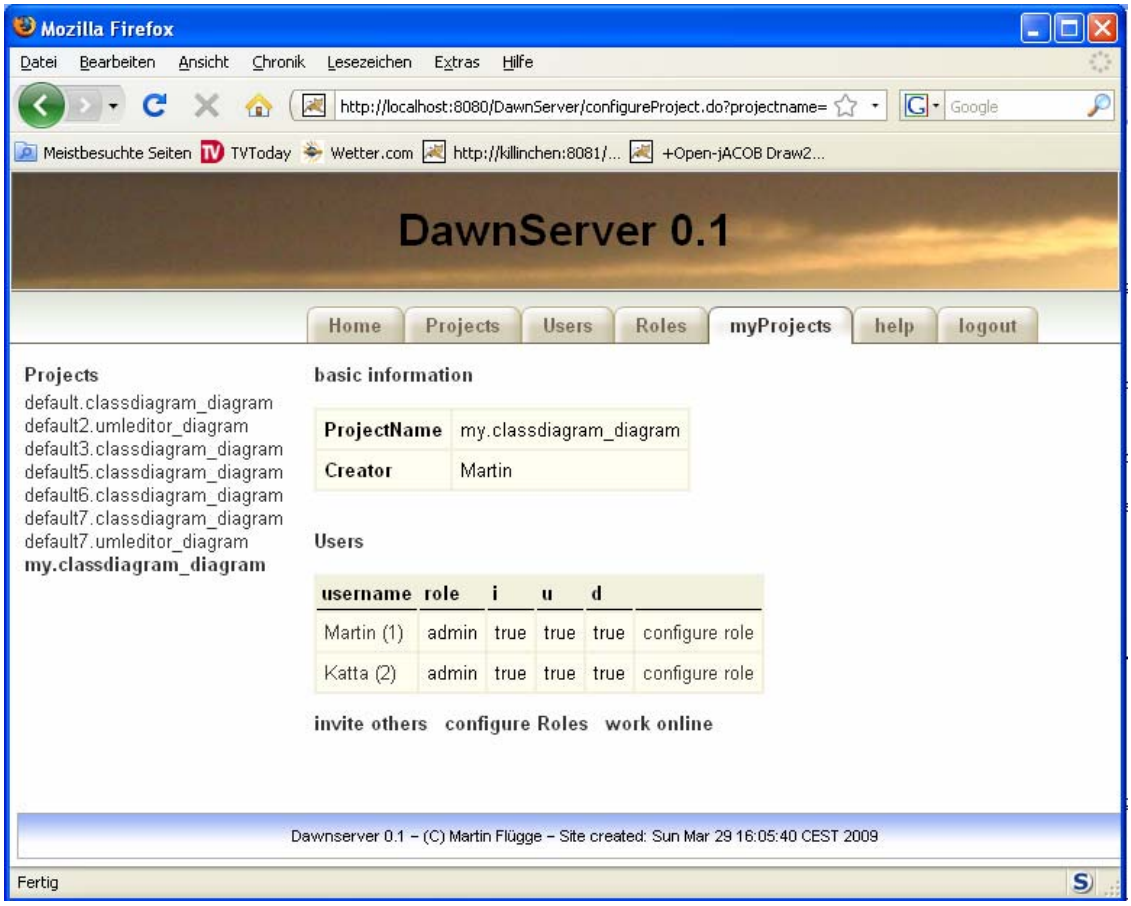

Abbildung 61 - Web-UI

Für einen Benutzer, der gerade in der Eclipse-IDE ein Projekt angelegt hat, wäre es aber umständlich in einen Browser zu wechseln, sich einzuloggen, das eben erstellte Projekt auszuwählen und dort die Änderungen vorzunehmen.

Aus diesem Grund wurde eine Möglichkeit geschaffen auch innerhalb von Eclipse auf das Web-Frontend zuzugreifen. Hierbei wird der in Eclipse integrierte Browser genutzt. Um einen komfortablen Zugang zu den Projekteinstellungen zu gewährleisten, wurde ein zusätzliches Kontext-Menü erstellt, welches direkt mit den einzelnen Projekt-Dateien verknüpft ist [vgl. Abbildung 62]. Eclipse bietet für derartige Erweiterungen den Extension Point **`org.eclipse.ui.popUpMenus`** an. Dieser lässt sich soweit konfigurieren, dass bestimmte Menü-Einträge nur auf Dateien sichtbar sind, die einem bestimmten Ausdruck entsprechen. Um nur auf den ausgewählten Projekt-Dateien sichtbar zu sein, wurde der Parameter **`nameFilter`** des Extension Points auf den Wert „*_diagram“ gesetzt.

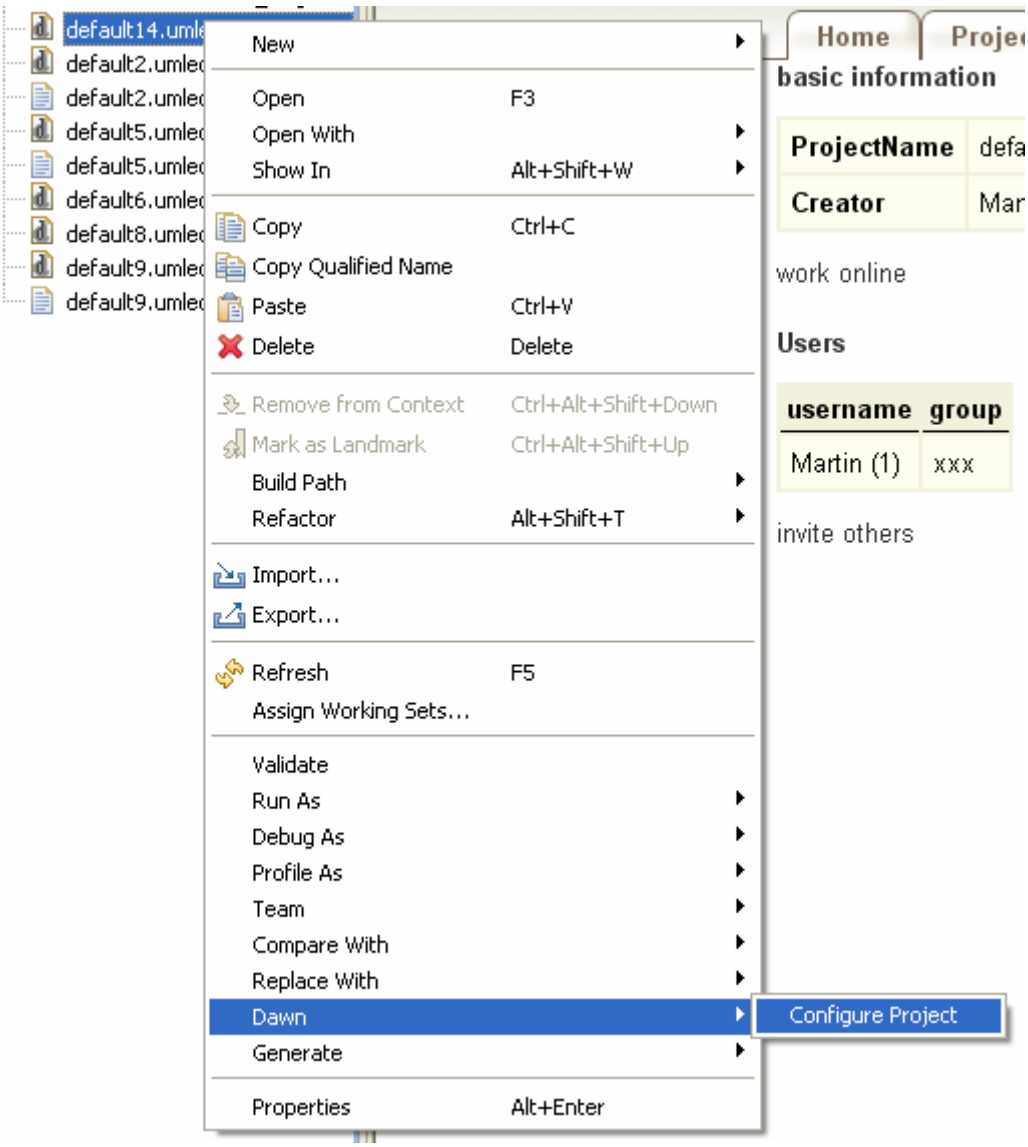

Abbildung 62 - Kontext-Menü auf einer Projekt Datei

Damit der Benutzer direkt mit der Projekt-Konfigurationsseite verbunden wird, übermittelt das System den Benutzernamen und den Projektnamen des gewählten Projektes. Der Server wertet diese Informationen aus und routet an die richtige Web-Seite weiter. Somit entfallen für den Benutzer das Einloggen und das Navigieren zu den Projekteinstellungen.

Um dem Nutzer, der sich bereits innerhalb der Eclipse-IDE authentifiziert hat, ein weiteres Einloggen auf dem Server zu ersparen, wurde mittels eines *Single-Sign-On* Verfahrens gewährleistet, dass der Nutzer seine Daten nicht unnütz auf dem Server eingeben muss, das System aber gegen Angriffe geschützt bleibt. In Abbildung 63 ist dargestellt, wie die Konfigurations-UI in dem Eclipse-internen Browser geöffnet ist. In [Flügge 2009e] wird die Anwendung dieses Mechanismus demonstriert.

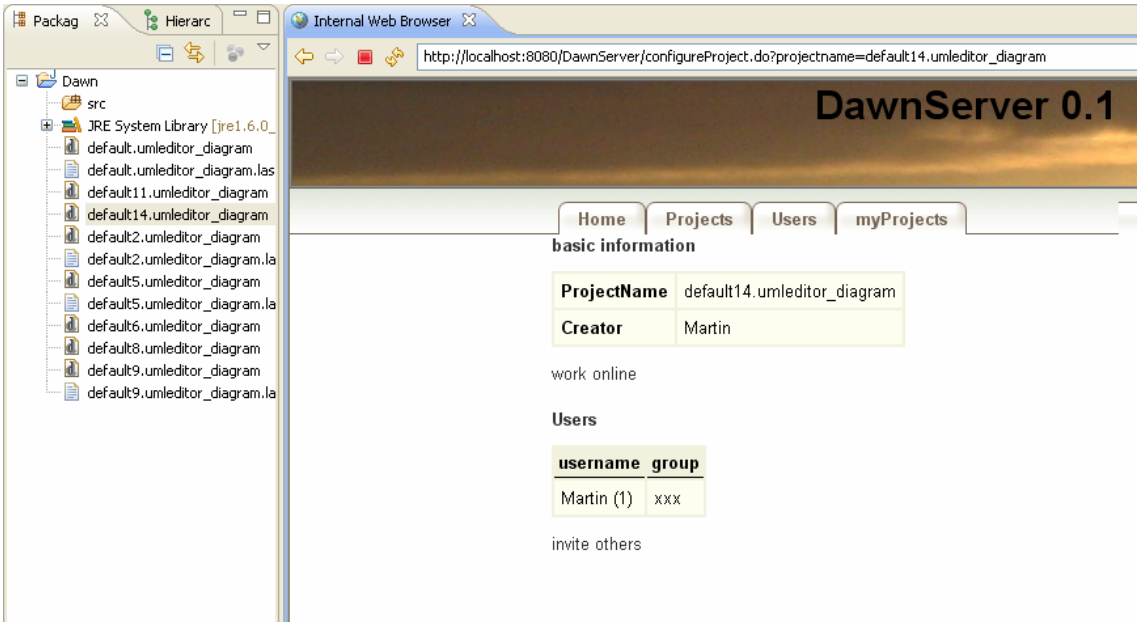

Abbildung 63 - Web-UI innerhalb von Eclipse

Zur Umsetzung des Single-Sign-On wurde ein Verfahren gewählt, welches mit Hilfe eines *Shared-Secrets* seine Identität gegenüber dem Server beweist. Bei diesem Verfahren verfügen sowohl der Client, als auch der Server über ein gemeinsames Geheimnis (Shared-Secret), welches in der Regel aus einem String besteht. Um sich nun gegenüber dem Server zu authentifizieren, generiert der Client einen Zufallswert (*Token*). Dieses Token wird an das Shared-Secret konkateniert. Nun wird über diesen neu entstandenen Wert ein Hash gebildet. Dieser Hash wird zusammen mit dem Token an den Server übermittelt. Der Server wertet die Daten aus und konkateniert seinerseits das Token mit seinem Shared-Secret. Über diesen Wert wird ebenfalls ein Hash gebildet. Nun vergleicht der Server den ermittelten Hash mit dem, den der Client gesendet hat. Stimmen beide Werte überein, so konnte der Client seine Identität beweisen, denn kein Dritter hätte diesen Hash-Wert bilden können, da er nicht im Besitz des Shared-Secrets ist.

Für die konkrete Implementierung innerhalb von Dawn wurde dieses System leicht modifiziert. Als Token wird die aktuelle Systemzeit in Millisekunden verwendet. Die Wahl eines dynamischen Wertes erschwert Angriffe. Als Shared-Secret wird das Passwort des Benutzers verwendet. Dies hat den Vorteil, dass Client und Server nicht erst zu Beginn der Sitzung über geeignete Verfahren (zum Beispiel Diffie-Hellman Austausch-Verfahren[42]) einen Wert aushandeln müssen. Außerdem ist dieses Shared-Secret benutzerabhängig. Wird also der Wert für einen Benutzer geknackt, so ist das

[42] Schlüssel-Austauschverfahren, bei dem sichergestellt ist, dass keine dritte Partei in Besitz des Schlüssel gelangen kann [vgl. Tanenbaum 2003, S. 791 ff.]

System nur für diesen Nutzer und seine Rechte geöffnet. Alle anderen Nutzer bleiben von dem Angriff unberührt.

