JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche
Fakultät

Konsistenzprüfung von Software Design Modellen in Mehrbenutzerumgebungen

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Masterstudium

Software Engineering

Eingereicht von:
Andreas Gruber

Angefertigt am:
Institut für Systems Engineering and Automation

Beurteilung:
Univ.-Prof. Dr. Alexander Egyed, M.Sc.

Linz, September, 2011

Kurzfassung

Softwaresysteme werden immer komplexer und bei der Entwicklung von neuen Systemen müssen daher viele Personen zusammenarbeiten. Um ein gemeinsames Verständnis über das System aufbauen zu können und den gemeinsamen Entwurfsprozess zu erleichtern, werden häufig Modelle eingesetzt. Diese Modelle werden von verschiedenen Entwicklern bearbeitet und werden daher Inkonsistenzen enthalten.
Mit den vorhandenen Technologien zur lokalen Konsistenzprüfung bei den einzelnen Entwicklern können diese Inkonsistenzen in den Modellen entdeckt werden. Für eine lokale Konsistenzprüfung ist es jedoch erforderlich, dass das gesamte Modell bei jedem Entwickler gespeichert wird. Wenn die einzelnen Benutzer das Modell mit den gleichen Konsistenzregeln evaluieren, wird sehr häufig die gleiche Konsistenzregel mit den gleichen Modelldaten von unterschiedlichen Benutzern evaluiert und es entsteht Redundanz. Sie entsteht auch, da meist ein großer Teil des Modells unverändert bleibt und somit nur einige wenige Konsistenzregeln durch Änderungen eines einzelnen Entwicklers betroffen sind. Lokale Konsistenzprüfung ist daher ineffizient und es entsteht Redundanz bei den Modell- und Konsistenzdaten.
Um diese Probleme zu lösen und kollaborative Modellierung bei gleichzeitiger Konsistenzprüfung zu ermöglichen, wird in dieser Masterarbeit eine Konsistenzprüfung für gemeinsam genutzte Software Design Modelle vorgestellt. Dabei wird das gemeinsame Modell auf einem Server in einem öffentlichen Bereich gespeichert. Um Redundanzen bei den Modelldaten zu vermeiden, werden nur die Änderungen der einzelnen Benutzer zum Server übertragen und in private Bereiche gespeichert. Für jeden dieser Bereiche gibt es eine eigene Konsistenzprüfung. Dabei greifen jene Prüfungen, die einen privaten Bereich betreffen, auf die Daten jener Prüfung zu, die für den öffentlichen Bereich zuständig ist. Durch diesen Ansatz können Konsistenzdaten wiederverwendet und Redundanzen vermieden werden.
Das Ergebnis dieser Arbeit ist ein Client-Server System auf Basis einer bereits bestehenden inkrementellen Konsistenzprüfung. Mit diesem System können mehrere Entwickler an einem gemeinsamen Modell arbeiten, wobei für jeden Entwickler individuell die Konsistenz seiner Änderungen am globalen Modell mit Hilfe der öffentlichen Modell- und Konsistenzdaten geprüft wird. Zusätzlich wurde das System evaluiert und mit der lokalen Variante verglichen. Diese Evaluierung zeigt, dass die Konsistenzprüfung bei kollaborativen Arbeiten an einem Modell effizient gestaltet werden kann.

Abstract

Software systems are getting more and more complex and in order to develop them, many people have to work together. To get a better common understanding about the system to build and to ease the collaborative design process, developers use models of the system. Different developers are involved in creating these models and therefore the models will contain inconsistencies.
With the existing technologies for local consistency checking the developers can discover inconsistencies in their local models. For a local consistency check it is necessary that the entire model is stored on each local machine. If each developer evaluates the model with the same set of design rules, the same model data will be checked with the same rules. Generally speaking, the developers do the same work although it would be enough that one does the work and share the results with the others. Often the developers have interest on some parts of the model and therefore only some of the rules are important to them. Local consistency checking is inefficient and it creates redundant model and consistency data.
To solve these problems and to allow collaborative modeling with consistent models, this thesis presents an approach for consistency checking in a multiuser-modeling environment. The shared model will be stored in a public area on a server. To avoid redundant model data, only the changes of each user will be transmitted to the server and stored in private areas. Each area has its own consistency checker. The consistency checkers for the private area can access the data of the consistency checker for the public area. With this mechanism the consistency data can be reused and redundancies are avoided.
The main result of this thesis is a client-server system based on an existing approach for incremental consistency checking. With this system many developers can work together on a shared model. The consistency of each private change on the public model will be evaluated individually for each developer, using the data of the public model and consistency checker. In addition, the system was evaluated and compared with the local variant of the consistency checker. This evaluation shows that consistency checking in a collaborative modeling environment can be done efficiently.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Kapitel 1

Einleitung

Software Systeme werden in der heutigen Zeit immer größer und umfangreicher. Es wird daher unumgänglich, dass eine größere Gruppe von Personen an der Entwicklung der Systeme beteiligt ist. Nicht nur bei der Implementierung der Software sind viele Personen beteiligt, sondern schon beim Entwurf der Software ist eine Zusammenarbeit notwendig. Die Modelle des Entwurfs sollen ein gemeinsames Verständnis über das System aufbauen und den gesamten Prozess der Entwicklung vereinfachen. Der Entwurf wird oft gemeinsam und mit Hilfe von entsprechenden Werkzeugen durchgeführt. Die daraus resultierenden Modelle können sehr groß und umfangreich werden. Die Komplexität der Modelle erhöht auch die Fehleranfälligkeit und es entstehen Inkonsistenzen. Um Inkonsistenzen zu entdecken und zu vermeiden wird eine Konsistenzprüfung der Modelle immer wichtiger.

1.1 Problemstellung

Mit den vorhandenen Technologien zur lokalen Konsistenzprüfung können Inkonsistenzen in Modellen entdeckt werden. Ein lokale Konsistenzprüfung bedeutet, dass bei jedem Entwickler das gesamte Modell verfügbar sein muss und auf diesem Modell wird eine vollständige Prüfung durchgeführt. Ein Szenario mit zum Beispiel 20 Entwicklern würde sich wie Abbildung 1.1 darstellen. Jeder Entwickler bekommt das Modell für seine Änderungen, eine Liste von Konsistenzregeln und bei jedem Benutzer wird eine lokale Konsistenzprüfung ausgeführt

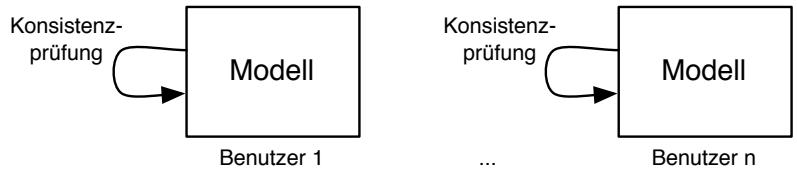


Abbildung 1.1: lokale Konsistenzprüfungen

Diese einfache Möglichkeit der Konsistenzprüfung von Modellen bringt Probleme mit sich, beziehungsweise sie skaliert nicht sehr gut. Jeder Benutzer besitzt das gesamte Modell und dadurch sind viele redundante Modelldaten im Umlauf. Weitere redundanten Daten entstehen bei der Konsistenzprüfung, denn jeder Benutzer prüft immer die gesamten Konsistenzregeln. Wenn die einzelnen Benutzer das Modell mit den gleichen Konsistenzregeln evaluieren, wird sehr häufig die gleiche Konsistenzregel mit den gleichen Modelldaten evaluiert und dies ergibt natürlich bei jedem Benutzer das gleiche Ergebnis. Dabei benötigen

viele Benutzer nur kleine Teiles des Modells für ihre Arbeit und auf diesen Bereichen führen sie ihre Änderungen aus. Durch diese Änderungen sind nur einige Konsistenzregeln betroffen, alle weiteren Konsistenzregeln werden eigentlich nicht benötigt, beziehungsweise werden unnötig berechnet. Eine erste Möglichkeit diese Probleme einzudämmen, wäre die Verteilung von Teilmodellen, wie sie in Abbildung 1.2 zu sehen ist. Die Benutzer bekommen nur mehr Teilmodelle und die Konsistenzprüfung bleibt lokal.

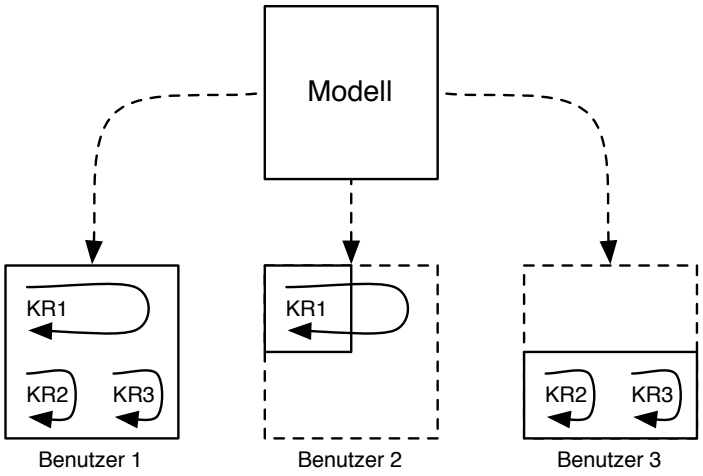


Abbildung 1.2: Verwendung von Teilmodellen

Auch bei dieser Aufteilung können Probleme entstehen. *Benutzer 1* kann ganz normal arbeiten und alle Konsistenzregeln können evaluiert werden. *Benutzer 2* besitzt nur einen kleinen Teil des Modells, dadurch fehlen der Konsistenzregel *KR1* Daten, diese müssten nachgeladen werden oder eine andere Möglichkeit wäre, das Ergebnis der Regel von *Benutzer 1* zu verwenden. *Benutzer 3* hat zwar genug Daten für seine Konsistenzregeln, jedoch sind seine Evaluierungen redundant, denn *Benutzer 1* hätte bereits die Ergebnisse für *KR3* und *KR3*.
Diese beiden Beispiele zeigen, dass eine lokale Konsistenzprüfung von gemeinsamen Modellen ineffizient ist und es entsteht Redundanz sowohl bei den Modelldaten als auch bei den Konsistenzdaten.

1.2 Ziel

Um die beschriebenen Probleme zu lösen und kollaborative Modellierung bei gleichzeitiger Konsistenzprüfung zu ermöglichen, wird in dieser Masterarbeit eine Konsistenzprüfung für gemeinsam genutzte Software Design Modelle vorgestellt. Dabei werden sowohl Modelldaten, als auch Konsistenzprüfung auf einem zentralen Server verwaltet. Das Modell soll in zwei Bereiche unterteilt werden, in einen öffentlichen Bereich für die gemeinsamen Daten und in mehrere private Bereiche für die einzelnen Benutzer. In diese privaten Bereiche werden die Änderungen der Benutzer am gemeinsamen Modell eingetragen. Für jeden dieser Bereiche ist eine eigene Konsistenzprüfung vorgesehen. Dabei greifen jene Prüfungen, die einen privaten Bereich betreffen auf die Konsistenzdaten der Prüfung für den öffentlichen Bereich zu. Durch dieses System können Modell- und Konsistenzdaten gemeinsam genutzt werden. Für die einzelnen Benutzer müssen nur die geänderten Modellelemente und die betroffenen Konsistenzdaten gespeichert werden. Die Konsistenzprüfung soll

Redundanzen bei Modell- und Konsistenzdaten vermeiden und auch bei vielen Benutzern skalieren.
Das Ergebnis dieser Arbeit wird ein Client-Server System auf Basis einer bereits bestehenden inkrementellen Konsistenzprüfung sein. Das System erweitert das Modellierungswerkzeug Rational Software Modeler von IBM und mit ihm können mehrere Entwickler an einem gemeinsamen Modell arbeiten. Für jeden Entwickler wird individuell die Konsistenz seiner Änderungen am globalen Modell mit Hilfe der öffentlichen Modell- und Konsistenzdaten geprüft. Der Server kann mehrere Projekte für viele Benutzer verwalten. Die Modelle werden am Client bearbeitet und jede Änderung wird sofort vom Client zum Server übertragen. Diese Änderungen werden in den entsprechenden Bereich eingetragen und stoßen die inkrementelle Konsistenzprüfung an. Das Ergebnis der Konsistenzprüfung kann unmittelbar von den Clients wieder angezeigt werden.
Der Ansatz dieser Arbeit und die daraus resultierende Implementierung sollen auch evaluiert werden, um Vor- und Nachteile diskutieren zu können. Daher soll das Client-Server System mit der lokalen Variante verglichen werden. Ziel dieser Evaluierung ist, dass überprüft wird ob die Konsistenzprüfung bei kollaborativen Arbeiten an einem Modell effizient gestaltet werden kann.

1.3 Übersicht

Kapitel 2 - Stand der Wissenschaft

Das erste Kapitel zeigt Bereiche der Wissenschaft auf, in denen sich diese Masterarbeit ansiedelt. Es werden Arbeiten im Bereich der Konsistenzprüfung von Software Design Modellen und im Bereich des kollaborativen Arbeiten an gemeinsamen Modellen vorgestellt.

Kapitel 3 - Technologie

Das Kapitel über die Technologie beschreibt theoretische Grundlagen und allgemeine Beispiele für die Technologien, die in dieser Arbeit eingesetzt werden, beziehungsweise die Basis für diese Arbeit und die Implementierung sind.

Kapitel 4 - Ideen und Varianten

Für eine Konsistenzprüfung in einer Mehrbenutzerumgebung gibt es viele verschiedene Lösungsansätze. Diese Kapitel fasst die verschiedensten Ideen und Varianten wie eine solche Konsistenzprüfung gestaltet werden kann, zusammen. Es wird auch der Lösungsansatz dieser Arbeit präsentiert.

Kapitel 5 - Lösungskonzept

Das Lösungskonzept greift die gefundenen Ideen und Varianten auf, um daraus ein konkretes Konzept für die Konsistenzprüfung in einer Mehrbenutzerumgebung zu erstellen. Das Kapitel bildet die allgemeine Basis für die Implementierung.

Kapitel 6 - Implementierung

Das Kapitel der Implementierung beschäftigt sich damit, wie das Lösungskonzept in eine funktionsfähige Konsistenzprüfung für eine Mehrbenutzerumgebung umgesetzt wurde. Es werden die wichtigsten Punkte der Implementierung des Prototyps beschrieben.

Kapitel 7 - Benutzerschnittstelle

Dieses Kaptitel beschreibt die Benutzerschnittstelle der Implementierung und demonstriert anhand eines Beispiels welche Funktionen vorhanden sind und wie sie bedient werden können.

Kapitel 8 - Evaluierung

Die Evaluierung vergleicht die Konsistenzprüfung in der Mehrbenutzerumgebung mit der Konsistenzprüfung für einzelne Benutzer. Mit Hilfe von Kennzahlen werden Vor- und Nachteile des Ansatzes dieser Arbeit diskutiert.

Kapitel 9 - Ausblick und Fazit

Im letzten Kapitel wird ein Ausblick gegeben, wie die Konsistenzprüfung von Software Design Modellen in Mehrbenutzerumgebungen fortgeführt und erweitert werden kann. Abgeschlossen wird die Arbeit mit einer Zusammenfassung und einem Fazit.

Kapitel 2

Stand der Wissenschaft

Bei der Konsistenzprüfung von Software Design Modellen in einer Mehrbenutzerumgebung können zwei Hauptberührungspunkte mit anderen Arbeiten der Wissenschaft beschrieben werden. Einerseits ist dies die Prüfung von Konsistenz in Software Design Modellen und andererseits das kollaborative Arbeiten von mehreren Personen an einem gemeinsamen Modell.

2.1 Kollaboration

Die Arbeit von Whitehead [Whi07] gibt einen allgemeinen Einstieg und Überblick in das Thema der Kollaboration im Software Engineering. Die Entwicklung eines komplexen Softwaresystems kann für eine einzelne Person schnell zu einer unüberschaubaren und fehlerbehafteten Aufgabe werden. Um ein komplexes System in entsprechender Zeit und Qualität zu erstellen, müssen mehrere Entwickler zusammenarbeiten, aber sobald in einer Gruppe gearbeitet wird, ergeben sich Probleme. Für den Einzelnen wird es schwierig, alle Details des Systems zu kennen und die einzelnen Personen können nicht mehr mitverfolgen, welche Aufgaben die anderen Mitglieder der Gruppe ausführen. Es besteht die Gefahr, dass die gleiche Arbeit mehrmals durchgeführt wird. Für ein System gibt es viele verschiedenen Lösungswege und Auffassungen der einzelnen Gruppenmitglieder über diese Wege, deshalb muss über die Architektur und das Design ein gemeinsames Verständnis herrschen. Mit den Methoden der Kollaboration im Software Engineering wird versucht die beschriebenen Probleme zu lösen und die Zusammenarbeit zu verbessern [Whi07].

Hildenbrand et al. [HRG+08] zeigen, in welchen Bereichen der Softwareentwicklung kollaborative Methoden zum Einsatz kommen können. Bei der kollaborativen Anforderungsanalyse werden gemeinsam die Interessen aller Beteiligten am Projekt ermittelt und in konkrete Anforderungen für das zu erstellende System umgewandelt. Die kollaborative Modellierung erstellt aus den Anforderungen Modelle für die Architektur und die einzelnen Komponenten. Die so entstandenen Modelle bilden die Basis für den nächsten Bereich, die Kollaboration in Implementierung, Test und Wartung. In diesem Bereich ist das wichtigste Artefakt der Quellcode und wie er gemeinsam erstellt, getestet und gewartet werden kann [HRG+08].

Da sich diese Masterarbeit mit Software Modellen in einer Mehrbenutzerumgebung beschäftigt, werden in weiterer Folge die unterschiedlichen Methoden, Probleme und Werkzeuge der kollaborativen Modellierung detaillierter beschrieben.

2.1.1 Herausforderungen der kollaborativen Modellierung

Bei der kollaborativen Modellierung arbeiten mehrere Personen an einem gemeinsamen Modell und somit können sie auch die gleichen Modellelemente bearbeiten. Durch dieses gleichzeitige Bearbeiten können Konflikte entstehen. Bang et al. [BPE+10] definieren einen Konflikt in einem Modell als eine Designentscheidung, die nicht mit den bisherigen Designentscheidungen vereinbar ist und daher noch nicht in das globale Modell übernommen werden kann.
Koegel et al. [KHvWH10] gehen in ihrer Arbeit näher auf die Themen der zentralen Speicherung eines gemeinsamen Modells und auf die Konflikterkennung ein. Da Modelle immer mehr im gesamten Lebenszyklus eines Projekts verwendet werden, wird eine zentrale Änderungsverfolgung und Versionskontrolle immer wichtiger. Versionskontrolle für textbasierte Dokumente, wie zum Beispiel für Quellcode, ist seit vielen Jahren im Einsatz. Die bisherigen Techniken eignen sich nur bedingt für Modelle, vor allem wenn die Modelle als Graph dargestellt und bearbeitet werden. Im Gegensatz zu Quellcode sind Modelle nicht zeilenorientiert. Wird beispielsweise in einem UML-Diagramm eine Assoziation zwischen zwei Klassen erstellt, ist dies eine strukturelle Änderung und es ist schwierig sie in einer zeilenorientierten Versionskontrolle darzustellen. Es gibt verschiedene Ansätze zur besseren Versionskontrolle von Modellen. Der zustandsbasierte Ansatz speichert nur den aktuellen Zustand eines Modells und wenn eine Änderung stattfindet, muss die Differenz zwischen den Zuständen ermittelt werden. Beim änderungsbasierten Ansatz werden die Änderungen am Modell zum Zeitpunkt der Änderung aufgezeichnet und in die Versionskontrolle gespeichert. Der letzte Ansatz ist der operationsbasierte, er ist eine Spezialform des änderungsbasierten Ansatzes, dabei wird eine Änderung als Transformation des Ursprungszustands des Modells in seinen geänderten Nachfolgezustand interpretiert [KHvWH10].
Eine Versionskontrolle erlaubt auch, dass mehrere Entwickler Änderungen an einem Modell durchführen, sollen diese Änderungen auch gleichzeitig durchgeführt werden können, gibt es zwei Möglichkeiten wie die Versionskontrolle dies erlauben kann. Bei der pessimistischen Variante wird gleichzeitiges und überschneidendes Ändern mittel Sperren verhindert. Will ein Entwickler ein Element schreiben, so muss er es sperren und alle anderen Entwickler haben dann nur mehr lesenden Zugriff auf das Element. In der Praxis skaliert diese Variante selbst bei kleinen Projektteams nicht sehr gut, da die Entwickler meist auf Sperren warten müssen. Die optimistische Variante erlaubt jedem Entwickler das Schreiben, ohne dass die Elemente vorher gesperrt werden müssen. Es besteht jedoch die Gefahr, dass es gleichzeitige oder überlappende Änderungen gibt und dadurch können Konflikte entstehen. Diese Konflikte müssen erkannt und gelöst werden, bevor in die Versionskontrolle geschrieben werden kann [KHvWH10].
Wenn ein kollaboratives System zur verteilten Modellierung über das Netzwerk erstellt werden soll, können parallele Änderungen und Konflikte bei der Modellierung auftreten. Eine parallele Änderung tritt auf, wenn mehrere Entwickler das gleiche Element oder miteinander verwandte Elemente gleichzeitig ändern. Diese Änderungen müssen noch nicht zwingend einen Konflikt hervorrufen, aber die Entwickler sollten benachrichtigt werden, denn durch weitere Aktionen und Änderungen könnten in späterer Folge doch noch Konflikte entstehen. Konflikte bei der Modellierung können durch Synchronisationsprobleme entstehen, also wenn das lokale und globale Modell noch nicht synchron ist und dazwischen eine Änderung eines anderen Entwicklers auftritt. Weiters können bei der Modellierung syntaktische Konflikte entstehen, indem dass die Entwickler zwar für sich syntaktisch korrekte Änderungen durchführen, diese Änderungen aber in Kombination nicht mehr

korrekt sind. Es können auch semantische Konflikte bei der Modellierung entstehen, diese können zwar durch Regeln entdeckt werden, benötigen aber meist zur Korrektur ein Eingreifen der Entwickler [BPE+10].

2.1.2 Werkzeuge zur kollaborativen Modellierung

Die ersten vorgestellten Ansätze zur kollaborativen Modellierung verwenden einen zentralen Server für das gemeinsame Modell. Die Arbeit von Bartelt et al. [BMS09] basiert auf der Kollaborationsplattform *Jazz* und die Modellierung orientiert sich am Prozess der Entwicklung. *Jazz* bietet neben der Versionskontrolle auch zusätzliche Unterstützung für Entwicklungsteams, wie zum Beispiel Änderungsverfolgung, Buildmanagement oder Testautomatisierung, an. Da die Plattform auf die Entwicklung von Quellcode ausgerichtet ist, wird von Bartelt et al. für gemeinsame Modellierung erweitert. Der Prozess für die Modellierung orientiert sich dabei an der Entwicklung für Quellcode, die einzelnen Entwickler arbeiten bei der Modellierung in ihren eigenen Zweigen in der Versionskontrolle. Für die Zusammenführung der einzelnen Zweige wird eine eigene Aufgabe, der Integrationstask, ausgeführt. Dabei werden die Konflikte beseitigt und die zusammengeführten Zweige können wieder in den Hauptentwicklungszweig oder einen anderen Zweig integriert werden. Die Lösung von Bang et al. [BPE+10] ist hingegen eine interaktivere Lösung. Sie präsentieren in ihrer Arbeit *CoDesign*, ein Softwaresystem für kollaborative Modellierung. Dabei wird das Modell in Echtzeit zwischen den einzelnen Entwicklern synchronisiert, es werden Konflikte erkannt und ein Teil dieser Konflikte kann auch automatisiert gelöst werden. Das System basiert auf einer Technologie, welche die Synchronisation des Modells und die Benachrichtigung über auftretende Konflikte übernimmt. Die Hauptaufgabe von *CoDesign* ist das Erkennen von Konflikten in der Modellierung und es bietet eine Schnittstelle an, damit weitere Algorithmen zur Erkennung von Inkonsistenzen hinzugefügt werden können.
Neben den zentralen Ansätzen zur Kollaboration in der Modellierung gibt es auch Vorschläge, wie die kollaborative Modellierung in einem Peer-to-Peer System funktionieren kann. Die Arbeit von Mougenot et al. [MBG09] schlägt einen Peer-to-Peer Ansatz vor, bei dem jeder Benutzer einen Teil des Modells lokal besitzt. Werden Modellelemente anderer Benutzer benötigt, können diese abonniert werden. Bei Änderungen der abonnierten Elemente werden Nachrichten mit den Änderungen zwischen den einzelnen Entwicklern gesendet. Die Erkennung und Auflösung von Konflikten erfolgt lokal nach jedem Empfang einer Änderung. Jeder der einzelnen Benutzer hat eine Partition des globalen Modells, das heißt das Gesamtmodell existiert nur virtuell. Das Modell wird als Sequenz von Konstruktionsoperationen gesehen und wenn zwei Operationen im Konflikt sind, dann wird die jüngere Operation behalten und die ältere Operation wird verworfen, ausgenommen sind ältere Löschoperationen. Die Arten der lösbaren Konflikte sind bei Mougenot et al. vorgegeben. Zum Beispiel führen verschiedene Werte für eine Eigenschaft, das Hinzufügen und Entfernen von Eigenschaftswerten oder verschiedene Werte von Referenzen zu Konflikten Die Arbeit von Michaux et al. [MBSS11] basiert auf den Ideen von Mougenot et al. [MBG09] und es werden die gleichen Arten von Konflikten bearbeitet. Die Arbeit verwendet jedoch ein anderes Peer-to-Peer System, bei dem es möglich ist, auch ohne Netzwerkverbindung zu arbeiten, da die Modelldaten bei jedem Benutzer repliziert sind. Sobald sich ein Benutzer wieder mit dem Netzwerk verbindet werden die Änderungen zu allen anderen Peers repliziert und es können Konflikte auftreten, welche gelöst werden müssen.

Es gibt auch Ansätze, bei denen Konflikte keine Rolle spielen, denn die Benutzer editieren das Modell direkt an einer zentralen Stelle. Bei Penichet et al. [PGTL08] wird dies mit dem Werkzeug *CE4WEB - Cooperative UML Editor for the Web* implementiert. Dazu wird ein gemeinsamer Server verwendet. Dieser Server ist für die Speicherung des Modells, die Verwaltung von Projekten und Modellversionen und für das Benutzermanagement verantwortlich. Der kollaborative UML-Editor ist in einem Webbrowser eingebettet und erlaubt das Editieren des Modells in Echtzeit, sowie das Kommunizieren mit den anderen Bearbeitern des Modells via eines Chats. In der vorgestellten Version kann *CE4WEB* nur Klassendiagramme verarbeiten. Das Werkzeug von Ling und Palaniappan [LP09] erlaubt es den Entwicklern in der Modellierung zusammenzuarbeiten, indem gemeinsam Klassen-, Anwendungsfall- und Sequenzdiagramme über ein Onlinewerkzeug erzeugt werden können. Die Diagramme werden nicht wie bei Penichet et al. [PGTL08] in Echtzeit editiert, sondern sie müssen als Liste von Tripel angegeben werden und daraus wird das Diagramm gerendert. Liu et al.[LZS+06] verzichten auf ein Onlinewerkzeug, sie versuchen stattdessen bestehende Programme transparent in kollaborative Programme umzuwandeln. Ihre Technik wird am Beispiel vom IBM Rational Sofware Architect demonstriert. Dabei werden die Modelldaten der einzelnen Programme ausgelesen, in ein allgemeines Modell transformiert und zentral gespeichert. Kommen neue Modelldaten in dieses zentrale Modell, werden sie wieder zurück transformiert und in die lokalen Modelle eingetragen, sodass alle Benutzer in Echtzeit an einem gemeinsamen Modell arbeiten können.
In dieser Masterarbeit wird das Modell auf einem zentralen Server gespeichert, die einzelnen Benutzer teilen sich einen gemeinsamen Teil des Modells und ihre Änderungen am gemeinsamen Modell werden in einem privaten Bereich abgelegt. Das heißt, für die einzelnen Benutzer hat es den Eindruck, als würden sie alleine am Modell arbeiten und Konflikte können dadurch noch nicht entstehen. Da das Hauptaugenmerk auf der Konsistenzprüfung der Modellbereiche liegt, ist es noch nicht notwendig, die privaten Bereiche in den öffentlichen Bereich zurückzuführen. Der Aufbau des Modells in zwei Bereichen erlaubt es aber, in zukünftigen Arbeiten neue Wege in der Kollaboration bei der Modellierung einzuschlagen. Welche Wege und Arten in der Zusammenarbeit dies sein könnten, wird zum Abschluss dieser Arbeit in Punkt 9.1 beschrieben.

2.2 Inkonsistenz

Noch bevor Inkonsistenz in Modellen erkannt und behoben werden kann, ist es wichtig zu verstehen, was sie ist und warum sie überhaupt auftreten kann. Spanoudakis und Zisman [SZ01] widmen sich in ihrer Arbeit genau diesen Fragen. Die Entwicklung komplexer Softwaresysteme ist auf verschiedene Rollen und Zuständigkeiten verteilt. Software Modelle beschreiben den Zustand von einem Softwaresystem aus verschiedenen Blickwinkeln, Level der Abstraktion, Feinheit der Beschreibung oder Formalität. Diese Beschreibungen sind meist in verschiedenen Notationen durchgeführt, werden von verschiedenen Personen erstellt und spiegeln die Interessen der einzelnen Beteiligten am Projekt wieder. Die Heterogenität der Beschreibung kann zu Inkonsistenz zwischen den einzelnen Modellen führen. Die Inkonsistenz entsteht, da sich die einzelnen Modelle überlappen, das heißt sie besitzen zwar gleiche Modellelemente, aber diese Elemente haben in den einzelnen Modellen eine andere Bedeutung und Ausprägung [SZ01]. In einem iterativen Entwicklungsprozess können auch Inkonsistenzen entstehen, da bei jeder Iteration neue und detaillierte Beschreibungen des Systems entstehen und das System somit einer ständigen Änderungen unterliegt [LMT09]. Die systematische Literaturrecherche von Lucas et at. [LMT09] im Bereich des Konsistenzmanagements hat die Inkonsistenz noch feiner kategorisiert. Bei der horizontalen Inkonsistenz ist die Konsistenz zwischen Modellen gleicher Abstraktion nicht gegeben, sie kann auch als Intra-Modell Inkonsistenz bezeichnet werden. Im Gegensatz dazu, ist bei der vertikalen oder auch Inter-Modell Inkonsistenz die Konsistenz verschiedener Revisionen des gleichen Modells verletzt. Eine weitere Form der Inkonsistenz ist die syntaktische Inkonsistenz. Dabei entspricht das Modell nicht seiner vorgegebenen Syntax, die durch das Metamodell spezifiziert ist. Als Letztes wird noch die semantische Inkonsistenz beschrieben, sie beschreibt Probleme der Konsistenz im Verhalten der Modelle [LMT09]. Die ursprüngliche Definition von Inkonsistenz in [SZ01] wäre also horizontale Inkonsistenz. In [SZ01] wird auch ein Prozess des Inkonsistenzmanagements beschrieben, also der Prozess wie Inkonsistenz behandelt wird, sodass die Ziele der Systementwicklung erreicht werden können. Wichtigster Teil des Prozesses für diese Masterarbeit ist das Erkennen von Inkonsistenzen in Software Modellen.

2.2.1 Erkennen von Inkonsistenz

Für die Konsistenzprüfung von Software Modellen gibt es in der Literatur viele verschiedene Ansätze, wie Algorithmen und Werkzeuge zur Erkennung der Inkonsistenzen aufgebaut werden können.

Der wichtigste Ansatz der Konsistenzprüfung für diese Masterarbeit wurde von Egyed [Egy06] für UML 1.3 entwickelt und in weitere Folge in [Egy11] verallgemeinert. Es handelt sich um einen automatisierten Ansatz zum Erkennen und Verfolgen von Inkonsistenzen in Echtzeit, also sobald sich das Modell ändert. Inkrementelle Konsistenzprüfung heißt, das im Gegensatz zu einer vollständigen Konsistenzprüfung, nur die betroffenen Konsistenzregeln nach jeder Modelländerung neu evaluiert werden und nicht alle Regeln. Inkrementelle Konsistenzprüfung ermöglicht eine unmittelbare Rückmeldung der Konsistenz des Modells an den Entwickler. Da die inkrementelle Konsistenzprüfung aus [Egy11] und die daraus resultierenden Werkzeuge [RE10] die Basis dieser Arbeit bilden, wird im Kapitel 3 in Punkt 3.1 detaillierter darauf eingegangen.

Blanc et al. [BMMM08] erkennen Inkonsistenzen durch die Analyse der Operationen bei der Modellerstellung. Dabei werden die Modelle nicht aus einer Menge von Modellelemen-

ten aufgebaut, sondern aus einer Sequenz von elementaren Konstruktionsoperationen, wie zum Beispiel den Operationen zum Erstellen oder Löschen von einem Modellelementen, sowie dem Setzen von Eigenschaften und Referenzen. Die Analyse der Sequenz erlaubt das Erkennen von struktureller und methodischer Inkonsistenz. Strukturelle Konsistenzregeln definieren, unabhängig davon wie das Modell erstellt wurde, welche Beziehung die Modellelemente zueinander haben sollen. Die methodischen Konsistenzregeln sind Regeln über die Konstruktionssequenz und der Reihenfolge der Operationen. Zur Definition der Konsistenzregeln wird Prädikatenlogik verwendet und die Regeln werden mit Prolog ausgewertet. Das System erstellt aus einem Modell die Konstruktionssequenz, speichert diese in die Prolog Wissensbasis, evaluiert die Regeln und präsentiert das Ergebnis dem Benutzer. Blanc et al. [BMMM09] erweitern ihren eigenen Ansatz auch zu einer inkrementellen Konsistenzprüfung. Diese inkrementelle Konsistenzprüfung basiert auch auf der Sequenz der Konstruktionsoperationen. Nach einer Änderung wird untersucht, welche Regeln neu evaluiert werden müssen, dazu wird eine sogenannte *Impact Matrix* verwendet. Diese Matrix muss im Vorhinein definiert werden, sie beinhaltet welche Konsistenzregeln durch welche Konstruktionsoperationen des Metamodells bei einer Änderung betroffen sind. Bei einer Änderung am Modell wird die konkrete Konstruktionsoperation auf ihre äquivalente Operation im Metamodell übersetzt, so können alle betroffenen Regeln ermittelt und bei Bedarf neu evaluiert werden.
Eine weitere Technik zur Konsistenzprüfung wurde von Nentwich et al. [NCEF02] präsentiert. Es handelt sich um *xlinkit*, eine Technologie zur Konsistenzprüfung von XML-Dokumenten. Sie basiert auf offenen Internettechnologien, wie zum Beispiel XML, XPath und XLink. Da Software Modelle auch in XML gespeichert werden können, kann *xlinkit* somit auch zur Konsistenzprüfung von Modellen verwendet werden. Bei der Implementierung wurde ein Webservice erstellt. Es werden Dokumente und Regeln zu dem Service gesendet, dort führt *xlinkit* die Konsistenzprüfung durch und das Ergebnis ist ein Dokument mit der Information welche Elemente in den Dokumenten welche Konsistenzregeln erfüllen oder verletzen. Für die Definition der Konsistenzregeln wird eine adaptierte Prädikatenlogik verwendet, die Elemente der Mengen werden aus den XML-Dokumenten mittels XPath ermittelt. Auch bei diesem Ansatz wird in einer Folgearbeit der Autoren Nentwich et al. [NEFE03] eine inkrementelle Variante der Konsistenzprüfung vorgestellt. Dabei wird dem System zusätzlich das Dokument mit dem bisherigen Konsistenzinformationen und die Änderungen der zu untersuchenden Dokumente übermittelt, daraufhin aktualisiert *xlinkit* das Dokumenten mit den Konsistenzinformationen.

2.2.2 Beheben von Inkonsistenz

Nachdem Inkonsistenzen in Modellen gefunden worden sind, stellt sich die Frage, wie die gefundenen Probleme in den Modellen weiter behandelt werden. Das Beheben von Inkonsistenz ist zwar für diese Masterarbeit nicht relevant, aber für zukünftige Arbeiten im Bereich der Konsistenzprüfung in einer Mehrbenutzerumgebung kann es eine sinnvolle Erweiterung sein.
Die Lösung zur Behebung von Inkonsistenz von Egyed [Egy07a] führt die Arbeit über die inkrementelle Konsistenzprüfung aus [Egy06] fort. Zur Behebung von Inkonsistenzen werden dem Benutzer Vorschläge präsentiert, wie die Inkonsistenz repariert werden kann. Zusätzlich werden auch die möglichen Seiteneffekte einer Reparatur aufgezeigt. Die existierenden Inkonsistenzen werden bewusst nicht automatisch repariert, sondern der Benutzer muss eine Auswahl treffen, die dem gewünschten Design des Modells entspricht. Die Ar-

beit stellt fest, dass Inkonsistenzen auch von einander abhängen können und dadurch sind komplexere Lösungen zur Behebung notwendig. Die Lösungsvorschläge werden aus dem Scope einer Regel ermittelt.
Die Arbeit von Mens und van der Straeten [MVDS07] beschreibt den iterativen und inkrementellen Prozess *SIRP - Simple Iterative Resolution Process* zum Beheben von Inkonsistenz in Modellen. In diesem Prozess werden als erstes Inkonsistenzen gesucht, danach Lösungen vorgeschlagen und zum Schluss werden die Lösungen am Modell angewendet. Dieser Ansatz basiert auf dem Graphersetzungssystem (graph transformation), das heißt das Modell wird als Graph dargestellt und eine Konsistenzregel entspricht einer gewünschten Struktur des Graphs. Das System ermittelt zum Beheben einer Inkonsistenz alle benötigten Operationen am Graphen.
Xiong et al. [XHZ+09] präsentieren in ihrer Arbeit einen Ansatz zum automatisierten Beheben von Inkonsistenzen. Sie führen dazu die Sprache *Beanbag* ein. Diese Sprache basiert auf OCL und erlaubt das Definieren von Prozeduren zum Beheben von Inkonsistenzen. Bei einem *Beanbag* Programm wird also einerseits die Konsistenz von einem Modell geprüft und andererseits werden bei Inkonsistenzen automatisch Änderungen generiert, damit das Modell wieder konsistent ist.

