Prototypische Migration eines webbasierten UML-Werkzeugs zur Steigerung der Performance und Verbesserung der Testbarkeit

Bachelorthesis

Fraunhofer FOKUS

Eingereicht von

Tobias Fox
Mat-Nr.: 843616
im Studiengang Medieninformatik
Fachbereich VI der Beuth-Hochschule für Technik Berlin

Zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

Betreuender Dozent: Dipl.-Inform. Klaus Scholl, M.Sc.
Gutachterin: Prof. Dr. Simone Strippgen
Wissenschaftliche und technische Betreuung: M. Sc. Marcus Engelhardt, Fraunhofer Fokus

Zusammenfassung

Das Fraunhofer Institut für offene Kommunikationssysteme (FOKUS) entwickelt eine webbasierte Anwendung, womit Modelle beispielsweise der Unified Modeling Language erstellt und modifiziert werden können. Dabei kommen verschiedene Frameworks zum Einsatz, wie beispielsweise die Remote Application Platform (RAP). Dieses Framework ermöglicht es, die Anwendung für unterschiedliche Klienten anzubieten. Während der Entwicklung mit RAP offenbarten sich Probleme des Frameworks, welche in Nachteilen der Testbarkeit des UML-Werkzeuges resultieren und sich negativ auf die Performance der Software auswirken.

Im Rahmen dieser Arbeit soll ein Ansatz zur Verbesserung der Testbarkeit und zur Steigerung der Performance entwickelt und anhand festgelegter Kriterien evaluiert werden. Nach der prototypischen Implementierung des Ansatzes, welcher einen Wechsel der bisher verwendeten Zielplattform Eclipse IDE auf die cloudbasierte Eclipse Che Plattform vorsieht, wird eine Evaluation des Prototyps durchgeführt und analysiert, welche Auswirkungen die Migration hat.

Abstract

The „Fraunhofer Institut für offene Kommunikationssysteme“ (FOKUS) develops a web-based application for creating and manipulating models, e.g. of the Unified Modeling Language (UML). This software uses different frameworks for example the Remote Application Platform (RAP). The RAP framework allows delivering the application to different clients. During the development, RAP revealed some problems which result in disadvantages of the testability of the UML-tool and also influence the tools performance in a negative way.

In this thesis, an approach to improve testability and performance will be developed and evaluated using established criteria. After the prototypical implementation of the approach, which involves a change from the previously used Eclipse IDE target platform to the cloud-based Eclipse Che platform, an evaluation of the prototype will be carried out and the effects of the migration will be analysed.

Vorbemerkungen

Vorausgesetzte Technologiekenntnisse

Aufgrund der schriftlichen Begrenzung werden folgende Technologiekenntnisse vorausgesetzt:

- Die Unified Modeling Language
- Die Client-Server-Architektur
- Das Entwurfsmuster Fassade
- REST
- Maven und das Software-Paradigma Konventionen vor Konfigurationen
- Das Eclipse Modeling Framework
- Die Anwendung von Java, JavaScript, HTML, CSS

Typografische Konventionen

Folgende typografische Konventionen werden für diese Arbeit verwendet:

- Die erstmalige Verwendung von Begriffen wird in Anführungszeichen sowie in kursiver Schreibweise dargestellt. Findet eine mehrfache Verwendung in der Arbeit statt, wird in Klammern die Abkürzung eingeführt.
- Quelltextfragmente werden als Listings bezeichnet und zeilenweise nummeriert dargestellt. Quelltext oder Namen von Dateien beziehungsweise Klassen werden im Fließtext mit der Schriftart `Courier New` hervorgehoben.
  Beispiel:

```
1  public class Test {
2  // Kommentar
3  }
```

- Literaturquellen im Autor-Datum-Format werden auf folgende Weise referenziert: [<4 Buchstaben des Autors oder der Autoren><Kurzschreibweise der Jahreszahl>].
  Beispiel: [ChSB07]
- Ist kein Autor angegeben, werden die ersten vier Buchstaben des Titels verwendet.
- Bei einigen Literaturquellen, welche kein Veröffentlichungsdatum besitzen, wird die Jahreszahl durch „00“ ersetzt.
  Beispiel: [Git00]
- Bevorzugt werden deutsche Begriffe verwendet. Bei einigen Textstellen werden jedoch englische Bezeichnungen benutzt, wenn diese aus Sicht des Autors fachlich passender sind.
- Platzhalter für einen Namen werden in spitzen Klammern angegeben.
  Beispiel: <Name>

Inhaltsverzeichnis

Abkürzungsverzeichnis

A

API ..... Application Programming Interface

E

E2E-Test ..... End-To-End-Test
EA ..... Enterprise Architect
EMF ..... Eclipse Modeling Framework

G

GUI ..... Graphical User Interface
GWT ..... Google Web Toolkit

I

IDE ..... Integrated Development Environment
IM ..... Intermediate Model

M

MOF ..... Meta Object Facility
MVC ..... Model-View-Controller

O

OMG ..... Object Management Group

P

POJO ..... Plain Old Java Object

R

RAP ..... Remote Application Platform
RCP ..... Rich Client Platform
RWT ..... RAP Widget Toolkit

S

SWT ..... Standard Widget Toolkit

U

UML ..... Unified Modeling Language

X

XMI ..... XML Metadata Interchange
XML ..... Extensible Markup Language

1 Einleitung

Bei dem von FOKUS entwickelten, webbasierten Werkzeug ModelICE zur Erstellung und Modifikation von Modellen werden diverse Softwaretests durchgeführt. Unter anderem wird dabei die grafische Benutzeroberfläche getestet. Für die Realisierung derartiger Tests muss der Klienten-spezifische Bestandteil der Anwendung eine geeignete Testbarkeit vorweisen. Dieser Quelltext wird bei ModelICE mithilfe des Frameworks RAP generiert. Es offenbarten sich jedoch während der Entwicklung von ModelICE Probleme des Frameworks bezüglich der Testbarkeit. Aus diesem Grund soll untersucht werden, welche Auswirkungen die prototypische Migration von ModelICE auf die cloudbasierte Eclipse Che Plattform bezüglich der Testbarkeit und Performance hat.

1.1 Ziel der Arbeit

Aufgrund der Problematik mit RAP entstand die Motivation für diese Arbeit und der Ansatz die Software ModelICE auf die Eclipse Che Plattform zu migrieren, um durch Nutzung eines anderen Frameworks für die grafische Benutzeroberfläche (engl. "*graphical user interface*" - GUI) die Testbarkeit zu erhöhen und durch eine effizientere Kommunikation zwischen einem Klienten und dem Server eine Steigerung der Performance zu erreichen. Im Rahmen dieser Arbeit wird dazu ein Prototyp entwickelt und evaluiert.

1.2 Aufbau

Nach einer Einführung in die methodischen und technologischen Grundlagen der modellgetriebenen Softwareentwicklung wird die cloudbasierte Eclipse Che Plattform als Zielumgebung der Migration vorgestellt. Aufbauend auf einer Übersicht der aktuellen Implementierung von ModelICE werden in der Problemanalyse die Herausforderungen von ModelICE bezüglich der Testbarkeit von E2E-Tests detailliert erläutert. Dem folgt ein Kapitel über die Erstellung eines Lösungsansatzes sowie der Implementierung, bevor die Implementierung der Migration bezüglich der Testbarkeit und Performance anhand festgelegter Kriterien evaluiert und ausgewertet wird. Abschließend werden die Ergebnisse zusammengefasst und ein Ausblick auf mögliche weitere Entwicklungen gegeben.

1.3 Abgrenzung

Die prototypische Implementierung der Migration von ModelICE auf Eclipse Che umfasst nicht alle Funktionalitäten der bisherigen Implementierung. Entwickelt werden nur Minimalanforderungen, um zu analysieren, ob mit dem gewähltem Ansatz die Testbarkeit verbessert beziehungsweise die Performance von ModelICE gesteigert werden kann. Anhand der Evaluationsergebnisse des Prototyps soll beurteilt werden, ob eine vollständige Migration bezüglich der zuvor genannten Kriterien vorteilhaft ist und realisiert werden sollte.

2 Grundlagen

In diesem Kapitel werden technische Grundlagen detailliert erläutert, welche im Zusammenhang mit der prototypischen Migration stehen. Im ersten Teil werden zunächst für die Arbeit relevante Grundkonzepte der modellgetriebenen Softwareentwicklung verdeutlicht. Insbesondere werden Technologien fokussiert, die zu Beginn dieser Arbeit bereits in der Implementierung von ModelICE Anwendung finden. Der zweite Teil beschäftigt sich mit der Zielplattform Eclipse Che und deren Architektur. Der dritte Teil stellt die Software ModelICE vor, beschreibt deren Architektur und erläutert, wofür die Software eingesetzt werden kann und welche Probleme mit ihr gelöst werden können.

2.1 Modellgetriebene Softwareentwicklung

Bei der modellgetriebenen Softwareentwicklung steht das Modell selbst im Vordergrund. Mit seiner Hilfe können Anforderungen und Bestandteile einer Software auf einer höheren Abstraktionsebene analysiert und beschrieben werden. Ein „*Modell*“ dient zur Abstraktion und Abbildung eines realen Aspekts beziehungsweise eines realen Systems. Dabei basiert ein Modell auf einem „*Metamodell*“. Ein Metamodell ist ein Modell für die Beschreibung der abstrakten Notationsweise eines Modells, insbesondere mit dessen Elementen und deren Beziehungen zueinander. Ein Modell kann mithilfe einer kontextunabhängigen Sprache, wie der „*Unified Modeling Language*“ (UML) [ChSB07], sowie mit einer domänenspezifischen Sprache, wie dem „*Requirements Interchange Format*“ [Requ16], entwickelt werden. Mithilfe von „*Modelltransformationen*“ lassen sich benötigte Artefakte des Entwicklungsprozesses automatisiert erzeugen. Mittels einer Modell-zu-Modell-Transformation wird mindestens ein Zielmodell basierend auf einem oder mehreren Ausgangsmodellen generiert. Die Modell-zu-Text-Transformation stellt die Generierung von Textartefakten, wie Quelltext, Dokumentation oder ähnliches ausgehend von einem Modell zur Verfügung.

Modellgetriebene Softwareentwicklung kann mit vielen verschiedenen Werkzeugen realisiert werden. Unter anderem ist dies mit dem „*Eclipse Modeling Framework*“ (EMF) [Gron00] (siehe Abschnitt 2.1.3) möglich. Um die Zusammenhänge des Frameworks mit der modellgetriebenen Softwareentwicklung sowie der Software ModelICE zu verdeutlichen, werden nachfolgend weitere Themen detailliert erläutert.

2.1.1 Meta Object Facility

Die „*Meta Object Facility*“ (MOF) stellt ein gemeinsames und standardisiertes Metamodell aller Modellierungssprachen der „*Object Management Group*“ (OMG) [Abou00a] dar. Die OMG ist ein internationaler Zusammenschluss verschiedener Unternehmen wie beispielsweise IBM [Ibm-19] und Oracle [Orac00]. Gemeinsam werden Standards für die Erstellung von Software entwickelt. Um einen Datenaustausch zu ermöglichen, müssen die Modelle zum Beispiel mit dem ebenso von der OMG standardisierten „*XML Metadata Interchange*“ (XMI)-Format [Abou00b] serialisiert werden. XMI basiert auf der ebenfalls standardisierten „*Extensible Markup Language*“ (XML) [Exte08].

Die MOF-Spezifikation wird in zwei Teilspezifikationen differenziert. Das „*Essential MOF*“ (EMOF), beinhaltet den notwendigen Kern des MOF-Modells, um objektorientierte Modelle beschreiben zu können. Somit stehen fundamentale Modellelemente zur Verfügung, um Metamodelle zu erstellen. Im Gegensatz dazu beinhaltet „*Complete MOF*“ (CMOF) den kompletten Sprachumfang, sodass

zusätzliche Anforderungen mit weiterführenden Elementen[1] realisiert werden können. Im Rahmen dieser Arbeit ist lediglich EMOF relevant, weshalb an dieser Stelle auf die Erläuterung weiterer Details zu CMOF verzichtet wird.

2.1.2 Unified Modeling Language

Die UML wird durch ein Metamodell definiert, welche auf der MOF-Architektur basiert. Entwickelt wird die UML von der OMG und ist als Standard akzeptiert, wie das nachfolgende Zitat beschreibt:

> „Schließlich wurde 1997 die UML in der Version 1.1 bei der Object Management Group (OMG) zur Standardisierung eingereicht und akzeptiert. Seitdem erfolgt die Weiterentwicklung der UML durch die OMG. Später hat auch die ISO [International Organization for Standardization, Anm. des Verf.] die UML als Standard akzeptiert." (Quelle: [WeOe06, Kap.1.2.1 S. 5])

Die UML ermöglicht die Modellierung objektorientierter, softwarebasierter Systeme und derer Prozesse. Die Systeme und Prozesse werden mit unterschiedlichen Diagrammtypen in einem Modell beschrieben, sodass verschiedene Facetten betrachtet werden. In dieser Weise kann zum Beispiel die Struktur eines Systems mit Klassendiagrammen und sein Verhalten in verhaltensbezogenen Diagrammen, wie Zustands-, Aktivitäts- oder Sequenzdiagrammen, modelliert werden.

2.1.3 Eclipse Modeling Framework

Das Eclipse Modeling Framework ist eine Eclipse-spezifische Implementierung der beschriebenen EMOF des Abschnittes 2.1.1. EMF wird von der Eclipse Open-Source-Gemeinschaft entwickelt und ist in die Eclipse Plattform integriert [SBPM08, Kap.2 S. 11]. Die Leitung der Gemeinschaft übernimmt die Eclipse Foundation [Mili00], welche ebenfalls die modulare Eclipse Plattform zur Entwicklung von Software [DFKK05] entwickelt. Daraus ergibt sich der Vorteil, dass die Features der Eclipse Plattform selbst für die zu erstellende Anwendung benutzt werden können. Modelle werden in EMF in Form von Klassen- oder Baumdiagrammen beschrieben und als Ecore-Modelle bezeichnet. Ein Ecore-Modell kann aus verschiedenen unterstützten Eingabeformaten, wie dem XML-Schema, Modellen der UML sowie annotierten Java Schnittstellen erzeugt werden. Für die Bearbeitung eines Ecore-Modells stellt EMF generische baumbasierte und diagrammbasierte Editoren zur Verfügung. Zudem ist die Generierung modellspezifischer Editoren zur Manipulation von Modellen möglich.

[1] Unter anderem wird die Klasse „*Link*" [Unif15] hinzugefügt, welche eine Instanz einer Assoziation auf die gleiche Weise repräsentiert wie Elemente in einer Klasse repräsentiert werden.

Eine Instanz eines Ecore-Modells ist wiederum ein EMF-Modell (vgl. Abbildung 2.1) und kann mithilfe einer Modelltransformation zum Beispiel in eine XML-Datei konvertiert werden.

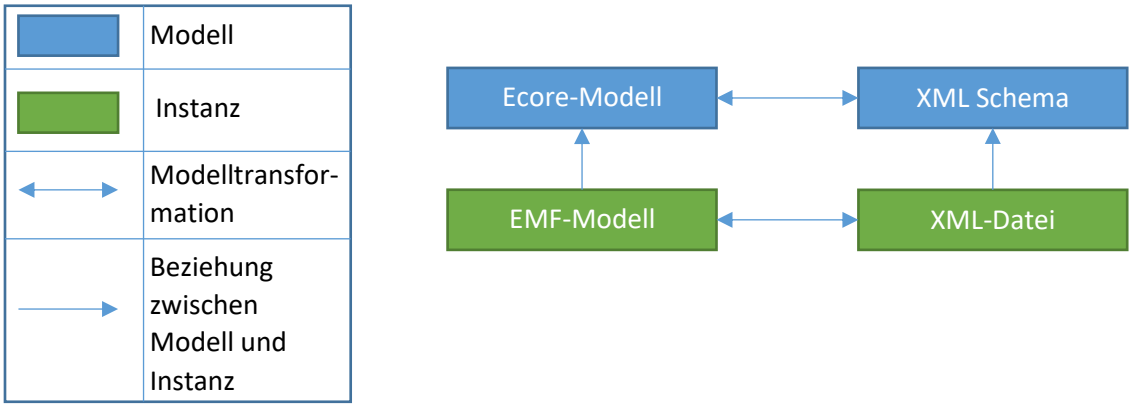


Abbildung 2.1: Beziehung zwischen Modellen und deren Instanzen

2.1.4 Model-View-Controller

Das Architekturmuster *„Model-View-Controller“* (MVC) [Mvca00] dient der Aufteilung von Zuständigkeiten innerhalb einer Software in drei Komponenten (vgl. Abbildung 2.2)[2].

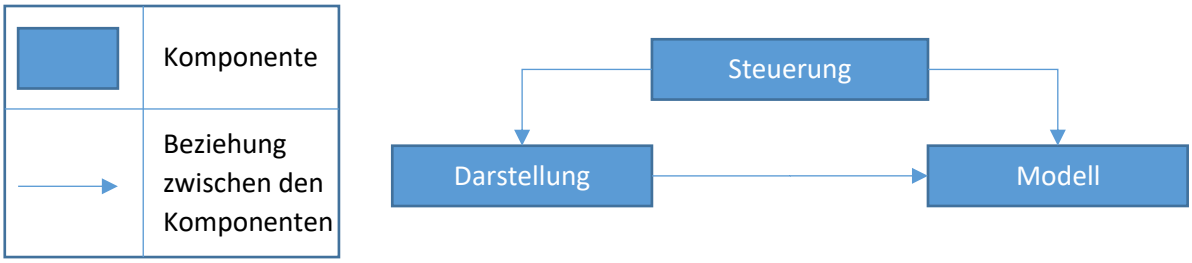


Abbildung 2.2: Schematische Darstellung von Model-View-Controller

Das *„Modell“* repräsentiert die interne Geschäftslogik mit dessen Daten, welche mithilfe der *„Darstellung“* (engl. *„View“*) präsentiert werden können. Die *„Steuerung“* (engl. *„Controller“*) reagiert auf Interaktionen des Benutzers, setzt die entsprechende Manipulation des Modells um und aktualisiert den Zustand der Darstellung. Mithilfe der Trennung zwischen den Komponenten können diese unabhängig voneinander entwickelt und getestet werden. So sind sowohl die Steuerung als auch die Darstellung abhängig vom Modell, welches selbst jedoch unabhängig von den beiden anderen Komponenten ist.

2.1.5 OSGi und Equinox

OSGi (früher eng. *„Open Services Gateway Initiative“*) ist eine Allianz, welche Spezifikationen zur Entwicklung dynamischer Java-Applikationen mithilfe eines modularen Komponentenmodells entwickelt. *„Bundles“* sind die grundlegenden OSGi-Komponenten und können zur Laufzeit nicht nur aktiviert und deaktiviert, sondern auch hinzugefügt, entfernt und aktualisiert werden. Die Verwaltung

[2] Das Muster existiert in zahlreichen Variationen. Hier wird die ursprüngliche Basisform nach Smalltalk-80 [Mvc00] dargestellt.

der Bundles wird durch eine zentrale Registratur[3] ermöglicht. Vorausgesetzt wird eine Java Laufzeitumgebung, welche mittels einer „Java virtuellen Maschine“ zur Verfügung steht.

Um die OSGi-Spezifikationen [Osgi18] praktisch einzusetzen, müssen diese zunächst implementiert werden. Zwar stellt OSGi eine Referenzimplementierung zur Verfügung, welche jedoch nicht für den Einsatz in einer Produktivumgebung vorgesehen ist. Stattdessen dient die gestellte Implementierung als Vorlage für Hersteller, die dieses Konzept mit individuellem Schwerpunkt implementieren wollen. So existieren sowohl kommerzielle als auch nicht-kommerzielle Realisierungen für mobile und eingebettete Systeme, wie beispielsweise Conciergo[4] [Mave00a]. Für Eclipse wurde ebenfalls eine OSGi-Variante namens Equinox [Equi00] entwickelt. Der Fokus der Equinox-Implementierung liegt auf der Nutzung von OSGi-Spezifikationen innerhalb der Eclipse Plattform sowie auf zusätzlichen Funktionalitäten, wie zum Beispiel einer Registratur[5] für „*Plug-Ins*“ [Plug00], welche ähnlich der OSGi-Bundles die kleinsten installierbaren Komponenten der Eclipse Plattform sind. Aus diesem Grund werden die Begriffe Bundle und Plug-In als Synonyme verwendet. Diese Registratur sowie die Plug-In-basierte Architektur von Eclipse ermöglichen die Erstellung von entwicklungs- oder projektspezifischen Erweiterungen, welche nahtlos in die integrierte Entwicklungsumgebung (engl. „*integrated development environment*„ – IDE) von Eclipse eingefügt werden können. Ist eine Erweiterung registriert und aktiviert, steht diese innerhalb einer Eclipse-Anwendung zur Verfügung und kann neue Funktionalitäten hinzufügen oder bestehende modifizieren. Ein Plug-In lässt sich in zwei Arten differenzieren: „*Erweiterungen*“ und „*Erweiterungspunkte*“ (vgl. Abbildung 2.3).