Über den Wert aus Shared-Secret (Passwort) und Token wird nun ein MD5-Hash ermittelt. Der Client übermittelt, wenn er lokal bereits authentifiziert ist, seinen Nutzernamen, das Token und den gebildeten Hash-Wert. Der Server sucht anhand des Nutzernamens das Passwort aus der Datenbank und konkateniert es an das übermittelte Token. Über diesen Wert kann er nun einen Hash bilden und ihn mit dem Hash des Clients vergleichen. Stimmen beide überein, so leitet der Server automatisch auf die angewählte Web-Seite weiter und registriert den Benutzer innerhalb der Server-Session. Sowohl auf dem Server als auch auf dem Client wird das Passwort verschlüsselt und nicht im Klartext benutzt. Dies liegt daran, dass der Server nur über die verschlüsselte Version verfügt. Ein Angreifer, dem es gelänge das Shared-Secret zu berechnen, würde trotzdem nicht das Passwort des Nutzers erhalten.

Listing 17 stellt eine beispielhafte Anfrage an den Server dar. Hierbei werden der Projektname, der Nutzername, das Token (aktuelle Systemzeit des Clients) und der Hash-Wert (Hash über Passwort + Token) an den Server übermittelt.

```
1  http://localhost:8080/DawnServer/configureProject.do?projectname=default15.um
2  leditor_diagram&username=Martin&token=1230663014218&hash=f4fe8a77768
3  4615f94b398ddf9311c0d3574b38
4
```

Listing 17 - Beispiel-Anfrage mittels SSO

5.8 Umsetzung des Rechte-Konzepts

Clientseitig fragt die Dawn-Runtime die Nutzer-Rechte vom Server ab. Hierzu muss der Nutzer sich authentifizieren. Dies geschieht, sobald der Editor gestartet wird und eine Verbindung zum Server hergestellt werden kann. Um zu verhindern, dass Änderungen am Diagramm vorgenommen und publiziert werden können, ohne sich authentifiziert zu haben, wird als erstes ein Login-Dialog angezeigt, der das Diagramm blockiert. Erst wenn der Nutzer authentifiziert und für das Diagramm autorisiert wurde, kann er Änderungen vornehmen. Sind die Nutzerinformationen validiert, werden sie in einer lokalen Session (`LocalSession`) hinterlegt. Die `LocalSession` ist als Singleton implementiert und kann beliebige Objekte unter einem Identifikator speichern. Sie kann also neben den Nutzereinstellungen auch noch andere Informationen verwalten und dem System global zur Verfügung stellen. Mit jeder *update*-Operation werden die aktuellen Rechte, die auch dem Server für das gerade geöffnete Diagramm hinterlegt sind, zum

Client übertragen. Haben sich Änderungen ergeben, so werden diese sofort umgesetzt. Wird dem Nutzer beispielsweise das Recht auf Änderungen entzogen, so wird das Diagramm gesperrt und kann nur noch betrachtet werden.

5.9 Dawn-Codegen

Für die Erstellung des Dawn-Quellcodes wurde ein eigenes Plugin erstellt, welches alle Methoden und Funktionalitäten bereitstellt, um den Code zu generieren und zu verwalten. Dieses Plugin trägt die Bezeichnung **`org.mftech.dawn.codegen`**. Es beinhaltet auch das Meta-Modell für das Dawn-GenModel, welches für die Generierung benötigt wird [vgl. 4.11.1]. Um es zu erzeugen, wurde auf das GMF-GenModel ein zusätzlicher Menüpunkt erstellt, mit dessen Hilfe das Dawn-GenModel erzeugt werden kann. Dem Dawn-GenModel liegt ein mit Hilfe von EMF erstelltes Meta-Modell zugrunde. Dadurch war es nicht nur möglich das Meta-Modell im Rahmen der MDSD zu nutzen, sondern auch gleichzeitig über EMF einen Editor für das Modell generieren zu lassen. Nach den Konventionen von EMF entstanden zusätzlich noch zwei weitere Plugins **`org.mftech.dawn.codegen.edit`** und **`org.mftech.dawn.codegen.editor`**, welche Klassen zur Bearbeitung des Modells und einen Editor bereitstellen. Auf dem GenModel wurden unterschiedliche Menüpunkte realisiert, damit der Quellcode auch in Teilen generiert werden kann. So ist eine Erstellung des Clients unabhängig vom Server möglich. Auch wenn sich nur die grafische Darstellung im GMF-Graph-Modell geändert hat, ist es möglich einzeln die JavaScript-Views zu erstellen, ohne den Server oder den Client neu generieren zu müssen.

Für das Ausführen der Generierung stellt das Dawn-Codegen Framework verschiedene Implementierungen des Interfaces **`Creator`** bereit. Dieses Interface ist sehr simpel gehalten und bietet lediglich eine Methode an – **`void create(IProgressMonitor)`** – in der die Erzeugung des Codes realisiert wird. Da die Generierung des Quellcodes teilweise recht lange dauert, bekommt diese Methode einen Progressmonitor übergeben. Die Implementierungen der Schnittstelle **`IProgressMonitor`** werden im Eclipse genutzt, um Informationen während eines Vorgangs anzuzeigen. Sie können von keiner Anzeige (**`NullProgressMonitor`**) bis hin zu komplexen Sub-Prozessen in einem Dialog gehen. Dawn nutzt einen **`SubProgressMonitor`**, um Teilschritte in einem Fortschrittsbalken darstellen zu können. Das Creator-Konzept wurde entwickelt, damit die Erzeugung des Codes einfach und flexibel gestaltet werden kann.

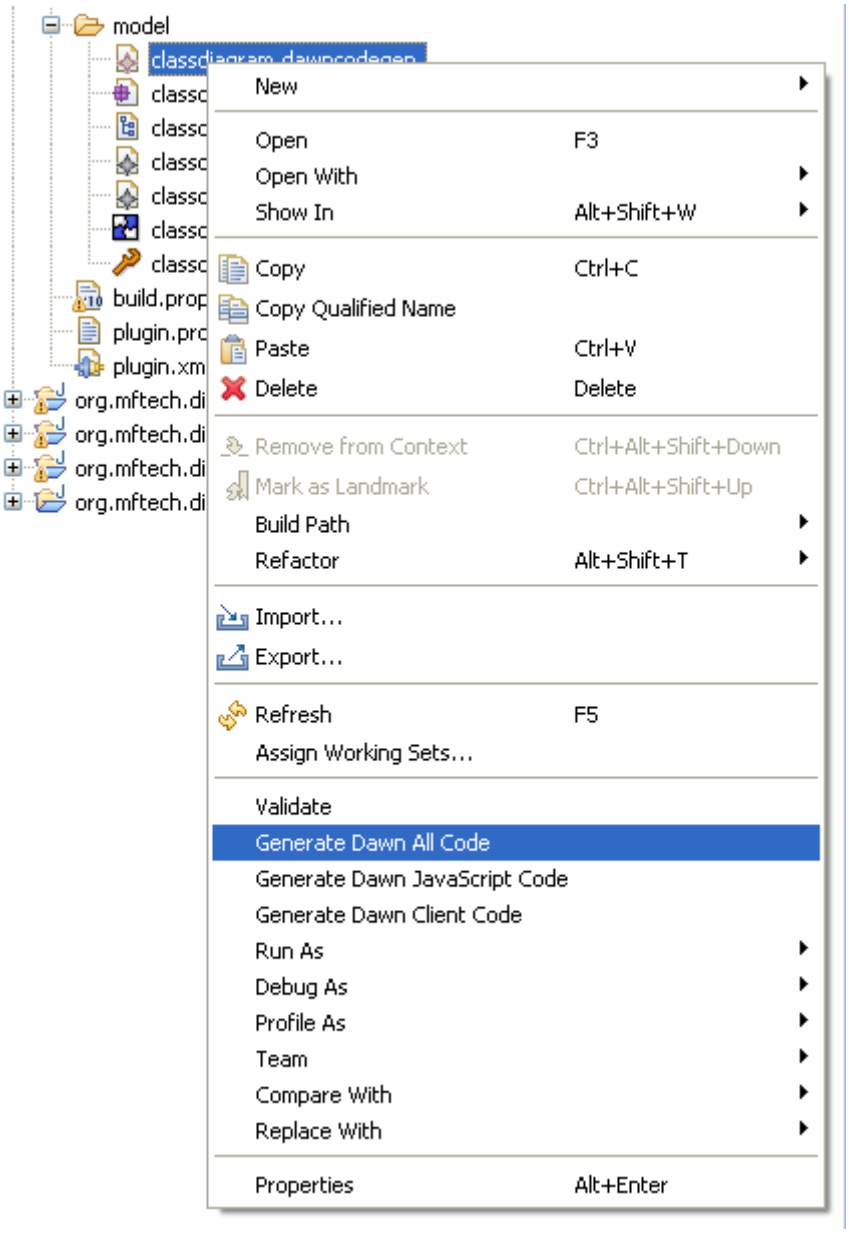

Abbildung 64 - Menü auf dem Dawn-GenModel

Über entsprechende Menüpunkte können die einzelnen Generierungsoptionen ausgewählt werden [vgl. Abbildung 64]. Dies bewirkt das Ausführen der entsprechenden Creator-Instanzen und dadurch die Generierung der gewählten Komponente. Der genaue Ablauf der Generierung kann unter [Flügge 2009a] angesehen werden.

5.9.1 Generierung des Servers

Der Creator für die Erzeugung des Server-Codes erstellt als erstes ein neues Sysdeo-Tomcat Projekt. Auch hier ist die Nutzung des Generator-Konzeptes von Vorteil, da bei

Bedarf alternative Implementierungen erstellt werden können, die den Server in eine andere Projekt-Art generieren. Um in Eclipse programmgesteuert Projekte anlegen zu können, bedarf es mehrerer Teilschritte. Zuerst muss über den Workspace von Eclipse ein Projekt erzeugt werden. Wichtig ist hierbei, dass die richtigen *Natures* an das Projekt gebunden werden. Natures beschreiben in Eclipse die Art von Projekten. Java-Projekte besitzen beispielsweise immer die *Java-Nature*. Für die Erstellung eines Tomcat Projekts müssen weitere *Naturen* eingebunden werden. Allerdings reicht das Hinzufügen der Naturen nicht aus, um aus dem Projekt ein reines Java-Projekt zu machen. Über die Klasse **`JavaCore`** muss das Projekt noch mit zusätzlichen Informationen konfiguriert werden [vgl. Listing 18, Zeile 10]. Listing 18 zeigt die programmtechnische Erstellung eines Java Projektes.

```
1   String[] natures = new String[]{ JavaCore.NATURE_ID, };
2
3   IProject project =
4   ResourcesPlugin.getWorkspace().getRoot().getProject(name);
5
6   IProjectDescription description = project.getDescription();
7                       description.setNatureIds(natures);
8                       project.setDescription(description, null);
9
10  IJavaProject javaProject = JavaCore.create(project);
```

Listing 18 - Erstellung eines Projektes in Eclipse

Neben besonderen Naturen (zum Beispiel **`com.sysdeo.eclipse.tomcat.tomcatnature`**) müssen für das Tomcat Projekt noch verschiedene Classpath-Einstellungen und Bibliothekszuordnungen getroffen werden. So benötigt es beispielsweise einen Verweis auf die Bibliotheken des Tomcat-Containers und die Java Runtime. All diese Operationen wurden in einer eigenen Komponente gekapselt, damit die Erstellung von Projekten einfacher erfolgen konnte und bestimmte Operationen wieder verwendet werden können.

Nachdem das Projekt erstellt wurde, kopiert der **`ServerCreator`** alle nötigen Klassen, Dateien und Bibliotheken in das Projekt und setzt die nötigen Verweise. Danach generiert er die JavaScript-Views basierend auf dem GMF-GenModel des Editors [vgl. 5.9.3]. Zum Schluss nutzt er die in dem Dawn-GenModel definierten Informationen, um nutzerspezifische Einstellungen zu treffen. So ergänzt er das Mapping-File für die Hibernate-Konfiguration (**`hibernate-cfg.xml`**) mit den Nutzereingaben für die ausgewählte Datenbank. Zusätzlich bindet er noch den gewählten Namen des Web-Projektes an den Server.

Durch die Nutzung des Sysdeo-Plugins ist damit der Server vollständig einsatzbereit. Er kann direkt aus Eclipse heraus gestartet werden. Das Sysdeo-Plugin übernimmt hierbei das Starten des Tomcat-Containers und das Deployen des Dawn-Servers.

5.9.2 Generierung der Client-Erweiterung

Der Client-Generator erstellt, wie der Generator des Servers, ein Java-Projekt. Allerdings werden hier die spezifischen Naturen für *Fragmente* hinzugefügt. Neben diesen Informationen wird in die fragment.xml der Name des Host-Plugins, also des GMF-Diagram Plugins, generiert. Der Name dieses Plugins wird über das GMF-GenModel ermittelt und so die Namenskonvention des GMF-Projektes eingehalten. Alle Klassen der Editor-Erweiterung werden über einen *oAW-Workflow* erzeugt. Das entsprechende Template für den Quellcode erstellt dabei nicht nur die Package-Struktur, sondern passt auch generisch die Paket- und Klassenbezeichnungen an die des Hostprojektes an [vgl. Abbildung 65].