2.2.3 Mehrbenutzerumgebungen

In der Literatur gibt es bereits Arbeiten darüber, wie die Konsistenzprüfung und die Kollaboration zusammengeführt werden können. Diese Ansätze setzen sich somit auch mit dem gleichen Thema wie diese Masterarbeit auseinander, schlagen jedoch unterschiedliche Richtungen in den vorgeschlagenen Konzepten ein.
Die erste Arbeit zu diesem Thema ist von Nentwich et al. [NEF03] und basiert auf deren Technologie *xlinkit* [NCEF02, NEFE03]. Das Hauptthema der Arbeit ist eigentlich das Erzeugen von Reparaturaktionen zum Beheben von Inkonsistenz, sie würde daher auch sehr gut in den vorangegangen Punkt passen. Sie ist aber hier aufgeführt, da diese Reparaturaktionen bei einer zentralen Stelle generiert werden. Das heißt es werden Dokumente und Konsistenzregeln zu einem sogenannten Reparaturadministrator geladen, dieser prüft die Konsistenz und erzeugt die Reparaturaktionen für die Benutzer. Genau wie Nentwich et al. verwendet diese Masterarbeit eine zentrale Stelle zur Konsistenzprüfung. Hauptunterschied ist aber die inkrementelle Konsistenzprüfung und somit auch die Interaktivität des Gesamtsystems. Bei Nentwich et al. werden Dokumente zu einem Server geladen und dort vom Reparaturadministrator verarbeitet. Bei dem Ansatz dieser Masterarbeit werden hingegen nur Änderungen zum Server geladen, dort automatisiert geprüft und das Ergebnis ist für den Benutzer unmittelbar verfügbar.
Auch bei Moutenot et al. [MBG10] werden bereits von ihnen vorgestellte Arbeiten kombiniert, konkret werden die Mechanismen der Konsistenzprüfung [BMMM08, BMMM09] mit dem Peer-to-Peer Protokoll zur Modellierung [MBG09] kombiniert. Jeder Entwickler hat lokal seine eigene Sicht auf das Gesamtmodell gespeichert. Wird ein Modellelement eines anderen Entwicklers benötigt oder darauf referenziert, wird dieses Element mittels des Peer-to-Peer Protokolls nachgeladen, sodass es auch lokal gespeichert ist. Konsistenzprüfungen werden auf der lokalen Sicht der einzelnen Benutzer durchgeführt. Wenn nun Konsistenzregeln ausgeführt werden kann es passieren, dass Modellelemente von anderen Entwicklern und deren Sichten benötigt werden. Wenn dieser Fall auftritt, wird mit Hilfe des Peer-to-Peer Protokolls zu der benötigten Information gesprungen, denn diese Informationen müssen nicht unbedingt lokal gespeichert werden. Die betroffenen Model-

lelemente einer Regel sind also eine Kombination aus lokalen Elementen und Sprüngen auf Elemente von anderen Sichten. Im Vergleich zu Moutenot et al. verwendet diese Masterarbeit einen zentralen Server und auch die Konsistenzprüfung läuft zentral auf diesem Server. Es gibt für jeden Benutzer eine eigene Konsistenzprüfung am Server, bei dem Ansatz in dieser Arbeit wird jedoch die Replikation von Regeln vermieden, wie sie bei Moutenot et al. hingegen entsteht.
Eine weitere Lösung zur Erkennung von Inkonsistenz wird von Bartelt und Schindler [BS10] vorgestellt und basiert auf der bereits beschriebenen Technologie [BMS09] zur Kollaboration. Dabei wird zum Integrationstask eine Konsistenzprüfung hinzugefügt. Wenn also zwei Zweige in der Versionskontrolle zu einem Zweig zusammengefügt werden, wird das resultierende Modell geprüft, ob es Inkonsistenzen enthält. Genau wie diese Masterarbeit werden die Modelle zentral gespeichert und es hat auch jeder Benutzer seine eigene Sicht auf das Modell. Während bei Bartelt und Schindler jeder Benutzer über ein eigenes Modell verfügt, speichert der Ansatz dieser Masterarbeit für jeden Benutzer nur seine Änderungen. Zusätzlich kommt bei der Lösung von Bartelt und Schindler keine inkrementelle Konsistenzprüfung zum Einsatz. Das heißt, wenn zwei Zweige zusammengeführt werden, wird das gesamte resultierende Modell geprüft.
Sabetzadeh et al. [SNEC08] stellen mit *TReMer+ - Tool for Relationship Driven Model Mergring* ein Werkzeug zur Konsistenzprüfung von verteilten Modellen vor. Das Werkzeug vereinigt verteilte Modelle zu einem Gesamtmodell und führt auf diesem Gesamtmodell eine Konsistenzprüfung durch. Bei verteilten Modellen müsste normalerweise eine Inter-Modell Konsistenzprüfung durchgeführt werden, durch das Vereinigen der verteilten Modelle kann mit *TReMer+* aber eine Intra-Modell Konsistenzprüfung durchgeführt werden und diese erlaubt einfachere Konsistenzregeln [SNEC08]. Genau wie Sabetzadeh et al. wird bei dieser Masterarbeit eine Intra-Modell Konsistenzprüfung durchgeführt, jedoch müssen die Modelle nicht vorher vereinigt werden, da sie bereits vollständig am Server vorhanden sind. Ein weitere Nachteil von Sabetzadeh et al. ist, dass keine inkrementelle Konsistenzprüfung eingesetzt wird. Es müssen die geänderten Modelle für jede Konsistenzprüfung wieder zu einem neuen Gesamtmodell vereinigt werden und auf diesem Gesamtmodell wird wieder eine komplette Konsistenzprüfung durchgeführt, bei dieser Masterarbeit wird hingegen nur eine Konsistenzprüfung auf den geänderten Modellelementen durchgeführt.

Kapitel 3

Technologie

Für die Implementierung in Kapitel 6 wird eine Reihe von Technologien und Bibliotheken benötigt. In diesem Kapitel werden theoretischen Grundlagen und allgemeine Beispiele für die verwendeten Technologien beschrieben. Es handelt sich dabei um die Konsistenzprüfung, RDF und die Kommunikation über HTTP.

3.1 Konsistenzprüfung

3.1.1 Model Analyzer

Die Konsistenzprüfung wird in dieser Masterarbeit nicht neu implementiert, sondern es wird ein bestehendes System auf die Gegebenheiten der Mehrbenutzerumgebung angepasst. Der Model Analyzer ist eine Technologie zum Erkennen von Inkonsistenzen in Software Design Modellen. Die Grundlage für den Model Analyzer bildet die Arbeit von Egyed [Egy11] und die Beschreibungen im Abschnitt über die Grundlagen sind aus diesem Paper entnommen. Die eingeführte und verwendete Terminologie wird für die gesamte Masterarbeit verwendet.

Grundlagen

Der Model Analyzer ist eine inkrementelle Konsistenzprüfung und wenn sich das Modell ändert, wird automatisch entschieden welche Konsistenzregeln neu evaluiert werden müssen. Dies ist möglich, da sich das System die Modellelemente merkt, auf welche Modellelemente bei der Evaluierung einer Konsistenzregel zugegriffen wurde.
Ein Modell besteht aus Modellelementen und diese Modellelemente sind Instanzen der Elemente eines Metamodells. Ein Beispiel für ein UML-Metaelement ist die Nachricht und im Beispielmodell aus Abbildung 3.1 sind *activate* und *deactivate* zwei Instanzen dieses Metaelements. Eine Konsistenzregel *ConsistencyRule* wird ausgehend von einem Element des Metamodells *MetaModelElements* evaluiert, dem sogenannten Kontextelement *ContextElement*. Dies erleichtert das Definieren und die Wartung der Konsistenzregeln, denn sie sind somit unabhängig von einem konkreten Modell. Eine Konsistenzregel benötigt auch eine zu evaluierende Bedingung *Condition* und liefert als Ergebnis wahr oder falsch, je nachdem ob die Bedingung erfüllt ist oder nicht.

$$\textit{ConsistencyRule} = <\text{Condition, ContextElement}> \rightarrow \text{Bool where ContextElement} \in \text{MetaModelElements}$$

Das Ergebnis einer Konsistenzregel kann sich durch Änderungen am Kontextelement verändern. Auch bei Änderungen von anderen Modellelementen, die nicht explizit angeführt sind, kann sich das Ergebnis der Konsistenzregel verändern. Die Herausforderung bei der inkrementellen Konsistenzprüfung ist es, alle Modellelemente zu finden, welche sich auf das Ergebnis einer Konsistenzregel auswirken. Das Ergebnis einer Konsistenzregel kann sich nur ändern, wenn sich das Modell ändert, deshalb muss die Konsistenzprüfung die Details der Änderungen am Modell kennen. Normalerweise löst eine Änderung durch den Benutzer eine Reihe von Änderungen am Modell *ChangeGroup* aus. Ein Modellelement ist üblicherweise eine Ansammlung von Feldern, daher ist eine Änderung *Change* die Änderung des Werts von einem Feld *Field* eines Modellelements *ModelElement*.

$$\textit{ChangeGroup} = \text{Set of } <\text{Change}>$$
$$\textit{Change} = <\text{ModelElement, Field}>\text{: oldValue} \rightarrow \text{newValue}$$

Eine Konsistenzregel muss für alle Instanzen eines Metaelements ausgeführt werden. Um bei einer Änderung nicht ganze Konsistenzregeln neu evaluieren zu müssen, wird eine feinere Unterscheidung getroffen, dies sind Konsistenzregelinstanzen oder auch Regelinstanzen *CRI - ConsistencyRuleInstance*. Sie sind definiert durch die Konsistenzregel *ConsistencyRule* und das Modellelement *ModelElement*. Bezogen auf das Beispielmodell aus Abbildung 3.1 und eine Konsistenzregel mit der UML-Nachricht als Kontextelement bedeutet dies, dass es für die Nachrichten *activate* und *deactivate* je eine Regelinstanz gibt.

$$\textit{CRI} = <\text{ConsistencyRule, ModelElement}> \text{ where ModelElement instanceof ContextElement(ConsistencyRule)}$$

Jede Regelinstanz startet die Evaluierung bei einer anderen Instanz des Kontextelements und sie greifen auf unterschiedliche Modellelemente zu, deshalb können die einzelnen Regelinstanzen verschiedene Ergebnisse liefern. Die einzelnen Regelinstanzen können unterschiedlich durch Änderungen betroffen sein, deshalb wird bei einer Änderung der Scope jeder Regelinstanz untersucht. Der Scope einer Regelinstanz *Scope(CRI)* besteht aus allen Elementen, auf die während der Evaluierung zugegriffen worden ist. Wenn sich ein Modellelement ändert, müssen daher jene Regelinstanzen *AffectedCRIs(Change)* neu evaluiert werden, welche das geänderte Modellelement in ihrem Scope haben.

$$\textit{Scope(CRI)} = \text{Set of } <\text{ModelElement, Field}> \text{ pairs accessed during Evaluation of CRI}$$
$$\textit{AffectedCRIs(Change)} = \{\text{CRI} \in \text{CRIs} \mid \text{CRI.Scope contains Change}\}$$

Regelinstanzen hängen von einem Modellelement ab, das heißt wird ein neues Modellelement erstellt, wird auch eine neue Regelinstanz erstellt, sofern eine Konsistenzregel mit dem geeignetem Kontextelement vorhanden ist. Wenn ein Modellelement gelöscht wird, werden auch alle Regelinstanzen gelöscht, welche von diesem Modellelement abhängen. Sobald eine Regelinstanz neu evaluiert wird, wird auch ihr Scope neu aufgebaut, denn der Scope kann sich durch Änderungen am Modell laufend ändern.

Model Analyzer Werkzeug

Das Werkzeug für die Konsistenzprüfung ist, genau wie die Basistechnologie, ständig weiterentwickelt worden. Eine der ersten Versionen des Model Analyzer ist in der Arbeit [Egy07b] beschrieben und wurde in IBM Rational Rose integriert. Die aktuelle Version

[RE10] ist ein Plug-in für den Eclipse-basierten IBM Rational Software Modeler (RSM). Der RSM ist ein Modellierungswerkzeug zum Entwurf von EMF-basierten [Ecl11c] UML-Modellen. Mit dem Model Analyzer Werkzeug können Entwickler beliebige Konsistenzregeln definieren und bekommen unmittelbares Feedback über den Konsistenzzustand des UML-Modells. Es können Konsistenzregeln in OCL und Java definiert werden. OCL-Regeln können zur Laufzeit hinzugefügt, geändert und gelöscht werden, während Java-Regeln mit dem Werkzeug kompiliert werden müssen. Die Hauptaufgabe des Model Analyzers sind das Erkennen von Inkonsistenzen in UML-Modellen und das Erzeugen von Reparaturaktionen für das Beheben der entdeckten Inkonsistenzen. Die Konsistenzprüfung, die Visualisierung und das Erzeugen der Reparaturaktionen wird inkrementell, also während das Modell bearbeitet wird, durchgeführt [RE10].
Um die Technologie auf andere Modellierungssprachen zu portieren, müssen drei Anforderungen beachtet werden [Egy11]:

- Änderungsbenachrichtigungen: Um auf Modelländerungen reagieren zu können, müssen Änderungen am Modell durch das Konsistenzwerkzeug erkennbar sein.
- Model Profiler: Um die Konsistenzregeln evaluieren zu können, benötigt die Konsistenzprüfung die Möglichkeit eines Zugriffs auf die Modellelemente des Modells.
- Konsistenzregeln: Die verschiedenen Modellierungssprachen können spezielle Konsistenzregeln benötigten, dies ist aber kein Problem da der Ansatz für die Konsistenzprüfung keine besonderen Anforderungen an die Konsistenzregeln stellt.

Die aktuelle Version [RE10] des Model Analyzers ist der Ausgangspunkt für die Konsistenzprüfung in einer Mehrbenutzerumgebung und es müssen die drei vorgestellten Anforderungen erfüllt werden. Änderungen werden am Client festgestellt und darauf hin zum Server gesendet. Dort müssen sie in ein Modell eingetragen werden und die Konsistenzprüfung muss Zugriff auf die Modellelemente bekommen. Die Definition der Konsistenzregeln kann aus der aktuellen Version des Model Analyzer übernommen werden.

Model Analyzer Framework

Basis für die Implementierung der angepassten Konsistenzprüfung für die Mehrbenutzerumgebung ist das Model Analyzer Framework. Das Framework wird im Zuge der Doktorarbeit von Reder [Red11] entwickelt. Die Doktorarbeit beschäftigt sich mit dem Entwurf eines Inkonsisitenzmanagement Frameworks für beliebige Modellierungssprachen und Konsistenzregeln. Ein Teil der Doktorarbeit ist die Entwicklung einer inkrementellen Konsistenzprüfung auf Basis des Model Analyzer Ansatzes [Egy07b] und die Analyse von Konsistenzregeln mit deren Syntax- und Evaluierungsbäumen. Ein weiterer Teil beschäftigt sich mit dem Erzeugen von Reparaturaktionen sowie deren Seiteneffekten. Das Framework definiert alle benötigten Interfaces für die Implementierung einer Konsistenzprüfung und es enthält zusätzlich abstrakte Implementierungen der Interfaces, welche den Ablauf der Konsistenzprüfung bereits vorgeben.
Die zentrale Datenstruktur des Frameworks ist das Interface *DataModel<CT, ET, PT>*. Mit dieser Datenstruktur werden die Konsistenzregeln, Regelinstanzen und Scopeelemente verwaltet. Das Framework ist generisch gehalten, mit *CT - ContextType* wird das Kontextelement definiert, *ET - ElementType* definiert die Modellelemente und *PT- PropertyType* legt den Typ der Eigenschaften der Modellelement fest. Diese drei generischen Typen sind vom Modell abhängig, bei der EMF-basierten Implementierung für den RSM ist das

Kontextelement jede Subklasse von *Element*, die Modellelemente sind vom Typ *Element* und die Eigenschaften sind vom Typ *EStructuralFeature*. Die abstrakte Implementierung von *DataModel* ist die Klasse *AbstractDataModel<CT, ET, PT>*. Sie definiert Listen und Zugriffsmethoden für alle Daten. Zusätzlich wird der Ablauf der Evaluierung der Konsistenzregeln und deren Regelinstanzen vorgegeben. Die Evaluierung wird mit einer Liste von Änderungen angestoßen. Die Änderungen sind im Interface *Change<CT, ET, PT>* definiert und müssen mindestens Kontextelement, Modellelement, Eigenschaft und Änderungstyp zu Verfügung stellen. Eine weitere Aufgabe der Klasse *AbstractDataModel* ist die Verwaltung von Listener. Das Framework bietet Listener für Änderungen der Konsistenzregeln, Regelinstanzen, Scopeelemente und Ausdrücke an.

Die Verwaltung der Scopeelemente wird mit einer eigenen Datenstruktur realisiert, da die einzelne Scopeelemente in mehreren Scopes vorkommen können und Duplikate vermieden werden sollen. Das Interface *ScopeElementRepository<ET, PT>* verwaltet die Scopeelemente, diese sind durch das Interface *ScopeElement<ET, PT>* definiert und müssen neben dem Modellelement und der Eigenschaft auch den konkreten Wert der Eigenschaft ermitteln können. Die abstrakte Implementierung *AbstractScopeElementRepository<ET, PT>* des Interfaces *ScopeElementRepository* verwaltet die Liste und Zugriffsmethoden der Scopeelemente.

Konsistenzregeln werden durch das Interface *DesignRule<CT, ET, PT>* definiert und es werden einige Attribute, wie zum Beispiel der Name, eine Beschreibung oder die Bedingung vorgegeben. Zusätzlich wird eine Methode *generateInstance(ET context)* vorgegeben, mit der eine Regelinstanz der Konsistenzregel für ein angegebenes Modellelement erzeugt werden soll. Regelinstanzen sind durch das Interface *Instance<CT, ET, PT>* definiert und erlauben zum Beispiel den Zugriff auf das Ergebnis der Evaluierung oder auf den Scope der Regelinstanz. Die Bedingungen der Konsistenzregeln werden, neben der textuellen Darstellung, auch als Baum von Ausdrücken *Expressions* gespeichert. Dieser Baum von Ausdrücken ist in der Sprache *ARL - Abstrace Rule Language* definiert und diese ist ein Teil des Frameworks. In den Blättern des Baums sind Ausdrücke, welche die Scopeelemente beinhalten und so kann bei der Auswertung auf das Modell zugegriffen werden.

3.1.2 Beispielmodell

In einigen Kapiteln wird ein Modell für Beispiele benötigt. Das Modell in Abbildung 3.1 wird am Institut in diversen Publikationen verwendet und es enthält bereits Inkonsistenzen. Es beschreibt einen einfachen Lichtschalter in UML.

Das Klassendiagramm (oben) repräsentiert die Struktur des Lichtschalters. Ein Lichtschalter *Switch* wird für das Ein- und Ausschalten des Lichts *Light* verwendet und speichert es im Attribut *light*. Die Klasse *Light* verfügt über die Methoden *turn-on* zum Einschalten und *deactivate* zum Ausschalten des Lichts. Für die Klasse *Light* ist auch ein Zustandsdiagramm *StateChart* (links, unten) definiert. Wenn das Licht im Zustand *Off* ist, kann es mit der Methode *activate* eingeschaltet werden und kommt in den Zustand *On*, mit der Methode *deactivate* wird es wieder ausgeschaltet. Das Sequenzdiagramm *Switch the light* (rechts, unten) beschreibt, dass Licht nur dann deaktiviert *deactivate* werden kann, nach dem es bereits aktiviert *activate* wurde.

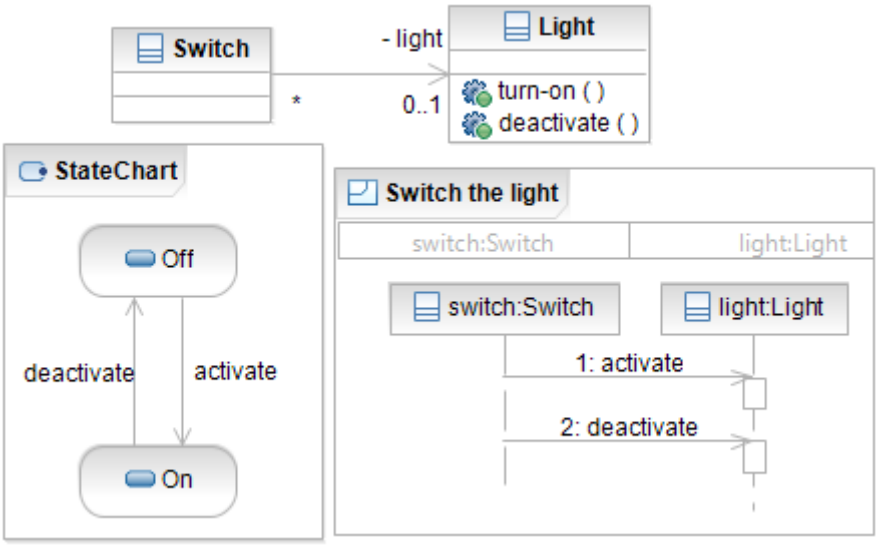


Abbildung 3.1: Beispielmodell

3.1.3 Beispielregeln

Für einige Kapitel werden auch Konsistenzregeln benötigt. Für die Evaluierung oder bei der Beschreibung der Benutzerschnittstellte wird eine Liste von 18 Konsistenzregeln verwendet. Aus Gründen der Lesbarkeit ist die Gesamtliste im Anhang A aufgelistet, in der Gesamtliste sind die Konsistenzregeln bereits im OCL-Format, wie sie der Model Analyzer benötigt. Regeln diese Liste werden auch in Publikationen des Instituts verwendet. An dieser Stelle wird eine Regel der Gesamtliste vorgestellt und es wird gezeigt, welche Ergebnisse sie liefert, wenn sie auf das Beispielmodell angewendet wird. Konsistenzregel *DR03* sagt aus, dass eine Nachricht in einem Sequenzdiagramm auch als Operation im Klassendiagramm der Empfängerklasse definiert sein muss. Die Bedingung der Konsistenzregel ist in OCL ausgedrückt und das Kontextelement ist eine Nachricht.

$$DR03 = \langle \text{self.receiveEvent.covered} \rightarrow \text{forAll(Lifeline l |} \\ \text{l.represents.type.ownedOperation} \rightarrow \text{exists(Operation o | o.name} = \text{self.name))}, \\ \text{Message} \rangle \rightarrow \text{Bool where Message} \in \text{UML-MetaModelElements}$$

Wird diese Konsistenzregel mit auf das Modell aus dem vorangegangen Punkt angewendet, werden zwei Regelinstanzen erzeugt, *CRI01* für die Nachricht *activate* und *CRI02* für die Nachricht *deactivate*.

$$CRI01 = \langle \text{DR03, activate} \rangle \text{ where activate instanceof ContextElement(DR03)}$$
$$CRI02 = \langle \text{DR03, deactivate} \rangle \text{ where deactivate instanceof ContextElement(DR03)}$$

Werden nun beide Regelinstanzen ausgewertet, wird ihr Scope aufgebaut. Bei der Evaluierung wird auch auf die Methoden der Klasse *Light* zugegriffen. Für die Instanz *CRI02* wird die Methode *deactivate* in der Klasse *Light* gefunden, die Evaluierung der Regelinstanz liefert somit ein positives Ergebnis und die Regelinstanz ist konsistent. Bei der Regelinstanz *CRI01* kann keine entsprechende Methode in der Klasse *Light* gefunden werden und die Evaluierung schlägt fehl, die Regelinstanz ist inkonsistent. Im Scope der beiden Regelinstanzen sind neben anderen Elementen auch die beiden Methoden der Klasse *Light*. Wird zum Beispiel die Methode *turn-on* in *activate* umbenannt, würden beide Regelinstanzen neu evaluiert werden müssen und darauf hin wären beide Regelinstanzen konsistent.

3.2 RDF und Jena

Das World Wide Web ist darauf ausgelegt, dass ein menschlicher Benutzer die zu Verfügung gestellten Informationen und deren Bedeutung problemlos erfassen kann. Maschinen können dies üblicherweise nicht. Damit die Informationen auch für Maschinen verarbeitbar werden, wurde die Idee des Semantic Web entwickelt. Beim Semantic Web wird Information von Vorhinein so zu Verfügung gestellt, dass sie auch für Maschinen nutzbar wird. Um die Informationen zu beschreiben, sind Standards notwendig. Einer dieser Standards ist RDF (Resource Description Framework) vom W3C (World Wide Web Consortium) [HKRS08]. RDF ist eine Sprache zur Repräsentation von Informationen über Ressourcen des World Wide Web. RDF kann generell zur Repräsentation von Informationen verwendet werden und ist dafür gedacht, Informationen mit Programmen zu bearbeiten. RDF wird auch zum Austausch von Informationen zwischen Programmen verwendet. Die Grundidee des Aufbaus ist, dass Objekte eindeutig mit URIs (Uniform Resource Identifiers) identifiziert werden können und dass Ressourcen mit verschiedenen Eigenschaften und Werten beschrieben werden. Dieser Aufbau ermöglicht es, RDF als einfachen Graphen aus Statements aufzubauen [MM04].

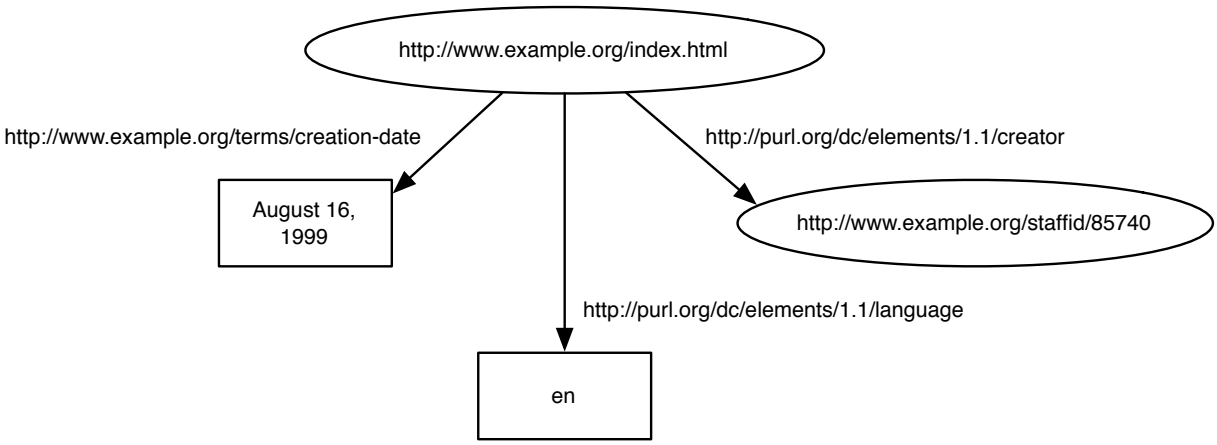


Abbildung 3.2: RDF als Graph [MM04]

Die Struktur von RDF-Ausdrücken kann als eine Liste von Tripel oder Statements angesehen werden. Jedes der Tripel besteht aus Subjekt, Prädikat und Objekt. Eine Menge solcher Tripel wird als RDF-Graph bezeichnet. Abbildung 3.2 zeigt einen illustrierten RDF-Graph, es entsteht ein Diagram mit Knoten und gerichteten Pfeilen. Ein Knoten im Graph ist entweder ein Subjekt oder Objekt, ein Pfeil stellt das Prädikat dar und endet immer bei einem Objekt. Ein Objekt kann ein einfacher Wert oder wieder eine URI auf eine Ressource sein. Das Objekt kann somit selbst zum Subjekt eines anderen Tripel werden [KC04]. RDF bietet auch eine XML-basierende Syntax (RDF/XML) an [MM04] und für den RDF-Graph aus Abbildung 3.2 ist die zugehörige XML-Notation in Quellcode 3.1 zu sehen. Damit in der gesamten XML-Datei nicht die kompletten URIs stehen, werden Namespaces verwendet.

```
1 <?xml version="1.0"?>
2 <rdf:RDF
3   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4   xmlns:dc="http://purl.org/dc/elements/1.1"
5   xmlns:terms="http://www.example.org/terms">
6   <rdf:Description rdf:about="http://www.example.org/index.html">
```

```
7       <terms:creation-date>AUgust 16, 1999</terms:creation-date>
8       <dc:creator rdf:resource="http://www.example.org/staffid/85740"/>
9       <dc:language>en</dc:language>
10    </rdf:Description>
11  </rdf:RDF>
```

Quellcode 3.1: RDF als XML

Unabhängig von den verschiedenen Darstellungsarten wird im Beispiel die Ressource *index.html* beschrieben. Sie wurde am 16. August 1999 erstellt und ist in englischer Sprache verfasst. Eine weitere Eigenschaft der Ressource ist der Ersteller, der Wert dieser Eigenschaft ist eine weitere Ressource. Als letzte Darstellungsart ist in Tabelle 3.1 die Darstellung als Statements, beziehungsweise in Tripel zu sehen.

Subjekt	Prädikat	Objekt
http://.../index.html	terms:creation-date	August 16, 1999
http://.../index.html	dc:language	en
http://.../index.html	dc:creator	http://.../staffid/85740

Tabelle 3.1: RDF als Tripel

Jena

Jena ist ein Java Framework für RDF und weiteren Technologien des semantischen Webs, es wurde ursprünglich von HP entwickelt und ist mittlerweile als Open Source Software erhältlich [Jen11a]. Für diese Arbeit ist vor allem die API für RDF interessant und wird nachfolgend mit Hilfe der Dokumentation der Projektseite [Jen11b] genauer beschrieben. Die weiteren Funktionen von Jena können der Dokumentation entnommen werden und in den beiden Papers [McB01] und [CDD+04] wird von den Mitarbeitern von HP beschrieben, wie Jena die RDF-Spezifikationen genau implementiert. Im folgenden Quellcode 3.2 ist der Java-Code zur Erstellung des RDF aus dem vorangegangenen Beispiel (Abbildung 3.2) zu sehen.

```
1  String indexURI = "http://www.example.org/index.html";
2  String creatorURI = "http://www.example.org/staffid/85740";
3  String dateProperty = "http://www.example.org/terms/creation-date";
4  Model model = ModelFactory.createDefaultModel();
5
6  Resource index = model.createResource(indexURI);
7  index.addProperty(DC.language, "en");
8  index.addProperty(DC.creator, model.createResource(creatorURI));
9  index.addProperty(model.createProperty(dateProperty),"August 16, 1999");
10
11 StmtIterator iter = model.listStatements();
12 while (iter.hasNext()) {
13   Statement stmt      = iter.nextStatement();
14   System.out.println(stmt);
15 }
16
17 model.write(System.out)
```

Quellcode 3.2: RDF mit Jena

Quellcode 3.2 beginnt mit der Definition der benötigten URIs und der Initialisierung eines leeren Modells *model*. Zur Erstellung eines speicherbasierten Modells wird die Methode *createDefaultModel()* der *ModelFactory* verwendet. In Jena gibt es noch weitere Implementierungen von Modellen, wie zum Beispiel die eines Modells mit einer relationalen Datenbank im Hintergrund. Im nächsten Schritt wird die Ressource *index* erstellt. Zu dieser Ressource werden die einzelnen Eigenschaften hinzugefügt. Eigenschaften werden entweder über Klassen mit Konstanten oder dynamisch mit der Funktion *createProperty()* erzeugt. Im Beispiel werden die Eigenschaften der Klasse *DC* (Dublin Core) verwendet. Jena bietet noch weitere Klassen von Eigenschaften für die bekanntesten Schemas an, zum Beispiel für RDF, VCARD oder den hier verwendeten DC. Wie bereits beschrieben, ist ein RDF-Graph eine Liste von Statements. In Jena können alle Statements mit einem Statement-Iterator durchgelaufen werden, dieser kann vom Model mittels *listStatements()* abgerufen werden. Bei den einzelnen Statements kann durch Methoden auf Subjekt, Prädikat und Objekt zugegriffen werden. Jena bietet auch Methoden zum Lesen und Schreiben von RDF an. Im Beispiel wird mit der Methode *write()* das Modell auf die Konsole ausgegeben. Die Methode verwendet standardmäßig RDF/XML zur Ausgabe und es entsteht ein XML wie in Quellcode 3.1. Die Methode zum Schreiben kann aber auch Tripel oder andere Darstellungsformen ausgeben. Modelle können mit einer *read()*-Methode wieder von einem beliebigen Stream eingelesen werden. Der Aufbau und die Beschreibung des Quellcodes sind eine Zusammenfassung der Tutorials der Dokumentation [Jen11b] von Jena.

3.3 HTTP-Kommunikation

In einer Mehrbenutzerumgebung werden Technologien für die Kommunikation zwischen Client und Server benötigt. Da die Arbeit für die Implementierung Java verwenden wird, beschreibt dieser Punkt die notwendigen Techniken für den Server und den Client damit beide mittels HTTP (Hypertext Transfer Protocol) kommunizieren können.

3.3.1 HttpComponents

Die Apache HttpComponents sind ein Projekt der Apache Software Foundation, dieses Projekt beschäftigt sich mit der Erstellung und Wartung von Java-Komponenten für die Unterstützung des HTTP-Protokolls. Alle nachfolgenden Beschreibungen sind den Informationen und Tutorials der Projektseite der HttpComponents [Apa11] entnommen.
Die Apache HttpComponents bestehen aus zwei Hauptprojekten, dem HttpCore und dem HttpClient. Zusätzlich ist ein asynchroner Client in Entwicklung und eine ältere Codelinie des HttpClient wird auch noch zum Download angeboten. Der HttpCore ist eine Sammlung der grundlegenden Funktionen des HTTP-Protokolls. Diese Funktionen können zur Entwicklung von HTTP-Diensten auf Client und Server verwendet werden. Der HttpClient ist eine auf dem HttpCore basierende Implementierung eines HTTP-Agenten. Der Sourcecode des gesamten Projekts frei verfügbar unter Apache Lizenz.

HttpClient

Der HttpClient ist die bevorzugte Variante, wenn aus einem Java-Programm aus HTTP-Anfragen erzeugt werden sollen, denn die vorhandenen Mechanismen in Java sind nicht komfortabel zu verwenden. Der HttpClient ist kein Browser, es ist eine Bibliothek für den HTTP-Transport, also dem Senden und Empfangen von HTTP-Nachrichten. Dieses Kommunikation ist blockierend, das heißt sobald eine Anfrage an den Server gesendet wird, wird bis zur Antwort gewartet. Der HttpClient ist eine auf Standards und reinem Java basierende Implementierung von HTTP 1.0 und 1.1 mit der Unterstützung aller HTTP-Methoden (GET, POST, PUT, DELETE, HEAD, OPTIONS und TRACE). Weitere wichtige Features sind Unterstützung verschiedenster Sicherheitsmechanismen und Authentifizierungsarten.
Nachfolgend wird die Grundvariante der Verwendung des HttpClient beschrieben, natürlich gibt es noch viele weitere Funktionen welche der Projektseite entnommen werden können. Die wichtigste Funktion des HttpClient ist das Ausführen der HTTP-Methoden. Der Benutzer stellt ein Request-Objekt zu Verfügung, welches vom HttpClient zum Server gesendet wird. Der HttpClient liefert dann ein Response-Objekt vom Server oder eine Exception im Fehlerfall.

```
1  HttpClient httpclient = new DefaultHttpClient();
2  try {
3    HttpGet httpget = new HttpGet("http://www.google.com/");
4    HttpResponse response = httpclient.execute(httpget);
5    HttpEntity entity = response.getEntity();
6    if (entity != null) {
7      InputStream instream = entity.getContent();
8      // InputStream verarbeiten
9    }
10  } finally {
11   httpclient.getConnectionManager().shutdown();
12 }
```

Quellcode 3.3: HTTP-GET Request erzeugen

Quellcode 3.3 zeigt die einfachste Form wie ein HTTP-Request abgesetzt werden kann. Als erstes wird ein HttpClient und die gewünscht HTTP-Methode erzeugt. Um einen HTTP-Response zu erhalten, wird die HTTP-Methode am HttpClient ausgeführt. Danach kann die Antwort des Servers verarbeitet werden. An HTTP-Nachrichten können optional Daten in einer sogenannten Entity angehängt werden. Die Entity wird üblicherweise für POST- und PUT-Requests, sowie für alle Antworten des Server verwendet. Auch in Quellcode 3.3 wird die Entity der Antwort des Servers verarbeitet und es ist zu sehen, dass dies meist Steams sind. Wichtig ist, wenn der HttpClient nicht mehr benötigt wird, ihn mit der *shutdown()*-Methode zu beenden.

3.3.2 Webserver

Die Arbeit wird einen Equinox-basierten Webserver einsetzen [Ecl11b]. Equinox ist ein Projekt der Eclipse Foundation und es ist eine Implementierung der wichtigsten OSGi Spezifikationen. OSGi ist ein umfangreiches Framework und eine Laufzeitumgebung für modulare Softwaresysteme [MVA10]. Seit der Version 3 von Eclipse gibt es intern einen Webserver, anfangs war dies Tomcat und in späteren Versionen wurde er durch Jetty ersetzt. Seit dem Europa-Release von Eclipse ist es auch möglich, den internen Server für eigene Zwecke zu verwenden, da seit diesen Release alle benötigten Bundles mitgeliefert

werden [Bac09]. Es kann mit einigen Plug-ins ein voll funktionsfähiger Webserver als OSGi-Anwendung gestartet werden. Der Vorteil sich in der Eclipse-Welt zu bewegen ist, dass bereits existierenden Erweiterungen, wie zum Beispiel der Model Analyzer, einfach eingebunden werden können.
Die folgende drei Hauptkonzepte sind zum Verständnis eines Webservers wichtig [MVA10]:

- Inhalt: Der Inhalt sind die Bytes, welche der Server als Antwort auf eine Anfrage sendet. Diese Bytes werden statisch oder dynamisch ermittelt. Statische Inhalte (Ressourcen) sind Dateien oder andere Daten welche vom Server blind ausgeliefert werden. Dynamische Inhalte werden in der Java-Welt durch Servlets erzeugt.

- Pfad: Auf Inhalte wird bei HTTP via URLs zugegriffen. Wenn Inhalte registriert werden, muss der Pfad des Inhalts in der URL angegeben werden. Diese Angabe wird als *alias* bezeichnet.

- Kontext: Alle Anfragen an einen Server werden innerhalb eines Kontexts verarbeitet. Der Kontext ist einem Inhalt vorgeschaltet und definiert wie genau die Anfragen verarbeitet werden. Er übernimmt zum Beispiel die Zuordnung von URLs zu lokalen Inhalten oder die Implementierung von Sicherheits- und Authentifizierungsmechanismen.

Um dynamische Inhalte zu Verfügung zu stellen, muss am Server ein Servlet registriert werden. Dazu wird ein neues Plug-In in Eclipse erstellt, dieses Plug-in verwendet die Extension *org.eclipse.equinox.http.registry.servlets*.

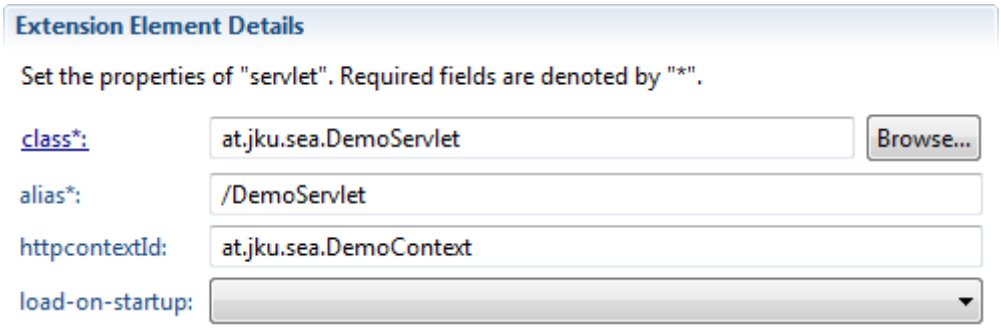


Abbildung 3.3: Extension Servlet

Abbildung 3.3 zeigt die Elementdetails der Extension in Eclipse. Die Extension hat vier Elemente für Einstellungen [Ecl11a, WHKL08]:

- class: Die Instanz des Servlets, welches registriert werden soll. Das angegebene Objekt muss das Interface *javax.servlet.Servlet* implementieren.

- alias: Das *alias*-Feld gibt an, unter welchem Pfad das Servlet registriert wird.

- httpcontextId: Hier kann ein Kontext-Objekt angegeben werden.

- load-on-startup: Dieser bool'sche Wert legt fest, ob das Servlet zum Start des Servers oder erst beim ersten Zugriff geladen werden soll.

```java
import javax.servlet.http.HttpServlet;

public class DemoServlet extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException {
    // Request verarbeiten und Response erstellen
  }
}
```

Quellcode 3.4: Demo Servlet

Der Quellcode 3.4 zeigt den Aufbau des in Abbildung 3.3 definierten Servlets. Ein Servlet erweitert üblicherweise die Klasse *javax.servlet.http.HttpServlet*. Die Klasse implementiert das vorgeschriebene Interface *javax.servlet.Servlet* und bietet den Vorteil, dass die benötigten HTTP-Methoden nur mehr überschrieben werden müssen. Durch beispielsweise dem Überschreiben der *doGet()*-Methode kann eine GET-Anfrage verarbeitet und ein entsprechender Response erzeugt werden. In Abbildung Abbildung 3.3 wird auch ein Kontext angegeben. Zur Definition eines Kontexts ist wiederum die Verwendung einer Extension notwendig. Konkret wird die Extension *org.eclipse.equinox.http.registry.httpcontexts* verwendet.

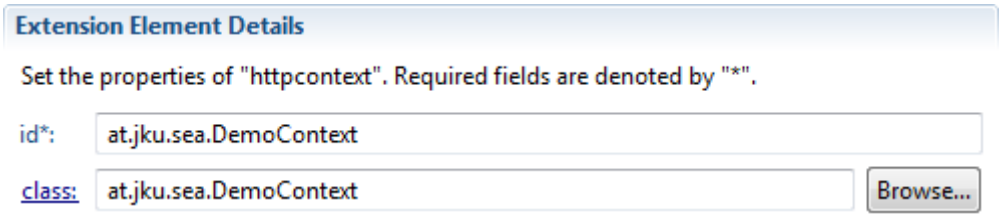


Abbildung 3.4: Extension Context

Wie Abbildung 3.4 zu sehen, hat die Extension für einen Kontext hat nur zwei Elemente [Ecl11a]:

- id: Die Identifikation des Kontext, diese wird von anderen Erweiterungen verwendet.
- class: Die Instanz des Kontexts, das angegebene Objekt muss das Interface *org.osgi.service.http.HttpContext* implementieren.