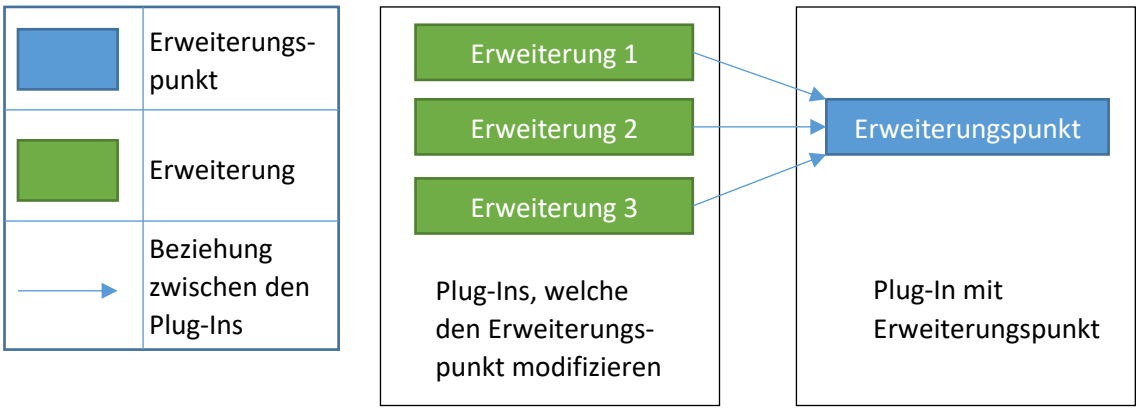


Abbildung 2.3: Zusammenhang der Erweiterungen mit einem Erweiterunspunkt

Das Prinzip lässt sich mit einer elektrischen Steckdose und einem passenden Stromstecker vergleichen. Die Steckdose selbst ist ein Erweiterungspunkt und stellt eine Schnittstelle (engl. „*application programming interface*“ - API) zur Verfügung, an der ein Stecker angedockt werden kann. Gleiches gilt für einen Erweiterungspunkt und eine Erweiterung: Der Erweiterungspunkt stellt eine Schnittstelle zur Verfügung, die von einer oder mehreren Erweiterungen implementiert werden kann, um neue Funktionalitäten hinzuzufügen oder bestehende zu modifizieren.

[3] Häufig wird für die Registrierung von Services der englische Begriff „*Service-Registry*“ verwendet.
[4] Dieses Framework umfasst laut Angaben der Entwickler eine Speichergröße von nur 80kB und ist daher für Systeme mit geringem Speicherplatz geeignet und wird im Rahmen der Arbeit nicht weiter erläutert.
[5] Der englische Begriff „*Extension-Registry*“ ist der konventionelle Name für die Registratur von Plug-Ins.

2.1.6 Eclipse

Ursprünglich stand die Entwicklung Java-basierter Anwendungen im Mittelpunkt der Eclipse IDE. Durch kontinuierliche Entwicklung der Software unterstützt Eclipse unter anderem zahlreiche andere Modellierungs- und Programmiersprachen, wie C++, Python und viele weitere [Ecli00a]. Realisiert wird dies durch Java-basierte Eclipse-Erweiterungen, welche mithilfe der beschriebenen OSGi-Implementierung Equinox in Form von Plug-Ins hinzugefügt werden. Diese nicht-kommerziellen oder kostenpflichtigen Erweiterungen können sprachspezifische Funktionalitäten, aber auch Werkzeuge für den Entwicklungsprozess zur Verfügung stellen. Zum Beispiel ermöglicht die Einbindung der Software Git [Git00] mittels der EGit Plug-Ins [Egit00] die integrierte Verwaltung und Versionierung von Entwicklungsartefakten, wie Quelltext.

Um die in Projekten zusammengefassten Entwicklungsartefakte und projektspezifische Konfigurationen mit der Eclipse IDE zu erstellen und zu modifizieren wird ein „*Workspace*“ verwendet. In diesem können beliebig viele Projekte enthalten sein. Jedes Projekt muss dabei einen Projekttyp[6] und einen eindeutigen Namen besitzen. Je nach gewähltem Projekttyp werden typspezifische Plug-Ins aktiviert, sodass die IDE kontextspezifisch angepasst wird. Unter anderem kann dadurch eine Autovervollständigung des Quelltextes ermöglicht werden. Außerdem ist jeder Workspace von allen anderen isoliert und unabhängig, sodass Erweiterungen, Werkzeuge und die benötigte Laufzeitumgebung auf dem aktuellen Computer manuell und separat installiert werden müssen.

2.1.7 Eclipse Rich Client Platform

Mithilfe der „*Rich Client Platform*“ (RCP) [Rich00] können Rich Client-Anwendungen entwickelt werden. Die entstehenden Desktop-Applikationen können die Client-Server-Architektur „Rich Client“ verwenden, wodurch die Konzepte eines „*Thin Clients*“ mit denen eines „*Fat Clients*“ vereint eingesetzt werden. Differenzieren lassen sich die beiden Konzepte anhand der Bereitstellung der applikationsspezifischen Funktionalitäten. Bei dem Thin Client stellt ein Server die Funktionalitäten zur Verfügung, während dies beim Fat Client durch den Klienten selbst realisiert wird. Bei einem Rich Client Ansatz werden dementsprechend Funktionalitäten sowohl vom Server als auch vom Klienten selbst zur Verfügung gestellt. Mithilfe der zugrundeliegenden Eclipse Plattform ist die entstehende Software dadurch vordergründig in der Programmiersprache Java implementiert und benötigt mindestens die Komponenten „Eclipse Core Platform“, welche den Lebenszyklus der Anwendung steuert sowie das „*Standard Widget Toolkit*“ (SWT) [Guin00] und „*JFace*“ [Jfac00] für die Erstellung grafischer Oberflächen. Dabei stellt SWT elementare Interaktionselemente, wie beispielsweise Textfelder oder Schaltflächen zur Verfügung, welche von JFace für die Erstellung komplexer Oberflächenkomponenten, wie den „Viewern“, verwendet werden. Viewer vereinfachen die Interaktion zwischen einer zugrundeliegenden Datenstruktur und einem SWT-Widget, wie einer Baumstruktur oder einer Tabelle, indem ein Viewer[7] ein derartiges SWT-Widget um Funktionalitäten erweitert, wie zum Beispiel das Hinzufügen, Entfernen oder Sortieren von Elementen. Darüber hinaus unterstützt ein Viewer das in Abschnitt 2.1.4 beschriebene Architekturmuster MVC. Die Repräsentation des Datenmodells wird von dem zugrundeliegenden SWT-Widget dargestellt und kann mit einem benötigten `LabelProvider` modifiziert werden. Zusätzlich ist ein `ContentProvider`

[6] Der Projekttyp und dessen Verhalten werden mithilfe der „*Project natures*“ [Proj18] beschrieben.
[7] Es existieren verschiedene Viewer, wie der „TreeViewer“ oder „TableViewer“. Es können zudem auch eigene Viewer implementiert werden.

erforderlich, um Zugriff auf ein Datenmodell zu besitzen. Der Viewer selbst stellt den Controller dar und sorgt für eine Aktualisierung der Darstellung bei Modifikationen des Modells.

Die Eclipse IDE stellt kontext- bzw. sprachabhängig unterschiedliche Anordnungen von Views, die sogenannten Perspektiven, bereit. In Abbildung 2.4 wird die Aufteilung der Standardperspektive für die Bearbeitung von Java-Quelltext dargestellt.

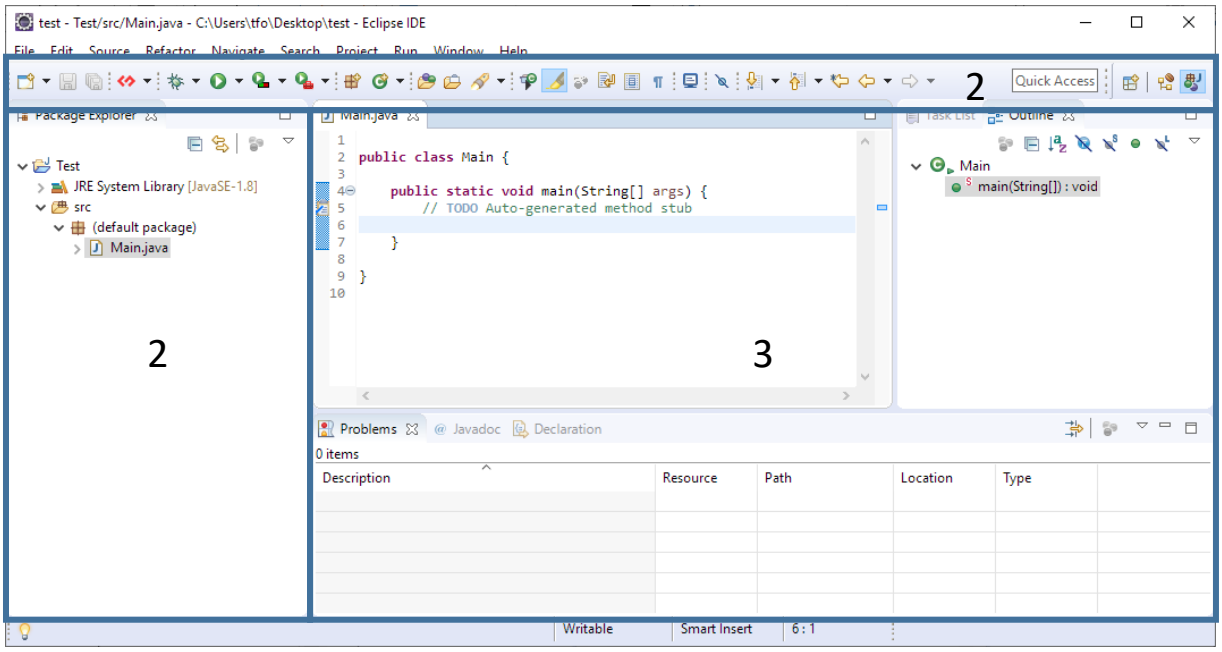


Abbildung 2.4: Aufteilung der Standardansicht der Java Perspektive der Eclipse Java Development Tools

Unterhalb der Menü-Leiste (1) befindet sich die vierteilige Eclipse „*Workbench*“. Alle Projekte mit deren Dateien werden im „*Workspace*“[8] (2) angezeigt und können mithilfe eines Editors (3) oder einer View bearbeitet werden. Weitere Informationen zu einer geöffneten Datei werden in dem Bereich (4) angezeigt. In Bereich (5) werden Informationen, wie Konsolenausgaben oder Kompilierungsfehler angezeigt. Zudem stehen bekannte Bedienkonzepte zur Verfügung, wie die Reiterstruktur bei geöffneten Editoren.

2.1.8 Remote Application Platform

RCP-Anwendungen können nur als Desktop-Version betrieben werden. Das Framework „*Remote Application Platform*“ (RAP) hingegen ermöglicht die Ausführung der Software auf entfernten Klienten, wie Internetbrowsern[9] oder Thin Clients. Hierzu wird die RAP-Anwendung in Java entwickelt und die relevanten Teile der klientseitigen Implementierung nach HTML, CSS und JavaScript umgewandelt. In Abbildung 2.5 werden die architektonischen Unterschiede zwischen RCP und RAP verdeutlicht.

[8] Aufgrund der fachlich passenderen Ausdrucksweise des englischen Begriffs wird dieser verwendet.
[9] Die Architektur einer auszuführenden Anwendung im Browser ist ausschlaggebend dafür, ob dieser ein Thin oder Fat Client ist.

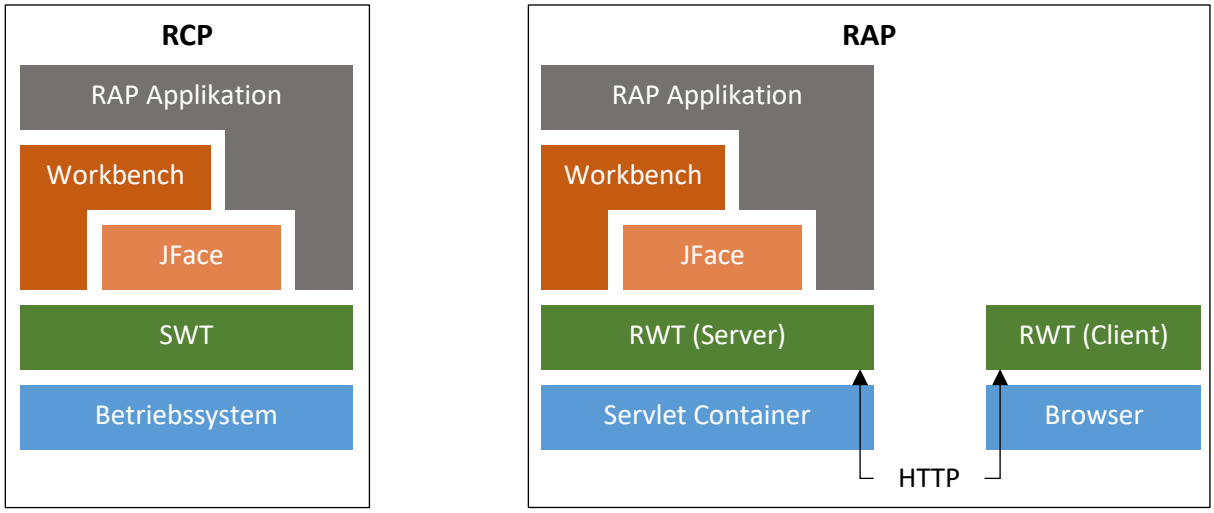


Abbildung 2.5: Unterschied der Architektur von RCP und RAP [Ou10]

Während RCP direkt auf dem Betriebssystem eines Klienten ausgeführt wird, nutzt RAP einen Servlet [ChBu17] Container[10] auf einem Server. Auf dem Klienten wird für die Erstellung der grafischen Benutzeroberfläche das *„RAP Widget Toolkit“* (RWT) [Rap/00a] verwendet. Wenn ein Internetbrowser auf die Anwendung zugreift, wird diese mithilfe von RAP zur Verfügung gestellt. Im Gegensatz zu SWT ist RWT hinsichtlich der bereitgestellten Funktionalitäten jedoch eingeschränkt[11]. Da RAP ein alternatives Framework für die Benutzeroberfläche zu SWT für RCP ist, können die Kernfunktionalitäten der Eclipse Plattform auch in dieser Architektur verwendet werden.

2.2 Die cloudbasierte Eclipse Che Plattform

Im Folgenden wird die cloudbasierte Eclipse Che Plattform mithilfe notwendiger Grundlagen vorgestellt.

2.2.1 Docker

Für das Verständnis von Eclipse Che ist eine grundlegende Einführung in Docker [Dock00] erforderlich. Die quelloffene Software Docker stellt eine Separierung von Anwendungen mithilfe der Container-Virtualisierung bereit. Ein *„Container“* [What00] ist eine Instanz einer gebündelten Einheit von Software mit allen benötigten Software-Abhängigkeiten, Laufzeitumgebungen und Konfigurationen. Die einzelnen Bestandteile eines Containers werden in einem textbasierten Dockerfile beschrieben, woraus anschließend ein Docker Image erstellt werden kann. Ein Docker Image dient als Basis für die Erzeugung beliebig vieler Container und enthält kein Betriebssystem, sondern wird mithilfe eines Betriebssystems ausgeführt. Somit kann ein Container als leichtgewichtig gegenüber einer virtuellen Maschine betrachtet werden. Zudem ist der Aufwand für die Übertragung eines Containers auf einen anderen Host auf einen Kopiervorgang des Docker Images reduziert[12]. Des Weiteren kann mithilfe horizontaler und vertikaler Skalierung (vgl. Abbildung 2.6) das vorhandene Leistungspotential der

[10] Ein Servlet ist ein Java-Programm, welches auf einem Java-Server ausgeführt wird und Anfragen von Klienten bearbeitet.
[11] Die detaillierten Unterscheidungen können unter folgendem Link eingesehen werden: https://www.eclipse.org/rap/developers-guide/devguide.php?topic=rwt.html
[12] Auf beiden Computern muss als Voraussetzung Docker installiert sein.

Anwendung bei unzureichenden Ressourcen erhöht beziehungsweise bei überschüssigen Ressourcen verringert werden (vgl. Abbildung 2.6).

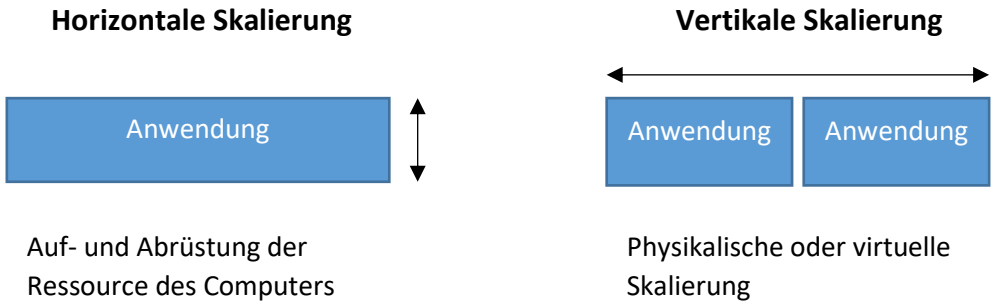


Abbildung 2.6: Horizontale im Vergleich zur vertikalen Skalierung

Bei der vertikalen Skalierung wird die Leistung eines einzelnen Servers verändert, indem die vorhandenen Ressourcen, wie der Prozessor oder Arbeitsspeicher, auf- oder abgerüstet werden. Die horizontale Skalierung modifiziert die Leistung, indem weitere Instanzen der Anwendung beziehungsweise Server physikalisch oder virtuell hinzugefügt oder entfernt werden.

2.2.2 Eclipse Che

Eclipse Che [Ecli00b] ist eine cloudbasierte IDE, welche innovative Möglichkeiten, wie die kollaborative Nutzung von Projekten beziehungsweise Softwareartefakten, für den Einsatz einer zu entwickelnden Software eröffnet. Des Weiteren ist Eclipse Che, wie die vorherigen RCP-basierten Eclipse-Versionen, eine Plattform, auf deren Basis neue Software entwickelt werden kann. In Abbildung 2.7 wird die Ähnlichkeit der Standardperspektive der Eclipse Che IDE zu einer RCP-Anwendung verdeutlicht. Zudem wird das quelloffene Projekt Eclipse Che mithilfe der Programmiersprache Java entwickelt.

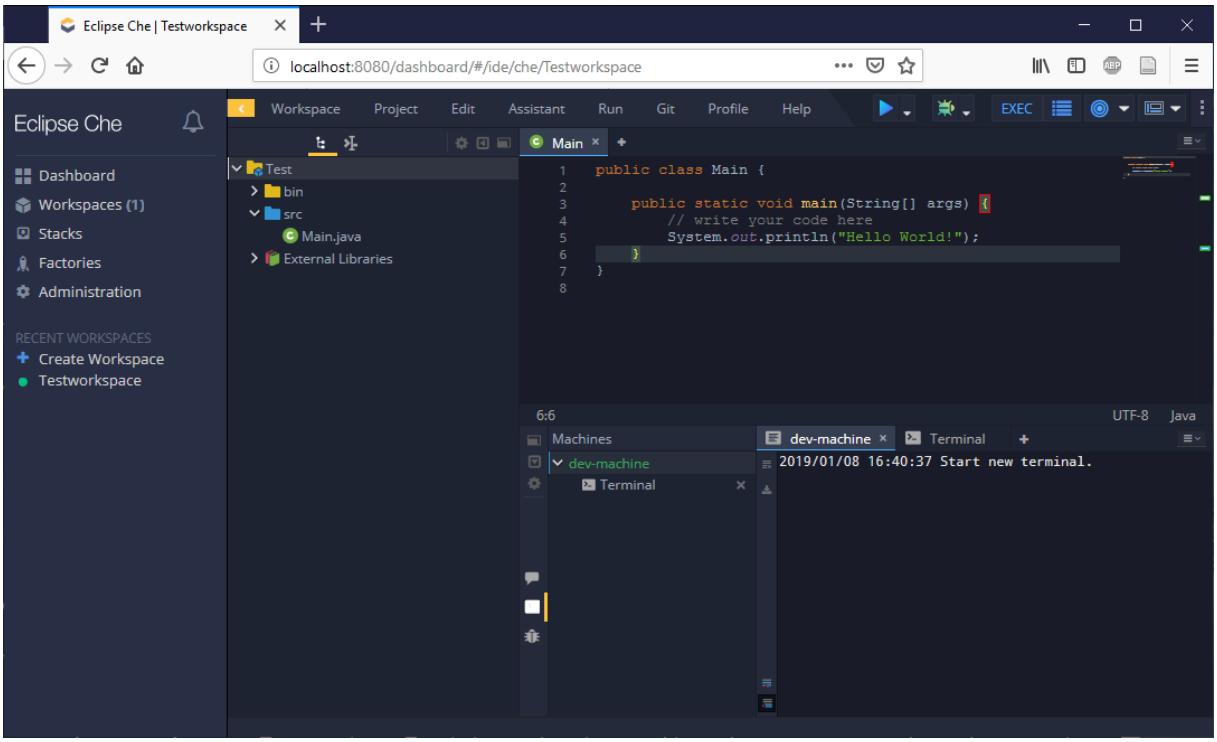


Abbildung 2.7: Darstellung eines Workspaces mit Eclipse Che

Die Eclipse Che Architektur

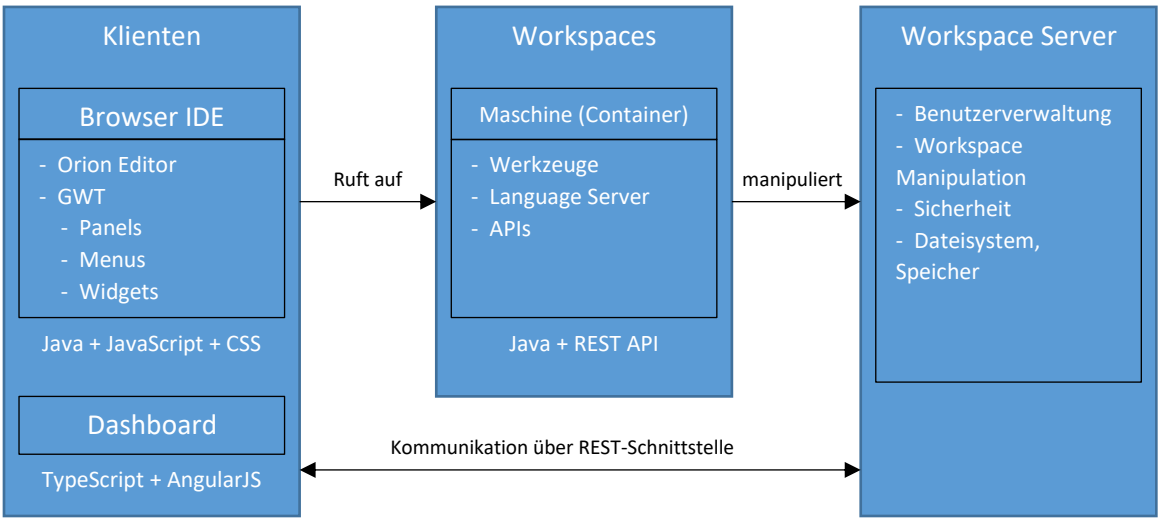


Abbildung 2.8: Überblick über die Zusammenhänge der Eclipse Che Architektur

Das Konzept eines Workspaces (vgl. Abbildung 2.8) existiert nicht nur bei der Eclipse IDE, sondern auch bei Eclipse Che. Ein Workspace bei Eclipse Che besteht, wie bei der Eclipse IDE, aus der IDE selbst, den Projekten sowie projektspezifischen Konfigurationen. Während bei der Eclipse IDE die benötigten Laufzeitumgebungen, Erweiterungen und Werkzeuge manuell und separat installiert werden müssen, sind derartige Installationen bei Eclipse Che nicht nötig. Ein Eclipse Che Workspace kann auch mehrere Laufzeitumgebungen beinhalten. Realisiert wird dies mithilfe der in Abschnitt 2.2.1 erwähnten Docker Container.