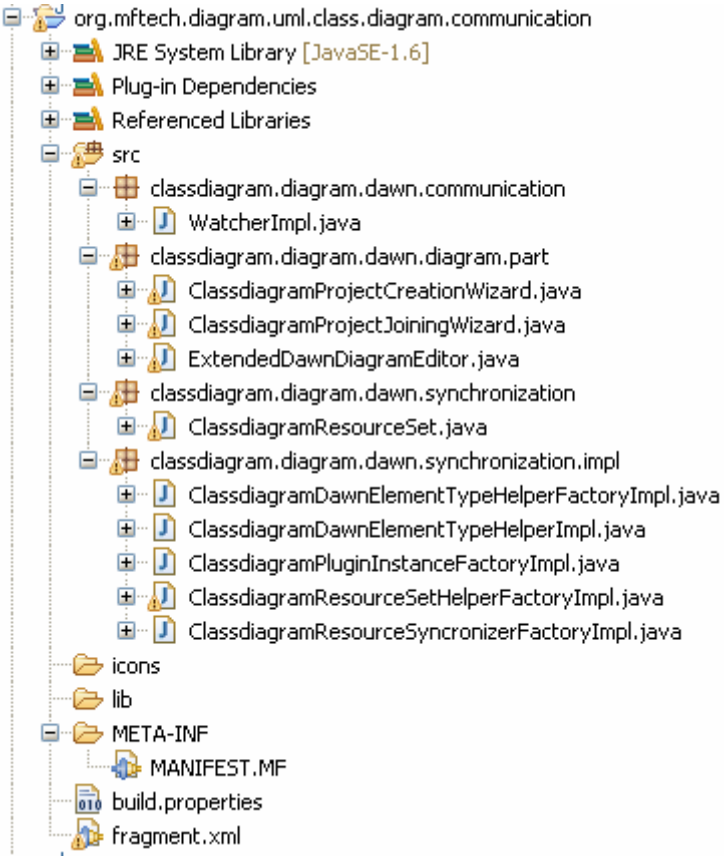

Abbildung 65 - generiertes Client Projekt

Mit die wichtigste Generierung betrifft hierbei den *ElementTypeHelper*, der die Zuordnung zwischen View und ElementType ermöglicht. Dabei greift das Template auf alle Knoten und Kanten im GMF-GenModel zu, welches die Information über die

Visual-ID eines jeden Knotens und dessen ElementType beinhaltet. Listing 19 zeigt einen Ausschnitt aus dem Template für die Generierung des ElementTypeHelpers. Der Übersicht halber sind Import-Statements, Kommentare und andere Deklarationen entfernt worden.

```
1   «DEFINE elementTypeHelperImpl FOR GenEditorGenerator»
2   «FILE
3   "src/"+getDiagramPackagePath()+"/dawn/synchronization/impl/"+model
4   ID+"DawnElementTypeHelperImpl.java"»
5
6   public class «modelID»DawnElementTypeHelperImpl implements
7   DawnElementTypeHelper
8   {
9     public IElementType getElementType(View view)
10    {
11
12     int visualId = «modelID»VisualIDRegistry.getVisualID(view);
13        switch (visualId)
14        {
15        «EXPAND elementMapping(this.diagram) FOR this.diagram»
16        «FOREACH this.diagram.childNodes AS e»
17        «EXPAND elementMapping(this.diagram) FOR e»
18        «ENDFOREACH»
19              «FOREACH this.diagram.topLevelNodes AS e»
20        «EXPAND elementMapping(this.diagram) FOR e»
21        «ENDFOREACH»
22              «FOREACH this.diagram.links AS e»
23        «EXPAND elementMapping(this.diagram) FOR e»
24        «ENDFOREACH»
25        }
26              return null;
27        }
28  }
29  «ENDFILE»
30  «ENDDEFINE»
31
32  «DEFINE elementMapping(GenDiagram d) FOR GenDiagram»
33        case «this.editPartClassName».VISUAL_ID:
34                    return
35  «d.elementTypesClassName».«getUniqueIdentifierName(this.elementTyp
36  e.uniqueIdentifier)»;
37  «ENDDEFINE»
```

Listing 19 - Ausschnitt oAW-Template für ElementTypeHelper

5.9.3 Generierung des Web-Viewers

Das Aussehen der Knoten und Kanten eines GMF-Editors kann vom GMF-Entwickler innerhalb des GMF-Graph Modells verändert werden [vgl. 2.8.7.1]. Er nutzt dabei einen baumbasierten Editor, um einzelne Elemente feingranularer zu gestalten. Zum Beispiel um zu einem Knoten Compartments und Unterknoten hinzuzufügen oder Kanten mit

bestimmten Endknoten zu verzieren. Um nun die grafische Darstellung der Elemente auch mittels JavaScript im Web-Viewer anzuzeigen, wären verschiedene Ansätze denkbar. Da innerhalb eines jeden EditParts auch die Repräsentation der dazugehörigen Figure generiert wird, könnte versucht werden einen Parser zu schreiben, der diese Informationen ausließt und den Java-Quellcode in ein entsprechendes JavaScript Pendant umzuwandelt. Diese Methode hätte zwei offensichtliche Vorteile. Zum einen könnte mit Hilfe des Reflection-API diese Generierung auch zur Laufzeit umgesetzt werden. Zum anderen würden auch alle Änderungen, die manuell an den Figuren des generierten Editors ausgeführt wurden mit übernommen. Allerdings ist der Aufwand für eine solche Implementierung viel zu hoch, da eine Unmenge von Parametern bei dem Parsen des Java-Quellcodes zu beachten sind.

Der umgesetzte Ansatz geht einen wesentlich eleganteren Weg. Alle JavaScript-Figuren werden, wie ihre Java-Verwandten, aus den Informationen des GMF-Graphen generiert. Hierbei kann, wie bei der Generierung des Servers und des Clients, ein XPand-Template und ein oAW-Workflow genutzt werden, um auf das Modell zuzugreifen und den Quellcode mittels Model-to-Text Transformation zu realisieren. Dawn kann dabei auf Änderungen im Modell reagieren und neuen Quellcode erzeugen. Da die Figuren losgelöst von dem eigentlichen Datenmodell sind, liefert das GMF-Graph-Modell demzufolge auch keine entsprechende Zuordnung. Die Controller-Komponente des Web-Viewers benötigt aber eine Zuordnung zwischen Daten-Modell und Figure in Form der config.xml [vgl. 4.8.1]. Um diese zu erstellen, muss der Generierungsprozess zusätzlich auf das GMF-GenModel zugreifen, indem diese Informationen vorhanden sind. Abbildung 66 zeigt welche Modell-Informationen für welche Komponente innerhalb der Generierung genutzt werden.

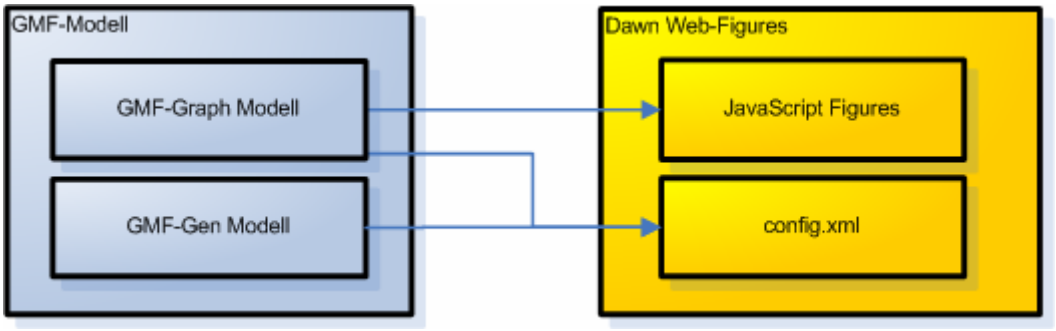

Abbildung 66 - Generierung Web-Viewer

Auf Grund der Fülle der Möglichkeiten einen GMF-Editor zu gestalten, wurden nicht alle möglichen Varianten umgesetzt. Die in Dawn aufgegriffenen Figurenbeschreibungen genügen aber, um einfache Editoren für das Web zu erstellen. Durch die einfache Handhabung des XPand-Template-Mechanismus können aber

zusätzliche Funktionen später einfach erstellt werden. Die wichtigsten Änderungen, auf die Dawn Einfluss nehmen kann, werden im Folgenden erläutert. Eine visualisierte Form des Generierungsprozesses findet sich unter [Flügge 2009b].

5.9.3.1 Knoten

Zur Erstellung rechteckiger Knoten kann innerhalb von *OpenJacob Draw2D* die Klasse **`CompartmentFigure`** genutzt werden. Diese erbt von der JavaScript-Klasse **`Node`** und kann zusätzlich Kindsobjekte verwalten, welche über entsprechende Methoden dem Objekt hinzugefügt werden können. Dies ermöglicht das dynamische Hinzufügen von Unterobjekten, wie das *Diagramm-Servlet* es nutzt. Da es aufwändig wäre diese Erweiterungen in jede zu erstellende Figure zu generieren, wurde eine Oberklasse für alle rechteckigen Knoten erstellt, von der die Figuren erben (**`org.mftech.dawn.basic.DawnNodeFigure`**). Dadurch kann das Generat klein und übersichtlich gehalten werden. Listing 20 stellt eine generierte Figure für einen rechteckigen Knoten dar.

```
1   org.mftech.diagram.uml.clazz.AClassFigure = function(className)
2   {
3         org.mftech.dawn.basic.DawnNodeFigure.call(this);
4         this.outputPort1 = null;
5         this.outputPort2 = null;
6         this.setDimension(50, 50);
7         this.setResizeable(false);
8         this.setClassName(className);
9   }
10  org.mftech.diagram.uml.clazz.AClassFigure.prototype = new
11  org.mftech.dawn.basic.DawnNodeFigure;
12
13  org.mftech.diagram.uml.clazz.AClassFigure.prototype.type =
14  "org.mftech.diagram.uml.clazz.AClassFigure";
15  org.mftech.diagram.uml.clazz.AClassFigure.prototype.
16  createHTMLElement = function()
17  {
18   var item =   org.mftech.dawn.basic.DawnNodeFigure.prototype.
19   createHTMLElement.call(this);
20   item.style.border = "1px solid #B0B0B0";
21   item.style.backgroundColor = "rgb(230,230,255)";
22   return item;
23  }
```

Listing 20 - Generat - AClassFigure

GMF ist aber nicht nur auf rechteckige Knoten beschränkt. Über den Figure-Deskriptor **`RoundedRectangle`** können Knoten erzeugt werden, bei denen die Kanten abgerundet sind. Dabei kann GMF über die Parameter **`CornerHeight`** und **`CornerWidth`** den Grad

der Rundung bestimmen. Bei der Darstellung von Knoten nutzt OpenJacob Draw2D allerdings HTML-Tabellen. Das Anzeigen von Tabellen mit runden Kanten ist derzeit mit standardisierten Mechanismen allerdings schwierig zu realisieren. Gängige Methoden versuchen mit Hilfe von transparenten Grafiken die Rundungen am Rahmen einer Tabelle zu simulieren. Dieses Vorgehen ist aber besonders bei dynamisch größenveränderlichen Objekten sehr kompliziert. Dawn setzt daher auf *CSS3* (*Cascading Style Sheets Version 3*). Dort sieht die Spezifikation vor, dass auch Kanten mit runden Ecken versehen werden können. Da aber CSS3 noch kein offizieller Standard ist, können nicht alle Browser ihn darstellen. Mozilla-basierte Browser, wie der Firefox bieten aber ein spezielles Style-Sheet (**`-moz-border-radius`**), mit dem sich runde Tabellenecken darstellen lassen. Der Safari-Browser kann ebenfalls runde Ecken darstellen, nutzt aber eine eigene Implementierung[43]. Dawn unterstützt die Variante von Mozilla.

Um Knoten mit runden Ecken zu erzeugen, wurde ebenfalls eine Basisklasse erzeugt, welche einen runden Knoten darstellen kann. Das XPand-Template prüft, ob der Knoten normal oder rundkantig ist, indem er die Klasse des Modells ausliest. Danach entscheidet es, von welcher Klasse der Knoten erben soll. Ist die Basisklasse des Figure-Deskriptors **`RoundedRectangle`**, so parametrisiert er zusätzlich noch die Klassen mit dem Wert für die Kantenbiegung.

5.9.3.2 Kanten

Dawn stellt für generierte Kanten die Basis-Klasse **`DawnConnectionFigure`** zur Verfügung. Kantendefinitionen bestehen aus zwei Teilen, der eigentlichen Kante und Dekoratoren, welche das Aussehen der Kantenenden definieren. Hierbei besteht das Problem, dass Dekoratoren mit Hilfe von Template Points definiert werden können, um ihnen ein spezielles Aussehen zu verleihen [vgl. 2.8.7.3]. Leider stimmt die Koordinatenbeschreibung innerhalb von GMF nicht mit den Zeichen-Methoden der Draw2D API von OpenJacob überein. Infolgedessen können die Koordinaten nicht einfach übernommen werden. Um die Dekoratoren trotzdem anhand der Beschreibung im GMF-Graph-Modell für den Web-Viewer generieren zu können, wurde empirisch eine Transformationsfunktion ermittelt, mit der eine Transformation der TemplatePoints von dem einen in das andere Koordinatensystem erfolgen kann [vgl. Abbildung 67].

[43] http://www.css3.info/border-radius-apple-vs-mozilla/

$$f(x,y) = \begin{pmatrix} -5x \\ 3y \end{pmatrix}$$

Abbildung 67 - Transformationsfunktion für Template Points

Zusätzlich zu dem Aussehen der Dekoratoren kann Dawn zusätzlich sowohl Einfluss auf die Kantenstärke als auch die Hintergrundfarbe im Fall einer *PolyGon Decoration* nehmen. Der Generator erkennt anhand des Modells welche Dekoratoren zu welchen Kanten zugeordnet sind und setzt die entsprechenden Beziehungen. Alle JavaScript-Dekoratoren erben von der Basisklasse **`DawnBasicConnectionDecorator`**.

6 Tests

6.1 Funktionale Testfälle

Alle in der Anforderungsdefinition aufgestellten Anforderungen an das System wurden durch manuelle Testfälle bestätigt. Zusätzlich wurden die einzelnen Komponenten des Systems mit Hilfe von automatischen Tests verifiziert.