Die Klasse 3.5 implementiert das in der Erweiterung geforderte Interface *org.osgi.service.http.HttpContext*. Bevor eine Anfrage an den Server zum entsprechenden Servlet geleitet wird, wird die Methode *handleSecurity()* aufgerufen. Wird der Request erfolgreich authentifiziert, liefert die Methode *true*, ansonsten wird *false* zurückgegeben. Die Methode *getResource()* wird zum Auslesen von statischen Inhalten verwendet und die Methode *getMimiType()* für den übergebenen Dateinamen den zugehörigen Mime-Type [WHKL08].

```java
import org.osgi.service.http.HttpContext;

public class DemoContext implements HttpContext {
  @Override
  public boolean handleSecurity(HttpServletRequest req,
      HttpServletResponse resp) throws IOException {
    // Authentifizierung
    return true;
```

```
8     }
9
10    @Override
11    public URL getResource(String name) { return null; }
12
13    @Override
14    public String getMimeType(String name) { return null; }
15 }
```

Quellcode 3.5: Demo Kontext

Der Jetty-Server verwendet normalerweise den HTTP-Port 80, dieser kann bei Bedarf geändert werden. Dazu wird ein VM-Parameter, der beim Start der OSGi-Anwendung angegeben wird, benötigt und wird zum Beispiel folgendermaßen auf Port 9090 geändert: *-Dorg.eclipse.equinox.http.jetty.http.port=9090* [Bac09].

3.3.3 REST

In dieser Arbeit orientiert sich die Architektur der Servlets an REST (Representational State Transfer).
Grundsätzlich ist das Internet mit HTTP einfach aufgebaut. Der Browser verbindet sich zum Server, gibt ihm einen Pfad und der Server liefert das gewünschte Dokument. Um komplexere Services anzubieten, wurden die sogenannten Web Services entwickelt. Für Web Services gibt es einige Standards und Protokolle. Die meisten basieren auf HTTP. Diese Standards werden üblicherweise als WS-* Stack bezeichnet, darunter sind zum Beispiel WSDL und SOAP. Web Services haben das Problem, dass sie auch für kleine Anwendung sehr komplex werden und viele Daten übertragen werden müssen. Ziel ist es aber, die Einfachheit von HTTP auch für komplexe Services zu nutzen, genau hier setzt REST an. Die Grundideen von REST basieren auf dem Kapitel 5 der Dissertation von Roy Fielding [Fie00]. REST definiert keine Architektur, sondern es ist viel mehr eine Sammlung von Design- und Entscheidungskriterien für eine Architektur. Aus den Ideen von REST kann aber eine Architektur entworfen werden und die folgenden Punkte sind dabei die wichtigsten Konzepte [RR07].

Ressourcen und URIs

Im Allgemeinen kann eine Ressource als ein Objekt bezeichnet werden, welches einen Namen hat und adressiert werden kann [Vin08]. Üblicherweise sind Ressourcen auf einem Computer gespeichert und ein Stream aus Bits, beispielsweise ein Dokument, eine Zeile aus einer Datenbank oder das Ergebnis eines Algorithmus. Wichtig für eine Ressource ist, dass sie mindestens eine URI besitzt. Die URI ist der Name und die Adresse der Ressource. Wenn eine Ressource keine URI besitzt, kann sie nicht adressiert werden und existiert somit nicht im Web [RR07]. Wenn eine Anfrage für eine bestimmte Ressource den Server erreicht, entscheidet der Server wie die Ressource verarbeitet werden muss. Die URI einer Ressource muss nicht unbedingt ein Pfad am Server sein, sondern kann intern anders verarbeitet werden [Vin08]. Jede Ressource hat einen Zustand und der Server speichert diesen Zustand, die Kommunikation hingegen muss zustandslos sein. Jede Anfrage vom Client zum Server muss alle notwendigen Informationen enthalten, der Server verlässt sich nicht auf vorige Requests oder speichert für die einzelnen Clients eine Session [Fie00].

Repräsentationen

Die Bedeutung des Namen *Representational State Transfer* sagt schon sehr viel aus, es wird zwischen Client und Server eine Repräsentation von einen Zustand übertragen. Sie müssen also das gleiche Verständnis über das Format der Repräsentation haben, um sich gegenseitig zu verstehen [Vin08]. Der Server sendet die Daten in einem definierten Format, dies ist die Repräsentation einer Ressource. Eine Ressource ist die Quelle für die verschiedensten Repräsentationen und eine Repräsentation ist daher eine Sammlung von Daten über den aktuellen Zustand der Ressource. Zum Beispiel kann eine Liste als XML-Dokument, als eine Webseite oder als ein CSV-Text vom Server gesendet werden, alle sind verschiedene Repräsentationen, aber sie sind von der gleichen Ressource [RR07]. Über HTTP können Client und Server das zu verwendende Format für die Repräsentation bestimmen. Der Client gibt mit dem Headerfeld *Accept* die akzeptierten Typen an, welche er verarbeiten kann. In der Antwort des Servers wird das Headerfeld *Content-type* mit dem Typ der gesendeten Repräsentation gesetzt [Vin08].

Methoden

HTTP bietet vier Basismethoden an: GET, PUT, POST und DELETE. Diese Methoden werden als Operationen auf die Ressourcen angewendet. Die GET-Methode wird zum Abrufen einer Repräsentation einer Ressource verwendet. Mit der PUT-Methode kann der Zustand einer Ressource durch einen neuen Zustand ersetzt werden, das heißt die Ressource bekommt einen neuen Wert. Die DELETE-Methode wird zum Löschen von Ressourcen verwendet. Bei REST-Umgebungen wird die POST-Methode zum Erstellen oder Erweitern von Ressourcen verwendet [Vin08]. Auch weitere HTTP-Methoden finden Einsatz in REST. Um die Metadaten einer Ressource abzufragen, wird die HEAD-Methode eingesetzt. Für eine Abfrage, welche Methoden von einer Ressource unterstützt werden, wird die OTPIONS-Methode verwendet. Die Antwort auf einen OPTIONS-Request enthält im Header das Feld *Allow*. Wenn eine Ressource GET und HEAD unterstützt, würde im Header *Allow: GET, HEAD* für die gültigen Methoden stehen. Die beiden verbleibenden HTTP-Methoden CONNECT und TRACE werden in einer REST-Umgebung üblicherweise nicht verwendet [RR07].
Wenn eine REST-Schnittstelle korrekt implementiert wird, erfüllt die Schnittstelle zwei wichtige Eigenschaften [RR07]. GET- und HEAD-Requests sind sicher. GET, HEAD, PUT und DELETE sind idempotent.

- Sicherheit: Mit einem GET- oder HEAD-Request werden nur Daten abgefragt und es werden am Server keine Daten geändert. Das heißt, ein Client kann eine Abfrage zum Beispiel 10 Mal, einmal oder gar nicht durchführen, es hat immer die selben Auswirkungen auf den Server.

- Idempotenz: Eine Operation auf eine Ressource ist genau dann idempotent, wenn das Durchführen einer Anfrage das gleiche Ergebnis hat, wie das Durchführen einer Serie gleicher Anfragen. Wenn zum Beispiel eine Ressource gelöscht wird, ist sie weg. Wird sie erneut gelöscht, ist sie noch immer weg.

Diese Eigenschaften sind wichtig, denn so können zuverlässig Anfragen über ein unzuverlässiges Netz durchgeführt werden. Wird eine Anfrage nicht beantwortet, kann ohne Probleme eine weitere Anfrage durchgeführt werden [RR07].

Kapitel 4

Ideen und Varianten

Um die Konsistenzprüfung aus ihrer bisherigen Form in eine Mehrbenutzerumgebung zu überführen, wurden viele Ideen und Varianten diskutiert. Die folgenden Punkte dieses Kapitels repräsentieren die einzelnen Problemfelder, welche es durch die Forderung nach einer Mehrbenutzerumgebung zu lösen gilt. Zu diesen Problemfeldern gehören der theoretische Hintergrund von verteilten Systemen und Mehrbenutzerumgebungen, die Architektur des Gesamtsystems, die Aufteilung auf Client und Server, das Verhalten von Regeln in Teilmodellen. Einige der folgenden Ideen werden dann im nächsten Kapitel aufgegriffen und es wird daraus eine konkrete Lösung geformt.

4.1 Architektur

Die Einführung einer Mehrbenutzerumgebung führt zur Entwicklung eines verteilten Systems. Ein verteiltes System wird in [CDK05] als ein System definiert, bei dem Computer in einem Netzwerk miteinander kommunizieren und Nachrichten austauschen. Eine der Hauptmotivationen für ein verteiltes System ist das Austauschen und gemeinsame Nutzen von Ressourcen. Das Design eines verteilten Systems bietet einige Herausforderungen [CDK05]:

- Heterogenität: Ein verteiltes System wird meist in einer heterogenen Umgebung betrieben, das heißt es muss mit verschiedensten Netzwerken, Betriebssystemen, Hardware oder auch Programmiersprachen arbeiten können.

- Offenheit: Ein verteiltes System soll offen und leicht erweiterbar sein. Dazu sollten anerkannte Standards verwendet werden und wichtige Komponenten, wie zum Beispiel neue Dienste zum gemeinsamen Verwenden von Ressourcen, sollen einfach hinzuzufügen sein.

- Sicherheit: Es ist notwendig vertrauliche Informationen und Nachrichten zu verschlüsseln und zu schützen, da ein verteiltes System auch über öffentliche Netze kommunizieren kann. Meist wird es notwendig sein, Benutzer zu authentifizieren und sie dann auch für bestimmte Ressourcen zu autorisieren.

- Skalierbarkeit: Ein verteiltes System skaliert, wenn die Kosten für das Hinzufügen eines neuen Benutzers konstant sind. Die verwendeten Algorithmen müssen entsprechend gestaltet sein.

- Fehlerbehandlung: Jeder Prozess, jeder Computer oder jedes Netzwerk kann Fehler verursachen. Jede Komponente muss daher entsprechende Routinen zur Fehlerbehandlung unterstützen, damit im Fehlerfall nicht das ganze System zusammenbricht.

- Parallelität: Viele Benutzer in einem System können parallele Anfragen auf eine Ressource erzeugen. Jede Ressource muss daher so gebaut sein, dass sie mit diesen parallelen Anfragen umgehen kann.

- Transparenz: Ziel ist es, die Verteilung und Komplexität des Systems zu verbergen. Ein verteiltes System soll sich weitgehend so verhalten, als wäre es ein lokales System.

Bei verteilten System kann die Architektur dadurch unterschieden und bestimmt werden, indem die Aufteilung der einzelnen Softwarekomponenten betrachtet wird. Im Folgenden wird zwischen zentralisierte und dezentralisierte Architektur unterschieden. Da die Unterscheidung nicht immer eindeutig sein muss, kann es auch hybride Architekturen geben [TS06].

4.1.1 Peer-to-Peer

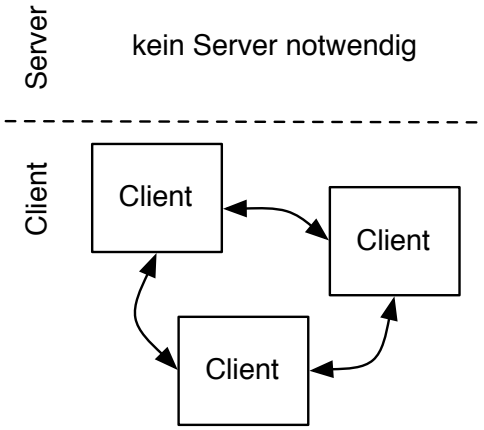


Abbildung 4.1: Peer-to-Peer Modell

Das Peer-to-Peer Modell ist ein dezentraler Ansatz [TS06]. In [SW04] wird es als System mit vollständig dezentraler Selbstorganisation und Ressourcennutzung beschrieben. Eine schematische Darstellung ist Abbildung 4.1 ist zu sehen. Neben den zwei grundlegenden Prinzipien gibt es noch weitere Eigenschaften, welche idealerweise alle gleichzeitig auf ein Peer-to-Peer System zutreffen [SW04]:

- Die wichtigsten Ressourcen (Bandbreite, Speicher, Rechenleistung) werden gleichmäßig genutzt und befinden sich auf den einzelnen Peers. Andere Peers nutzen wiederum diese Ressourcen.

- Die einzelnen Peers sind meist in einem globalen Netzwerk miteinander verbunden und haben keine fixe Internet-Adresse. Viele Peer-to-Peer Systeme verwenden daher eine eigene Adressierung.

- Die Nutzung und der Zugriff von Ressourcen erfolgt direkt zwischen den Peers, es wird keine zentrale Steuerung benötigt.

- Die einzelnen Peers sind Client und Server zugleich, das heißt sie bieten Dienste an und nutzen selbst auch andere Dienste oder auch den gleichen Dienst eines anderen Peers. Untereinander sind die einzelnen Peers gleichwertig und gleichberechtigt.

- Die Suche nach geeigneten Ressourcen sollte ohne zentrale Instanz auskommen und selbstorganisierend passieren.

Einer der größten Vorteile von Peer-to-Peer Systemen ist, dass zentrale Ressourcen können nicht zum Flaschenhals werden können [SW04]. Weiters sind sie gut geeignet, um ungenutzte Rechenkapazitäten zugänglich zu machen und sie skalieren sehr gut, da einfach neue Clients hinzugenommen werden können [CDK05]. Natürlich gibt es auch nachteilige Aspekte, wenn zum Beispiel viele veränderliche Daten verwendet werden, ist es teuer diese immer wieder über alle Peers zu verteilen [CDK05]. Zusätzlich lässt sich noch ein Nachteil in der Architektur selbst feststellen, wenn nur wenige Peers gleichzeitig online sind, kann es passieren dass Zugang zu bestimmten Ressourcen fehlt und Daten nicht vollständig verfügbar sind.

4.1.2 Client-Server

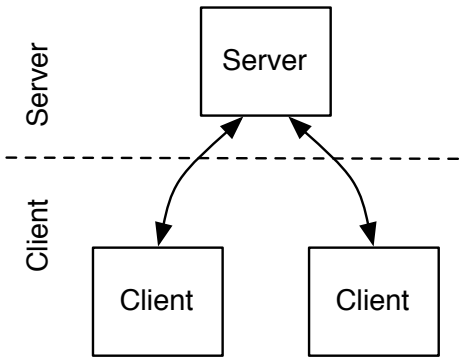


Abbildung 4.2: Client-Server Modell

Das Client-Server Modell kann auch als zentrale Architektur gesehen werden. In einem einfachen Client-Server Modell (Abbildung 4.2) sind die Prozesse des Mehrbenutzersystems in zwei Gruppen unterteilt. Der Server ist ein Prozess, welcher einen speziellen Service implementiert, wie zum Beispiel einen Datenbank-Service oder einen Web-Service. Der Client ist hingegen ein Prozess, welcher einen Service von einem Server aufruft. Dazu wird eine Anfrage an den Server gesendet und der Client wartet auf die Antwort des Servers [TS06]. Das Client-Server-Modell hat folgende Charakteristika [Sin92]:

- Ein Server stellt dem Client verschiedenste Dienste zu Verfügung, diese reichen von wenig rechenintensiv (Druckserver, Dateiserver) bis hin zu sehr rechenintensiv (Datenbankserver, Bildbearbeitung).

- Der Client startet die Kommunikation mit einem Server, ein Server wird nie die Kommunikation mit dem Client starten.

- Im Idealfall ist die Komplexität des Servers vor dem Client verborgen und der Client kann so komplexe Dienste über eine einfache Schnittstelle nutzen.

Der Einsatz eines Client-Server Modells hat den Vorteil, dass die Clients nicht gleichzeitig online sein müssen, denn der Server ist die zentrale Instanz für alle benötigten Ressourcen. Dies ermöglicht auch das Arbeiten über Zeitzonen hinweg, denn durch einen Server kann in verschiedensten Zeitzonen gearbeitet werden, ohne dass die Clients zu einem Zeitpunkt gemeinsam online sein müssen. Bei einem Peer-to-Peer System könnten verschiedene Zeitzonen einige Probleme bereiten. Ein Nachteil im Vergleich zum Peer-to-Peer System liegt auch wieder in der Architektur des Client-Server Modell, denn der Server könnte zum Flaschenhals werden, wenn immer mehr Ressourcen gefordert werden.
Eines der Hauptprobleme bei Client-Server Modell ist, eine klare Linie zwischen Client und Server zu ziehen. Es stellt sich die Frage, wo sollen welche Softwarekomponenten vorhanden sein. Durch diese Frage ergibt sich viel Variabilität, beginnend mit Thin-Clients, welche nur für Ein- und Ausgabe zuständig sind, bis hin zu Fat-Clients, welche auch Aufgaben des Servers selbst übernehmen können [TS06].

Thin-Client

Bei Thin-Clients wird nahezu die gesamte Funktionalität vom Server zu Verfügung gestellt, der Client führt nur mehr einfache Aufgaben durch [Jer98]. Für die Konsistenzprüfung würde dies bedeuten, dass am Client nur die Manipulation der Modelldaten und die Anzeige der Konsistenzregeln stattfindet. Das eigentliche Modell und die Evaluierung der Konsistenzregeln würde am Server passieren. Ein Thin-Client würde die Mobilität steigern, da alle Ressourcen am Server vorhanden sind und der Client daher einfach ausgetauscht werden kann. Ein weiterer Vorteil ist, dass ein Thin-Client die Anforderungen an Rechenzeit und Speicher am Client senkt und somit auch der Einsatz von mobilen Geräten möglich wäre.

Fat-Client

Fat-Clients ermöglichen interaktivere Benutzerschnittstellen, da sie nicht auf Antworten von einem Server warten müssen [Jer98]. Die Konsistenzprüfung in ihrer Grundform entspricht einem Fat-Client, denn sie verfügt über alle Funktionen zum eigenständigen Arbeiten. Ein Fat-Client hat den Vorteil, dass ein Offlinebetrieb leichter zu realisieren wäre. Wenn keine Verbindung zum Server vorhanden ist, sind Daten und Logik am Client vorhanden und er kann autonom weiter arbeiten.

4.1.3 Architekturentscheidung

Neben den allgemeinen Anforderungen, sowie den Vor- und Nachteilen der beiden diskutierten Architekturen gibt es noch weitere Kriterien, welche zur Entscheidungsfindung für die geeignete Architektur der Konsistenzprüfung in einer Mehrbenutzerumgebung beitragen.

- Große Modelle beinhalten üblicherweise viele verschiedene Teilbereiche und -modelle. Diese Teile erfordern meist spezialisierte Entwicklungs- und Modellierungswerkzeuge. Modelle sollen auch über die Grenzen der einzelnen Entwicklungswerkzeuge hinweg validiert werden können. Ein Benutzer kann nur jene Teile des Modells bearbeiten, welche auch wirklich von seinem Werkzeug unterstützt werden. Natürlich können aber Konsistenzregeln definiert werden, welche Modellelemente der unterschiedlichsten Teilbereiche verwenden. Solche Konsistenzregeln können nicht mehr lokal evaluiert werden, da das verwendete Werkzeug nicht auf das gesamte Modell zugreifen kann und somit fehlen der lokalen Konsistenzprüfung die benötigte Daten. Auf einem gemeinsamen Server hingegen, ist das gesamte Modell gespeichert. Zusätzlich muss das Modell vom Server nicht bearbeitet werden, es werden nur Modellelemente gelesen. Ein gemeinsamer Server hat somit Zugriff auf das gesamte Modell und daher kann eine Konsistenzprüfung am Server auch Regeln über das gesamte Modell evaluieren.

- Das System muss feststellen können, ob und wann gleiche Modellbereiche bearbeitet werden. Für jeden Benutzer werden die Änderungen in einem eigenen Bereich gespeichert, damit Konflikte zu anderen Benutzern vermieden werden können. Die Erkennung der Parallelität ist notwendig, da durch Änderungen von Modellelementen meist Konsistenzregeln betroffen sind und neu evaluiert werden müssen. Wenn jeder Benutzer gleiche Modellbereiche anders ändert, so liefern auch die Konsistenzregeln jeweils unterschiedliche Ergebnisse. Durch das Erkennen von Parallelität wird für jeden Benutzer eine eigene Kopie der betroffenen Konsistenzregel angelegt, so hat jeder Benutzer seine eigene Regel und somit auch sein eigenes Ergebnis der Evaluierung. Teile der Konsistenzregel, welche nicht verändert wurden, können natürlich von den Benutzer gemeinsam verwendet werden.

Um die Konsistenzprüfung in einer Mehrbenutzerumgebung zu verwenden und die erforderlichen Kriterien zu erfüllen, eignet sich das Client-Server Modell besser als das Peer-to-Peer Modell. Bei einer Peer-to-Peer Lösung könnte zwar die Konsistenzprüfung in seiner bisherigen Form verwendet werden, es wäre jedoch notwendig sie zu erweitern, damit sie mit anderen Konsistenzprüfungen kommunizieren kann und Daten untereinander ausgetauscht werden können. Das Peer-to-Peer Modell kommt als Architektur nicht in Frage, da es die zusätzlichen Kriterien nicht optimal erfüllen kann. Für das Client-Server Modell spricht die einfache zentrale Datenhaltung und damit die Möglichkeit die Konsistenzprüfung für diese zentralen Daten zu optimieren.

4.2 Aufteilung

Als zentrale Architektur für diese Arbeit wird das Client-Server Modell gewählt. Die Verwendung des Client-Server Modells bringt aber nicht nur Vorteile mit sich, wird das Modell und die Konsistenzprüfung für jeden Benutzer einfach auf dem Server anstatt auf dem Client verlagert, bedeutet dies eine große Mehrbelastung für den Server. Der Server müsste die Leistung aller Clients erbringen. Es ist daher unumgänglich den Server zu optimieren und dazu zählt die Aufteilung der einzelnen Komponenten. Die Suche nach der optimalen Aufteilung beschäftigt sich mit der Frage, wie kann das Modell und die Konsistenzprüfung zwischen Client und Server verteilt werden. Die Aufteilung kategorisiert nach zwei Hauptkriterien. Einerseits nach dem Kriterium der Skalierbarkeit, das heißt die Last am Server soll nicht linear mit der Anzahl der Benutzer steigen, andererseits nach dem Kriterium der Teilmodelle, das heißt jeder Benutzer soll nur seinen relevanten Teil vom Modell besitzen. Es wird bei den Aufteilungen davon ausgegangen, dass alle Benutzer über die gleichen Konsistenzregeln verfügen. Zusätzlich beginnen die Benutzer bei einem nicht leerem Modell, es wird die Situation bei einem größerem Modell analysiert.

4.2.1 Skalierbarkeit

Das erste Kriterium ist die Skalierbarkeit und es bedeutet im Folgenden, dass die Einführung der Mehrbenutzerumgebung auch einen Vorteil bezüglich der verbrauchten Ressourcen mit sich bringen soll. Der schlechteste Fall wäre, dass sich die verbrauchten Ressourcen und die Anzahl der Benutzer linear verhalten und so keine Verbesserung mit der Einführung der Mehrbenutzerumgebung eintritt. Das System würde sich verhalten, als würde jeder Benutzer für sich selbst arbeiten. Dieser Fall würde auch eintreten, wenn für jeden Benutzer am Server ein eigenes Gesamtmodell gespeichert wird und darauf eine eigene Konsistenzprüfung durchgeführt. Einzusparende Ressourcen sind zum Beispiel Rechenzeit und Speicherverbrauch, denn jede zusätzliche Evaluierung einer Konsistenzregel verursacht Kosten. Abstraktere Ressourcen sind die Evaluierungen von Regelinstanzen und auch die Modelldaten, also wie viel Speicher durch die Modellelemente belegt wird. Der Verbrauch von Ressourcen soll möglichst gering ausfallen und wenn ein weiterer Benutzer zum System kommt, sollen möglichst wenig neue Ressourcen benötigt werden.

lokale Konsistenzprüfung

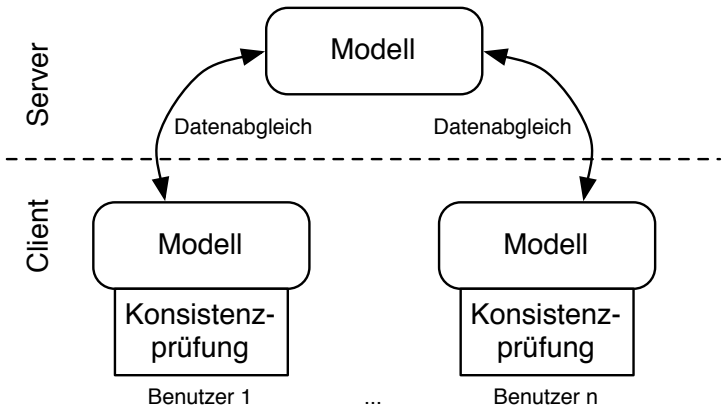


Abbildung 4.3: Aufteilung 1 mit lokaler Konsistenzprüfung

Bei der ersten Aufteilung (Abbildung 4.3) verfügt jeder Benutzer über eine eigene Konsistenzprüfung. Diese Konsistenzprüfung evaluiert vollständig das lokale Modell und jeder Benutzer besitzt das gesamte Modell. Damit das Modell über alle Benutzer hinweg synchronisiert werden kann, werden Änderungen am Modell zu einen gemeinsamen Server übertragen. Diese Variante der Aufteilung skaliert nicht sehr gut, da jeder Benutzer alle Daten lokal besitzt, das heißt bei jedem Benutzer müssen alle Konsistenzregeln evaluiert werden. Kommt ein weiterer Benutzer zum System, muss das gesamte Modell übertragen werden und es müssen alle Konsistenzregeln erneut evaluiert werden, obwohl sie bereits von anderen Benutzern evaluiert wurden. Da bei dieser Aufteilung das gesamte Modell zur Konsistenzprüfung am Client benötigt wird, kann dies zu einem Sicherheitsproblem führen, denn unter Umständen ist nicht jeder Benutzer berechtigt, das gesamte Modell sehen und zu bearbeiten. Diese Aufteilung kann auch als erweiterte Stand-Alone Variante gesehen werden, bei der zusätzlich mit einem gemeinsamen Modell gearbeitet wird, ansonsten verhalten sich die Clients weitgehend autonom. Diese Autonomie bringt zwar Flexibilität, jedoch werden keine Regeln oder Ergebnisse der Evaluierung gemeinsam benutzt. Aufteilung 1 würde mit einem Fat-Client entsprechen.

globale Konsistenzprüfung

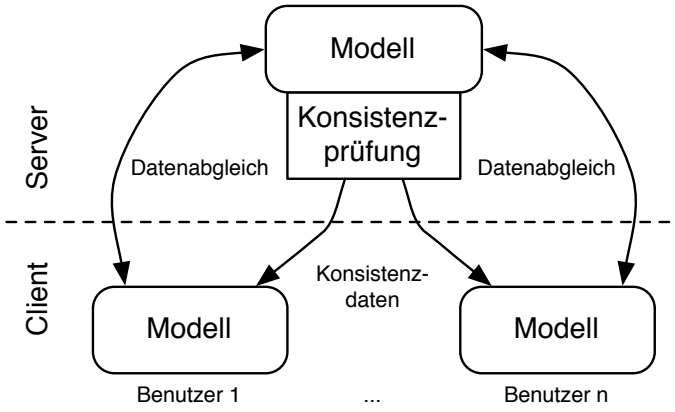


Abbildung 4.4: Aufteilung 2 mit globaler Konsistenzprüfung

Bei Aufteilung 2 wandert die Konsistenzprüfung von den einzelnen Benutzern zum Server, der Client wird zu einem Thin-Client. Wie in Abbildung 4.4 zu sehen, wird wie bei Aufteilung 1 das Modell über alle Benutzer hinweg synchronisiert, jeder Benutzer besitzt somit die gleichen Modelldaten. Wenn es verschiedene Berechtigungsebenen auf dem Modell gibt, ist es möglich nicht die gesamten Modelldaten zu synchronisieren, denn der Client benötigt nicht zwingend alle Modelldaten. Die Konsistenzprüfung evaluiert das Modell am Server und die Konsistenzdaten werden von den einzelnen Benutzern abgerufen. Im Vergleich zur ersten Aufteilung werden sehr viele Evaluierungen von Regeln eingespart, da die Konsistenzprüfung nur einmal für alle Benutzer ausgeführt werden muss. Wenn ein neuer Benutzer dem System beitritt, muss das Modell und die evaluierten Konsistenzregeln übertragen werden. Es muss keine Regeln neu evaluiert werden und der neue Benutzer profitiert von den bereits errechneten Ergebnissen. Wenn die Konsistenzprüfung zum Server wandert, kann sie für den Mehrbenutzerbetrieb optimiert werden, da auf dem Server alle Daten verfügbar sind. Bleibt sie wie bei Aufteilung 1 bei den einzelnen Clients ist der Mehrbenutzerbetrieb schwieriger realisierbar, denn es müssten vorher Daten aller anderen Clients besorgt werden.

4.2.2 Teilmodelle

Wenn das Gesamtsystem betrachtet wird, ist es besser die Konsistenzprüfung auf den Server zu verlagern. Durch diese Verlagerung können Ressourcen eingespart werden und am Server können gemeinsame Daten wiederverwendet werden. Aus diesem Grund basieren die nächsten beiden Varianten auf der Aufteilung 2 (Abbildung 4.4), jedoch muss diese Aufteilung noch verfeinert werden. Wie bereits beschrieben, werden Änderungen eines einzelnen Benutzers zum Server übertragen und alle Benutzer arbeiten an einem gemeinsamen Modell. Jeder Benutzer sieht somit sofort, welche Änderungen von den anderen Benutzern durchgeführt worden sind. Ein gemeinsam benutztes Modell kann zu Konflikten und gegenseitigem Überschreiben von Modellelementen führen, vor allem wenn mehrere Benutzer gleiche Modellabschnitte bearbeiten. Die radikale Methode diese Konflikte zu vermeiden, wäre für jeden Benutzer am Server ein eigenes Gesamtmodell zu speichern und zu evaluieren. Dieser Schritt würde bedeuten, dass der Server durch eine große Datenmenge überlastet wird. Um die Last am Server einzudämmen, ist es besser wenn für jeden Benutzer nur mehr seine Änderungen gespeichert und evaluiert werden. Ziel ist es also, dass für jeden Benutzer ein eigener Modellbereich vorhanden ist, in dem seine Änderungen abgespeichert und evaluiert werden. Durch diese eignen Bereiche beeinflussen sich die einzelnen Benutzer nicht mehr gegenseitig, die Konsistenzprüfung kann jedoch auf die anderen Bereiche und vor allem auf den öffentlichen Bereich zugreifen.

lokale Änderungsprüfung

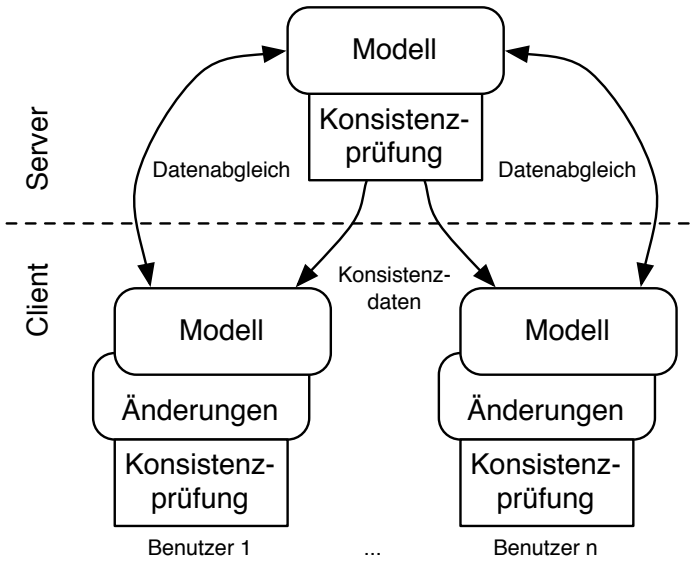


Abbildung 4.5: Aufteilung 3 mit lokaler Änderungsprüfung

Die lokale Änderungsprüfung (Abbildung 4.5) verwendet für jeden Client eine eigene Konsistenzprüfung, der Client entspricht somit einem Fat-Client. Die eigene Konsistenzprüfung evaluiert aber nur die Änderungen am lokalen Modell, beziehungsweise wird nur auf dem Teilmodell des Benutzers die Konsistenzprüfung durchgeführt. Die Änderungen werden zusätzlich zum Server übertragen und wiederum in das gemeinsame Modell zusammengeführt. Die Ergebnisse der globalen Konsistenzprüfung werden auch zum Client übertragen und so sieht der Benutzer evaluierte Regeln aus seiner lokalen und der globalen Konsistenzprüfung. Für einen neuen Benutzer müssen keine zusätzlichen Konsistenzregeln

evaluiert werden, es ist nur notwendig das Modell zu übertragen. Diese Aufteilung hat jedoch das Problem noch nicht gelöst, dass Konflikte im gemeinsamen Modell entstehen können, deshalb wird noch eine weitere Variante benötigt. Weiters kann bei dieser Variante der Fall eintreten, dass Modelldaten nachgeladen werden müssen, wenn diese von der Konsistenzprüfung benötigt werden. Wie bei Aufteilung 1 ergibt sich dadurch wieder ein Sicherheitsproblem, wenn Daten nicht für einen Benutzer bestimmt sind, sollen sie auch nicht für die Konsistenzprüfung den Server verlassen.

globale Änderungsprüfung

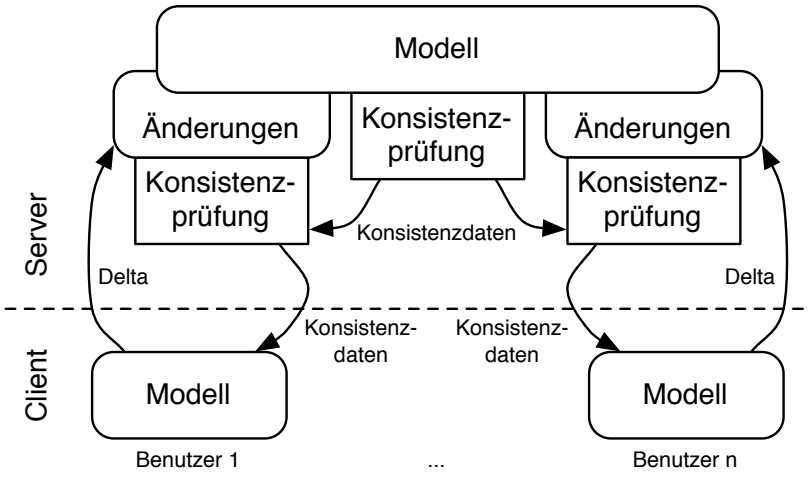


Abbildung 4.6: Aufteilung 4 mit globaler Änderungsprüfung

Die letzte Aufteilung ist in Abbildung 4.6 zu sehen. Die Änderungen oder auch Deltas der einzelnen Benutzer werden zum Server gesendet. Am Server ist einerseits das gesamte Modell, andererseits für jeden Benutzer ein eigener Bereich für Änderungen vorhanden. Für jeden dieser Bereiche gibt es eine eigene Konsistenzprüfung. Die Konsistenzprüfung der einzelnen Benutzer verwendet zur Evaluierung der Konsistenzregeln die Ergebnisse der gemeinsamen Konsistenzprüfung. Das Ergebnis der Konsistenzprüfung wird vom jeweiligen Benutzer abgeholt. Für den einzelnen Benutzer hat es den Eindruck, als würde er autonom arbeiten, obwohl seine Daten auf einem gemeinsamen Server vorhanden sind und die Konsistenzprüfung globale Daten verwendet. Eine Rückführung der lokalen Modelldaten in das globale Modell muss als eigenständiger Schritt durchgeführt werden. Wenn ein neuer Benutzer zum System kommt, ist nur eine Übertragung des gemeinsamen Modells notwendig.

4.2.3 Ergebnis

Das Ergebnis der Variantenfindung ist, dass Aufteilung 4 alle Kriterien an eine Konsistenzprüfung in einer Mehrbenutzerumgebung erfüllt. Sie bietet große Flexibilität, da der Server optimiert werden kann, um möglichst viele Daten wieder zu verwenden. Die Verwendung eines Servers bietet auch die Möglichkeit, Teilmodelle für jeden Benutzer zu speichern. Durch die Verwendung von Änderungsbereichen für die einzelnen Benutzer bleibt die Last am Server überschaubar. Am Server können Konsistenzregeln über das gesamte Modell hinweg evaluiert werden, ohne dass die einzelnen Clients das gesamte

Modell benötigen. Auch Sicherheitsbedenken können durch diese Aufteilung beseitigt werden, da den einzelnen Benutzern nur jene Modelldaten gegeben werden müssen, welche sie wirklich benötigen. Da aber am Server alle Modelldaten vorhanden sind, kann die Konsistenzprüfung ohne Einschränkungen arbeiten. Aufteilung 4 verwendet Thin-Clients und somit kommen auch alle in Punkt 4.1.2 beschriebenen Vorteile zum tragen. Die Vorteile des Fat-Clients, wie zum Beispiel der Offlinebetrieb, werden dadurch ausgeglichen, dass die Konsistenzprüfung für einen Benutzer bereits existiert und wenn nur lokal gearbeitet werden soll, kann die bereits vorhandene Lösung verwendet werden.

4.3 Regeln in Teilmodellen

Durch den Einsatz eines Servers mit gemeinsamen Modell ist es notwendig, das Verhalten von Regeln in diesem gemeinsamen Modell zu analysieren. Wie beschrieben, wird um den Server nicht zu überlasten, ein zweigeteiltes gemeinsames Modell eingesetzt, anstatt für jeden Benutzer ein eigenes Modell zu speichern. Alle Benutzer teilen sich einen Bereich des Modells, das öffentliche Modell. Dieser Teil ist für die Benutzer nur lesbar und wird üblicherweise nicht geändert. Der öffentliche Bereich muss zu Beginn der gemeinsamen Arbeit von einen der Benutzer initialisiert werden. Zusätzlich zu diesem öffentlichen Modell besitzt jeder Benutzer ein eigenes privates Modell, dieses private Modell beinhaltet die Änderungen der einzelnen Benutzer am öffentlichen Modell. Der Aufbau des Gesamtmodells kann mit einem Baum verglichen werden. Das öffentliche Modell ist der Stamm und ausgehend davon, gibt es für jeden der Benutzer einen eigenen Ast. Das Hinzufügen, Ändern und Löschen von Modellelementen hat unmittelbare Auswirkungen auf die einzelnen Konsistenzregeln und deren Regelinstanzen. Wenn eine lokale Konsistenzprüfung auf einem eigenen Modell durchgeführt wird, bedeutet das Hinzufügen von Modellelementen das Erstellen von neuen Regelinstanzen der betroffenen Konsistenzregeln. Das Ändern von Modellelementen bewirkt, dass die Regelinstanzen, welche diese Modellelemente verwenden, neu evaluiert werden müssen. Das Löschen eines Elements zieht das Löschen betroffener Regelinstanz nach sich. Dieses einfache Verhalten der Konsistenzregeln muss nun auf die gemeinsamen Modelle der Mehrbenutzerumgebung übertragen werden und führt zu einem komplexeren Verhalten. Das neue Verhalten der Konsistenzregeln wird nachfolgend mit einem Beispiel beschrieben.

öffentliches Modell

Das öffentliche Modell ist der gemeinsam verwendete Teil des Modells am Server und es muss eine initiale Konsistenzprüfung durchgeführt werden. Die einzelnen Pfeile repräsentieren Konsistenzregeln. Eine Konsistenzregel beginnt bei einem Modellelement und erstreckt sich meist über weitere Elemente. In Abbildung 4.7 sind fünf Konsistenzregeln zu sehen, jede erhält eine Nummer zur Identifikation. Die Zahl in der Klammer bedeutet, dass eine Regelinstanz erstellt wurde und dem Modell in der Klammer zugeordnet worden ist. Diese Zahl in der Klammer kann auch als Version der Konsistenzregel angesehen werden, da jeder Benutzer eine eigene Version seines Teils des öffentlichen Modells erzeugt. In der Abbildung bedeutet *1(0)* beispielsweise, dass eine Instanz der Regel *1* dem öffentlichen Modell *(0)* zugeordnet ist. Insgesamt gibt es fünf Konsistenzregeln mit je einer öffentlichen Regelinstanz, das Ergebnis der einzelnen Regelinstanzen steht allen Benutzern zu Verfügung.