Die Projekte eines Workspaces können mit einer Browser IDE erstellt und mittels RESTful[13] Web Services manipuliert werden. Aufgrund der ressourcenschonenden[14] Benutzung des Arbeitsspeichers [Ecli00c] auf dem Klienten kann die IDE nicht nur mit einem Internetbrowser aufgerufen, sondern auch von weiteren Klienten beziehungsweise Thin Clients verwendet werden.

Wenn der Eclipse Che Server einen neuen Workspace erstellt, wird dieser zusammen mit einem eigenen Workspace Server gestartet, sodass die überwiegende Rechenarbeit von dem Workspace Server anstelle des Che Servers ausgeführt wird, wodurch eine geringe Belastung auf dem zentralen Che Server der Installation entsteht. Über eine REST-basierte Programmierschnittstelle wird eine Kommunikation zwischen dem Che Server und dem Workspace Server ermöglicht. Zudem kann ein Workspace mit einer eindeutigen URL von mehreren Benutzern simultan verwendet werden.

[13] https://www.eclipse.org/che/features/

[14] Laut Angaben des Herstellerst werden ungefähr 100 MB des Arbeitsspeichers pro Reiter eines Internetbrowsers des Klienten verwendet.

Die Architektur von Eclipse Che besteht aus drei Komponenten:

- **Workspace Server**
    - Auf einem Applikationsserver wie Tomcat [Apac00a] läuft beispielsweise der Eclipse Che Server. Neben der Erstellung von Workspaces oder der Verwaltung von Benutzern ist der Che Server für weitere Aufgaben (vgl. Abbildung 2.8) zuständig.
- **Workspace**
    - Ein Workspace ist eine Gruppierung von zusammenhängenden Projekten mit projektspezifischen Konfigurationen. Jeder Workspace ist isoliert und unabhängig von anderen Workspaces. Dabei kann jeder von diesen verschiedene Laufzeitumgebungen in Form von Docker Container besitzen.
- **Klient**
    - Aus Sicht der Che Architektur ist der Klient entweder die Browser IDE oder eine andere Desktop IDE, welche mithilfe einer SSH-Verbindung angebunden ist. Mithilfe der REST-Schnittstelle werden beispielsweise veränderte Konfigurationen oder Manipulationen an Dateien an den Workspace Server übertragen.

Eclipse Che setzt Maven-Assemblies [Apac00] (siehe Abschnitt 2.2.4) ein, um benötigte Build-Artefakte zu erzeugen. Nachfolgend ist eine Übersicht der Eclipse Che Projekte dargestellt mit einer dazugehörigen Erklärung, welche Artefakte sich in den jeweiligen Assemblies befinden.

- **assembly-ide-war**
    - Beinhaltet Plug-Ins der Browser IDE.
- **assembly-wsagent-war**
    - Beinhaltet Plug-Ins, welche auf dem Workspace-Server ausgeführt werden.
- **assembly-wsagent-server**
    - Paketiert den Workspace Agent in einen Tomcat Webserver.
- **assembly-wsmaster-war**
    - Beinhaltet die Kernbibliotheken der Che Plattform, wie beispielsweise die Schnittstelle für die Kommunikation mit einem Workspace.
- **assembly-main**
    - Bündelt alle Che Module zu einem eigenständigen Server oder einer installierbaren Desktop-Applikation.

Weiterhin bietet Eclipse Che die Möglichkeit, Workspaces mithilfe von Erweiterungen zu individualisieren. Die Erweiterungen können mit der Browser IDE selbst oder mit einer beliebigen Desktop IDE, wie Eclipse oder IntelliJ [Inte00] entwickelt werden. Zudem ist eine Erweiterung ein Maven-Modul, welches aus beliebig vielen Projekten bestehen kann. In Tabelle 2.1 sind Konventionen für die Benennung der Projekte zu den einzelnen Architekturkomponenten in Che aufgelistet.

Präfix	Endung	Verwendung
<Name des Plug-Ins>	`-ide`	Klienten-spezifische Implementierung
	`-server`	Server-spezifische Implementierung
	`-shared`	Gemeinsam verwendete Artefakte von Klient und Server

Tabelle 2.1: Namenskonventionen eines Plug-Ins

Insgesamt existieren drei verschiedene Erweiterungstypen:

- **IDE Erweiterung**
    - Diese Erweiterungen modifizieren die grafische Benutzeroberfläche. Es können bestehende Ansichten verändert oder neue hinzugefügt werden. Der in Java entwickelte Quelltext wird mithilfe des Google Web Toolkit (siehe Abschnitt 2.2.3) in eine JavaScript-basierte Web-Applikation übersetzt und als WAR-Datei[15] auf dem Che Server gehostet.
- **Che Server Erweiterung (Workspace Master)**
    - Erweiterungen des Che Servers fügen neue Funktionalitäten hinzu oder manipulieren bestehende Funktionalitäten der Kernbibliotheken. Dabei beeinflussen diese Erweiterungen zum Beispiel die Verwaltung der Workspaces. Gebündelt werden die Java-basierten Erweiterungen in JAR-Dateien[16], welche auf dem Eclipse Che Server ausgeführt werden.
- **Workspace Erweiterung (Workspace Agent)**

    Mithilfe dieser Erweiterung können projektspezifische Plug-Ins hinzugefügt oder modifiziert werden. Durch den Zugriff auf die Projektdateien können Funktionalitäten, wie die Autovervollständigung des Quelltextes oder die Erstellung von Quelltext-Vorlagen, ermöglicht werden. Die Erweiterungen mit optional verwendeten Che Kernbibliotheken werden auf dem Workspace Server ausgeführt. Damit Prozesse, wie das Bauen und Starten einer Software, bewerkstelligt werden können, muss eine Ausführungsumgebung vorhanden sein. Für die lokale Nutzung kann der aktuelle Computer benutzt werden. Soll der Workspace Server cloudbasiert verwendet werden, muss eine Ausführungsumgebung auf dem Server installiert werden. Um sicherzustellen, dass die Umgebungen isoliert und skalierbar sind, werden zum Beispiel Docker-Container eingesetzt. Welche Software die einzelnen Docker-Container beinhalten, kann individualisiert werden. Eclipse Che stellt bereits Docker-Container zur Verfügung, welche für verschiedene Projekttypen angepasst wurden.

[15] WAR bedeutet „*Web Application Archive*“ und ist ein komprimiertes Archiv, welches Dateien mit einer vorgegebenen Struktur enthält. Diese erfüllen die Java-Servlet-Spezifikation.
[16] Eine JAR-Datei ist ein komprimiertes Archiv von Java-Klassen mit zusätzlichen Meta-Eigenschaften.

Für die Erstellung eines Java-Projektes werden etwa das Java Development Kit, die Versionsverwaltungssoftware Git sowie das Build-Verwaltungswerkzeug Maven (siehe 2.2.4) vorinstalliert. Zusätzlich besteht die Möglichkeit eigene Software-Zusammenstellungen mittels spezifischer Docker-Dateien[17] zu erstellen.

2.2.3 Google Web Toolkit

Das Google Web Toolkit (GWT) [Gwtp00] ist ein Framework für die Entwicklung von Webanwendungen. GWT Programme werden in der Programmiersprache Java geschrieben und anschließend werden clientspezifische Artefakte mithilfe eines Compilers nach HTML, CSS und JavaScript übersetzt. Mittels einer Schnittstelle[18] kann im Java-Quelltext ebenfalls JavaScript ausgeführt werden.

Mithilfe von „*Remote Procedure Calls*“ [Eule05] wird die Kommunikation zwischen Klienten und Server ermöglicht. Nach dem Senden asynchroner Anfragen und deren Bearbeitung durch den Server wird ein Ergebnis an den Klienten zurückgeschickt. Für die Übermittlung von Daten zwischen Klient und Server können GWT-spezifische Klassen, welche die Serialisierung und Deserialisierung von Java-Objekten unterstützen, verwendet werden, sodass ein Austausch vollständiger Java-Objekte ermöglicht werden kann. Zudem gestattet GWT die JSON-basierte [Ecma17] Übertragung von Informationen.

Das Framework GWT wird von Eclipse Che für die Erstellung der grafischen Benutzeroberfläche verwendet. Es können neue Ansichten erstellt oder bestehende modifiziert werden. Dabei werden Interaktionselemente, wie Text- oder Auswahlfelder, aber auch Layouts, wie die horizontale Organisation von Oberflächenkomponenten[19], eingesetzt.

2.2.4 Maven

Ein „*Build*“ ist ein automatisierter Prozess, um eine lauffähige Software inklusive aller benötigten Ressourcen zu erzeugen sowie die Bezeichnung für das Resultat des Build-Prozesses selbst [Augs18]. Maven ist ein Java basiertes und kostenfreies Build-Verwaltungswerkzeug, welches von der Apache Software Foundation [Thea00] entwickelt wird.
Bei Einsatz des Werkzeuges wird die Durchführung eines komplexen Build-Prozesses für den Benutzer vereinfacht, indem eine auf dem Software-Paradigma „*Konventionen vor Konfigurationen*“ [Bach09, Kap.1.2.1 S. 16f.] basierende Vorlage eines Lebenszyklus mit voreingestellten Standardkonfigurationen verwendet wird. Diese Einstellungen können vom Benutzer projektspezifisch angepasst werden. Der Lebenszyklus besteht aus einzelnen anpassbaren Phasen (vgl. Tabelle 2.2), welche sich aus Zielen zusammensetzen, wie beispielsweise der Kompilierung des Quelltextes. Zudem werden bei Start des Maven-Prozesses alle benötigten Abhängigkeiten transitiv aufgelöst, sodass diese nicht mehrfach und nur bei Bedarf in das lokale Maven-Repository heruntergeladen und automatisch installiert beziehungsweise bei Zustimmung aktualisiert werden.

[17] Mit dem Dockerfile und dem Composefile können die Zusammenstellungen der Software erstellt werden.
[18] Die Schnittstelle sollte laut der Dokumentation [Gwtd00] von GWT sparsam eingesetzt werden, da der kompilierte Code dieser Schnittstelle potentiell weniger portabel in Browsern ist. (http://www.gwtproject.org/doc/latest/DevGuideCodingBasicsJSNI.html)
[19] Als „HorizontalPane“ wird in GWT ein horizontal organisiertes Layout bezeichnet.

Phase	Beschreibung
validate	Das Projekt wird validiert und alle benötigten Informationen sind angegeben.
compile	Der Quelltext wird kompiliert.
test	Vorhandene Tests werden mithilfe eines Testframeworks ausgeführt.
package	Der kompilierte Quelltext wird in das vorgesehene Format komprimiert.
verify	Integritätstests werden ausgeführt.
install	Das erstellte Paket wird lokal installiert und kann von anderen Projekten benutzt werden.
deploy	In diesem Schritt wird das finale Paket in die Ausführungsumgebung, wie zum Beispiel einen Webserver, integriert.

Tabelle 2.2: Übersicht der Phasen des Maven-Lebenszyklus

Jedes Modul eines Maven-Projektes besitzt ein obligatorisches Konfigurationsmodell `pom.xml` (Project Object Model), welches auch „*Projektmodell*“ genannt wird. Durch die Hierarchiestruktur eines Maven-Projektes werden Konfigurationen übergeordneter Projekte an untergeordnete Projekte übertragen. In dem Konfigurationsmodell werden zum Beispiel Projektinformationen, Abhängigkeiten zu weiteren Bibliotheken und zusätzliche Plug-Ins deklariert. Zu den Projektinformationen gehört eine eindeutige Identifikation, welche sich aus einer Gruppierungsbezeichnung, einer Artefaktbezeichnung und einer Versionsnummer zusammensetzt. Maven stellt einen zentralen Ablageort für veröffentlichte Maven-Projekte zur Verfügung [Mave00b]. Erfordert ein Projekt eine Software-Bibliothek, wird in der `pom.xml` die Identifikation des korrespondierenden Maven Moduls als Abhängigkeit angegeben.

2.3 Die Software ModelICE

ModelICE[20] (vgl. Abbildung 2.9) ist ein flexibles Werkzeug für die cloudbasierte Modellierung von Software [InFr00], welche sowohl die Integration von bestehenden Anwendungen ermöglicht, wie beispielsweise das UML Werkzeug „*Enterprise Architekt*“ [Spar00] (EA), als auch in bestehende Anwendungen integriert werden kann.

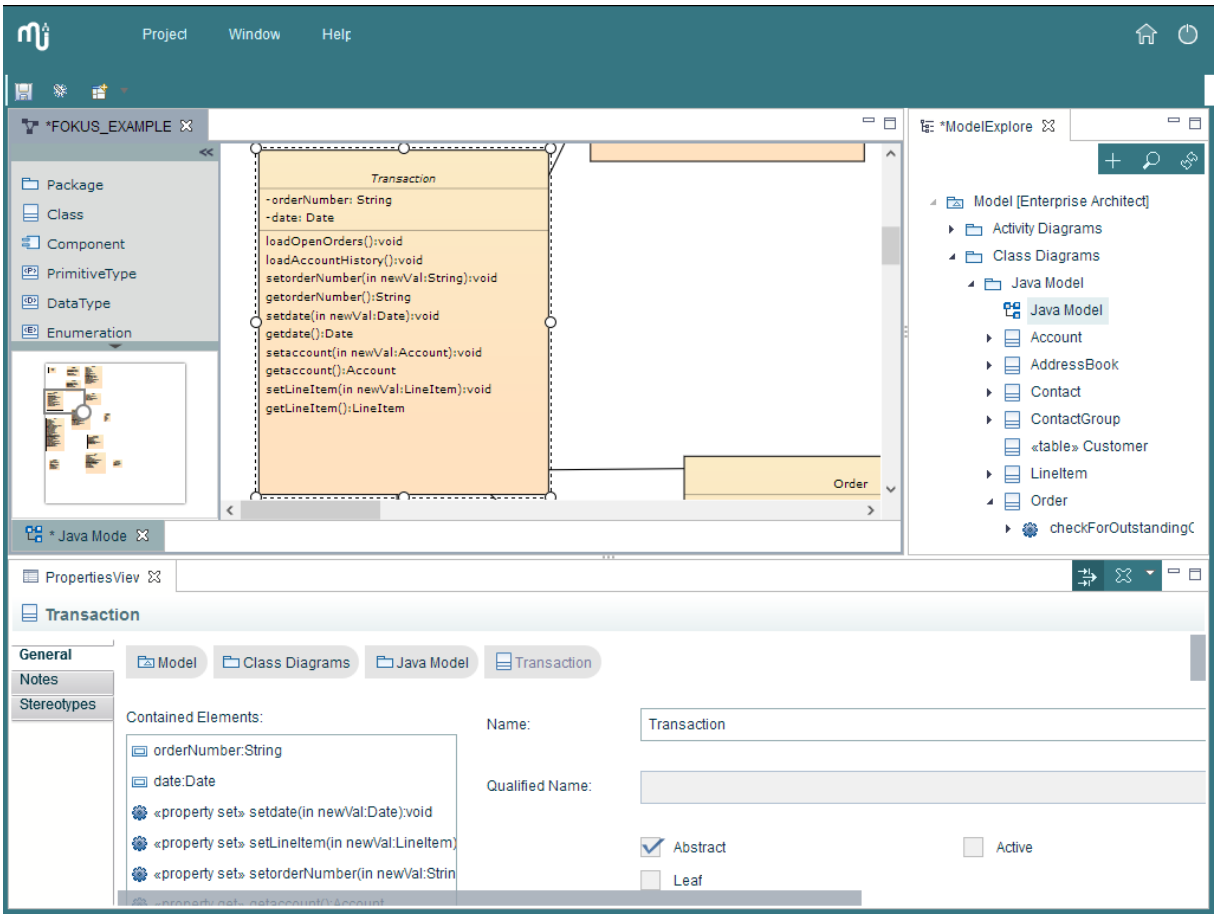


Abbildung 2.9: Beispielhafte Darstellung eines Klassendiagramms mithilfe von ModelICE

Mithilfe dieser beiden Nutzeffekte ermöglicht ModelICE die effiziente und integrierte Realisierung individueller Software-Entwicklungsprozesse. Da ModelICE eine Eclipse RAP Anwendung ist, kann ModelICE auch als Eclipse RCP Anwendung gestartet werden. Durch zusätzliche Einhaltung und Umsetzung von Cloud-Prinzipien kann ModelICE jedoch nicht nur mit einem Internetbrowser auf einem Desktop PC verwendet werden, sondern auch mit weiteren Klienten. Zudem lässt sich ModelICE ohne Installation direkt nutzen, sodass der administrative Aufwand für den Benutzer entfällt.

Da ModelICE wie beschrieben eine RAP-Anwendung ist, können die RCP-spezifischen Funktionalitäten verwendet werden, wie beispielsweise die in Abschnitt 2.1.7 beschriebene visuelle Anordnung der Komponenten in einer Perspektive. In Abbildung 2.10 sind relevante Kernkomponenten dargestellt und werden nachfolgend erläutert.

[20] Der Name ModelICE beinhaltet ein Wortspiel mit dem englischen Verb „to modelize“. Das Akronym ICE (engl.: „*Integrated Cloud Environment*“) verdeutlicht zudem, dass ModelICE eine cloudbasierte Anwendung ist.

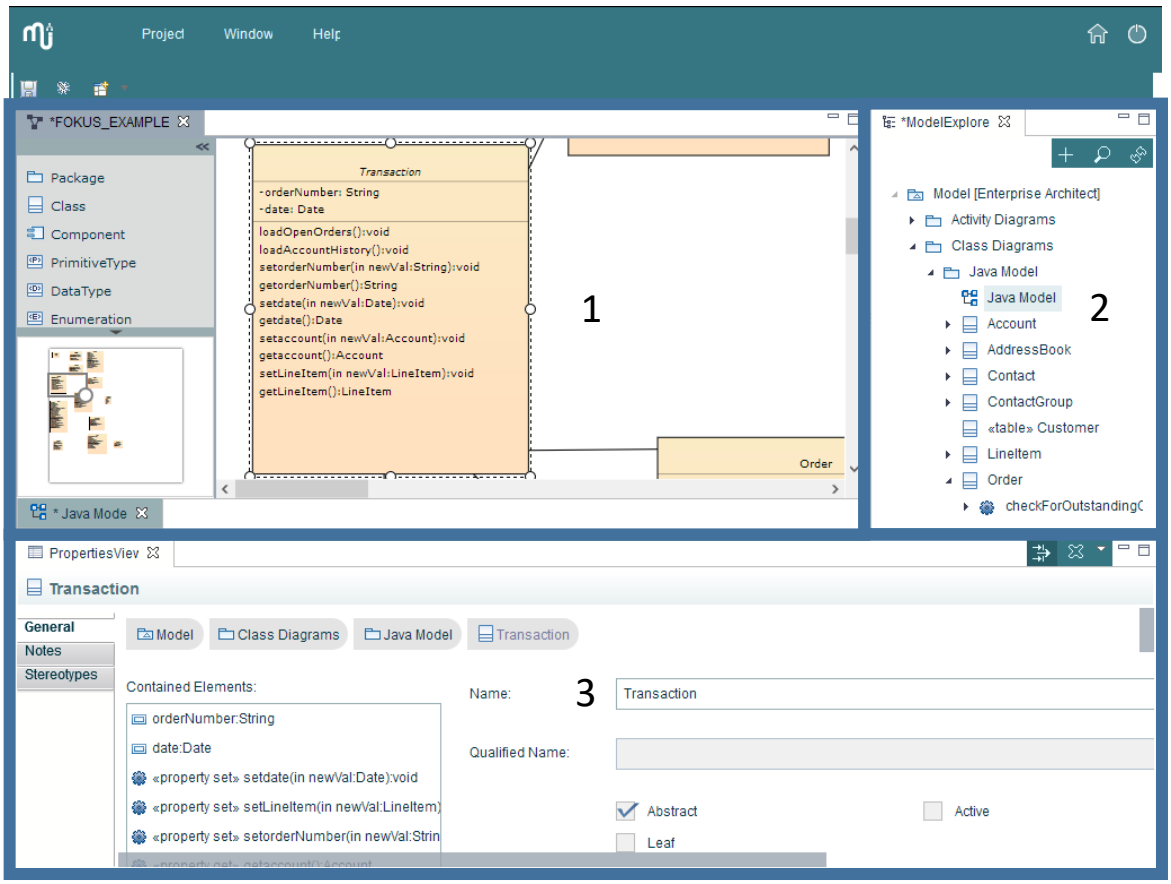


Abbildung 2.10: Markierung der einzelnen ModelICE Komponenten in einer Perspektive

Die Erstellung und Modifikation eines Modells kann diagrammbasiert mithilfe des ModelICE „*ModelEditors*“ (1) und baumbasiert unter Einsatz des „*ModelExplorers*“ (2) durchgeführt werden. Der „*Properties View*“ (3) kann für die Modifikation von Eigenschaften eines aktuell ausgewählten Elementes verwendet werden. Alle drei Komponenten besitzen die Eigenschaft ihre Darstellung eines Elementes an ein vom Benutzer selektiertes Element anzupassen. Nachfolgend werden weitere Funktionalitäten der drei Komponenten erläutert.