6.2 Komponentenorientierte Tests

Um schon während der Entwicklung umgehend Fehler erkennen zu können, wurde das Verfahren des Unit-Testings verwendet. Dabei werden für einzelne Komponenten Tests entwickelt, welche die Funktionalitäten dieser Komponenten überprüfen. In der Regel werden Unit Tests am Anfang der Entwicklung einer Komponente erstellt und dann zusammen mit dieser weiterentwickelt. Dies ermöglicht nach jeder Änderung am System sofort eine Reihe von Test auszuführen, um zu prüfen, ob das System nach diesen Änderungen noch im definierten Rahmen agiert.

Für Java gibt es ein weit verbreitetes Framework, mit dessen Hilfe Unit Tests einfach durchgeführt werden können. Dieses Framework nennt sich *JUnit*[44]. In JUnit werden Testfälle erstellt, indem eine eigene Implementierung von der Klasse `TestCase` erbt. Alle Methoden innerhalb dieser Klasse, die mit dem Präfix *test* beginnen, werden automatisch vom Framework ausgeführt. In den Methoden können die Komponenten mit Werten instanziiert und danach auf Korrektheit geprüft werden. Hierzu können Vergleichsmethoden genutzt werden, um auf erwartete Ereignisse zu prüfen. Listing 21 zeigt einen einfachen Testfall, bei dem über eine `RoleFactory` eine Rolle erzeugt und im Anschluss geprüft wird, ob die Rolle auch die erwarteten Rechte besitzt. Schlägt eine der Überprüfungen fehl, ist auch der Test fehlgeschlagen.

Testfälle können zu einer größeren Einheit, den TestSuites zusammengefasst werden. Für jede Komponente wurde eine Vielzahl von Tests entworfen, die mit gestiegenem Funktionsumfang erweitert wurden. Nach jeder Änderung wurden alle JUnit-Testfälle ausgeführt, damit sichergestellt werden kann, dass die getätigten Änderungen keine Fehler produziert haben.

[44] http://www.junit.org/

```
1   public void testAdmin()
2   {
3
4    Role role = RoleFactory.mfInstance.createRole(RoleFactory.ADMIN);
5    assertTrue(role.getName().equals("admin"));
6    assertTrue(role.isAllowed(Operations.INSERT));
7    assertTrue(role.isAllowed(Operations.UPDATE));
8    assertTrue(role.isAllowed(Operations.DELETE));
9   }
```

Listing 21 - einfacher Unit-Test

6.3 Performance Tests

Die durchgeführten Performance Tests versuchen einen Überblick über die Leistungsfähigkeit des Systems zu geben. Hierbei teilen sich die Tests in zwei Kategorien. Zum einen wurden Vergleichsmessungen durchgeführt, um Unterschiede zwischen den beiden implementierten Protokollen SOAP und RMI feststellen zu können. Andererseits wurden Skalierungstests ausgeführt, um das Verhalten des Systems unter Last beurteilen zu können.

6.3.1 Testumgebung

Für die Performance Tests wurde eine Testumgebung bestehend aus einem Server und zwei Clients aufgebaut. Hierbei wurde ein Client mit aktueller Hardware und einer mit älterer Hardware genutzt, um auch schwächere Clients zu simulieren. Dabei wurde besonders bei der Übertragung mit SOAP ein deutlich erkennbarer Leistungseinbruch auf der älteren Hardware erwartet. Die software- und hardwaretechnische Ausstattung der Clients ist Tabelle 5 zu entnehmen. Server und Client wurden über einen 10/100 MBit Switch verbunden. Auf beiden Rechnern wurde Eclipse 3.4.0 (Ganymede Release) verwendet.

System	Client 1	Client 2	Server
CPU	AMD Duron (Morgan) 1,0 GHz	Intel Core2Duo (T7250); 2,0 GHz	Intel Core2Duo (E6600); 2,4 GHz
RAM	768 MB SD-RAM	2 GB/ DDR2	3,6 GB / DDR2
Festplatte	120 GB	180 GB (RAID 0)	300 GB (RAID 0)
Netwerkkarte	10/100 MBit	100/1000 MBit	100/1000 MBit
Betriebssystem	Windows 2000 SP4	Windows XP SP 3	Windows XP SP 3

Tabelle 5 - Systemumgebung für Performance Tests

Alle Untersuchungen liefen mit dem unmodifizierten Server. Für die Ausführung des Clients wurde ein kleines Testframework integriert, welches in einer gestarteten

Eclipse-Instanz über ein eigenes Menü aufrufbar ist. Es ermöglicht *Testfälle* in Klassenform zu definieren, welche in *TestCollections* zusammengeführt werden können. Diese TestCollections können über das Menü zur Ausführung gebracht werden. Die Testergebnisse werden in formatierter Form ausgegeben.

6.3.2 Protokoll Vergleich

Die Vergleichstests basierten darauf, dass für beide Protokolle dieselben Abfolgen von Funktionen ausgeführt wurden. Um die Übertragungsgeschwindigkeit der Ressourcen zu testen, wurde ein spezielles Diagramm mit 35 Interfaces, 70 Klassen und 105 Verbindungen entworfen. Die ausgeführten Operationen für die Protokolle sind in Tabelle 6 aufgeleistet.

Operation	**Erklärung**
Update	Eine komplette Ressource wird an den Client übertragen, bestehend aus 70 Klassen, 35 Interfaces, 105 Konnektoren
Lock	5 Objekte werden gesperrt
unlock	5 Objekte werden entsperrt
User rights	Die Rechte des Nutzers werden abgefragt
getProjectNames	Alle Projektnamen werden vom Server geladen
getMyProjectNames	Alle Projektnamen des Nutzers werden vom Server geladen

Tabelle 6 - Ausgeführte Vergleichsoperationen

Zur Glättung der Testergebnisse wurden die Tests jeweils 1000-mal ausgeführt und der Durchschnittswert der einzelnen Ergebnisse ermittelt. Die Ausführungszeiten der einzelnen Operationen wurden in Millisekunden erfasst. Um die Ergebnisse nicht zu verfälschen, wurden die Tests auf beiden Clients zeitlich unabhängig voneinander ausgeführt.

Das ermittelte Ergebnis entspricht den vorher erwarteten Werten. Durch die Serialisierung der Methodenparameter und der Rückgabewerte nach XML produziert SOAP einen auffallenden Overhead gegenüber RMI. Besonders deutlich ist beim Vergleich zwischen Client 1 und Client 2 [vgl. Abbildung 68 und Abbildung 69] zu sehen, dass das Marshelling und Unmarshelling, also das Umwandeln der Transportdaten nach Java, bei SOAP einen starken Performanceeinbruch bewirkt, während sich RMI annähernd konstant verhält.

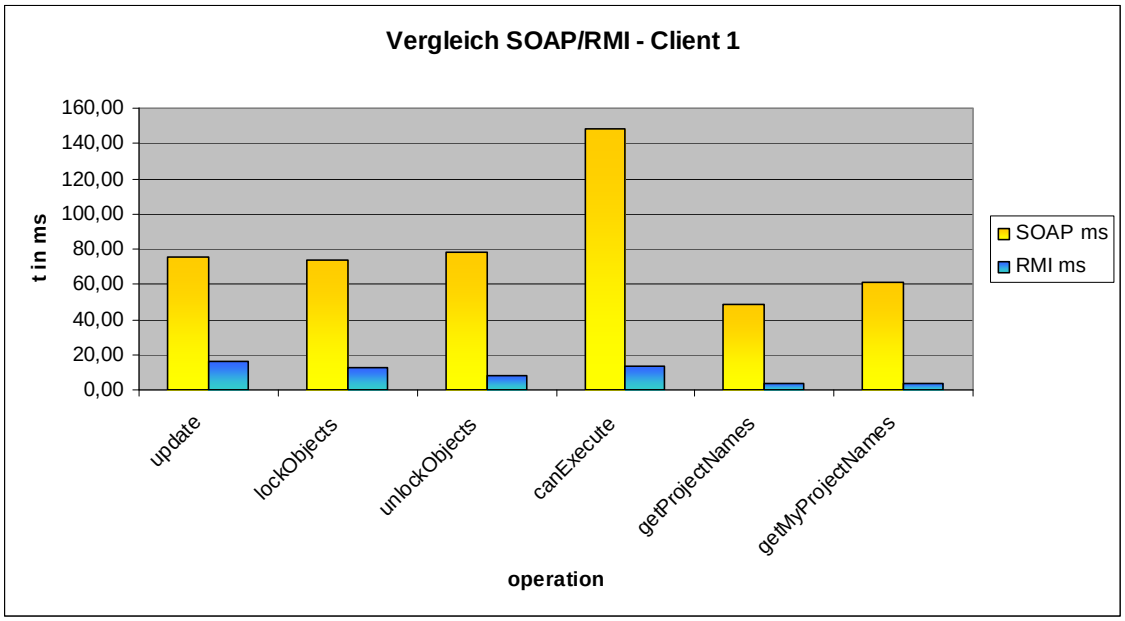

Abbildung 68 - Vergleich SOAP/RMI – Client 1

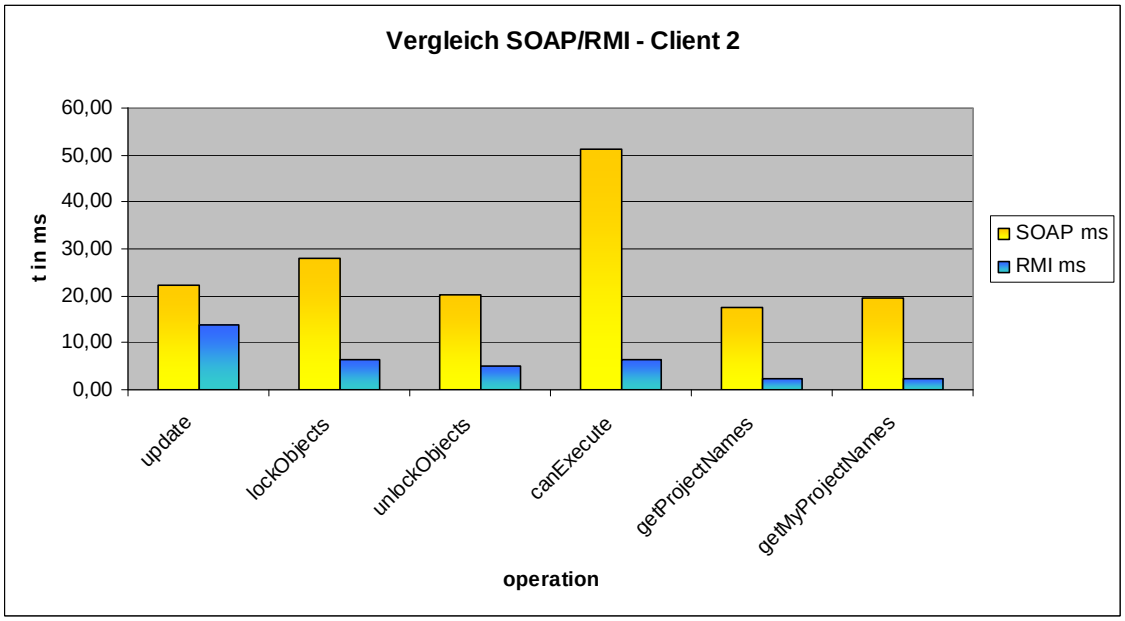

Abbildung 69 - Vergleich SOAP/RMI – Client 2

In Betrachtung dieser Ergebnisse empfiehlt sich bei Clients mit reduzierter Hardwareausstattung auf RMI zu wechseln. Allerdings muss in diesem Fall berücksichtigt werden, dass mögliche Firewall-Restriktionen die Kommunikation unterbinden können.

Die Abfragen der einzelnen Clients hatten auf den Server keinerlei Auswirkungen. Die Prozessorlast lag bei diesen Tests bei durchschnittlich 9 Prozent.

6.3.3 Skalierungstests

Zur Messung der Abarbeitungsgeschwindigkeit des Servers, wurden Tests implementiert, bei denen parallele Zugriffe auf den Server erfolgten. Dabei wurden nur Lese-Operationen betrachtet, da der Server die Schreibzugriffe synchronisiert. Jeder simulierte Nutzer arbeitete die in Tabelle 6 dargestellten Operationen abzüglich der Locking-Vorgänge ab. Es erfolgten 50 Testdurchläufe, bei denen jeweils ein zusätzlicher Client hinzugenommen wurde. Es wurde die Zeit gemessen, welche benötigt wurde, bis der letzte Client seine Aufgaben abgearbeitet hatte.