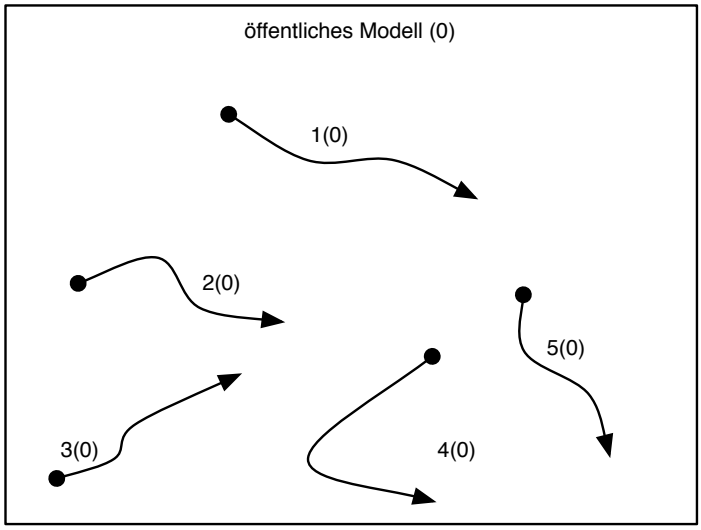


Abbildung 4.7: öffentliches Modell

privates Modell

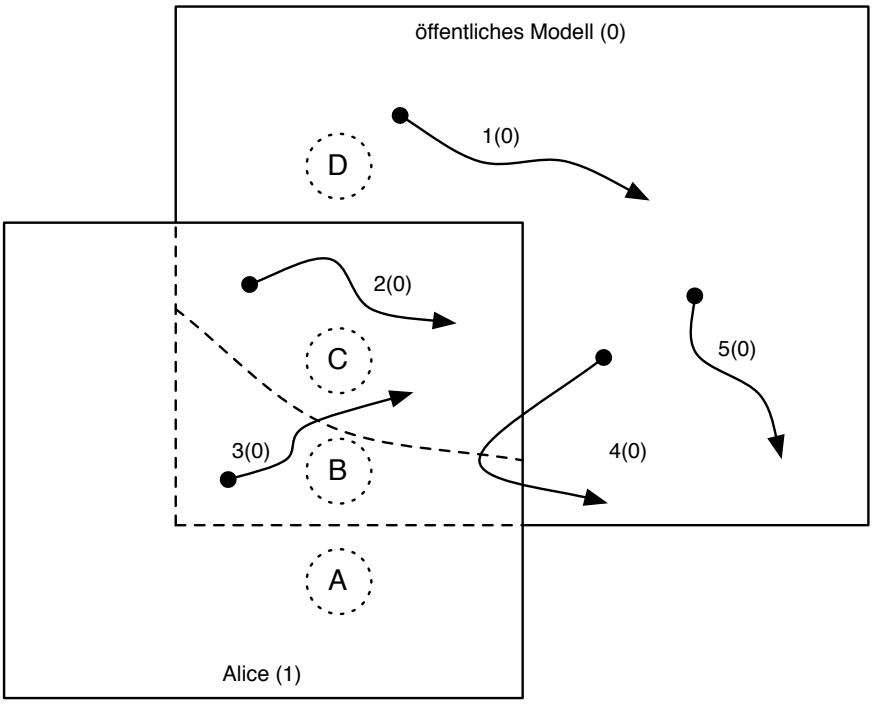


Abbildung 4.8: privates Modell

Als erstes verbindet sich Alice mit dem Server und will mit dem Modell arbeiten. Generell gilt, wenn ein neuer Benutzer zum System kommt, wird ein privater Modellbereich angelegt und es entstehen vier Bereiche, zu denen Modellelemente zugeordnet werden können (Abbildung 4.8). Der äußere Bereich *A* ist für neue Modellelemente. Wird ein neues Element angelegt oder hinzugefügt, kommt es in diesen Bereich. Andere Benutzer können diese Elemente weder sehen, noch können sie darauf zugreifen. Der mittlere Bereich *B* beinhaltet geänderte Elemente, also Elemente aus dem öffentlichen Bereich, welche der Benutzer modifiziert hat. Konkret gibt es dann das gleiche Element im öffentlichen und

privaten Bereich, es unterscheidet sich nur durch den geänderten Wert. Zu diesem Bereich gehören auch gelöschte Elemente aus dem öffentlichen Bereich. Bei der Mehrbenutzerumgebung darf nicht einfach ein Element aus dem öffentlichen Bereich gelöscht werden, da dieses von anderen Bereichen noch verwendet werden könnte. Das entsprechende Element muss als gelöscht markiert werden und wird so dem entsprechenden Benutzer verborgen. Der innere Bereich C beinhaltet alle Elemente, welche vom Benutzer verwendet werden, jedoch nicht geändert wurden. Alle anderen Elemente aus dem öffentlichen Modell (Bereich D) werden von Alice nicht verwendet. Die Gründe, warum Alice diese Elemente nicht verwendet, sind vielfältig. Der naheliegenste Grund ist, dass die Modellelemente einfach nicht benötigt werden. Ein weiterer Grund könnte sein, dass das Modellierungswerkzeug von Alice diese Bereiche und Modellelemente nicht unterstützt und sie somit nicht bearbeitet werden können. Auch Sicherheitsmechanismen können ein Grund sein, warum Modellelemente nicht verwendet werden können. Diese geschützten Elemente könnten eine spezielle Berechtigung benötigen, damit sie von Alice verwenden werden können. Obwohl Elemente von einzelnen Benutzern nicht verwendet werden können oder nicht gesehen werden dürfen, können sie trotzdem am Server ein Teil von Konsistenzregeln sein.

Änderungen von Alice

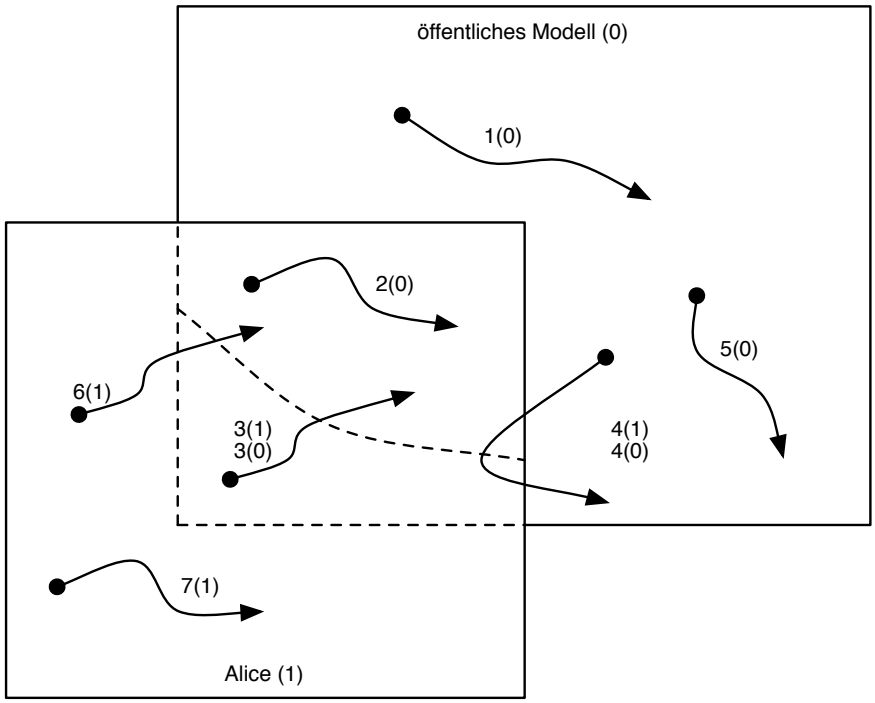


Abbildung 4.9: Änderungen von Alice

Alice führt einige Änderungen am Modell durch, diese Änderungen haben Auswirkung auf die Konsistenzregeln und deren Regelinstanzen. Es wurden neue Elemente angelegt, dadurch ist die komplett neue Regelinstanz *7(1)* erzeugt worden, auch die Regelinstanz *6(1)* beginnt im Bereich der neuen Modellelemente. Regelinstanz *6* verwendet auch geänderte und nicht geänderte Modellelemente. Weiters wurden von Alice einige bestehende Modellelemente geändert, durch diese Änderungen müsste die Regelinstanz *3(0)* neu evaluiert werden, da dies aber eine öffentliche Regelinstanz ist, muss eine Kopie *3(1)* erstellt werden und es wird die Kopie neu evaluiert. Auch Modellelemente der Konsistenzregel

4 sind durch die Änderungen betroffen. Bei dieser Konsistenzregel tritt nun ein weiterer Spezialfall auf, da Teile nicht mehr im Bereich von Alice liegen. Auch in solchen Fällen kann eine Kopie der Regelinstanz angelegt und evaluiert werden. Hier zeigt sich wieder ein Vorteil der Verwendung eines gemeinsamen Servers. Alice steht das Ergebnis der Regelinstanz *4(1)* zu Verfügung, obwohl sie Teile der verwendetet Modellelemente nicht lesen, beziehungsweise nicht verarbeiten kann. Die Regelinstanzen *7(1)*, *6(1)*, *4(1)*, *3(1)* kann nur Alice sehen. Sie interessiert sich auch für die Regelinstanz *2(0)*, da sich keine zugehörigen Modellelemente geändert hat, kann Alice die öffentliche Version verwenden. Die Konsistenzregeln *1* und *5* werden von Alice nicht verwendet.

Änderungen von Bob

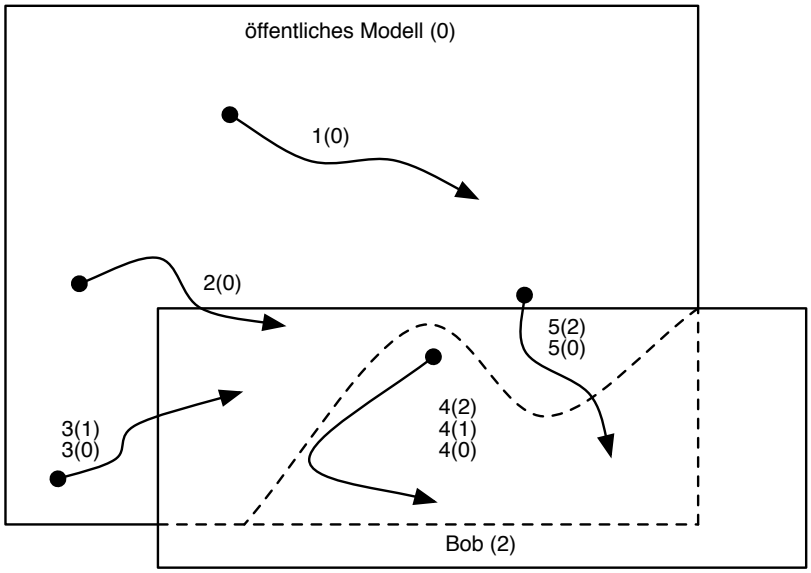


Abbildung 4.10: Änderungen von Bob

Nun verbindet sich Bob mit dem Server und er bekommt einen eigenen Modellbereich zugewiesen (Abbildung 4.10). Bob ändert nur bestehende Modellelemente. Durch die Änderungen von Bob sind die Konsistenzregeln *4* und *5* betroffen, es wird jeweils eine Kopie erstellt. Die beiden neuen Regelinstanzen *4(2)* und *5(2)* kann nur Bob sehen. Zusätzlich ist Bob an der Regel *2* und *3* interessiert. Zwar gibt es schon eine weitere Variante der Konsistenzregel *3*, diese sieht Bob aber nicht, da sie dem Modell von Alice zugeordnet ist und Bob verwendet deshalb die Regelinstanz *3(0)*. Konsistenzregel *1* wird auch von Bob nicht verwendet.

Gesamtübersicht der Bereiche

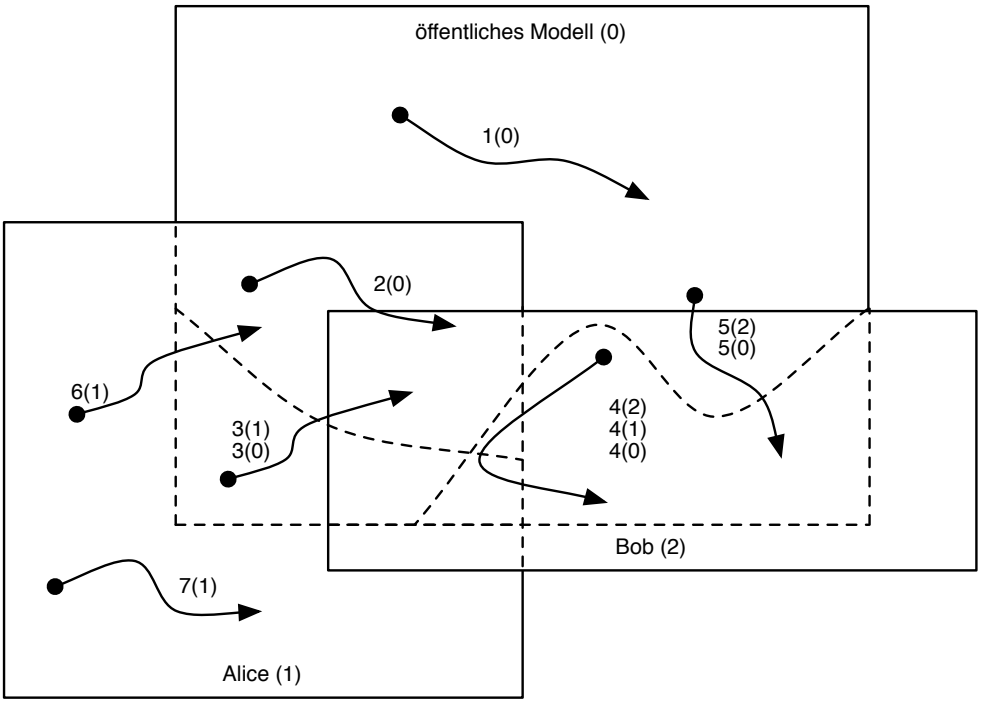


Abbildung 4.11: Gesamtübersicht der Bereiche

Abbildung 4.11 zeigt eine Gesamtübersicht der Modellbereiche mit allen Konsistenzregeln und deren Regelinstanzen:

1. Kein Benutzer verwendet diese Konsistenzregel.
2. Alice und Bob verwenden die öffentliche Regelinstanz dieser Konsistenzregel.
3. Alice besitzt eine eigene Regelinstanz, wohingegen Bob die öffentliche Variante nutzt.
4. Diese Konsistenzregel ist durch Änderungen von Alice und Bob betroffen, deshalb gibt es insgesamt drei verschiedene Ausprägungen im System. Ein neue Benutzer würde die öffentliche Regelinstanz der Konsistenzregel verwenden.
5. Es existiert eine öffentliche und eine private Regelinstanz der Konsistenzregel. Die private Regelinstanz gehört Bob.
6. Eine Regelinstanz wurde von Alice erstellt und verwendet sowohl geänderte als auch nicht geänderte Modellelemente. Sie ist nur für Alice sichtbar.
7. Eine weitere private Regelinstanz von Alice, diese Regelinstanz verwendet nur private Modellelemente von Alice.

Kapitel 5

Lösungskonzept

Im Kapitel 4 wurden verschiedenste Ideen und Varianten zur Konsistenzprüfung von Software Design Modellen in Mehrbenutzerumgebungen diskutiert, dieses Kapitel wird nun die Ideen aufgreifen und daraus ein konkretes Lösungskonzept formen. Das Konzept umfasst eine Übersicht der benötigten Funktionen, die daraus resultierende Architektur und den Umgang mit Konsistenzregeln am Server. Die Implementierung des Konzepts folgt dann mit dem Kapitel 6.

5.1 Funktionsübersicht

Durch die verschiedenen Ideen und Varianten lassen sich die Funktionen ableiten, welche das neue Client-Server System bieten soll. In der Funktionsübersicht werden alle benötigten Funktionen der Konsistenzprüfung in einer Mehrbenutzerumgebung zusammengefasst und in den anschließenden Punkten detaillierter beschrieben. Abbildung 5.1 zeigt die einzelnen Kommandos und Daten, welche von den Benutzern zum Server gesendet werden können. Bei den einzelnen Operationen ist auch die zeitliche Abfolge zu beachten, beispielsweise kann ein Import erst nach einen Export erfolgen, oder die einzelnen Benutzer können mit den Änderungen am Modell erst starten, nachdem das Modell am Server initialisiert worden ist.

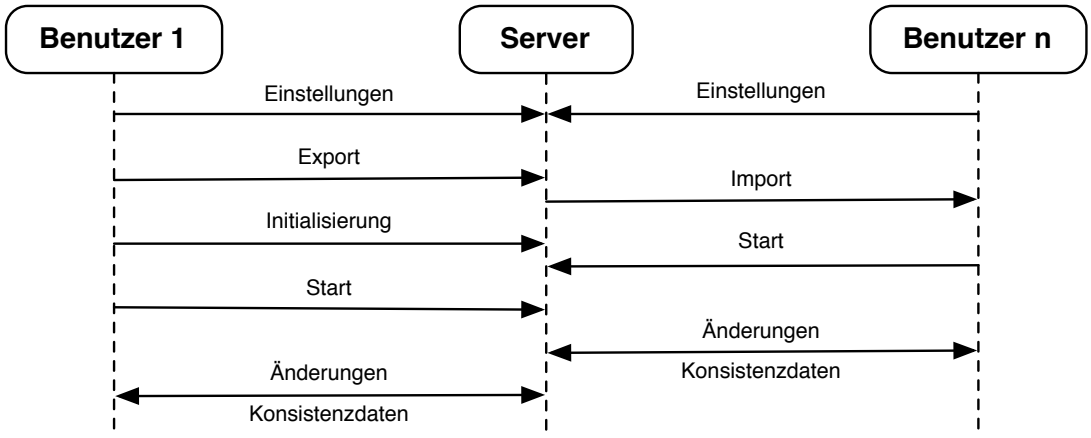


Abbildung 5.1: Funktionsübersicht

5.1.1 Funktionen

Einstellungen

Die Benutzer sollen diverse Einstellungen am Server vornehmen können. Dazu müssen sich die einzelnen Benutzer zunächst am Server authentifizieren, damit jede Anfrage an den Server einem Benutzer zugeordnet werden kann. Die Benutzerdaten werden am Server hinterlegt. Die Verwaltung der Daten passiert vom Client aus und es soll das Hinzufügen, Ändern und Löschen von Benutzerdaten möglich sein. Damit der Server auch mehrere Modelle gleichzeitig unterstützen kann, werden die einzelnen Modelle und die zugehörigen Daten in Projekten organisiert. Die Verwaltung der Projekte passiert auch vom Client aus und ermöglicht das Hinzufügen und Löschen von Projekten. Damit nicht jeder Benutzer alle Einstellungen am Server durchführen kann, ist ein Rollenkonzept vorgesehen.

Import und Export

Die Modelle der einzelnen Benutzer sollen einfach untereinander ausgetauscht werden können, dazu können die Modelle in Dateiform am Server gespeichert werden. Beim Exportieren werden die Dateien einem Projekt am Server zugeordnet und vom Client zum Server geladen. Andere Benutzer können diese Modelle vom Server in ihr lokales Modellierungswerkzeug importieren. Diese Funktion ist wichtig, da zu Beginn das Modell auf alle Benutzer verteilt werden muss. Die Modelldateien werden von einem der Benutzer initialisiert und stellen somit das öffentliche Modell dar, so kann jeder der Benutzer mit der gleichen Version des Modells seine Arbeit beginnen.

Initialisierung

Nachdem das Modell in Dateiform am Server verfügbar ist, muss es geöffnet und in den Speicher des Servers geladen werden. Da die Werkzeuge der Benutzer unterschiedlich sein können und der Server nicht alle Datenformate unterstützen kann, wird die Initialisierung vom Client aus durchgeführt. Es wird das geöffnete Modell in das vom Server vorgegebenen Format konvertiert und zum Server gesendet. Wichtig ist, dass das gleiche Modell initialisiert wird, welches auch in Dateiform vorliegt, denn alle Benutzer müssen mit dem gleichen Modell beginnen. Während der Initialisierung wird auch die erste Konsistenzprüfung durchgeführt. Es werden alle Regelinstanzen der betroffenen Konsistenzregeln angelegt und ab diesem Zeitpunkt können alle Benutzer die öffentlichen Daten verwenden. Die Initialisierung muss nur von einem Benutzer genau einmal durchgeführt werden.

Start

Durch das Starten des Systems am Clients kann die eigentliche Arbeit am lokalen Modell beginnen, denn es werden alle notwendigen Dienste zur Datenübertragung gestartet. Am lokalen Client wird das Monitoring der Änderungen am lokalem Modell gestartet. Es soll auf das Hinzufügen, Ändern und Löschen von Modellelementen reagiert werden. Sobald eine dieser Modelloperationen einritt, muss die Änderung zum Server gesendet werden. Ein periodischer Prozess holt die geänderten Regelinstanzen der Konsistenzregeln vom Server ab. Im Zuge der ersten Kommunikation des Clients mit dem Server wird ein privater Bereich für die Änderungen angelegt.

Änderungen und Konsistenzdaten

Wie bereits beschrieben, bewirkt das Starten des Clients, dass alle lokalen Änderungen am Modell aufgezeichnet und zum Server übertragen werden. Diese Änderungen werden in das private Modell eingetragen und es werden die entsprechenden Operationen an den Konsistenzregeln durchgeführt. Wie sich diese Operationen genau verhalten, ist in Punkt 5.3 beschrieben. Die Änderungen an den Konsistenzdaten können anschließend vom Client wieder abgeholt und angezeigt werden.

5.1.2 Ablauf

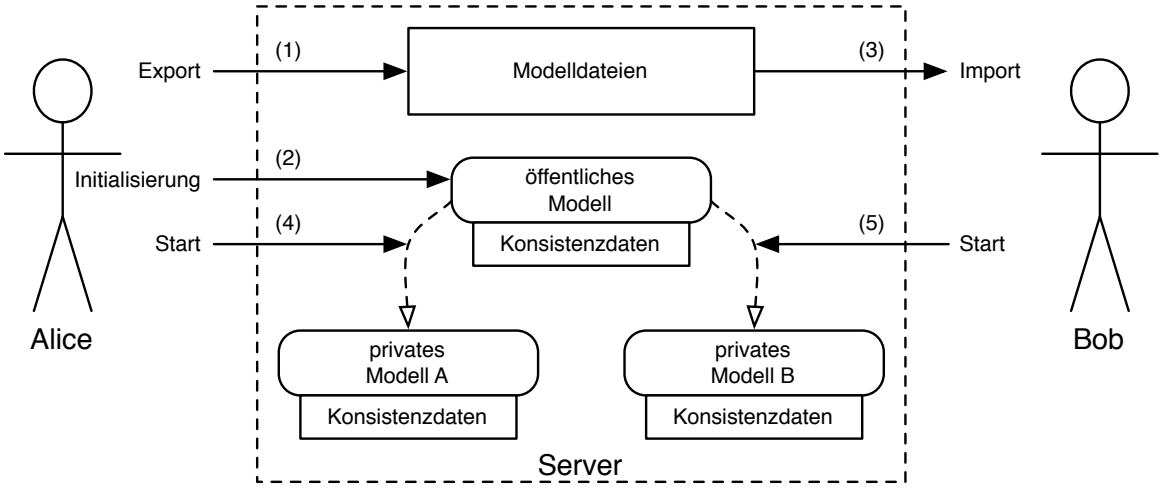


Abbildung 5.2: Ablauf der Funktionen

Ein typischer Ablauf der zuvor beschriebenen Funktionen ist in Abbildung 5.2 dargestellt. Es wird gezeigt, welche Funktionen die Benutzer in der richtigen Reihenfolge verwenden müssen, damit sie das System korrekt verwendet kann. In diesem Beispiel wollen Alice und Bob an einem gemeinsamen Modell arbeiten und dabei die Konsistenz des Modells fortlaufend prüfen. Alice lädt mittels Exportfunktion die gemeinsam benötigten Modelldateien zum Server *(1)*. Als nächsten Schritt muss sie das öffentliche Modell initialisieren *(2)*. Nach diesen beiden Schritten sind am Server ein öffentliches Modell und öffentliche Regelinstanzen für alle anderen Benutzer verfügbar. Bob lädt nun mittels Importfunktion die Modelldateien in sein Modellierungswerkzeug *(3)*. Nun können Alice *(4)* und Bob *(5)* mittels der Startfunktion die Arbeit am Modell beginnen und jeder der Benutzer startet mit der gleichen Version des Modells. Wenn später neue Benutzer zum System kommen, sind nur die Schritte von Bob notwendig.

5.2 Architektur

Die Architektur für die Konsistenzprüfung in der Mehrbenutzerumgebung basiert auf der Architekturentscheidung aus Punkt 4.1 und der Variantenfindung aus Punkt 4.2. Abbildung 5.3 zeigt eine Übersicht der Architektur und ihrer wichtigsten Komponenten. Der Server wird die Hauptkomponente des Systems und die einzelnen Clients können mittels HTTP mit ihm kommunizieren.

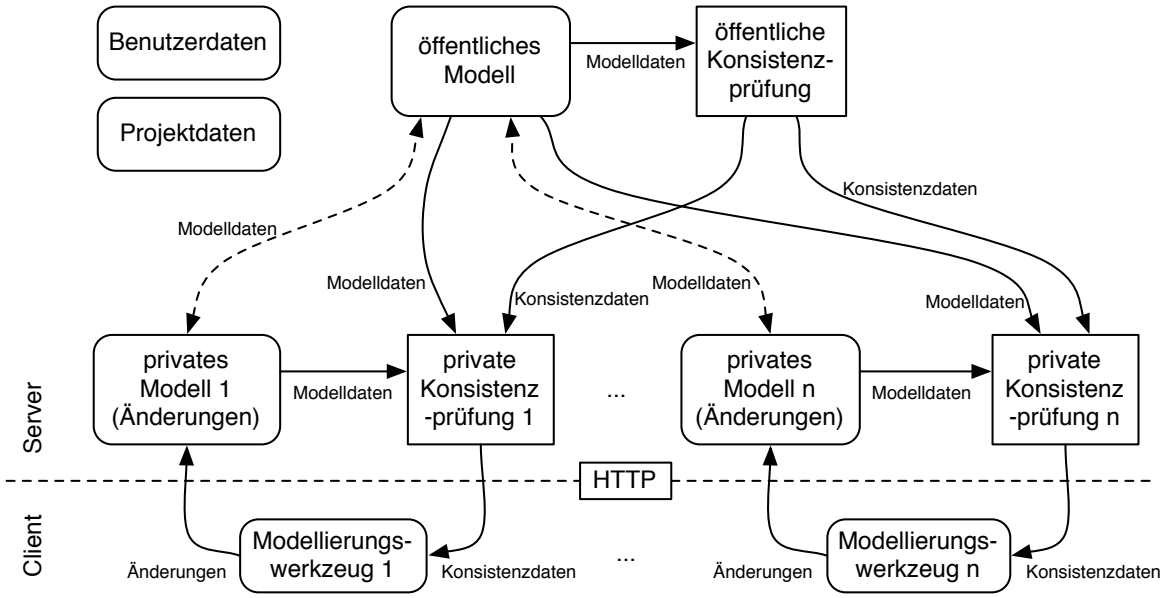


Abbildung 5.3: Architektur

5.2.1 Server

Die Hauptaufgabe des Servers ist die Konsistenzprüfung von Software Design Modellen. Dazu werden Modelle in Bereiche aufgeteilt und für jeden dieser Bereiche gibt es eine zugehörige Konsistenzprüfung. Benutzer senden Änderungen am Modell, diese Änderungen werden in privaten Bereichen verwaltet. Damit mehrere Modelle gleichzeitig geprüft werden können, sind die Modelle in Projekten organisiert. Da alle Anfragen personalisiert werden müssen, werden Dienste für die Benutzerverwaltung angeboten. Alle Dienste des Servers werden via HTTP erreichbar sein. Um die Schnittstellen einfach gehalten zu können, orientiert sich der Aufbau der Dienste an REST (siehe Punkt 3.3.3). Die HTTP-Kommunikation wird eine Authentifizierung verlangen, damit jeder Request einem Benutzernamen zugeordnet werden kann. Zusätzlich wird jeder URL an den Server der Projektname angehängt, damit auch eine Zuordnung zum richtigen Projekt stattfinden kann. In den nachfolgenden Punkten werden die einzelnen Dienste des Servers ausführlich beschrieben.

Benutzer- und Projektdaten

Am Server werden Benutzer- und Projektdaten gespeichert. Diese Daten können die einzelnen Clients ändern. Die Benutzerdaten benötigen mindestens Benutzername und Passwort, da jeder Zugriff auf den Server personalisiert stattfindet und der Server diese Daten für eine Authentifizierung benötigt. Die Authentifizierung wird über HTTP realisiert.

Die einzelnen Modelle und Modelldateien werden in Projekten verwaltet. Projekte sind Ordner am Server, in denen die zusammengehörigen Daten gesammelt werden. Für die Verwaltung ist mindestens ein Projektname notwendig. Zu den Projekten werden auch die Modelldateien gespeichert werden. Diese Modelldateien können von den Clients durch Export- und Importdienste übertragen werden. Die Modelldateien sind notwendig, damit die Clients das Modell austauschen können, zur eigentlichen Konsistenzprüfung werden diese Dateien nicht verwendet, es wird ein Modell in einem eigenen Datenformat verwendet.

öffentlicher und privater Bereich

Damit nicht jeder Benutzer ein komplettes Modell benötigt und somit der Server überlastet wird, wird das Modell in zwei Bereiche unterteilt. Es wird ein öffentlichen Bereich für alle Benutzer angelegt und pro Benutzer ein privater Bereich zugewiesen. Der private Modellbereich beinhaltet alle Änderungen des Benutzers am öffentlichen Modell, das heißt die Kombination von privaten und öffentlichen Modellbereich würde das Modell ergeben, welches der Benutzer in seinem lokalen Modellierungswerkzeug sieht und bearbeitet. Die Modellbereiche existierten nicht als Datei am Server, sie sind im Speicher geladene Modelle. Der Aufbau dieser geladenen Modelle wird in einem eigenen Unterpunkt genauer beschrieben. Die einzelnen Modellbereiche brauchen keine Verbindung zueinander, denn der Client lädt zu Beginn die zu dem Projekt zugehörige Modelldatei vom Server und besitzt somit den öffentlichen Bereich. Danach kommen alle Änderungen in den privaten Bereich, die Unterscheidung welche Modelldaten in welchen Bereich gehören ist somit getroffen. Ein weiterer Grund warum die Modellbereiche keine Verbindung benötigen ist, dass die Konsistenzprüfungen wissen auf welchen Bereich sie zugreifen müssen, um das richtige Modellelement zu erhalten.

Konsistenzprüfung

In der Mehrbenutzerumgebung übernimmt der Server die Konsistenzprüfung der Software Design Modelle. Basierend auf der Aufteilung der Modelle in einen öffentlichen und mehrere privaten Bereiche, wird auch die Konsistenzprüfung entsprechend gestaltet. Für die Implementierung der Konsistenzprüfung wird das Model Analyzer Framework aus Punkt 3.1 angepasst. Im Gegensatz zu den Modellbereichen haben die Konsistenzprüfungen eine Beziehung zueinander. Jede private Konsistenzprüfung kann auf die Daten der öffentlichen Konsistenzprüfung zugreifen. Die öffentliche Konsistenzprüfung greift nur auf den öffentlichen Modellbereich zu und wird daher nur dann ausgeführt, wenn das öffentliche Modell initialisiert wird. Die privaten Konsistenzprüfungen greifen hingegen auf beide Modellbereiche zu, fehlt im privaten Modellbereich eine Informationen, wird auf den öffentlichen Bereich zugegriffen. Zusätzlich verwenden die einzelnen privaten Konsistenzprüfungen auch die öffentliche Konsistenzprüfung, um bereits vorhandene Daten wiederverwenden zu können. Die privaten Konsistenzprüfungen werden durch die Änderungen der einzelnen Benutzer angestoßen. Alle Konsistenzprüfungen teilen sich die gleichen Konsistenzregeln, diese sind am Server fix vorgegeben.
Kommt ein neuer Benutzer zum System, importiert er die Modelldateien in sein Modellierungswerkzeug und kann sofort loslegen. Die importierten Modelldateien entsprechen dem öffentlichen Bereich und seine Änderungen kommen in einen eigenen privaten Bereich. Vom neuen Benutzer können auch alle bereits evaluierten, öffentlichen Regelinstanzen abgerufen und verwendet werden, auch seine private Konsistenzprüfung wird alle öffentlichen

Daten verwenden. Der Server bietet einen Dienst zum Abrufen der Konsistenzregeln sowie deren Instanzen. Dieser Dienst liefert die Daten personalisiert und als Delta aus, damit jeder Benutzer seine Daten bekommt und die Datenmenge nicht zu groß wird.

Modelldaten

Durch den Export- und Importmechanismus können Benutzer untereinander Modelldateien austauschen. Der Aufbau und die Handhabung dieser Modelldateien ist abhängig von den verwendeten Modellierungswerkzeugen der einzelnen Benutzer. Da die Konsistenzprüfung am Server auf alle Modelldaten zugreifen muss und damit der Server nicht alle Formate kennen und öffnen muss, wird ein eigenes Format verwendet. Es handelt sich um ein Tripel-Format, das heißt alle Modellelemente werden in Tripel zerteilt und so können die einzelnen Konsistenzprüfungen auf die benötigten Modellelemente zugreifen. Für den Zugriff auf die Tripel muss das Model Analyzer Framework entsprechend angepasst werden. Tripel haben den Vorteil, dass sie einfach als dreispaltige Tabelle darstellbar sind und an die Implementierung am Server keine speziellen Anforderungen stellen. Wenn ein Benutzer das öffentliche Modell initialisiert, dann wird das in seinem Client geöffnete und spezifische Modell in das Tripel-Format des Server konvertiert und zum Server gesendet. Die Änderungen am lokalen Modell werden auch in das Format konvertiert und in den entsprechenden Modellbereich gespeichert. Die verwendeten Tripel bestehen aus den folgenden drei Teilen:

- Element: Der erste Teil des Tripels sagt aus, zu welchem Modellelement das Tripel gehört, üblicherweise ist dies eine Identifikationsnummer des Modellelements.
- Eigenschaft: Eigenschaft sagt aus, um welches Merkmal des Modellelements es sich handelt.
- Wert: Das letzt Feld ist der Wert der Eigenschaft, dies kann ein primitiver Wert oder auch ein weiteres Element sein, damit komplexe Strukturen aufgebaut und Verbindungen zwischen den Tripel ausgedrückt werden können.

Die Darstellung von Modelldaten als Tripel lässt sich am Besten mit einem Beispiel erklären, dazu wird die Klasse *Light* aus dem Beispielmodell (Punkt 3.1.2) verwendet. Die Tabelle 5.1 zeigt die Aufschlüsselung der Klasse in Tripel.

Element	Eigenschaft	Wert
k1	Name	Light
k1	Methode	m1
k1	Methode	m2
m1	Name	turn-on
m2	Name	deactivate

Tabelle 5.1: Klasse Light in Tripel-Darstellung

Die Tabelle mit den Tripel beinhaltet alle sichtbaren Elemente des Modellelements. In der ersten Zeile ist der Name der Klasse angegeben, gefolgt von zwei Links auf weitere Tripel. Diese Links sind für die Methoden und notwendig, da Methoden wieder ein komplexes Element sind. Die letzten beiden Zeilen beinhalten die Methodennamen. Die Tabelle wird umfangreicher, wenn auch für jede Methode die Eingabe- und Rückgabeparameter mit

aufgeschlüsselt werden. Jede weitere Komponente eines Klassendiagramms wird in gleicher Art und Weise als Tripel aufgeschlüsselt. Wäre die Klasse zum Beispiel eine private Klasse, würde die Tabelle um das Tripel *k1.Sichtbarkeit=private* erweitert werden.

Änderungen

Der Server verlangt, dass die Änderungen in einer Gruppe von Tripeln gesendet werden. Die Änderungen verwenden die selben Tripel wie der Server. Die einzelnen Änderungen können auch mit einem Zeitstempel versehen werden, damit sie am Server in der richtigen Reihenfolge verarbeiten werden können. Wie der Server bei Änderungen reagiert, ist in Punkt 5.3 beschrieben. Änderungen lassen sich in drei Hauptkategorien einteilen:

- Hinzufügen - ADD: Beim Hinzufügen wird am Server ein neues Tripel im entsprechenden Bereich erstellt.
- Ändern - MODIFY: Sollte das Tripel noch nicht im entsprechenden Bereich sein wird es erstellt, ansonsten wird der Wert aktualisiert.
- Löschen - DELETE: Beim Löschen wird das entsprechende Tripel aus dem privaten Bereich entfernt. Sollte ein Tripel aus dem öffentlichen Bereich betroffen sein, ist ein spezielles Vorgehen notwendig.

Der Ablauf einer Änderung, wird nachfolgend anhand eines Beispiels demonstriert. Als öffentliches Modell wird das Beispiel aus dem vorangegangen Punkt verwendet, also Tabelle 5.1.

Kategorie	Element	Eigenschaft	Wert
MODIFY	m1	Name	activate
ADD	k1	Attribut	a1
ADD	a1	Name	pulp

Tabelle 5.2: Änderungsgruppe 1 in Tripel-Darstellung

Die erste Änderungsgruppe (Tabelle 5.2) ändert den Namen der Methode von *turn-on* auf *activate* und fügt der Klasse ein neues Attribut mit dem Namen *pulp* hinzu.

Kategorie	Element	Eigenschaft	Wert
MODIFY	a1	Name	counter
MODIFY	m2	Sichtbarkeit	public

Tabelle 5.3: Änderungsgruppe 2 in Tripel-Darstellung

Bei der zweiten Änderungsgruppe (Tabelle 5.3) wird der Name des Attributs in *counter* erneut geändert und die Sichtbarkeit der Methode *deactivate* wird auf *public* gesetzt.

Element	Eigenschaft	Wert
m1	Name	activate
k1	Attribut	a1
a1	Name	counter
m2	Sichtbarkeit	public

Tabelle 5.4: privater Bereich nach den Änderungen

Tabelle 5.4 zeigt den privaten Bereich nach den beiden Änderungen. Wenn jetzt für den Benutzer ein Gesamtmodell gebildet werden soll, würden die Tabellen 5.4 und 5.1 kombiniert werden. Zu Beachten ist, dass die Änderungstabelle, also der private Bereich des Benutzers, alle Einträge der Tabelle des öffentlichen Bereichs überschreiben würden. Das Ergebnis ist in Tabelle 5.5 dargestellt und entspricht dem Modell, wie es der Benutzer in seinem Modellierungswerkzeug sieht.

Element	Eigenschaft	Wert
k1	Name	Light
k1	Attribut	a1
a1	Name	counter
k1	Methode	m1
k1	Methode	m2
m1	Name	activate
m2	Name	deactivate
m2	Sichtbarkeit	public

Tabelle 5.5: Kombination privater und öffentlicher Bereich

Abgrenzung

Die Konzeption des Servers zeigt, welche Bereiche in dieser Arbeitet behandelt werden und welche nicht. Natürlich gibt es noch weitere Funktionen, welche der Server zu Verfügung stellen kann, aber der Fokus der Arbeit liegt auf der Konzeption und Implementierung der Konsistenzprüfung in einer Mehrbenutzerumgebung. Die Mehrbenutzerumgebung wirft viele weitere Fragen auf, wie zum Beispiel das Rückführen eines privaten Bereichs in den öffentlichen Bereich oder dass die verschiedenen Konsistenzprüfungen auch verschiedene Konsistenzregeln verwenden können. Einen Ausblick, welche Bereiche nicht abgedeckt wurden, beziehungsweise wie die Arbeit fortgeführt werden kann gibt der Punkt 9.1 am Ende der Arbeit.

5.2.2 Client

Dem Client kommt in der Architektur eine weniger große Bedeutung zu und daher soll er einfach aufgebaut sein. Wie im vorangegangen Kapitel in Punkt 4.1.2 beschrieben, handelt es sich um eine Art Thin-Client, denn der Hauptverwendungszweck ist die Ein- und Ausgabe von Daten. Konkret wird ein Modellierungswerkzeug zum Bearbeiten des Modells benötigt. Die wichtigsten Anforderungen an das Werkzeug sind einerseits die Fähigkeit fortlaufende Änderungen am Modell feststellen zu können und andererseits die Möglichkeit das Werkzeug einfach erweitern zu können. Die aufgezeichneten Änderungen müssen in das Tripel-Format des Servers konvertiert und mittels HTTP zum Server gesendet werden können. Damit die Änderungen dem richtigen Bereich am Server zugeordnet werden können, müssen sie im HTTP-Request an den Server den Benutzernamen beinhalten und die URL muss zum Schluss den gewünschten Projektnamen beinhalten. Zusätzlich sollen Konsistenzdaten vom Server empfangen und angezeigt werden können. Eine weitere Aufgabe des Clients wird die Konfiguration des Servers sein. Der Client wird auch eine eigene Benutzerschnittstelle mit folgenden Elementen zu Verfügung stellen:

- Funktionsmenü: Der Client stellt ein Menü zum Ausführen der wichtigsten Funktionen, wie zum Beispiel die Initialisierung des öffentlichen Modells oder das Starten der Datenübertragung, zu Verfügung.
- Informationsanzeige: Es sollen alle für den Benutzer gültigen Konsistenzregeln sowie deren Regelinstanzen angezeigt werden. Diese Anzeige muss aktualisierbar sein, denn jede Änderung am Modell kann Änderungen an den Regelinstanzen verursachen.
- Benutzer- und Projektverwaltung: Beide Datensätze sollen vom Client aus administrierbar sein. Die wichtigsten Funktionen sind das Hinzufügen, Ändern und Entfernen der einzelnen Daten.
- Import und Export: Damit die Modelldateien einfach zum und vom Server geladen werden können, ist je ein Dialog zum Importieren und zum Exportieren der Modelldateien vorgesehen.