2.3.1 ModelEditor

Der nativ in JavaScript und auf Basis von „*Scalable Vector Graphics*“ [Scal01] entwickelte ModelEditor unterstützt die diagrammbasierte Bearbeitung von aktuell folgenden UML-Diagrammarten:

1. Klassendiagramm - „*Class Diagram*“
2. Anwendungsfalldiagramm - „Use Case Diagram“
3. Paketdiagramm - „*Package Diagram*“
4. Aktivitätsdiagramm - „*Activity Diagram*“
5. Zustandsdiagramm - „State *Machine Diagram*“
6. Komponentendiagramm - „*Component Diagram*“
7. Sequenzdiagramm - „*Sequence Diagram*“
8. Verteilungsdiagram - „*Deployment Diagram*“

Die einzelnen Bestandteile des ModelEditors sind in Abbildung 2.11 exemplarisch an einem Klassendiagramm farblich hervorgehoben und werden nachfolgend beschrieben.

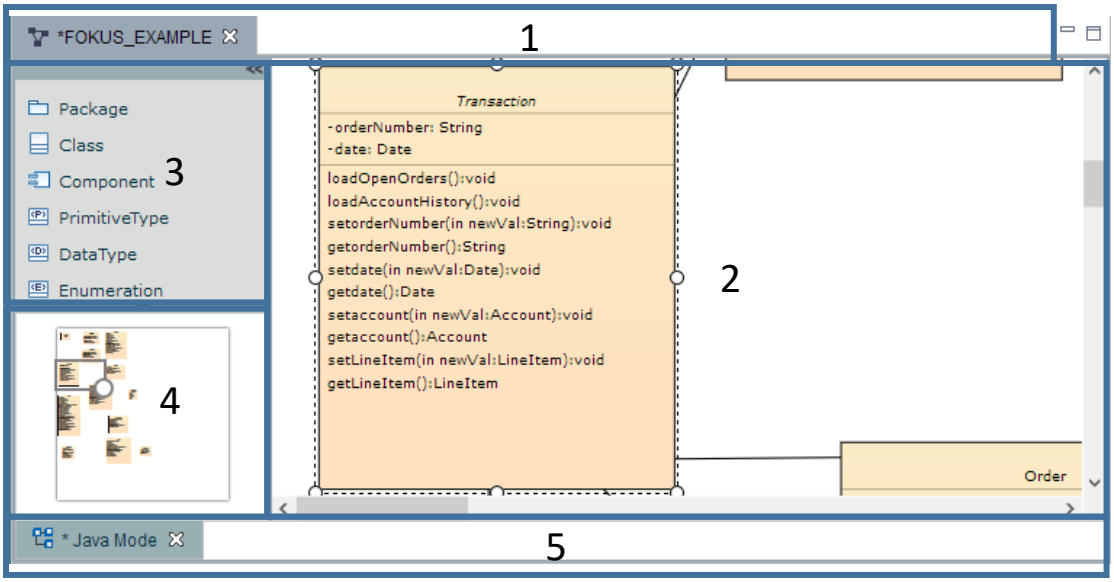


Abbildung 2.11: Detailabbildung des ModelEditors am Beispiel eines Klassendiagramms

Die Reiterleiste (1) zeigt den Namen des aktuell geöffneten Modells. Das Diagramm selbst wird im Bereich (2) angezeigt. Neue diagrammspezifische Elemente, zum Beispiel eine Klasse oder ein Package, können per Selektion und Doppelklick innerhalb des Diagrammbereiches oder „Ziehen und Ablegen“ hinzugefügt werden. Mithilfe des Rechtecks innerhalb der Miniaturübersicht des Diagramms (4) kann durch Skalierung der sichtbare Bereich angepasst und durch Verschieben navigiert werden. Der Name jedes geöffneten Diagramms wird in der Reiterleiste (5) angezeigt, so dass ein Wechsel zwischen den Diagrammen ermöglicht wird.

2.3.2 ModelExplorer

Die Bestandteile des ModelExplorers sind in Abbildung 2.12 farblich markiert und werden im Folgenden erläutert.

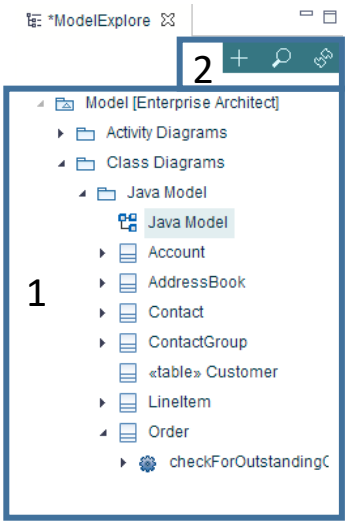


Abbildung 2.12: Detaildarstellung des ModelExplorers

Der ModelExplorer zeigt alle vorhandenen Elemente eines Projektes in einer baumbasierten Struktur (1) an. Zudem verfügt der ModelExplorer über Funktionalitäten, wie das Hinzufügen und Suchen von Elementen oder Aktualisieren der angezeigten Elemente (2).

2.3.3 Properties View

Der Properties View zeigt alle Eigenschaften eines selektierten Elementes an und erlaubt die Bearbeitung durch den Benutzer. In Abbildung 2.13 sind die Bestandteile farblich markiert und werden nachfolgend erläutert.

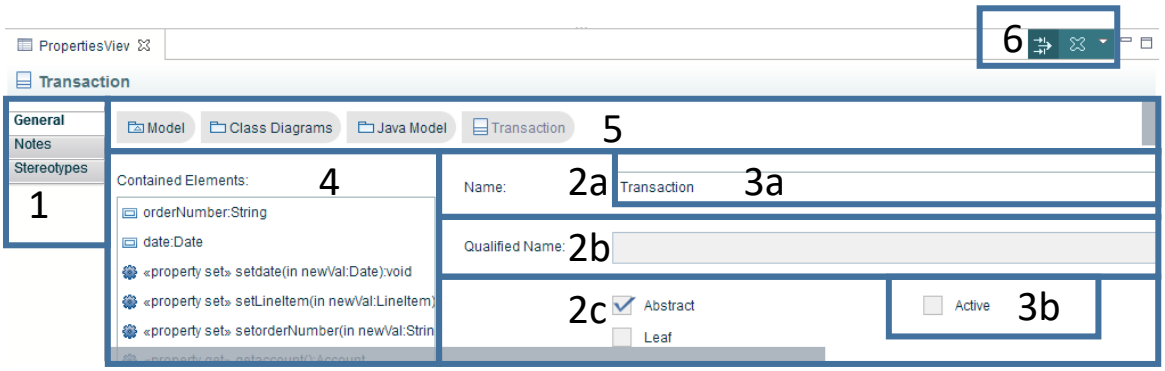


Abbildung 2.13: Detaildarstellung des Properties View

Der Properties View basiert auf dem Eclipse Properties View [Hunt06] und gliedert sich in Reiter (1), Sektionen (2) und Widgets (3). Mithilfe der Reiterleiste kann zwischen den verschiedenen Reitern gewechselt werden. In Abhängigkeit des selektierten Elementes werden die darzustellenden Eigenschaften generisch und reflektiv ausgelesen und gebündelt in Sektionen dargestellt. So existiert beispielsweise eine Sektion mit dem Namen (2a) des Elementes, welcher in einem entsprechend des Datentyps passenden Widget, in diesem Fall ein Textfeld (3a), dargestellt wird. In der Sektion (2b)

werden alle textbasierten und in Sektion (2c) alle Eigenschaften des Datentyps Boolean mit einem entsprechendem Widget (3b) repräsentiert. Zusätzlich werden alle untergeordneten Elemente eines selektierten Elementes in einer tabellarischen Übersicht (4) sowie die jeweiligen übergeordneten Elemente des selektierten Elementes bis zum Wurzelelement mithilfe einer Brotkrumen-Navigationsleiste (5) dargestellt. In der Statusleiste (6) kann zwischen einer einfachen oder fortgeschrittenen Ansicht des Properties Views umgeschaltet oder das selektierte Element gelöscht werden. Realisiert werden die unterschiedlichen Ansichten der anzuzeigenden Sektionen und Widgets mithilfe von Filter-Klassen.

2.4 Differenzierung von Teststufen

Um die Einhaltung der zuvor festgelegten Anforderungen an eine Anwendung, wie zum Beispiel ModelICE, zu überprüfen und zu gewährleisten, werden Softwaretests entwickelt und ausgeführt [SpLi05, Kap.2.1 S.6]. Mit diesen Tests können Laufzeitfehler frühzeitig entdeckt und behoben werden. Damit ein Test durchgeführt werden kann, muss sich die zu testende Anwendung in einem bestimmten Kontext befinden. Beispielsweise muss ein Benutzer angemeldet sein, damit eine zu testende Funktion ausgeführt werden kann. Diese Voraussetzungen können die Komplexität eines Tests beeinflussen.

Die Tests werden in vier Stufen unterteilt [Hoff13, Kap.4.2 S. 159] (vgl. Abbildung 2.14):

- Unit-Tests
- Integrationstests
- Systemtests
- Abnahmetests

Im Rahmen dieser Arbeit werden die Abnahmetests nicht berücksichtigt, da diese vom Auftraggeber und in dessen realen Einsatzumgebung ausgeführt werden [Hoff13, Kap.4.2.1.4 S. 168 - 170].

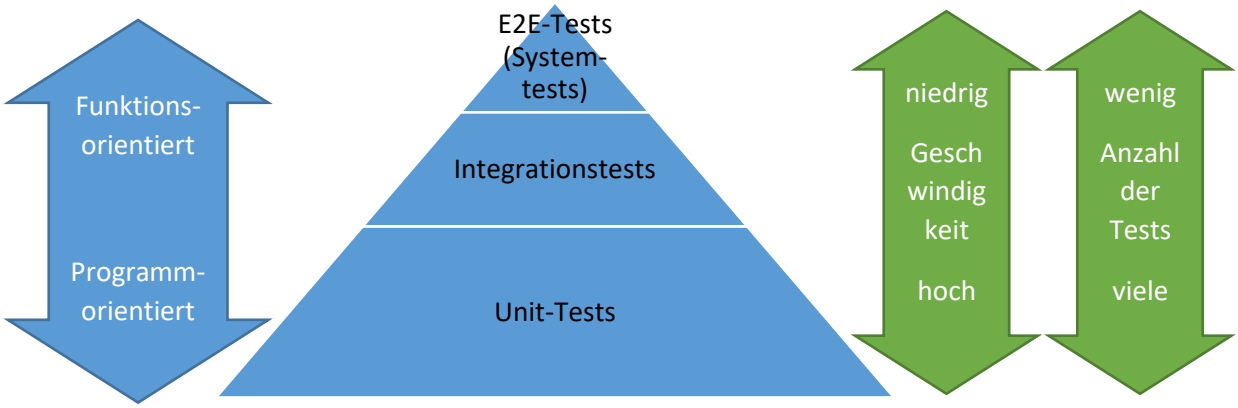


Abbildung 2.14: Übersicht der unterschiedlichen Testarten

Unit-Tests oder „*Modultests*“ stellen die Basis der Pyramide dar und testen isoliert eine Programmeinheit [Hoff13, Kap.4.2.1.1 S. 159-163]. Die Größe einer Einheit ist nicht präzise definiert, sodass einzelne Methoden, aber auch Klassen umfasst werden können. Unit-Tests überprüfen programmorientiert, ob spezifizierte Funktionalitäten der Testfälle anforderungsgemäß und unabhängig voneinander erfüllt werden. Dabei werden Modultests entweder von den Entwicklern

selbst entwickelt oder es wird bewusst darauf verzichtet [Mosi17][21]. Zudem können Modultests während der Entwicklung ständig ausgeführt werden, um im Falle einer Modifikation der zu testenden Einheit Laufzeitfehler direkt zu erkennen und zu beheben. Derartige Tests werden „*Regressionstests*“ genannt.

Aufbauend werden Integrationstests entwickelt, welche die überprüften Programmiereinheiten der Modultests im Zusammenspiel testen [Hoff13, Kap.4.2.1.2 S. 163ff]. Die Integration der Komponenten kann sich dabei an einer Struktur oder Funktionalität orientieren. Die strukturorientierte Integration berücksichtigt die Abhängigkeiten zwischen den Komponenten. So werden beispielsweise zuerst grundlegende Komponenten getestet und anschließend die direkt abhängigen Komponenten. Die funktionsorientierte Integration gliedert die Komponenten unabhängig von ihren Abhängigkeiten. Zum Beispiel werden ausgehend von einer bestimmten Testanforderung die notwendigen Komponenten für diesen Test ausgewählt und getestet.

Die Spitze der Pyramide bilden die funktionsorientierten Systemtests, welche - ähnlich wie Integrationstests - das Zusammenspiel zwischen Komponenten und die Einhaltung der vereinbarten Anforderungen prüfen, hierbei jedoch alle Komponenten der Software einbeziehen [Hoff13, Kap.4.2.1.3 S. 166-168]. Eine Abwandlung eines Systemtests ist der „*Ende-zu-Ende*“-Tests (E2E). Diese testen speziell die grafische Benutzeroberfläche einer Anwendung, indem eine Abfolge von benutzerbasierten Aktionen manuell und wiederholt ausgeführt und ausgewertet wird. Da das manuelle Testen insbesondere fehleranfällig sowie zeitaufwendig ist, existieren Frameworks um diesen Prozess zu automatisieren.

[21] Diese Quelle informiert über die detaillierten Hintergründe, sodass keine Erklärung im Rahmen dieser Arbeit erfolgt.

3 Ausgangslage und Problemanalyse

Nach Beschreibung der ModelICE Architektur wird die Ausgangslage für die Problemanalyse bezüglich der Testbarkeit und Performance der aktuellen Implementierung von ModelICE mit dem Framework RAP durchgeführt.

3.1 Ausgangslage mithilfe der Architektur von ModelICE

Der im Abschnitt 2.3 erläuterte Überblick der aktuellen Implementierung des Werkzeugs ModelICE dient als Grundlage, um die Ausgangslage mithilfe der Architektur von ModelICE in diesem Abschnitt zu beschreiben.

Bei der Architektur von ModelICE mit Eclipse RAP wird das Architekturmuster Model-View-Controller (siehe Abschnitt 2.1.4) verwendet. Die Darstellung des Modells wird vom entsprechenden Klienten repräsentiert und der Server beinhaltet das Modell, welches mithilfe des Controllers modifiziert wird und anschließend die View aktualisiert. Eine Kernkomponente des Controllers ist der „*Connector*“ (vgl. Abbildung 3.1), welcher die bidirektionale Benutzung und Bearbeitung zwischen einer „*Datenquelle*“ und einem Modell ermöglicht, wie zum Beispiel einem UML-Modell. Mithilfe der Datenquelle können Datenmodelle eines zu integrierenden Werkzeuges zum Beispiel von Enterprise Architect verwenden und modifiziert werden.

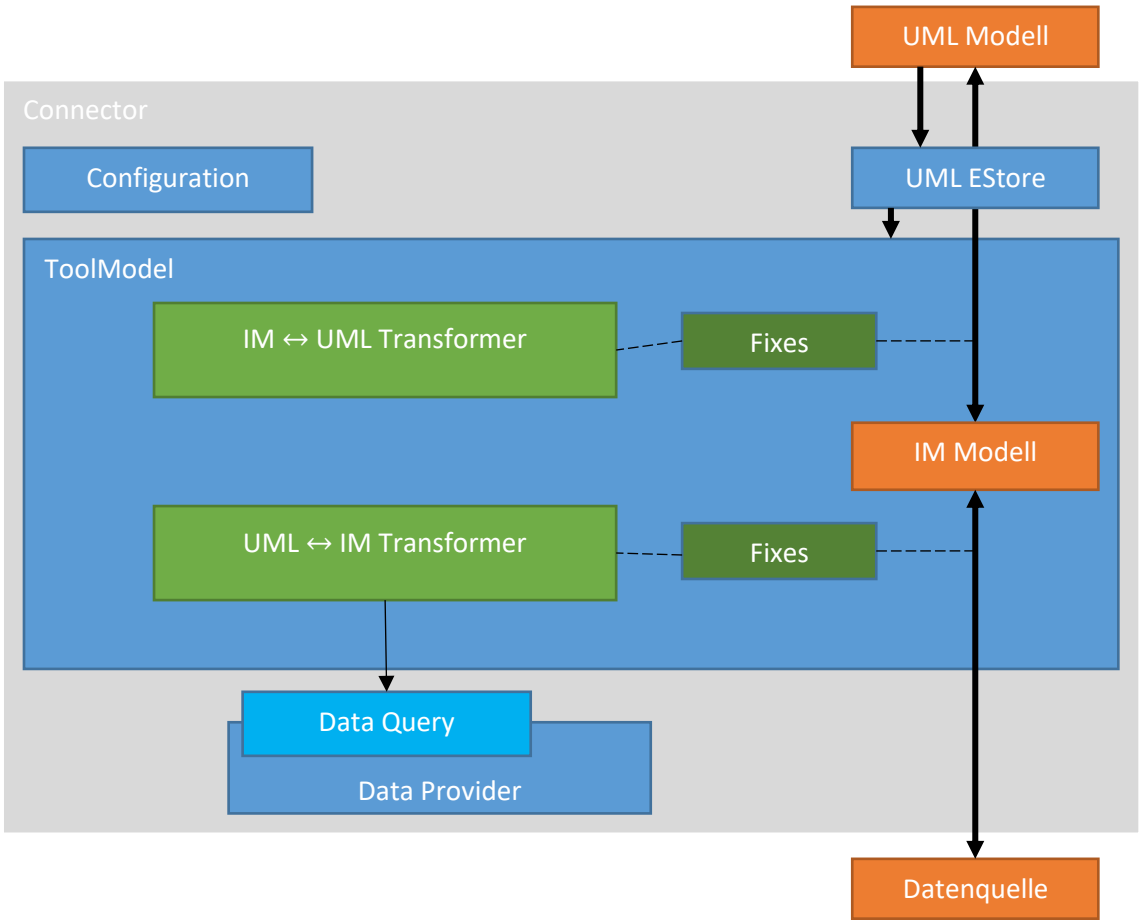


Abbildung 3.1: Die Software-Architektur von ModelICE

Nachfolgend wird kompakt die bestehende Architektur beschrieben.

Die bidirektionale Konvertierung innerhalb des Connectors wird ermöglicht, indem zuerst ein Transformer das Quellmodell in ein werkzeugeigenes Modell (engl. „*Intermediate Model*“ - IM) umwandelt und ein weiterer Transformer das IM Modell zu einem standardkonformen UML-Modell der Version 2.5 [Unif15] transformiert. Mithilfe der losen Kopplung, welche durch den Einsatz des IM Modells entsteht, können Eingabeformate von weiteren Werkzeugen oder Datenquellen angebunden und in unterschiedliche Ausgabeformate transformiert werden. Durch „*Modelfixes*“ werden zu integrierende UML-Werkzeuge bidirektional an die von ModelICE verwendete UML-Version angepasst. Insbesondere für die Integration des EA sind die Modelfixes relevant, da mit EA UML-Modelle erstellt werden können, welche nicht unbedingt standardkonform sind. Ist zum Beispiel der Datentyp eines Elementes von EA `Text`, so wird dieser mit Verwendung des `CommentTypeFixes` in den Datentyp `Comment` manipuliert.

Die bidirektionale Modelltransformation wird in der Komponente „*ToolModel*“ zusammengefasst, welche die beschriebene Komplexität mithilfe des Entwurfsmusters „Fassade“ für die zu verwendende Komponente abstrahiert und eine Schnittstelle für lesenden und schreibenden Zugriff auf das Datenmodell bereitstellt.
Um den Zugriff auf eine Datenquelle zu vereinfachen, existieren die Komponenten „*Data Provider*“ und „*Data Query*“. Der Data Provider stellt dabei Basisfunktionalitäten für den Zugriff zur Verfügung, während Data Query diese Funktionalitäten verwendet, um von der konkreten Syntax einer Abfrage zu abstrahieren.
In der Komponente „*Configuration*“ können Einstellungen angepasst werden, wie zum Beispiel die zu verwendende Strategie für das Laden eines Modells in Abhängigkeit von Anforderungen konkreter Anwendungsszenarien. So können etwa für einen Anwendungsfall erforderliche Daten mit wenigen, hochgradig optimierten Anfragen mit einem Mal geladen werden, während es auf der anderen Seite auch möglich ist, Daten erst dann selektiv nachzuladen, wenn sie benötigt werden. Der „*UML EStore*“ ermöglicht die Integration der Daten in die API von EMF, sodass das Datenmodell als EMF Modell behandelt werden kann. Zudem werden unter anderem die Modifikationen lokal als auch global zwischengespeichert, um bei erneuter Durchführung die Performance durch direkte Ausführung aus dem Zwischenspeicher zu steigern.

