CARL VON OSSIETZKY universität OLDENBURG

Department für Informatik – Abteilung Softwaretechnik

Dokumentation

Projektgruppe Kollaborative Modellierung

Inhaltsverzeichnis

Teil I

Zielvereinbarung

Kapitel 1

Zielvereinbarung

1.1 Motivation

UML-Modelle sind essenziell für die Planung und Entwicklung in Softwareprojekten, da sie die Kommunikation mit diversen Stakeholdern, wie z.B. Kunden und Entwicklern, vereinfachen.

Beim Erstellen von Modellen modelliert jeder Entwickler nach seinen eigenen Vorstellungen. Diese können zwischen einzelnen Mitgliedern eines Teams, das an einem Modell arbeitet, teilweise stark auseinandergehen. Zudem werden besonders in frühen Phasen des Entwurfsprozesses von Software Modellelemente in schneller Reihenfolge erstellt und bearbeitet [WLS+13]. Daher ist es unerlässlich, dass Änderungen von allen Beteiligten möglichst schnell gesehen werden können. Der klassische Ansatz, ein Versionskontrollsystem (VCS), z.B. Git oder SVN, zu verwenden, um die Arbeit an mehreren Standorten zu synchronisieren, ist dabei nicht ausreichend. Der Arbeitsablauf (manuelles Check-Out und Check-In), den diese Tools vorsehen, ist für eine Synchronisation in Echtzeit nicht geeignet.

Um effizientes verteiltes Arbeiten zu ermöglichen, ist es also notwendig, dass Teammitglieder gleichzeitig an den selben Modellen arbeiten können. Derzeit verfügbare Tools, wie Cacoo [Nul14], die eine solche Form des kollaborativen Arbeitens ermöglichen, sind reine Diagrammeditoren und keine Modellierungswerkzeuge. Sie kennen die einzelnen Diagrammformen, wie z.B. Rechtecke und Pfeile, jedoch nicht deren Semantik, wie z.B. Klassen und Assoziationen.

Aber bereits die Versionierung von Modellen stellt ein Problem. Versionierung ist z.B. notwendig, um die Modellhistorie nachvollziehen und einen früheren Stand der gemeinsamen Arbeit wieder herstellen zu können. Die klassische Versionierung erfolgt über Versionskontrollsysteme wie z.B. Git, Subversion, CVS oder Mercurial [Bla14]. Diese Versionskontrollsysteme ermöglichen das Verwalten aller Versionen von Dateien, sowie das Zusammenführen mehrerer Versionen (Mergen).

Das Mergen in klassischen VCS erfolgt textbasiert [Git14]. Wenn ein automatisches Mergen nicht möglich ist, muss der Konflikt manuell behoben werden. Manuelles Mergen von Modelldifferenzen ist jedoch kaum möglich, da die Speicherstruktur von Modellen für Menschen nicht nachvollziehbar ist. Die Folge ist, dass der Zugriff auf die Dateien abwechselnd erfolgen muss, was dem Team bereits ab einer Größe von zwei Teilnehmern ein hohes Maß an Koordination aufzwingt. Dabei steigt der Mehraufwand der Koordination exponentiell pro weiterem Teilnehmer.

Dies hat zur Folge, dass Entwickler gemeinsam an einem Rechner ein einziges Modell erstellen, um Konfliktsituationen zu vermeiden. Dabei leidet die Produktivität der Entwickler.

In einem kollaborativen Modellierungswerkzeug ist eine **Modellimplementierung** notwendig, um z.B. die Korrektheit von Modellen zu überprüfen. Ein Modell ist eine Beschreibung eines (realen) Systems, z.B. in Form eines UML-Klassendiagramms. Da Modelle wiederum durch Metamodelle beschrieben werden, ist ein Modell Instanz eines Metamodells. Diagramme, also verschiedene Sichten auf das Modell, werden samt ihrer Layoutinformationen als Schnittstelle zwischen dem Anwender und dem Modell benötigt. Um Diagramme implementieren und versionieren zu können, werden Informationen über Diagramme im Metamodell integriert. Somit beschreibt das Metamodell Modelle samt zu einem Modell dazugehörige Diagramme.

Um die **Kollaborativität** in einem Modellierungswerkzeug zu erzielen, ist eine Modellversionierung notwendig. Diese synchronisiert die Änderungen zwischen den einzelnen Anwendern (Mikroversionierung) und erlaubt den Anwendern auf frühere Versionen zurückzugreifen (Makroversionierung). Die Softwaretechnik-Abteilung der Universität Oldenburg (ST-Abteilung) forscht aktuell unter dem Namen **DOLSA** (Delta Operations Language with Services and Applications) an der effizienten Versionierung von Modellen [KW14].

Diese Forschungsarbeit definiert die Delta Operation Language (DOL) als Metamodell-generische Implementierung von operationsbasierten Diff-Repräsentationen für Modelle. Desweiteren stellt die Softwaretechnik-Abteilung der Projektgruppe einen funktionalen Prototypen zur Verfügung, welcher als Drittkomponente angebunden werden soll. Diese bietet uns zwei Schnittstellen: DOL-Services für die Mikroversionierung und DOL-Application für die Makroversionierung.

Somit soll in der Projektgruppe ein kollaboratives Modellierungstool auf der Basis dieser Forschung entwickelt werden.

1.2 Zielsetzung

Das Ziel des Projekts ist die Entwicklung einer Anwendung zur kollaborativen Modellierung von Klassendiagrammen unter Verwendung von DOL.

Damit dieses Ziel erreicht werden kann, werden vier Teilziele definiert, die das Hauptziel repräsentieren und für dessen Realisierung notwendig sind.

Das erste Teilziel ist die **Modellierung von Klassendiagrammen**. Dafür müssen die Anwender die Möglichkeit haben, Klassendiagramme zu modellieren. Dabei sollen die Klassendiagramme angelehnt an den UML 2.4.1 Standard [Obj11] realisiert werden. Der Umfang wird auf den Inhalt der ST-Vorlesung reduziert und mit den Auftraggebern abgesprochen. Der zu realisierende Umfang wird in einem Metamodell festgehalten (siehe Kapitel 10). Die grafische Oberfläche muss dabei dem Anwender ermöglichen das Modell zu bearbeiten.

Das zweite Teilziel ist die **Echtzeit-Kollaboration**. Damit die Anwender zusammen an einem Modell arbeiten können, müssen die Änderungen eines Anwenders an alle Anwender, die an dem Modell arbeiten, verteilt werden. Dieser Schritt der Synchronisation wird in diesem Dokument als Mikroversionierung bezeichnet und zur seiner Realisierung sollen die DOL-Services aus der DOLSA-Arbeit verwendet werden. Dabei darf das Modell während der Bearbeitung durch das System für andere Anwender nicht gesperrt werden. Jeder Anwender soll jederzeit das Modell verändern können. Damit die Kollaboration in Echtzeit geschieht, muss eine verschickte Änderung unter einer Sekunde bei allen anderen Anwendern ankommen. Dabei sollen Änderungen der einzelnen Anwender voneinander optisch unterschieden werden können.

Das dritte Teilziel ist die **Modellverwaltung**. Damit Anwender an verschiedenen Modellen arbeiten können, muss das System eine Modellverwaltung bereitstellen. Diese muss es dem Anwender ermöglichen, neue Modelle anzulegen und vorhandene Modelle zu löschen. Dies gilt auch für die in den Modellen enthaltenen Diagramme.

Das vierte Teilziel ist die **Versionierung**. Damit Anwender zu jedem Modell eine Änderungshistorie haben können, wird eine Versionsverwaltung benötigt. Dabei müssen die Anwender die Möglichkeit haben den aktuellen Stand eines Modells versionieren können. Dieser Schritt der Versionierung wird in diesem Dokument als Makroversionierung bezeichnet und zur seiner Realisierung soll die DOL-Application der DOLSA-Arbeit verwendet werden. Zusätzlich sollen die Anwender die Möglichkeit haben die vorhandenen Versionen einzusehen und zu diesen zurück zu kehren.

Die Verwendung der DOLSA-Arbeit reduziert den Implementierungsaufwand in der PG und schafft ein Anwendungsfall für die DOL. Gleichzeitig wird der Ansatz mit einem weiteren Diagrammtyp erprobt, Klassendiagramme statt der bisher getesteten Aktivitätsdiagramme.

1.3 Abgrenzung

In diesem Kapitel wird eine Einschränkung in der Funktionalität beschrieben. Diese ist notwendig, da eine vollständige Entwicklung eines kollabaratives Modellierungstools zu umfangreich und in einer PG nicht machbar ist.

Bestimmte Funktionalitäten sind für ein kollaboratives Modellierungstool notwendig, aber im Rahmen der PG zur Realisierung der Hauptfunktionalitäten nicht kritisch. Deswegen werden diese aus Mangel an Zeit nicht umgesetzt oder nur rudimentär behandelt.

Die **Usability des Produktes** wird wie von den verwendeten Frameworks gegeben realisiert. Die Benutzerschnittstelle muss dem Anwender die Möglichkeit zur Verwendung der vorhandenen Funktionalitäten bieten. Genaue Studien zur Analyse und Verbesserung der Usability können aus Mangel an Zeit in der PG nicht bewerkstelligt werden.

Anwenderkommunikation, in Form eines Text- oder Sprachchats, ist für ein marktfähiges kollabaratives Modellierungstool unerlässlich. Im Rahmen der PG wird diese Funktionalität aber nicht umgesetzt, da sie für die Realisierung der Hauptfunktionalität (siehe Kapitel 1.2) nicht vonnöten ist.

Als **Diagrammtyp** zur Modellierung werden Klassendiagramme realisiert. Aufgrund der Kürze der verbleibenden Zeit, welche 5,5 Monate beträgt, können keinen anderen Diagrammtypen umgesetzt werden. Die Evaluation der Hauptziele (siehe Kapitel 1.2) kann mit einem einzigen Diagrammtyp vollzogen werden.

Bei der Synchronisation von Clients während der kollaborativen Modellierung entstehen Konflikte, wenn zwei Nutzer das selbe Element bearbeitet haben [WLS+13]. Ein angemessenes **Konfliktmanagement** benötigt eine Fallunterscheidung und mehrere Auflösungsstrategien. Aufgrund der Kürze der verbleibenden Zeit werden auftretende Konflikte nach einer simplen Strategie, wie z.B. „der erste gewinnt“, aufgelöst. Diese Strategie wie so realisiert, dass sie bei einer Weiterentwicklung durch eine andere Strategie ausgetauscht werden kann.

1.4 Lösungsidee

In diesem Kapitel wird die Lösungsidee grob skizziert. Eine detaillierte Beschreibung der Lösung befindet sich im Architekturdokument.

Das Produkt wird mittels einer Client-Server Architektur umgesetzt (siehe Abbildung 1.1). Der Server bildet dabei den „Single Point of Truth“. D.h. obwohl jeder Client ein Abbild jedes geöffneten Modells einschließlich aller zum Modell gehörenden Diagramme besitzt, dient die Version auf dem Server bei Asynchronität als Referenz und der Server ist für die Persistenz der Modelle (einschließlich der Diagramme) verantwortlich. Dabei werden Mo-

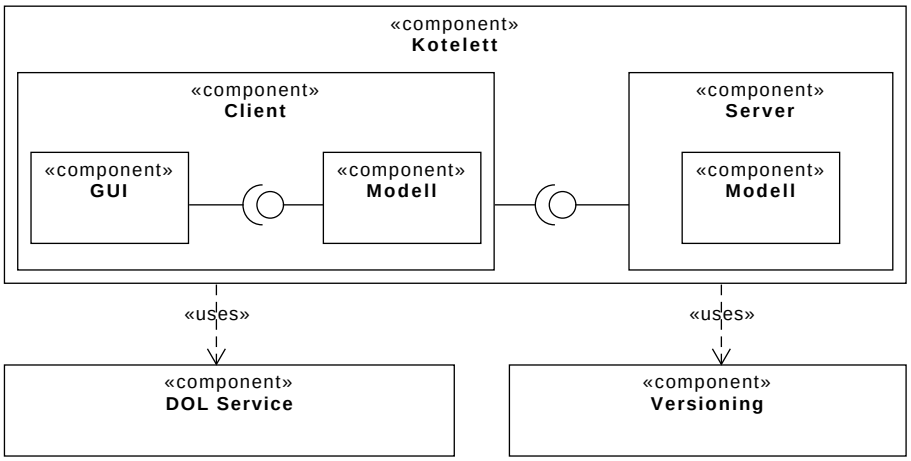

Abbildung 1.1: Abstrakte Architekturübersicht

delle und Diagramme (Layoutinformationen) durch ein gemeinsames Metamodell beschrieben und werden somit permanent mittels einer gemeinsamen Datenstruktur verwaltet.

Der **Client** stellt die zu einem Modell gehörenden Klassendiagramme dem Anwender dar. Zu jedem Modell gibt es mindestens ein Klassendiagramm, der Anwender kann aber beliebig viele Diagramme zu einem Modell anlegen. Diese stellen dann in der Regel verschiedene Sichten auf das Modell dar. Änderungen des Anwenders an einem Diagramm werden direkt auf das Modell übertragen. Wenn andere Diagramme des selben Modells ebenfalls von den Änderungen betroffen sind, wirken diese sich auch dort aus. Da sowohl Layoutinformationen als auch Modellelemente Instanzen des gemeinsamen Metamodells sind, kennt der Client zu einem Diagramm immer das gesamte Modell.

Von dem Anwender ausgelöste Änderungen des Modells (bzw. eines Diagramms) werden an den Server verschickt. Der Server hat ebenfalls eine Version des gleichen Modells (und der Diagramme), auf welchem die Änderungen angewandt werden. Danach wird die Änderung an alle Clients, die an dem gleichen Modell arbeiten, verschickt. Ankommende Änderungen von dem Server werden auf dem Client auf das Modell angewandt und in den Diagrammen visualisiert.

Der **Server** erfüllt somit zwei Hauptaufgaben: die Verwaltung der verschiedenen Modelle (einschließlich ihrer Diagramme) und die Synchronisierung der verschiedenen Clients samt den Modelländerungen, die sie vornehmen.

Eine zusätzliche Aufgabe des Servers besteht darin, die Modelle zu versionieren. Dabei wird der aktuelle Zustand des Modells einschließlich der

Layoutinformationen gespeichert und der Server bietet die Möglichkeit, zu einem älteren Stand des Modells zurückzukehren. Das Speichern geschieht dabei manuell, d.h. der Anwender teilt dem System durch die GUI auf dem Client mit, wenn eine neue Version angelegt werden soll.

Als technische Unterstützung zur Umsetzung des kollaborativen Modellierungwerkzeugs stellt die ST-Abteilung zwei Services aus der DOLSA-Arbeit [KW14] der PG zu Verfügung:

DOL Service wird für die Übersetzung der Änderungen in eine einheitliche Syntax sowie zur Anwendung dieser Änderungen auf das Modell verwendet.

Versioning speichert die Modelle mit den dazugehörige Versionen und ermöglicht Interaktion mit diesen.

1.5 Meilensteine

- Planung (bis Ende November)
  - Metamodell für Klassendiagramme
  - Anforderungskatalog des Produktes
  - Zielvereinbarung (Vision 2.0 der PG)
  - Abstimmung der Funktionalität mit DOLSA
  - Architektur des Produktes
  - Fertige Dokumentation der Planung
- Implementierung der kollaborativen Modellierung (bis 16. Januar mit zwei Wochen Weihnachtsferien)
  - Horizontale Realisierung
  - Realisierung der Client-Server-Architektur
  - Realisierung der Klassendiagramme
  - Realisierung der Synchronisation
  - Verwendung von DOLSA
- Implementierung der Modellverwaltung (bis Ende Februar)
  - Vertikale Erweiterung
  - Realisierung der Modellverwaltung
  - Realisierung der Anwenderverwaltung
  - Realisierung der Versionierung
  - Evaluation von DOLSA
- Abschluss (bis Ende März)

- Vollständige Dokumentation der PG
- Erstellung der Abschlusspräsentation der PG

Die verbleibende Zeit (5,5 Monate) wird in vier Meilensteine eingeteilt. Dies ist möglich, da die verbleibende Zeit überschaubar ist und die zu verwendeten Technologien bis auf die Kommunikationstechnologie feststehen.

1.5.1 Meilenstein: Planung

Bis zum ersten Meilenstein muss eine neue Zielvereinbarung und das Metamodell für Klassendiagramme erstellt werden. Die Zielvereinbarung wird für die Abstimmung der Anforderungen mit den Auftraggebern und für die Planung des Produktes im Team benötigt. Hieraus wird ein Anforderungskatalog für das Produkt erstellt.

Das Metamodell wird von allen Komponenten des Produktes benötigt. Es soll möglichst früh erstellt werden, damit alle Komponenten eine bereits fortgeschrittene Version des Metamodells verwenden können.

Ausgehend von der Zielsetzung und dem Metamodell kann die Schnittstellen zu DOLSA mit Dilshodbek Kuryazov abgestimmt und die detaillierte Architektur für das Produkt erstellt werden. Während des Architekturentwurfs müssen einige kleine Prototypen geschrieben werden, um die Schnittstellen zwischen den einzelnen Komponenten auszuprobieren. Die Schnittstelle und die Architektur werden mit den Auftraggebern abgestimmt und gegebenenfalls angepasst.