5.3 Modellelemente und Regeln

Im vorangegangenen Kapitel wurde in Punkt 4.3 anhand eines Beispiels gezeigt, wie sich Konsistenzregeln und deren Regelinstanzen in der Mehrbenutzerumgebung verhalten können. Für die Konzeption des System ist es nun auch wichtig, genau zu definieren, wie sich die Operationen an den Modellelemente auf die Konsistenzregeln und deren Regelinstanzen in den verschiedenen Bereichen auswirken. Die Operationen an den Konsistenzregeln sind an die Operationen an den Modellelementen gekoppelt. Wird ein Modellelemente hinzugefügt, geändert oder gelöscht, so müssen auch die betroffenen Konsistenzregeln entsprechend verarbeitet werden. Zusätzlich werden auch die Besonderheiten der Speicherung der Modellelemente berücksichtigt. Bei der Konzeption der Architektur in Punkt 5.2 wurde die Darstellung des Modells und der Modellelemente in Tripel eingeführt. Da Modellelemente aus Tripel bestehen, kann das Verhalten der Operationen und der Zugriff auf die Modellelemente auch auf das Verhalten der Tripel angewendet werden.

Damit die einzelnen Regelinstanzen evaluiert werden können, müssen sie auf die richtigen Modellelemente zugreifen können. Die Regelinstanzen werden immer den entsprechenden Bereichen zugeordnet. Wenn eine Regelinstanz einem privaten Bereich zugeordnet ist, greift sie zuerst auf die privaten Modellelemente zu. Wird ein Modellelemente im privaten Bereich nicht gefunden, wird auf das öffentliche Modell zugegriffen. Die Modellbereiche können mit übereinander gelegten Papierblättern verglichen werden. Das private Modell liegt oben und verdeckt Teile des öffentlichen Modells. Der Zugriff beginnt auf dem ersten Blatt, dem privaten Modell. Werden die benötigten Informationen nicht gefunden, wird einfach umgeblättert und es wird auf dem darunter liegenden Blatt, dem öffentlichen Modell, weiter gesucht. Wenn eine Regelinstanz dem öffentlichen Bereich zugeordnet ist, werden nur Modellelemente aus dem öffentlichen Bereich verwendet. Das heißt, eine private Regelinstanz darf sowohl auf private, als auch auf öffentliche Modelldaten zugreifen, während eine öffentliche Regelinstanz nur auf öffentliche Modelldaten zugreifen darf.

5.3.1 Hinzufügen

Neue Modellelemente werden immer in den privaten Bereich des Benutzers gespeichert, nur beim Initialisieren des öffentlichen Modells werden die Modellelemente in den öffentlichen Bereich gespeichert. Das Hinzufügen von Modellelementen erfordert keine spezielle Behandlung der Regelinstanzen. Für jedes neue Modellelement werden die Konsistenzregeln durchsucht, ob es passende Regeln gibt. Sobald eine passende Regel gefunden ist, wird von dieser Konsistenzregel eine neue Regelinstanz erzeugt und evaluiert. Bei der Initialisierung des öffentlichen Modells werden öffentliche Regelinstanzen erzeugt, kommen neue Modellelemente in die privaten Bereiche, werden private Regelinstanzen für den jeweiligen Benutzer erzeugt.

5.3.2 Ändern

Änderungen werden immer im privaten Bereich durchgeführt, denn die einzelnen Benutzer dürfen nicht den öffentlichen Bereich des Modells ändern. Bei einer Änderung eines Modellelements wird im privaten Modell das betroffene Element gesucht und wenn es vorhanden ist, wird es geändert. Alle privaten Regelinstanzen, welche das geänderte Modellelemente verwenden, müssen neu evaluiert werden. Sollte das geänderte Modellelement noch nicht im privaten Bereich vorhanden sein, betrifft die Änderung ein Element aus dem öffentlichen Bereich. Das zu ändernde Element wird aus dem öffentlichen in den privaten Modellbereich kopiert und dann kann die Änderung im privaten Bereich durchgeführt werden. Das geänderte Modellelemente überdeckt nun das ursprüngliche Modellelement aus dem öffentlichen Modell. Wurde das Element aus dem öffentlichen Bereich übernommen, müssen auch die Regelinstanzen speziell behandelt werden. Es werden alle Regelinstanzen des öffentlichen Bereich gesucht, welche durch die Änderung betroffen wären, wenn die Änderung im öffentlichen Bereich durchgeführt werden würde. Diese Regelinstanzen werden in den privaten Bereich kopiert und überdecken somit die öffentlichen Regelinstanzen. Bei der Evaluierung der kopierten Regelinstanzen werden nun auch die privaten Modellelemente verwendet, sind diese nicht vorhanden, wird auf das öffentliche Modell zurückgegriffen. Die Regelinstanz verwendet auf jeden Fall das kopierte und geänderte Modellelement aus dem öffentlichen Bereich, welches nun im privaten Bereich ist.

5.3.3 Löschen

Beim Löschen von Modellelementen muss unterschieden werden, aus welchen Bereich das Modellelement gelöscht wird, denn die einzelnen Benutzer dürfen nicht direkt aus dem öffentlichen Bereich löschen. Je nach Bereich müssen die Modellelemente und Regelinstanzen unterschiedlich behandelt werden.

- öffentlicher Bereich: Befindet sich das zu löschende Modellelement im öffentlichen Bereich, wird es dort nicht gelöscht, stattdessen wird eine Kopie im privaten Bereich angelegt und als gelöscht markiert. Diese Vorgangsweise ist notwendig, da andere Benutzer das Element noch benötigen und es nicht gelöscht werden darf. Durch den bereits beschriebenen Mechanismus, wie auf ein Modellelement zugegriffen wird, bleibt es durch die Markierung dem Benutzer verborgen. Da primär Elemente aus dem privaten Bereich verwendet werden, kann nun festgestellt werden, wenn ein Element nicht vorhanden ist, ob noch im öffentlichen Bereich zugegriffen werden darf, oder ob das Element für den Benutzer bereits gelöscht ist. Zusätzlich werden im öffentlichen Bereich alle Regelinstanzen gesucht, welche durch das Löschen des Modellelements betroffen wären. Es wird in der privaten Konsistenzprüfung eine Liste mit allen betroffenen Regelinstanzen gespeichert. Aus der Sicht des Benutzers gibt es diese Regelinstanzen nicht mehr, obwohl sie im öffentlichen Bereich natürlich noch existieren.

- privater Bereich: Wenn das zu löschende Modellelement im privaten Bereich eines Benutzers ist, kann es einfach aus dem privaten Modell gelöscht werden. Zusätzlich muss auch der öffentliche Bereich kontrolliert werden, denn das zu löschende Modellelement könnte ein geändertes Element aus dem öffentlichen Bereich gewesen sein, das bedeutet es hat ein öffentliches Element überdeckt und dieses öffentliche Element darf auch nicht mehr für den Benutzer existieren. In diesem Fall ist die Vorgehensweise wie im vorangegangen Absatz beschrieben. Zusätzlich werden im privaten Bereich alle Regelinstanzen gesucht, welche ein zu löschendes Modellelemente beinhalten. Diese Regelinstanzen können einfach gelöscht werden und stehen dem Benutzer damit nicht mehr zu Verfügung.

Kapitel 6

Implementierung

Dieses Kapitel beschreibt die wichtigsten Punkte der Implementierung des Prototyps. Die Implementierung verwendet das Konzept aus Punkt 5. Wie auch schon im Konzept ersichtlich, liegt das Hauptaugenmerk der Implementierung auf dem Server und der dortigen Konsistenzprüfung. Die wichtigsten Punkte der Implementierung sind die Kommunikation zwischen Client und Server, die Verarbeitung des Modells, die Anpassungen des Model Analyzer Frameworks am Server und schlussendlich die Verwaltung der Benutzer- und Projektdateien.
Die Implementierung verwendet die in Kapitel 3 beschriebenen Technologien. Auf der Clientseite wird der Eclipse-basierte Rational Software Modeler 7.5 (RSM) von IBM verwendet. RSM verwendet die Eclipse-Version 3.4 (Ganymede). Alle Erweiterungen am Client werden als Plug-in zu RSM hinzugefügt. Auch der Server wird als Plug-in für RSM entwickelt, jedoch wird der Server als OSGi-Anwendung gestartet und kommt ohne grafische Benutzeroberfläche aus. Alle Informationen werden auf die Konsole ausgegeben. Dadurch dass der Server eine OSGi-Anwendung ist, kann die Anwendung die Eclipse-Infrastruktur, wie zum Beispiel den Dateizugriff über Projekte, verwenden. Damit der Server die Modelle in Tripel darstellen kann, wurde nach einer Lösung gesucht, wie diese Informationen verarbeitet und dargestellt werden kann. Die Entscheidung ist auf RDF gefallen. RDF ist aufgrund seiner Offenheit und der Möglichkeit beliebige Information darstellen zu können, gewählt worden. Um nicht noch zusätzliche Bibliotheken für die Übertragung aller anderen Daten verwenden zu müssen, werden auch die Kommunikationsdaten als RDF-Modell dargestellt. Für die RDF-Funktionalität wird die Bibliothek Jena verwendet. Damit die Konsistenzprüfung am Server auf die Modelldaten zugreifen kann, muss sie auch RDF unterstützen. Die Konsistenzprüfung basiert auf dem Model Analyzer Framework und wurde sowohl für RDF, als auch für die weiteren Anforderungen an die Konsistenzprüfung in der Mehrbenutzerumgebung angepasst. Die Designregeln für die Konsistenzprüfung müssen in OCL angegeben werden.

6.1 Kommunikation

Einer der ersten Teile der Implementierung war die Kommunikation. Als Kommunikationsprotokoll wird HTTP verwendet. Der Server ist eine OSGi-Anwendung mit einem integrierten Jetty-Webserver und stellt seine Dienste mittels Servlets zu Verfügung. Die Implementierung der Servlets orentiert sich an REST (Punkt 3.3.3), vor allem wird die Idee der Zuordnung der HTTP-Methoden zu Operationen verwendet. Der Client verwendet zum Erstellen der HTTP-Requests den HttpClient von Apache. Alle benötigten URLs

des Servers werden über einen Einstellungsdialog verwaltet.
Auf dem Client werden alle Methoden für die Kommunikation von einer Klasse gekapselt. Diese Klasse stellt für jeden Aufruf eines Servlets eine HTTP-Methode zu Verfügung. Bei PUT- und POST-Requests muss noch eine Entity für den Inhalt des Requests angegeben werden. Die HTTP-Methode kann dann, unter Angabe einer Entity für den Response, ausgeführt werden. Für jede Art des Inhalts einer Antwort gibt es angepasste Response-Handler zur Verarbeitung der Entity. Wenn zum Beispiel ein RDF-Modell vom Server empfangen wird, wird das Modell aus der Entity ausgelesen und der Response-Handler liefert das fertige RDF-Modell zurück. Da der HttpClient ein blockierendes Verfahren zur Kommunikation verwendet, werden die HTTP-Requests und die Abarbeitung der HTTP-Responses in einem eigenen Thread ausgeführt. Würde die Kommunikation nicht in einem eigenen Thread ausgeführt, täte sie die kurzzeitig GUI blockieren.
Bevor ein Request abgeschickt werden kann, muss er noch personalisiert werden. Der Server verlangt zur Authentifizierung einen Benutzernamen und ein Passwort, es wird daher Basic-Authentication verwendet. Basic-Authentication ist das einfachste Verfahren zur Authentifizierung in HTTP und die Benutzerdaten werden dabei unverschlüsselt übertragen. Der genaue Ablauf für das Erzeugen einer GET-Methode mit Basic-Authentication ist in Quellcode 6.1 dargestellt. Dem neuen HTTP-Request wird ein zusätzliches Headerfeld *Authorization* hinzugefügt. Der Wert des neuen Felds ist ein Base64-codierter String mit Benutzername und Passwort und wird mit der Methode *getAuthorizationHeader()* erzeugt. Beispielsweise wird für den Benutzernamen *alice* mit dem Passwort *password* der String *Basic YWxpY2U6cGFzc3dvcmQ=* erzeugt.

```
1  import org.apache.commons.codec.binary.Base64;
2
3  public HttpGet createGet(String uri, String user, String pass) {
4    HttpGet httpGet = new HttpGet(uri);
5    ((HttpMessage) httpGet).addHeader("Authorization",
       getAuthorizationHeader(user, pass));
6    return httpGet;
7  }
8
9  private String getAuthorizationHeader(String user, String pass) {
10   StringBuffer input = new StringBuffer();
11   input.append(user).append(":").append(pass);
12   String base64 = new String(Base64.encodeBase64(input.toString().
       getBytes()));
13
14   return "Basic " + base64;
15 }
```

Quellcode 6.1: Basic-Authentication

Nach dem Personalisieren kann der fertige HTTP-Request zum angegebenen Servlet am Server gesendet werden. Als erstes muss der Request am Server durch einen Kontext, dieser Kontext ist für die Authentifizierung zuständig. Der Kontext implementiert das Interface *HttpContext*, welches bereits in Punkt 3.3.2 beschrieben wurde. Da keine statischen Inhalte verwendet werden, wird am Server nur die Methode *handleSecurity()* verwendet (Quellcode 6.2).

```
1  @Override
2  public boolean handleSecurity(HttpServletRequest req,
       HttpServletResponse resp) throws IOException {
3    String auth = req.getHeader("Authorization");
4    if (auth == null) return failAuthorization(req, resp);
5
6    StringTokenizer tokens = new StringTokenizer(auth);
7    String authscheme = tokens.nextToken();
8    if (!authscheme.equals("Basic")) return failAuthorization(req, resp);
9
10   String base64credentials = tokens.nextToken();
11   String credentials = new String(Base64.decodeBase64(base64credentials.
       getBytes()));
12   int colon = credentials.indexOf(':');
13   String username = credentials.substring(0, colon);
14   String password = credentials.substring(colon + 1);
15   Resource userData = RDFUserServer.getDefault().login(username,
       password);
16   if (userData == null) return failAuthorization(req, resp);
17
18   req.setAttribute(HttpContext.REMOTE_USER, username);
19   req.setAttribute(HttpContext.AUTHENTICATION_TYPE, authscheme);
20   req.setAttribute(HttpContext.AUTHORIZATION, userData.getRole());
21
22   return true;
23 }
```

Quellcode 6.2: Methode zur Authentifizierung

Die Funktionsweise einer Authentifizierung mit der *handleSecurity()*-Methode ist dem Buch [MVA10] entnommen und angepasst worden. Als erstes wird geprüft, ob der Request-Header das Feld *Authorization* enthält, sollte dieses Feld und somit auch keine Benutzerdaten im Request vorhanden sein, wird die Authentifizierung abgebrochen. Die aufgerufene *failAuthorization()*-Methode setzt beim Response den HTTP-Status 401 für eine fehlgeschlagene Authentifizierung und liefert *false* zurück. Der Request wäre somit nicht authentifiziert worden. Wenn das Feld *Authorization* vorhanden ist wird der Wert des Felds geprüft, ob er mit dem Schlüsselwort *Basic* für die Basic-Authentication beginnt. Ist das Schlüsselwort vorhanden kann die Authentifizierung weiter arbeiten, ansonsten wird sie abgebrochen. Im nächsten Schritt müssen Benutzername und Passwort Base64-decodiert werden. Sind Benutzername und Passwort aus dem Header ermittelt, werden sie dem Dienst für die Benutzerverwaltung übergeben. Bei erfolgreicher Authentifizierung gibt die Benutzerverwaltung die Benutzerdaten zurück, bei erfolgloser Authentifizierung wird die Kommunikation zurückgewiesen. Der HTTP-Request wird abschließend noch um drei Attribute erweitert, diese Attribute können die nachfolgenden Servlets verwenden:

- REMOTE_USER: Benutzername des authentifizierten Benutzers
- AUTHENTICATION_TYPE: verwendete Authentifizierungsmethode
- AUTHORIZATION: Rolle des Benutzers

Bei erfolgreicher Authentifizierung wird die Anfrage an das entsprechende Servlet weitergeleitet. Die Beschreibung der einzelnen Servlets ist in den einzelnen Unterpunkten zu finden, zu denen das jeweilige Servlet zugeordnet werden kann.

6.2 Modellverarbeitung

Die Modellverarbeitung findet sowohl am Client, als auch am Server statt. Im Konzept wurde festgelegt, dass die Modelldaten unabhängig von Darstellungsarten am Client gespeichert werden sollen, deshalb wird am Server RDF mit *Jena* verwendet. Am Client muss einerseits die Initialisierung und andererseits die Verarbeitung von Änderungen durchgeführt werden. Bei der Initialisierung wird das geladene Modell in RDF konvertiert und zum Server gesendet, bei Änderungen werden diese zu RDF konvertiert und ebenfalls zum Server gesendet. Der Ablauf der Modellverarbeitung ist in Abbildung 6.1 zusammengefasst.

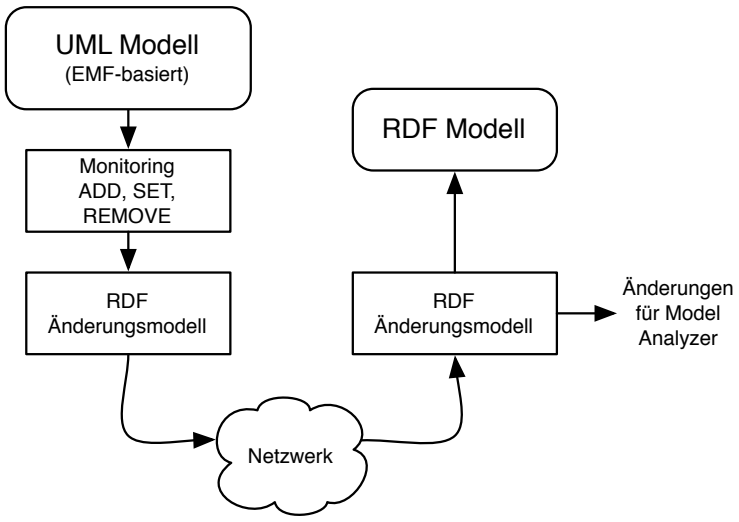


Abbildung 6.1: Modellverarbeitung

Die Verarbeitung und Konvertierung des Modells beginnt am Client. Das Modell liegt in einem Eclipse-spezifischen Format vor. Eclipse und somit auch der RSM verwenden für die Modellierung von UML-Diagrammen das Eclipse Modeling Framework (EMF). Für diese Modelle gibt es ein Monitoring, um das Hinzufügen, Ändern und Löschen von Modellelementen feststellen zu können. Sobald eine Änderung festgestellt wird, wird diese in RDF konvertiert und so entsteht ein RDF-Modell für die Änderungen. Bei der Initialisierung wird das geladene Modell von EMF in RDF umgewandelt. Die am Client erstellten RDF-Modelle werden zum Server gesendet. Am Server wird das ankommende RDF Modell bei der Initialisierung in das öffentliche Modell eingetragen, Änderungen werden in das private RDF-Änderungsmodell des jeweiligen Benutzers eingetragen. Wenn am Server ein RDF-Modell ankommt und ausgewertet wird, werden auch die Änderungen für die Konsistenzprüfung generiert. Die weitere Verarbeitung in der Konsistenzprüfung wird in Punkt 6.3 beschrieben, die Modellverarbeitung wird in Punkt 6.2.1 für den Client und in Punkt 6.2.2 für den Server noch weiter vertieft.

6.2.1 Client

Die Modellverarbeitung beginnt am Client mit dem Monitoring der Änderungen am Modell, beziehungsweise mit dem Initialisieren des Modells. Bei der Initialisierung werden alle Elemente des geöffneten Modells durchlaufen und die Methode *init()* der RDF-Generierung in Quellcode 6.3 aufgerufen. Änderungen am Modell werden mit einer Klasse,

welche das Interface *ResourceSetListener* implementiert, festgestellt. Das Monitoring reagiert auf das Hinzufügen, Ändern und Entfernen von Modellelementen. Tritt eine dieser Operationen auf, wird die entsprechende Methode aus Quellcode 6.3 zur RDF-Generierung aufgerufen.

```
1  public void init(EObject element) {
2    modifyComplexResource(element, ChangeType.INIT);
3  }
4
5  public void add(EObject element, EStructuralFeature feature, EObject
       newElement) {
6    modifyComplexResource(newElement, ChangeType.ADD);
7    modifyEList(element, feature, newElement, ChangeType.SET_ADD);
8  }
9
10 public void remove(EObject element, EStructuralFeature feature, EObject
       oldElement) {
11   modifyComplexResource(oldElement, ChangeType.REMOVE);
12   modifyEList(element, feature, oldElement, ChangeType.SET_REMOVE);
13 }
14
15 public void set(EObject element, EStructuralFeature feature, Object
       newValue) {
16   createResourceFromFeature(element, feature, newValue, ChangeType.SET);
17  }
```

Quellcode 6.3: Methoden für die RDF-Generierung

Die Methoden zur Konvertierung benötigen zwei RDF-Namespaces. Der *MA*-Namespace definiert allgemeine Eigenschaften für die Ressourcen, diese sind der Änderungstyp *ma:changetype*, das übergeordnete Modellelement *ma:parent*, der Name des Features *ma:name*, einen Wert *ma:value* und den ursprünglichen Typ *ma:type*. Die Verwendung eines zweiten Namespaces ist notwendig, da es sonst Überschneidungen geben könnte. Der *UML*-Namespace generiert dynamische RDF-Eigenschaften für die einzelnen Features eines Modellelements. Ein typisches Modellelement hat unter anderem einen Namen *uml:name* und eine Sichtbarkeit *uml:visibility*. Neben den Eigenschaften benötigen die Ressourcen in RDF auch eine eindeutige URI. In EMF besitzen die Klassen zur Speicherung von Modellelementen eine eindeutige Identifikation. Diese Identifikationen der sogenannten EObjects wird für das Erstellen der URIs verwendet.

- modifyComplexResource: Diese Methode erzeugt bei der Initialisierung *INIT*, beim Hinzufügen *ADD* und beim Löschen *REMOVE* aus dem EObject *element* eine Ressource, EObjects sind die Container der Modellelemente. Die erzeugte Ressource beinhaltet den Änderungstyp und den Typ des, in dem EObject gespeicherten, Modellelements. Danach werden alle Features des EObject und somit des Modellelements durchgelaufen und für jedes Feature wird eine eigene Ressource angelegt, die Ressource des EObject *element* verweist mit den dynamischen Eigenschaften des *UML*-Namespace auf die neuen Ressourcen. Entweder kann das Feature eine Liste oder ein weiteres Elements sein. Wenn das Feature eine Liste ist, wird ein RDF-Bag mit den Listenelementen angelegt. Diese Listenelemente sind wiederum Links auf andere Ressourcen von EObjects. Ist das Feature ein weiteres Element, wird mittels *createResourceFromFeature()*-Methode eine Ressource mit *ma:parent*-Eigenschaft auf die ursprüngliche Ressource angelegt. Der Grund warum neben der

Initialisierung und dem Hinzufügen auch beim Löschen die ganzen Ressourcen angelegt werden ist, dass der Server die Information benötigt, welche Ressourcen zu löschen sind.

- modifyEList: Beim Hinzufügen und Löschen muss zusätzlich noch die übergeordnete Liste bearbeitet werden, welche die Ressourcen des eben angelegten EObjects beinhaltet. In der Funktion wird der Änderungstyp auf *SET_ADD* oder *SET_REMOVE* aktualisiert und ein Link auf die Ressource des EObject angelegt. Beispielsweise hat jedes Package in UML eine Liste mit allen Elementen des Diagramms. Wird ein neues Element hinzugefügt, ist genau diese Liste des Packages betroffen und muss geändert werden.

- createResourceFromFeature: Bei einer Änderung *SET* wird der Wert eines Features geändert. Jede Ressource für ein Feature enthält den Namen, Typ und Wert. Der Wert eines Features kann entweder wieder ein Link auf ein weiteres Modellelement und somit auf eine weitere Ressource, ein primitiver Wert oder *null* sein. Jedes Feature besitzt die *ma:parent*-Eigenschaft, um eine Verbindung zur übergeordneten Ressource herstellen zu können

Nach der Konvertierung von EMF nach RDF können die erzeugten RDF-Modelle mit Jena serialisiert und deserialisiert werden. Diese Funktionalität kommt auch bei der Kommunikation zum Einsatz. Es gibt mehrere Darstellungsarten der Serialisierung, eine davon ist XML und in Quellcode 6.4 ist ein Beispiel dieser Darstellung zu sehen. Hierbei handelt es sich um eine Änderung eines Namen einer Methode in einem Klassendiagramm. Zu Beginn stehen die verwendeten Namespaces, darunter auch der *UML*- und *MA*-Namespace. Als nächstes folgen die einzelnen Ressourcen mit der eindeutigen URI im *rdf:about*-Attribut. Innerhalb der Ressourcen folgen die Elemente für den neuen Wert der Ressource *ma:value*, dem Typ des Werts *ma:type*, dem Namen des geänderten Features *ma:name* und einen Link auf die übergeordnete Ressource *ma:changetpye*. Jede Ressource beinhaltet auch den Typ der Änderung und jede Änderung ist mit einem Zeitstempel versehen, damit die Ressourcen einer Änderung untereinander sortiert werden können.

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:uml="http://sea.uni-linz.ac.at/uml/"
  xmlns:ma="http://sea.uni-linz.ac.at/ma/" >
  <rdf:Description rdf:about=
      "http://sea.uni-linz.ac.at/uml/_JOdy0EiREd-Xye0lIHC77w/name">
    <ma:value>turn-on</ma:value>
    <ma:type>java.lang.String</ma:type>
    <ma:name>name</ma:name>
    <ma:parent rdf:resource=
        "http://sea.uni-linz.ac.at/uml/_JOdy0EiREd-Xye0lIHC77w"/>
    <ma:changetype>31535151952273-SET</ma:changetype>
  </rdf:Description>
</rdf:RDF>
```

Quellcode 6.4: RDF in XML-Darstellung

In Anhang A ist ein weiterer Ausschnitt eines RDF-Modells in XML-Darstellung zu finden. Es handelt sich um das Hinzufügen einer Klasse zu einem Modell. Zum Abschluss wird das RDF-Modell zum Server gesendet. Am Client wird das Änderungsmodell in der Klasse für die RDF-Generierung wieder geleert und damit ist sie bereit für die nächsten Änderungen des Benutzers.

6.2.2 Server

Der Server stellt für die Initialisierung und Änderungen das *ChangeServlet* zu Verfügung. Bei einer Initialisierung wird die POST-Methode des Servlets aufgerufen, für Änderungen ist die PUT-Methode vorgesehen. Beide Methoden sind in Tabelle 6.1 beschrieben. Damit die Zuordnung der Daten zum Projekt stattfinden kann, muss das Servlet mit einem Projektnamen in der URL aufgerufen werden. Würde das Servlet ohne Projektnamen in der URL aufgerufen werden, wird das Standardprojekt *default* verwendet.

http://Server/ChangeServlet/<Projektname>/	
POST	Initialisierung des öffentlichen Modells mit dem an den Request angehängten RDF-Modell.
PUT	Verarbeitung der Änderungen des Benutzers, welcher den Request abgesetzt hat. Die Änderungen müssen an den Request angehängt werden.

Tabelle 6.1: ChangeServlet

Das Servlet ruft Methoden des Dienst zur Verwaltung der Projekte auf. Pro Projekt wird eine Instanz der Klasse *RDFModel* angelegt. Diese Klasse verwaltet die einzelnen Bereiche mit den zugehörigen Modellen und Konsistenzprüfungen. Es wird eine *init()*-Methode für die Initialisierung und eine *update()*-Methode für Änderungen zu Verfügung gestellt. Das *ChangeServlet* muss den Methoden einen Stream für den Zugriff auf die RDF-Modelldaten und den Benutzernamen des aufrufenden Benutzers mitgeben. In diesen Methoden werden auch die Änderungslisten für den RDF Model Analyzer erstellt. Die weitere Handhabung der Änderungslisten ist in Punkt 6.3.2 beschrieben.

Initialisierung

Bei der Initialisierung wird in der Klasse *RDFModel* das öffentliche RDF-Modell und die öffentliche Konsistenzprüfung angelegt. Die Konsistenzprüfung bekommt Zugriff auf das RDF-Modell. In Quellcode 6.5 ist die Verarbeitung der Initialisierung zu sehen.

```
1 publicRDFModel.read(inputStream, null);
2 ResIterator iter = publicRDFModel.listSubjects();
3 while (iter.hasNext()) {
4   Resource resource = iter.nextResource();
5   if (resource.getProperty(MA.parent) == null) {
6     changes.add(new NewChange(resource));
7   }
8 }
9 validate(publicDataModel, changes);
```

Quellcode 6.5: Initialiserung am Server

Zu Beginn wird das öffentliche RDF-Modell aus dem Stream des POST-Requests ausgelesen, danach werden alle Ressourcen des öffentlichen Modells durchgelaufen. Jede Ressource ohne der Eigenschaft *MA.parent* entspricht einem Modellelement und daher kann solch eine Ressource auch ein Kontextelement einer Konsistenzregel sein. Für jede dieser Ressourcen wird eine *NewChange* in die Liste der Änderungen für den RDF Model Analyzer angelegt. Alle Ressourcen mit der Eigenschaft *MA.parent* sind Features des Modellelements und können keine Kontextelemente werden. Abschließend wird in der Methode *validate()* die öffentliche Konsistenzprüfung *publicDataModel* mit der Liste der Änderungen initial evaluiert.

Änderungen

Die privaten RDF-Änderungsmodelle und die zugehörigen Konsistenzprüfungen werden angelegt, sobald der jeweilige Benutzer die erste Änderungen zum Server senden. Bei einer Änderung muss das im PUT-Request gesendete RDF-Modell in das private RDF-Änderungsmodell des Benutzers eingetragen werden und es muss die Änderungsliste für den RDF Model Analyzer erzeugt werden. Je nach Änderungstyp wird die Ressource unterschiedlich in das private RDF Modell eingefügt. Es wird wie in Punkt 5.3 des Konzepts vorgegangen:

- ADD: Hinzufügen der Ressource in den privaten Bereich und Erzeugung eines *NewChange* für die Änderungsliste.
- REMOVE: Ist die zu löschende Ressource im privaten Modell vorhanden, kann sie dort gelöscht werden. Soll jedoch eine Ressource aus dem öffentlichen Modell gelöscht werden, wird die betroffene Ressourcen in den privaten Bereich kopiert und mit einer Löschmarkierung versehen. Es wird ein *DeleteChange* für den RDF Model Analyzer erzeugt.
- SET: Die alte Ressource wird durch die neue Ressource im privaten Modell ersetzt. Sollte die geänderte Ressource im privaten Modell noch nicht vorhanden sein, bedeutet dies dass eine öffentliche Ressource geändert wurde und dass die geänderte Ressource in den privaten Bereich hinzugefügt werden muss. Es wird ein *ModifyChange* für die Änderungsliste erzeugt.
- SET_ADD und SET_REMOVE: Beide Änderungstypen sind Änderungen von Listen. Ist die Liste bereits im privaten Bereich, wird das entsprechende Listenelement hinzugefügt oder entfernt. Ist die Liste noch nicht im privaten Modell vorhanden, wird sie komplett aus dem öffentlichen Bereich kopiert. Ab diesem Zeitpunkt wird im privaten Bereich die Liste verwaltet. Bei diesen Änderungstypen gibt es keine spezielle Änderungsart für den RDF Model Analyzer, es wird daher ein *ModifyChange* für die Änderungsliste erzeugt.

Abschließend wird die private Konsistenzprüfung mit der Liste der Änderungen evaluiert.

6.3 RDF Model Analyzer

Das Model Analyzer Framework wurde im ersten Schritt am Institut für RDF angepasst und danach in dieser Masterarbeit an die Mehrbenutzerumgebung optimiert. Die wichtigsten Anpassungen für die Mehrbenutzerumgebung sind in den folgenden Unterpunkten beschrieben. Diese sind der Modellzugriff auf die RDF-Modelle, das Verhalten der Regelinstanzen und der Zugriff auf die Instanzen. Die Hauptklasse des RDF Model Analyzer ist *RDFDataModel*, sie verwaltet alle Daten der Konsistenzregeln und deren Instanzen. Diese Klasse erweitert die abstrakte Klasse *AbstractDataModel<CT, ET, PT>* aus dem Framework, der Typ für Kontextelement *CT* und Modellelement *ET* ist die *Resource* aus Jena und der Typ der Eigenschaften *PT* ist das *Property* aus Jena. Für die öffentliche und private Konsistenzprüfung wird der gleiche RDF Model Analyzer verwendet und jede Konsistenzprüfung kann direkt auf das ihr zugeordnetes RDF-Modell zugreifen. Der

einzige Unterschied besteht darin, dass eine private Konsistenzprüfung über die Variable *parentRDFDataModel* die öffentliche Konsistenzprüfung zugreifen kann. Eine Konsistenzprüfung kann mit der Funktion *hasParentRDFDataModel()* feststellen, ob sie eine übergeordnete Konsistenzprüfung hat und somit selbst privat ist.
Der RDF Model Analyzer akzeptiert drei Arten von Änderungen, diese wurden bereits in Punkt 6.2.2 bei der Modellverarbeitung am Server verwendet. Bei allen drei Änderungsarten muss der Kontext der Änderung nicht angegeben werden, er wird aus dem Modellelement ermittelt.

- NewChange: Ein neues Modellelement wird zum Modell hinzugefügt. Die Änderung benötigt als Parameter beim Anlegen die RDF-Ressource des neuen Elements.
- ModifyChange: Eine Änderung benötigt das geänderte Modellelement als RDF-Ressource, das geänderte Feature des Modellelements sowie den alten und neuen Wert des Features.
- DeleteChange: Wenn ein Modellelement gelöscht wird, wird ein DeleteChange mit dem gelöschten Modellelement als RDF-Ressource benötigt.

6.3.1 Modellzugriff

Um eine Instanz einer Konsistenzregel auszuwerten zu können, muss sie auf Modellelemente und Konstanten zugreifen, da deren Werte benötigt werden. Wie in Punkt 3.1.1 beschrieben, ist eine Instanz ein Baum aus Ausdrücken und die Blätter sind entweder Konstanten oder PropertyCallExpressions für den Modellzugriff. Für den Zugriff auf RDF wird aus dem Model Analyzer Framework ein Ausdruck mit Scopeelement implementiert und er wird zur konkreten Klasse *RDFPropertyCallExpression*. Der implementierte Ausdruck greift bei der Auswertung über ein Scopeelement auf das RDF-Modell zu. Auch das Scopeelement muss aus dem Framework implementiert werden und das Resultat ist die Klasse *RDFScopeElement*.