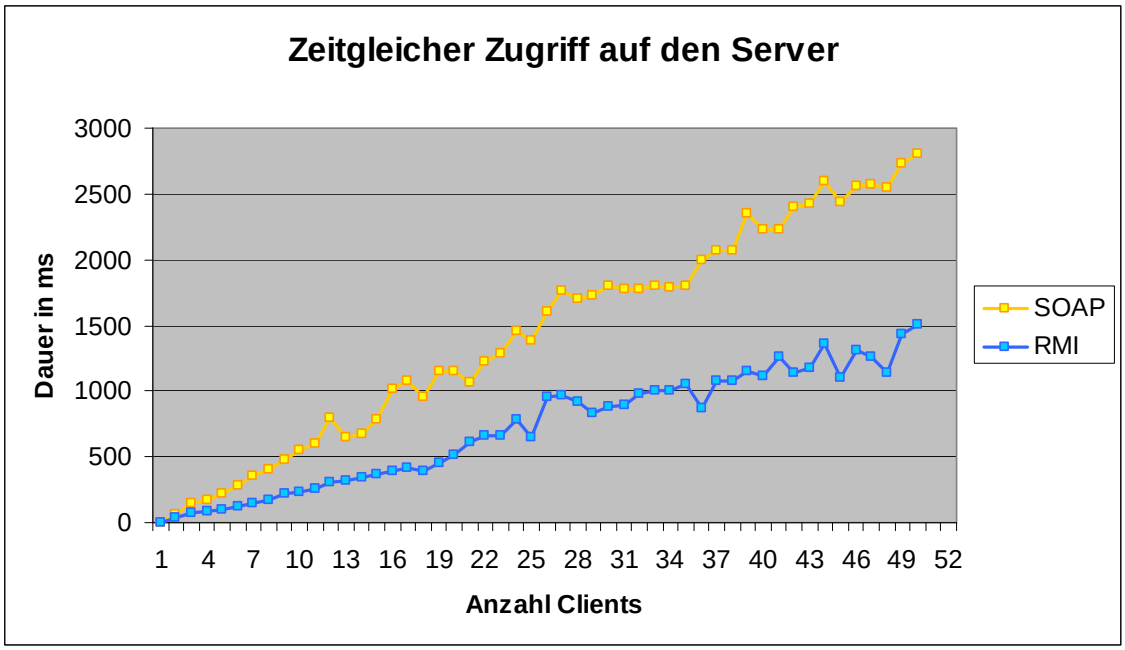

Abbildung 70 - Gesamtzeit mit steigender Client Anzahl

Im Ergebnis lässt sich ablesen, dass der Server die Aufgaben linear skaliert. Hierbei sind die Ergebnisse aber kritisch zu betrachten, da alle Threads von einem Client (Client 2) ausgeführt wurden. Die Zugriffe sind folglich nur pseudoparalleler Natur. Die Anzahl der Netzwerkanfragen an den Server konnten skaliert werden, nicht aber die Abarbeitungsgeschwindigkeit der einzelnen Prozesse, da diese maximal auf zwei Kernen der CPU gemeinsam laufen konnten. Des Weiteren zeigen diese Ergebnisse nur das Serververhalten unter direkter Last. Es wurde der zeitgleiche, parallele Zugriff aller Threads getestet. Dieses Verhalten würde in einer realen Umgebung nur in Stoßzeiten auftreten, wenn alle Nutzer gleichzeitig Anfragen an den Server stellen.

7 Zusammenfassung

Die folgenden Kapitel beleuchten die Ergebnisse der Arbeit sowohl in konzeptueller als auch implementierungstechnischer Hinsicht und fassen diese zusammen. Dabei sollen die einzelnen Ansätze auch kritisch betrachtet werden. Zusätzlich werden die eigenen Erfahrungen, die während der Arbeit mit den Eclipse-Frameworks und den anderen in dieser Arbeit benutzten Projekten gemacht wurden, thematisiert. Abschließend wird ein kleiner Ausblick auf die zukünftigen Erweiterungen und Entwicklungsmöglichkeiten des Dawn-Projektes gegeben.

7.1 Ergebnisbetrachtung

Innerhalb der Arbeit wurde eine lose gekoppelte, kollaborative Erweiterung für GMF-Editoren entwickelt. Dabei ist es gelungen eine GMF-erweiternde Architektur aufzubauen, mit denen GMF-Editoren kollaborativ genutzt werden können. Durch die generischen Kommunikationsadapter kann das System flexibel die zugrunde liegende Kommunikationsstruktur einstellen. Dadurch wird es ermöglicht, mit Hilfe von webbasierten Technologien, Firewall-Restriktionen zu umgehen, weshalb das System von nahezu jedem Ort nutzbar ist. Mittels der Web-Viewer Komponente können auch Nutzer an der Gestaltung von Diagrammen teilhaben, die keine Eclipse-IDE auf ihrem Gerät installiert haben. Durch die Erkennung von Konflikten und das Sperren von Diagrammbereichen kann dabei die Konsistenz der Daten gewährleistet werden.

Das Konzept der austauschbaren Netzwerkadapter bietet dem Nutzer die Möglichkeit das Übertragungsprotokoll an seine Bedürfnisse anzupassen. Daneben ist es einfach möglich eigene Adapter zu implementieren, um beispielsweise die Performance zu erhöhen. Auf Grund der Entscheidung für einen webbasierten Ansatz ist das System flexibel einsetzbar. Allerdings ist dabei zu beachten, dass bei sehr großen Modellen (mehrere Gigabyte) die Performance erheblich leiden kann. Für solche Modelle wäre es sinnvoller auf eine Technologie zu setzen, welche auf die hochperformante Übertragung von EMF-Modellen spezialisiert ist. Allerdings ist im Rahmen der grafischen Modellierung eher weniger mit einer derartigen Modellgröße zu rechnen, da solche grafischen Modelle bei Weitem die Grenze der Unübersichtlichkeit überschreiten würden.

Die Verwendung von bereits in GMF-Editoren genutzten Ressourcen, als Transportmedium für die Übermittlung der Modelldaten, vereinfachte die Entwicklung hinsichtlich der Serialisierung der Daten. Diese wird direkt vom GMF-Framework

übernommen und musste nicht zusätzlich implementiert werden. Dadurch liegen die Daten in exakt derselben Form, wie auf dem Client vor. Da GMF nach XMI serialisiert, ist zusätzlich ein interoperables Austauschformat verfügbar, welches in späteren Entwicklungen auch von anderen Produkten genutzt werden kann.

Die implementierten Konfliktbehandlungs- und Locking-Mechanismen erlauben eine Bearbeitung des Modellbestandes ohne die Gefahr von ungewollten Änderungen. Dabei werden Konflikte auch von Clients berücksichtigt, die längere Zeit ohne Server-Verbindung gearbeitet haben. Durch das Nutzen einer Offline-Server Implementierung ist der Wegfall der Netzwerkverbindung für den Nutzer nahezu transparent, was es dem System erlaubt, gelockte Objekte und Nutzerrechte in beschränktem Umfang auch getrennt vom Server nutzen zu können. Die Konflikterkennungsmechanismen hätten auch vollkommen serverseitig implementiert werden können. Die Auslagerung auf den Client ermöglicht aber das System mit möglichen Eigenschaften auszustatten, die anfangs gar nicht geplant waren [vgl. hierzu 7.3].

Das Dawn zugrunde liegende Rechtesystem wurde so generisch gestaltet, dass es mit einfachen Mitteln erweitert oder sogar ganz abgeschaltet werden kann, ohne die Funktionen des Systems zu beeinträchtigen. Bei Bedarf kann eine Detaillierung der einzelnen Rechte vorgenommen werden. So könnte das System derart erweitert werden, dass Administratoren die Möglichkeit haben Aktionen wie Einfügen, Ändern oder Löschen auch explizit auf einzelne View-Typen des Editors festzulegen. Die Grundlagen für derartige Erweiterungen sind im Rahmen der Arbeit gelegt worden.

Ferner ist es gelungen einen generischen Ansatz zu erschaffen, der Änderungen am bereits existierenden GMF-Editor verhindert und so jederzeit eine Trennung zwischen Dawn und GMF ermöglicht. Bereits über Dawn verbundene Diagramme können ohne weiteres Zutun wieder zu normalen GMF-Editoren umgewandelt werden. Hierzu genügt es das Plugin mit der Netzwerkerweiterung zu deaktivieren.

Durch die Nutzung von modellgetriebenen Techniken zur Bereitstellung der editorspezifischen Erweiterungen integriert sich das Produkt in den Entwicklungszyklus von GMF-Editoren. Entwickler müssen also nicht umdenken und können einfach die Erweiterungen generieren. Dadurch, dass Dawn keine Änderungen am Editor vornimmt, können auch bereits existierende Editoren erweitert werden.

Allerdings kann in dem begrenzten Zeitrahmen nicht jedes Problem für ein komplexes, verteiltes System bis ins kleinste Detail analysiert und behoben werden. Dies war auch nicht der Anspruch an diese Arbeit, sondern vielmehr das Aufzeigen von verschiedensten Möglichkeiten, welche zu weiteren Entwicklungen anregen sollen. Ziel

war es Grundlagen zu legen, von denen aus weitere Entwicklungen erfolgen können. Aus diesem Grund können auch noch nicht beliebige Editoren angebunden werden. Bestimmte Komplexitätsgrade kann Dawn derzeit noch nicht nahtlos integrieren.

Auch der entwickelte Web-Viewer kann in der aktuellen Implementierung nur eine Teilmenge der in GMF möglichen Diagramme darstellen. Durch das einfache Konzept der XPand-Templates kann der Generator für die Figuren allerdings einfach erweitert werden. Die Basis für den Ausbau des Systems, auf mit GMF erstellbare Figuren, ist somit geschaffen. Dabei bietet der Web-Viewer eine alternative Ansicht für alle Nutzer, welche die Eclipse-Umgebung nicht installieren wollen oder können.

7.2 Schlussfolgerungen und Erfahrungen

Neben der Implementierung des Systems war es Ziel der Arbeit einen Einblick in die Welt der Eclipse-Entwicklung zu bekommen. Wenn auch anfangs das Bewusstsein vorhanden war, dass Eclipse mehr als eine Entwicklungsumgebung ist, so wurden die Erwartungen bei Weitem übertroffen. Die zahlreichen Eclipse-Projekte decken einen umfangreichen Bereich von softwaretechnischen Lösungen ab. Dabei ist nicht nur die Anwendung der Frameworks auf Nutzerebene interessant, sondern ebenfalls die dahinter liegende Softwarearchitektur.

Allerdings ist es im Eclipse-Umfeld besonders bei neueren Projekten oft schwer gezielte Informationen zu erhalten. So ist es besonders im Bereich von GMF kaum möglich auf zusammenhängende Printmedien zurückzugreifen. Dies zwingt zur Nutzung von nicht immer aktuellen Internet-Quellen, was die Einarbeitung in die Thematik schwieriger als erwartet gestaltete. Die dabei besten Quellen für Probleme und Lösungen sind die zahlreichen Eclipse-Newsgroups, bei denen schnell Lösungsansätze für bestehende Probleme bereitgestellt werden.

Der Einsatz und der Umgang mit der modellgetriebenen Softwareentwicklung haben sich als höchst interessant und praktikabel herausgestellt. Besonders interessant war es zu erfahren, wie weit die Entwicklungen auf diesem Gebiet bereits sind. Frameworks, wie openArchitectureWare bieten bereits eine breite Basis an Software an, mit der modellgetriebene Softwareprozesse bearbeitet werden können. Wenngleich auch die MDSD noch nicht bis in alle Zweige der Softwareentwicklung vorgestoßen ist, wurde ihr Potential bereits von verschiedenen Firmen erkannt. Nach der Meinung des Autors wird sich dieser Prozess in Zukunft noch fortsetzten und die modellgetriebene Softwareentwicklung eine wichtige Rolle in der Informatik spielen.

7.3 Zukünftige Erweiterungen

Wie bereits beschrieben, bilden das Konzept und die Implementierungen dieser Arbeit die Basis für zukünftige Entwicklungen. Die wohl offensichtlichste Erweiterungsmöglichkeit besteht in der Ausweitung des Web-Viewers zu einem funktionierenden Web-Editor. Hierbei könnten mit Hilfe der Ajax-Technologie die auf dem Client getätigten Änderungen an den Server übertragen werden. Die Konflikt- und Locking-Mechanismen in den Web-Client zu integrieren, dürfte dabei ein höchst interessantes und spannendes Projekt sein. Dieser Ansatz könnte soweit entwickelt werden, dass die Nutzung von Eclipse innerhalb eines Dawn-Projektes optional wird.

Neben dem Web-Viewer bieten aber auch die anderen Komponenten ein umfangreiches Potential an kreativen Erweiterungsmöglichkeiten. So könnte auf dem Server eine Historie eingeführt werden, mit der alle Projektänderungen protokolliert werden. Diese könnten dann genutzt werden, um Projektstände zurückzusetzen. Natürlich würden für derartige Aktionen entsprechende Operationen in das Rechte-System integriert werden müssen. Diese Art der Versionskontrolle könnte vom einfachen Speichern jeder Ressource bis hin zur komplexen Speicherung der Unterschiede zwischen einzelnen Versionsschritten führen.

In der Arbeit wurden der Einfachheit halber die EMF-Ressource und die GMF-Ressource in einer Datei verwaltet. Diese Anforderung an die Speicherung besteht momentan auch an alle GMF-Editoren, die das System nutzen wollen. In zukünftigen Versionen wird diese Beschränkung aufgehoben werden. Dies wäre der erste Schritt, um nicht nur GMF-Editoren, sondern auch EMF-Editoren über Dawn miteinander verbinden zu können.

Mit Dawn ist es möglich, dass ein Nutzer unabhängig vom Netzwerk arbeiten kann. Möchten aber mehrere Nutzer getrennt vom Server arbeiten, ist dies noch nicht durchführbar. Interessant sind zum Beispiel Anwendungsszenarien, in denen mehrere Entwickler sich zusammen an einem Ort befinden, der keinen Netzzugang bietet. In diesem Fall könnte Dawn um geeignete Technologien erweitert werden. Durch die bereits implementierte clientseitige Konflikterkennung ist es dabei denkbar, dass das System ohne Netzwerk-Verbindung in einen *Peer-to-Peer* Modus wechselt, in welchem die Clients sich untereinander abgleichen. Es ist ebenso vorstellbar, dass einer der Offline-Server die lokale Kontrolle übernimmt und solange im System als Server agiert, bis der eigentliche Server wieder erreichbar ist. Ebenfalls, bedingt durch die Design-Entscheidung, dass der Client die Ressourcen auf Konflikte überprüft und zusammenführt, könnte Dawn um einen Import erweitert werden, welcher automatisch

Konflikte erkennt und darstellt. Dadurch könnten auch unabhängig vom Server neue Informationen in das System einfließen. Hierbei ist sogar der Import aus anderen Programmen oder Datenformaten denkbar.