3.2 Problemanalyse

Mithilfe der zuvor definierten Ausgangslage wird eine Problemanalyse mit dem Fokus auf die Testbarkeit sowie der Performance durchgeführt.

3.2.1 Testbarkeit der grafischen Benutzeroberfläche

Bei der Software ModelICE werden zahlreiche Tests automatisiert ausgeführt. In diesem Abschnitt werden die verwendeten Test-Frameworks mit deren Einsatzbereichen in der aktuellen Implementierung von ModelICE sowie die entstehenden Schwierigkeiten hinsichtlich der Testbarkeit von E2E-Tests beschrieben.

Eingesetzt wird zum einen das Framework „*SWTBot*“ [Swtb00], welches auf das Testen von Anwendungen mit SWT spezialisiert ist. Das Framework verfügt über eine Aufnahmefunktion, die einen Interaktionsablauf aufzeichnet und Quelltext für den entsprechenden Test generiert, welcher anschließend angepasst werden kann.

Wie bereits in Abschnitt 2.1.8 erwähnt, kann mithilfe von RCP eine Anwendung nicht nur mit RWT auf einem Klienten, sondern auch mit SWT auf einem Desktop ausgeführt werden. Aufbauend auf der Annahme, dass die relevanten SWT-Funktionalitäten identisch zu RWT-Funktionalitäten implementiert wurden, ist es möglich, Testfälle in der Desktopversion mit SWTBot zu entwickeln und auszuführen, obwohl bei entfernten Klienten RWT eingesetzt wird[22].

Für Testfälle, die aufgrund der beschriebenen Einschränkungen von RWT nicht mit SWTBot realisierbar sind, wird - wenn möglich - das Framework *„Selenium“* [Sele00] eingesetzt. Mit Selenium kann die Nutzung eines Internetbrowsers durch einen Benutzer simuliert werden. Dadurch ist die Realisierung von E2E-Tests oder wiederkehrenden Vorgängen im Browser möglich. Die auszuführenden Testschritte werden in der Programmiersprache Java entwickelt.

Da der ModelICE ModelEditor durch einen JavaScript basierten Editor realisiert wird, kann dieser nicht mit SWTBot getestet werden. Stattdessen werden die E2E-Tests des Editors ebenfalls mit Selenium entwickelt und ausgeführt.

Beide genannten Test-Frameworks müssen während eines E2E-Tests ein HTML-Element eindeutig identifizieren können. In Tabelle 3.1 ist eine Übersicht dargestellt, welche Attribute für die Identifikation zur Verfügung stehen.

Attribut	Erklärung
`id`	Eindeutige XML-valide Identifikationsmöglichkeit.
`class`	Identifikationsmöglichkeit für zusammengehörige HTML-Elemente. Ein Element kann beliebig viele Klassen besitzen.
`data-<eigener Name>`[23]	Mithilfe dieses Attributes können eigene HTML-valide Attribute definiert und verwendet werden.
Selbst definiert	selbst definierte Attribute sind in der Regel Bestandteil einer eigenen XML-basierten Sprache und können nicht von einem Browser interpretiert werden.

Tabelle 3.1: Übersicht der Attribute zur Identifikation eines HTML-Elementes

Für eine brauchbare Anwendung dieser vier Identifikationsmöglichkeiten innerhalb der E2E-Tests von ModelICE ist es daher notwendig, dass die Identifikation eines Elementes unveränderlich beziehungsweise stabil ist. Nur so kann garantiert werden, dass ein automatisiert durchgeführter E2E-Test ein zu selektierendes Element bei jedem Test wiederholt eindeutig identifizieren kann.

Das RAP Framework, welches den HTML-Quelltext der Anwendung ModelICE zur Laufzeit erzeugt, bietet jedoch selbst keine Funktionalität zur Realisierung der genannten Identifikationsmöglichkeiten an. Aus diesem Grund müssen alternative Identifikationsmöglichkeiten zu einem HTML-Element innerhalb eines Test-Frameworks eingesetzt werden. Während eines E2E-Tests kann zwar mithilfe relativer Pfadangaben von *„CSS-Selektoren“* [ÇEGH18] oder der *„XML Path Language“* [RoDS00] eine Identifikation eines Elementes erfolgen. Diese relativen Pfadangaben eines E2E-Tests eines eingesetzten Test-Frameworks können jedoch nach Aktionen des Benutzers, wie zum Beispiel dem Hinzufügen mehrerer Elemente über den ModelExplorer oder den ModelEditor, obsolet werden, da der Pfad zu einem bestimmten Element potentiell verändert werden kann. Zudem werden die

[22] Mithilfe der OSGi-Funktionalität „Import-Package“ ist es möglich, die zu importierende Implementierung einer Klasse zur Laufzeit auszutauschen.

[23] https://www.w3.org/TR/html52/dom.html#embedding-custom-non-visible-data-with-the-data-attributes

Positionen der HTML-Elemente durch das RAP Framework mit absoluten Koordinaten beschrieben und müssen modifiziert werden, wenn Verschiebungen von Elementen, etwa resultierend aus Benutzerinteraktionen, eintreten. Insbesondere bei baumartigen Strukturen, wie sie im ModelExplorer Anwendung finden, kann die Anzahl der Modifikationen erheblich steigen, wenn zum Beispiel ein erweiterbares Element mit zahlreichen untergeordneten Elementen expandiert wird. In Extremfällen ist es sogar unmöglich ein bestimmtes Element zu selektieren, weil sich der Pfad einer Selektion durch die unzähligen Modifikationen verändert.

3.2.2 Performance der ModelICE Software

Die Performance ist bedeutend für die Interaktion eines Benutzers mit ModelICE-Komponenten, wie zum Beispiel dem ModelExplorer oder dem Properties View. In diesem Abschnitt werden Schwierigkeiten der Performance in der aktuellen Implementierung von ModelICE beschrieben, welche auf die Verwendung des klientseitigen UI-Frameworks RAP zurückzuführen sind.

RAP leitet die grafische Benutzeroberfläche aus dem SWT Quelltext ab und berechnet alle einzelnen HTML-Elemente serverseitig. Die Berechnung jedes HTML-Elementes umfasst die Ermittlung der absoluten Position, dessen Größe und Layout sowie die verwendeten CSS-Regeln, welche überwiegend innerhalb des Kopfteils der zu übertragenden HTML Webseite beschrieben werden. Die daraus entstehenden Daten zur Beschreibung der Benutzeroberfläche werden anschließend mithilfe des RAP Protokolls [Rap/00b] an den Klienten übermittelt. Bei jeder Manipulation der Benutzeroberfläche durch den Benutzer müssen somit sämtliche Informationen zur Beschreibung der Oberflächenlayouts erneut vom Server berechnet und übertragen werden.

Zusätzlich wird für das Referenzieren einer klientseitigen Grafik der Hashwert der entsprechenden Pixelmatrix verwendet, welcher allerdings bei jeder, also auch mehrfacher, Verwendung neu berechnet wird. Dieser Vorgang ist verhältnismäßig ressourcenintensiv und unflexibel. Werden beispielsweise zusätzliche Dekorationen, also Überlagerungen mit einer anderen Grafik, für ein Icon benötigt, fügt RAP diese beiden Grafiken überlagernd zu einer einzigen zusammen und errechnet darauf erneut den Hashwert. Die Verwendung der Dekorationen werden in ModelICE unter anderem dazu benötigt den Lock-Status[24] im ModelExplorer anzuzeigen. Die Anzahl der Kombinationsmöglichkeiten, welche sich aus den etwa 260 Elementtypen der UML und vier verschiedenen Lockarten in ModelICE ergeben, illustriert die problematische Ressourcenbelastung bei Verwendung des RAP-Frameworks.

Die aus den genannten Gründen resultierenden großen Datenmengen und Häufigkeiten ressourcenintensiver Abfragen zwischen Klient und Server verlangsamen die Darstellung der Anwendung, sodass die Interaktion als nicht zufriedenstellend bewertet wird.

[24] Mit einem Lock ist die Sperrung einzelner Element oder ganzer Pakete möglich, sodass die Bearbeitung dieser gesperrten Elemente nur freigegebenen Benutzern erlaubt ist.

4 Realisierung

Mithilfe der in diesem Kapitel beschriebenen Realisierung einer prototypischen Migration lässt sich eine Evaluation der Testbarkeit beziehungsweise Performance im Kapitel 5 durchführen. In diesem Kapitel wird zuerst ein Lösungsansatz für die Realisierung der Migration von ModelICE zu Eclipse Che definiert und anschließend werden die Meilensteine der Implementierung beschrieben.

Die Eclipse Che Plattform verwendet im Gegensatz zu der bisher verwendeten Eclipse Plattform nicht SWT, JFace oder RAP sondern GWT, ein alternatives Framework zur Erstellung der grafischen Benutzeroberfläche (siehe Abschnitt 2.2.3). Insbesondere die enthaltene Funktionalität einem HTML-Element ein ID-Attribut hinzuzufügen, kann die Testbarkeit von ModelICE erhöhen, da nun ein Test-Framework ein zu testendes Element auch bei Modifikationen der HTML Struktur eindeutig identifizieren und selektieren kann. Zudem kann die Kommunikation zwischen dem Klienten und dem Server effizienter implementiert werden, indem nur die Modellinformationen der anzuzeigenden Elemente an den Klienten übertragen werden und der Klient selbst für die Repräsentation der Daten zuständig ist. Außerdem ist Eclipse Che cloudbasiert, wodurch Vorteile, wie etwa die beschriebene horizontale oder vertikale Skalierung von Docker Containern (siehe 2.2.1) bei ModelICE Anwendung finden kann.

4.1 Lösungsansatz

Damit die Migration von ModelICE erfolgen kann, müssen Eclipse Che und ModelICE vorbereitet werden. So mussten aufgrund der neuen, Che basierten Softwarearchitektur geringfügige Änderungen an ModelICE vorgenommen und Eclipse Che um ModelICE spezifische Komponenten erweitert werden.

Der Build von Eclipse Che soll lokal erfolgen, sodass ein entsprechender Build mit Plug-Ins für die Komponenten von ModelICE erstellt und konfiguriert werden muss. Da Eclipse Che nicht auf OSGi basiert, muss OSGi aus ModelICE herausgelöst werden, indem OSGi spezifische Implementierungen durch entsprechende Äquivalente ersetzt werden. Zudem wird die Erstellung der grafischen Benutzeroberfläche bei Eclipse Che mit GWT und nicht mit RAP realisiert, sodass entsprechende RAP-spezifische Teile von ModelICE angepasst und vor allem gemeinsam genutzte Bestandteile herausgelöst werden müssen. Dabei sollen bereits vorhandene Funktionalitäten von ModelICE hauptsächlich wiederverwendet werden und die aktuelle Implementierung mit RAP weiterhin uneingeschränkt lauffähig sein. Zusätzlich soll klientseitiger Quelltext möglichst wenige Abhängigkeiten besitzen, da GWT den entsprechenden Quelltext vollständig nach JavaScript (siehe 2.2.3) übersetzt.

Für die Realisierung des gewählten Ansatzes, findet nach Anpassung der Software ModelICE die Migration der beiden ausgewählten ModelICE-Komponenten ModelExplorer und Properties View in die erstellten Eclipse Che-Plug-Ins statt.

4.2 Implementierung

Um den zuvor beschriebenen Lösungsansatz zu implementieren, werden Meilensteine definiert:

1. Eclipse Che Entwicklungsumgebung einrichten
    a. Lokaler Build von Eclipse Che
    b. Erstellung von Eclipse Che Plug-Ins für die ModelICE-Komponenten
2. ModelICE vorbereiten
    a. Erstellung der OSGi unspezifischen ModelICE-Software
    b. Abstrahierung des Quelltextes für die grafische Benutzeroberfläche zur Verwendung eines beliebigen Frameworks
        i. Abstrahierung des ModelExplorers
        ii. Abstrahierung des Properties Views
3. Migration von ModelICE nach Eclipse Che
    a. Erstellung des Modell-Explorers
    b. Erstellung der Properties-View

4.2.1 Eclipse Che Entwicklungsumgebung einrichten

Dieser Abschnitt befasst sich mit dem Einrichten der Eclipse Che Entwicklungsumgebung. Als Resultat der Implementierung wird ein Che Assembly mit den benötigten Plug-Ins erstellt.

Um mit der Entwicklung einer Eclipse Che-Anwendung zu beginnen, muss die entsprechende Entwicklungsumgebung eingerichtet werden. Ein passender Leitfaden ist in der ausführlichen Eclipse Che Dokumentation [Ecli00d] zu finden. Diese beinhaltet benötigte Informationen zum Implementieren einer Anwendung mit Che oder zum Erstellen eines Plug-Ins für Che. Zudem werden weitere Themen behandelt, wie zum Beispiel die Benutzerverwaltung oder die Administration von Workspaces. Auch Informationen über die interne Architektur sowie den Einsatz verwendeter Software, wie Docker oder Keycloak[25], stehen zur Verfügung.

4.2.1.1 Erstes Zwischenziel: Lokaler Build von Eclipse Che

Das erste Zwischenziel stellt der Build von Eclipse Che mit lokalen Quelldateien und anschließender Inbetriebnahme der Anwendung dar. Bevor der Build-Prozess gestartet werden kann, müssen Java, Maven und Docker[26] auf dem Rechner installiert sein.

Der Build-Prozess aller Quelldateien von Eclipse Che scheiterte aufgrund der fehlenden Abhängigkeiten und Konfigurationen. Zudem ist der vollständige Build zeitaufwendig, was sich als erheblicher Nachteil herausstellte.
Als Alternative wird eine Vorlage eines Che-Assemblies verwendet. Der Versuch einen lokalen Build-Prozess dieses Assemblies zu starten, misslingt aufgrund eines nicht validen Dateipfades eines Che internen Skripts. Nach der manuellen Behebung dieses Pfades wurde der Build erfolgreich abgeschlossen, jedoch mit Quelldateien der offiziellen Eclipse Server, sodass die lokalen Quelldateien nicht berücksichtigt wurden.
Das neu konfigurierte Kommando ist weiterhin fehlerhaft. Die Nachfrage in dem thematisch passenden

[25] Keycloak ist eine Software, welche die Autorisierung und Authentifizierung von Benutzern verwaltet. https://www.keycloak.org/
[26] Die Mindestversionen der Software sind: Oracle oder OpenJDK Java 1.8, Docker 1.8 und Maven 3.3.1

Mattermost Kanal[27] zu Eclipse Che resultierte in keiner hilfreichen Antwort. Erst die Nachfrage in der Mailingliste der Eclipse Che Gemeinschaft informiert darüber, dass das bisher verwendete Docker-Kommando nicht nur die lokalen Quelldateien, sondern ebenfalls lokale Skripte an einem angegebenen Pfad erwartet. Da jedoch keine lokalen Skripte erstellt wurden, scheiterten die Build-Versuche. Die in einer E-Mail angegebene Lösung sieht eine zusätzliche Angabe einer Konfiguration vor. Das Resultat ist die Auslassung der lokalen Skriptdateien, sodass die Skripte von den Eclipse Che Servern verwendet werden.

Der Build-Prozess funktioniert mit Angabe der Konfiguration erfolgreich. Nach Start der Docker-Container von Eclipse Che lässt sich hingegen kein Workspace starten. Mithilfe der Lösung eines ähnlichen Problems[28] eines Benutzers der Eclipse Che Plattform lässt sich die Ursache des entstandenen Workspace-Problems auf die fehlende Zugriffsberechtigung von Dateien und Ordnern eingrenzen. Um das Workspace-Problem zu beheben, muss die `assembly.xml` des *„assembly-wsagent-server“*-Projektes um die Angabe einer Zugriffsberechtigung (vgl. Listing 4.1, Z.9) ergänzt werden.

```
1  ...
2  <fileSets>
3      <fileSet>
4          <directory>${project.build.directory}/dependency/</directory>
5          <outputDirectory></outputDirectory>
6          <excludes>
7              <exclude>webapps/ROOT.war</exclude>
8          </excludes>
9          <fileMode>0700</fileMode>
10     <fileSet>
11 <fileSets>
12 ...
```

Listing 4.1: Ausschnitt der `assembly.xml` des „assembly-wsagent-server“-Projektes mit Ergänzung der Zugriffsberechtigung

[27] https://mattermost.eclipse.org/eclipse/channels/eclipse-che
[28] https://github.com/eclipse/che/issues/5043

Der vollständige Docker-Befehl zum Starten von Eclipse Che ist im Listing 4.1 und dessen Erklärung in Tabelle 4.1 zu sehen.

```
1  docker run -ti --rm -v /var/run/docker.sock:/var/run/docker.sock -v
   ~/Desktop/eclipse-che-modelice/data:/data -v ~/Desktop/eclipse-che-
   modelice/eclipse-che:/repo eclipse/che:6.9.0 start --skip:scripts
```

Listing 4.2: Vollständiger Docker-Befehl zum Starten von Eclipse Che mit lokalen Quelldateien

Konfiguration		Bedeutung
docker run		Startet einen Docker Container
-ti		Stellt eine Verbindungsmöglichkeit mit einem Terminal des Containers zur Verfügung.
-rm		Entfernt den Container automatisch, wenn dieser gestoppt wird.
-v	<lokaler Pfad>:/<Pfad im Container>	Bereitstellung bestimmter Pfade des Computers für Eclipse Che.
	/var/run/docker.sock:/var/run/docker.sock	Freigabe des lokalen Docker Dämons für den Eclipse Che Container.
	~/Desktop/eclipse-che-modelice/data:/data	Definiert den Pfad zu einem Ordner, in welchem Eclipse Che-spezifische Daten gespeichert werden können.
	~/Desktop/eclipse-che-modelice/eclipse-che:/repo	Definiert einen Pfad für die Verwendung von lokalen Quell- und Skriptdateien.
eclipse/che:6.9.0 start		Startet die angegebene Eclipse Che Version.
--skip:scripts		Überspringt die lokalen Skriptdateien und verwendet stattdessen die Skripte von Eclipse Che.

Tabelle 4.1: Details des Docker-Kommandos für Eclipse Che

Der erste erfolgreiche Build benötigt ungefähr 15 Minuten auf einem bereitgestelltem Entwicklungslaptop Lenovo Thinkpad T530. Etwa zwei weitere Minuten sind erforderlich, um die Eclipse Che mit Docker zu starten und einen Workspace zu öffnen. Aufgrund der heruntergeladenen und installierten Abhängigkeiten des ersten Builds, verkürzt sich die Dauer weiterer Builds um durchschnittlich drei Minuten. Die Che Umgebung konnte nun erfolgreich eingerichtet werden, jedoch ist der Entwicklungsprozess schwerfällig, da bei jeder Veränderung eines Plug-Ins der Build neu gestartet werden muss.

4.2.1.2 Zweites Zwischenziel: Erstellung eigener Plug-Ins für die Kernfunktionalitäten von ModelICE

Das zweite Zwischenziel ist die Erstellung eines Assemblies für ModelICE mit dessen Kernkomponenten mithilfe eigener Plug-Ins. Die Modifikation des existierenden Plug-Ins der Eclipse Che-Vorlage führt zu weiteren Fehlern während des Build-Prozesses. Insbesondere eine Änderung der voreingestellten Gruppierungsbezeichnungen der vorhandenen und neu erstellten Plug-Ins resultierte in diesen Fehlern. Als Abhilfe wird die Vorlage von Eclipse Che mit einem Plug-In zur Bearbeitung von JSON-Dateien verwendet. Der Build und Start funktionieren einwandfrei. Das Hinzufügen weiterer Plug-Ins erfolgt durch Änderungen in den hervorgehobenen POM-Dateien der Abbildung 4.1. In der

`pom.xml` (1) des gesamten Eclipse Che Projektes müssen alle Teilprojekte eines Plug-Ins als Abhängigkeit definiert werden. Zudem muss das Server-Projekt als Abhängigkeit in dem Projektmodell *„assembly-wsagent-war“* (2) und das zugehörige IDE-Projekt in der `pom.xml` des Projektes *„assembly-ide-war“* (3) hinzugefügt werden. In dem Projektmodell des *„plugin-parent“* (4), muss der Name des Plug-Ins als Maven-Modul ergänzt werden. Anschließend werden alle Teilprojekte des Plug-Ins in der `pom.xml` (5) des zu erstellenden Plug-Ins ebenfalls als Maven-Module hinzugefügt. Sollen Artefakte aus einem der anderen Projekte des gleichen Plug-Ins verwendet werden, müssen diese Projekte als Abhängigkeiten in den jeweiligen Projektmodellen (6) definiert sein. Im Falle des JSON-Beispiels von Eclipse Che, ist das *„shared“*-Projekt eine Abhängigkeit in dem IDE-Projekt als auch im Server-Projekt. Mithilfe dieser Änderungen wurden Plug-Ins für den ModelICE Modelleditor, den Modell-Explorer, den Properties-View sowie der Search-View erstellt.