```
1  if (dataModel.hasParentRDFDataModel()){
2    if (dataModel.hasScopeElement(element, property)) {
3      scopeElement = dataModel.getScopeElement(element, property);
4    }
5    if (scopeElement == null && dataModel.getParentRDFDataModel().
         hasScopeElement(element, property)) {
6      scopeElement = dataModel.getParentRDFDataModel().getScopeElement(
           element, property);
7    }
8    if (scopeElement == null) {
9      scopeElement = dataModel.getScopeElement(element, property);
10   }
11 } else {
12   scopeElement = dataModel.getScopeElement(element, property);
13 }
14 Resource value = (Resource) scopeElement.getValue();
```

Quellcode 6.6: Ermittlung des Scopeelements

Kern dieser beiden Klassen ist der RDF-Zugriff, daher zeigt Quellcode 6.6 den Teil der Auswertung der *RDFPropertyCallExpression*, in dem das richtige *RDFScopeElement* für

den Zugriff ermittelt wird. Jede *RDFPropertyCallExpression* kennt die Konsistenzprüfung *dataModel*, der sie zugeordnet ist. Wenn es sich um die öffentliche Konsistenzprüfung handelt, also wenn es keine übergeordnete Konsistenzprüfung mehr gibt, wird das *RDFScopeElement* direkt aus dem *dataModel* ermittelt. Die Methode *getScopeElement()* ermittelt anhand des angegebenen Modellelements und des Features das gewünscht Scopeelement. Falls es noch nicht existiert, wie zum Beispiel bei der initialen Evaluierung einer Regel, wird es neu angelegt. Wenn es sich um eine private Konsistenzprüfung handelt, gibt es drei Möglichkeiten wie das richtige Scopeelement ermittelt werden kann. Die private Konsistenzprüfung soll weitgehend die Scopeelemente der öffentlichen Konsistenzprüfung wieder verwenden, denn mit den öffentlichen Scopeelementen kann direkt auf das öffentliche RDF-Modell zugegriffen werden. Als erstes wird die private Konsistenzprüfung geprüft, ob sie bereits ein angelegtes Scopeelement besitzt. Wenn ja, dann wird es verwendet, ansonsten wird die öffentliche Konsistenzprüfung geprüft ob sie das gewünscht Scopeelement bereits besitzt. Erst wenn weder ein privates noch ein öffentliches Scopeelement ermittelt werden kann, wird es aus der lokalen Konsistenzprüfung angefordert und somit auch dort angelegt, falls es noch nicht existiert.
Für den Zugriff auf den eigentlichen Wert des Modellelements ruft die *RDFPropertyCallExpression* die Methode *getValue()* des ermittelten *RDFScopeElement* auf. Diese Methode versucht im eigenen RDF-Modell den Wert zu ermitteln. Bei der öffentlichen Konsistenzprüfung ist dies das öffentliche Modell, bei der privaten Konsistenzprüfung das private Modell. Ein Scopeelement in der privaten Konsistenzprüfung kann zusätzlich auch im öffentlichen Modell nach dem gewünschten Wert suchen, wenn er im lokalen Modell nicht gefunden werden kann. Sollte für es für die gesuchte Kombination von Modellelement und Feature keinen Wert geben, kann er auch *null* sein. Wenn er gefunden wird, ist der ermittelte Wert eine Ressource oder eine Liste mehrere Ressourcen. Das Ergebnis der *getValue()*-Methode ist auch das Ergebnis der Auswertung der *RDFPropertyCallExpression*.

6.3.2 Regelinstanzen

Der RDF Model Analyzer implementiert für die Verwaltung von Konsistenzregeln und deren Instanzen das Konzept aus Punkt 5.3. Das *AbstractDataModel* des Model Analyzer Frameworks definiert für jeden Änderungstyp eine Methode zur Behandlung der Änderung. Diese Methoden werden vom RDF Model Analyzer überschrieben und für die speziellen Anforderungen der Mehrbenutzerumgebung angepasst. Die Änderungsliste der Modellverarbeitung wird Element für Element abgearbeitet und für jeden Änderungstyp gibt es eine eigene Methode für die Operationen an den Konsistenzregeln. Die Änderungen an den Konsistenzregeln und Instanzen werden in den Validierungsdaten gesammelt. Nachdem die Änderungsliste abgearbeitet ist, werden die gesammelten Instanzen und Ausdrücke in den Validierungsdaten neu evaluiert.

NewChange

Die Verarbeitung von einem *newChange* ist sowohl bei der Initialisierung, als auch bei Änderungen gleich. Wie in Punkt 5.3.1 beschrieben, werden die neuen Instanzen bei der Initialisierung in der öffentlichen Konsistenzprüfung und bei Änderungen in der jeweiligen privaten Konsistenzprüfung angelegt. Zu Beginn der *validateNew()*-Methode werden anhand des Kontexts der Änderung alle passenden Konsistenzregeln ermittelt. Von diesen Konsistenzregeln wird jeweils eine Regelinstanz angelegt und nachdem alle Änderungen abgearbeitet sind, werden die neuen Regelinstanz evaluiert.

ModifyChange

Ein *ModifyChange* wird üblicherweise nur in der privaten Konsistenzprüfung eines Benutzers durchgeführt, da das öffentliche Modell von den Benutzern nicht geändert wird. Der RDF Model Analyzer überschreibt die Methode *validateModify()* des Model Analyzer Framework. Bei der Implementierung werden zusätzlich zur beschriebenen Vorgehensweise bei Instanzen (Punkt 5.3.2) auch die Ausdrücke einer Instanz betrachtet. Als erstes wird das durch die Änderung betroffene Scopeelement in der privaten Konsistenzprüfung ermittelt. Zusätzlich wird auch überprüft, ob die Änderung ein öffentliches Scopeelement betrifft, wenn ja wird auch dieses Scopeelement verwendet.Danach werden für die ermittelten Scopeelemente die betroffenen Ausdrücke der privaten Konsistenzprüfung gesucht, denn diese Ausdrücke müssen neu evaluiert werden. Ebenso werden alle Ausdrücke des öffentlichen Modells ermittelt, die durch die Änderung betroffen wären. Durch die Suche der öffentlichen Ausdrücke kann festgestellt werden, welche öffentliche Regelinstanzen durch die Änderung betroffen wären. Falls diese öffentliche Regelinstanzen nicht in der Liste der als gelöscht markieren Regelinstanzen und privat noch nicht angelegt sind, werden sie in der privaten Konsistenzprüfung neu angelegt. Diese Regelinstanzen erhalten eine Referenz auf die öffentliche Regelinstanzen, damit bei einer weiteren Änderung nicht eine weitere Kopie angelegt wird. Werden keine betroffenen Ausrücke gefunden, wird die Suche nach Änderungen noch für komplette Regelinstanzen fortgesetzt. Der Ablauf ist der Selbe, wie bei den Ausdrücken. Nachdem alle Änderungen abgearbeitet sind, werden die gefunden Ausdrücke und die neu angelegten Regelinstanzen evaluiert.

DeleteChange

Bei einer *DeleteChange* wird die *validateDelete()*-Methode aufgerufen und es muss wie in Punkt 5.3.3 beschrieben, auf den Bereich geachtet werden, aus dem die Instanz gelöscht wird. Wie bei einer Änderung, darf nicht direkt aus der öffentlichen Konsistenzprüfung gelöscht werden, sondern nur aus der privaten Konsistenzprüfung. Im ersten Schritt werden alle Instanzen aus dem öffenltichen Bereich gesucht, welche durch das Löschen des Kontextelements betroffen sind. Diese Instanzen müssen für den Benutzer als gelöscht markiert werden. Die als gelöscht markierten Instanzen aus der öffentlichen Konsistenzprüfung verwaltet jede private Konsistenzprüfung in der Liste *deletedDesignRuleInstances*. Nach der Behandlung der öffentlichen Konsistenzprüfung kann in der privaten Konsistenzprüfung weiter gesucht werden, dort können die Regelinstanzen und alle Referenzen darauf gelöscht werden.

6.3.3 Abrufen von Regeln

Damit der Benutzer am Client auch Ergebnisse der Konsistenzprüfung sieht, müssen diese auch vom Server abgerufen werden können. Es können die Regelinstanzen des öffentlichen Bereichs, eines privaten Bereichs oder die Regelinstanzen aus der Sicht eines Benutzers abgerufen werden. Das Abrufen der Regelinstanzen eines Bereiches erfordert in der Mehrbenutzerumgebung keine Änderung des Model Analyzer Framework, es kann die bereits vorhandene Methode *getDesignRuleInstances()* des *AbstractDataModel* verwendet werden. Werden jedoch die Regelinstanzen aus der Sicht eines Benutzers aufgerufen, müssen beide Bereiche kombiniert werden. Im *RDFDataModel* wird eine zusätzliche Methode *getAllDesignRuleInstances()* eingeführt, welche zur Erstellung einer Gesamtliste folgende Schritte ausführt:

- Die Gesamtliste wird mit allen Regelinstanzen aus dem öffentlichen Bereich gefüllt.
- Danach kommen alle Regelinstanzen aus dem privaten Bereich in die Gesamtliste. Überdeckt eine private Regelinstanz eine öffentliche Regelinstanz, wird die entsprechende öffentliche Regelinstanz aus der Gesamtliste entfernt, so dass nur mehr die private Regelinstanz in der Gesamtliste ist.
- Zum Schluss werden alle zur Regelinstanz aus der Gesamtliste entfernt, welche im privaten Bereich als gelöschte öffentliche Regelinstanz markiert sind.

Damit die Clients die Konsistenzregeln und deren Regelinstanzen abrufen können, stellt der Server das *RulesServlet* für Konsistenzregeln und das *InstancesServlet* für Regelinstanzen zu Verfügung (Tabelle 6.2). Beide Servlets liefern immer Deltas, das heißt es werden nur jene Regeln und vor allem Regelinstanzen übertragen, welche sich auch wirklich geändert haben.

http://Server/RulesServlet/<Projektname>/	
GET	Der Aufruf des Servlets unter Angabe eines Projektnamen liefert eine Liste aller Konsistenzregeln. Da alle Konsistenzprüfungen die gleichen Regeln verwenden, werden die Konsistenzregeln der öffentlichen Konsistenzprüfung gesendet.
http://Server/InstancesServlet/<Projektname>/	
GET	Bei Aufruf des Servlets liefert der Server alle Regelinstanzen des in der URL angegebenen Projekts. Bei einem erneuten Aufruf werden nur die Änderungen seit dem letzten Aufruf gesendet. Der private Bereich, aus dem die Instanzen abgerufen werden, wird durch die Authentifizierung bestimmt. Jeder Benutzer kann nur seinen eigenen Bereich abrufen.

Tabelle 6.2: RulesServlet und InstancesServlet

6.4 Benutzerverwaltung

Alle Benutzerdaten werden am Server in einem RDF-Modell gespeichert, welches beim Start des Servers auch wieder eingelesen wird. Beim ersten Start legt der Service zur Benutzerverwaltung im Workspace des Servers einen Ordner *userdata* an, in dem das RDF-Modell *data.rdf* mit den Benutzerdaten gespeichert wird. Für die Benutzerdaten wird ein eigener RDF-Namespace *USER* definiert. Gespeichert werden der Vorname *user:firstname*, der Nachname *user:lastname*, der Benutzername *user:username*, das Passwort *user:password* und die Rolle *user:role* des Benutzers. In der derzeitigen Form unterstützt der Server zwei Rollen. Eine privilegierte Rolle für Administratoren und eine Rolle für normale Benutzer. Das in Tabelle 6.3 beschriebene *UserServlet* ruft die entsprechenden Funktionen des Service zur Benutzerverwaltung auf.

http://Server/UserServlet/	
GET	Der direkte Aufruf des Servlets liefert eine Liste aller am Server registrierten Benutzer.
POST	Anlegen eines neuen Benutzers. Die Benutzerdaten werden aus dem Inhalt des Requests ausgelesen. Nur Administratoren dürfen neue Benutzer anlegen. War das Anlegen erfolgreich, wird eine Benutzerliste zurückgegeben.
PUT	Ändern von Benutzerdaten. Die zu ändernden Benutzerdaten werden aus dem Inhalt des Requests ausgelesen. Ein Administrator darf alle Benutzerdaten ändern und jeder Benutzer kann seine eigenen Daten ändern. War die Operation erfolgreich, liefert das Servlet eine Benutzerliste.
http://Server/UserServlet/<Benutzername>/	
GET	Der Aufruf des Servlets unter Angabe eines Benutzernamens liefert die Benutzerdaten des angegeben Benutzers. Auf diesem Weg können nur die eigenen Benutzerdaten abgerufen werden, wird ein anderer Benutzername wie bei der Authentifizierung angegeben, liefert der Server eine Gesamtliste der Benutzer.
DELETE	Löschen des in der URL angegebenen Benutzers. Administratoren dürfen alle Benutzer löschen, alle anderen Benutzer dürfen nur sich selbst löschen. Als Antwort wird eine Benutzerliste geliefert.

Tabelle 6.3: UserServlet

Auf dem Client wird der RSM mit einem einer eigenen Seite in den Einstellungen für die Benutzerverwaltung erweitert. Alle Benutzers des Servers werden in einer Tabelle dargestellt. Administratoren können die Benutzerdaten vollständig bearbeiten, alle anderen Benutzer können die eigenen Benutzerdaten verwalten. Über die Hauptseite der Einstellungen muss Benutzername und Passwort angegeben werden, damit sich der Benutzer am Server authentifizieren kann.

6.5 Projektverwaltung

Die Projekte werden als Ordner- und Dateistruktur im Workspace des Servers verwaltet. Für jedes Projekt wird ein neuer Ordner angelegt. Innerhalb des Projektordners wird ein weiterer Ordner *data* für die Projektdaten angelegt. Direkt in den Projektordner werden serialisierte Versionen des öffentlichen RDF-Modells und der privaten RDF-Änderungsmodelle abgelegt. Diese RDF-Modelle wurden zur Analyse am Server während der Implementierung benötigt und werden nicht mehr weiter verwendet. Zur Verwaltung der Projekte stellt der Server das *ProjectServlet* aus Tabelle 6.4 zu Verfügung. Für die Projektdateien wird ein eigenes Servlet verwendet. Die Projektliste und die Dateiliste für ein Projekt teilen sich den RDF-Namespace *PROJECT*. Für die Projektliste wird der Projektname *project:projectname* verwendet, die Dateiliste beinhaltet die einzelnen Dateinamen *project:filename*. Beide Listen werden aus der Ordner- und Dateistruktur des Servers dynamisch bei einer Abfrage generiert.

http://Server/ProjectServlet/	
GET	Der direkte Aufruf des Servlets liefert eine Liste der verfügbaren Projekte.

http://Server/ProjetServlet/<Projektname>/	
GET	Der Aufruf des Servlets unter Angabe eines Projektnamens liefert eine Dateiliste des angegebenen Projekts.
POST	Anlegen des in der URL angegebenen Projekts. Nur Administratoren dürfen neue Projekte anlegen. Als Antwort wird eine Projektliste geliefert.
DELETE	Löschen des in der URL angegebenen Projekts. Diese Operation ist nur für Administratoren erlaubt und liefert als Antwort eine Projektliste.

Tabelle 6.4: ProjectServlet

Am Client gibt es eine eigenen Seite in den Einstellungen zur Verwaltung von Projekten. Jeder Benutzer kann die Projektliste vom Server sehen und auch ein Projekt auswählen. Administratoren können die einzelnen Projekte auch verwalten.
Das *ProjectDataServlet* bietet die Dateioperationen für ein Projekt an. Die an die Requests angehängten Streams werden vom Service für die Projektverwaltung in Dateien geschrieben. Wie in Tabelle 6.5 beschrieben, sind die Dateinamen in der URL enthalten und die Inhalte der Dateien an den Request angehängt.

http://Server/ProjectDataServlet/<Projektname>/<Datei.Endung>	
GET	Download der in der URL angegebenen Datei. Der Inhalt der Datei wird in den Response geschrieben und ist somit auch die Antwort.
POST	Upload der in der URL angegebenen Datei. Der Inhalt der Datei wird aus dem Request gelesen. Als Antwort wird eine Dateiliste des Projekts geliefert.
DELETE	Löschen der in der URL angegebenen Datei. Als Antwort wird eine Dateiliste des Projekts geliefert.

Tabelle 6.5: ProjectDataServlet

Projektdateien werden am Client mittels Wizards verwaltet. Für das Exportieren können Dateien aus einem lokalen Projekt ausgewählt werden und zum Server in das entsprechende Projekt hochgeladen werden. Der Wizard für das Importieren funktioniert in die entgegengesetzte Richtung, es werden Dateien vom Server ausgewählt und in ein lokales Projekt gespeichert.

Kapitel 7

Benutzerschnittstelle

In vorangegangenen Kapiteln wurde die Idee und Implementierung der Arbeit genauer beschrieben, dieses Kapitel beschäftigt sich nun mit einer anderen Sicht auf diese Arbeit. Es wird die Benutzerschnittstelle beschrieben, also die Sicht der einzelnen Benutzer auf das Programm. Das Kapitel wird anhand eines konkreten Beispiels zeigen, welche Möglichkeiten die beiden Benutzer Alice und Bob haben, den Client-Server Model Analyzer zu bedienen. Für alle Benutzer steht ein Server mit den in Punkt 3.1.3 definierten Regeln zu Verfügung. Alice startet das Projekt und verwendet einen RSM mit installiertem Client. Das von Alice verwendete Modell wurde bereits in Punkt 3.1.2 beschrieben.
Als erstes kontrolliert Alice die Konfiguration des Systems. Der Client ermöglicht es, die Projekt- und Benutzerdaten am Server zu warten. Da jede Modelländerung personalisiert zum Server gesendet wird, muss zu Beginn der Benutzername und das Passwort angegeben werden. Dies passiert auf der Startseite der Konfiguration des Clients (Abbildung 7.1). Zusätzlich kann Alice noch das Projekt angeben, zu dem die Änderungen gesendet werden. Am Server existiert immer ein Standardprojekt mit dem Namen *default*.

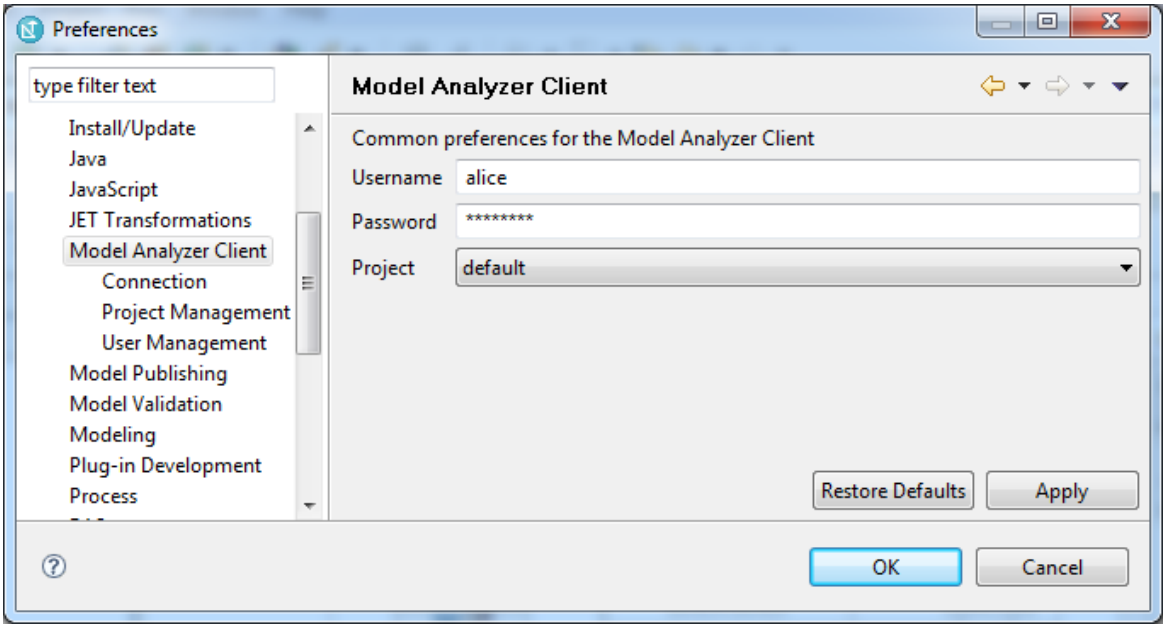


Abbildung 7.1: Allgemeine Einstellungen

Im ersten Unterpunkt der Einstellungen sind die Verbindungseinstellungen zu finden. Es können die URLs der einzelnen Servlets angepasst werden. Für jedes, der in Kapitel 6

vorkommenden Servlets, gibt es eine eigene URL. In diesem Beispiel (Abbildung 7.2) wird von Alice ein lokal installierter Server auf dem Port 9090 verwendet. Bob muss sich somit zum lokalen Rechner von Alice verbinden, damit sie gemeinsam Arbeiten können.

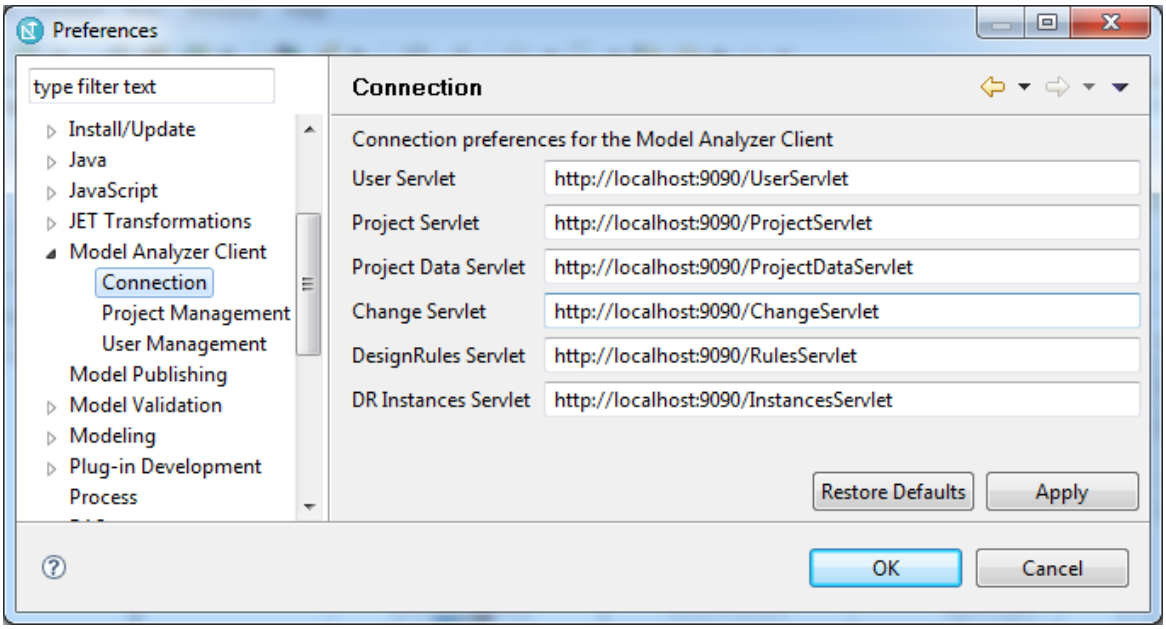


Abbildung 7.2: Verbindungseinstellungen

Die nächsten Einstellungen betreffen die Projektverwaltung. Abbildung 7.3 zeigt das Anlegen eines neuen Projekts. Für ein neues Projekt ist nur ein Name anzugeben. Alice legt ein Projekt mit dem Namen *LightSwitch* an. Projekte können auch wieder gelöscht werden, nur das Standardprojekt kann nicht gelöscht werden. Wird ein neues Projekt angelegt, so ist dies auch auf der Startseite der Konfiguration auswählbar.

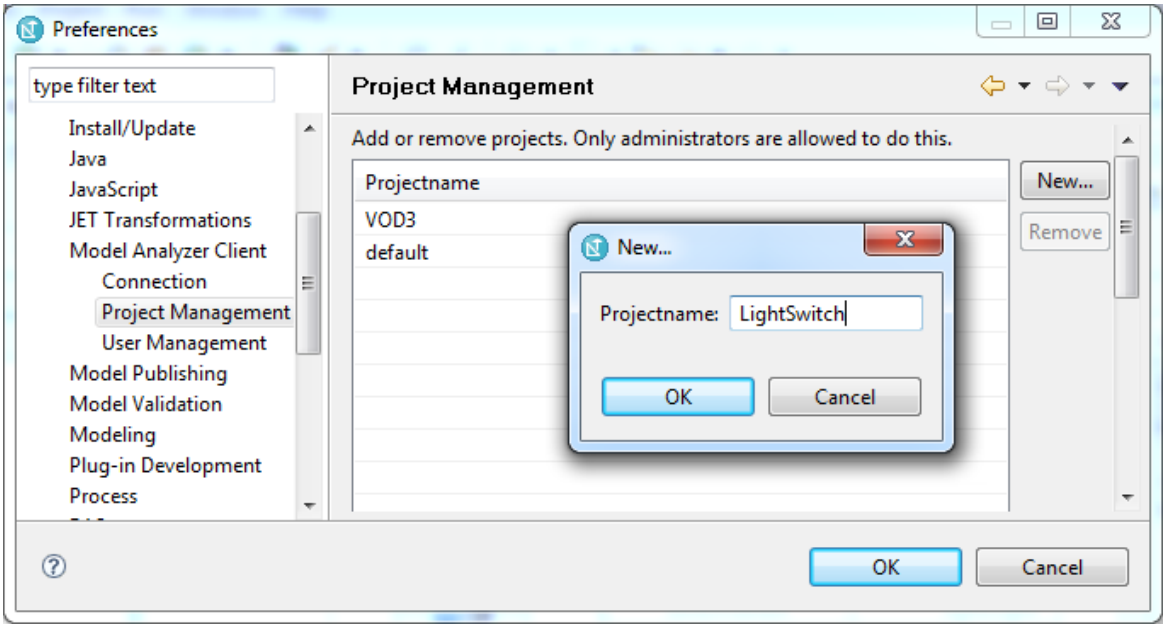


Abbildung 7.3: Projektverwaltung

Als letzten Punkt der Konfiguration gibt es noch eine Benutzerverwaltung, damit kann Alice die Benutzerdatei am Server verwalten. Zur Verwaltung gibt es ein einfaches Rollen-

konzept. Alice besitzt die Rolle *ADMIN*, diese Rolle erlaubt es ihr, Benutzer anzulegen, zu bearbeiten und natürlich auch zu löschen. Bob besitzt hingegen nur die Rolle *USER* und er kann nur seine eigenen Benutzerdaten verwalten. Administratoren können auch Projekte verwalten, während normale Benutzer nur eine Liste der Projekte sehen. Abbildung 7.4 zeigt Alice beim Anlegen eines neuen Benutzers. Sie muss Vorname, Nachname, Benutzername und ein Passwort angegeben, zusätzlich muss von ihr noch eine Rolle ausgewählt werden.

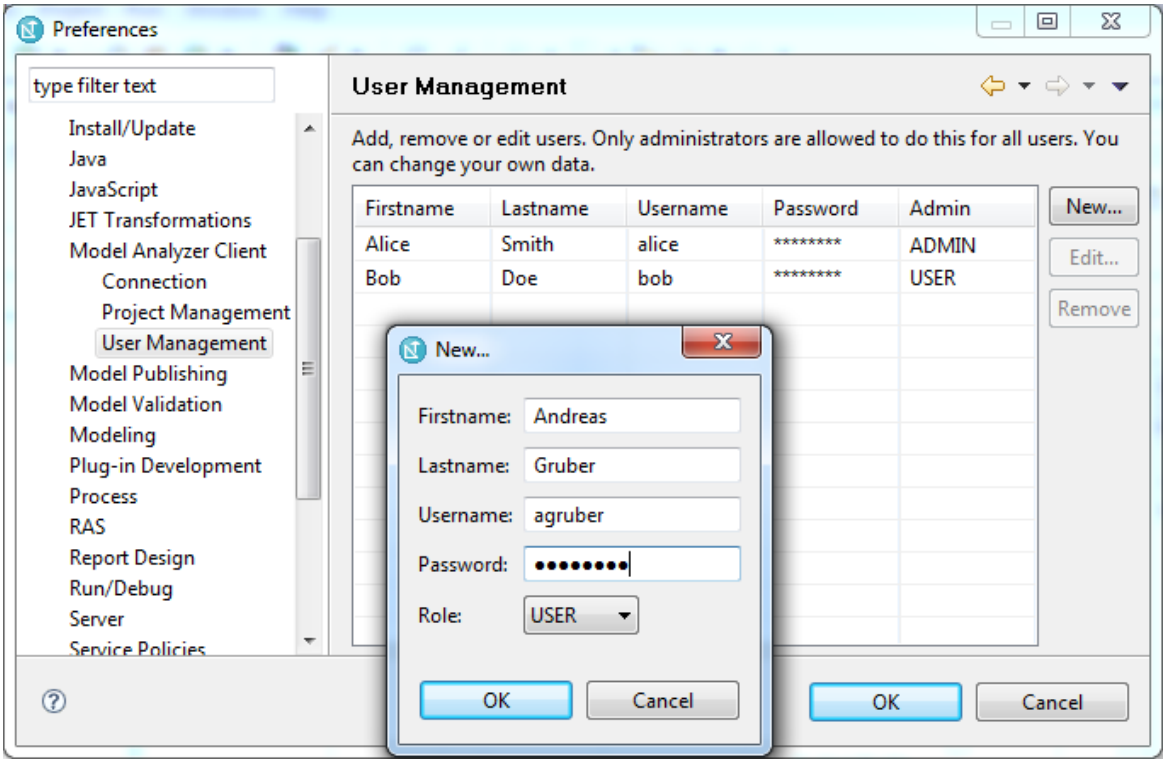


Abbildung 7.4: Benutzerverwaltung

Neben den Einstellungen verfügt der Client über ein Menü zum Steuern der wichtigsten Funktionen. Das Menü (Abbildung 7.5) bietet die folgenden drei Einträge:

- Die *Publish Private To Public* Funktion erlaubt es, seinen eigenen privaten Bereich in den öffentlichen Bereich zu übertragen. Es werden sowohl Modelldaten, als auch Konsistenzdaten in den öffentlichen Bereich geschrieben. Diese experimentelle Funktion ist nur verfügbar, wenn Alice oder Bob alleine an einem Projekt arbeiten würden.

- Mit dem *Start ModelAnalyzer* Menüeintrag wird die Kommunikation zwischen Client und Server gestartet. Der Client sendet Modelldaten zum Server und empfängt die Konsistenzdaten. Bei einem erneuten Klick auf die Funktion kann die Kommunikation mit dem Server wieder gestoppt werden. Der Menüeintrag ändert, je nach Funktion welche durchgeführt werden kann, seinen Namen.

- Die Initialisierung des Projekts am Server wird mit *Init Project on Server* durchgeführt und braucht nur von einem der Benutzer ausgeführt werden. Die Funktion bewirkt, dass das geöffnete Modell in RDF konvertiert, zum Server gesendet und eine Konsistenzprüfung durchgeführt wird.

(a) Menü mit Start (b) Menü mit Stopp

Abbildung 7.5: Menü

Um Informationen über die Konsistenzregeln zu erhalten, benötigt Alice die verfügbaren Views aus Abbildung 7.6. Die View *Design Rule Instances* listet alle Regelinstanzen mit deren Details auf. Die View *Design Rules* listet die zugehörigen Konsistenzregeln auf. Beide Views sind auch in der Stand-Alone Variante des Model Analyzers vorhanden und sind vom Design unverändert.

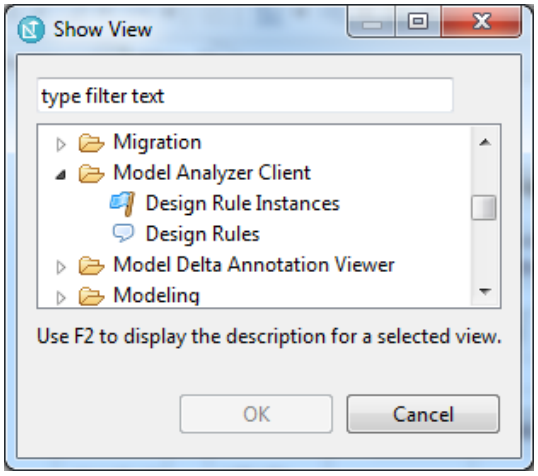


Abbildung 7.6: Views

Nach dem Editieren der diversen Einstellungen und dem Einrichten der Entwicklungsumgebung will Alice das geöffnete Modell Bob zu Verfügung stellen. Dieser Schritt passiert noch vor dem eigentlichen Bearbeiten des Modells, denn auch Bob soll mit dem ursprünglichen Modell arbeiten können. Sie verwendet den Export Wizard *Export to Server*. Dieser Export Wizard ermöglicht Alice das Speichern von Dateien am Server. Dazu wählt sie die einzelnen Dateien aus und entscheidet sich für das Projekt am Server, wo die Dateien gespeichert werden sollen. Mit der Bestätigung des Dialogs in Abbildung 7.7 werden die Dateien zum Server gesendet. Alle anderen Benutzer haben ab diesem Zeitpunkt die Möglichkeit, die Projektdatei *.project* und die Modelldatei *LightSwitch.emx* vom Server auf den Client zu laden.
Als letzten Schritt, bevor Alice mit dem Bearbeiten des Modells beginnt, muss sie das Modell am Server initialisieren. Um die Initialisierung zu starten, wird im Menü die Option *Init Project on Server* gewählt. Diese Funktion bewirkt, dass das von Alice geöffnete Modell serialisiert und zum Server übertragen wird. Das übertragene Modell wird zum öffentlichen Modell für alle anderen Benutzer, welche an diesem Projekt mitarbeiten. Am Server wird auf dem Modell eine Konsistenzprüfung durchgeführt und es werden die öffentlichen Konsistenzdaten erzeugt und evaluiert. Nach dieser Aktion ist am Client von Alice noch nichts zu sehen. Da Alice aber Zugang zum Server hat, kann sie dort die Konsole öffnen und sieht Informationen über die initiale Konsistenzprüfung. Ein Teil der

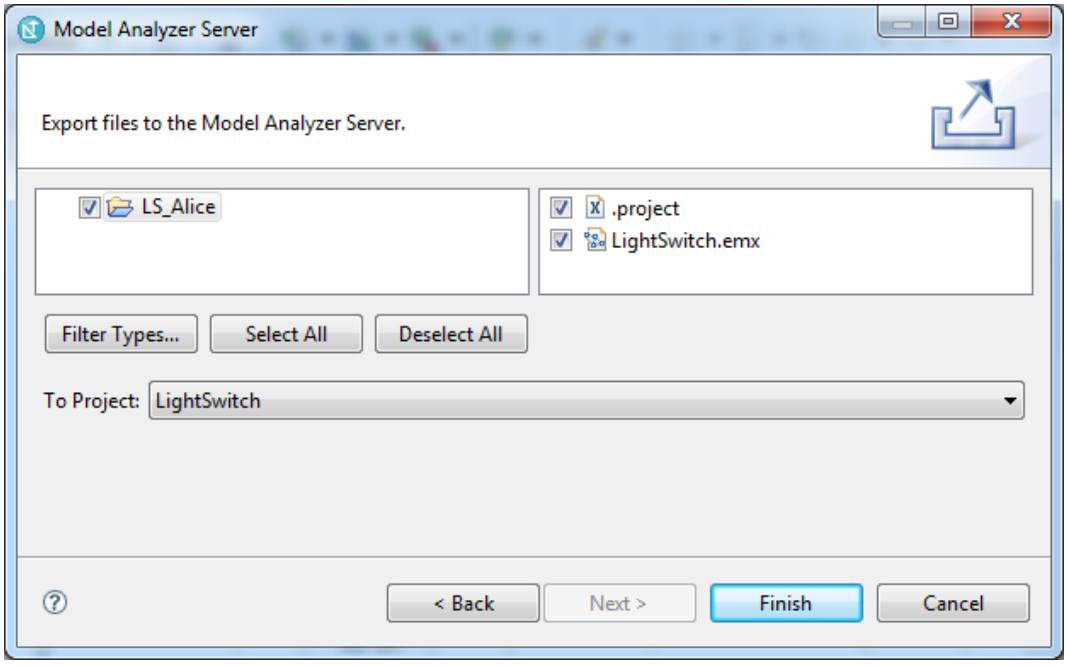


Abbildung 7.7: Export Wizard

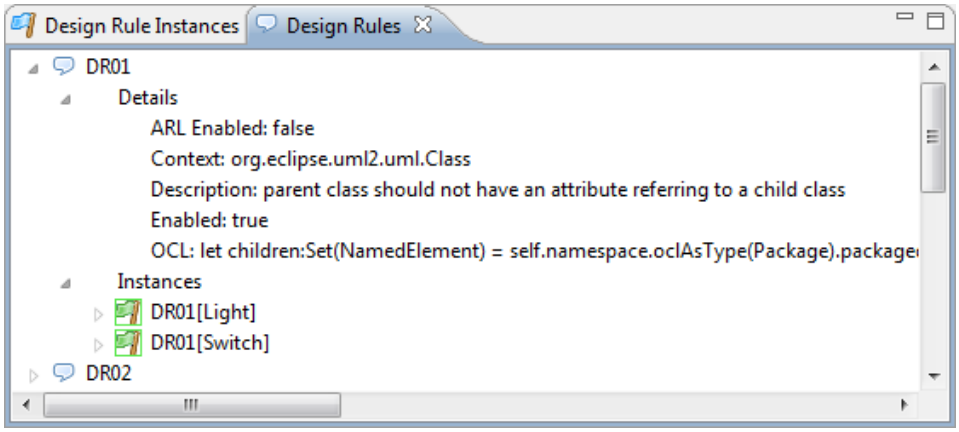
Ausgabe der Serverkonsole ist in Abbildung 7.8 zu sehen. Es wird jede der 30 ermittelten Regelinstanzen aufgelistet und natürlich auch das zugehörige Ergebnis der Prüfung. Jede Zeile der Konsole am Server beginnt mit dem Benutzer, welcher die Aktion ausgelöst hat, wenn die Aktion wie in Abbildung 7.8 das öffentliche Modell betrifft, so beginnt die Zeile mit *public*. Jeder privaten Konsistenzprüfung der einzelnen Benutzer stehen diese 30 Regelinstanzen zu Verfügung.

```
public: DR14[deactivate,_imy-8EiPEd-Xye0lIHC77w] -> CONSISTENT
public: DR18[activate,_JOdy0EiREd-Xye0lIHC77w] -> INCONSISTENT
public: DR02[Switch,_WodksEiPEd-Xye0lIHC77w] -> CONSISTENT
public: DR03[activate,_dEyScEiQEd-Xye0lIHC77w] -> INCONSISTENT
public: DR10[switch,_RtV5EEiQEd-Xye0lIHC77w] -> CONSISTENT
public: DR13[Light,_X0nOwEiPEd-Xye0lIHC77w] -> CONSISTENT
public: DR13[Switch,_WodksEiPEd-Xye0lIHC77w] -> CONSISTENT
public: DR10[light,_TAC3EEiQEd-Xye0lIHC77w] -> CONSISTENT
public: 30 DesignRuleInstances
        30 public DesignRuleInstances
```

Abbildung 7.8: Serverkonsole nach Initialisierung

Damit Alice auch die evaluierten Regelinstanzen in ihrer Entwicklungsumgebung sieht, muss sie im Menü die Funktion *Start ModelAnalyzer* auswählen. Sobald die Kommunikation mit dem Server gestartet wird, werden nicht nur Informationen zu den Konsistenzregeln zum Client übertragen, sondern es werden auch Modelländerungen von Alice zum Server übertragen. Abbildung 7.9a zeigt die *Design Rule Instances* View mit allen evaluierten Regelinstanzen. Die Regelinstanz *DR01[Light]* wurde von Alice erweitert und zeigt somit auch weitere Details und Scopeelemente an. Nach dem Starten wird auch die *Design Rules* View in Abbildung 7.9b befüllt. Hier ist die Konsistenzregel *DR01* von Alice erweitert worden und es können Details der Konsistenzregel angezeigt werden, zusätzlich werden auch alle Regelinstanzen angezeigt.

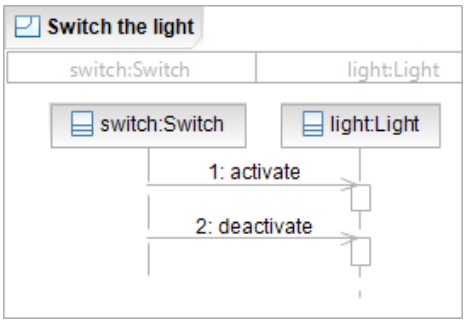
(a) View mit Instanzen



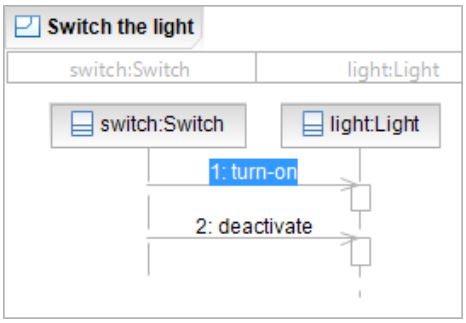
(b) View mit Regeln

Abbildung 7.9: Views nach Initialisierung

Jetzt startet Alice mit dem eigentlichen Bearbeiten des Modells. Sie ändert im Sequenzdiagramm *Switch the light* die Nachricht *activate* auf *turn-on* um. Der ursprüngliche Teil des Modells ist in Abbildung 7.10a zu sehen, die Änderung von Alice ist in Abbildung 7.10b blau markiert.



(a) Modell

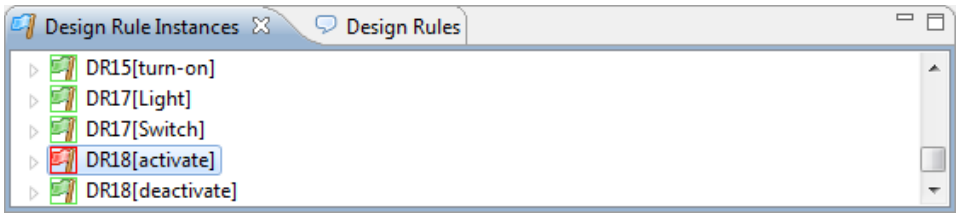


(b) Modell

Abbildung 7.10: Modell

Die Änderung von Alice kann nun Schritt für Schritt nachverfolgt werden. Abbildung 7.11a zeigt die Liste der Regelinstanzen vor der Änderung an. Die Regelinstanz *DR18[activate]* tritt nun in den Vordergrund, da diese durch die Änderung betroffen sein wird. Sobald die Änderung durchgeführt wurde, wird die Serverkonsole in Abbildung 7.11b wichtig. Der erste Teil der Ausgabe zeigt, dass im privaten Bereich von Alice eine neue Regelinstanz angelegt wurde, diese ist jedoch noch nicht evaluiert. Konkret wird eine Kopie der Regelinstanz *DR18[activate]* aus dem öffentlichen Bereich erstellt. Sobald alle Änderungen

vom Server bearbeitet wurden, werden die geänderten Regelinstanzen evaluiert, dies ist im zweiten Teil der Ausgabe zu sehen. Weiters ist nun zu sehen, dass Alice nun eine private Regelinstanz besitzt und noch 29 Regelinstanzen aus dem öffentlichen Bereich verwendet. Schlussendlich wird in der *Design Rule Instances* View die neu evaluierte Regelinstanz angezeigt (Abbildung 7.11c). Zu beachten ist, dass sich der Name der betroffenen Regelinstanz in *DR18[turn-on]* am Client geändert hat, damit sie einfacher im Modell gefunden werden kann.



(a) Instanzen vor der Änderung

```
alice: DR18[activate,_JOdy0EiREd-Xye0lIHC77w] -> NOT_VALIDATED
alice: 30 DesignRuleInstances
        0 private DesignRuleInstances
        30 public DesignRuleInstances

alice: DR18[activate,_JOdy0EiREd-Xye0lIHC77w] -> CONSISTENT
alice: 30 DesignRuleInstances
        1 private DesignRuleInstances
        29 public DesignRuleInstances
```

(b) Serverkonsole

(c) Instanzen nach der Änderung

Abbildung 7.11: Änderung von Alice

Da Alice mit einem lokalen Server arbeitet, kann sie sich auch den Workspace des Servers genauer betrachten. Die Ordnerstruktur des Servers ist einfach aufgebaut und in Abbildung 7.12 dargestellt.

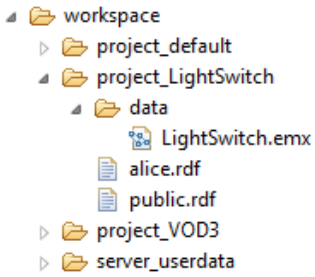


Abbildung 7.12: Ordnerstruktur Server

Für jedes Projekt wird ein Ordner mit dem Präfix *project_* angelegt. Innerhalb dieses Ordners werden die einzelnen Modelle abgelegt. Zu sehen ist das öffentliche Modell *public.rdf* und das Modell von Alice *alice.rdf*. Da Bob noch nicht am Projekt teilnimmt, ist

auch noch kein Modell angelegt. Zusätzlich gibt es im Projektordner immer einen Ordner mit dem Namen *data*. In diesen Ordner werden jene Dateien abgelegt, welche mit dem Export Wizard im entsprechenden Projekt am Server gespeichert werden.
Nun steigt Bob in das Projekt ein und verwendet, genau so wie Alice, einen RSM mit installiertem Client. Er beginnt mit einer leeren Entwicklungsumgebung, das heißt er muss als ersten Schritt das Modell vom Server laden. Dazu wird ein leeres Projekt angelegt und der Import Wizard *Import from Server* aufgerufen. Die einzige Seite des Wizards ist in Abbildung 7.13 zu sehen. Bob muss ein Projekt auswählen, von diesem Projekt kann er wiederum entscheiden, welche Dateien geladen werden sollen. Zum Schluss muss noch der Zielordner, in diesem Fall der Projektordner *LS_Bob*, angegeben werden. Mit der Bestätigung des Dialogs werden die ausgewählten Dateien vom Server zum Client geladen. Nach erfolgtem Import kann das Modell geöffnet werden.

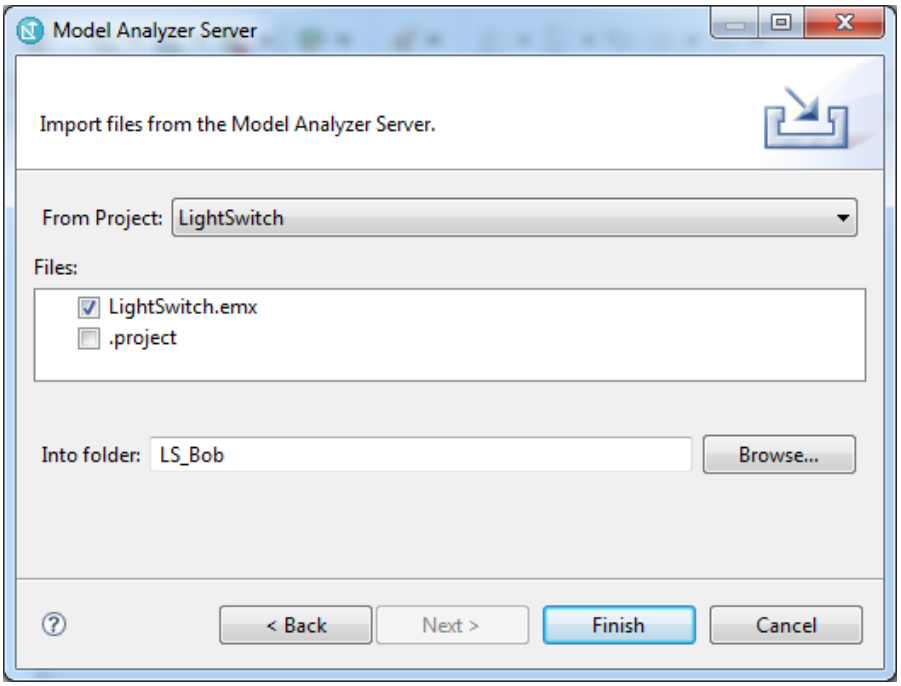


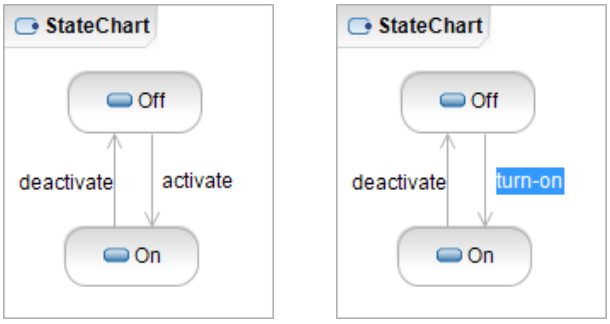
Abbildung 7.13: Import Wizard

Wenn das Modell geöffnet ist, muss Bob nur mehr den Client starten, damit Änderungen zum Server gesendet werden und Konsistenzdaten vom Server abgerufen werden können. Dazu muss er im Menü den Punkt *Start ModelAnalyzer* aufrufen. Eine Initialisierung des Projekts ist nicht mehr notwendig. Genau wie bei Alice werden nach dem Starten beide Views befüllt und Bob sieht alle 30 öffentlichen Regelinstanzen. Dass ein weiterer Benutzer zum System gekommen ist, ist auch auf der Serverkonsole ersichtlich (Abbildung 7.14).