Den Abschluss dieser Phase soll eine Dokumentation der bisherigen Aktivitäten und Artefakten in der PG darstellen. Darüber hinaus kann ein Prototyp der GEF-jGraLab Kommunikation in dem Client und die Kommunikation mit den DOLSA-Services vorgestellt werden.

1.5.2 Meilenstein: Implementierung der kollaborativen Modellierung

Bis zum Ende des ersten Implementierungsmeilensteins soll die geplante Client-Server Architektur horizontal umgesetzt werden. Dabei soll der Client, der Server und geplante Kommunikation zwischen diesen grob realisiert werden. Wie bei der horizontalen Implementierung vorgesehen, ist dabei der gesamte Ablauf wichtiger als vollständige Funktionalität.

Der Fokus liegt auf den ersten beiden Teilzielen aus der Zielsetzung (siehe Kapitel 1.2), **Modellierung von Klassendiagrammen** und **Echtzeit-Kollaboration**.

Der Client soll den Anwendern ermöglichen Klassendiagramme zu modellieren und Änderungen an den Server verschicken. Der Server soll die Änderungen der Clients annehmen und die einzelnen Clients mikroversionieren.

Damit die Modellverwaltung, die Versionsverwaltung und die Anwenderverwaltung bereits im Projekt enthalten sind, sollen diese grob implementiert werden. Dabei soll die Realisierung auf dem Server vorgezogen und nur bei ausreichend Zeit auf dem Client fortgeführt werden.

Es sollen auch bereits die DOL-Services aus der DOLSA-Arbeit der ST-Abteilung zur Generierung und Anwendung der Änderungen verwendet werden. Damit dies möglich ist, muss die Implementierung der DOLSA-Arbeit zu Beginn der Implementierung, also am 1.12.2014, vorliegen. Die Mikroversionierung geschieht mit der generierten DOL.

Die Architekturplanung soll während der Implementierung weiter verfeinert werden, damit die Implementierung immer architekturkonform bleibt.

1.5.3 Meilenstein: Implementierung der Modellverwaltung

Bis zum Ende des zweiten Implementierungsmeilensteins soll die Architektur vertikal verfeinert werden. Dabei sollen vor allen der Client in der grafischen Repräsentation und Funktionalität erweitert werden. Wie bei der vertikalen Implementierung vorgesehen, sollen die vorhandenen Abläufe mit zusätzlichen Funktionalität erweitert werden.

Der Fokus liegt auf den letzten beiden Teilzielen aus der Zielsetzung: (siehe Kapitel 1.2) **Modellverwaltung** und **Versionierung**.

Es sollen die fehlenden Modellelemente von Klassendiagrammen realisiert werden. Die Modellverwaltung und die Versionsverwaltung sollen, aufbauend auf der vorbereiteten Implementierung, weiter implementiert werden.

Bei ausreichend Zeit soll die Anwenderverwaltung mit der gefärbten Änderungsdarstellung umgesetzt werden.

Während der Implementierung kann die Verwendung von DOLSA in einem Produktivsystem getestet werden. Es können Zugriffszeiten und Echtzeitverhalten für die Evaluation aufgezeichnet werden. Benötigte Änderungen an DOLSA müssen mit Dilshodbek Kuryazov abgesprochen werden, was wiederum Auswirkungen auf die Architektur haben kann.

Die Architekturplanung soll abschließend verfeinert werden.

1.5.4 Meilenstein: Abschluss

Bis zum Ende der PG soll die Dokumentation für die Abgabe vorbereitet werden. Es muss die bisherige Dokumentation überprüft und gegebenenfalls angepasst werden.

Es soll auch die Erfahrung des Teams mit dem Arbeitsprozess dokumentiert werden. Dabei soll auf gelungene Entscheidungen, Probleme und dazugehörige Lösungsvorschläge eingegangen werden.

Zum Abschluss soll eine Präsentation der Arbeitsergebnisse und der gesammelten Erfahrungen vorgestellt werden.

1.6 Evaluation

Um das Ergebnis des Projektes zu evaluieren, wird die Erfüllung der Hauptanforderungen überprüft. Da ein Kriterium von Anforderungen an Softwaresystemen die Erfüllbarkeit ist, sind diese als Evaluierungskriterien geeignet [PR11].

FA-I **Das System muss dem Anwender die Modellierung von Klassendiagrammen ermöglichen.** Die verfügbaren Sprachmittel sind dabei durch das Metamodell definiert. Die grapische Repräsentation soll sich dabei am UML 2.4.1 Standard orientieren [Obj11]. Als Akzeptanztest soll der Nutzer in der Lage sein die StClock zu modellieren. Dabei soll das Ergebniss semantisch äquivalent zu dem Klassendiagramm in Abbildung 1.2 sein.

FA-II **Das System muss den Zustand eines Modells unter allen bearbeitenden Anwendern in Echtzeit synchronisieren.** Um kollaboratives arbeiten zu ermöglichen, müssen die Clients der Anwender in Echtzeit synchronisiert werden. Ein asynchroner Arbeitsablauf, wie in Git und SVN umgesetzt, genügt für den gewünschten Grad der Kollaborativität nicht. In klassischen Versionskontrollsystemen fügt der Benutzer seine Änderungen manuell dem Repository hinzu, und er muss auch manuell eine neue Version des Repositories abrufen. Beides soll vom System automatisiert geschehen, so dass Änderungen sofort hinzugefügt werden, und dann sofort an alle anderen Bearbeiter verteilt werden.

FA-III **Das System muss es dem Anwender ermöglichen, mehrere Modelle zu verwalten.** Anwender wollen mehrere Modelle erstellen, löschen und bearbeiten können, da bereits verhältnismäßig kleine Softwaresysteme mehrere Modelle benötigen. Diese Anforderung ist zudem für die Erweiterbarkeit des Systems sehr wichtig.

FA-IV **Das System muss es dem Anwender ermöglichen, zu früheren Versionen eines Modells zurückzuspringen.** Anwender in der Softwareentwicklung benötigen Versionierung um auf frühere Versionen zurückzugreifen. Dies kann genutzt werden, um in Fehlerfällen frühere Versionen zu verwenden oder die Modellhistorie nachzuvollziehen.

NFA-I **Die Synchronisierung aus Anforderung FA-II muss unter Verwendung der DOL-Services geschehen.** Der Auftraggeber will, dass die DOL als Austauschformat zur Synchronisierung verwendet wird. Um dies zu realisieren, wird dem Team die „DOL-Service“ Komponente von DOLSA zur Verfügung gestellt.

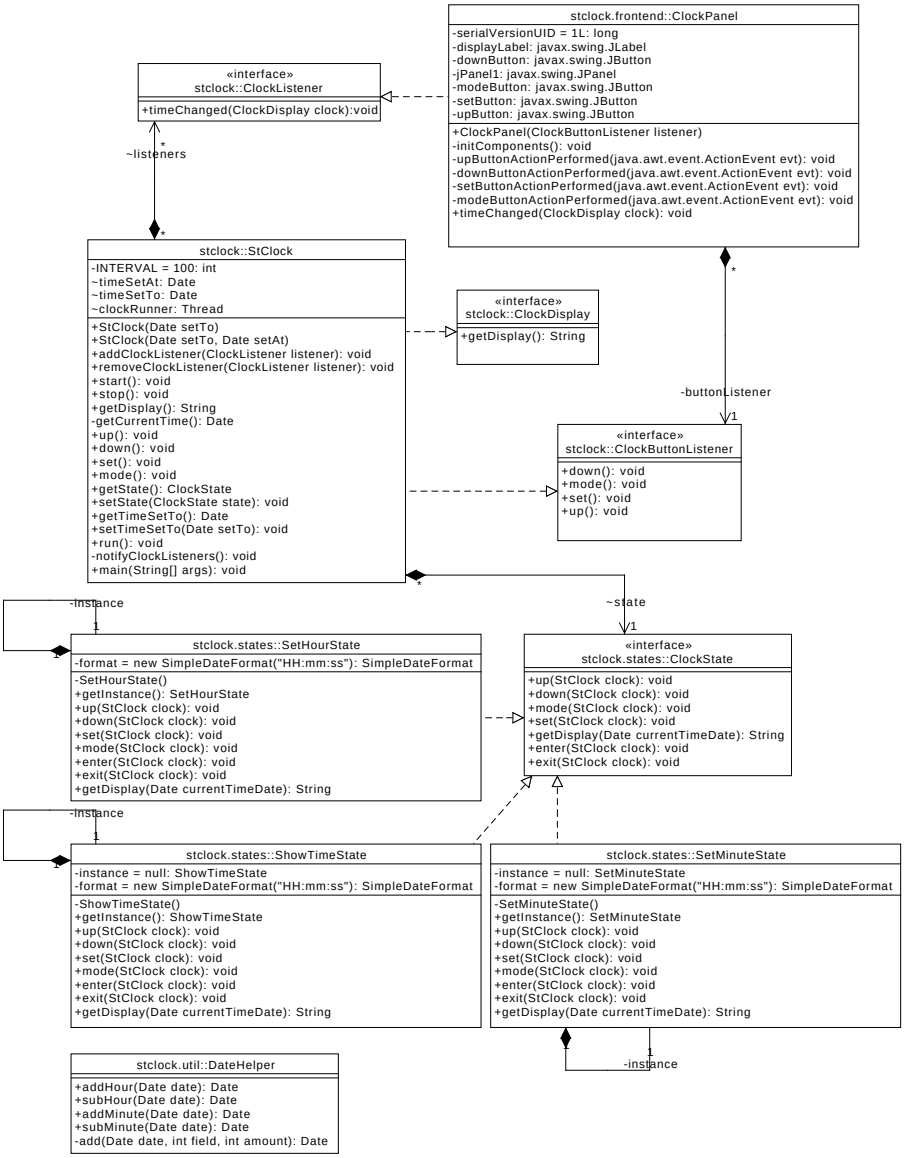

Abbildung 1.2: StClock

NFA-II **Die Modellversionierung aus Anforderung FA-IV muss durch die Einbindung von Versioning erfolgen.** Da Modellversionierung ein aufwendiges Feature ist, wird eine Schnittstelle zum Modellversionierungssystem Versioning entwickelt, womit die PG die Versionierung realisieren darf. Dieses System wird von der ST-Abteilung bereitgestellt.

Auf weitere Akzeptanztests wird verzichtet, da der Kunde selbstständig in der Lage ist die Anforderungen auf Erfüllung zu überprüfen.

1.7 Endergebnis

Die PG wird dem Kunden, der ST-Abteilung, ein kollaboratives Modellierungswerkzeug wie in Abschnitt 1.4 beschrieben, sowie die dazugehörige Prozess- und Produktdokumentation als Artefakte liefern. Nachfolgend werden Software und Dokumente kurz beschrieben.

1.7.1 Software

Die ausgelieferte Software wird aus einem graphischen Modellierungstool für Klassendiagramme (Client) und einem Diagrammserver bestehen. In Abbildung 1.3 ist ein erster Entwurf des Userinterfaces des Clients abgebildet. Dieser Entwurf ist nicht endgültig, sondern stellt den momentanen Stand der Planung dar und soll einen Eindruck vom Endprodukt vermitteln.

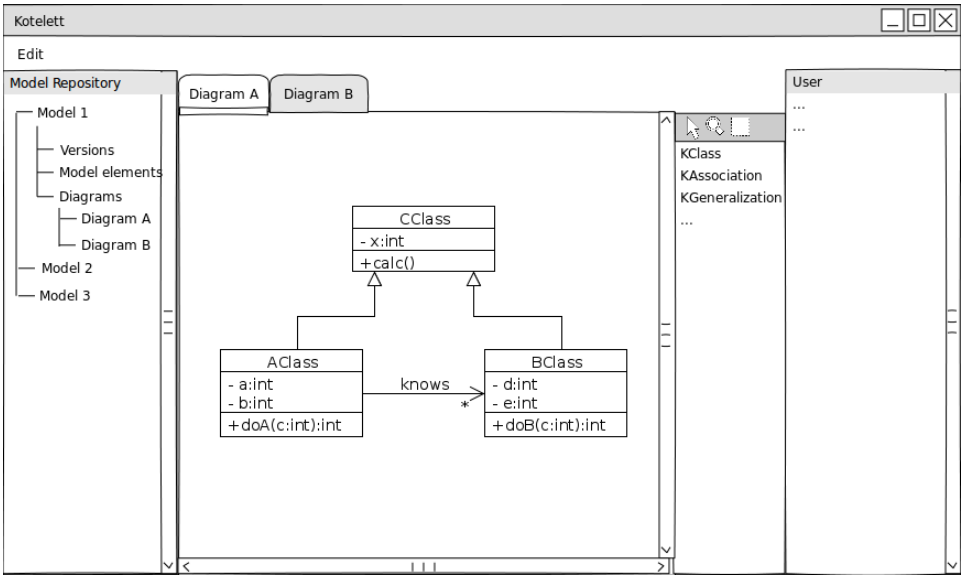

Abbildung 1.3: GUI-Entwurf des kollaborativen Modellierungswerkzeugs

Mithilfe der Baumansicht links können die Modelle auf dem Server vom Client aus verwaltet werden. Hier werden zugleich die Modellelemente und Diagramme zu einem Modell aufgelistet, um Diagramme auswählen zu können. Wie die Baumstruktur im Detail aussehen soll, steht noch nicht fest. Eine weitere Überlegung ist, über der Baumstruktur ein Drop-Down-Menü einzufügen, so dass der Anwender dort eine Version eines Modells auswählen kann und in der Baumstruktur dann nur noch die Elemente und Diagramme für die gewählte Version angezeigt bekommt. Ansonsten würde die Baumstruktur unter Umständen unübersichtlich werden.

Um neue Diagramme erstellen zu können und neue Versionen anzulegen, wird ein Kontextmenü (Rechtsklick) angeboten. Die Diagrammverwaltung bietet somit Möglichkeiten, alle auf dem Server gespeicherten Diagramme und ihre Versionen aufzulisten, Diagramme zu öffnen oder zu früheren Versionen zurückzukehren.

In der Mitte befinden sich die eigentlichen Editoren. Durch die Verwendung von Tabs können mehrere Diagramme gleichzeitig – jedes in einem eigenen Editor – geöffnet sein. Die Änderungen eines Anwenders in einem Diagramm werden in Echtzeit an alle anderen Clients übertragen. So werden sämtliche Änderungen von allen Anwendern sofort wahrgenommen.

Die Zeichentools für Diagrammelemente (Klassen, Assoziationen etc.) sowie Bearbeitungstools (Auswahl, Zoom, etc.) befinden sich in dem Menü direkt rechts neben dem Editor. In Abbildung 1.3 sind hier nur einige Bearbeitungs- und Zeichentools exemplarisch abgebildet.

In der Spalte am rechten Rand ist Platz für eine einfache Anwenderverwaltung. Wie diese aussehen soll, ist aber noch unklar. Wahrscheinlich wird sie sich auf eine Auflistung aller momentan im System angemeldeten Anwender und eventuell ihnen zugeordneten Farben zur Änderungsverfolgung im Diagramm beschränken. Da voraussichtlich GEF als Framework verwendet wird, wird die Anwenderverwaltung sich nicht ohne einen erheblichen Mehraufwand bei der Implementierung in eine Spalte mit den Modellierungstools bringen lassen.

1.7.2 Dokumentation

Die Dokumentation für dieses Projekt wird die folgenden Dokumente beinhalten:

- Ein Endbericht, welcher das Produkt und die dazugehörigen Anforderungen, Architektur, Implementierung und Evaluation dokumentiert.
- Ein Benutzerhandbuch, welches aus Anwendersicht den Umgang mit dem Produkt dokumentiert.
- Ein Installationshandbuch, welches die Installation und Einrichtung des Produkts dokumentiert.

- Ein Projekthandbuch, welches die Rahmenbedingungen des Projektes samt dem Entwicklungsprozess dokumentiert.
- Ein Prozessbericht, welches Prozessprobleme und die dazugehörigen Lösungen sowie 'Lessons Learned' aus diesem Projekt dokumentiert.

1.8 Glossar

DOL Delta Operation Language – Operationsbasierte Repräsentationssprache für Modellunterschiede.

DOLSA Delta Operations Language with Services and Applications – Ein Modellversionierungssystem, welches in der Softwaretechnik-Abteilung der Universität Oldenburg entwickelt wird.

Echtzeitsynchronisation Darstellung der Änderungen bei allen Anwendern im nahezu selben Moment.

Makroversionierung Klassische Versionierung, wie sie von gängigen Versionskontrollsystemen (Git, Subversion, Mercurial, o.ä.) umgesetzt wird.

Mikroversionierung Die Synchronisation zweier Modelle mittels Techniken, die aus der Versionierung kommen (z.B. diff, merge, etc)

Metamodell Ein Metamodell beschreibt Aufbau und Semantik eines Modells. Es kann somit auch als Datenstruktur gesehen werden.

Modell Für den Begriff Modell gibt es viele verschiedene Definitionen. Häufig wird ein Modell als eine Beschreibung eines Aspektes aus der Realität verstanden [MFBC12]. Wir verwenden den Modell-Begriff nach dieser Definition, da ein Modell die Dinge beschreiben sollte, die für das zu entwickelnde Softwaresystem relevant sind. Ein Modell ist dabei Instanz eines Metamodells, welches die Struktur des Modells und dessen Semantik beschreibt.