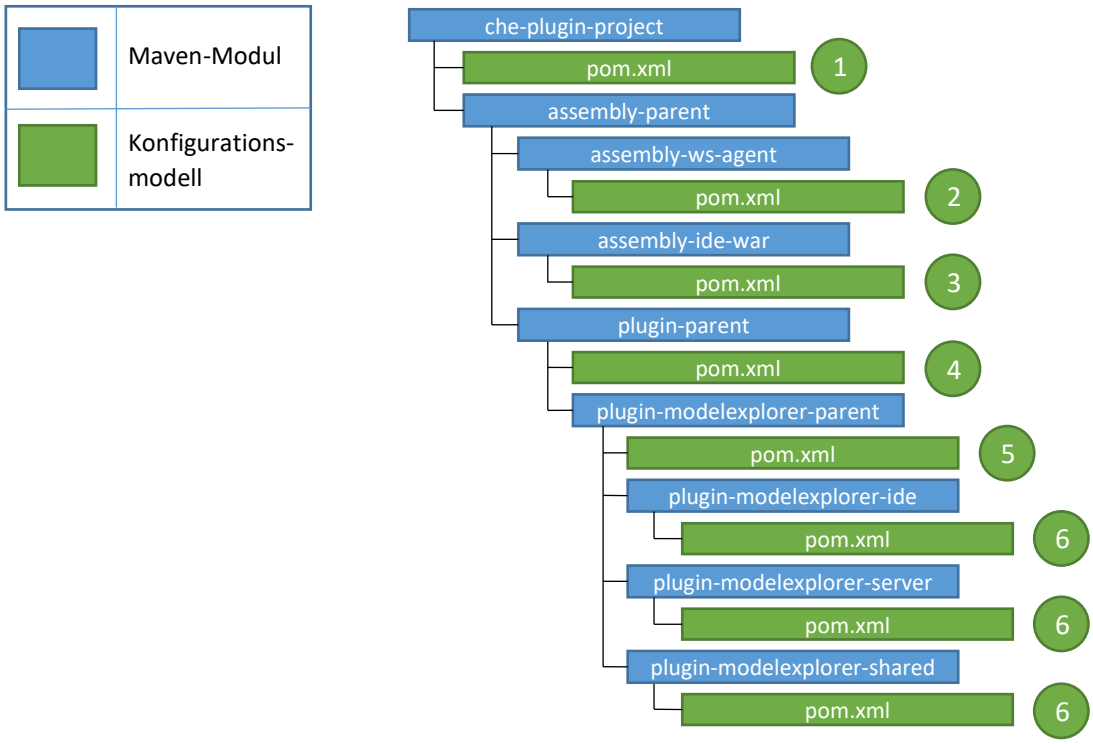


Abbildung 4.1: Übersicht der zu ändernden POM-Dateien bei dem Hinzufügen eines Plug-Ins

4.2.2 ModelICE vorbereiten

Dieser Abschnitt befasst sich mit der Erstellung des RAP unspezifischen Bestandteile der ModelICE-Software. Dabei wird die Umwandlung der OSGi-Anwendung auf eine nicht-OSGi-Anwendung erläutert sowie eine Abstrahierung der ModelICE-Komponenten realisiert, so dass resultierend ein beliebiges Framework für die Erstellung der grafischen Benutzeroberfläche benutzt werden kann, welche alle benötigten Bibliotheksabhängigkeiten der Software ModelICE in Form von Jar-Artefakten verwenden kann.

4.2.2.1 Erstellung der OSGi unspezifischen ModelICE-Software

Eclipse Che basiert nicht auf OSGi, so dass keine Komponente existiert, welche den Lebenszyklus der einzelnen Bundles steuert und diese beispielsweise aktiviert. Ebenfalls entfällt die Registratur, bei der die bisher entwickelten Erweiterungen und Erweiterungspunkte der Plug-Ins registriert wurden. Diese

Plug-Ins sollen in Eclipse Che jedoch wiederverwendet werden. Aus diesem Grund muss eine Alternative entwickelt werden, welche die bisherigen Plug-Ins trotz fehlender Registratur der Eclipse Che Anwendung hinzufügt und korrekt initialisiert.
Mithilfe eines Tutorials [Zerr10] kann eine Registratur für die Plug-Ins ohne die Verwendung von OSGi erstellt und benutzt werden. Die Klasse `ExtensionProcessor` analysiert die Plug-Ins auf ihre `plugin.xml` und fügt das entsprechende Plug-In mithilfe der enthaltenen Daten der XML-Datei einer Registratur hinzu.
EMF-spezifische Plug-Ins können mithilfe einer EMF-spezifischen Registratur verwaltet werden. Seit der Version 2.9[29] bietet EMF eine derartige Registratur für einen nicht-OSGi-Kontext an, welche daher in diesem Kontext verwendet wird.

4.2.2.2 Abstrahierung des Quelltextes für die grafische Benutzeroberfläche zur Verwendung eines beliebigen Frameworks

Nachdem nun ModelICE auch außerhalb eines OSGi Containers betrieben werden kann, wird die ModelICE-Implementierung derart modifiziert, dass ein beliebiges Framework für die grafische Benutzeroberfläche eingesetzt werden kann - mit der Bedingung, dass ModelICE mit RAP lauffähig bleibt. Dafür werden gemeinsam verwendete Implementierungen ausgelagert beziehungsweise refaktorisiert, welches am folgenden Beispiel der RAP „*UISession*“ [Rap-00] verdeutlicht wird.

Die beiden ModelICE-Komponenten ModelExplorer und Properties View verwenden in der aktuellen Implementierung von ModelICE eine RAP UISession, um beispielsweise einen Klienten eindeutig zu identifizieren und Klienten-spezifische Daten zu speichern. Mithilfe der eindeutigen Identifikation eines Benutzers werden zudem bidirektionale Manipulationen der Datenmodelle mithilfe des `ToolModels` realisiert. Diese Funktionalität einer UISession existiert ebenfalls bei GWT. Damit ModelICE mit beiden Varianten einer UISession interagieren kann, wird das Interface `ISessionContext` (vgl. Abbildung 4.2) für die gemeinsam verwendeten Funktionalitäten erstellt. Neben gemeinsam definierten Funktionalitäten einer UISession erlaubt diese Abstrahierung das Hinzufügen spezifischer Funktionalitäten einer RAP oder GWT UISession.

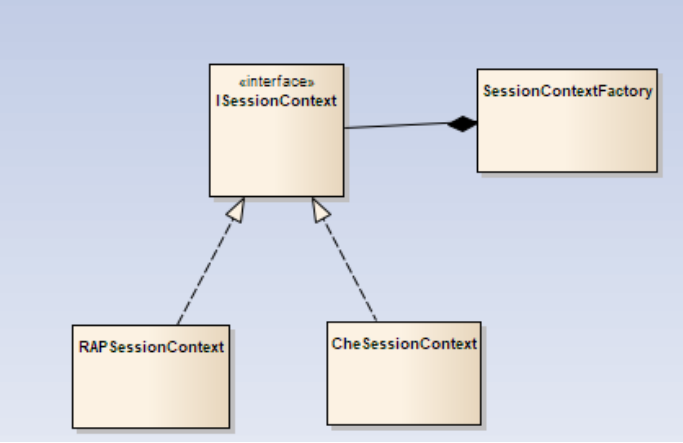


Abbildung 4.2: Abstrahierung der UISession bei ModelICE

[29]https://github.com/eclipse/emf/blob/e946acfddd5ef90a4ea99f801f72aa5e91c400e4/plugins/org.eclipse.emf.ecore/src/org/eclipse/emf/ecore/plugin/EcorePlugin.java#L698

Ebenfalls verwenden beide ModelICE-Komponenten das in Abbildung 4.3 dargestellte Metamodell für den Datenaustausch zwischen Klient und Server.

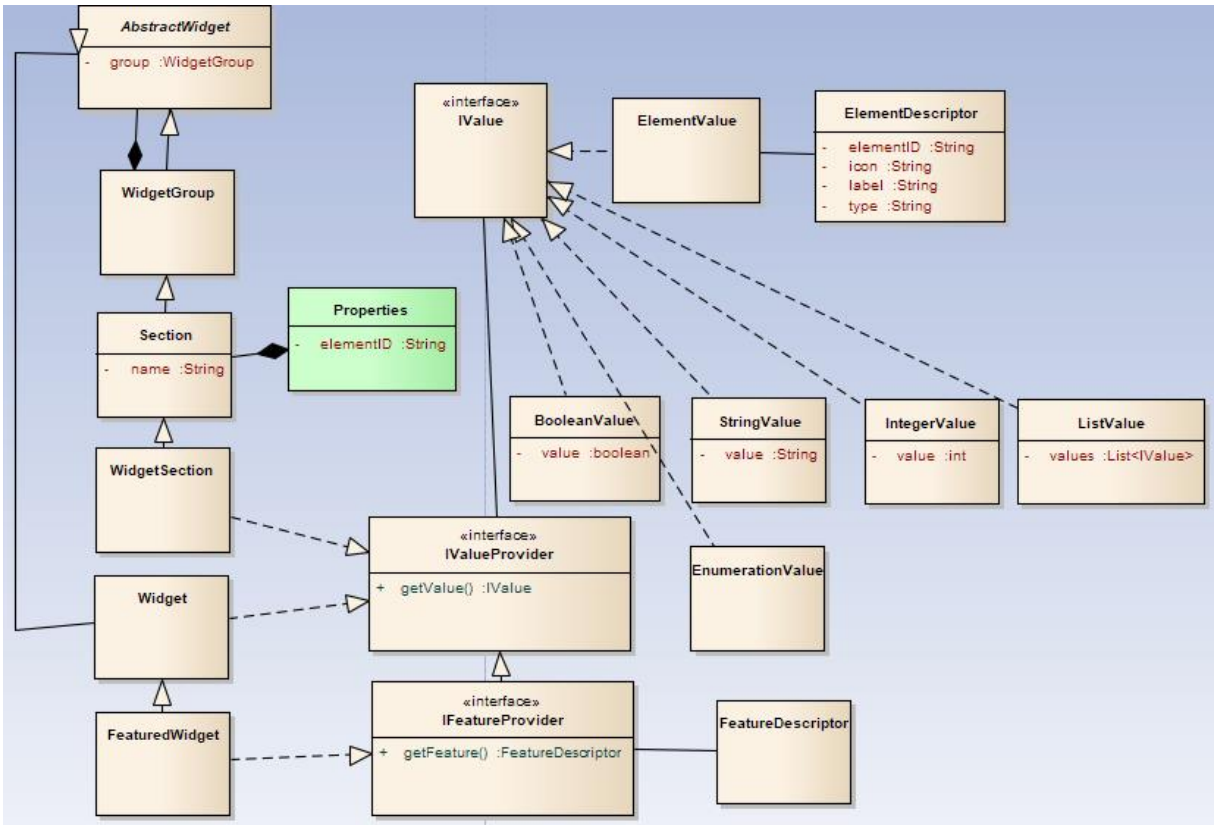


Abbildung 4.3: Metamodell für den Datenaustausch der Komponenten ModelExplorer und Properties View

Aufgrund der vollständigen Übersetzung des klientseitigen Quelltextes von Java zu JavaScript durch GWT werden bei diesem Metamodell nur POJOs (engl.: „plain old java objects“) verwendet. Beide ModelICE-Komponenten verwenden die Klasse `FeatureDescriptor`, um EMF Features und die Klasse `ElementDescriptor`, um ein Element mit Informationen wie den Namen, den Datentyp oder sein Icon zu beschreiben. Das Icon, inklusive optionaler Dekoration, wird nicht, wie bei RAP überlagert gespeichert und ein Hashwert für das klientseitige Referenzieren berechnet (siehe Abschnitt 3.2.2), sondern der Klient kann mithilfe der Pfade die Grafiken unabhängig vom Server anfragen und zwischenspeichern. Für die Aufteilung des Properties Views in Sektionen und den darin enthaltenen Widgets werden ebenfalls POJOs erstellt, wobei Instanzen der Widgets die serverseitigen berechneten Eigenschaftswerte in ihrer Rolle als `IValueProvider` transportieren.

Nach Entwicklung beider ModelICE-Komponenten sowohl in RAP als auch in GWT und der Abstrahierung der RAP-UISession werden jeweils für die einzelnen ModelICE-Komponente die gemeinsam verwendeten Funktionalitäten abstrahiert und in neue Basisimplementierungen in zusätzliche Eclipse Plug-Ins verlagert. Diese neuen Plug-Ins können als Basis für die Entwicklung der grafischen Benutzeroberfläche mit einem alternativem UI Framework zu RAP wie GWT, als Software-Abhängigkeit hinzugefügt werden, um Zugriff auf die grundlegenden Funktionalitäten der jeweiligen Komponente zu erhalten und redundante Implementierungen zu vermeiden. Nachfolgend werden die wichtigsten Bestandteile der Basisimplementierungen für den ModelExplorer sowie den Properties View kompakt beschrieben.

Abstrahierung des ModelExplorers

Das fundamentale ModelExplorer Plug-In beinhaltet insbesondere Funktionalitäten, um eine grafische, baumbasierte Repräsentation des Datenmodells anzuzeigen. So kann für die Darstellung eines Elementes der Baumstruktur ermittelt werden, ob ein Element untergeordnete Elemente enthält und bei Bedarf können die untergeordneten Elemente in einer entsprechenden Datenstruktur herausgefunden werden. Den anzuzeigenden Namen und das Icon können mit weiteren Methoden ausgelesen werden.

Abstrahierung des Properties Views

Ähnlich wie beim ModelExplorer wird für den Properties View ein zusätzliches Plug-In mit grundlegenden Funktionalitäten erstellt. Darin befindet sich unter anderem eine Klasse, welche mehrfach verwendete Vorgänge implementiert, wie zum Beispiel die generische Manipulation eines Elementes und die Übertragung der Modifikation zur Datenquelle. Zudem befinden sich in dem Plug-In neben weiteren abstrahierten Bestandteilen die Filter-Klassen für die Filterung der Ansicht des Properties View und die Basisklassen der Widgets.

4.2.3 Migration von ModelICE nach Eclipse Che

Dieser Abschnitt befasst sich mit der Migration der ModelICE Komponenten ModelExplorer sowie Properties View nach Eclipse Che. Im Abschnitt 4.2.1.2 wurde bereits die Erstellung von Eclipse Che Plug-Ins beschrieben, sodass dies in den folgenden Abschnitten nicht erwähnt wird. Aufgrund hinzukommender ModelICE-Abhängigkeiten zu den Eclipse Che Assemblies muss der Build-Prozess angepasst werden, indem entsprechende Projektmodelle modifiziert werden. Im Rahmen dieser Arbeit, wird nicht auf diese Änderungen eingegangen.

4.2.3.1 Erstellung des ModelExplorers in Eclipse Che

Nach der Beschreibung der gemeinsam genutzten Schnittstelle für den Austausch von Daten zwischen Klient und Server wird die Implementierung des ModelExplorers differenziert für den Klienten und den Server beschrieben.

Damit die Beschreibung eines Elementes beim Server als auch beim Klient eindeutig ist, wird das Interface `IModelExplorerEntry` (vgl. Listing 4.3) erstellt, welches ein Objekt des `ElementDescriptors` (siehe Abschnitt 4.2.2.2) aggregiert und die API dieser Klasse im Sinne eines Proxies auf ein implementiertes `IModelExplorerEntry` Objekt abbildet. Als zusätzliche Information für die Baumdarstellung des ModelExplorers in GWT wird ein Wahrheitswert benötigt, welcher angibt, ob das aktuelle Element untergeordnete Elemente besitzt.

```
1  public interface IModelExplorerEntry{
2      String getElementID();
3      String getLabel();
4      String getElementType();
5      String getElementIcon();
6      boolean isLeaf();
7      void setLeaf(boolean isLeaf);
8      ElementDescriptor getElement();
9      void setElement(ElementDescriptor element);
10 }
```

Listing 4.3: Das Interface IModelExplorerEntry für den gemeinsamen Datenaustausch

ModelExplorer: Klient - Erweiterung

Für die Darstellung des baumbasierten ModelExplorers muss ein dynamischer Baum entwickelt werden, welcher beim Expandieren eines Elementes asynchrone RPC-Anfragen an den Server schickt. Realisiert wird diese Anforderung mithilfe des GWT Oberflächenelementes *„CellTree“*[30]. Bei der Instanziierung des CellTrees muss, ähnlich einem JFace-Viewer aus Abschnitt 2.1.7, ein Datenmodell angegeben werden. In diesem Fall wird das Datenmodell mithilfe des Interfaces `TreeViewModel` entwickelt und muss somit eine Implementierung der beiden Methoden `getNodeInfo(T value)` und `isLeaf(Object value)` beinhalten, welche im Listing 4.4 dargestellt sind.

```
1  public class ModelExplorerTreeModel implements TreeViewModel {
2  ...
3      @Override
4      public <T> NodeInfo<?> getNodeInfo(T value) {
5          String parentID = null;
6          if (value instanceof IModelExplorerEntry) {
7              ModelExplorerEntry entry = (ModelExplorerEntry) value;
8              parentID = entry.getId();
9          }
10         ModelExplorerCell cell = new ModelExplorerCell();
11         return new DefaultNodeInfo<IModelExplorerEntry>(
               new ModelExplorerDataProvider(parentID,
                   modelExplorerServiceHelper),
               cell, selectionModel, null);
12     }
13     @Override
14     public boolean isLeaf(Object value) {
15         return value == null ? false :
   ((ModelExplorerEntry)value).isLeaf();
16     }
17 ...
18 }
```

Listing 4.4: Darstellung der Implementierung des TreeViewModel

Für die Darstellung eines neuen Elementes wird die Methode `getNodeInfo(T value)` aufgerufen, um die Repräsentation des entsprechenden Elementes zu ermitteln, welche mithilfe des Interfaces `NodeInfo` und der Implementierung `DefaultNodeInfo` gespeichert wird. Der Parameter `value` entspricht dem darzustellendem Element. Wenn es sich um eine Instanz eines ModelExplorersEntries handelt, wird die ID des Elementes zwischengespeichert (siehe Listing 4.4, Z. 5 - 9). Diese ID wird anschließend mit den erwähnten asynchronen Anfragen an den Server des ModelExplorers übertragen, um dessen untergeordneten Elemente zu ermitteln und das Datenmodell sowie die Baumdarstellung entsprechend zu aktualisieren.

Die Repräsentation eines ModelExplorerEntries wird in der `ModelExplorerCell` Klasse mithilfe eines HTML-Templates (siehe Listing 4.5, Z.4) festgelegt. Dieses Template wird vorkompiliert, sodass

[30] http://www.gwtproject.org/javadoc/latest/com/google/gwt/user/cellview/client/CellTree.html

zur Laufzeit für das etwaige Element die ID, das Label und der Pfad zu einem Icon eingesetzt werden können.

```
1  public class ModelExplorerCell extends AbstractCell<IModelExplorerEntry> {
2  ...
3      private interface ModelExplorerTextCellTemplate extends
   SafeHtmlTemplates {
4          @Template("<div id=\"{0}\"><img src=\"{2}\" width=\"16\"" +
5              "height=\"16\" border=\"0\"/>{1}</div>")
6          SafeHtml cellWithIconAndLabel(String id, String label,
               String iconPath);
7      }
8  ...
9  }
```

Listing 4.5: HTML-Template eines ModelExplorerEntries

Die asynchronen Anfragen zum Server des ModelExplorers werden von dem CellTree innerhalb der `ModelExplorerDataProvider` Klasse gestartet, welche im Listing 4.6 dargestellt wird.

```
1  public class ModelExplorerDataProvider extends
   AbstractDataProvider<IModelExplorerEntry> {
2  ...
3      @Override
4      protected void onRangeChanged(HasData<IModelExplorerEntry> display) {
5          modelExplorerServiceHelper.getChildren(parentID).then(children -> {
6              Range range = display.getVisibleRange();
7              int start = range.getStart();
8              updateRowData(start, children);
9          }).catchError(error -> {
10             modelExplorerServiceHelper.getNotificationManager().
                   notify("Failed to load new Entries",
                       StatusNotification.Status.FAIL,
                       StatusNotification.DisplayMode.FLOAT_MODE);
11         });
12     }
13 ...
14 }
```

Listing 4.6: Darstellung der Klasse `ModelExplorerDataProvider` mit asynchronen RPC-Anfragen an den Server

Der `ModelExplorerServiceHelper` (siehe Listing 4.7) bündelt Anfragen an den Server des ModelExplorers und erstellt aus den vom Server übermittelten Daten im JSON-Format `ModelExplorerEntry`-Instanzen auf dem Klienten. Die vom `ModelExplorerDataProvider` verwendete `getChildren` Methode des `ModelExplorerServiceHelpers` gibt ein Objekt der Klasse `Promise` zurück. Mithilfe eines Promises wird ein Platzhalter für Quelltext zur Verfügung gestellt, welcher nach Erhalt von Ergebnissen einer Aktion zu einem zeitlich späteren, unbekanntem

Zeitpunkt ausgeführt wird. In diesem Fall wird auf eine Antwort vom ModelExplorer Server gewartet und im Erfolgsfall die Methode `.then()` (Listing 4.6, Z. 5 - 9) ausgeführt, welche das Datenmodell mit den vom Server übertragenen Daten aktualisiert. Im Fehlerfall wird die Methode `.catchError()` (Listing 4.6, Z. 9 – 11) aufgerufen, welche in dieser Implementierung eine Fehlermeldung für den Benutzer anzeigt.

```
1  public class ModelExplorerServiceHelper {
2  ...
3      public Promise<List<IModelExplorerEntry>> getChildren(String id) {
4          String url = appContext.getWsAgentServerApiEndpoint()
               + "/modelexplorer/children/" + id;
5          return asyncRequestFactory.createGetRequest(url)
               .loader(loaderFactory.newLoader())
               .send(modelExplorerEntryListUnmarshaller);
6      }
7  ...
8  }
```

Listing 4.7: Die Klasse `ModelExplorerServiceHelper` für die Erstellung einer asynchronen Anfrage an den Server

ModelExplorer: Server - Erweiterung

Wenn asynchrone Anfragen des Klienten gestartet werden, sendet der Server des ModelExplorers in Abhängigkeit der übermittelten ID eines Elementes alle untergeordneten Elemente in Form von ModelExplorerEntries. Ist die ID `null`, was bei der ersten Anfrage des CellTrees eintritt, wird der ModelExplorer initialisiert und sendet dem Klienten alle direkt unterhalb des Projektes beinhalteten UML-Modelle. Wird jedoch eine gültige ID übertragen, werden die entsprechenden Klassen der in Abschnitt 4.2.2.2 beschriebenen Abstrahierung des ModelExplorers verwendet, um zu ermitteln, welche untergeordneten Elemente existieren und ob diese untergeordneten Elemente ihrerseits wiederum untergeordnete Elemente besitzen.