```
bob joined the project: project_LightSwitch
```

Abbildung 7.14: Start des Model Analyzer Client

Jetzt startet auch Bob mit dem eigentlichen Bearbeiten des Modells. Er ändert im Zustandsdiagramm *StateChart* den Übergang *activate* in *turn-on* um. Der ursprüngliche Teil des Modells ist in Abbildung 7.15a zu sehen, die Änderung von Bob ist in Abbildung 7.15b blau markiert.



(a) Modell (b) Modell

Abbildung 7.15: Modell

Genau wie bei Alice (Abbildung 7.11) kann auch die Änderung von Bob Schritt für Schritt nachverfolgt werden. Durch die Änderung wird die Regelinstanz *DR03[activate]* in Abbildung 7.16a betroffen sein. Am Server (Abbildung 7.16b) wird eine Kopie der betroffenen Regelinstanz angelegt und Bob besitzt nun auch eine private Regelinstanz. Das Ergebnis der Änderung ist in Abbildung 7.16c zu sehen. Alice und Bob können ab diesem Zeitpunkt am gleichen Modell weiter arbeiten, alle zukünftigen Änderungen werden nach dem gleichen Schema ablaufen.

(a) Demo 16a

```
bob: DR03[activate,_dEyScEiQEd-Xye0lIHC77w] -> NOT_VALIDATED
bob: 30 DesignRuleInstances
        0 private DesignRuleInstances
        30 public DesignRuleInstances