Diagramm Ein Diagramm ist eine bestimmte Sicht auf ein Modell. Während das Modell einen Teilaspekt der Realität als Ganzes betrachtet zeigt ein Diagramm nur einen kleinen Teil des Modells. Es ist somit eine grafische Repräsentation des Modells, wobei es mehrere Diagramme je Modell geben kann, da in der Regel ein Diagramm alleine nicht alles beinhaltet, was im Modell beschrieben ist. Somit ist eine vollständige Sicht auf das Modell nur durch Kombination der verschiedenen Sichten möglich. Aus technischer Sicht ist ein Diagramm – wie ein Modell – Instanz eines Metamodells, das vor allem das Layout anstelle der Semantik beschreibt. Da ein Diagramm eine Sicht auf das Modell beschreibt, stellt es aus Datensicht eine Referenz auf das Modell dar [EL11].

UML Die Unified Modeling Language (kurz UML) ist ein Industriestandard für die Modellierung von Softwaresystemen. Sie wird von der Object Management Group veröffentlicht und entwickelt.

Versionierung Der Begriff der Versionierung beinhaltet im Rahmen dieses Projektes zwei Arten der Versionierung: Mikroversionierung und Makroversionierung.

Teil II

Anforderungen

Kapitel 2

Einleitung

Dieses Dokument beschreibt die wesentlichen Anforderungen an das zu entwickelnde kollaborative Modellierungstool. Hierbei unterscheiden wir zwischen funktionalen und nichtfunktionalen Anforderungen. Die Anforderungen werden für das gesamte System gesammelt.

Generell unterscheiden wir Muss-Kriterien, Wunsch-Kriterien sowie Abgrenzungs-Kriterien.

Die Muss-Kriterien beschreiben die wesentlichen Anforderungen der PG, die zum Erreichen des Projektziels erfüllt werden müssen. Wunsch-Kriterien sind Features, die durchaus wünschenswert sind, aber im Zeitrahmen der PG voraussichtlich nicht mehr erfüllbar sind. Für den Fall, dass jedoch noch etwas Zeit übrig ist, sind dies Features, die noch umgesetzt werden könnten. Wunsch-Kriterien müssen ebenfalls für Architektur-Entscheidungen berücksichtigt werden, da diese nicht vorzeitig eine Realisierung der Wunsch-Kriterien unmöglich bzw. nur mit hohem Aufwand möglich machen darf.

Die Abgrenzungs-Kriterien beschreiben dagegen Anforderungen, die das Projekt ausdrücklich nicht als Ziel gesetzt hat. Dabei handelt es sich um Anforderungen, die an der Idee des Projektes vorbeigehen oder Features, die so umfangreich sind, dass sie für unser Projekt auch dann nicht mehr betrachtet werden können, wenn noch etwas Zeit übrig sein sollte. Dennoch ist es wichtig diese zu beschreiben, damit Klarheit darüber herrscht, was nicht mehr geschafft werden kann bzw. was nicht Ziel des Projektes ist, aber leicht als ein solches gesehen werden könnte, wenn dies nicht explizit ausgeschlossen wird.

Struktur der Anforderungen

Die Anforderungen werden gemäß der im Projekthandbuch definierten Struktur für Anforderungen aufgeschrieben. Der erste Teil dieser Struktur (PBI<ID>) wird allerdings ausgelassen, da in dieser Phase noch keine Zuordnung zu PBIs festgelegt wurde. Weitere auf PBIs bezogene Informationen

werden ebenfalls noch nicht aufgeführt (z.B. User Stories und Akzeptanztests).

Das Projektziel kann grob in 4 wesentliche Hauptanforderungen unterteilt werden (siehe Abschnitt 1.2):

I Modellierung von Klassendiagrammen

II Entwicklung einer Lösung zur Synchronisation zwischen Clients und Server

III Verwaltung mehrerer Modelle (Laden, Speichern etc.)

IV Versionierung von Modellen

Die Anforderungen werden nach diesen 4 Hauptanforderungen gruppiert aufgeschrieben. Um dies zu verdeutlichen, wird die entsprechende Nummer in römischer Schreibweise der Anforderung vorangestellt. Die erste Anforderung zur Entwicklung eines kollaborativen Modellierungstools erhält somit die Nummer I-FA1, die zweite Anforderung zur Versionierung von Modellen würde dagegen IV-FA2 als Nummer bekommen (Lies: Funktionale Anforderung aus der Kategorie 4, davon Anforderung Nummer 2). Darüber hinaus werden zusammengehörige Unteranforderungen zur besseren Übersichtlichkeit ebenfalls gruppiert dargestellt. Übergeordnete Anforderungen werden nur im ersten der jeweiligen Kategorien erwähnt und ggfs. in den anderen Kategorien referenziert.

Falls Anforderungen in keine der genannten Kategorien passen, werden sie als Sonstiges mit der Nummer 5 erfasst. Um nachvollziehbar zu machen, woher die Anforderungen stammen, wird die Quelle jeweils in Klammern hinter den einzelnen Anforderungen genannt.

In diesem Dokument werden die Begriffe Diagramm, Modell, Metamodell bzw. Diagramm/Modellelement verwendet. Eine genauere Begriffsdefinition befindet sich in unserem Glossar.

Kapitel 3

Muss-Kriterien

In diesem Kapitel werden alle Muss-Kriterien gesammelt, also Anforderungen, die erfüllt werden müssen, um das Projektziel zu erreichen. Diese sind nach den 4 bereits vorgestellten wesentlichen Anforderungen sowie funktionalen und nichtfunktionalen Anforderungen einsortiert und gruppiert.

3.1 I - Modellierung von Klassendiagrammen

Hauptanforderung I: Das System muss den Anwendern ermöglichen Klassendiagramme zu modellieren.
Die Modellierung von Klassendiagrammen ist eine Hauptanforderung. Dabei geht es im wesentlichen darum, dass Klassendiagramme gezeichnet werden können und hinter dem Diagramm auch ein entsprechendes Modell liegt. Die genaueren Unteranforderungen werden im folgenden erläutert.

Die Typen Klasse, abstrakte Klasse, Enumeration und Interface werden im folgenden als Knoten bezeichnet. Assoziationen, Aggregationen, Kompositionen, Generalisierungen und Implementierungen werden dagegen als Kanten bezeichnet.

Funktionale Anforderungen

– *Klassendiagramm zeichnen*:

I-FA1 **Wenn ein Anwender ein neues Modell erstellt, muss das System diesem Anwender ein leeres Diagramm zur Bearbeitung zur Verfügung stellen.**
Zu Beginn muss zu einem leeren Modell ein ebenfalls leeres Diagramm zur Verfügung stehen, in dem Klassendiagramme gezeichnet werden können. Zur Erstellung von neuen Modellen und Diagrammen siehe auch: Abschnitt 3.3

I-FA2 **Das System muss es dem Anwender ermöglichen, die in Tabelle 3.1 genannten Änderungen der jeweils aufgeführten Dia-**

grammelemente durchzuführen. Die aufgeführte Tabelle gibt zu den verschiedenen Diagrammelementen eine Aktion an (Art der Änderung) und ein Änderungsobjekt – also was in dem Fall geändert wird. Aus der Tabelle wird somit ersichtlich, welche Aktionen der Anwender auf dem Modell durchführen kann.

I-FA3 **Das System muss dem Anwender ermöglichen, die in Tabelle 3.2 genannten Änderungen an Generalisierungen im Diagramm durchzuführen.** Da für Generalisierungen etwas andere Regeln gelten werden diese hier gesondert aufgeführt.

I-FA4 **Das System muss es dem Anwender ermöglichen, ein Diagrammelement auszuwählen.**
Die Selektion ist ein wesentliches Feature, um ein Diagramm bearbeiten zu können. Die Selektion ist damit auch Basis für die folgende Anforderung.

I-FA4.1 **Wenn ein Anwender ein Diagrammelement ausgewählt hat, muss das System ihm ermöglichen, das ausgewählte Element zu bewegen.**
Die Position eines Elements im Diagramm muss nach der Auswahl des Elements veränderbar sein.

I-FA4.2 **Wenn ein Anwender ein Diagrammelement ausgewählt hat, muss das System ihm ermöglichen, die Größe des ausgewählten Elements zu ändern.**
Die Größe eines Elements im Diagramm muss nach der Auswahl des Elements veränderbar sein.

I-FA4.3 **Wenn ein Anwender ein Diagrammelement ausgewählt hat, muss das System ihm ermöglichen, das ausgewählte Element zu entfernen.**
Das Löschen von Elementen aus einem Diagramm soll nach vorheriger Selektion möglich sein.

I-FA5 **Wenn ein Anwender ein Diagramm ändert, muss das System die Änderung in das Modell übertragen**
Das Diagramm ist die Schnittstelle und die Visualisierung für den Anwender, an dem er Änderungen ausführt und diese angezeigt bekommt, gespeichert werden die Daten im Model, an das die Eingaben des Nutzer im Diagramm weitergeleitet werden müssen.

– *Visualisierung*:

I-FA6 **Das System muss Änderungen aus Tabelle 3.1, die der Anwender am Diagramm durchführt, im Diagramm visualisieren.**
Die in der Tabelle genannten Änderungen wurden auf Modellebene beschrieben. Der Anwender interagiert jedoch mit dem Diagramm, daher

Diagrammelement	Änderungsobjekt	Art der Änderung
Klassen Interfaces abstrakte Klassen Enumerationen	selbst	hinzufügen entfernen
	Name	ändern
Klassen Interfaces abstrakte Klassen	Methoden Attribute	hinzufügen entfernen
Enumeration	Literal	hinzufügen entfernen
Assoziation Aggregation Komposition	zwischen zwei Knoten	hinzufügen entfernen
	Quelle Ziel	ändern
	Typ (Assoziation, Aggregation und Komposition)	ändern
	Multiplizitäten Rollenbezeichner Sichtbarkeiten	ändern
	Navigierbarkeiten	setzen entfernen
Methode	Abstrakt Statisch	setzen entfernen
	Name Sichtbarkeit Rückgabetyp	ändern
	Methodenparameter	hinzufügen entfernen
Methodenparameter	Name Rückgabetyp	ändern
Attribut	Statisch	setzen entfernen
	Name Sichtbarkeit Rückgabetyp	ändern

Tabelle 3.1: Mögliche Änderungen an Diagrammelementen(außer Generalisierungen)

Art der Änderung	Quelle	Ziel (Superklasse)
einfügen	Interface	Interface
entfernen	Klasse	Interface
	Klasse	Klasse

Tabelle 3.2: Mögliche Änderungen an Generaliserungen

müssen von ihm initialisierte Änderungen am Modell auch direkt im Diagramm visualisiert werden.

I-FA7 **Das System muss ein ausgewähltes Diagrammelement hervorheben.**
Da besondere Aktionen mit selektierten Elementen möglich sind (Verschieben, Löschen) muss für den Anwender ersichtlich sein, welche Elemente derzeit selektiert sind.

I-FA8 **Das System muss des dem Anwender ermöglichen die eigenen Arbeitsschritte rückgängig zu machen.**
Um durchgeführte Änderungen schnell rückgängig zu machen muss eine Undo-Funktionalität umgesetzt werden. Dabei kann jedoch das Problem auftreten, dass andere Anwender zwischenzeitlich ihre Änderungen durchführen. Um diese Änderungen nicht versehentlich wieder rückgängig zu machen, darf die Undo-Funktion nur für eigene Änderungen gelten.

I-FA8.1 **Das System muss es dem Anwender ermöglichen eigene, rückgängig gemachte Arbeitsschritte zu wiederholen, solange keine neuen Änderungen von ihm durchgeführt wurden.**
Die Redo-Funktion muss ähnlich der Undo-Funktion die letzten eigenen rückgängig gemachten Änderungen wiederherstellen. Um die Undo/Redo Funktion schlank zu halten muss dies nur möglich sein, solange zwischendurch keine neuen Änderungen des gleichen Anwenders durchgeführt wurden. Dies würde ansonsten ein erheblich umfangreicheres Speichern der Änderungen erfordern, um bei sämtlichen Kombinationen das korrekte Ergebnis zu liefern.

I-FA8.2 **Das System muss einem Undo/Redo-Befehl ignorieren, wenn es sich auf ein Element bezieht, das bereits gelöscht wurde - mit Ausnahme der Befehle Undo-Delete und Redo-Create.**
Im Falle von Undo/Redo können Konflikte auftreten, da andere User zwischenzeitig ebenfalls Änderungen an den gleichen Elementen vornehmen. Ein großes Problem ist, wenn das Element, an

dem Änderungen durchgeführt werden sollen, zwischendurch von einem anderen Anwender gelöscht wurde. In diesem Fall soll das System eine einfache Konfliktlösungsstrategie verwenden, indem es einen derartigen ungültigen Undo/Redo-Command dann einfach ignoriert. Ausnahme bilden hierbei Undo-Delete sowie Redo-Create, da durch diese beiden Befehle die zuvor gelöschten Elemente wiederhergestellt werden sollen. Der Unterschied ist hier, dass der selbe Anwender zuvor die Elemente auch gelöscht hatte und dieses Löschen damit rückgängig machen möchte.

I-FA8.3 **Das System muss Änderungen durch Undo/Redo - Commands auch dann übernehmen können, wenn zwischendurch Änderungen anderer User am selben Element durchgeführt wurden.**
Ein weiterer Konfliktfall besteht darin, dass andere Anwender das gleiche Element zwar nicht löschen, aber daran ebenfalls etwas ändern. So könnte es sein, dass Anwender 1 den Klassennamen von `Test` auf `MeinTest` ändert. Ein anderer Anwender könnte den Namen dann auf `UnserTest` ändern. Wenn Anwender 1 nun Undo ausführt würde die Änderung des zweiten Anwenders durch den ursprünglichen Klassennamen `Test` überschrieben.

– *Layout*:

I-FA9 **Das System muss es dem Anwender ermöglichen, die in Tabelle 3.3 genannten Änderungen am Layout der jeweils aufgeführten Diagrammelemente durchzuführen.** Die aufgeführte Tabelle gibt zu den verschiedenen Diagrammelementen eine Aktion an (Art der Änderung) und ein Änderungsobjekt – also was in dem Fall geändert wird. Aus der Tabelle wird somit ersichtlich, welche Aktionen der Anwender am Layout des Diagramms durchführen kann.

Diagrammelement	Änderungsobjekt	Art der Änderung
Klassen Interfaces abstrakte Klassen Enumerationen	selbst	Größe ändern verschieben
Assoziation Aggregation Komposition Generalisierung	Bendpoints	hinzufügen entfernen verschieben

Tabelle 3.3: Mögliche Änderungen an Diagrammelementen

I-FA9.1 **Wenn ein Knoten oder eine Kante verschoben wird muss**

das System sicherstellen, dass alle dazugehörigen Texte mitverschoben werden.
In Klassen werden neben dem Klassennamen auch Attributsnamen und Methoden als Text erfasst. An Assoziationen entsprechend Multiplizitäten oder Rollenbezeichner. Es muss daher sichergestellt werden, dass derartige Texte mitverschoben werden, wenn das übergeordnete Element (Klasse, Assoziation usw.) verschoben wird.

I-FA9.2 **Wenn ein Knoten in einem Diagramm verschoben wird, muss das System sicherstellen, dass die Verbindung mit allen ein- und ausgehenden Kanten erhalten bleibt.**
Analog zu den Texten müssen auch an Knoten hängende Kanten (z.B. Assoziationen an Klassen) fest an der Klasse hängen bleiben.

I-FA10 **Wenn die Länge eines Textes in einem Diagrammelement die Grenzen des ihn umschließenden Elements überschreitet, muss das System die Darstellung des Textes so behandeln, dass er in das Diagrammelement passt und der vollständige Text für den Anwender einsehbar ist.**
Diese Anforderung definiert, wie mit überlangen Texten umgegangen werden soll. Man könnte zunächst eine Scrollbar einblenden, welche in kleinen Elementen wie z.B. Klassen aber eher stören würde. Eine andere Möglichkeit wäre es, dass die Klassen automatisch vergrößert werden, um den gesamten Text darstellen zu können. Diese Lösung würde dem Benutzer aber die Option verwehren, die Klasse kleiner zu gestalten, auch wenn dann Teile des Textes nicht mehr lesbar sind. Die Lösungsmöglichkeiten sollen nicht vorgegeben werden, aber der Text könnte z.B. verkürzt oder automatisch umgebrochen werden.

– *Editor*:

I-FA11 **Das System muss dem Anwender ein grafisches Interface bieten, auf dem erstellbare Diagrammelemente angeboten werden, die in das Diagramm eingefügt werden können.**
Dem Anwender muss ein graphischer Editor in der GUI angeboten werden, in dem er alle erstellbaren Elemente auswählen kann. Diese müssen sich dann in das Diagramm einfügen lassen. Durch Anforderung I-FA5 wird dies dann ins Modell synchronisiert.

I-FA12 **Das System muss sicherstellen, dass alle Modelle jederzeit zu ihrem Metamodell konform sind.**
Ein wesentliches Artefakt der Projektgruppe ist das Metamodell für Klassendiagramme. Dieses wird sowohl auf dem Server, als auch auf den Clients verwendet. Außerdem wird es zur Erstellung der DOL-Statements von DOLSA verwendet. Bei der Modellierung muss das System daher jederzeit sicherstellen, dass dieses Metamodell eingehal-

ten wird.