Durch das Anbieten von offenen, textbasierten Schnittstellen wie SOAP können nicht nur Eclipse-Clients das System nutzen. Da die Kommunikationsschnittstelle bewusst sehr einfach gehalten wurde, ist es denkbar, dass auch andere Implementierungen auf diese standardisierten Schnittstellen zugreifen. Diese Entwicklung wird dadurch begünstigt, dass die Ressource als XMI-Text übertragen wird. Da XMI ein standardisiertes Austauschformat ist, könnte dieser Gedanke soweit getragen werden, dass andere IDEs, auch abseits der Java-Welt, den Dawn-Server als Kommunikationsschnittstelle nutzen, um mit anderen Clients zu kommunizieren. Dadurch könnte eine heterogene IDE-Struktur aufgebaut werden, in der unterschiedliche Entwicklungsumgebungen denselben Modellbestand bearbeiten. Die Entwickler wären dadurch unabhängig von Plattform und Sprachen. Neben dieser stark verteilten Struktur könnte auch der SOAP-Adapter genutzt werden, um die grafischen Modelldaten als Service im Sinne einer SOA anbieten zu können. Beispielsweise wären Szenarien denkbar, in denen in GMF-Editoren Geschäftsprozesse modelliert werden. Diese könnten von den Verantwortlichen in Eclipse erstellt werden. Über den Dawn-Server würden diese dann anderen Diensten zur Verfügung gestellt, um sie beispielsweise für die automatisierte Abarbeitung in einen Workflow umzuwandeln.

Neben diesen Erweiterungen gibt es eine Vielzahl kleiner Änderungen, die die Arbeit mit Dawn erleichtern könnten, sei es ein Button, mit dem die Verbindung zum Server hergestellt oder auf Wunsch unterbrochen werden kann, oder andere nützliche Hilfsmittel. Da Dawn als Open-Source Projekt weitergeführt wird, werden auch solche kleinen Ergänzungen über kurz oder lang implementiert werden.

Literaturverzeichnis

Burghardt 2004 Markus Burghardt:
Web-Services
Deutsche Universitäts-Verlag, Wiesbaden, 2004

Daum 2006 Dr. Berthold Daum:
Das Eclipse-Codebuch
dpunkt.verlag GmbH, Heidelberg, 2006

Daum 2007 Dr. Berthold Daum:
Rich-Client Entwicklung mit Eclipse 3.2 (2. Auflage)
dpunkt.verlag GmbH, Heidelberg, 2007

Daum 2007a Dr. Berthold Daum:
Java 6 programmieren mit der Java Standard Edition
Pearson Education, München, 2007

Funke 2009 Holfer Funke:
Devide et Impera: Das OSGi Framework
Eclipse Magazin 01/09, S.12-15
Software & Support Verlag, Frankfurt am Main, 2009

Gronback 2009 Richards Gronback:
Eclipse Modeling Project – A Domain-Specific Language (DSL) Toolkit
Pearson Education, Inc., Boston, 2009

Gruhn et al. 2006 Volker Gruhn, Daniel Pieper, Carsten Röttgers:
MDA®
Springer-Verlag, Berlin Heidelberg, 2006

Haiges 2006 Sven Haiges:
Neues von der Tag-und-Nachtgleiche
Eclipse Magazin 03/06, S.35-37
Software & Support Verlag, Frankfurt am Main, 2006

Haischt 2006 Daniel S. Haischt:
Das Medium
Eclipse Magazin 03/06, S.76-78
Software & Support Verlag, Frankfurt am Main, 2006

Hennig et al 2008 Manfred Hennig, Markus Seeberger:
Einführung in den „Extension-Point“ Mechanismus von Eclipse
JavaSpektrum 01/2008, S.19-24
SIGS DATACOM GmbH, Troisdorf, 2008

Henning 2003 Dr. Peter A. Henning:
Taschenbuch Multimedia
Carl Hanser Verlag, München Wien, 2003

Jung 2002 Volker Jung, H.-J. Warnecke:
Handbuch für die Telekommunikation (2. Auflage)
Alex Springer Verlag, Berlin, 2002

Künneth 2008 Thomas Künneth:
Einstieg in Eclipse 3.4 (2. aktualisierte und erweiterte Auflage)
Galileo Press, Bonn, 2008

Melzer et al. 2008 Ingo Melzer et al.:
Service-orientierte Architekturen mit Web-Services (3. Auflage)
Spektrum Akademischer Verlag, Heidelberg, 2008

Merks et al. 2009 Ed Merks, Davis Steinberg, Frank Budinsky, Marcelo Peternostro:
Eclipse Modelling Framework (2nd Editon)
Addison-Wesley Longman, Amsterdam, 2009

Oates et al. 2008 Richard Oates, Thomas Langer, Stefan Wille, Torsten Lueckow, Gerald Bachlmayer:
Spring& Hibernate: Eine praxisbezogene Einführung (2. Auflage)
Carl Hanser Verlag, München Wien, 2008

O'Driscoll 2008 Gerard O'Driscoll:
Next Generation IPTV Services and Technologies
John Wiley & Sons Ltd., Sussex/UK, 2008

Rupp et al. 2007 Chris Rupp, Stefan Queins, Barbara Zengler:
UML 2 Glasklar (3. Auflage)
Carl Hanser Verlag, München Wien, 2007

Schemberg et al. 2006 Axel Schemberg, Martin Linten:
PC-Netzwerke (3.Edition)
Galileo Press, Bonn, 2006

Schmidt 2007 Stephan Schmidt:
PHP Design Patterns (2. Auflage)
O'Reilly Verlag GmbH & Co. KG, Köln, 2007

Schönböck 2006 Johannes Schönböck:
Modellbasierte MIDP-Entwicklung mit MDA4ME
JavaSpektrum 01/2006; S.14-17
SIGS DATACOM GmbH, Troisdorf, 2006

Schumacher 2008 Edwin Schumacher:
Teamwork ohne Kompatibilitätsgrenzen
Eclipse Magazin 04/08, S.75-78
Software & Support Verlag, Frankfurt am Main, 2009

Schuster 2008 Andreas Schuster:
Komplett Effizient
Eclipse Magazin 01/08, S.43-48
Software & Support Verlag, Frankfurt am Main, 2008

Seeberger 2009 Heiko Seeberger:
OSGi in kleinen Dosen – Bundles und Life Cycle
Java Magazin 1/2009; S.26-32
Software & Support Verlag, Frankfurt am Main, 2009

Sippel et al 2008 Heiko Sippel, Michael Jastram, Jens Bendisposto:
Eclipse Rich Client Platform
Software & Support Verlag GmbH, München, 2008

Stahl et al. 2007 Thomas Stahl, Markus Völter, Sven Efftinge, Arno Haase:
Modellgetriebene Softwareentwicklung (2. Auflage)
dpunkt.verlag GmbH, Heidelberg, 2007

Stepper 2009 Eike Stepper:
Modelling goes Enterprise
Eclipse Magazin 03/09, S.38-42
Software & Support Verlag, Frankfurt am Main, 2009

Szypersky et al. 2002 Clemens Szypersky, Dominik Gruntz, Stephan Murer:
Component Software, Beyond Object Oriented Programming (2nd Edition)
Person Education, New York, 2002

Tanenbaum 2001 Andrew S. Tanenbaum:
Modern Operating Systems (2nd Edition, International Version)
Prentice Hall Inc., New Jersey, 2001

Tanenbaum 2003 Andrew S. Tanenbaum:
Computer Networks (International Edition)
Pearson Education Inc., New Jersey, 2003

Tanenbaum et al. 2003 Andrew S. Tanenbaum, Maarten van Steen:
Verteilte Systeme – Grundlagen und Paradigmen
Pearson Education Deutschland GmbH, München, 2003

Teufel 2008 Mark Teufel:
Odysee im Weltraum?
Eclipse Magazin 04/08, S.13-16
Software & Support Verlag, Frankfurt am Main, 2008

Wenz 2008 Christina Wenz:
JavaScript und Ajax (8. Auflage)
Galileo Press, Bonn, 2008

Widder 2004 Oliver Widder:
Adapter in Eclipse
Eclipse Magazin 01/04, S.70-80
Software & Support Verlag, Frankfurt am Main, 2004

Wunderlich 2006 Lars Wunderlich:
Calisto in Mind
Eclipse Magazin 04/06, S.21-23
Software & Support Verlag, Frankfurt am Main, 2006

Wütherich et al. 2008 Gerd Wütherich, Nils Hartmann, Bernd Kolb, Matthias Lübken:
Die OSGi Service Platform
dpunkt.verlag GmbH, Heidelberg, 2008

Zeitner et al. 2008 Alfred Zeitner, Birgit Linner, Martin Maier:
Spring 2.5: Eine pragmatische Einführung
Pearson Education, München, 2008

Internetquellen

Alle Internetquellen sind in dem zuletzt geprüften Zustand auf der beigefügten CD hinterlegt.

ComputerZeitung 2009	Susanne Franke: **OSGi-Framework bezieht weitere Java-Rahmenwerke in die Spezifikation ein** http://www.computerzeitung.de/articles/osgi-framework_bezieht_weitere_java-rahmenwerke_in_die_spezifikation_ein:/2009010/31856360_ha_CZ.html?thes=9845,10228,10231&tp=/ausrichtungen/prozesse *zuletzt geprüft: 08.05.2009*
ECF 2008	ECF: **Extending Real-Time Shared Editing for Use with Other Editors** http://wiki.eclipse.org/Extending_Real-Time_Shared_Editing_for_Use_with_Other_Editors *zuletzt geprüft: 08.05.2009*
Eclipse 2009	Eclipse Community: **GEF Programmer's Guide (Release 3.4 Ganymede)** http://help.eclipse.org/ganymede/topic/org.eclipse.gef.doc.isv/guide/guide.html *zuletzt geprüft: 08.05.2009*
Eclipse 2009a	Eclipse Foundation: **Eclipse Projects Overview** http://www.eclipse.org/projects/listofprojects.php *zuletzt geprüft: 08.05.2009*
EMFT 2008	Laurent Goubet: **[EMF Compare] Problem comparing identical GMF-resources** http://dev.eclipse.org/newslists/news.eclipse.technology.emft/msg06068.html *zuletzt geprüft: 08.05.2009*

Flügge 2009a Martin Flügge:
Screencast – Generating the source
http://www.mftech.org/dawn/screencasts/12_generating%20the%20source/generating.htm
zuletzt geprüft: 08.05.2009

Flügge 2009b Martin Flügge:
Screencast – Generating JavaScript Figures
http://www.mftech.org/dawn/screencasts/10_generationJavascript/10_generationJavascript.htm
zuletzt geprüft: 08.05.2009

Flügge 2009c Martin Flügge:
Screencast – Locking
http://www.mftech.org/dawn/screencasts/9_locking/locking.htm
zuletzt geprüft: 08.05.2009

Flügge 2009d Martin Flügge:
Screencast – Conflicts
http://www.mftech.org/dawn/screencasts/5_Conflicts/5_Conflicts.htm
zuletzt geprüft: 08.05.2009

Flügge 2009e Martin Flügge:
Screencast – Creating Projects
http://www.mftech.org/dawn/screencasts/6_creating_projects/creating_projects_new.htm
zuletzt geprüft: 08.05.2009

Flügge 2009f Martin Flügge:
Screencasts – More sophisticated diagrams
http://mftech.org/dawn/screencasts/8_more%20sophisticated%20Diagrams/8_more_sophisticated_Diagrams.htm
zuletzt geprüft: 08.05.2009

GMF 2009 Eclipse Foundation:
GMFgraph Hints
http://wiki.eclipse.org/GMFGraph_Hints
zuletzt geprüft: 08.05.2009

GMF 2009a Seweryn Niemiec, Jaap Retimes and Richard Gronback and others:
GMF Tutorial – Part 2
http://wiki.eclipse.org/GMF_Tutorial_Part_2
zuletzt geprüft: 08.05.2009

GMF 2009b Eclipse Foundation:
Developer Guide to the GMF Runtime Framework
http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.gmf.doc/prog-guide/runtime/Developer%20Guide%20to%20Diagram%20Runtime.html
zuletzt geprüft: 08.05.2009

IBM 2008 Chris Aniszczyk:
Plug-in development 101, Part 1: The fundamentals
http://www.ibm.com/developerworks/library/os-eclipse-plugindev1/index.html?ca=dgr-eclipse-1
zuletzt geprüft: 08.05.2009

Mammana 2008 Jean-Charles Mammana , Romain Meson, Jonathan Gramain:
GEF (Graphical Editing Framework) Tutorial
http://www.psykokwak.com/blog/images/gef/GEF_Tutorial.pdf
zuletzt geprüft: 08.05.2009

OMG 2006 Object Management Group:
MOF Core specification Version 2.0
http://www.omg.org/docs/formal/06-01-01.pdf
zuletzt geprüft: 08.05.2009

OSGi 2009 OSGi Alliance:
Automotive Electronics Market
http://www.osgi.org/Markets/Automotive
zuletzt geprüft: 08.05.2009

Raible 2007 Matt Raible:
Comparing Java Web Frameworks
http://static.raibledesigns.com/repository/presentations/ComparingJavaWebFrameworks-ApacheConUS2007.pdf
zuletzt geprüft: 08.05.2009

Schnepel 2008 Enrico Schnepel:
Entwicklung eines graphischen Editors zur Unterstützung eines modellgetriebenen Softwareentwicklungsprozesses
http://www.randomice.net/files/Diplomarbeit_Schnepel.pdf
zuletzt geprüft: 08.05.2009

Stepper 2008 Eike Stepper:
Preparing EMF Models for CDO
http://wiki.eclipse.org/Preparing_EMF_Models_for_CDO
zuletzt geprüft: 08.05.2009

SUN 2008 SUN:
GlassFish V3 runs on OSGi
http://blogs.sun.com/dochez/entry/glassfish_v3_runs_on_osgi
zuletzt geprüft: 08.05.2009

Voelter Blog 2007 Markus Völter:
Blog – Markus Völter
http://voelterblog.blogspot.com/2007/02/terminology-hel.html
zuletzt geprüft: 08.05.2009

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Glossar

Ajax	Asynchronous JavaScript and XML. Ajax ist eine webbasierte Technologie zur Kommunikation zwischen einem Webbrowser und ein Backend. Ajax-Anfragen können im Hintergrund ausgeführt werden und erfordern deshalb kein Neuladen der Webseite.
API	Application Programming Interface bezeichnet eine Schnittstelle für die Entwicklung von Anwendungen, die von einem System bereitgestellt wird.
AWT	Abstract Window Toolkit. UI-Bibliothek von Sun, welche auf den nativen GUI-Elementen des Betriebssystems aufbaut. Unterstützt nicht alle GUI-Elemente. Zum Beispiel keine Tree-Views.
Axis	Ist ein Framework für die Erstellung von SOAP-Anwendungen der Apache Software Foundation.
CDO	Connected Distributed Objects ist ein Projekt der Eclipse Foundation, welches sich mit der Verteilung von EMF-Modellen beschäftigt.
CIM	Das Computation Independent Model ist im Sinne der MDSD ein Modell, welches unabhängig von jeglicher Implementierung ist. Es beschreibt das System ohne Hinblick auf die spätere Umsetzung.
CMOF	Complete Meta Object Facility bezeichnet den vollständigen Umfang der Meta Object Facility Spezifikation der OMG.
CORBA	Common Object Request Broker Architecture ist eine Middleware zur Umsetzung von Verteilten System. Sie wird von der OMG spezifiziert.
CSS	Cascading Style Sheets ist eine Stylesheet-Sprache für HMTL- und XML-basierte Dokumente. Mit Hilfe von CSS kann die Formatierung dieser Dokumente beeinfluss werden.