Aufgrund der geltenden Geheimhaltungsvereinbarungen mit dem Fraunhofer Institut FOKUS kann an dieser Stelle nicht auf weitere Details eingegangen werden.

4.2.3.2 Erstellung des Properties Views in Eclipse Che

Nach der Beschreibung der gemeinsam genutzten Schnittstelle für den Austausch von Daten zwischen Klient und Server (siehe Abbildung 4.4) wird die Implementierung des Properties View differenziert für den Klienten und den Server beschrieben.

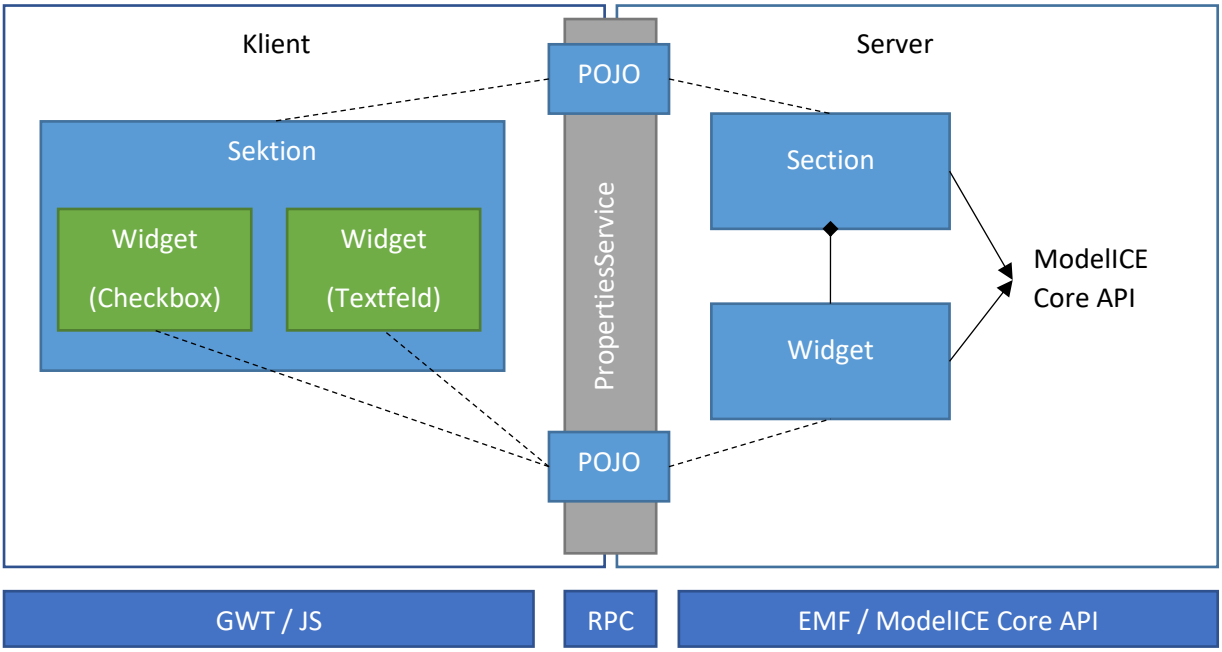


Abbildung 4.4: Trennung des Klienten und Servers mit Austausch der POJOs über den PropertiesService

Damit der Klient die Sektionen mit deren beinhalteten Widgets darstellen kann wird eine asynchrone Anfrage mittels `PropertiesService` an den Server des Properties View gesendet. Die ModelICE Core API erstellt daraus die entsprechenden Instanzen der Sektionen und Widgets und überträgt diese mithilfe von POJOs an den Klienten. Anschließend erstellt der Klient auf Basis der POJOs die Sektionen mit deren Widgets.

Aufgrund der zahlreichen Klassen im Klienten werden an dieser Stelle keine Quelltextbeispiele gezeigt. Diese können jedoch auf dem beigelegten Datenträger angeschaut werden

Properties View: Klient - Erweiterung

Für die Darstellung des Properties View in GWT muss die reiterbasierte Struktur mit der Aufteilung in Sektionen und Widgets prototypisch entwickelt werden. Nachfolgend wird die Realisierung dieser Anforderung kompakt beschrieben.

Nach Start der Anwendung wird der Properties View mittels einer asynchronen Anfrage serverseitig initialisiert und liefert die Daten der enthaltenen Elemente eines Projektes mithilfe der beschriebenen POJOs an den Klienten. Die übermittelten Elemente werden in einer `ListBox` dargestellt und bei Selektion eines dieser Elemente wird eine asynchrone Anfrage an den Server mit der entsprechenden ID des selektierten Elementes geschickt. Die Antwort des Servers beinhaltet in den Sektionen und ihren entsprechenden Widgets, alle relevanten Eigenschaften des selektierten Elementes. Besitzt das Elemente Beispielsweise mit Boolean getypte Eigenschaften, wird eine `BooleanPropertySection` erstellt. Anschließend wird für jedes dieser Elemente ein `BooleanWidget` instanziiert, welches zur Darstellung des Booleans eine `CheckBox` verwendet.

Der Benutzer kann mithilfe eines `ToggleButtons` auswählen, ob die einfache oder erweiterte Ansicht des Properties Views dargestellt werden soll. Für diese GWT-Implementierung ist standardmäßig die erweiterte Ansicht aktiviert, sodass alle Eigenschaften und somit keine Filterung der Sektionen angewendet wird. Ein Wechsel zur einfachen Ansicht ist jederzeit möglich.

Properties View: Server - Erweiterung

Der Server übermittelt bei Anfragen des Klienten die Eigenschaften des jeweiligen Elements in Form einer Instanz des POJOs Properties. Dieses Objekt enthält in Abhängigkeit des gewählten Modus die relevanten Elementeigenschaften in den jeweiligen Sektionen beziehungsweise deren Widgets. Die zu erstellenden POJOs werden mithilfe der in Abschnitt 4.2.2.2 beschriebenen Abstrahierung erzeugt und anschließend an den Klienten übertragen.

Aufgrund der geltenden Geheimhaltungsvereinbarungen mit dem Fraunhofer Institut FOKUS kann an dieser Stelle nicht auf weitere Details eingegangen werden.

4.2.4 Aufgetretene Probleme der Migration

Bei der Erstellung der Eclipse Che Plug-Ins traten diverse Build- und Konfigurationsfehler (siehe Tabelle 7.1 im Anhang) auf, sodass die vollständige Migration der ModelICE-Komponenten ModelExplorer und Properties View nicht realisiert werden kann. Für die Evaluation im Rahmen dieser Arbeit ist jedoch die Betrachtung der GWT-Implementierung ausreichend und repräsentativ. Aus diesem Grund werden die ModelICE-Komponenten in jeweils eigenständigen GWT-Applikationen unter Verwendung einer hinzukommenden Abhängigkeit implementiert, welche gemeinsam verwendete Funktionalitäten bündelt. Aufgrund verschiedener Implementierung asynchroner Aufrufe in GWT und der Eclipse Che Plattform werden entsprechende Klassen modifiziert und können auf dem beigelegten Datenträger betrachtet werden.

5 Evaluation mit Auswertung

In diesem Kapitel wird die prototypische Implementierung mit den zunächst definierten Kriterien im Vergleich zu der bisherigen ModelICE Implementierung verglichen und anschließend ausgewertet. Aufgrund der im Abschnitt 4.2.4 erwähnten Build-Fehlern, wird die Evaluation mit den entwickelten GWT-Applikationen anstelle einer Che Anwendung durchgeführt. Da jedoch Eclipse Che das Framework GWT verwendet kann mithilfe dieser Evaluation ein vergleichbares Ergebnis erzielt werden.

5.1 Kriterien

Für die Evaluation des gewählten Migrationsansatzes werden auf Grundlage der Problemanalyse des Abschnitts 3.2 nachfolgend Kriterien bezüglich der Testbarkeit sowie der Performance definiert.

5.1.1 Testbarkeit

Anhand der in Abschnitts 3.2.1 beschriebenen Schwierigkeiten beim Testen der grafischen Oberfläche stellt sich heraus, dass bei dem verwendeten UI-Framework RAP keine Funktionalität zur Identifikation eines HTML-Elementes existiert. Daraus resultierend werden relative Pfade für die Selektion eines zu testenden HTML-Elementes der eingesetzten Test-Frameworks der Software ModelICE eingesetzt. Aufgrund der potentiell zahlreichen Modifikationen des Pfades eines HTML-Elementes durch RAP können diese relativen Pfade obsolet werden.

Daher gilt die Testbarkeit der Software ModelICE als verbessert, wenn HTML-Elemente eindeutig identifiziert und selektiert werden können und die Identifikation bei Benutzerinteraktionen oder einem Neustart der Software unverändert bleibt.

5.1.2 Performance

Die in Abschnitt 3.2.2 beschriebenen Performanceengpässe des RAP Frameworks verlangsamen die Anwendung ModelICE. Gründe hierfür sind der Austausch großer Datenmengen resultierend auf der vollständigen serverseitigen Berechnung der grafischen Benutzeroberfläche, der beschriebene Umgang mit Icons und die Häufigkeit ressourcenintensiver Anfragen an den Server.

Daher gilt die Performance der Software ModelICE als gesteigert, wenn die auszutauschenden Datenmengen sowie die Häufigkeit der Anfragen zwischen Klient und Server reduziert werden können.

5.2 Durchführung

Die Durchführung der Vergleiche des ModelExplorers sowie des Properties View zu der aktuellen ModelICE Implementierung werden mithilfe der eigenständigen GWT Applikationen durchgeführt. Als Webbrowser wird „*Mozilla Firefox*“ [Mozi19] Version 64.0 verwendet und die Messungen werden mithilfe der Firefox eigenen Netzwerkanalyse durchgeführt, womit der Datenverkehr zwischen Klient und Server betrachtet werden kann. Die Messungen der Performance werden auf einem bereitgestelltem Entwicklungslaptop Lenovo Thinkpad T450 durchgeführt.

5.2.1 Testbarkeit

Um die Testbarkeit von ModelICE zu erhöhen, verfügt GWT über die Funktionalität ein HTML-Element eindeutig zu identifizieren und zu selektieren.

In GWT erbt jede Oberflächenkomponente von der Klasse `UIObject`[31], wodurch jeder Komponente unter Verwendung der Methode `ensureDebugID(String id)` eine eindeutige Identifikation zugeordnet werden kann. Diese Methode fügt dem zugrundeliegenden HTML-Element ein ID-Attribut hinzu, welches aus der gewählten Zeichenkette und einem in den Einstellungen konfiguriertem Präfix besteht. Dieses ID-Attribut wird nur verwendet, wenn die GWT-Applikation in einer Testumgebung gestartet wird.

Beim Einsatz in der Produktivumgebung werden den HTML-Elementen die so gesetzten ID-Attribute nicht hinzugefügt, um die zum Client zu übermittelnde Datenmenge zu reduzieren. Um dennoch ID-Attribute in der Produktivumgebung nutzen zu können, ermöglicht die Klasse `UIObject` den direkten Zugriff auf das zugrundeliegende HTML-Element mithilfe der Methode `getElement()`. Repräsentiert wird das HTML-Element durch die Klasse `Element`[32]. Die ID kann durch die Methode `setID(String id)` der Element-Klasse modifiziert werden.

Eine weitere Möglichkeit ist die Erstellung eines HTML Templates, wie es bereits in der Implementierung der baumbasierten Darstellung des ModelExplorer (siehe Listing 4.5, Z. 4) angewendet wird.

Die drei beschriebenen Varianten für das Hinzufügen eines ID-Attributes zu einem HTML-Element dienen als Basis für die Durchführung manueller Tests zur Evaluation der Testbarkeit. Überprüft wird, ob ein ID-Attribut mit den beschriebenen Varianten hinzugefügt werden kann.

5.2.2 Performance

Der Versuchsaufbau sowie die Messungen der Performance werden im Folgenden einzeln für den ModelExplorer sowie für den Properties View beschrieben. Für die Beschreibung des Versuchsaufbaus wird ein von FOKUS erstelltes ModelICE-Referenzprojekt verwendet. Um einen Vergleich zwischen der RAP und der GWT Implementierung zu erstellen, werden für beide Komponenten die Anzahl der Aufrufe, die zu übermittelnden Daten sowie die Gesamtdauer der Anfragen gemessen.

[31] http://www.gwtproject.org/javadoc/latest/com/google/gwt/user/client/ui/UIObject.html
[32] http://www.gwtproject.org/javadoc/latest/com/google/gwt/user/client/Element.html

ModelExplorer

Anhand des Paketes „State Machine Diagrams“ (siehe Tabelle 5.1) mit 74 direkt untergeordneten Elementen werden die Messungen des ModelExplorers durchgeführt.

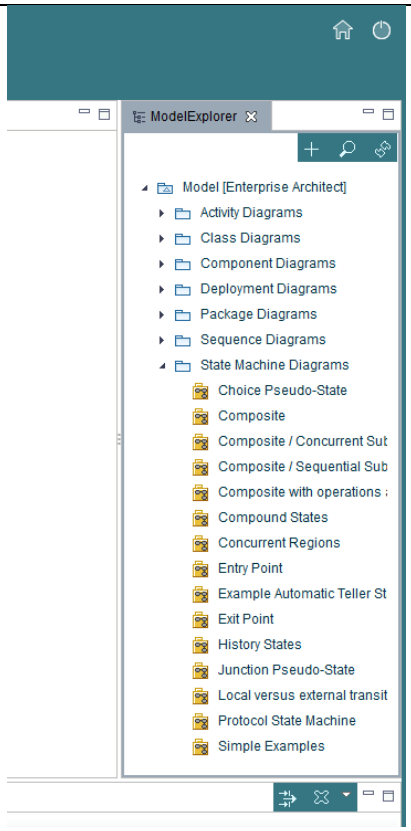
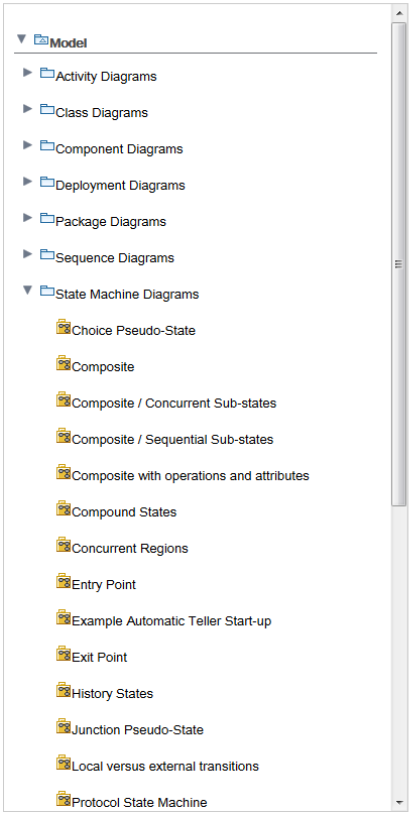
RAP	GWT
	

Tabelle 5.1: Vergleich der Darstellung des ModelExplorers

Properties View

Die Messungen werden anhand der Klasse `ShoppingBasket` durchgeführt, welches sieben untergeordnete Elemente, nämlich sechs Operationen und ein Attribut, beinhaltet (siehe Tabelle 5.2).

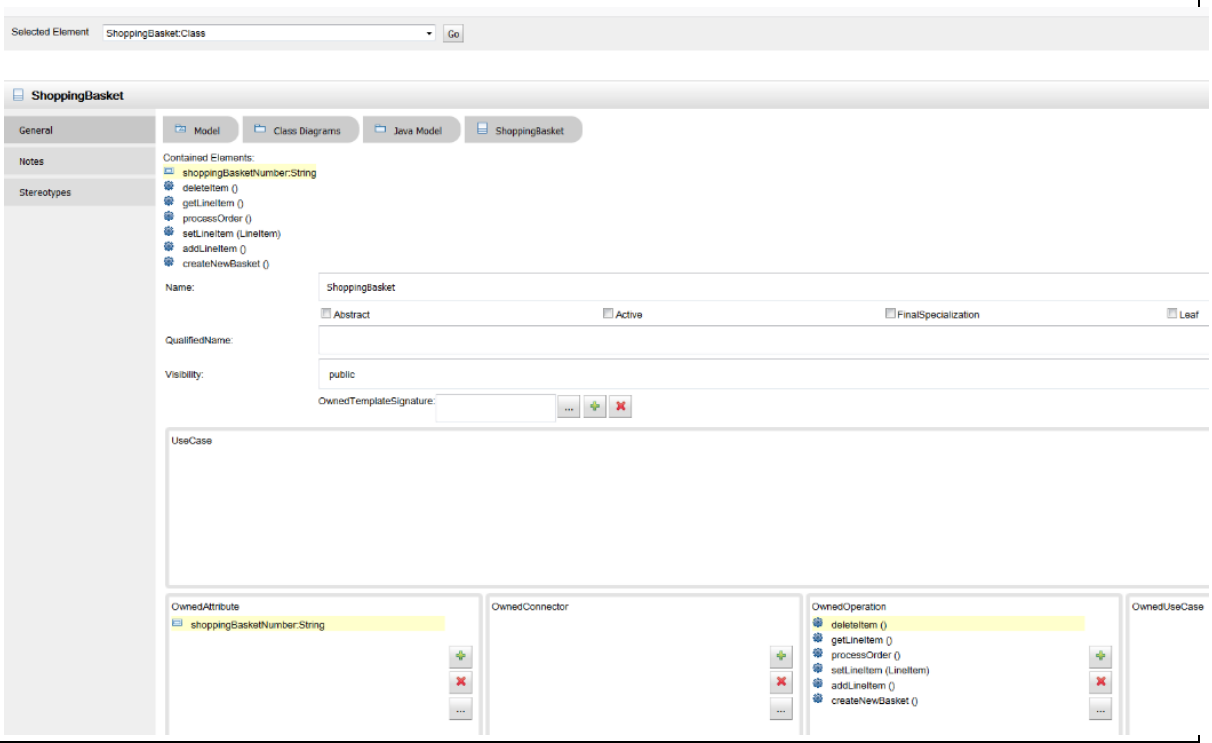
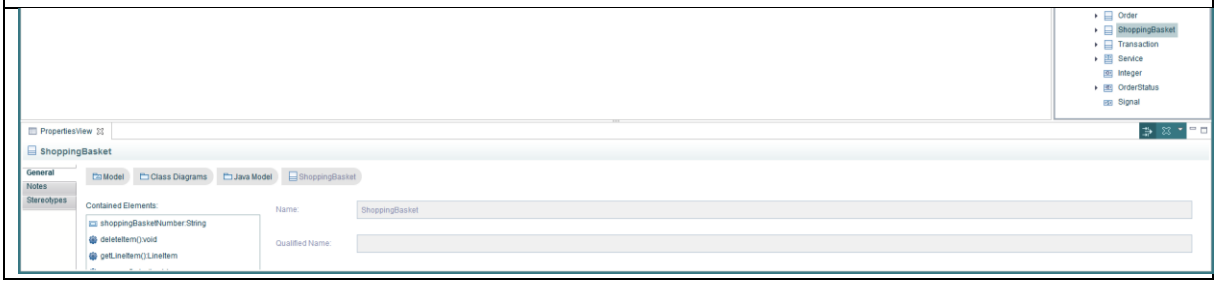
GWT

RAP


Tabelle 5.2: Vergleich der Darstellung des Properties View

5.3 Auswertung

Anhand der Ergebnisse der durchgeführten Evaluation wird die GWT Implementierung im Vergleich zu der bisherigen ModelICE Implementierung mithilfe der festgelegten Kriterien bewertet.

5.3.1 Testbarkeit

Die Verwendung der Funktionalität des GWT-Frameworks, einem HTML-Element ein ID-Attribut hinzuzufügen, ermöglicht die Realisierung automatisiert durchgeführter E2E-Tests mit Test-Frameworks wie Selenium. Im Vergleich zur aktuellen Implementierung von ModelICE können somit alle Komponenten der grafischen Oberfläche getestet werden, sodass die Testbarkeit verbessert wird.

5.3.2 Performance

Tabelle 5.3 fasst die Ergebnisse aus der vorangegangenen manuellen Messungen der Performance zusammen und zeigt die durchschnittliche Steigerung der Performance.

Interaktion		ModelExplorer			Properties View		
		RAP	GWT	Reduzierung in %	RAP	GWT	Reduzierung in %
Selektion / Aufklappen eines Elementes	Anzahl der Anfragen	3	1	66	6	1	83
	Übertragene Daten in kB	17.6	7.9	55	37.8	4.45	88
	Gesamtdauer der Anfragen in ms	386	272	30	806	117	86

Tabelle 5.3: Gemessene Daten der GWT und RAP Implementierungen bei der Selektion eines Elementes

Durch die GWT Implementierung des ModelExplorers können sowohl die zu übertragenden Daten um 55% Daten als auch die Gesamtdauer der Anfragen um ein Drittel verkürzt werden.

Auch die Implementierung des Properties View in GWT zeigt eine Steigerung der Performance, aufgrund der Reduzierung der zu übermittelnden Daten sowie die Verringerung der Gesamtdauer aller Anfragen zum Server um etwa 85%.