3.2 II - Echtzeitkollaboration und Synchronisation von Modellen

Hauptanforderung II: Das System muss den Anwendern ermöglichen Diagramme kollaborativ und in Echtzeit mit anderen Anwendern zu bearbeiten.

Bei der zweiten Hauptanforderung handelt es sich um die Echtzeitkollaboration und eine damit verbundene Synchronisation von Modelländerungen. Modelländerungen, die von einem Anwender durchgeführt wurden, müssen dabei auf das Modell aller anderen Anwender übertragen werden, sofern diese am gleichen Modell arbeiten. Änderungen am Modell ziehen dabei auch Änderungen am Diagramm nach sich, sofern der geänderte Teil im Diagramm dargestellt wird. Außerdem muss ersichtlich werden, wer welche Änderungen durchgeführt hat. Diese Unteranforderungen werden daher nachfolgend weiter konkretisiert.

Anmerkung: Auch wenn nachfolgend von Diagrammen und Modellen gesprochen wird, sind im Rahmen dieses Projektes damit Klassendiagramme und ein dahinter stehendes Modell gemeint.

Funktionale Anforderungen

– *Synchronisation der Diagramm- und Modelländerungen*:

II-FA1 **Das System muss Änderungen, die ein Anwender an einem Modell durchführt, an alle Anwender (bzw. deren Clients) des gleichen Modells übertragen und in allen von der Änderung betroffenen Diagrammen visualisieren.**
Eine wesentliche Anforderung an das zu entwickelnde System ist es, dass die Anwender nicht bloß ein Diagramm zeichnen, sondern das dahinterliegende Modell verändern. Dabei kann es grundsätzlich mehrere Diagramme zu einem Modell geben, wobei zwei oder mehr Diagramme auch die gleichen Modellelemente anzeigen können. Wenn diese nun verändert werden muss sich diese Änderung auf sämtliche Diagramme auswirken, in denen das Modellelement dargestellt wird. Dies hat zur Folge, dass alle Anwender, die am selben Modell arbeiten, über sämtliche Modelländerungen informiert werden müssen, auch wenn sie ein ganz anderes Diagramm bearbeiten, da es eventuell auch Auswirkungen auf ihr Diagramm haben kann.

– *Anwenderverfolgbarkeit*:

II-FA2 **Das System muss Diagrammänderungen, die von anderen**

Anwendern durch Veränderung des Modells ausgelöst wurden, gesondert darstellen, um eine visuelle Trennung der eigenen Änderungen von denen anderer Teilnehmer zu erhalten.
Im Rahmen der kollaborativen Bearbeitung ist es erforderlich, dass ein Anwender erkennen kann welche Änderungen von Anderen durchgeführt wurden, um den Überblick zu behalten. Dem Anwender muss dies in irgendeiner Form ersichtlich gemacht werden, z.B. indem die betroffenen Diagrammelemente in einer anderen Farbe dargestellt werden. Da Diagrammänderungen immer durch eine Änderung des Modells ausgelöst werden schließt dies sowohl den Fall ein, dass zwei Anwender das gleiche Diagramm bearbeiten, als auch den Fall, dass unterschiedliche Diagramme bearbeitet werden, die Änderung aber Einfluss auf beide Diagramme hat.

II-FA3 **Das System muss bei der Darstellung der Änderungen jeweils den letzten Bearbeiter eines Knotens oder einer Kante darstellen.**
Eine wesentliche Fragestellung bei der Darstellung von Änderungen, die durch andere Anwender durchgeführt wurden, ist es, wie feingranular diese dargestellt werden sollen. Im Rahmen dieses Projektes soll dies eher grob durchgeführt werden, indem nur jeweils die letzte Änderung z.B. einer Klasse oder Assoziation angezeigt wird. Dabei ist es unerheblich, ob z.B. der Klassenname, eine Methode oder ein Attribut geändert, hinzugefügt oder gelöscht wurde. So könnte bei einer der genannten Änderungen die Klasse z.B. die Farbe des Anwenders erhalten, der die Änderung durchgeführt hat. Eine genauere Identifikation, was geändert wurde (z.B. welche Zeichen geändert wurden), ist nicht erforderlich.

II-FA4 **Das System muss dem Anwender ermöglichen, die Darstellung des letzten Bearbeiters aus- und wieder einzublenden.**
Eine Kennzeichnung, welcher Anwender was geändert hat, ist im Laufe der Bearbeitung sehr sinnvoll. Um ein gemeinsam erstelltes Diagramm aber weiterzuverwenden ist eine solche Darstellung (z.B. unterschiedliche Farben) nicht gewünscht. Es muss daher möglich sein die Darstellung bisher durchgeführter Änderungen aus- und wieder einzublenden.

– *Konfliktbehandlung*:

II-FA5 **Das System muss Änderungskonflikte, die durch die parallele Bearbeitung eines Modelles entstehen können, abfangen und nach der First-Come First-Serve Strategie behandeln, sodass kein inkonsistenter Zustand oder Fehler entsteht.**
Im Rahmen kollaborativer Modellierung ist es möglich, dass Konflikte auftreten. Eine Möglichkeit ist es, dass ein Element von einem Anwender gelöscht wird, während zeitgleich ein anderer Änderungen daran

durchführt. Das führt dazu, dass eine Änderung auf einem Element ausgeführt werden soll, das nicht mehr existiert. Derartige Fehlerfälle müssen vom System erkannt und behandelt werden. Hierbei wird die einfache Lösungsstrategie First-Come, First-Serve angewandt. Dies hat zur Folge, dass Änderungen von Anwendern vom Server verworfen werden, da sie im Konflikt zu vorherigen Änderungen stehen.

II-FA5.1 **Das System muss einem Client, durch dessen Änderung ein Konflikt aufgetreten ist, den aktuell auf dem Server liegenden Stand schicken, damit dieser wieder synchron ist.**
Im zuvor genannten Fall, dass zwei Änderungen zueinander im Konflikt stehen, soll der zweite Anwender, dessen Änderung verworfen wurde, eine Resynchronisation seines Modells und damit des geöffneten Diagramms erhalten. Dazu soll der auf dem Server gespeicherte, aktuell gültige Stand nur diesem Client neu zugeschickt werden.

II-FA5.2 **Das System muss den Anwender informieren, wenn seine Änderung verworfen wurde.**
Als Konsequenz aus II-FA5 und II-FA5.1 muss der Anwender darüber informiert werden, dass seine letzte Änderung vom Server verworfen wurde. Dies könnte z.B. durch eine Texteinblendung geschehen.

II-FA6 **Das System muss es dem Anwender ermöglichen, sich alle Anwender, welche am gleichen Modell arbeiten, anzeigen zu lassen.**

Nichtfunktionale Anforderungen

II-NFA1 **Das System soll Änderungskonflikte auf dem Server behandeln.**
Alle Konflikte, die bei der Synchronisation der Clients entstehen können, sollen vom Server geprüft werden. Der Server entscheidet darüber, ob Änderungen gültig sind und löst im Falle von Fehlern eine Resynchronisation einzelner Clients aus. Die Clients müssen damit lediglich Fehler, die durch ungültige Änderungen entstehen können, abfangen (ohne weitere Aktion). Da in einem solchen Fall aber auch ein Konflikt auf dem Server auftritt muss der Server diesen durch eine Resynchronisation beheben.

II-NFA2 **Das System muss Änderungen aller Anwender an den Server übertragen, damit dieser den aktuell gültigen Stand speichern und im Falle von Konflikten die Clients synchronisieren kann.**
Änderungen, die durch die Anwender durchgeführt wurden, müssen

nicht nur zwischen den Anwendern ausgetauscht werden, sondern auch auf einen Server übertragen werden. Dieser bildet den Single Point of Truth (siehe Abschnitt 1.2) und ist dafür verantwortlich den insgesamt gültigen Stand des Modells zu halten, zu speichern, neuen Clients zur Verfügung zu stellen und auftretende Konflikte zu behandeln.

II-NFA3 **Das System muss zur Darstellung eines vom Anwender geöffneten, bereits existierenden Diagrammes das zugehörige Modell vom Server laden und verwenden.**
Eine grundsätzliche Anforderung ist neben einer Client-Server-Architektur die Festlegung, dass der Server den Single Point of Truth bildet (siehe Abschnitt 1.2). Als Konsequenz muss ein Anwender, der ein Diagramm öffnet, einen aktuellen Modell-Stand vom Server geliefert bekommen, sodass er den letzten gültigen Stand erhält.

II-NFA4 **Das System soll Diagrammänderungen in Echtzeit ($<$ 1 Sekunde innerhalb des ARBI-Netzes) übertragen.**

II-NFA5 **Zur Synchronisation zwischen den Clients und dem Server muss die Delta Operating Language (DOL)[KW14] verwendet werden.**
Ein wesentlicher Bestandteil des Systems ist die Synchronisation zwischen den Clients. Dabei ist eine Anforderung vom Auftraggeber, dass diese mithilfe der Delta Operating Language durchgeführt werden soll. DOL bietet bereits ein Verfahren, um mithilfe eines fest definierten Metamodells DOL-Statements zum Austausch zu definieren, die aufgetretenen Differenzen durch die Änderung eines Anwenders zu bestimmen und von einem anderen Client empfangene DOL-Statements auf dessen Modell anzuwenden. Es handelt sich hierbei um eine wesentliche Anforderung, da das Projekt somit ein Fallbeispiel für die Arbeit aus der ST-Abteilung darstellt.

II-NFA6 **Der Server und die Clients müssen ein gemeinsames Metamodell zur Differenzberechnung/-Anwendung verwenden.**
Eine weitere technische Entscheidung, die explizit vom Auftraggeber gefordert wurde und somit zu einer wesentlichen Anforderung wurde, ist die Verwendung eines gemeinsamen Metamodells zur Berechnung von Modelldifferenzen und zur Anwendung dieser Differenzen. Diese Anforderung ergibt sich aus II-NFA5, da ein einheitliches Metamodell sowohl auf Client, Server als auch innerhalb der DOL-Services verwendet werden muss, damit die Synchronisation mit DOL durchgeführt werden kann.

II-NFA7 **Die Konfliktlösungsstrategie muss austauschbar sein.**
Der Auftraggeber wünscht sich eine Austauschbarkeit der geplanten, einfachen Konfliktlösungsstrategie auf Seiten des Servers.

II-NFA8 **Das System muss es Anwendern ermöglichen von verschiede-**

nen Standorten kollaborativ zu Modellieren
Das System soll es Anwendern ermöglichen von verschiedenen Standorten (z.B. zwei Büros in verschiedenen Universitäten) kollaborativ an einem Modell zu arbeiten.

3.3 III - Verwaltung mehrerer Modelle

Hauptanforderung III: Das System muss dem Anwender ermöglichen mehrere Modelle zu verwalten.

Um die Kollaboration für mehr als ein Modell zu ermöglichen muss das System dem Anwender eine Modellverwaltung bereitstellen. Ohne diese wäre das System auf genau ein Modell beschränkt, welches den effektiven Einsatz des Systems unterbinden würde. Zudem ist die Modellverwaltung kritisch für die Skalierbarkeit und Erweiterbarkeit des Systems.

3.3.1 Funktionale Anforderungen

– *Modelle*:

III-FA1 **Das System muss es dem Anwender ermöglichen, Modelle zu erstellen.**

III-FA1.1 **Wenn der Anwender ein Modell erstellt, muss das System den Anwender dazu auffordern, dem Modell einen eindeutigen Namen zu geben.**
Der Anwender muss beim Erstellen eines Modells einen eindeutigen Namen angeben, damit er und andere Anwender das Modell später wiederfinden können.

III-FA2 **Das System muss es dem Anwender ermöglichen, Modelle zu löschen.**

III-FA3 **Wenn ein Modell gelöscht wird muss das System sicherstellen, dass auch alle dazugehörigen Diagramme gelöscht werden**

III-FA4 **Das System muss es dem Anwender ermöglichen Modelle zur Bearbeitung zu öffnen.**
Ein Modell, das bereits geöffnet und gespeichert wurde, muss zu einem späteren Zeitpunkt wieder geöffnet werden können, damit weitere Änderungen daran durchgeführt werden können.

III-FA5 **Das System muss dem Anwender eine Liste aller verfügbaren Modelle anbieten.**

Um ein Modell zu öffnen muss dem Anwender eine Liste aller verfügbaren Modelle angeboten werden, aus denen er dann eines auswählen kann.

III-FA6 **Das System muss es dem Anwender ermöglichen, seine Bearbeitung eines Modells zu beenden.**
Der Anwender muss die Bearbeitung eines Modelles beenden können, z.B. durch Schließen der Anwendung. Dies muss vom System erkannt werden, damit festgestellt werden kann, ob und welche Anwender an einem Modell aktiv arbeiten.

– *Diagramme*:

III-FA7 **Das System muss es dem Anwender ermöglichen ein Diagramm zu einem Modell zu erstellen.**
Der Anwender muss zu dem Modell, das er geöffnet hat, ein neues Diagramm erstellen können. Dabei können zu einem Modell mehrere Diagramme angelegt werden.

III-FA7.1 **Wenn der Anwender ein Diagramm erstellt, muss das System den Anwender dazu auffordern, dem Diagramm einen Namen zu geben.**

III-FA8 **Das System muss es dem Anwender ermöglichen Diagramme zu löschen.**

III-FA9 **Das System muss es dem Anwender ermöglichen, Diagramme zur Bearbeitung zu öffnen.**
Auch Diagramme müssen geöffnet werden können, damit der Anwender weitere Änderungen daran durchführen kann.

III-FA10 **Das System muss dem Anwender eine Liste aller zum aktuell geöffneten Modell verfügbaren Diagramme anbieten.**
Damit ein Diagramm geöffnet werden kann muss der Anwender auch eine Liste aller verfügbaren Diagramme angezeigt bekommen, damit er eines zur Bearbeitung öffnen kann.

III-FA11 **Das System muss es dem Anwender ermöglichen, seine Bearbeitung eines Diagramms zu beenden.**
Der Anwender muss die Bearbeitung eines Diagramms beenden können, z.B. durch Schließen des Diagramms.

III-FA12 **Das System muss sicherstellen, dass jedes Modell mindestens ein Diagramm hat.** Jedes Modell muss mindestens ein Diagramm haben.

3.3.2 Nicht-Funktionale Anforderungen

III-NFA1 **Das System muss zur Speicherung der einzelnen Modelle und Diagramme das Modellversionierungssystem GMoVerS verwenden.**
GMoVerS [KW14] bietet bereits das Speichern von Modellen an, die mit leichten Anpassungen auch für die Modellverwaltung unseres Projektes verwendet werden kann. Es wurde daher in Absprache mit den Auftraggebern vereinbart, dass diese verwendet wird.

3.4 IV - Versionierung von Modellen

Hauptanforderung IV: Das System muss dem Anwender die Versionierung von Modellen ermöglichen.

Anwender in der Softwareentwicklung benötigen Versionierung um auf frühere Versionen zurückzugreifen. Dies kann genutzt werden, um unter anderem in Fehlerfällen frühere Versionen zu verwenden oder die Modellhistorie nachzuvollziehen.

3.4.1 Funktionale Anforderungen

IV-FA1 **Das System muss dem Anwender alle verfügbaren Versionen des geöffneten Modells anzeigen.**
Um auf ältere Versionen zurückspringen zu können muss den Anwender eine Liste aller Versionen eines Modells angezeigt werden können. Dabei ist eine Unterscheidung der Diagramme nicht vorgesehen. Es ist also nicht erforderlich, dass eine Liste aller Versionen, die ein bestimmtes Diagramm aus dem Modell betreffen, angezeigt werden kann.

IV-FA2 **Das System muss dem Anwender alle zu einer Version zugehörigen Modellelemente anzeigen.**
Der Anwender muss sehen können welche Modellelemente in einer gewählten Version vorhanden sind.

IV-FA3 **Das System muss es dem Anwender ermöglichen den jeweils aktuellen Zustand des Modells als eine neue Version zu speichern.**
Um eine neue Version anzulegen muss der Anwender die Möglichkeit haben eine neue Version vom Zustand des Modells, das auf dem Server liegt, zu erstellen.

IV-FA4 **Wenn der Anwender eine neue Version speichert muss das System dem Anwender ermöglichen, der Version einen Namen zu geben.**
Der Anwender muss die Möglichkeit haben einer Version einen festen

Namen zu geben, damit er diese später wiederfinden kann, falls er zu ihr zurückspringen möchte.

IV-FA5 **Das System muss die Liste aller Versionen in zeitlicher Reihenfolge darstellen.**
Der Anwender muss erkennen können wie alt eine Version ist. Aus diesem Grund ist es erforderlich, dass die Versionen nach Anlagedatum sortiert angezeigt werden.

IV-FA6 **Wenn kein Anwender mehr an einem Modell arbeitet, muss das System den derzeitigen Zustand des Modells als eine neue Version speichern, bevor das Modell geschlossen wird.**
Sobald der letzte Anwender die Bearbeitung eines Modells beendet hat muss eine neue Version gespeichert werden, damit der erarbeitete Stand nicht verloren geht. Es kann nicht davon ausgegangen werden, dass die Anwender dies regelmäßig manuell durchführen, insbesondere auch, da technische Ausfälle ebenfalls dafür sorgen könnten, dass ein Anwender von einem Modell abgemeldet wird. Um Datenverlust zu vermeiden soll der letzte Stand also immer als neue Version gespeichert werden.