Domäne	Domänen innerhalb der MDSD bezeichnen ein in sich geschlossenes Anwendungsgebiet. Sie bilden die Grundlage für die Entwicklung von domänenspezifischen Sprachen.
DSL	Domain Specific Language (domänenspezifische Sprachen) bezeichnet eine Sprache mit Bezug zu einer Domäne innerhalb der MDSD. DSLs können sowohl textuell, als auch grafisch sein.
ECF	Eclipse Communication Framework ist ein Projekt der Eclipse Foundation, welche sich mit der Message-basierten Kommunikation innerhalb von Eclipse beschäftigt.
Eclipse	Bezeichnet sowohl eine integrierte Entwicklungsumgebung, als auch ein Open-Source Projekt.
ECOOP	European Conference of Object-Oriented Programming. Eine jährlich stattfindende Konferenz auf der Themen rund um die Objektorientierte Programmierung diskutiert werden.
EMF	Eclipse Modeling Framework. Ein Projekt der Eclipse Foundation mit dem Ziel der Entwicklung und Verbreitung von modellgetriebenen Softwarearchitekturen.
EMFT	Eclipse Modeling Framework Tooling. Ein Unterprojekt des EMF Projektes, welches nützliche Werkzeuge im Zusammenhang mit EMF bereitstellt.
EMOF	Das Essential Meta Object Facility bildet den essentiellen Kern der MOF-Spezifikation der OMG wieder.
Equinox	Bezeichnet die OSGi-Implementierung von Eclipse und den Kern von RCP-Anwendungen.
GEF	Graphical Editing Framework beschäftigt sich mit der Bereitstellung von Werkzeugen und Architekturen zur einfachen Entwicklung von grafischen Editoren basierend auf Eclipse.
GMF	Das *Graphical Modeling Framework* ist ein Framework zur Erstellung von Editoren innerhalb von Eclipse. Es baut auf dem EMF und dem GEF auf, welches ebenfalls Frameworks der Eclipse Foundation sind.

GPRS	General Packet Radio Service ist eine Datenübertragungstechnologie innerhalb von GSM-Mobilfunknetzen. Die Daten werden dabei paketorientiert übertragen.
GUI	Das Graphical User Interface ist eine grafische Schnittstelle für Nutzerinteraktionen. Es kann sowohl die Eingaben des Anwenders entgegen nehmen, als auch die Ausgaben des Systems grafisch darstellen. In seiner Darstellungsweise unterscheidet es sich von der reinen textbasierten Nutzerschnittstelle (Textual User Interface, TUI)
Hibernate	Ist ein objektrelationaler Mapper zum Umwandeln von objektorientierten Sachverhalten auf eine relationale Datenbank.
HQL	Hibernate Query Language bezeichnet eine an SQL angelehnt Abfragesprache innerhalb von Hibernate. Im Gegensatz zu SQL werden allerdings Objekte und nicht Relationen abgefragt.
HTTP	Hypertext Transfer Protocol ist ein zustandsloses Protokoll zur Übertragung von Daten in einem Netzwerk. In der Regel wird es zum Transport von Webseiten verwendet. Es nutzt den TCP-Port 80.
IDE	Integrated Development Environment bezeichnet eine Software zur Entwicklung von Anwendungen. Sie erleichtern die Arbeit durch integrierte Werkzeuge wie Texteditoren oder Debugger.
IRC	Internet Relay Chat ist ein textbasiertes System zum Austausch von Chat-Nachrichten zwischen mehreren Nutzern.
Java EE / JEE	Java Enterprise Edition ist eine Plattform von Sun zur Entwicklung von Unternehmensanwendungen.
Java ME / JME	Java Micro Edition ist eine Plattform von Sun zur Entwicklung von Anwendungen auf embedded Systemen wie PDA, Handy oder Smartphone.
Java RE / JRE	Java Runtime Environment bezeichnet die Laufzeitumgebung der Programmiersprache Java. Sie umfasst unter anderem die Virtuelle Maschine von Java.
Java SE / JSE	Java Standard Edition ist eine Plattform von Sun zur Entwicklung von Java-basierten Anwendungen.

JET	Java Emitter Templates bezeichnet eine Template-basierte Sprache zur Transformation von Modellen in andere Modelle oder Texte.
JFace	JFace ist ein UI-Bibliothek, welche auf SWT aufsetzt und das Programmieren SWT-basierter Oberflächen wesentlich vereinfacht.
JMS	Java Messaging Service ist ein von SUN spezifizierter Standard zur nachrichtenbasierten Übertragung von Informationen.
JSF	Java Server Faces bezeichnet ein von Sun bereitgestelltes Framework zur Entwicklung von Webanwendungen.
JSP	Java Server Page ist eine Tag-basierte Sprache zur dynamischen Erzeugung von Dokumenten. Innerhalb eines Web-Containers werden JSPs zur Laufzeit in Servlets umgewandelt.
LAN	Local Area Network bezeichnet ein lokales Netzwerk, welches in der Regel auf TCP/IP beruht. LANs sind in der Regel räumlich begrenzt und erstrecken sich selten über mehr als ein Gebäude.
M2C	Model-to-Code Transformation. Eine Transformation, welche ein Modell in Quellcode umwandelt.
M2M	Model-to-Model Transformation. Eine Transformation, welche ein Modell in ein Modell umwandelt.
M2T	Model-to-Text Transformation. Eine Transformation, welche ein Modell in Text umwandelt.
MDA	Model Driven Architecture. Ein Ansatz zur Entwicklung von modellgetriebener Software. Basiert auf dem MOF der OMG.
MDD	Model Driven Development. Siehe MDSD.
MDSD	Model Driven Software Development. Ein konzeptueller Ansatz zur Entwicklung von modellgetriebener Software.
MDSE	Model Driven Software Engineering. Siehe MDSD.
MOF	Meta Object Facility bezeichnet das Meta-Modell der OMG, welchem auch die UML zugrunde liegt.

MVC	Model View Control ist ein Entwurfsmuster zur Trennung von Anzeige und Datenhaltung.
oAW	openArchitectureWare ist ein Framework, welches modellgetriebene Entwicklungsprozesse unterstützt.
OCL	Object Constraint Language. Eine Sprache zur Erstellung von Beschränkungen innerhalb eines Systems.
OMG	Object Management Group. Ein non-profit Industrie Konsortium.
OOSE	Object-Oriented Software Engineering. Eine Entwurfsmethode für objektorientierte Software.
ORM	Object Relational Mapping bezeichnet das Abbilden von Objekten auf relationale Datenbanken.
OSGi	Open Services Gateway Initiative ist eine Organisation, welche die OSGi Services Platform spezifiziert. Zuweilen wird auch diese Plattform, beziehungsweise die dahinter stehenden Technologien, als OSGi bezeichnet.
PDE	Plugin Development Environment ist ein Eclipse-Projekt zur Entwicklung von Plugins und OSGi-Bundles.
PIM	Platform Independent Model bezeichnet ein Modell innerhalb der MDSD welches keinen Zusammenhang zu einer Plattform, zum Beispiel einer Programmiersprache oder einem Betriebssystem, hat.
Plattform	Plattformen bezeichnen im Kontext der MDSD Systeme, die als Grundlage für die Generierung genutzt werden. Plattformen können zum Beispiel Betriebssysteme oder Programmiersprachen sein.
PSM	Platform Specific Model bezeichnet ein Modell innerhalb der MDSD welches einen Bezug zu einer Plattform hat.
QVT	Query View Transformation ist eine Spezifikation der OMG innerhalb des MOF zur Umsetzung von Modell-zu-Modell-Transformationen.
RCP	Rich Client Platform ist eine Konzept von Eclipse zur Erstellung von Anwendungen basierend auf dem Equinox-Kern.

REST	Representational State Transfer bezeichnet ein HTTP-basiertes Kommunikationsmodell. Es wird häufig für die Implementierung von Webservices genutzt.
RMI	Remote Method Invocation bezeichnet den Aufruf einer Methode eines Objektes innerhalb eines verteilten Systems. Ebenfalls wird unter RMI die entsprechende Implementierung für Java bezeichnet.
RPC	Remote Procedure Call. Ein entfernter Prozedur-Aufruf. Im Gegensatz zur RMI findet dieser nicht an einem Objekt statt.
SOA	Service Oriented Architecture bezeichnet ein Konzept zur Nutzung von Diensten in einer meist webbasierten Architektur.
SOAP	Simple Object Access Protocol (veraltet, heute: SOAP) ist ein Netzwerkprotokoll zur Implementierung von Remote Procedure Calls. Als Protokollbasis wird XML verwendet.
SQL	Structured Query Language ist eine Abfragesprache für relationale Datenbanken.
SWT	Standard Widget Toolkit. Eine UI-Bibliothek von IBM. Die Eclipse IDE basiert auf SWT.
UML	Unified Modeling Language ist eine universelle Modellierungssprache zur Erstellung von Softwaresystemen. Sie wird durch die OMG spezifiziert.
UMTS	Universal Mobile Telecommunication System bezeichnet das Mobilfunksystem der dritten Generation. UMTS ist der Nachfolger von GMS.
URI	Uniform Resource Identifier beschreibt ein universelles Format zur Beschreibung von Ressource Locations.
Webservice	Ein auf Web-Technologien basierender Dienst, der über einen URI eindeutig identifiziert und gefunden werden kann. Dabei stellt er seine Funktionalitäten in Form von Schnittstellenbeschreibungen anderen Systemen zur Verfügung.
WLAN	Wireless LAN bezeichnet einen Standard zur kabellosen Übertragung von Daten.

WSDD	Web Service Deployment Descriptor. Eine Beschreibung eines Webservices innerhalb des Axis Frameworks wird über eine WSDD definiert.
WSDL	Web Service Description Language ist eine Sprache zur plattformunabhängigen Beschreibung von Webservice-Schnittstellen.
XMI	XML Meta Interchange ist ein interoperables XML-basiertes Austauschformat der OMG für MOF-konforme Modelle.
XML	eXtensible Markup Language. Eine Auszeichnungssprache für strukturierte Inhalte. XML ist textbasiert und die Basis für viele verschiedene Anwendungen.
XML-RPC	XML-Remote Procedure Call. Ein entfernter Methodenaufruf basierend auf XML. Es wird als Vorgänger der heutigen SOAP-Version bezeichnet.
XMPP	eXtensible Messaging and Presence Protocol ist ein Internetstandard für XML-basierte Kommunikation.

Index

A

B

C

D

E

F

G

H

J

K

L

M

N

O

P

R

S

T

U

W

X

Anhang

A Anforderungskatalog

A.1 Einleitung

Der nachfolgende Anforderungskatalog bildet die Grundlage für die Entwicklung einer kollaborativen, webbasierten Erweiterung für GMF-Editoren, nachfolgend als Dawn bezeichnet.

A.2 Zielbestimmungen

Dawn bietet als webbasierte Erweiterung von GMF-Editoren die Möglichkeit einer kollaborativen Nutzung von GMF-Diagrammen. In den folgenden Kapiteln werden die mindesten Anforderungen sowie Wunsch- und Abgrenzungskriterien spezifiziert.