Zusammenfassend kann eine Steigerung der Performance beider Komponenten erreicht werden. Entscheidend für die Verbesserung ist die Ablösung des Frameworks GWT, da nun keine serverseitigen Berechnungen der grafischen Benutzeroberfläche stattfinden und stattdessen lediglich sämtliche Daten des Modells übertragen werden. Zusätzlich entsteht durch eine Auslagerung mehrfach verwendeter CSS-Regeln in eine CSS-Stylesheet eine Reduktion der zu übermittelnden Daten je HTML-Element. Die zudem getrennt heruntergeladenen Grafiken werden nicht, wie bei RAP überlagert als neue Grafik gespeichert, sondern können vom Klienten unabhängig referenziert und individuell arrangiert werden.

6 Zusammenfassung und Ausblick

Nach einem kompakten Rückblick auf die Arbeit fasst dieses Kapitel das Ergebnis der Migration bezüglich der Testbarkeit sowie der Performance zusammen und erstellt einen Ausblick über mögliche weitergehende Schritte.

6.1 Rückblick

Die Aufgabenstellung dieser Arbeit bestand darin, eine prototypische Implementierung der Software ModelICE zu erarbeiten und einen Prototyp zu entwickeln, um die Testbarkeit und die Performance zu untersuchen. Dabei wurde wie folgt vorgegangen:

Kapitel 2 führte in die Grundlagen der modellgetriebenen Softwareentwicklung ein. Der Fokus lag auf den Konzepten und Technologien der Metamodellierung sowie auf der in einer Eclipse RCP-Anwendung verwendeten Equinox Implementierung der OSGi-Spezifikationen in Verbindung mit dem Eclipse Modeling Framework. Zudem wurde die cloudbasierte Zielplattform Eclipse Che mit den erforderlichen Grundlagen sowie die aktuelle Implementierung des Werkzeugs ModelICE beschrieben.

Kapitel 3 beschreibt die Ausgangslage mithilfe der ModelICE-Architektur und fokussiert die Problemanalyse in Hinblick auf die Testbarkeit und die Performance.

Kapitel 4 beschreibt einen Lösungsansatz zur Migration ausgewählter ModelICE Komponenten auf die Zielplattform Eclipse Che. Da Eclipse Che nicht die OSGi-Spezifikationen unterstützt, wurde die Architektur der Software ModelICE entsprechend modifiziert. Zudem wurden für die zu migrierenden Komponenten Abstraktionsschichten hinzugefügt, um ein beliebiges UI-Framework zu verwenden und die aktuell implementierten Funktionalitäten von ModelICE größtmöglich wiederzuverwenden.

Kapitel 5 widmet sich der Evaluation der Migration. Anhand von festgelegten Kriterien wird die neue Implementierung mit der vorherigen verglichen.

6.2 Bewertung

Die Ergebnisse der Evaluation aus Kapitel 5 zeigen eine deutliche Verbesserung der Testbarkeit und signifikante Steigerung der Performance im Vergleich zu der bisherigen ModelICE Implementierung. Nachfolgend werden Testbarkeit und Performance einzeln beurteilt.

Testbarkeit

Insbesondere mithilfe der Funktionalität, des von Eclipse Che verwendeten UI-Frameworks GWT, einem HTML-Element ein ID-Attribut hinzuzufügen, konnte die Testbarkeit der bisherigen Implementierung von ModelICE erhöht werden, da ein zu testendes Element nun eindeutig beispielsweise vom Test-Framework Selenium identifiziert und selektiert werden kann. Dadurch können automatisiert durchgeführte E2E-Tests mit Selenium nicht nur bei dem ModelEditor, sondern auch bei weiteren ModelICE Komponenten entwickelt und ausgeführt werden.

Performance

Durch die Ablösung des RAP Frameworks wird die Performance gesteigert. Insbesondere durch die strikte Trennung zwischen Klient und Server müssen lediglich die Daten von darzustellenden Elementen an den Klient übermittelt werden. Die daraus resultierende Datenmenge und Übertragungszeit sind im Vergleich zu RAP am Beispiel des Properties View zu 85% geringer.

6.3 Ausblick

Mithilfe der GWT-Implementierungen der ModelICE-Komponenten ModelExplorer und Properties View und den Ergebnissen der Evaluation dieser Arbeit konnte erfolgreich gezeigt werden, dass die Migration zu GWT erhoffte Vorteile bezüglich der Testbarkeit und Performance gegenüber der bisherigen Implementierung erfüllt.

Da der Prototyp dieser Arbeit aufgrund zeitintensiver Fehler nicht fertiggestellt werden konnte, soll dies als kommende Anforderung realisiert werden. Nach erfolgreicher Migration des ModelExplorers sowie des Properties View, kann die vollständige Migration mit allen Funktionalitäten der Software ModelICE in Betracht gezogen werden. Die Migration beinhaltet in den ersten Schritten die komplette Migration des ModelExplorer, des Properties Views und als neues Plug-In den ModelEditor.

Der nächste Schritt beinhaltet eine Realisierung automatisiert durchgeführter E2E-Tests von Test-Frameworks, welche die in dieser Arbeit durchgeführten manuellen Tests ablösen.

Eine weitere Option ist die Verwendung der Version 7 von Eclipse Che. Diese wurde auf der Mitte Oktober 2018 stattgefundenen Entwicklerkonferenz „*EclipseCon Europe 2018*“[33] angekündigt und ist ab März 2019 [Stev18] verfügbar. Bei Version 7 wird die Architektur von Eclipse Che verändert, wodurch Auswirkungen auf die Entwicklung von Che Plug-Ins entstehen. Um in der Version 6 Che Plug-Ins den Endbenutzer zur Verfügung zu stellen, musste der Build-Prozess für die gesamte Eclipse Che Anwendung gestartet werden. Es gab keine Möglichkeit ein Plug-In zur Laufzeit hinzuzufügen oder zu entfernen. In der neuen Version 7 wird diese Schwierigkeit durch ein dynamisches Plug-In Modell gelöst, indem das UI -Frameworks GWT durch das Framework Theia [Thei00] ausgetauscht wird. Mit Version 7 endet zudem die Unterstützung der Container-Virtualisierung mit Docker. Stattdessen soll OpenShift [Open19] oder Kubernetes [Kube19] verwendet werden.

Ein wesentlicher Vorteil für den Umstieg der Software ModelICE auf Version 7 ist das dynamische Plug-In Modell. Im Vergleich zur Version 6 kann damit die Build-Dauer der Che Plug-Ins verkürzt und somit der Build-Prozess von ModelICE beschleunigt werden.

Aus Sicht des Autors ist eine Migration von ModelICE zur cloudbasierten Eclipse Che Plattform vorteilhaft. Es sollte jedoch mit der vollständigen Migration auf die kommende Che Version 7 gewartet werden, da das dynamische Plug-In Modell essentielle Vorteile verspricht. In der noch verbleibenden Zeit zur Version 7 sollte die Abstrahierung vom verwendeten Framework RAP bisheriger ModelICE-Komponenten vervollständigt werden, um eine möglichst reibungslose Migration dieser Komponenten auf das UI-Framework Theia zu ermöglichen.

[33] https://www.eclipsecon.org/europe2018

7 Anhang

7.1 Literaturverzeichnis

[Abou00a] *About OMG | Object Management Group*. URL https://www.omg.org/about/index.htm - abgerufen am 14.01.2019

[Abou00b] *About the XML Metadata Interchange Specification Version 2.5.1*. URL https://www.omg.org/spec/XMI/ - abgerufen am 14.01.2019

[Apac00] *Apache Maven Assembly Plugin – Introduction*. URL http://maven.apache.org/plugins/maven-assembly-plugin/ - abgerufen am 09.10.2018

[Augs18] AUGSTEN, STEPHAN: *Was ist ein Build?* URL https://www.dev-insider.de/was-ist-ein-build-a-702737/ - abgerufen am 01.11.2018

[Bach09] BACHMANN, KAI UWE: *Maven 2: Eine Einführung, aktuelle zur Version 2.0.9* : Addison-Wesley, 2009 — ISBN 978-3-8273-2835-9

[ÇEGH18] ÇELIK, TANTEK ; ETEMAD, ELIKA J. ; GLAZMAN, DANIEL ; HICKSON, IAN ; LINSS, PETER ; WILLIAMS, JOHN: *Selectors Level 3*. URL https://www.w3.org/TR/2018/PR-selectors-3-20180911/ - abgerufen am 15.10.2018

[ChBu17] CHAN, SHING WAI ; BURNS, ED: Java™ Servlet Specification (2017), S. 246

[ChSB07] CHRIS, RUPP ; STEFAN, QUEINS ; BARBARA, ZENGLER: *UML 2 Glasklar, Praxiswissen für die UML-Modellierung*. 3. Aufl. München : Carl Hanser Verlag München Wien, 2007 — ISBN 978-3-446-41118-0

[DFKK05] D'ANJOU, JIM ; FAIRBROTHER, SCOTT ; KEHN, DAN ; KELLERMANN, JOHN ; MCCARTHY, PAT: *The Java™ Developer's Guide to Eclipse*. 2nd Edition. : Addison Wesley, 2005 — ISBN 978-3-8273-2254-8

[Dock00] *Docker*. URL https://www.docker.com/index.html - abgerufen am 14.01.2019

[Ecli00a] *Eclipse Packages | The Eclipse Foundation - home to a global community, the Eclipse IDE, Jakarta EE and over 350 open source projects...* URL https://www.eclipse.org/downloads/packages/ - abgerufen am 06.01.2019

[Ecli00b] *Eclipse Che | Eclipse Next-Generation IDE, Cloud IDE, and Workspace Server*. URL https://www.eclipse.org/che/ - abgerufen am 14.01.2019

[Ecli00c] *Eclipse Che | Features*. URL https://www.eclipse.org/che/features/ - abgerufen am 09.10.2018

[Ecli00d] *Eclipse Che Documentation | Eclipse Che Documentation*. URL https://www.eclipse.org/che/docs/ - abgerufen am 14.01.2019

[Ecma17] *ECMA-404*. URL http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf - abgerufen am 05.10.2018. — The JSON Data Interchange Sytnax

[Egit00] *EGit | The Eclipse Foundation*. URL https://www.eclipse.org/egit/ - abgerufen am 23.11.2018

[Equi00] *Equinox | The Eclipse Foundation*. URL https://www.eclipse.org/equinox/ - abgerufen am 14.01.2019

[Eule05] EULER, STEPHAN: *Know-how: Kommunikation mit Remote Procedure Calls*. URL https://www.tecchannel.de/a/know-how-kommunikation-mit-remote-procedure-calls,402428 - abgerufen am 14.01.2019

[Exte08] *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. URL https://www.w3.org/TR/xml/ - abgerufen am 14.01.2019

[Git00] *Git*. URL https://git-scm.com/ - abgerufen am 14.01.2019

[Gron00] GRONBACK, RICHARD: *Eclipse Modeling Project | The Eclipse Foundation*. URL https://www.eclipse.org/modeling/emf/ - abgerufen am 14.01.2019

[Guin00] GUINDON, CHRISTOPHER: *SWT: The Standard Widget Toolkit | The Eclipse Foundation*. URL https://www.eclipse.org/swt/ - abgerufen am 14.01.2019

[Gwtd00] *GWT Documentation Overview*. URL http://www.gwtproject.org/doc/latest/DevGuide.html - abgerufen am 09.10.2018

[Gwtp00] *GWT Project*. URL http://www.gwtproject.org/ - abgerufen am 14.01.2019

[Hoff13] HOFFMANN, DIRK W.: *Software-Qualität*. 2. Berlin : Springer Vieweg, 2013 — ISBN 978-3-642-35699-5

[Hunt06] HUNTER, ANTHONY: *The Eclipse Tabbed Properties View*. URL https://www.eclipse.org/articles/Article-Tabbed-Properties/tabbed_properties_view.html - abgerufen am 13.01.2019

[Ibm-19] *IBM - Deutschland*. URL https://www.ibm.com/de-de/ - abgerufen am 14.01.2019

[InFr00] INGRANO SOLUTIONS ; FRAUNHOFER FOKUS: *ModelICE - Your Ideas To Solutions*. URL http://www.modelice.io/ - abgerufen am 12.11.2018

[Jfac00] *JFace - Eclipsepedia*. URL https://wiki.eclipse.org/JFace - abgerufen am 14.01.2019

[Kube19] *Kubernetes* : Cloud Native Computing Foundation, 2019

[Mave00a] *Maven - Concierge OSGi - An optimized OSGi R3 implementation for mobile and embedded systems - Overview*. URL http://concierge.sourceforge.net/ - abgerufen am 14.01.2019

[Mave00b] *Maven Repository: Search/Browse/Explore*. URL https://mvnrepository.com/ - abgerufen am 01.11.2018

[Mili00] MILINKOVICH, M.: *About the Eclipse Foundation | The Eclipse Foundation*. URL https://www.eclipse.org/org/ - abgerufen am 14.01.2019

[Mosi17] MOSIG, JAN: *Schreibe niemals Unit Tests!* URL https://blogs.itemis.com/de/schreibe-niemals-unit-tests - abgerufen am 26.11.2018

[Mozi19] *Mozilla Firefox*. URL https://www.mozilla.org/de/firefox/ - abgerufen am 13.01.2019. — Dein neuer, schneller Browser für Mac, PC und Linux | Firefox

[Mvc00] *MVC*. URL http://wiki.squeak.org/squeak/1767 - abgerufen am 05.01.2019

[Mvca00] *MVC architecture*. URL https://developer.mozilla.org/en-US/docs/Web/Apps/Fundamentals/Modern_web_app_architecture/MVC_architecture - abgerufen am 14.01.2019. — MDN Web Docs

[Open19] *OpenShift: Container Application Platform by Red Hat, Built on Docker and Kubernetes* : Red Hat, 2019

[Orac00] *Oracle | Integrated Cloud Applications and Platform Services*. URL https://www.oracle.com/index.html - abgerufen am 14.01.2019

[Osgi18] *OSGi Core*. URL https://osgi.org/download/r7/osgi.core-7.0.0.pdf

[Ou10] OU, OWEN: *Introducing Eclipse RAP*. URL https://owenou.com/introducing-eclipse-rap - abgerufen am 12.11.2018. — Owen Ou's Blog

[Plug00] *Plug-ins and bundles*. URL https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fruntime_model_bundles.htm - abgerufen am 01.11.2018

[Proj18] *Project natures - Eclipse Platform*. URL https://help.eclipse.org/2018-12/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2FresAdv_natures.htm - abgerufen am 06.01.2019

[Rap/00a] *RAP/RWT - Eclipsepedia*. URL https://wiki.eclipse.org/RAP/RWT - abgerufen am 14.01.2019

[Rap/00b] *RAP/Protocol - Eclipsepedia*. URL https://wiki.eclipse.org/RAP/Protocol - abgerufen am 11.01.2019

[Rap-00] *RAP - Scopes in RAP*. URL https://www.eclipse.org/rap/developers-guide/devguide.php?topic=scopes.html&version=3.6 - abgerufen am 11.01.2019

[Requ16] *Requirements Interchange Format Specification Version 1.2*. URL https://www.omg.org/spec/ReqIF/About-ReqIF/ - abgerufen am 10.10.2018

[Rich00] *Rich Client Platform - Eclipsepedia*. URL https://wiki.eclipse.org/Rich_Client_Platform - abgerufen am 14.01.2019

[RoDS00] ROBIE, JONATHAN ; DYCK, MICHAEL ; SPIEGEL, JOSH: *XML Path Language (XPath) 3.1*. URL https://www.w3.org/TR/xpath-31/ - abgerufen am 14.01.2019

[SBPM08] STEINBERG, DAVE ; BUDINSKY, FRANK ; PATERNOSTRO, MARCELO ; MERKS, ED: *EMF: Eclipse Modeling Framework, 2nd Edition*, *The Eclipse Series*. 2nd Edition. : Addison-Wesley Professional, 2008 — ISBN 0-321-33188-5

[Scal01] *Scalable Vector Graphics (SVG) 1.0 Specification*. URL https://www.w3.org/TR/2001/REC-SVG-20010904/ - abgerufen am 14.01.2019

[Sele00] *Selenium - Web Browser Automation*. URL https://docs.seleniumhq.org/ - abgerufen am 15.10.2018

[Spar00] *Sparx Systems UML Modellierung mit Enterprise Architect*. URL https://www.sparxsystems.de/start/startseite/ - abgerufen am 05.12.2018. — SparxSystems Europe

[SpLi05] SPILLNER, ANDREAS ; LINZ, TILO: *Basiswissen Softwaretests Aus- und Weiterbildung zum Certified Tester – Foundation Level nach ISTQB-Standard*. 3. Aufl. : dpunkt, 2005 — ISBN 3-89864-358-1

[Stev18] STEVAN, LEMEUR: *Eclipse Che 7 is Coming and It's Really Hot (4/4)*. URL https://developers.redhat.com/blog/2018/12/21/eclipse-che-7-is-coming-and-its-really-hot-4-4/ - abgerufen am 05.01.2019. — RHD Blog

[Swtb00] *SWTBot | The Eclipse Foundation*. URL https://www.eclipse.org/swtbot/ - abgerufen am 15.10.2018

[Thea00] *The Apache Software Foundation*. URL http://www.apache.org/foundation/ - abgerufen am 11.10.2018

[Thei00] *Theia - Cloud and Desktop IDE*. URL https://www.theia-ide.org/ - abgerufen am 05.01.2019

[Unif15] *Unified Modeling Language Specification Version 2.5*. URL https://www.omg.org/spec/UML/2.5/About-UML/ - abgerufen am 07.01.2019

[WeOe06] WEILKIENS, TIM ; OESTEREICH, BERND: *UML 2 Zertifizierung: Fundamentl, Intermediate und Advanced*. 1. : dpunkt, 2006 — ISBN 3-89864-424-3

[What00] *What is a Container*. URL https://www.docker.com/resources/what-container - abgerufen am 14.01.2019

[Zerr10] ZERR, ANGELO: *Eclipse Extension Points and Extensions without OSGi*. URL https://angelozerr.wordpress.com/2010/09/14/eclipse-extension-points-and-extensions-without-osgi/ - abgerufen am 30.11.2018. — Angelo's Blog

7.2 Abbildungsverzeichnis

7.3 Tabellenverzeichnis

7.4 Listingverzeichnis

7.5 Fehlerauflistung der Migration von ModelICE zu Eclipse Che

ID	Problem	Gelöst?	Fehlerbehebung
P1	Auflösung von Maven Abhängigkeiten	Ja	Hinzufügung der entsprechenden Abhängigkeiten in den Konfigurationsmodellen.
P2	Keine Kompilierung von generischen Klassen von ModelICE-Abhängigkeiten	Ja	Verwendung der gleichen Java Version mit der die ModelICE-Abhängigkeiten erstellt wurden beim Build von der Che Anwendung den Eclipse Compiler.
P3	*„IllegalArgumentException“* beim Build-Vorgang des ModelExplorer Server Plug-Ins.	Nein	Keine Lösung bekannt, auch nicht durch Nachfrage im Mattermost Kanal und der Mailingliste von Eclipse Che. Konnte mit der Neuerstellung einer Eclipse Che Anwendung mit ModelICE Plug-Ins gelöst werden.
P4	Nicht valides Eclipse Che Verzeichnis beim Start der Anwendung durch Docker	Ja	Deinstallation und erneute Installation von Docker
P5	Der Start durch Docker scheitert aufgrund fehlender und benötigter Skript-Dateien.	Ja	Hinzufügen des Argumentes *„--skip:scripts“*
P6	Nicht erfolgreicher Start der Che Anwendung durch Docker.	Ja	Fehlende Zugriffsberechtigung in der entsprechenden Datei des Che Servers (ws-agent-server) ergänzt.
P7	Der Workspace einer Eclipse Che mit ModelICE Plug-Ins startete nicht.	Nein	Neue Erstellung der Eclipse Che Grundstruktur und erneutes Hinzufügen der ModelICE Plug-Ins.
P8	*„SecurityException“*: java.lang.SecurityException: class "org.eclipse.core.runtime.CoreException"'s signer information does not match signer information of other classes in the same package java.lang.SecurityException: class "org.eclipse.core.runtime.IStatus"'s signer information does not match signer information of other classes in the same package Die angegebenen Klassen werden aus verschiedenen Jars verwendet.	Ja	Ermittlung der unterschiedlichen Jars und Entfernung entsprechender Abhängigkeiten im dazugehörigen Konfigurationsmodell.

Tabelle 7.1: Fehlerauflistung der Migration von ModelICE zu Eclipse Che

7.6 Softwareverzeichnis

Alle Links wurden am Montag, 14. Januar 2019 auf ihre Richtigkeit überprüft.

C

Conciergo ..... http://concierge.sourceforge.net/

D

Docker ..... https://www.docker.com/

E

Eclipse Che ..... https://www.eclipse.org/che/
Eclipse Modeling Framework ..... http://www.eclipse.org/modeling/emf/
Enterprise Architect ..... https://www.sparxsystems.de/start/startseite/

G

Google Web Toolkit ..... http://www.gwtproject.org/

J

Java ..... https://www.java.com/de/
JFace ..... https://wiki.eclipse.org/JFace

K

Keycloak ..... https://www.keycloak.org/
Kubernetes ..... https://kubernetes.io/

M

Maven ..... http://maven.apache.org/
ModelICE ..... http://www.modelice.io/

O

OpenShift ..... https://www.openshift.com/

R

RAP Widget Toolkit ..... https://wiki.eclipse.org/RAP/RWT
Remote Application Platform ..... http://www.eclipse.org/rap/
Rich Client Platform ..... https://wiki.eclipse.org/Rich_Client_Platform

S

Selenium ..... https://www.seleniumhq.org/
Standard Widget Toolkit ..... https://www.eclipse.org/swt/
SWTBot ..... http://www.eclipse.org/swtbot/

T

Theia ..... https://www.theia-ide.org/index.html#features