IV-FA7 **Das System muss dem Anwender ermöglichen auf frühere Versionen des Modells zurückzugreifen.**
Die Anwender müssen eine beliebige Version eines Modells auswählen und öffnen können. Diese Version wird dann für alle Anwender geöffnet, die an dem Modell arbeiten. Der davor letzte Stand des Modells wird dabei mit dem Stand der geladenen Version überschrieben. Alle Änderungen, die zwischen der geladenen Version und dem letzten Stand des Modells durchgeführt wurden, werden damit verworfen. Derartige Zwischenversionen können aber wiederhergestellt werden, indem eine entsprechende Version geladen wird.

3.4.2 Nichtfunktionale Anforderungen

IV-NFA1 **Das System muss zur Modellversionierung das Modellversionierungssystem GMoVerS verwenden [KW14].**
Eine wesentliche Anforderung des Auftraggebers ist es, dass zur Versionierung GMoVerS verwendet wird.

Kapitel 4

Wunsch-Kriterien

Die in diesem Kapitel vorgestellten Anforderungen sind im Rahmen des Projektes durchaus wünschenswert, allerdings aktuell nicht fest eingeplant. Falls die Bearbeitung der geplanten Features unter den Muss-Kriterien schneller als geplant verläuft und noch Zeit übrig bleibt können Features aus diesem Katalog noch mit aufgenommen werden.

4.1 I - Modellierung von Klassendiagrammen

Funktionale Anforderungen

I-WA1 **Das System soll dem Anwender ermöglichen, mehrere Diagrammelemente gleichzeitig auszuwählen.** Dies ist eine Erweiterung von Anforderung I-FA4, um auch mehrere Diagrammelemente gleichzeitig auswählen und verschieben bzw. entfernen zu können.

I-WA1.1 **Wenn ein Anwender mehrere Elemente ausgewählt hat, muss das System ihm ermöglichen, die ausgewählten Elemente zusammen zu bewegen.**
Selektierte Elemente müssen vom Anwender verschoben werden können. Dies muss auch gelten, wenn mehr als ein Element selektiert wurde. Dazu siehe auch I-FA4.1

I-WA1.2 **Wenn ein Anwender mehrere Elemente ausgewählt hat, muss das System ihm ermöglichen, die ausgewählten Elemente zusammen zu entfernen.**
Selektierte Elemente müssen vom Anwender gelöscht werden können. Dies muss auch gelten, wenn mehr als ein Element selektiert wurde. Dazu siehe auch I-FA4.3

I-WA2 **Das System muss es dem Anwender ermöglichen, Elemente zum Modell hinzuzufügen, ohne dass diese auch Teil eines Diagramms sind.**

Zur Modellierung kann es nötig sein, Elemente zu erstellen aber nicht zu visualisieren. Bspw. Datenstrukturen die nur als Parameter von Methoden verwendet werden. Dies wird ermöglicht, in dem der Anwender Elemente direkt im Modell erstellen kann. Diese sind dann Teil des Modells aber in keinem der dem Modell zugeordneten Diagramme enthalten.

I-WA2.1 **Das System muss es dem Anwender ermöglichen, die Eigenschaften eines Elements das nicht in einem Diagramm visualisiert wird, zu verändern, ausgenommen von Eigenschaften die mit der Visualisierung zusammenhängen würden.**
Auch Elemente die Teil keines Diagramms sind müssen vom Anwender verändert werden können. Dies schließt die Änderung von Layoutinformationen (Position, Größe) explizit aus.

4.2 II - Echtzeitkollaboration und Synchronisation von verteilten Modellen

Funktionale Anforderungen

II-WA1 **Das System muss dem Anwender ermöglichen, die Darstellung des letzten Bearbeiters aus- und wieder einzublenden.**
Eine Kennzeichnung, welcher Anwender was geändert hat, ist im Laufe der Bearbeitung sehr sinnvoll. Um ein gemeinsam erstelltes Diagramm aber weiterzuverwenden ist eine solche Darstellung (z.B. unterschiedliche Farben) nicht gewünscht. Es muss daher möglich sein die Darstellung bisher durchgeführter Änderungen aus- und wieder einzublenden.

II-WA2 **Das System muss dem Anwender ermöglichen den letzten Bearbeiter aller Elemente im Modell zu entfernen, sodass alle bisherigen Elemente in einer neutralen Farbe dargestellt und durch weitere Änderungen wieder eingefärbt werden.**
Im Gegensatz zu II-WA1 soll es auch möglich sein die Farbe aller Elemente auf eine neutrale Farbe zurückzusetzen, indem der letzte Bearbeiter des Elementes im Modell entfernt wird. Dadurch geht die Information des letzten Bearbeiters verloren. Damit ist es möglich einen gemeinsamen Zwischenstand zu definieren. Nachfolgende Änderungen sind damit dann für alle besser ersichtlich.

4.3 III - Verwaltung mehrerer Modelle

Funktionale Anforderungen

III-WA1 **Das System muss es dem Anwender ermöglichen, ein Modellelement zu mehreren Diagrammen des Modells hinzuzufügen.** Da mehrere Diagramme pro Modell nach III-FA7 möglich sein müssen, wäre es wünschenswert, wenn im Rahmen von Klassendiagrammen auch multiperspektivische Modellierung möglich ist. D.h., dass sich bspw. dieselben Klassen aus einem Modell in unterschiedlichen Diagrammen des Modells unterschiedlich betrachten lassen.

III-WA1.1 **Wenn der Anwender ein Modellelement ändert, dass in mehreren Diagrammen enthalten ist, dann muss diese Änderungen in allen Diagrammen dargestellt werden.** Um die Konsistenz des Modells und den Diagrammen zu gewährleisten, müssen Änderungen an Modellelementen immer auch in den zugeordneten Diagrammen widergespiegelt werden.

III-WA2 **Das System soll dem Anwender ermöglichen alte Versionen zu betrachten ohne den aktuellen Stand zu verwerfen** Falls ein Benutzer sich eine ältere Version anschauen möchte soll das System ihm die Möglichkeit dazu bieten, ohne dass der aktuelle Stand direkt übersprungen wird (d.h. die ausgewählte Version geladen wird). Es soll sich also um eine Ansicht älterer Versionen handeln.

Kapitel 5

Abgrenzungskriterien

Die in diesem Abschnitt behandelten Anforderungen sind solche, die explizit nicht im Rahmen der PG behandelt werden, auch dann nicht, wenn noch etwas Zeitreserve übrig sein sollte. Dies soll zum einen abgrenzen, was das Produkt nicht sein will und zum anderen, was nicht mehr im Rahmen der PG behandelt werden kann.

Nachfolgend werden funktionale Anforderungen beschrieben, die im Rahmen der PG nicht mehr erfüllt werden können oder die am Ziel der PG vorbeigehen.

5.1 Grobe funktionale Anforderungen

Zusätzliche Diagrammtypen:
Das System könnte neben den Klassendiagrammen die Modellierung weiterer Diagrammtypen (Aktivitätsdiagramm, Sequenzdiagramm usw.) unterstützen.

Dies ist im Rahmen der PG nicht mehr realistisch, da ein weiterer Diagrammtyp an sich sehr umfangreich ist (eigenes Metamodell, zusätzliche grafische Elemente) und zusätzlich eine Typ-Auswahl für den Anwender eingebunden werden müsste bzw. eine Unterscheidung bei der Persistierung berücksichtigt werden müsste.

E-Mail-Benachrichtigung:
Das System könnte alle beteiligten Anwender per E-Mail über Änderungen informieren. Ein solches Feature ist im Rahmen der PG nicht mehr möglich, da hierzu genauer analysiert werden müsste, wann genau Mails verschickt werden sollen und eine Möglichkeit angebunden werden muss dieses Verhalten abzustellen. Da keine Nutzerverwaltung geplant ist, fehlen dazu die nötigen Voraussetzungen.

Offlinemodus:
Das System könnte einen Offlinemodus anbieten, sodass Anwender auch ohne Internetzugriff weiter am Modell arbeiten können. Neben der GUI-

Umsetzung (Offline-Modus starten/stoppen) ist vor allem die Konfliktbehandlung hier ein großes Problem. Die aufgetretenen Konflikte müssten in irgendeiner Form aufgelöst werden. Eine nähere Betrachtung dieser Problematik würde den Rahmen der PG bei weitem sprengen.

Textchat:
Das System könnte einen Textchat anbieten, sodass Anwender, die am selben Modell arbeiten, untereinander kommunizieren können.

Voicechat: Das System könnte den Anwendern ermöglichen, untereinander durch Sprachnachrichten zu kommunizieren.

Speicherung von Logindaten Das System könnte einem Anwender beim Laden Änderungen seit der letzten Öffnung des Diagramms anzeigen.

Teil III

Architektur

Dieses Dokument dient als Beschreibung der grundlegenden Architektur des Kotelett-Projektes zur kollaborativen Modellierung. Die Architektur ist grob nach dem Prinzip der Siemens-Sichten erstellt. Zunächst gibt es die globale Analyse in Kapitel 6, die wesentliche Abhängigkeiten für die Architektur darstellt.

Darauf folgt die Konzeptsicht int Kapitel 7, aus der der grundsätzliche Aufbau der Software hervorgeht. Danach werden verwendeten Technologien in Kapitel 8 vorgestellt, da diese die folgende Modulsicht beeinflussen. Die Modulsicht aus Kapitel 9 verfeinert die Konzeptsicht. Sie beschreibt die Architektur genauer und näher an der Implementierung. Dabei wird zum Beispiel die Art und Weise, wie Fremdsoftware (z.B. DOLSA [KW14]) in das Projekt integriert werden soll, festgelegt. Schließlich wird das verwendete Metamodell in Kapitel 10 vorgestellt.

Kapitel 6

Globale Analyse

Angelehnt an dem mit den Siemens-Sichten verbundenen Vorgehensmodell sollte zunächst eine Globale Analyse durchgeführt werden. Aufgrund eines Personalausfalls ist diese nicht fertiggestellt worden, wordurch die Verbindung zwischen Anforderungen und Architektur fehlt. Dieses Kapitel beschreibt die unvollständige und veraltete Globale Analyse.

6.1 Einführung

Zu Beginn der globalen Analyse wurden die wesentlichen Faktoren erhoben. Dies sind Faktoren die Einfluss auf die gesamte Struktur der Architektur haben und nicht innerhalb einer Komponente abgedeckt werden können. Hierzu zählen organisatorische Faktoren, also Faktoren, die direkt aus dem Unternehmen an sich erwachsen, technischer Faktoren wie allgemeine oder spezielle Hardware, Architektur-Technologien, Standards sowie die einflussreichsten Faktoren, die Produktfaktoren, die sich auf die Fähigkeiten und Eigenschaften des Produktes selbst beziehen.

Die Faktoren werden mit ihrer Flexibilität, also der Umfang des Einflusses des Entwicklers auf den Faktor und ihrer Variabilität, also der Möglichkeit einer externen Änderung gekennzeichnet. Flexibilität dient zumeist der Charakterisierung der organisatorischen, die Variabilität der technischen sowie beide zusammen der Produktfaktoren. Das letzte Merkmal eines Faktors ist die Wirkung, die er auf andere Faktoren bzw. durch seine Änderung hat. So kann für jeden Faktor festgestellt werden, in welche Komponenten er oder seiner Änderungen in der Architektur berücksichtigt werden müssen. Unten sind die wichtigsten (einflussreichsten) Faktoren aufgeführt, die sich im wesentlichen aus dem Inhalt der Zielvereinbarung sowie eigener Überlegungen innerhalb der Architekturgruppe des Teams ergeben haben.

Viele Faktoren, besonders funktionale Eigenschaften lassen sich relativ einfach als Komponenten in die Architektur integrieren. Oft kommt es aber auch vor, dass sich aus mehreren Komponenten spezifische Probleme erge-

ben, für die dann eine spezielle Lösungsstrategie entwickelt werden muss. In unserem Fall z.B. wirft der enge Zeitplan Probleme auf, für die eine Lösung gefunden werden muss. Aus der Analyse der Faktoren und den Strategien wurden dann die funktionalen Komponenten für die Konzeptsicht erstellt, sowie deren Verbindungen, den Konnektoren. In der Konzeptsicht wurden zunächst nur die Funktionen des Systems berücksichtigt. Diese wurde dann zur Modulsicht erweitert, die implementierungsspezifische Elemente enthält.

6.2 Organisatorische Faktoren

Die wichtigsten organisatorischen Faktoren beeinflussen die Architektur dahingehende, dass diese den Kauf von externer Software und somit seinen Einsatz in der Architektur untersagen, der enge Zeitplan und das kleine Team eine eigene Implementierung aller Komponenten nicht gestattet. Die Funktionalität kann verhandelt werden, was den Umfang an Komponenten in der Architektur ebenfalls verhandelbar macht.

Faktor	Flexibilät / Variabilität	Wirkung
O1: Management		
O1.1: Beschaffen oder Entwickeln		
Sinn des Projektes ist die Entwicklung, es gibt kein Budget, aber Fremdsoftware ohne Nutzungskosten ist möglich	Keine Flexibilität	Wirkung auf die einsetzbare Fremdsoftware moderater Einfluss auf Produktumfang durch großen Umfang der Eigenentwicklung
O1.2: Zeitplan oder Funktionalität		
Zeitplan	Feste Projektdauer von 1 Jahr	Große Wirkung auf das gesamte Projekt(Planung, Produktumfang)
O4: Zeitplan		
O4.1: Time to Market		
Feste Deadline (31.03.2014)	Änderung der Deadline nicht mögilch	Großer Einfluss auf das gesamte Projekt
O4.2: Lieferung von Funktionalität		
Funktionalität ist durch den Auftraggeber bestimmt	Funktionalität ist verhandelbar	Einfluss auf die gesamte Entwicklung
O4.3: Auslieferungsplan		

Erstellung einer Roadmap	Roadmap muss hinsichtlich des Endtermins und des erzielten Fortschritts stetig angepasst werden	Großer Einfluss auf umzusetzende Funktionalität
O5: Budget		
O5.1: Mitarbeiter		
7 Projektteilnehmer	Ausstiege möglich	Einfluss auf das gesamte Projekt
O5.2: Werkzeuge		
kein Budget	Feste Vorgabe	Einfluss auf einsetzbare Fremdsoftware

Wie bereits oben erwähnt, gehen mehrere Faktoren auf einen zu engen Zeitplan zurück, dieses Problem soll hier mit Hilfe von Strategien gelöst werden.

Enger Zeitplan
Die Zeit bis zum Abschluss des Projektes ist durch entstandene Verzögerungen begrenzt. Eine vollständige Entwicklung in der gegeben Zeit mit dem vorhandenen Team ist nicht möglich. **beeinflussende Faktoren** • O1.2: Zeitplan ist fest und nicht verhandelbar, eine komplette Entwicklung eine Systems zur kollaborativen Modellierung mit allen Fähigkeiten würde schätzungsweise mindestens ein weiteres Jahr dauern. • O1.1: Der Sinn der Projektes ist die Entwicklung und nicht der Zusammenkauf des Systems. • O1.4: Ziel des Projektes ändert sich oft. • O4.1 Das Ende des Projektes ist auf den 31.03.2015 begrenzt. • O5.1 Zahl der Projektteilnehmer ist bei 7, aber noch nach unten flexibel.
Lösung Die vollständige Implementierung des geforderten Systems würde schätzungsweise ein weiteres Jahr dauern. **Strategie: Nutzung von Frameworks** Die Implementierung einer Benutzerschnittstelle durch ein zu evaluierendes Framework verkürzt die Zeit für die Implementierung. **Strategie: Fremdentwicklung** Die Anpasung von DOLSA an die Anforderungen des Projektes ist mit der vorhandenen Teamgröße nicht zu realisieren. Das System wird vom Entwickler an das Projekt angepasst. **Strategie: Anpassung der Systemfähigkeiten** Die Implementierung des vollständigen UML-Standarts im System ist nicht zu schaffen, stattdessen wird das System auf die Modellierung von Klassendiagrammen, mit künftiger Erweiterbarkeit, angepasst. Die Fehlerbehandlung wird rudimentär ausgeführt und die Anwenderfreundlichkeit wird vernachlässigt.

Leichtes Hinzufügen und Entfernen von Komponenten

Zahlreiche Fähigkeiten und Eigenschaften des Systems sind verhandelbar und müssen somit in die Struktur des Systems eingefügt werden, außerdem verlangt der Auftraggeber nach einer Möglichkeit zur Austauschbarkeit von Konfliktlösungsstrategien und Mechanismen zur Erzeugung von DOL-Befehlen. Austauschbare Komponenten müssen unbedingt frühzeitig bei der Planung der Architektur berücksichtigt werden.

beeinflussende Faktoren

- O4.2: Funktionalitäten im System sind mit dem Auftraggeber verhandelbar.
- T3.3.3: Der Auftraggeber fordert zwingend die Umsetzung der Client-Server-Synchronisation via GMoVerS' DOL.
- T4.2.3: Für die Lösung von Synchronisationskonflikten und die Erzeugung der DOL-Befehle fordert der Auftraggeber die Austauschbarkeit von Strategien.
- T5.4.3: Der Auftraggeber fordert zwingend eine Behandlung von Konflikten.
- P5.1.1: Zur Vorbereitung syntaktischer Prüfungen von Nutzereingaben muss das System erweiterbar sein.
- P5.1.2: Die Art der Lösung von Synchronisationskonflikten ist verhandelbar.