A.2.1 Musskriterien

Kommunikation

ID	Name	Beschreibung
M-KM-01	Firewall Transparenz	Das System soll unabhängig von Firewall-Konfigurationen arbeiten können.
M-KM-02	Ausfallsicherheit und Transparenz	Das System soll sich robust gegen Ausfall des Servers verhalten.
M-KM-03	Netzwerk-unabhängigkeit	Der Nutzer soll unabhängig von einer Netzwerkverbindung am Editor arbeiten können. Bei bestehender Verbindung sollen die Modellbestände synchronisiert werden.
M-KM-04	Starting Watcher-Thread when opening editor	Wenn der Editor gestartet wird, soll, wenn der Server vorhanden ist, automatisch eine Verbindung aufgenommen werden.
M-KM-05	Stopping Watcher-Thread when closing editor	Wenn der Editor geschlossen wird, soll die Verbindung sauber abgebaut werden.
M-KM-06	Open Exchange Format	Die Kommunikation soll über ein interoperables Austauschformat erfolgen, um später andere Clients (ggf. andere Programmiersprachen) mit in das System einbinden zu können

Ressourcenmanagement und Synchronisation

ID	Name	Beschreibung
M-RS-01	Synchronize Changed Nodes - Client	Geänderte Knoten sollen auf dem Client mit dem Server synchronisiert werden können. Die GUI soll die Änderungen anzeigen.
M-RS-02	Synchronize Changed Nodes - Server	Geänderte Knoten sollen auf dem Server mit den Client Informationen synchronisiert werden können
M-RS-03	Synchronize Inserted Nodes - Client	Eingefügte Knoten sollen auf dem Client mit dem Server synchronisiert werden können. Die GUI soll die Änderungen anzeigen.
M-RS-04	Synchronize Inserted Nodes - Server	Eingefügte Knoten n sollen auf dem Server mit den Client Informationen synchronisiert werden können.
M-RS-05	Synchronize Deleted Nodes - Client	Gelöschte Knoten sollen auf dem Client mit dem Server Synchronisiert werden können. Die GUI soll die Änderungen anzeigen.
M-RS-06	Synchronize Deleted Nodes - Server	Gelöschte Knoten sollen auf dem Server mit den Client Informationen synchronisiert werden können
M-RS-07	Synchronize Changed Edges - Client	Geänderte Kanten sollen auf dem Client mit dem Server Synchronisiert werden können. Die GUI soll die Änderungen anzeigen.
M-RS-08	Synchronize Changed Edges - Server	Geänderte Kanten sollen auf dem Server mit den Client Informationen synchronisiert werden können
M-RS-9	Synchronize Inserted Edges - Client	Eingefügte Kanten sollen auf dem Client mit dem Server synchronisiert werden können. Die GUI soll die Änderungen anzeigen.
M-RS-10	Synchronize Inserted Edges - Server	Eingefügte Kanten sollen auf dem Server mit den Client Informationen synchronisiert werden können.
M-RS-11	Synchronize Deleted Edges - Client	Gelöschte Kanten sollen auf dem Client mit dem Server Synchronisiert werden können. Die GUI soll die Änderungen anzeigen.
M-RS-12	Synchronize Deleted Edges - Server	Gelöschte Kanten sollen auf dem Server mit den Client Informationen synchronisiert werden können
M-RS-13	Object-Identification via XMI-Ids	Alle EObjects, auch die Elemente in den Views, sollen über eine XMI ID identifiziert werden können.
M-RS-14	Store Diagram in DB	Der aktuelle Zustand des Diagramms soll auf dem Server in einer Datenbank gespeichert werden.
M-RS-17	Conflict-Update-Blocking	Wenn Konflikte erkannt werden, dann soll keine Information zum Server übertragen werden. Das Übertragen der Informationen an den Server soll so lange unterbunden werden, bis alle Konflikte beseitigt wurden.
M-RS-18	Conflict Detection	Konflikte sollen im System erkannt werden können.
M-RS-19	Conflict Avoidance	Konflikte sollen vom System wenn möglich vermieden werden
M-RS-20	Conflict Resolving	Konflikte sollen im System gelöscht werden können
M-RS-21	Change multiple objects before publish	Bevor der Modellbestand an den Server geschickt werden soll, soll es möglich sein, dass mehrere Objekte geändert werden können. Dies bedingt, dass lokale Änderungen erkannt und geblockt werden müssen.
M-RS-22	Joined resource files	Die Synchronisierung soll für alle GMF Resource funktionieren, und denen GMF und EMF Resource in einer Datei gespeichert werden
M-RS-23	Optimistic Locking	Durch ein optimistische Lock-Verfahren soll verhindert werden, dass Race-Conditions auftreten können
M-RS-24	Pessimistic Locking	Durch ein manuelles pessimistisches Lockverfahren sollen Konflikte vermieden werden.

User Management

ID	Name	Beschreibung
M-UM-01	Add User to Project	Benutzer sollen Projekten hinzugefügt werden können.
M-UM-02	Store User-Information in DB	Die Nuterzinformationen sollen in der Datenbank gespeichert werden.
M-UM-03	Verschlüsselte Passwörter	Passwörter der Nutzer sollen verschlüsselt abgelegt werden.
M-UM-04	Join only to assigned projects	User dürfen nur Projekte joinen, zu denen sie eingeladen wurden.

Eclipse-Editor

ID	Name	Beschreibung
M-EE-01	User-Login	Benutzer sollen sich über die Eclipse GUI am System anmelden können.
M-EE-02	User-Logoff	Nutzer sollen sich über die Eclipse GUI vom System abmelden können.
M-EE-03	Preferences – Server	Der Nutzer soll über die Editor Preferences die Angaben zum Server editieren können. Hierzu zählen URL und Port-Nummer
M-EE-04	Preferences – Update Intervals	Der Nutzer soll über die Eclipse Preferences die Angabe der Update Intervalls bestimmen können. Die Angebe erfolgt in Millisekunden
M-EE-05	Create Projects	Über den Eclipse Client sollen Projekte auf dem Server angelegt werden können.
M-EE-06	Visuals for Conflict Detection	Das System soll eine geeignete Visualisierung für Konflikte innerhalb des Editors anzeigen
M-EE-07	GUI Components for Conflict Resolving	Es sollen dem Nutzer Dialoge zur Verfügung gestellt werden, um Konflikte zu beseitigen
M-EE-08	Project-Management Dialog	Es soll einen Dialog geben, über den der Initialtor eines Project die Nutzer verwalten kann

Web-Komponenten

ID	Name	Beschreibung
M-WE-01	User-Login	Benutzer sollen sich über das Web-Frontend am System anmelden können.
M-WE-02	User-Logoff	Nutzer sollen sich über das Web-Frontend vom System abmelden können.
M-WE-03	Web-Viewer	Über das Web-Frontend sollen alle Änderungen der Eclipse Nutzer nachvollzogen werden können

Generierung

ID	Name	Beschreibung
M-GE-01	Generation Model Wizard	Über einen Wizard soll das PSM für die Netzwerkschicht erstellt werden können
M-GE-02	Generation Option	Über eine geeignete Option soll der Quellcode für den Server und das Client-Erweiterungsplugin erzeugt werden.
M-GE-03	Generation for simple Editors	Die Generierung der Netzwerkschicht soll für einfache GMF-Editoren möglich sein.
M-GE-04	Generate GMF-Figure for Web	Es die Repräsentation der GMF-Figures für den Web-Viewer generiert werden.

A.2.2 Wunschkriterien

Kommunikation

ID	Name	Beschreibung
W-KM-01	Dynamic Network Communication	Das System soll selbstständig entscheiden können, wie es die Daten verschickt. zwei Möglichkeiten stehen zur Verfügung: a) die Daten werden als Plain XML verschickt • Vorteil: Client und Server sparen Rechenzeit • Nachteil: Langsame Verbindung bei großen Projekten b) die Daten werden gezipped verschickt • Vorteil: schnellere Verbindung bei großen Projekten • Nachteil: mehr Rechenarbeit bei Server und Client Der Server soll in der Lage sein beide Formate transparent entpacken zu können. Der Client soll anhand seiner Verbindung prüfen können, ob er ZIP oder PLAIN verschickt.
W-KM-02	Extensible Network Layer	Das zugrunde liegende Netzwerkprotokoll soll geändert, bzw. erweitert werden können. Soll es möglich sein, dass System später auf andere Technologien wie zum Beispiel RMI, JMS, aufbauen zu können.
W-KM-03	Runtime System Update	Das System soll in der Lage sein durch ein Update *zur Laufzeit* mit neuen Editoren umgehen zu können.
W-KM-04	Chat	Die Nutzer sollen sich über einen Chat unterhalten können
W-KM-05	Audio-Communication	Die Nutzer sollen sich über eine Sprachverbindung (z.B. Skype) unterhalten können.
W-KM-06	Information via email	Nutzer sollen per Mail über die Einladung in ein Project informiert werden können.
W-KM-07	Secure Communication	Die Netzwerkverbindung soll verschlüsselt erfolgen können.

Ressourcenmanagement und Synchronisation

ID	Name	Beschreibung
W-RS-01	Multi-Object-Editing	Mehrere Nutzer sollen gleichzeitig an einem Objekt arbeiten können, solang sie nicht am selben bereich des Objektes arbeiten
W-RS-02	Multiple resource files	Die Synchronisierung soll auch für alle GMF Projekte funktionieren, und denen GMF und EMF Ressource in unterschiedlichen Datei gespeichert werden
W-RS-03	Store Diagram History in DB	Die History des Diagramms soll in der Datenbank gespeichert werden
W-RS-04	Reorient of Edges	Edges, welche eine Verbindung zwischen Knoten A und Knoten B darstellen, sollen von Knoten B zum Knoten C geändert werden können, damit die Verbindung zwischen Knoten A und C besteht. Dieses Verhalten soll in allen anderen Clients synchronisiert werden können.

Eclipse-Editor

ID	Name	Beschreibung
W-EE-01	Change User Data Extended	Der Dialog für die Nutzerdaten soll um folgende Eigenschaften erweitert werden - ICQ Nummer - Skype Nummer - Nickname
W-EE-02	Internationalisierung	Die Dialoge sollen über Fragmentprojekte internationalisierbar sein.

W-EE-03	Preference Publish-Mode	Über den Publish Mode soll bestimmt werden können, wie publiziert wird. „on save“ oder „on interval“
W-EE-04	Locking not available while offline	Das Locking Menü darf nicht angezeigt werden, wenn der Server nicht verfügbar ist.
W-EE-09	Change User Data	Es soll einen Dialog geben, über den der Benutzer seine Daten ändern kann -Name -Passwort - email

Web-Editor

ID	Name	Beschreibung
W-WE-01	Create Projects	Über das Web-Frontend sollen Projekte auf dem Server angelegt werden können.
W-WE-02	Edit Diagram	Das Diagram soll über das Web-Frontend bearbeitet werden können.
W-WE-03	Project-Management Dialog (Web-Based)	Es soll einen Dialog geben, über den der Initialtor eines Project die Nutzer verwalten kann
W-WE-04	Change User Data (Web-based)	Es soll einen Dialog geben, über den der Benutzer seine Daten ändern kann -Name -Passwort - Nickname

Generierung

ID	Name	Beschreibung
W-GE-01	Generation for complex GMF-Editors	Die Generierung der Netzwerkschicht soll für *komplexe* GMF-Editoren möglich sein.

A.2.3 Abgrenzungskriterien

ID	Name	Beschreibung
A-AK-01		Die Generierung basiert ausschließlich auf GMF andere Generierungen werden nicht in Betracht gezogen.
A-AK-02		Die Eclipse-basierten Editoren werden ausschließlich als Lösung für GMF-Editoren entwickelt. EMF-Editoren werden nicht berücksichtigt
A-AK-03		Die Implementierung erfolgt ausschließlich in Java >= 1.5. Andere Programmiersprachen werden nicht berücksichtigt.
A-AK-04		Das System wird ausschließlich innerhalb der Eclipse Platform lauffähig sein. Eine Entwicklung als Rich Client Platform wird nicht stattfinden.
A-AK-05		Das Testen der Funktionalität erfolgt ausschließlich unter Windows XP. Unter anderen Betriebssystemen finden keine Tests statt.
A-AK-06		Das System wird ausschließlich mit den unter 0 aufgeführten Systemen und Komponenten entwickelt und getestet. Entwicklungen und Tests mit bzw. unter anderen Komponenten finden nicht statt.

A.3 Produkteinsatz

Das System soll genutzt werden um GMF-Editoren kollaborativ zu verbinden. Dabei ist die genaue Zielbestimmung abhängig von dem Einsatzzweck des entwickelten GMF-Editors. Zusätzlich wird über den WebViewer die Möglichkeit gegeben die Projektverlauf auch auf System ohne Eclipse zu verfolgen. Das System ist also auch für den mobilen Einsatz gedacht.

A.4 Produktumgebung

A.4.1 Software

Softwarekomponente	Version
Java	1.5
GMF	2.1.0
EMF	2.4.0
Eclipse	3.4.0 Ganymede
MySQL	3.21
Apache Tomcat	5.x
Apache Axis	1.3

A.4.2 Hardware

Softwarekomponente	Version
Client für kollaborativen GMF-Editor	Internetfähiger Rechner, der den Anforderungen an das aktuelle Eclipse Release genügt.
Cient für Web-Editor	Internetfähiger Rechner, der über einen Mozilla-kompatiblen Browser verfügt
Server	Internetfähiger Rechner, der den Anforderungen an das aktuelle Eclipse Release genügt. Genügend Festplattenplatz die die Speicherung der Daten der Projekte

B Inhalt der beigefügten CD

- CD
  - Quellcode
    - DawnServer
    - org.mftech.dawn.codegen
    - org.mftech.dawn.codegen.edit
    - org.mftech.dawn.codegen.editor
    - org.mftech.dawn.runtime.client
    - org.mftech.diagram.uml.class
    - org.mftech.diagram.uml.class.diagram
    - org.mftech.diagram.uml.class.diagram.communication
    - org.mftech.diagram.uml.class.diagram.extension
    - org.mftech.diagram.uml.class.edit
    - org.mftech.diagram.uml.class.editor
  - Weblinks
    - ComputerZeitung 2009
    - ECF 2008
    - Eclipse 2009
    - Eclipse 2009a
    - EMFT 2008
    - Flügge 2009a
    - Flügge 2009b
    - Flügge 2009c
    - Flügge 2009d
    - Flügge 2009e
    - GMF 2009
    - GMF 2009a
    - IBM 2008
    - Mammana 2008
    - OSGi 2009
    - Schnepel 2008
    - Stepper 2008
    - SUN 2008
    - Voelter Blog 2007
    - GMF 2009b
    - Raible 2007
    - OMG 2006