bob: DR03[activate,_dEyScEiQEd-Xye0lIHC77w] -> CONSISTENT
bob: 30 DesignRuleInstances
        1 private DesignRuleInstances
        29 public DesignRuleInstances
```

(b) Serverkonsole

(c) Demo 16c

Abbildung 7.16: Änderung von Bob

Kapitel 8

Evaluierung

Zu Beginn dieser Masterarbeit sind Ziele definiert worden, welche die Konsistenzprüfung in einer Mehrbenutzerumgebung erreichen soll. Eines der wichtigsten Ziele ist die Vermeidung von Duplikaten bei Regelinstanzen und Modellelementen. In diesem Kaptitel wird die Client-Server Variante des Model Analyzers (MA) mit der lokalen Variante verglichen. Durch die systematisch Ermittlung von Kennzahlen wird die Basis für einen Vergleich geschaffen und mit Hilfe dieser Kennzahlen können Vor- und Nachteile der Konsistenzprüfung in einer Mehrbenutzerumgebung diskutiert werden.

8.1 Kennzahlen

Um die Kennzahlen systematisch definieren zu können, wird der Goal Question Metric (GQM) Ansatz von Basili et al. [BCR94] verwendet. Der GQM-Ansatz ist ein Modell zum Definieren, Messen und Interpretieren von Kennzahlen von Software. Ein GQM-Modell wird in drei Ebenen unterteilt [BCR94]:

- Ziel - Goal: Die oberste Ebene ist die konzeptuelle Ebene, sie definiert die Ziele, welche durch die Evaluierung erreicht werden sollen. Ein Ziel ist durch eine Aufgabe *Purpose*, einen zu untersuchender Sachverhalt *Issue*, ein zu untersuchendes Objekt *Object* und eine Sichtweise *Viewpoint* definiert. Typische Objekte zur Messung sind Produkte, Prozesse und Ressourcen.
- Frage - Question: Die operative Ebene legt Fragen zu den Zielen fest. Diese Fragen definieren, wie das Ziel erreicht und gemessen werden kann.
- Metrik - Metric: Die Metriken bilden die quantitative Ebene. Sie sind eine Menge von Kennzahlen, welche einer Frage zugeordnet sind und diese Frage mit einem konkreten Zahlen beantworten. Eine Metrik kann auch mehreren Fragen zugeordnet sein. Metriken können objektiv sein, also sie hängen nur von einem Objekt der Software ab, wie zum Beispiel von einer Größe einer Liste oder der Ausführungszeit einer Methode. Sie können aber auch subjektiv sein, das heißt sie hängen zusätzlich von der Sichtweise einer Person ab, zum Beispiel die Lesbarkeit von Texten oder die Bewertung der Bedienbarkeit der Software.

Konsistenzprüfung

Die ersten Kennzahlen (Tabelle 8.1) beschäftigen sich mit der Konsistenzprüfung. Ziel ist es, den Client-Server MA mit dem Stand-Alone MA aus der Sicht eines Benutzers zu

vergleichen. Die Kennzahlen für den Bereich der Regelinstanzen umfassen die Anzahl der Regelinstanzen und Evaluierungen pro Benutzer und auch die Dauer einer Evaluierung. Die Kennzahlen für den Bereich der Modelldaten sind die Anzahl der Scopeelemente und der Modellzugriffe während der Konsistenzprüfung, sowie die Menge der Modellelemente pro Benutzer.

Goal 1	Purpose Issue Object Viewpoint	Vergleich Daten der Konsistenzprüfung Konsistenzprüfung von Software Design Modellen Sicht eines Benutzer
Question	Q 1.1	Wie groß ist der Unterschied bei den Regelinstanzen beim Vergleich von Client-Server MA und einzelnen Stand-Alone MA?
Metric	M 1.1.1 M 1.1.2 M 1.1.3 M 1.1.4	Anzahl der Regelinstanzen Evaluierungen der Regelinstanzen Evaluierungen der Expressions Dauer einer Evaluierung in Millisekunden
Question	Q 1.2	Wie groß ist der Unterschied bei den Modelldaten beim Vergleich von Client-Server MA und einzelnen Stand-Alone MA?
Metric	M 1.2.1 M 1.2.2 M 1.2.3	Anzahl der Scopeelemente Anzahl der Modellzugriffe Menge der Modellelemente

Tabelle 8.1: GQM Konsistenzprüfung

Initialisierung

Das nächste Ziel beschäftigt sich mit der Initialisierung des Modells. Die Tabelle 8.2 zeigt alle Kennzahlen. Die erste Frage beschäftigt sich mit den Größen der Modelldaten in verschiedenen Speicherungsverfahren. Bei der zweiten Frage werden der Client-Server MA und der Stand-Alone MA bei der Initialisierung verglichen, wichtig für den Vergleich sind die Zeitunterschiede zwischen den beiden Verfahren.

Goal 2	Purpose Issue Object Viewpoint	Vergleich Initialisierung der Konsistenzprüfung Konsistenzprüfung von Software Design Modellen Sicht des Systems
Question	Q 2.1	Wie groß ist der Unterschied bei der Speicherung eines Modells als EMX- oder RDF-Datei?
Metric	M 2.1.1 M 2.1.2	Größe der Modelldatei in Kilobyte Größenunterschied der Modelldateien als Faktor
Question	Q 2.2	Wie groß ist der Unterschied bei Initialisierung des Modells im Client-Server MA im Vergleich zum Stand-Alone MA?
Metric	M 2.2.1 M 2.2.2	Zeit für die Konvertierung nach RDF in Millisekunden Zeit für die Evaluierung des Modells in Millisekunden

Tabelle 8.2: GQM Initialisierung

Kommunikation

Das letzte Ziel beschäftigt sich mit den Auswirkungen der Kommunikation. Das zugehörige GQM-Modell in Tabelle 8.3 beschreibt zwei Fragen. Eine Frage beschäftigt sich mit den übertragenen Datenmengen, also wie groß ist eine Änderungen und wie groß sind im Durchschnitt die verschiedenen Änderungstypen. Die andere Frage beschäftigt sich mit der Zeit, die vom Absenden einen Änderung bis zur Antwort des Servers verstreicht. Dazu werden Kennzahlen eines lokalen und eines entfernten Servers benötigt.

Goal 3	Purpose Issue Object Viewpoint	Charakterisierung Auswirkungen der Kommunikation Konsistenzprüfung von Software Design Modellen Sicht eines Benutzers
Question	Q 3.1	Welche Datenmengen werden im laufenden Betrieb des Client-Server MA übertragen?
Metric	M 3.1.1 M 3.1.2 M 3.1.3 M 3.1.4	übertragene Datenmenge beim Hinzufügen in Kilobyte übertragene Datenmenge beim Ändern in Kilobyte übertragene Datenmenge beim Löschen in Kilobyte durchschnittliche Größe einer Änderung in Kilobyte
Question	Q 3.2	Welche Zeitverzögerungen sind durch die notwendige Kommunikation zwischen Client und Server vorhanden?
Metric	M 3.2.1 M 3.2.2	Gesamtzeit einer Änderung bei einem lokalen Server in Millisekunden Gesamtzeit einer Änderung bei einem Server über das Internet in Millisekunden

Tabelle 8.3: GQM Kommunikation

8.2 Datenerzeugung

Damit die Kennzahlen ermittelt und diskutiert werden können, müssen zuerst Rohdaten vorhanden sein, beziehungsweise erzeugt werden. Die Datenbasis für die Kennzahlen der Initialisierung sind fünf Modellen verschiedenster Größe. Tabelle 8.4 zeigt eine Übersicht der Modelle und ihrer wichtigsten Daten.

Modell	Modellelemente	EMX-Modelldatei
caBIO	1656	1290 kB
Calendarium	2820	2510 kB
LightSwitch	49	38 kB
MicrowaveOven	628	268 kB
VOD	113	67 kB

Tabelle 8.4: Modelle für Initialisierung

Die Datenbasis für alle anderen Kennzahlen mit einem konkreten Beispiel erzeugt, es wird ein System mit vier Benutzern analysiert. Diese Benutzer bearbeiten das Beispielmodell aus Punkt 3.1.2, die Konsistenzregeln sind für alle Benutzer gleich. Die vollständige Liste

der verwendeten Konsistenzregeln ist im Anhang A angegeben. In der Stand-Alone Variante des MA lädt jeder Benutzer das Modell, startet die Konsistenzprüfung und führt seine Aktionen aus. Bei der Client-Server Variante initialisiert ein Benutzer das Modell am Server, jeder Benutzer kann dann starten und die Aktionen ausführen. Die einzelnen Aktionen und das daraus resultierende Modell sind nachfolgend angegeben.

Benutzer 1

- Nachricht *activate* im Sequezdiagramm *Switch the light* auf *turn-on* ändern
- Aktion *activate* im Zustandsdiagramm *StateChart* auf *turn-on* ändern
- Interface mit dem Namen *Room* erstellen
- private Methode mit dem Namen *enter* im Interface *Room* erstellen
- 2 Attribute mit dem Namen *size* in der Klasse *Light* erstellen
- Nachricht *turn-on* im Sequezdiagramm *Switch the light* auf *light-on* ändern
- Klasse mit dem Namen *Switch* anlegen

Das Ergebnis ist in Abbildung 8.1 zu sehen (40 Regelinstanzen).

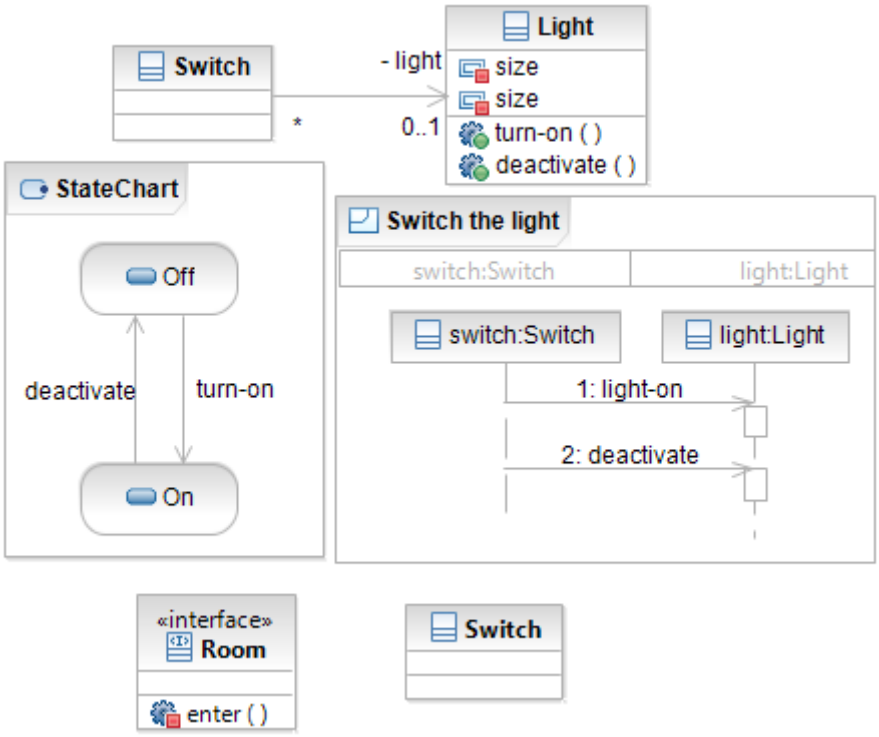


Abbildung 8.1: Ergebnismodell von Benutzer 1

Benutzer 2

- Methode *deactivate* in der Klasse *Light* auf *turn-off* umbenennen
- Aktion *deactivate* im Zustandsdiagramm *StateChart* auf *turn-off* ändern
- Aktion *activate* im Zustandsdiagramm *StateChart* auf *turn-on* ändern
- Löschen der Klasse *Switch*
- Methode *dimm-up* in der Klasse *Light* erstellen
- Übergang *dimm-up* im Zustandsdiagramm *StateChart* von *Off* nach *On* erstellen
- Übergang *dimm-down* im Zustandsdiagramm *StateChart* von *On* nach *Off* erstellen

Das Ergebnis ist in Abbildung 8.2 zu sehen (25 Regelinstanzen).

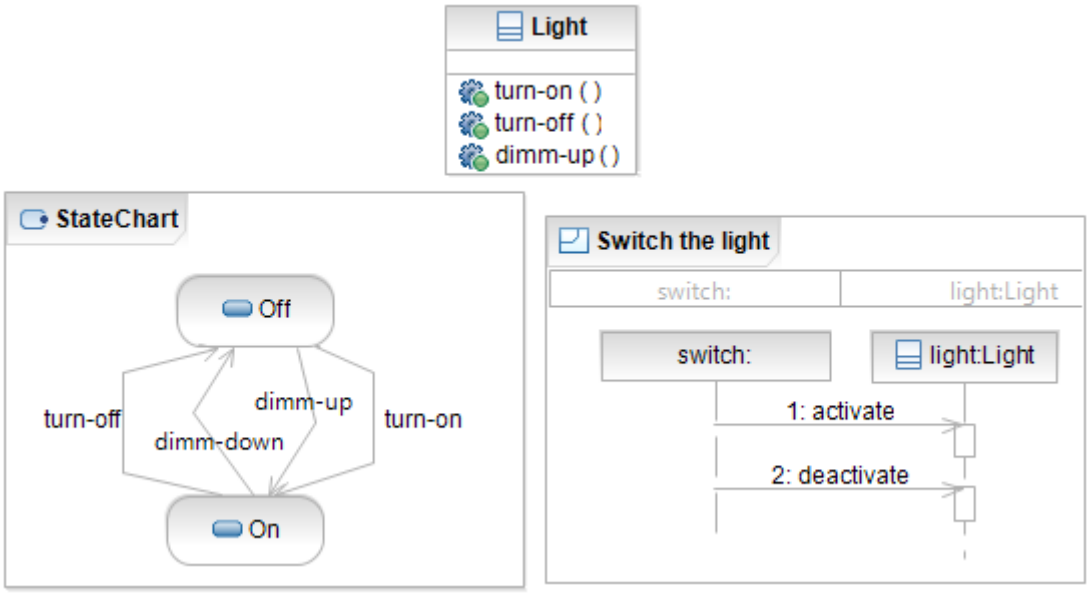


Abbildung 8.2: Ergebnismodell von Benutzer 2

Benutzer 3

- Klasse mit dem Namen *Pulp* anlegen
- Klasse mit dem Namen *EnergySavingPulp* anlegen
- Generalisierung von *EnergySavingPulp* nach *Pulp* und wieder zurück erstellen
- falsche Generalisierung von *Pulp* nach *EnergySavingPulp* löschen
- Attribut mit dem Namen *pulp* und dem Typ *EnergySavingPulp* in der Klasse *Pulp* erstellen
- Methode mit dem Namen *shine* in der Klasse *Pulp* anlegen
- Parameter mit dem Typ *EnergySavingPulp* zur Methode *shine* hinzufügen

Das Ergebnis ist in Abbildung 8.3 zu sehen (46 Regelinstanzen).

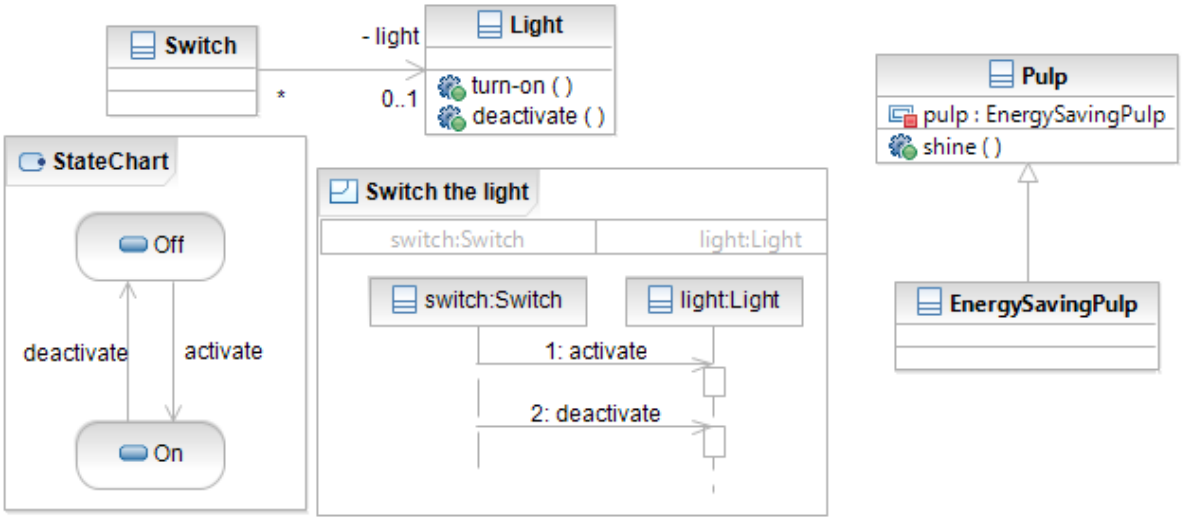


Abbildung 8.3: Ergebnismodell von Benutzer 3

Benutzer 4

- Zustandsdiagramm *StateChart* löschen
- Richtung der Assoziation zwischen den Klassen *Switch* und *Light* ändern
- am Ende der Assoziation die Variable *switch* in *light* umbenennen
- zur Methode *deactivate* der Klasse *Light* zwei Parameter mit dem gleichen Namen hinzufügen
- Methode *turn-on* der Klasse *Light* in *deactivate* umbenennen
- Parameter der Methode *deactivate* löschen

Das Ergebnis ist in Abbildung 8.4 zu sehen (28 Regelinstanzen).

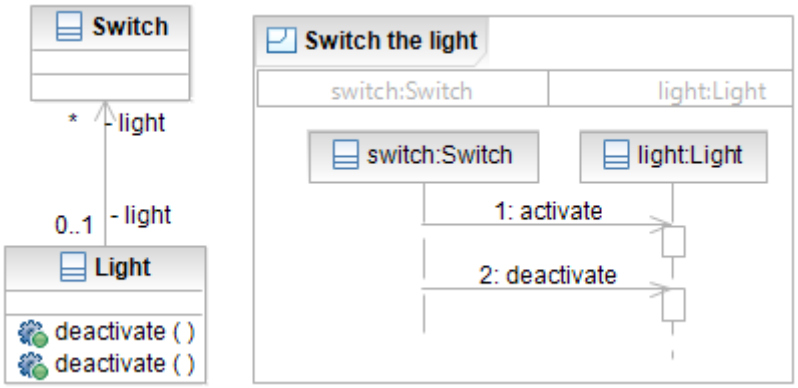


Abbildung 8.4: Ergebnismodell von Benutzer 4

8.3 Auswertung

In diesem Abschnitt wird die Auswertung der Kennzahlen und ihre Bedeutung diskutiert. Die Auswertung umfasst die drei definierten Ziele, wobei Ziel 1 in die Punkte Konsistenzdaten und Modelldaten aufteilt wurde, Ziel 2 wird im Punkt Initialisierung und Ziel 3 im Punkt Kommunikation behandelt.

8.3.1 Konsistenzdaten

Als Erstes wird Frage *Q 1.1* behandelt, also wie groß der Unterschied bei den Regelinstanzen zwischen Client-Server und Stand-Alone MA ist. Für diese Frage werden Kennzahlen der beiden Verfahren benötigt, diese wurden mittels Simulation eines konkreten Beispiels ermittelt. In Tabelle 8.5 sind die ermittelten Zahlen für den Stand-Alone MA aufgelistet. Für jeden Benutzer ist die Anzahl der Evaluierungen, also wie oft die Konsistenzprüfung aufgerufen wurde, angegeben. Weiters ist die durchschnittliche Anzahl der Regelinstanzen angegeben. Dieser Wert sagt aus, wie viele Regelinstanzen der Benutzer durchschnittlich pro Aufruf der Evaluierung hatte. Die nächsten beiden Spalten sind die Gesamtzahl der Evaluierungen von Regelinstanzen und Expressions, diese sind absolut und werden pro Evaluierung aufsummiert. Die letzte Spalte ist die durchschnittliche Dauer eine Evaluierung. In der letzten Zeile der Tabelle ist der Durchschnitt der vier Benutzer gebildet, für die Berechnung der durchschnittlichen Zeit einer Evaluierung ist die Initialisierung nicht berücksichtigt worden.

Benutzer	Eval.	Regelins.	Eval. Regelins.	Eval. Exp.	Eval. Dauer
Benutzer 1	14	33,14	40	82	5,79 ms
Benutzer 2	11	25,91	34	39	4,73 ms
Benutzer 3	15	42,13	47	233	6,21 ms
Benutzer 4	11	28,18	32	74	1,45 ms
pro Benutzer	12,75	32,34	38,25	107	4,54 ms

Tabelle 8.5: Konsistenzdaten mit Stand-Alone MA

Die Konsistenzdaten des Client-Server MA (Tabelle 8.6) wurden auf gleiche Art und Weise wie die Daten für den Stand-Alone MA erzeugt. Für die Durchschnittsberechnung werden die aufsummierten Werte durch die vier Benutzer gerechnet und für die durchschnittliche Dauer einer Evaluierung ist die Initialisierung nicht berücksichtigt worden.

Benutzer	Eval.	Regelins.	Eval. Regelins.	Eval. Exp.	Eval. Dauer
öffentlicher Bereich	1	30,00	30	0	119,89 ms
Benutzer 1	13	10,08	18	71	7,58 ms
Benutzer 2	10	11,80	14	24	5,90 ms
Benutzer 3	16	18,00	22	286	18,41 ms
Benutzer 4	10	11,20	18	49	2,91 ms
pro Benutzer	12	20,27	25,50	107,50	8,7 ms

Tabelle 8.6: Konsistenzdaten mit Client-Server MA

In Tabelle 8.7 sind die Durchschnittsdaten der beiden Verfahren als prozentuelle Änderung gegenübergestellt und liefern so die geforderten Kennzahlen für Frage *Q 1.1*.

Regelinstanzen M 1.1.1	Eval. Regelinstanzen M 1.1.2	Eval. Expressions M 1.1.3	Evaluierungsdauer M 1.1.4
-37,32%	-33,33%	+0,47%	+91,63%

Tabelle 8.7: Kennzahlen der Konsistenzdaten

Die erste Kennzahl *M 1.1.1* zeigt eine Verbesserung bei der Anzahl der Regelinstanzen durch den Einsatz der Client-Server Variante, es gibt um durchschnittlich 37,32% weniger Regelinstanzen pro Benutzer als bei der Stand-Alone Variante. Das heißt, während bei der Stand-Alone Variante jeder Benutzer alle Regelinstanzen besitzt, können bei der Client-Server Variante viele Regelinstanzen gemeinsam benutzt werden und müssen daher nur einmal gespeichert werden. Auf ein Gesamtsystem gesehen heißt dies auch, dass um 37,32% weniger Speicher pro Benutzer für die Regelinstanzen benötigt wird. Wie Abbildung 8.5 zu sehen, tritt dieser Effekt schon ab zwei Benutzern ein, denn solange nur ein Benutzer das Client-Server System einsetzt, besitzt er neben seinen eigenen Regelinstanzen auch die gesamten die öffentlichen Regelinstanzen. Sobald mehr Benutzer zum System kommen, sinkt die durchschnittliche Anzahl der Regelinstanzen pro Benutzer, denn ab diesem Zeitpunkt werden Regelinstanzen gemeinsam benutzt. Bei der Stand-Alone Variante hingegen, besitzt jeder Benutzer immer die durchschnittliche Anzahl der Regelinstanzen.

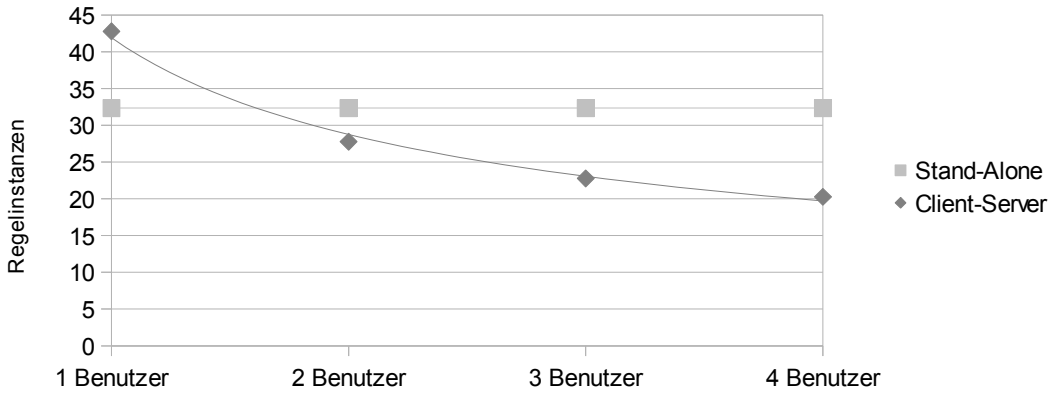


Abbildung 8.5: Vergleich Regelinstanzen

Durch die geringere durchschnittliche Anzahl an Regelinstanzen sinkt auch die durchschnittliche Anzahl der Evaluierungen der Regelinstanzen *M 1.1.2* um einen ähnlichen Wert, es gibt eine Verbesserung von 33,33% bei der Verwendung des Client-Server MA. Das heißt, es muss für das gleiche Ergebnis bei der Konsistenzprüfung in einem Client-Server System um 33,33% pro Benutzer weniger Rechenzeit aufgewendet werden, als bei einem Stand-Alone System. Die durchschnittliche Anzahl der Evaluierungen von Expressions *M 1.1.3* bleibt mit einem Anstieg von 0,47% nahezu identisch. Doch auch hier sollte eine Verbesserung zu erreichen sein, deshalb wäre dieser Bereich der Implementierung ein Einstiegspunkt für Verbesserungen.
Einen deutlichen Anstieg verzeichnet hingegen die durchschnittliche Dauer einer Evaluierung *M 1.1.4*, diese hat sich nahezu verdoppelt. Da die Verdoppelung noch immer im Millisekundenbereich ist, ist dies für den Benutzer nicht weiter von Bedeutung, doch eine Verdopplung einer Kennzahl nicht wünschenswert. Da der Ablauf der Konsistenzprüfung

durch das Model Analyzer Framework vorgegeben ist und daher die Zeiten für die Evaluierung grundsätzlich gleich sein sollten, muss der Grund für die Zunahme im Modellzugriff liegen, da dieser für RDF angepasst worden ist. Während bei der Stand-Alone Variante beim Zugriff auf Modelldaten EMF-Objekte verwendet werden, muss beim Client-Server MA am Server im RDF-Modell auch gesucht werden. Wenn im privaten Modell die benötigte Information nicht gefunden wird, wird zusätzlich das öffentliche Modell durchsucht. Ein Teil der Verzögerung könnte auch durch die verwendete RDF Implementierung entstehen. Diese Kennzahl zeigt auf jeden Fall noch einen Bereich für Verbesserungen des Systems auf.
Die ermittelten Kennzahlen sind von der Größe des gemeinsamen Modells, den verwendeten Konsistenzregeln und den vorhandenen Regelinstanzen abhängig. Beginnen die Benutzer mit einem großen Modell und sehr vielen Regelinstanzen wird der Effekt der Client-Server Variante sehr deutlich, denn die Benutzer verwenden viele öffentliche Regelinstanzen und durch die Änderungen der Benutzer entstehen verhältnismäßig wenig private Regelinstanzen. Je länger Änderungen von den Benutzer durchgeführt werden, ohne dass das Modell und die Konsistenzdaten auch wieder auf eine gemeinsame Basis zusammengeführt werden, desto kleiner wird der Vorteil der Client-Server Variante, da jeder einzelne Benutzer immer mehr eigene Konsistenzdaten besitzt und somit auch immer weniger von den gemeinsamen Konsistenzdaten profitieren kann. Der Extremfall bei der Client-Server Variante wäre, dass die Benutzer mit einem leeren Modell beginnen und somit nur eigene Konsistenzdaten erzeugen, dies würde dann einer Stand-Alone Variante entsprechen.

8.3.2 Modelldaten

Als nächstes wird Frage *Q 1.2* behandelt und damit der Client-Server mit dem Stand-Alone MA im Bereich der Modelldaten verglichen. Die Kennzahlen wurden mit den Kennzahlen der Konsistenzprüfung ermittelt. In Tabelle 8.8 sind die ermittelten Daten des Stand-Alone MA aufgelistet. Die Scopeelemente und Modellelemente sind Durchschnittszahlen, das heißt sie sagen aus wie viele Elemente der Benutzer durchschnittliche während des gesamten Ablaufes des Beispiels hatte. Die Modellzugriffe sind aufsummiert und sind somit die gesamten Modellzugriffe des Benutzers während der Simulation. Auch bei diesen Zahlen ist in der letzten Zeile der Tabelle der Durchschnittswert pro Benutzer gebildet.

Benutzer	Scopeelemente	Modellelemente	Modellzugriffe
Benutzer 1	75,71	51,43	2824,00
Benutzer 2	58,27	45,99	1628,00
Benutzer 3	79,67	52,33	4224,00
Benutzer 4	61,73	44,36	1224,00
pro Benutzer	68,85	48,28	2475,00

Tabelle 8.8: Modelldaten mit Stand-Alone MA

Die Konsistenzdaten des Client-Server MA (Tabelle 8.9) wurden auf die gleiche Art und Weise wie die Daten für den Stand-Alone MA erzeugt. Die Modellzugriffe des öffentlichen Bereichs entstehen einerseits durch die Initialisierung und andererseits durch den laufenden Betrieb, denn die privaten Regelinstanzen können auch die Modellelemente des öffentlichen Modells zugreifen. Die Modellzugriffe der einzelnen Benutzer entstehen somit

nur durch Zugriffe deren private Modelle. Die Durchschnittswerte in der letzten Zeile der Tabelle beziehen das öffentliche Modell in die Berechnung mit ein.

Benutzer	Scopeelemente	Modellelemente	Modellzugriffe
öffentlicher Bereich	73,00	49,00	3157,00
Benutzer 1	14,85	2,62	227,00
Benutzer 2	7,10	2,60	125,00
Benutzer 3	17,21	3,57	1143,00
Benutzer 4	8,80	8,30	167,00
pro Benutzer	30,24	16,52	1204,75

Tabelle 8.9: Modelldaten mit Client-Server MA

In Tabelle 8.10 sind die ermittelten Durchschnittsdaten als prozentuelle Änderung gegenübergestellt und liefern so die geforderten Kennzahlen für Frage *Q 1.2*.

Scopeelemente M 1.2.1	Modellzugriffe M 1.2.2	Modellelemente M 1.2.3
-56,08%	-51,32%	-65,78%

Tabelle 8.10: Kennzahlen der Modelldaten

Der Client-Server MA zeigt in allen Bereichen bei den Modelldaten eine deutliche Verbesserung im Vergleich zum Stand-Alone MA. Die Anzahl der Scopeelemente *M 1.2.1* und die Anzahl der Modellzugriffe *M 1.2.2* sind jeweils um mehr als 50% gesunken. Diese Zahlen können durch die bereits festgestellte Einsparung von Regelinstanzen erklärt werden. Dadurch dass weniger Regelinstanzen evaluiert werden müssen, müssen auch weniger Scopeelemente angelegt werden und diese Scopeelemente erzeugen dann auch weniger Modellzugriffe. Weniger Scopeelemente im Durchschnitt pro Benutzer bedeutet auch eine Einsparung von Speicherplatz in diesem Bereich. Generell werden sich die Anzahl der Scopeelemente und die Modellzugriffe ähnlich wie die Regelinstanzen aus Punkt 8.3.1 verhalten, denn sie hängen unmittelbar mit den Regelinstanzen zusammen.

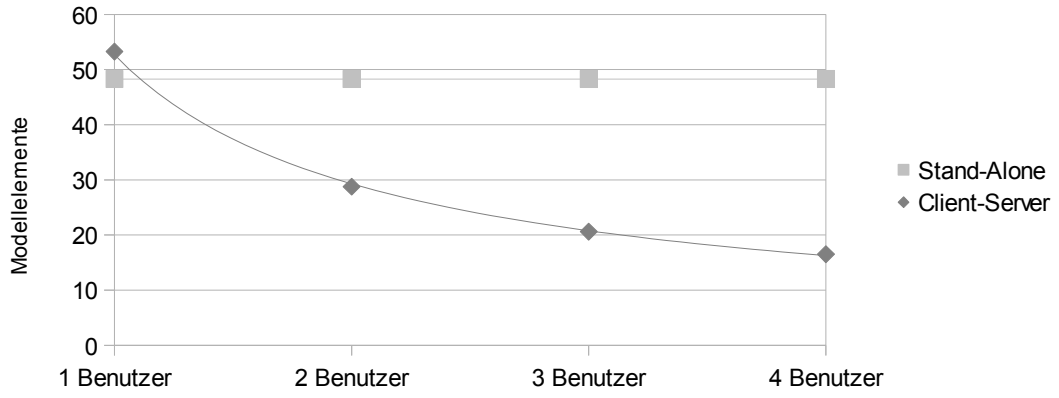


Abbildung 8.6: Vergleich Modellelemente

Die deutlichste Verbesserung beim Einsatz des Client-Server MA zeigt sich bei der durchschnittlichen Anzahl der Modellelemente pro Benutzer *M 1.2.3*. Hier kann eine Verbesserung von 65,78% festgestellt werden. Wie bei den Regelinstanzen tritt die Verbesserung bei der durchschnittlichen Anzahl der Modellelemente pro Benutzer schon ab zwei Benutzern ein. Dieses Verhalten ist in Abbildung 8.6 dargestellt. Während bei einem Benutzer noch das gesamte öffentliche Modell zu seinen privaten Modellelemente hinzukommt, wird ab dem zweiten Benutzer schon ein großer Teil des öffentlichen Modells gemeinsam benutzt, bei weiteren Benutzern verstärkt sich dieser Effekt weiter. Da beim Stand-Alone MA das gesamte Modell verwendet wird, besitzen die einzelnen Benutzer nahezu das gleiche Modell, obwohl sich nur wenige Modellelemente unterscheiden.
Auch die Kennzahlen der Modelldaten hängen von der Größe des gemeinsamen Modells ab, dies gilt vor allem für die durchschnittliche Anzahl der Modellelemente pro Benutzer. Zusätzlich spielen die vorgenommenen Änderungen am Modell eine große Rolle, denn sowohl das Hinzufügen und als auch das Löschen von Modellelemente aus dem öffentlichen Modell erhöht die Anzahl der Modellelemente im privaten Modell. Das Löschen von privaten Modellelementen senkt die Anzahl der Modellelemente. Werden nur Änderungen von Werten durchgeführt, erhöht dies die Anzahl der Modellelemente kaum, denn es werden nur die die neuen Werte im privaten Modell gespeichert. Wird mit einem großen Modell begonnen und die einzelnen Benutzer ändern nur Werte, wird die Verbesserung noch deutlicher, sobald viel hinzugefügt und auch aus dem öffentlichen Modell gelöscht wird, verschlechtert sich das Verhältnis zur Stand-Alone Variante. Auch hier kann der Extremfall eintreten, dass alle Benutzer alle Modellelemente ändern oder dass mit einem leeren öffentlichen Modell begonnen wird. In diesem Fall würde sich der Client-Server MA wieder wie ein Stand-Alone MA verhalten. Ein weiterer Grund, warum die durchschnittliche Anzahl der Modellelemente pro Benutzer ein besseres Ergebnis als die durchschnittliche Anzahl der Regelinstanzen pro Benutzer liefert, ist die Implementierung der Konsistenzprüfung. Bei einer Änderung eines Werts werden die betroffenen Regelinstanzen privat, während das Modellelement noch öffentlich bleibt. Es wird nur der eine geänderte Wert des Modellelements in das private Modell gespeichert. Das Hinzufügen von einem Modellelemente kann zur Folge haben, dass zwar nur ein privates Modellelemente angelegt wird, aber mehrerer neue private Regelinstanzen erzeugt werden.

8.3.3 Initialisierung

Der nächste Bereich der Evaluierung beschäftigt sich mit der Initialisierung des Modells und beantwortet zunächst mit einer Analyse von Modelldateien die Frage *Q 2.1*. Der Rational Software Modeler vom IBM speichert ein Modell im XML-Format XMI (XML Metadata Interchange), beim Client-Server MA wird RDF und zur Serialisierung eine XML-Darstellung verwendet. Es werden zwar keine RDF-Dateien gespeichert, aber die Größe einer RDF-Datei entspricht der Datenmenge, wie sie auch bei der Initialisierung vom Client zum Server übertragen werden muss. Aus diesem Grund werden die Dateigrößen der Serialisierung verglichen und daraus werden die benötigten Kennzahlen abgeleitet. Diese Frage ist wichtig, denn je kleiner die Menge der zu initialisierenden Daten ist, desto schneller können sie von Client zum Server übertragen werden und dadurch entsteht weniger Verzögerung für den Benutzer. In Tabelle 8.11 sind fünf Modelle mit einer Größe von 49 bis 2820 Modellelementen angeführt, für die Ermittlung der Größe der RDF-Datei (RDF 1) wurde das Modell am Client initialisiert, anstatt es zum Server zu senden, wurde es als Datei geschrieben.

Modell	Elemente	EMX M 2.1.1	RDF 1 M 2.1.1	Faktor 1 M 2.1.2	RDF 2	Faktor 2
caBIO	1656	1290 kB	16178 kB	20,29	8305 kB	6,44
Calendarium	2820	2510 kB	40426 kB	16,11	13447 kB	5,36
LightSwitch	49	38 kB	617 kB	16,24	203 kB	5,34
MicrowaveOven	628	268 kB	6733 kB	25,12	2322 kB	8,66
VOD	113	67 kB	1578 kB	23,55	468 kB	6,99
				20,26		6,56

Tabelle 8.11: Modelldateien bei der Initialisierung

Die Werte für die Speicherung als Datei *M 2.1.1* zeigen einen deutlichen Anstieg der Dateigröße und somit auch der Datenmenge, die vom Client zum Server übertragen werden muss. Auch der Faktor der Vergrößerung *M 2.1.2* mit durchschnittliche 20,26-fachen Vergrößerung verdeutlicht diesen Anstieg. Diese Größe der Datei ist damit zu erklären, dass die gesamte Objektstruktur des Modells serialisiert wird, selbst wenn Eigenschaften von Modellelementen nicht gesetzt sind. In Tabelle 8.11 ist mit Spalte RDF 2 noch eine weitere Variante der RDF-Serialisierung angeführt, sie wurde testweise durchgeführt und verzichtet auf die Serialisierung nicht gesetzter Modellelemente. Zwar wird diese Variante noch nicht beim Client-Server MA eingesetzt, doch könnte diese Verbesserung den Faktor der Vergrößerung der Datei auf 6,56 senken. Die zu übertragende Datenmenge kann wahrscheinlich durch Optimierungen noch weiter in Richtung der Größe einer EMX-Datei gesenkt werden. Das Diagramm aus Abbildung 8.7 verdeutlicht das Problem auch grafisch. Während das EMX-Format die Datenmenge auch bei größeren Modellen noch gering halten kann, steigt sie bei der verwendeten RDF-Konvertierung (RDF 1) sehr stark an. Bereits eine kleine Verbesserung, wie durch die Variante RDF 2, kann die Datenmenge beträchtlich senken. Ziel von Verbesserungen muss eine Annäherung an das EMX-Format sein.

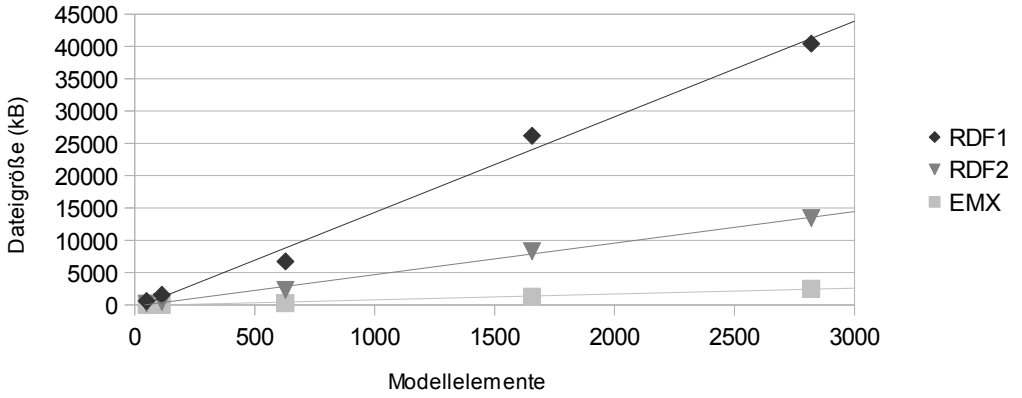


Abbildung 8.7: Verhältnis der Modelldateien

Die zweite Frage *Q 2.2* zur Initialisierung beschäftigt sich mit der Dauer der Initialisierung. In Tabelle 8.12 sind die benötigten Kennzahlen zur Beantwortung der Frage angeführt. Die Dauer der Konvertierung in das RDF-Format *M 2.2.1* hängt unmittelbar von der Anzahl der zu konvertierenden Modellelemente ab und entspricht in etwa dem

Doppelten der Modellelemente in Millisekunden. Dies bedeutet die Konvertierung kann einige Sekunden benötigen und ist somit für den Benutzer wahrnehmbar.Es wird jedoch nur bei der Initialisierung des Modells eine Konvertierung benötigt und somit muss nur ein Benutzer sie ausführen. Bei der Initialisierung kommt auch noch eine weitere Kennzahl zum Tragen, die Dauer der initialen Konsistenzprüfung am Server *M 2.2.2*. Diese Kennzahl kann mit der Dauer der initialen Konsistenzprüfung des Stand-Alone MA verglichen werden. Ähnlich wie bei der Dauer einer Evaluierung einer Änderung aus Punkt 8.3.1, benötigt auch die initiale Konsistenzprüfung mit der Verwendung von RDF-Modellen länger. Die initiale Konsistenzprüfung am Server benötigt durchschnittliche um einen Faktor von 4,32 länger.

Modell	Elemente	Stand-Alone	Client-Server M 2.2.2	Faktor	Konvertierung M 2.2.1
caBIO	1656	3029 ms	21247 ms	7,01	3526 ms
Calendarium	2820	1316 ms	5100 ms	3,88	6911 ms
LightSwitch	49	40 ms	97 ms	2,43	62 ms
MicrowaveOven	628	382 ms	2148 ms	5,62	1654 ms
VOD	113	80 ms	212 ms	2,65	312 ms
				4,32	

Tabelle 8.12: Dauer der Initialisierung

Damit die Gesamtzeit der Initialisierung ermittelt werden kann, muss zu der Zeit für die Konvertierung und der initialen Konsistenzprüfung noch die Zeit der Übertragung der Daten von Client zum Server gerechnet werden. Die Übertragungszeit hängt von der Verbindung des Clients zum Server ab, aber da die zu übertragenden Datenmengen bei großen Modellen im Megabyte-Bereich sind, kann die Initialisierung unter Umständen einige Minuten dauern. Zwar ist dies eine große Verzögerung, doch ist die Initialisierung in der implementierten Form nicht optimiert und sie dient nur dazu, dass am Server das Modell initialisiert und eine erste Konsistenzprüfung durchgeführt wird. Generell ist die Initialisierung ein Bereich, welcher in zukünftigen Versionen noch durch andere Ansätze, wie zum Beispiel einer dauerhaften Speicherung des Modells am Server oder einer Kopplung der Initialisierung mit einer Versionskontrolle, ersetzt werden kann.

8.3.4 Kommunikation

Das letzte Ziel und somit der letzte Bereich der Evaluierung beschäftigt sich mit den Auswirkungen der Kommunikation, Frage *Q 3.1* behandelt die übertragenen Datenmengen. Im Zuge des Beispiels wurden die Größen der einzelnen Änderungen aufgezeichnet und in Tabelle 8.13 sind die daraus resultierenden Durchschnittsgrößen der verschiedenen Änderungstypen aufgelistet. Eine einfache Änderung *M 3.1.1* besitzt mit durchschnittlich 0,61 kB eine sehr geringe Größe. Das Hinzufügen von Modellelementen *M 3.1.2* benötigt mit 19,86 kB deutlich mehr Daten und die größten Änderungen werden durch das Löschen von Modellelementen *M 3.1.3* mit durchschnittlich 44,70 kB erzeugt. Die genauere Betrachtung der einzelnen Größen der Änderungen zeigt, dass das Ändern und Hinzufügen von Modellelemente konstant große Änderungen in RDF erzeugt. Beim Löschen von Modellelementen kann die Größe der gesendeten Änderung nicht vorhergesagt werden, denn sie hängt von den gelöschten Elementen ab. Je mehr Modellelemente mit dem zu löschenden Modellelement zusammenhängen, desto größer wird der gelöschte Modellbereich. Dadurch

dass im Beispiel der Änderungstyp einer Modelländerung am häufigsten vorkommt, ist im Allgemeinen eine Änderung *M3.1.4* mit durchschnittlich 12,70 kB nicht sehr groß. Diese geringe Größe hat den Vorteil, dass die Anforderungen an die Verbindung zwischen Client und Server gering gehalten werden können und dass ein normaler Breitbandanschluss genug Kapazität hat, damit die Änderungen schnell übertragen werden können.

Aktion	Anzahl	Größe M 3.1.1-3.1.4		Zeit 1 M 3.2.1	Zeit 2	Zeit 3	Zeit 4 M 3.2.2
Ändern	24	622 b	0,61 kB	96 ms	351 ms	553 ms	329 ms
Hinzufügen	18	20338 b	19,86 kB	151 ms	681 ms	1273 ms	624 ms
Löschen	5	45776 b	44,70 kB	185 ms	1052 ms	1640 ms	680 ms
Änderung	47	13001 b	12,70 kB	129 ms	552 ms	936 ms	471 ms

Tabelle 8.13: durschnittliche Änderungsgröße pro Typ

Die geringe durchschnittliche Größe einer Änderung und die daraus zu erwartenden kurzen Übertragungszeiten führen direkt zu der letzten Frage dieser Evaluierung. Tabelle 8.13 beinhaltet auch die Kennzahlen zur Beantwortung von Frage *Q 3.2* über die Zeitverzögerungen der Kommunikation. Für die Erzeugung dieser Kennzahlen wurde das Beispiel in verschiedenen Umgebungen durchgespielt, Zeit 1 ergibt sich durch die Verwendung von Client und Server am gleichen Rechner. Danach wurden Client und Server physisch getrennt und verschiedene Verbindungsarten mit dem Internet gewählt. Für Zeit 2 wurde Internet mit 8 Mbit/s Down- und 786 kbit/s Upstream verwendet, Zeit 3 ergibt sich durch die Verwendung vom mobilem UMTS-Internet und Zeit 4 wurde im WLAN der Universität ermittelt. Wenn Client und Server am selben Rechner sind, ist die durchschnittliche Zeit für die Übertragung und Durchführung einer Änderung *M 3.2.1* mit 129 ms sehr gering. Werden die beiden Komponenten getrennt, wächst die Zeit für eine durchschnittliche Änderungszeit *M 3.2.2* je nach verwendeter Internetverbindung an. Das WLAN der Universität schneidet am besten ab, während das UMTS-Internet schon eine deutlichere Verzögerung spüren lässt. Es kann zusätzlich festgestellt werden, dass die Dauer einer Übertragung von der zu übertragenen Datenmenge abhängt, die Zeit der Evaluierung am Server fällt dabei, wie in Punkt 8.3.1 ermittelt, kaum ins Gewicht. Damit der Benutzer möglichst wenig Verzögerungen in der Benutzerschnittstelle spürt, ist asynchrone Kommunikation von Vorteil, also dass die Kommunikation in einem eigenen Thread läuft und die Benutzerschnittstelle nicht blockieren kann.

Kapitel 9

Ausblick und Fazit

9.1 Ausblick

Bei der Diskussion und der Suche nach der geeigneten Variante, wie die Konsistenzprüfung in einer Mehrbenutzerumgebung funktionieren kann, gab es die drei Hauptbereiche Architektur, Aufteilung und Regeln in Teilmodellen. Im Zuge der Ausarbeitung dieser Themen wurden auch weitere Bereiche der Mehrbenutzerumgebung diskutiert. Einige dieser Ideen konnten nicht berücksichtigt werden, da sie über das Kernthema der Arbeit hinausgehen. Auch die Konzeption, Implementierung und Evaluierung zeigten weitere Bereiche auf, wie sich diese Masterarbeit und generell das Thema der Konsistenzprüfung in einer Mehrbenutzerumgebung weiter entwickeln kann. Die nachfolgenden Punkte sind eine Zusammenfassung welche Bereiche nicht berücksichtigt wurden und wie die Arbeit weitergeführt werden kann.

Rückführung in ein Gesamtmodell

Eine der wichtigsten Punkte, wie diese Arbeite erweitert werden kann, ist die Rückführung der privaten Bereiche in den öffentlichen Bereich, denn das Ergebnis der kollaborativen Arbeit soll ein gemeinsamen Modell sein. Ein Gesamtmodell ist der nächste logische Schritt, nachdem die Benutzer Änderungen durchgeführt haben, diese Änderungen in die privaten Bereiche geschrieben wurde und auch in diesen Bereichen die Konsistenz geprüft wurde. Neben der Rückführung der Modelldaten müssen auch die Konsistenzdaten vom privaten in den öffentlichen Bereich übertragen werden. Natürlich können bei einer Zusammenführung der Bereiche auch Konflikte auftreten, da die einzelnen Benutzer auch gleiche Modellbereiche geändert haben können. Es muss eine Möglichkeit gefunden werden, wie diese Konflikte erkannt und behandelt werden können. Eine Möglichkeit, wie Konflikte vermieden werden können, basiert auf der Idee einer Versionskontrolle und ist die Versionierung des öffentlichen Modells. Sobald ein privater Bereich in den öffentlichen Bereich gespeichert wird, entsteht eine neue Version des öffentlichen Modells mit den getätigten Änderungen. Die anderen Benutzer können unabhängig davon mit der vorherigen Version des öffentlichen Bereichs weiter arbeiten. Wenn es keine Konflikte mit ihren Daten gibt, können auch die anderen Benutzer die neue Version des öffentlichen Bereichs verwenden, bei Konflikten kann die alte Version verwenden werden und die Konflikte brauchen nicht sofort behandelt werden.

Möglichkeiten der Zusammenarbeit

Am Server gibt es ein gemeinsames Modell und jeder der einzelnen Benutzer hat sein eigenes privates Modell. Normalerweise können andere Benutzer nicht auf diese privaten Bereiche zugreifen. Erst durch die Rückführung der privaten Bereiche in den öffentlichen Bereich werden die Daten für die anderen Benutzer zugänglich. Dadurch dass die Modelldaten aber bereits am Server vorhanden sind, können noch vor einer Rückführung neue Wege in der Zusammenarbeit gegangen werden. Für Entwickler kann es wichtig sein zu wissen, welche Änderungen die Kollegen durchführen. Die einzelnen Benutzer sollen Modellbereiche für andere Benutzer freigeben können. Diese Bereiche können dann zum Beispiel abonniert werden und so können Änderungen in Echtzeit auch in den privaten Bereich eines anderen Benutzers geschrieben werden. Neben der Variante der automatischen Änderungen soll auch die Möglichkeit bestehen, dass ein Benutzer manuell den freigegebenen Bereich eines anderen Benutzers in seinen eigenen Bereich importieren kann. Eine weitere Möglichkeit der Zusammenarbeit wäre, dass für einen Benutzer die Kombination aus öffentlichen und privaten Bereich eines anderen Benutzers angezeigt wird und der Benutzer kann das Modell so verwenden, als wäre es bereits zusammengeführt.

Regelverwaltung

In der implementierten Form der Konsistenzprüfung in einer Mehrbenutzerumgebung werden die Regeln am Server im Quellcode statisch definiert und sind für alle Benutzer gleich. Sie können nicht dynamisch von den einzelnen Benutzern verwaltet werden. Zukünftige Versionen können die Möglichkeit bieten, dass die Benutzer die Regeln selbst verwalten können. Zusätzlich soll es möglich sein, dass für jeden Benutzer und Bereich eigene Konsistenzregeln definiert werden können. Es soll auch die Möglichkeit bestehen, dass die Konsistenzregeln von anderen Benutzern verwendet werden können, beziehungsweise dass Konsistenzregeln für andere Benutzer explizit freigegeben oder gesperrt werden können.

Sicherheit

Im Kapitel über die Ideen und Varianten wurden Sicherheitsmechanismen und Berechtigungsstufen innerhalb eines Modells diskutiert und als Vorteil für den Einsatz eines Servers angegeben. Diese Mechanismen sind in der derzeitigen Form noch nicht implementiert, da sie über das Kernthema hinaus gehen, für Erweiterungen sind sie aber ein interessantes Thema. Die Berechtigungen können neben den Modelldaten auch auf die Konsistenzdaten ausgeweitet werden, sodass bestimmte Konsistenzregeln nur von bestimmten Benutzern bearbeitet, verwendet oder neu evaluiert werden dürfen. Auch der Einsatz von verschlüsselten Datenverbindungen eine sinnvolle Erweiterung. Wenn über das Internet gearbeitet wird, sollen die Änderungen am Modell für Dritte nicht lesbar sein, da die Modelle meist internes Wissen beinhalten.

Persistenz

Es fehlt eine Möglichkeit die Konsistenzdaten und Modelldaten dauerhaft zu speichern, beziehungsweise sie wieder einzulesen. Die Speicherung der Daten war für die Entwicklung in dieser Arbeit nicht notwendig, da das Modell wieder am Server immer initialisiert werden kann. Eine Speicherung ist aber auch für die privaten Bereiche wichtig, da die Änderungen mit jedem Neustart des Servers oder mit jeder neuen Initialisierung verloren gehen. Wenn alle Modell- und Konsistenzdaten am Server gespeichert werden, kann ein

Benutzer immer wieder mit seiner Arbeit fortfahren. Es ist dann auch möglich, den Client zu wechseln, ohne dass Daten verloren gehen oder erneut berechnet werden müssten.

9.2 Fazit

Diese Arbeit hat einen Ansatz für die Konsistenzprüfung von Software Design Modellen in Mehrbenutzerumgebungen präsentiert. Als gemeinsame Datenbasis wird ein zentrales Modell eingesetzt, welches in zwei Bereiche unterteilt ist. Diese sind ein öffentlicher Bereich für die gemeinsamen Modelldaten und je ein privater Bereich für die Änderungen der einzelnen Benutzern. Für jeden dieser Bereiche gibt es eine eigene Konsistenzprüfung, wobei auch diese Konsistenzprüfungen wieder gemeinsam Daten verwenden.
Das wichtigste Ergebnis dieser Arbeit ist die Konsistenzprüfung und der spezielle Aufbau der Modelle. Durch den öffentlichen Bereich mit seinen privaten Änderungsbereichen kann eine Möglichkeit der kollaborativen Modellierung angeboten werden, bei der keine redundanten Modelldaten entstehen und bei der effizient Konsistenz geprüft werden kann. Dieses Kapitel hat bereits einen Ausblick gegeben, wie der Ansatz noch weiter verfolgt werden kann und dieser Modellaufbau als Basis für viele Möglichkeiten der interaktiven Kollaboration bei der Modellierung verwendet werden kann.
Das entwickelte Werkzeug hat gezeigt, dass die Ideen dieser Arbeit auch in der Praxis funktionieren und vor allem dass sie auch in guter Geschwindigkeit funktionieren. Zwar hat die Evaluierung gezeigt, dass die Initialisierung etwas länger dauert, doch die Änderungen werden nahezu ohne Verzögerung verarbeitet. Dieser Umstand ist für das Design und die Bedienbarkeit einer Benutzerschnittstelle sehr wichtig, denn Änderungen am Modell werden laufend durchgeführt und Verzögerungen würden den Entwickler bei der Arbeit stören. Die Konsistenzprüfung soll, obwohl Daten zum Server hin und wieder zurück gesendet werden müssen, möglichst unmittelbar statt finden. Durch die kleine Größe der Änderung ist dies auch möglich und unmittelbar nach einer Änderung ist das Ergebnis der Konsistenzprüfung verfügbar.
Die Evaluierung hat Vor- und Nachteile des Ansatzes dieser Arbeit ermittelt. Die Konsistenzprüfung ein einer Mehrbenutzerumgebung hat die Erwartungen der Ziele erreicht. Der Einsatz des Client-Server Systems ist bei kollaborativen Modellierung dem Einsatz von einzelnen lokalen Konsistenzprüfungen in den Bereichen der Modell- und Konsistenzdaten überlegen. Die entdeckten Nachteile haben Bereiche für Verbesserungen aufgezeigt, jedoch das Konzept und der Ansatz dieser Arbeit sind durch die Evaluierung bestätigt worden.
Abschließend kann festgehalten werden, dass die vorgestellten Konzepte dieser Arbeit eine gute Basis für zukünftige Forschung und Systeme bilden. Durch die weltweite Vernetzung rücken Entwickler immer näher zusammen und die Arbeit an gemeinsamen Projekten wird zur täglichen Aufgabe. Kollaborative Modellierung von Software und die daraus resultierende Notwendigkeit die Modelle durch Konsistenzprüfung möglichst fehlerfrei zu halten, sind wichtige Beiträge und Hilfsmittel zum vernetzten Arbeiten.

Literaturverzeichnis

[Apa11] Apache Software Foundation. Apache HttpComponents, 2011. http://hc.apache.org.

[Bac09] Markus Bach. OSGi als Webserver. *JavaSPEKTRUM*, pages 52–55, 2009.

[BCR94] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The Goal Question Metric Approach. In *Encyclopedia of Software Engineering*. Wiley, 1994.

[BMMM08] Xavier Blanc, Isabelle Mounier, Alix Mougenot, and Tom Mens. Detecting Model Inconsistency through Operation-Based Model Construction. In *Proceedings of the 30th international conference on Software engineering*, pages 511–520, 2008.

[BMMM09] Xavier Blanc, Alix Mougenot, Isabelle Mounier, and Tom Mens. Incremental Detection of Model Inconsistencies Based on Model Operations. In *Advanced Information Systems Engineering*, pages 32–46. Springer Berlin / Heidelberg, 2009.

[BMS09] Christian Bartelt, Georg Molter, and Tim Schumann. A Model Repository for Collaborative Modeling with the Jazz Development Platform. In *42nd Hawaii International Conference on System Sciences*, pages 1–10, 2009.

[BPE+10] Jae young Bang, Daniel Popescu, George Edwards, Nenad Medvidovic, Naveen Kulkarni, Girish M. Rama, and Srinivas Padmanabhuni. CoDesign – A Highly Extensible Collaborative Software Modeling Framework. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 243–246, 2010.

[BS10] Christian Bartelt and Björn Schindler. Technology Support for Collaborative Inconsistency Management in Model Driven Engineering. In *43rd Hawaii International Conference on System Sciences*, pages 1–7, 2010.

[CDD+04] Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: Implementing the Semantic Web Recommendations. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 74–83, 2004.

[CDK05] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, 4 edition, 2005.

[Ecl11a] Eclipse Foundation. Eclipse 3.4 Documentation, 2011. http://help.eclipse.org/ganymede/.

[Ecl11b] Eclipse Foundation. Eclipse Equinox, 2011. http://www.eclipse.org/equinox.

[Ecl11c] Eclipse Foundation. Eclipse Modeling Framework Project (EMF), 2011. http://www.eclipse.org/emf.

[Egy06] Alexander Egyed. Instant Consistency Checking for the UML. In *Proceedings of the 28th international conference on Software engineering*, pages 381–390, 2006.

[Egy07a] Alexander Egyed. Fixing Inconsistencies in UML Design Models. In *Proceedings of the 29th international conference on Software Engineering*, pages 292–301, 2007.

[Egy07b] Alexander Egyed. UML/Analyzer: A Tool for the Instant Consistency Checking of UML Models. In *Proceedings of the 29th International Conference on Software Engineering*, pages 793–796, 2007.

[Egy11] Alexander Egyed. Automatically Detecting and Tracking Inconsistencies in Software Design Models. *IEEE Transactions on Software Engineering*, 37(2):188 –204, 2011.

[Fie00] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[HKRS08] Pascal Hitzler, Markus Krötzsch, Sebastian Rudolph, and York Sure. *Semantic Web*. Springer Berlin / Heidelberg, 2008.

[HRG+08] T. Hildenbrand, F. Rothlauf, M. Geisser, A. Heinzl, and T. Kude. Approaches to Collaborative Software Development. In *International Conference on Complex, Intelligent and Software Intensive Systems*, pages 523 –528, 2008.

[Jen11a] Jena. Jena – A Semantic Web Framework for Java, 2011. http://jena.sourceforge.net/.

[Jen11b] Jena. Jena Documentation, 2011. http://jena.sourceforge.net/documentation.html.

[Jer98] Mikael Jern. "Thin" vs. "Fat" visualization clients. In *Proceedings of the working conference on Advanced visual interfaces*, pages 270–273, 1998.

[KC04] Graham Klyne and Jeremy J. Carroll. RDF Concepts and Abstract Syntax. W3C, 2004. http://www.w3.org/TR/rdf-concepts/.

[KHvWH10] Maximilian Koegel, Markus Herrmannsdoerfer, Otto von Wesendonk, and Jonas Helming. Operation-based Conflict Detection. In *Proceedings of the 1st International Workshop on Model Comparison in Practice*, pages 21–30, 2010.

[LMT09] Francisco J. Lucas, Fernando Molina, and Ambrosio Toval. A systematic review of UML model consistency management. *Information and Software Technology*, 51:1631–1645, 2009.

[LP09] Louis Ling and Sellappan Palaniappan. Online CASE Tool for Collaborative Software Modelling. In *Proceedings of the 11th International Conference on Information Integration and Web-based Applications & Services*, pages 452–456, 2009.

[LZS+06] Siyuan Liu, Yang Zheng, Haifeng Shen, Steven Xia, and Chengzheng Sun. Real-Time Collaborative Software Modeling Using UML with Rational Software Architect. In *International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 1–9, 2006.

[MBG09] Alix Mougenot, Xavier Blanc, and Marie-Pierre Gervais. D-Praxis : A Peer-to-Peer Collaborative Model Editing Framework. In *Distributed Applications and Interoperable Systems*, volume 5523, pages 16–29. Springer Berlin / Heidelberg, 2009.

[MBG10] Alix Mougenot, Xavier Blanc, and Marie-Pierre Gervais. Inconsistency Detection in Distributed Model Driven Software Engineering Environments. In *Proceedings of the third Workshop on Living with Inconsistencies in Software Development*, pages 6–11, 2010.

[MBSS11] Jonathan Michaux, Xavier Blanc, Marc Shapiro, and Pierre Sutra. A Semantically Rich Approach for Collaborative Model Edition. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1470–1475, 2011.

[McB01] Brian McBride. Jena: Implementing the RDF Model and Syntax Specification. In *Proceedings of the 2nd International Workshop on the Semantic Web.*, 2001.

[MM04] Frank Manola and Eric Miller. RDF Primer. W3C, 2004. http://www.w3.org/TR/rdf-primer/.

[MVA10] Jeff McAffer, Paul VanderLei, and Simon Archer. *OSGi and Equinox: Creating Highly Modular Java Systems.* Addison-Wesley, 2010.

[MVDS07] Tom Mens and Ragnhild Van Der Straeten. Incremental Resolution of Model Inconsistencies. In *Recent Trends in Algebraic Development Techniques*, pages 111–126. Springer Berlin / Heidelberg, 2007.

[NCEF02] Christian Nentwich, Licia Capra, Wolfgang Emmerich, and Anthony Finkelstein. xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2(2):151–185, 2002.

[NEF03] Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. Consistency Management with Repair Actions. In *Proceedings of the 25th International Conference on Software Engineering*, pages 455–464, 2003.

[NEFE03] Christian Nentwich, Wolfgang Emmerich, Anthony Finkelstein, and Ernst Ellmer. Flexible Consistency Checking. *ACM Transactions on Software Engineering and Methodology*, 12:28–63, 2003.

[PGTL08] Victor Penichet, Jose Gallud, Ricardo Tesoriero, and Maria Lozano. Design and Evaluation of a Service Oriented Architecture-Based Application

to Support the Collaborative Edition of UML Class Diagrams. In *Lecture Notes in Computer Science*, volume 5103, pages 389–398. Springer Berlin / Heidelberg, 2008.

[RE10] Alexander Reder and Alexander Egyed. Model/Analyzer: A Tool for Detecting, Visualizing and Fixing Design Errors in UML. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 347–348, 2010.

[Red11] Alexander Reder. Inconsistency Management Framework for Model-Based Development. In *ICSE*, pages 1098–1101, 2011.

[RR07] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly, first edition, 2007.

[Sin92] Alok Sinha. Client-server computing. *Commun. ACM*, 35:77–98, 1992.

[SNEC08] Mehrdad Sabetzadeh, Shiva Nejati, Steve Easterbrook, and Marsha Chećhik. Global Consistency Checking of Distributed Models with TReMer+. In *Proceedings of the 30th international conference on Software engineering*, pages 815–818, 2008.

[SW04] Ralf Steinmetz and Klaus Wehrle. Peer-to-Peer-Networking and -Computing. *Informatik-Spektrum*, 27(1):51–54, 2004.

[SZ01] George Spanoudakis and Andrea Zisman. Inconsistency Management in Software Engineering: Survey and Open Research Issues. In *Handbook of Software Engineering and Knowledge Engineering*, pages 329–380. World Scientific, 2001.

[TS06] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., 2006.

[Vin08] Steve Vinoski. RESTful Web Services Development Checklist. *Internet Computing, IEEE*, 12(6):96–95, 2008.

[Whi07] Jim Whitehead. Collaboration in Software Engineering: A Roadmap. In *Future of Software Engineering*, pages 214–225, 2007.

[WHKL08] Gerd Wütherich, Nils Hartmann, Bernd Kolb, and Matthias Lübken. *Die OSGi Service Platform: Eine Einführung mit Eclipse Equinox*. dpunkt.Verlag, 2008.

[XHZ+09] Yingfei Xiong, Zhenjiang Hu, Haiyan Zhao, Hui Song, Masato Takeichi, and Hong Mei. Supporting Automatic Model Inconsistency Fixing. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 315–324, 2009.

Anhang A

Informationen

Designregeln

DR01 Class	**parent class should not have an attribute referring to a child class** let children : Set(NamedElement) = self.namespace.oclAsType(Package). packagedElement -> select(pe : PackageableElement | pe.oclIsTypeOf(Class) and pe.oclAsType(Class).allParents() -> includes(self)) in self.ownedAttribute -> forAll(p : Property | p.type.oclIsTypeOf(Class) implies children -> excludes(p.type.oclAsType(Class)))
DR02 Class	**parent class should not have a method with a parameter referring to a child class** let children : Set(NamedElement) = self.namespace.oclAsType(Package). packagedElement -> select(pe : PackageableElement | pe.oclIsTypeOf(Class) and pe.oclAsType(Class).allParents() -> includes(self)) in self.ownedOperation -> forAll(o : Operation | o -> ownedParameter -> forAll(p : Parameter | p.type.oclIsTypeOf(Class) implies children -> excludes(p.type.oclAsType(Class))))
DR03 Message	**message action must be defined as an operation in receiver's class** self.receiveEvent.oclAsType(InteractionFragment).covered -> forAll(represents.type.oclAsType(Class).ownedOperation -> exists(name = self.name))
DR04 Message	**message direction must match class association** self.receiveEvent.oclAsType(InteractionFragment).covered -> exists(let rc : Class = represents.type.oclAsType(Class) in self.sendEvent.oclAsType(InteractionFragment).covered -> exists(let sc : Class = represents.type.oclAsType(Class) in sc.ownedAttribute -> exists(association <> null implies type = rc)))

DR05 Association	**the connected classifiers of the association end should be included in the namespace of the association** self.memberEnd <> null and self.memberEnd -> forAll(p | p.type <> null and p.type.namespace = self.namespace)
DR06 Association	**association ends must have unique name within the association** self.memberEnd -> forAll(p1, p2 : Property | p1 <> p2 implies p1.name <> p2.name)
DR07 Association	**at most one association end may be an aggregation or composition** self.memberEnd -> size() > 0 implies self.memberEnd -> select(p | p.aggregation <> AggregationKind::none) -> size() <= 1
DR08 Class	**a classifier my not declare an attribute that has been declared in parent classifiers** self.allParents() -> forAll(c : Classifier | c.oclAsType(Class).ownedAttribute -> forAll(p : Property | class -> ownedAttribute -> collect(name) -> excludes(p.name)))
DR09 Class	**a class may use unique attribute names** self.ownedAttribute -> forAll(p1, p2:Property | p1 <> p2 implies p1.name <> p2.name)
DR10 Property	**a classifier may not belong by composition to more than one composite classifier** (self.association <> null and self.aggregation = AggregationKind::composite) implies (self.upper >= 0 and self.upper <= 1)
DR11 Package	**the elements owned by a namespace must have unique names** self.packagedElement -> forAll(e1, e2:PackageableElement | (e1 <> e2) implies (e1.name <> e2.name))
DR12 Interface	**an interface can only contain public operations and no attributes** self.ownedAttribute -> forAll(pr : Property | pr.association <> null or pr.visibility = VisibilityKind::public) and self.ownedOperation -> forAll(o:Operation | o.visibility = VisibilityKind::public)
DR13 Class	**no two methods may have the same signature in a classifier** self.ownedOperation -> forAll(o1, o2:Operation | o1 <> o2 implies (o1.name <> o2.name or o1.ownedParameter->size() <> o2.ownedParameter -> size() or let ops1 : Collection(Type) = o1.ownedParameter -> collect(type) in let ops2:Collection(Type) = o2.ownedParameter -> collect(type) in ops1 -> exists(t : Type | ops2 -> excludes(t)) or ops2 -> exists(t : Type | ops1 -> excludes(t))))
DR14 Operation	**method parameters must have unique names** self.ownedParameter -> forAll(p1, p2:Parameter | p1 <> p2 implies p1.name <> p2.name)

DR15 Operation	**type of method parameters must be included in the namespace of the method owner** self.ownedParameter -> forAll(p : Parameter|p.type <> null implies p.type.namespace = self.owner.oclAsType(Class).namespace)
DR16 Generalization	**the parent must be included in the namespace of the generalizable element** self.source -> forAll(e1: Element | e1.oclIsKindOf(NamedElement) implies self.target -> forAll(e2 : Element | e2.oclIsKindOf(NamedElement) and e1.oclAsType(NamedElement).namespace = e2.oclAsType(NamedElement).namespace))
DR17 Class	**no circular inheritance** not allParents() -> includes(self)
DR18 Transition	**statechart action must be defined as an operation in owner's class** self.owner.oclAsType(Region).stateMachine <> null implies let classifier : BehavioredClassifier = self.owner.oclAsType(Region).stateMachine.getContext() in classifier.oclIsTypeOf(Class) implies classifier.oclAsType(Class).ownedOperation -> exists(o : Operation|o.name = self.name)

RDF Beispiel

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:uml="http://sea.uni-linz.ac.at/uml/"
  xmlns:ma="http://sea.uni-linz.ac.at/ma/" >

  <rdf:Description rdf:about="http://sea.uni-linz.ac.at/uml/_-17-gL52EeCL7dVeniE2CQ">
    <ma:changetype>9046185221864-ADD</ma:changetype>
    <ma:type>org.eclipse.uml2.uml.Class</ma:type>
    <uml:name rdf:resource=
      "http://sea.uni-linz.ac.at/uml/_-17-gL52EeCL7dVeniE2CQ/name"/>
    <uml:visibility rdf:resource=
      "http://sea.uni-linz.ac.at/uml/_-17-gL52EeCL7dVeniE2CQ/visibility"/>
    <uml:package rdf:resource=
      "http://sea.uni-linz.ac.at/uml/_-17-gL52EeCL7dVeniE2CQ/package"/>
    <uml:isAbstract rdf:resource=
      "http://sea.uni-linz.ac.at/uml/_-17-gL52EeCL7dVeniE2CQ/isAbstract"/>
    <uml:owner rdf:resource=
      "http://sea.uni-linz.ac.at/uml/_-17-gL52EeCL7dVeniE2CQ/owner"/>

    <!-- Viele der UML-Features sind wegen der Lesbarkeit entfernt worden. -->
    <!-- Beispiele: ownedTrigger, ownedComment, powertypeExtent, ownedBehavior, ... -->

  </rdf:Description>

  <rdf:Description rdf:about=
      "http://sea.uni-linz.ac.at/uml/_-17-gL52EeCL7dVeniE2CQ/name">
    <ma:value>Class2</ma:value>
    <ma:type>java.lang.String</ma:type>
    <ma:name>name</ma:name>
    <ma:changetype>9046185476925-ADD;9046189712379-SET</ma:changetype>
    <ma:parent rdf:resource="http://sea.uni-linz.ac.at/uml/_-17-gL52EeCL7dVeniE2CQ"/>
  </rdf:Description>

  <rdf:Description rdf:about=
```

```
35        "http://sea.uni-linz.ac.at/uml/_-17-gL52EeCL7dVeniE2CQ/visibility">
36      <ma:value>public</ma:value>
37      <ma:type>org.eclipse.uml2.uml.VisibilityKind</ma:type>
38      <ma:name>visibility</ma:name>
39      <ma:parent rdf:resource="http://sea.uni-linz.ac.at/uml/_-17-gL52EeCL7dVeniE2CQ"/>
40      <ma:changetype>9046185520506-ADD</ma:changetype>
41    </rdf:Description>
42
43    <rdf:Description rdf:about=
44        "http://sea.uni-linz.ac.at/uml/_-17-gL52EeCL7dVeniE2CQ/package">
45      <ma:value rdf:resource="http://sea.uni-linz.ac.at/uml/_VYiN4EiPEd-Xye0lIHC77w"/>
46      <ma:type>org.eclipse.uml2.uml.Package</ma:type>
47      <ma:name>package</ma:name>
48      <ma:parent rdf:resource="http://sea.uni-linz.ac.at/uml/_-17-gL52EeCL7dVeniE2CQ"/>
49      <ma:changetype>9046186389331-ADD</ma:changetype>
50    </rdf:Description>
51
52    <rdf:Description rdf:about=
53        "http://sea.uni-linz.ac.at/uml/_-17-gL52EeCL7dVeniE2CQ/isAbstract">
54      <ma:value>false</ma:value>
55      <ma:type>java.lang.Boolean</ma:type>
56      <ma:name>isAbstract</ma:name>
57      <ma:parent rdf:resource="http://sea.uni-linz.ac.at/uml/_-17-gL52EeCL7dVeniE2CQ"/>
58      <ma:changetype>9046186487109-ADD</ma:changetype>
59    </rdf:Description>
60
61    <rdf:Description rdf:about=
62        "http://sea.uni-linz.ac.at/uml/_-17-gL52EeCL7dVeniE2CQ/owner">
63      <ma:value rdf:resource="http://sea.uni-linz.ac.at/uml/_VYiN4EiPEd-Xye0lIHC77w"/>
64      <ma:type>org.eclipse.uml2.uml.Package</ma:type>
65      <ma:name>owner</ma:name>
66      <ma:parent rdf:resource="http://sea.uni-linz.ac.at/uml/_-17-gL52EeCL7dVeniE2CQ"/>
67      <ma:changetype>9046185396747-ADD</ma:changetype>
68    </rdf:Description>
69
70    <rdf:Description rdf:about=
71        "http://sea.uni-linz.ac.at/uml/_VYiN4EiPEd-Xye0lIHC77w/packagedElement">
72      <rdf:_1 rdf:resource="http://sea.uni-linz.ac.at/uml/_-17-gL52EeCL7dVeniE2CQ"/>
73      <ma:name>packagedElement</ma:name>
74      <ma:parent rdf:resource="http://sea.uni-linz.ac.at/uml/_VYiN4EiPEd-Xye0lIHC77w"/>
75      <ma:changetype>9046187783922-SET_ADD</ma:changetype>
76      <rdf:type rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Bag"/>
77    </rdf:Description>
78
79    <!-- Details der nicht beschriebenen UML-Features beginnen ab hier  -->
80
81  </rdf:RDF>
```

Anhang B

Lebenslauf

Anhang C

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplom- bzw. Magisterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, 30.09.2011 Andreas Gruber