Lösung

Um die Integration und Entfernung von Komponente zur ermöglichen, muss das System in Aufgaben-bezogene Komponenten zerlegt werden, um Teilmodule dieser Aufgaben ein- und ausbauen zu können. Weiterhin muss ein Architekturmuster gefunden werden, um Strategien austauschen zu können.

Strategie: Aufteilung der Komponenten nach dem Prinzip der Belange

Um die Auswirkung der Integration und Entfernung von Komponenten so gering wie möglich zu halten wird das System nach den Hauptanforderungen aufgeteilt. So ist sichergestellt, dass bei Änderungen in einer Komponente, andere Komponenten möglichst wenig beeinflusst werden.

Strategie: Nutzung des Strategy-Entwursmusters

Um die Möglichkeit zu haben Strategien auszutauschen, biete sich das Strategy-Muster nahezu an, da dies für solche Aufgaben entwurfen wurde.

6.3 Technische Faktoren

Viele Hardwarefaktoren werden in Java durch die JVM gekapselt. Bemerkenswert sind bei den technischen Faktoren zahlreiche externe Anforderungen des Auftraggebers, der in der Regel nur funktionale Aspekte des Systems definiert.

Faktor	Flexibilität / Variabilität	Wirkung
T3.1: Betriebssysteme		
Windows, Mac-OS X, Linux	Vorgabe durch den Auftraggeber, Anforderung aus der Gruppe	Wirkung auf den Buildprozess
T3.2: Benutzerschnittstelle		
Maus und Tastatur	langfristig stabile Schnittstelle	Großer Einfluss auf die Benutzerschnittstelle
T3.3: Software-Fremdkomponenten		
T3.3.1 Der Auftraggeber fordert den Einsatz von JGraLab	stabile Vorgabe	Großer Einfluss auf die Datenhaltung
T3.3.2 Der Auftraggeber fordert eine generische Metamodellimplementierung auf Client	stabile Vorgabe	Großer Einfluss auf Client
T3.3.3 Der Auftraggeber fordert die Umsetzung der Synchronisation via DOLSAs DOL	stabile Vorgabe	Großer Einfluss die Synchronisation
T3.3.5 Der Auftraggeber fordert, dass DOLSA zur Modellveratung verwendet wird	stabile Vorgabe	Großer Einfluss auf die Modellverwaltung
T3.3.4 Einsatz des GEF-Frameworks zur Erstellung des Editors	verhandelbar, eigene Evaluation	Großer Einfluss auf den Client
T3.4: Entwicklungssprache		
Java	stabile Vorgabe aufgrund der verwendeten Technologien	Großer Einfluss auf das gesamte System
T4: Architektur		
T4.2: Architekturmuster		

Modell-View-Controller	Vorraussetzung für GEF	Einfluss auf das gesamte System
Austauschbarkeit von Strategien	verhandelbar	Einfluss auf die gesamte Synchronisation

T5: Standards		
T5.1: Schnittstellen zum Betriebssystem		
JVM	wird von Java vorgegeben	Großer Einfuss auf das gesamte System
drei verschiedenes Betriebssysteme	Vorgabe des Auftraggebers, Anforderung der Gruppe	moderaten Einfluss auf den Buildprozess
T5.3: Datenformate		
XMI 2.1	Wird von DOLSA verwendet, feste Vorgabe	Einfluss auf das Modell und Werkzeuge
T5.4: Kommunikation (Protokolle)		
T5.4.2 Single Point of Truth	feste Vorgabe	Einfluss auf das gesamte System
T5.4.3 Konfliktbehandlung	voll verhandelbar	Einfluss auf das gesamte System
T5.4.4 Beschränkung auf GraphUML	feste Vorgabe durch Verwendung in JGraLab	Einfluss auf Modell

Die Anforderungen des Auftraggbers schränken viele technische Möglichkeiten ein und binden Personal, dem soll mit den folgenden Strategien entgegengewirkt werden.

Die Produktfaktoren und besonders die funktionalen Eigenschaften sind die wichtigsten Faktoren bei Erstellung der Konzeptsicht, da für viele Funktionen eigene Komponenten benötigt werden.

Technische Anforderungen des Auftraggebers

Zur zukünftigen Realisierbarkeit einer Syntaxüberprüfung auf den verteilten Clients verlangt der Auftraggeber ein lokale Version des Metamodelles beim Client in Verbindung mit JGraLab, sowie einen Single Point of Truth.

beeinflussende Faktoren

- T3.3.1: Der Auftraggeber fordert zwingend den Einsatz von JGraLab.
- T3.3.2: Der Auftraggeber fordert fordert zwingend eine Metamodellimplementierung auf Client & Server.
- T3.3.3: Der Auftraggeber fordert zwingend die Umsetzung der Client-Server-Synchronisation via DOLSAs DOL
- T5.4.2: Der Auftraggeber fordert zwingend einen Single Point of Truth des Modells.

Lösung

Zur Lösung der technischen Anforderungen des Auftraggabers werden verschiedene Strukturen in die Architektur intergriert.

Nutzung von Java als Programmiersprache

DOLSA verlangt durch die Intergration von JGraLab zwingend den Einsatz von Java. Eine Nutzung von verschiedenen Programmiersprachen im System würde einen Risiko-behafteten Overhead bedeuteten. Weiterhin wurde mit GEF bereits ein Framework in Java für die Erstellung des Editors evaluiert.

Strategie: Nutzung von GEF

Ein Metamodell in JGraLab verlangt Java als Programmiersprache für den Client, so dass statt eines bisher realisierten Webclients in Javascript ein Java-Framework für die Diagrammerstellung verwendet werden muss. GEF wurde bereits von der Gruppe evaluiert und ist für diesen Zweck geeignet.

Strategie: Nutzung von MVP

Die Nutzung des Architekturmuster Modell-View-Presenter ist Vorraussetzung für GEF. Außerdem kann durch die Extrahierung des Modells die Verwendung von JGraLab realisiert werden.

Strategie: Trennung von Client und Server

Um einen Single Point of Truth zu realisieren, wird ein Server eingesetzt, um die gültige Version des Modells zu speichern.

Strategie: Nutzung von DOLSA als Blackbox

Um die vom Auftraggeber verlangte Synchronisation via DOL zu ermöglichen wird zu Vereinfachung das gesamte DOLSA auf dem Client gehalten und die Services Delta Calculator und Delta Applier daraus verwendet.

6.4 Produkt Faktoren

Faktor	Flexibilität / Variabilität	Wirkung
P1: funktionale Eigenschaften		
P1.1 Erstellung von UML-Modellen per GUI	Umfang ist auf einzelne Diagramme beschränkt, einzelne Feinheiten sind verhandelbar	Einfluss auf das gesamte System
P1.2 Kollaboration	Umfang der Kollaboration ist begrenzt verhandelbar / feste Vorgabe	Einfluss auf das gesamte System
P1.3 Versionierung von UML-Modellen	feste Vorgabe	Einfluss auf die Datenhaltung
P1.4 Echtzeitsynchronisation	feste Vorgabe	Einfluss auf das gesamte System
P1.5 Modellverwaltung	feste Vorgabe	Einfluss auf die Datenhaltung
P1.6 mehrere Modelle	feste Vorgabe	Einfluss auf die gesamte Architektur
P4: Verlässlichkeit		
P4.1: Verfügbarkeit		
P4.1.1 Örtlich unabhängige Verwendbarkeit des Systems	feste Vorgabe	Einfluss auf die gesamte Architektur
P5: Fehlererkennung-, Dokumentation und Wiederherstellung		
P5.1: Diagnose		
P5.1.1 Syntaktische Prüfung der Nutzereingaben	völlig verhandelbar	Einfluss auf den Client
P5.1.2 Lösung von Synchronisationskonflikten	eingeschränkt verhandelbar	Einfluss auf Synchronisation

Mehrere verteilte Modelle im System

In dem System werden viele verschiedene Modelle verwendet. Weiterhin sind die Benutzer des Systems im System verteilt, in Verbindung mit der zwingenden Implementierung eines Metamodells auf jedem Client ergibt das auch eine technischer Verteilung des Systems. Im System soll zwingend DOL verwendet werden. Daher muss die Wahrung der Konsistenz und Widerspruchsfreiheit der verschieden Modelle in der Architektur berücksichtigt werden. Eine Modellverwaltung ist zwingende Anforderung des Auftraggebers. Diese muss in der verteilten Architektur aufgespalten werden.

beeinflussende Faktoren

- T3.3.3: Vorgabe von DOLSAs DOL zu Synchronisation.
- T3.3.5: Vorgabe von DOLSAs DOL zu Modellverwaltung.
- T4.2.1: Vorgabe einer Metamodellimplementierung auf allen Clients.
- T5.4.2: Vorgabe eines Single-Point-of-Truth
- P1.3: Zwingende Versionierung von UML-Modellen
- P1.5: Zwingende Realisierung einer Modellverwaltung.
- P1.6: Die vorgegeben Fähigkeit des Systems mit mehreren Modellen umgehen zu können

Lösung

Die Durchführung des Zugriffs auf und die Anzeige der Diagramme eines Modelles beim Benutzer werden, wie die Auswahl auf verschiedene Modelle auf dem Client realisiert. Der Server verwaltet die verschiedenen Modelle und ihre Nutzer. Es müssen Komponenten zur Erzeugung und Versendung von DOLs auf dem Client und dem Server realisiert werden. Das Repository muss DOL durch Komponenten berechnen und anwenden können. **Strategie: Implementierung einer DOL Komponente auf den Clients, dem Server und dem Repository**
Da diese verteilten Komponenten untereinander in DOL kommunizieren, muss auch die Erzeugung, Versendung und Verarbeitung von DOL auf allen drei Teilsystem implementiert sein. Clients erhalten Komponenten zum senden und Empfangen von Komponenten. Der Server hat Komponenten, um diese DOL-Befhele zu Empfangen und zu verteilen. Das Repository muss zu Verwaltung DOLs berechnen und anwenden können.
Strategie: Zentrale Modellverwaltung beim Server
Der Server verwendet Komponenten zur Verwaltung der aktiven Modelle auf den Clients. Weiterhin verteilt er die Modellliste durch eine Komponente an die Clients und hat eine Komponente für die Nutzerverwaltung. Auf den Clients wird nur das aktive Modell gespeichert und durch eine Komponente für die Behandlung der Nutzereingaben die Bearbeitung dessen ermöglicht. Weiterhin wird eine Komponente für die Auswahl eines Modells benötigt. Das Repository enthält die Komponenten zur Behandlung der verschiedenen Modelle und ihrer Versionen.

Die so gefunden Faktoren und Strategien wurden nun bei der Erstellung der Konzept- und der Modulsicht verwendet.

Kapitel 7

Konzeptsicht

In diesem Abschnitt wird aus den Hauptanforderungen an das zu entwickelnde System ein grober Architekturentwurf erstellt. Unter Berücksichtigung der Faktoren und Lösungsstrategien aus der globalen Analyse in Kapitel 6 erfolgt hier bereits eine Unterteilung in Komponenten und sehr grobgranulare Module um herauszustellen, welche Subsysteme nötig sind, um die geforderten Grundfunktionalitäten zu erfüllen.

7.1 Hauptfunktionalitäten des Zielsystems

Ausgehend von den Merkmalen, die in der globalen Analyse aufgestellt wurden und den bekannten Anforderungen wird im zweiten Schritt eine Konzeptsicht erstellt. Diese soll die Komponenten des Systems und die Systemabgrenzung gegenüber Fremdsystemen aufzeigen.

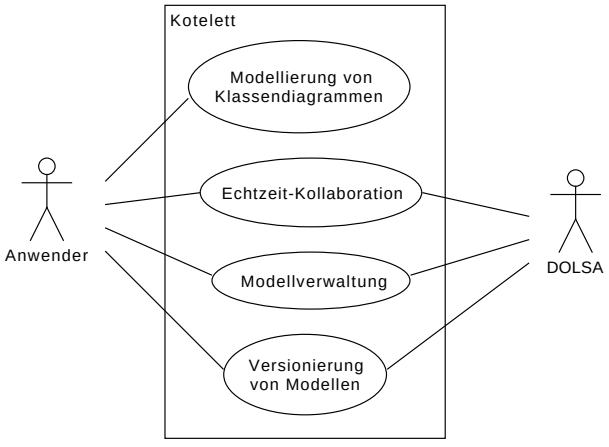

Abbildung 7.1: Konzeptsicht des Projektes

Wie bereits in der Anforderungsdefinition beschrieben, stellt das System dem Benutzer 4 Hauptfunktionalitäten bereit (siehe Abbildung 7.1).

Die erste Funktionalität ist die **Modellierung von Klassendiagrammen**. Das System muss dazu eine Möglichkeit der Modellierung bieten. Die Modellierung ist dabei, nach Absprache mit dem Auftraggebern, auf Klassendiagramme reduziert.

Die zweite Funktionalität ist die **Echtzeit-Kollaboration**. Damit die Anwender zusammen an einem Diagramm arbeiten können, wird eine Komponente mit Kollaboration benötigt. Diese Komponente wird um Echtzeit erweitert, damit die Anwender gleichzeitig und ohne Blockaden arbeiten können.

Die dritte Funktionalität ist die **Modellverwaltung**. Diese ermöglicht dem Anwender die Modelle zu erstellen, zu bearbeiten oder zu löschen.

Die vierte Funktionalität ist die **Versionierung von Modellen**. Damit sollen die Modelle um eine Historie erweitert werden. Dies ermöglicht dem Anwender Versionen von dem Modell zu erstellen und zu älteren Versionen zurückzukehren.

Als **Fremdsystem** wird der Projektgruppe von der ST-Abteilung DOLSA [KW14] zur Verfügung gestellt. Dieses System soll zur Realisierung der Kollaboration, Modellverwaltung und der Versionierung verwendet werden. Dadurch wir der Projektgruppe ein Teil des Implementierungsaufwandes abgenommen. Für DOLSA bietet das zu entwickelnde System wiederum einen Anwendungsfall in dem die entwickelten Services in der Praxis erprobt werden können.

7.2 Zerteilung der Hauptfunktionalitäten

Im Folgenden sollen die in Abschnitt 7.1 genannten Hauptfunktionalitäten zerlegt werden, um deutlich zu machen machen, welche Teilaufgaben an welcher Stelle im System behandelt werden.

Aus der zweiten Hauptanforderung des Anforderungsdokumentes ergibt sich, dass eine verteilte Architektur für das Zielsystem benötigt wird. Zusätzlich hat der Auftraggeber einen **Single Point of Truth** gefordert, was eine Client-Server-Lösung ergibt. Somit wird das zu entwickelnde System in eine Client- und eine Server-Komponente unterteilt. Darüber hinaus wird das zu entwickelnde System aus einer Eigenentwicklung und einer Anbindung des Fremdsystems DOLSA bestehen.

Entsprechend können die Hauptfunktionalitäten auf diese drei Komponenten, Client, Server und externes System (DOLSA) aufgeteilt werden.

7.2.1 Modellierung von Klassendiagrammen

Die Modellierung von Klassendiagrammen zerfällt in zwei Teilaufgaben: Die Anzeige sowie die Verarbeitung von Nutzereingaben. Beide Aufgaben werden

auf dem Client erledigt.

7.2.2 Echtzeit Kollaboration

Bei der Echtzeit Kollaboration sind sowohl Client als auch Server beteiligt. Der Client erstellt unter Nutzung der DOL Services Deltas der Nutzeränderungen. Er verschickt diese Deltas an der Server und empfängt Änderungen der anderen Nutzer sowie Kontrollnachrichten vom Server. Die empfangenen Deltas lässt er durch den DOL Service anwenden.

7.2.3 Modellverwaltung

Nutzereingaben und Anzeigen der Modellauswahl werden vom Client behandelt. Dazu hält der Client das aktuelle Modell. Der Server verwaltet die aktiven Modelle und die dazu gehörigen Nutzer. Sollte sich die Modellliste ändern, so bringt er die Clients auf den neusten Stand. Die DOL Applikation stellt CRUD-Operationen für Modelle bereit.

7.2.4 Versionierung

Nutzereingaben und Anzeigen der Versionen wird vom Client übernommen. Der Server bringt die Clients auf den neusten Stand, sollte sich die Versionsliste ändern. Die DOL Application ermöglich es Versionen zu speichern und wieder zu laden, alle Versionen aufzulisten, sowie zu einer Version das aktive Delta abzufragen.

7.3 Komponenten zur Umsetzung der Hauptfunktionalitäten

Aus Abschnitt 7.2 geht hervor, dass jede Hauptfunktionalität von einer oder mehreren Komponenten bereitgestellt wird. Durch die Zerlegung von vorausgesetzten Komponenten in Client, Server und Fremdsystem ergibt sich die Abbildung 7.2.

Die Komponente **Kotelett** stellt dabei das zu entwickelnde System in seiner Gesamtheit dar. Die Komponente **DOLSA** ist das verwendete Fremdsystem und stellt Services zur Realisierung der geforderten Funktionalität bereit.

Die in den Komponenten enthaltenen Module, die jeweils einer Hauptfunktionalität entsprechen, sind farblich hinterlegt, um ihre Zusammengehörigkeit zu verdeutlichen. Darüber hinaus werden Module, die in einer Komponente vorrangig verarbeitet werden mit einem durchgezogenem Rechteck und Module, die nebensächlich verarbeitet werden mit einem gestricheltem Rechteck abgebildet.

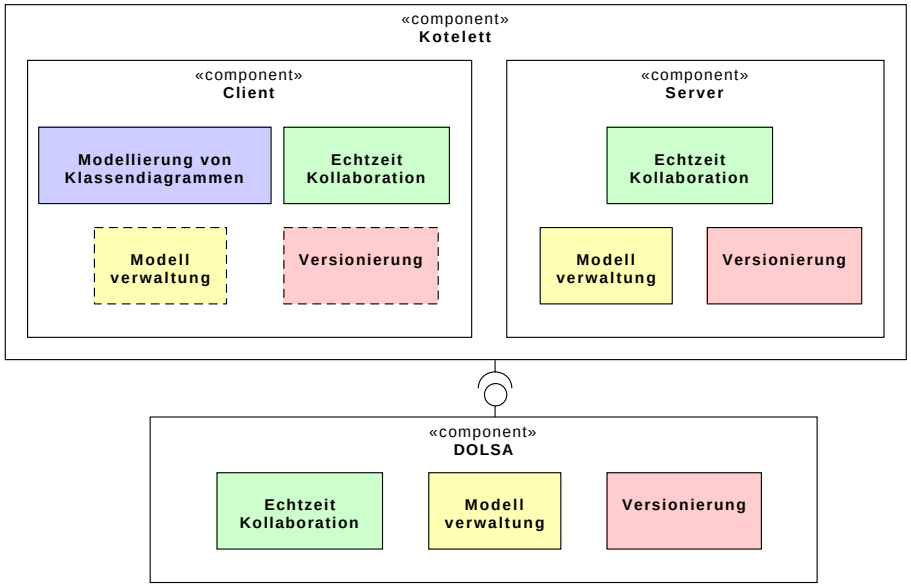

Abbildung 7.2: Aufteilung der Hauptfunktionalitäten zu Komponenten der Zielarchitektur

Die Komponente **Client** ist hauptsächlich für die Modellierung von Klassendiagrammen und die Echtzeit-Kollaboration verantwortlich. Die Echtzeit-Kollaboration ist eine Hauptaufgabe, da der Client das Verschicken der Änderung von Modellen zum Server initialisieren muss. Die beiden anderen Module Modellverwaltung und Versionierung sind Nebenaufgaben, da der Client diese nur visualisiert und dem Anwender eine Schnittstelle zur Interaktion bietet. Die Entscheidungsgewalt, ob solche Aktionen durchgeführt werden liegt letztendlich beim Server.

Die Komponente **Server** beschäftigt sich nicht mit der Modellierung von Klassendiagrammen. Die Module Echtzeitkollaboration, Modellverwaltung und Versionierung stellen die Hauptaufgaben dieser Komponente dar. Der Server muss die Clients miteinander synchronisieren. Der Server ist zudem der Single Point of Truth und kann gegebenenfalls die Synchronität der Clients erzwingen. Der Server entscheidet bei einer Anfrage des Clients ob ein Modell erstellt bzw. verändert und ob von einem Modell eine neue Version gespeichert werden soll.

Die **DOLSA-Komponente** bietet eine Reihe von Services zur Realisierung von Echtzeit-Kollaborativität, Modellverwaltung und Versionierung. Diese Services werden sowohl von der Client- als auch von der Server-Komponente verwendet.

7.4 Verfeinerung der Hauptfunktionalitäten

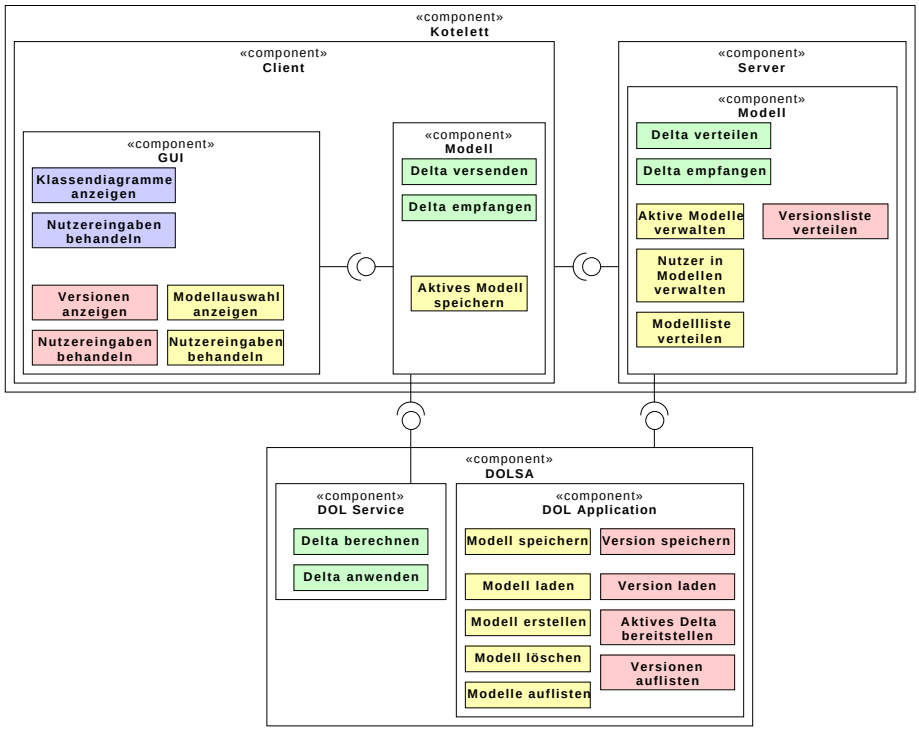

Abbildung 7.3: Verfeinerte Architektur

Wie bereits in Abschnitt 7.3 angedeutet, können bereits auf dieser Granularitätsebene des Architekturentwurfs, Unterfunktionalitäten aus den Hauptfunktionalitäten abgeleitet werden. Dies ist in Abbildung 7.3 dargestellt. Die Farbgebung zur Verdeutlichung von zusammengehörigen Modulen, welche der gleichen Hauptfunktionalität dienen (vgl. Abschnitt 7.3) wird hier beibehalten.

Die wesentlichen Änderungen der verfeinerten Architektur in Abbildung 7.3 sind die Aufteilung des Clients und des Fremdsystems DOLSA. Der Client wurde zur sauberen Trennung von Daten und Anzeige in GUI und Modell aufgeteilt. DOLSA wurde in DOL Services und DOL Application aufgeteilt, wobei die DOL Services von Client und Server verwendet werden und zustandslos sind. Die DOL Application wird nur vom Server verwendet und persistiert Modelle und deren Versionen.

Desweiteren können die Hauptfunktionalitätsmodule, bspw. Modellierung von Klassendiagrammen in der Komponente Client, weiter unterteilt werden. In der GUI-Komponente des Clients werden getrennte Module zur Anzeige und zur Verarbeitung von Nutzerinteraktionen sowohl für Klassen-

diagramme, Versionen als auch Modellauswahl benötigt.

Die Kommunikation zwischen Client und Server besteht aus dem bidirektionalen Senden und Empfangen von Deltas was zwei Modulen mit ähnlichen aber doch unterschiedlichen Aufgaben entspricht. Die Modellverwaltung zerfällt in die reine Verwaltung und Bereitstellung vorhandener Modelle sowie die Zuordnung von Nutzern zu Modellen.

Das Fremdsystem DOLSA stellt verschiedene Module für die notwendigen Funktionalitäten bereit. Als Services reichen die Module zur Berechnung und Anwendung von Deltas aus. Für die Modellverwaltung werden im Fremdsystem Module zum Speichern, Laden, Erstellen und Löschen von Modellen, sowie die Bereitstellung vorhandener Modelle benötigt. Für die Versionierung muss es möglich sein, Versionen zu Speichern und zu Laden, alle vorhandenen Versionen und aktive Deltas abzurufen.

Aufbauend auf dieser funktional basierten, konzeptionellen Zerlegung der Komponenten kann im Kapitel 9 die Modulsicht aufgebaut werden. Ziel ist es, Technologieentscheidungen zu ermöglichen und mögliche Probleme bei der Umsetzung der Anforderungen an das Softwaresystem bei der geplanten Architektur aufzudecken.

Zunächst werden aber im Kapitel 8 die Technologien erläutert, die für die Umsetzung der vorgestellten Funktionalität notwendig sind. Anschließend werden diese Technologien in der Beschreibung der Modulsicht im Kapitel 9 verwendet.

Kapitel 8

Verwendete Technologien

In diesem Kapitel werden die Technologien vorgestellt, die zur Erstellung des Produkts eingesetzt wurden.

DOLSA (Abschnitt 8.4) und **JGraLab** (Abschnitt 8.3) müssen verwendet werden, da diese von den Auftraggebern verlangt werden. DOLSA muss dabei zur Synchronisation, Modellverwaltung und Versionierung verwendet werden. JGraLab muss zur Repräsentation und Verwendung eines Metamodells auf dem Client und Server verwendet werden.

Daraus ergibt sich die Abhängigkeit, dass der Client und Server in Java implementiert werden müssen. Bei den Untersuchungen nach einem Modellierungstool aus dem ersten Halbjahr, war bereits **GEF** (Abschnitt 8.2) als Kandidat für ein Java-Modellierungstool gewählt worden.

Für die Implementierung benötigt GEF ein Metamodell für die Datenrepräsentation. JGraLab kann dieses Metamodell stellen und ist auch mit GEF kompatibel.

Somit passt GEF sehr gut zu den geforderten Technologien und wird für die Implementierung des Clients verwendet.

Darüber hinaus kennt sich das Teammitglied Jan Becker sehr gut mit Eclipse aus und hat sich bereits vertieft in GEF eingearbeitet. Somit besitzt das Team eine Person, die ggf. für Fragen bereit steht und diese auch beantworten kann.

Die Abbildung 8.1 soll die Zuordnung zwischen den definierten Funktionalitäten aus Kapitel 7 mit den benötigten Technologien erleichtern.

Für die Realisierung der grafischen Benutzeroberfläche werden **SWT** (Abschnitt 8.1) und **GEF** (Abschnitt 8.2) verwendet (siehe Abbildung 8.1). Die Darstellung der Diagramme und die Möglichkeit diese zu verändern wird als Editor für Klassendiagramme mittels GEF umgesetzt. Der Editor wird in eine übergeordnete GUI eingebettet, in der sich die Modell- und die Versionsauswahl befinden. Diese übergeordnete GUI wird mit SWT umgesetzt. Durch die Entscheidung, GEF als Framework für den Editor zu verwenden, wird die verwendung von SWT vorausgesetzt.

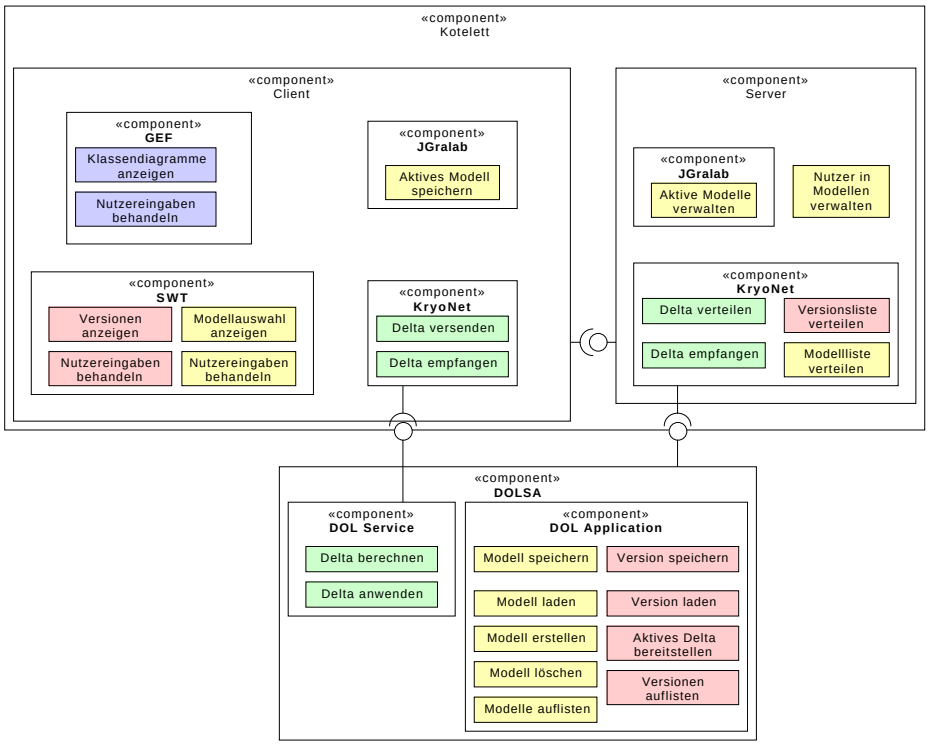

Abbildung 8.1: Zuordnung der Technologien

Zur Realisierung des Modells auf dem Client und Server wird **JGraLab** (Abschnitt 8.3) verwendet. Dabei speichert der Client sein aktuelles Modell und der Server verwaltet alle aktiven Modelle. Um die Modelle und dazugehörige Versionen zu persistieren verwendet der Server die **DOL Application** aus **DOLSA** (Abschnitt 8.4).

Für die Realisierung des Datenaustauschs zwischen Client und Server wird **KryoNet** (Abschnitt 8.5) verwendet. Verschicket werden Modellinformationen und einzelne Änderungen eines Modells. Für die Repräsentation der Modelländerungen wird **DOL** verwendet, welche ebenfalls im Abschnitt 8.4 beschrieben wird. Zur Generierung und Anwendung der DOL stellt DOLSA die **DOL Services** dem Client und dem Server zur Verfügung (siehe Abbildung 8.1).

8.1 SWT und JFace

SWT (Standard Widget Toolkit) und JFace sind Teil des Standard-GUI-Frameworks, das in Eclipse-Produkten eingesetzt wird [MLA10, Ecl13]. Da der Diagrammeditor in Kotelett mithilfe des Graphical Editor Frameworks (GEF, [RWC11]) implementiert wird (siehe Abschnitt 8.2) und GEF auf SWT basiert, sollte SWT ebenfalls für den Rest der GUI eingesetzt werden. In Kotelett wird SWT daher für die gesamte GUI (Baumansichten, Dialoge, usw.) außerhalb des Klassendiagramm-Editors eingesetzt. So ist eine optimale Integration mit GEF möglich.

SWT stellt Standard-Steuerelemente wie Buttons, Eingabefelder etc. bereit. Im Gegensatz zu Swing nutzt SWT native Elemente des Betriebssystems, im Gegensatz zu AWT allerdings mit eigenem Look-And-Feel, d.h. die Steuerelemente haben statt des betriebsystemspezifischen Aussehens, wie es bei AWT direkt übernommen wird, das von Eclipse gewöhnte Layout.

Die Eclipse Plugin API arbeitet in Verbindung mit SWT und stellt (Ober-)Klassen bereit, um Views, Editoren und Menüeinträge zu erstellen.

In Eclipse sind Views Fenster, die Informationen visualisieren, wie z.B. der Projekt-Explorer oder das Konsolenfenster. Diese werden standardmäßig am Rand des Hauptfensters angeordnet. Von jeder View existiert zur Laufzeit gewöhnlich nur eine Instanz. In Kotelett basieren z.B. die Baumansicht und die Anwenderliste (siehe Abbildung 9.8 in Abschnitt 9.5) auf Eclipse Views.

Editoren sind Fenster zur Bearbeitung von Inhalten, wie der Java-Editor. Sie werden standardmäßig in Tabs in der Mitte der Anwendung angezeigt. Von einer Editor-Klasse kann es zur Laufzeit beliebig viele Instanzen geben. Die Diagrammeditoren in Kotelett sind solche Editoren.

Die GUI ist in Eclipse immer sehr modular aufgebaut. Editoren und Views werden getrennt voneinander entwickelt. Gewöhnlich werden diese nicht im Code instanziiert. Stattdessen werden Views und Editoren mittels

einer XML-Datei in der Anwendung registriert und dann von der Eclipse-Laufzeitumgebung automatisch erstellt und angeordnet.

8.2 GEF

Zur Darstellung der Diagramme wird mindestens eine Zeichen-Bibliothek benötigt. Dazu kommt GEF (Graphical Editing Framework) zum Einsatz, da es bereits im Rahmen des Projektes evaluiert wurde und als beste Lösung zum Zeichnen von Diagrammen in Java betrachtet wurde. Hierzu wurde ein Kriterienkatalog aufgestellt. Nähere Informationen dazu können dem Prozessbericht (Beschreibung der ersten Iteration) entnommen werden.

Bei GEF handelt es sich um eine auf dem Eclipse-Framework basierende Lösung zum Erstellen von Diagramm-Editoren. Es gibt bereits eine View-Model-Struktur vor. Die für das Kotelett-Projekt wesentlichen Struktur-Elemente werden im Folgenden grob beschrieben. Für tiefergehende Informationen muss auf [Ecl14, RWC11] verwiesen werden.

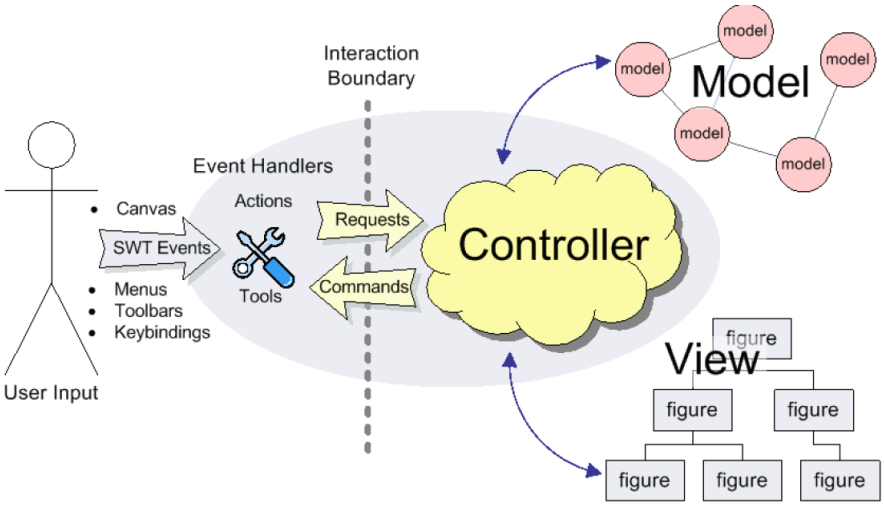

Abbildung 8.2: Übersicht über GEF. Quelle: `http://help.eclipse.org/luna/topic/org.eclipse.gef.doc.isv/guide/guide.html`

8.2.1 Visualisierung

Wie in Abbildung 8.2 dargestellt, schreibt GEF eine Art Model-View-Control vor. Dabei macht GEF keine Vorgaben, wie das Modell auszusehen hat. Als Konsequenz müssen Layoutinformationen im Modell gehalten bzw. aus diesem abgeleitet werden. Der Modellbegriff meint bei GEF damit also eher den Diagrammbegriff, wie er im Rahmen dieses Projekts geprägt wurde. Im folgenden wird dazu trotzdem der Begriff Modell verwendet.

Die Schnittstelle zwischen Modell und visueller Darstellung mit Draw2d bilden *GraphicalEditParts*. *GraphicalEditPart* ist die Oberklasse für diese Controller-Klassen. GEF nimmt an, dass für jedes Modellelement, das dargestellt wird und mit dem Nutzer interagieren soll, ein EditPart existiert. Die EditParts bilden somit eine Baumstruktur, die parallel zur Struktur des Modells und der View ist (siehe Abbildung 8.3.

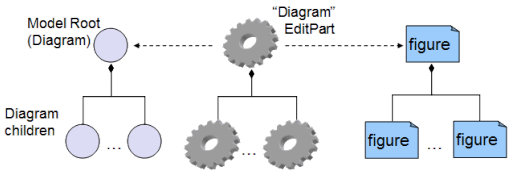

Abbildung 8.3: Modellelemente, EditParts und Figures. Quelle: `http://help.eclipse.org/luna/topic/org.eclipse.gef.doc.isv/guide/guide.html`

Draw2d ist ein mit GEF verbundenes Objektorientiertes Zeichen-Framework, mit dem die konkrete Darstellung erfolgt. In Draw2d wird eine Abbildung baumartig aus Formen (Figures) und ihren Kindern sowie Kanten zwischen Figures aufgebaut.

GEF baut die View dann grob nach folgendem Algorithmus auf. Um die EditParts zum Modell zu instanziieren, muss eine Factory implementiert werden.

1. Das Wurzel-Objekt des Modells wird an den Editor/Viewer übergeben.
2. Mithilfe der Factory wird der EditPart instanziiert und bekommt das Modellobjekt zugewiesen.
3. Das GEF-Framowork ruft spezifizierte Methoden (wie getModelChildren()) des EditParts auf, um Kinder des Modellelements sowie Kanten zu ermitteln die ebenfalls visualisiert werden müssen.
4. Schritt 2 und 3 werden rekursiv wiederholt, bis das gesamte Modell aufgebaut ist.

Alle GraphicalEditParts haben zudem eine Methode getFigure(), die das Draw2d-Figure-Objekt zurückgibt, das in der View angezeigt werden soll. Diese bekommt als Kinder automatisch die Figures der Kinder des EditParts. Die Top-Level Figure wird dann zusammen mit ihren Kindern gezeichnet. So wird die gesamte View aufgebaut.

8.2.2 Anwenderinteraktion

Die Anwenderinteraktion läuft ebenfalls über die EditParts. Jedes EditPart implementiert eine oder mehrere Schnittstellen, genannt EditPolicies, die

als Event-Listener fungieren und als Antwort auf eine Eingabe des Anwenders ein Command-Objekt zurückgeben. Auf diese Weise sind in Kotelett fast sämtliche Bearbeitungsschritte in Diagrammen, wie z.B. das Erstellen und Verschieben von Klassen und Assoziationen, implementiert. Diese Commands sind Teil eines Command-Pattern. Sie werden von GEF in einem Command-Stack verwaltet, was ein UNDO und REDO des Anwenders ermöglicht. Diese Funktionen sind in GEF standardmäßig vorhanden.

Die Commands verändern bei ihrer Ausführung in der Regel das Modell, nicht die View. Die View wird danach lediglich aktualisiert.

Die Events, die bei den EditParts ankommen, sind keine einfachen Maus-Events, sondern liegen bereits eine Abstraktionsebene höher. GEF wandelt Eingaben abhängig vom gerade ausgewählten Werkzeug (GEF bringt bereits eine Palette an Standardwerkzeugen mit, die dem Anwender zur Verfügung gestellt werden können) z.B. in Events zum Erstellen oder Bewegen eines Modellobjekts um. Eingaben können also weitestgehend auf Modellebene behandelt werden.

8.3 Modellrepresäntation mit JGraLab

Aufgrund mehrerer Faktoren ist es notwendig eine geeignete Technologie zur Modellrepräsentation zu wählen:

- Das verwendete Diagrammbearbeitungs-Framework GEF (s. Abschnitt 8.2) schreibt zur Implementierung das MVC-Pattern vor. Das Modell (im Sinne von MVC) ist dabei nicht vorgeschrieben aber notwendig. Im Rahmen des Projekts ist das Modell des GEF-MVC ebenso das Modell im Sinne der kollaborativen Modellierung.

- Die Zielanwendung ist ein Produkt zur (kollaborativen) Modellierung. Die unterstützten Modellarten sollen dabei erweiterbar sein.

- Der Auftraggeber wünscht explizit die Ablage des Modells und die Verwendung eines Metamodells auf Client und Server.

- Der Auftraggeber wünscht explizit die Verwendung des in Abschnitt 8.4 beschriebenen Systems DOLSA zur Modellversionierung. Da dieses System metamodell-generisch entwickelt wurde, ist die Verwendung eines Metamodells zwingend erforderlich.

Entsprechend wird eine Modelltechnologie benötigt, mit der sich Klassendiagramme und deren Metamodell repräsentieren lassen. Ein Abwägung von Alternativen ist für diese Technologie nicht notwendig da der Auftraggeber explizit die Verwendung vom Java Graphen Labor (JGraLab) fordert und das zwingend zu verwendete Fremdsystem DOLSA auf Basis von JGraLab arbeitet.

Entsprechend verwendet die Anwendung, sowohl in der Client- als auch der Server-Komponente, zur internen Modellrepräsentation das Framework JGraLab zur Behandlung von TGraphen [Kah06]. TGraphen sind typisierte, geordnete und gerichtete Graphen. Sie eignen sich mit diesen Eigenschaften zur Darstellung/Repräsentation jeglicher Modelle was bereits in mehreren Arbeiten als auch innerhalb von DOLSA erprobt wurde. Desweiteren ist durch diverse Abschlussarbeiten und nebenberufliche Erfahrung bereits Expertenwissen zu JGraLab und dessen Funktionalität im Team vorhanden.

Dieser Abschnitt widmet sich der Abgrenzung zwischen dem zu entwickelnden System und JGraLab und beschreibt, welche Komponenten von JGraLab im Rahmen der Anwendung verwendet werden.

8.3.1 TGraphen und Schemata

Die grundlegendste Aufgabe von JGraLab besteht in der Bereitstellung einer API zur programmatischen Handhabung von TGraphen. TGraphen sind Graphen, gemäß dem mathematischen Konzept, die folgende Eigenschaften erfüllen:

- **Typisierung:** Jeder Knoten und jede Kante des Graphen ist typisiert.
- **Attribute:** Jeder Knoten und jede Kante des Graphen kann Attribute besitzen.
- **Ordnung:** Alle Knoten und Kanten des Graphen sind geordnet, so dass es eine ausgezeichnete Ordnung gibt in der über die Graphelemente iteriert werden kann.
- **Richtung:** Jede Kante im Graphen ist gerichtet kann aber birektional genutzt werden. Es ist also zu jeder Kante möglich von ihrem Zielknoten zu ihrem Startknoten zu gelangen und umgekehrt.

TGraphen genügen einem eindeutigen Graphschema, dem Metamodell für TGraphen. In diesem wird festgelegt, welche Typen es gibt, welche Attribute diese besitzen und wie Elemente miteinander mittels Kanten verbunden werden können.

8.3.2 grUML

Die Modellierungssprache Graph UML (grUML) erlaubt das Erstellen von Metamodellen (Schemata) für TGraphen mithilfe von UML Klassendiagrammen. Diese können bspw. mit dem Rational Software Architect erstellt, als XMI exportiert, und in JGraLab importiert werden. JGraLab erstellt dann automatisch eine generische API die die Manipulation und Betrachtung des Schemas auf Programmebene erlaubt. Dies kann zur Kompilier- oder zur Laufzeit erfolgen. Ebenso wird von JGraLab eine generische API zur

Handhabung der Graphelemente einer Instanz des Schemas erzeugt. Da die Zielvereinbarung des Projekts die Umsetzung von UML-Klassendiagrammen vorsieht, ist davon auszugehen, dass das Metamodell im Laufe des Projekts größtenteils unverändert und bei kleineren Änderungen nur Teile der API regeneriert werden müssen. Entsprechend wird eine zur Kompilierzeit generierte API verwendet.

8.4 Generic Model Versioning System und Delta Operation Language

Wie aus den technischen Faktoren der globalen Analyse hervorgeht, sollen laut Auftraggeber zur Modellsynchronisation die Komponenten des momentan in der zur Forschung entwickelten Delta Operations Language with Services and Applications (DOLSA) verwendet werden. In diesem Abschnitt wird zum besseren Verständnis die Grundfunktionalität und Architektur von DOLSA beschrieben. Zudem wird beschrieben, welche Komponenten und zu welchem Zweck diese genutzt werden.

Wie bereits erwähnt handelt es sich bei DOLSA um ein System zur Modellversionierung ähnlich zu textbasierten Ansätzen die bspw. zur Quellcodeversionierung in Softwareprojekten genutzt werden. Bei einem Modellversionierungssystem müssen aber die Semantiken von Modellen berücksichtigt werden, was durch mögliche textuelle Repräsentationen der Modelle und Versionierung dieser nicht ohne weiteres möglich ist.

Grundlegend für die Versionierungsfunktionalität ist die Handhabung von Differenzen zwischen Versionen eines Objekts anstatt jede Version in seiner Gesamtheit zu verarbeiten. Für die Repräsentation von Modelldifferenzen verwendet DOLSA einen operationsbasierten Ansatz der zunächst beschrieben werden soll da diese Repräsentation die Grundlage für DOLSA darstellt und außerdem zur Modellsynchronisation im Rahmen des Projekts verwendet werden soll.

8.4.1 Delta Operation Language

Zur Repräsentation von Modelldifferenzen wird in DOLSA ein operationsbasierter Ansatz mit dem Namen *Delta Operations Language (DOL)* verwendet. Dieser Ansatz basiert auf Operationen von den Basistypen Hinzufügen, Ändern und Löschen. Jede Operation kann sich auf ein Modellelement, z.B. eine bestimmte Instanz eines Kanten- oder Knotentyps, oder auf Attribute eines solchen Elements beziehen. Damit ist es möglich, jede Art von Differenz zwischen konsekutiven Versionen des gleichen Modells auszudrücken. Die genaue Syntax der DOL-Operationen zu einem Modell folgt direkt aus dem Metamodell des behandelten Modells.

Zum Beispiel kann man die Änderung eines Methodennamens in einem Klassendiagramm durch eine Methode der Form

```
g0.changeMethodName("newName")
```

repräsentieren. Hier gibt `g0` die durch den Namen eindeutige zu ändernde Instanz an. Die Punktnotation, wie aus objektorientierten Sprachen wie Java gewohnt, gibt an, dass eine Operation auf diesem Objekt durchgeführt wird. Der `change`-Teil der Operation bedeutet, dass eine Änderung vorgenommen wird. `Method` spezifiziert den Elementtypen und `Name` den Namen des zu ändernden Attributs. In Klammern ist mit `"newName"` der neue Wert des zu ändernden Attributs als Parameter angegeben.

Ein ähnliches Schema wird zum entfernen von Elementen angewendet, nur dass hier auf Attributspezifikation und Parameter verzichtet werden kann. Beim hinzufügen von Elementen entfällt bei der Operation die Punktnotation und stattdessen wird das Ergebnis der Operation, das neue Element, mittels eines Zuweisungssoperators `=` einem neuen, modellweit eindeutigen Identifier, bspw. `g1`, zugewiesen.

Im Rahmen dieser Projektgruppe soll erprobt werden, ob sich dieser Ansatz auch zur Synchronisation von kollaborativer Echtzeitmodellierung nutzen lässt. Entsprechend werden Änderungen die ein Nutzer an einem Modell auf seinem Client vornimmt, lokal in Modelldifferenzen gemäß der DOL-Repräsentation umgewandelt und in dieser Form an den Server und ggf. weitere Clients gesendet.

8.4.2 DOLSA Gesamtüberblick

In Abbildung 8.4 ist die Gesamtstruktur des Systems mit allen relevanten Komponenten dargestellt. Das System besteht aus drei Teilen:

1. **DOL Generation:** Bereitstellung der DOL
2. **DOL Services:** Atomar nutzbare Services zur Handhabung von Modelldifferenzen in DOL
3. **DOL Applications:** Versionierungsmodule zur Modellverwaltung und -analyse

Diese Teilsysteme und deren Relevanz für die Entwicklung des Zielsystems werden einzeln in den folgenden Abschnitten betrachtet.

8.4.3 DOL Generation

Die Syntax des in Unterabschnitt 8.4.1 beschriebene Modelldifferenzrepräsentation DOL ist direkt vom Metamodell des repräsentierten Modells abhängig. Beispielsweise müssen Methoden zur Änderung von Elementen ent-

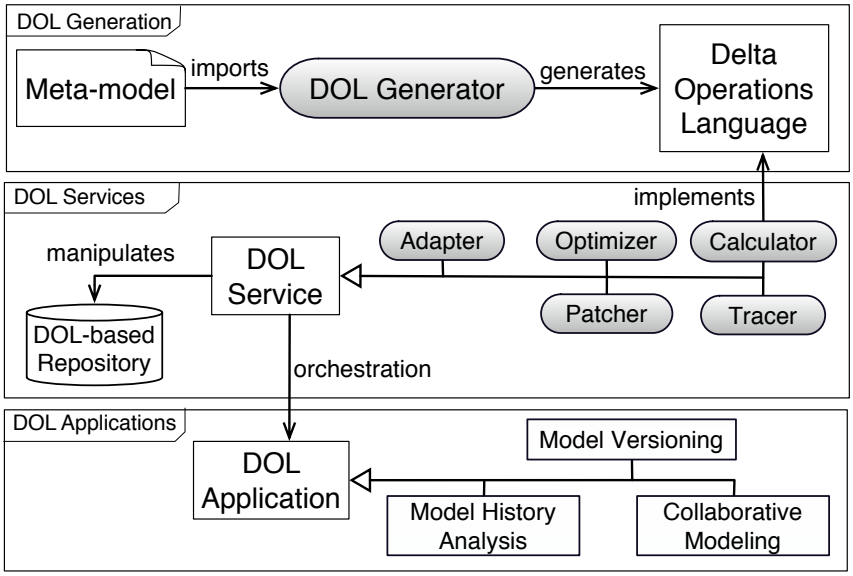

Abbildung 8.4: Übersicht über das Gesamtsystem DOLSA [Kur14]

sprechend ihres Typs benannt werden. Im Subsystem DOL Generation verfügt DOLSA über die Möglichkeit, den genauen Sprachumfang zu einem Metamodell direkt aus diesem heraus zu generieren. Dies erfolgt im *DOL Generator*.

Der Generator erhält als Eingabe ein Metamodell in Form eines TGraphenschemas und generiert daraus eine Java-API die dann den DOL Services zur Verfügung gestellt wird.

Wie bereits in Abschnitt 8.3 dargelegt, wird der Umfang des Metamodells aufgrund der Reduzierung auf Klassendiagramme begrenzt sein und sich im Verlauf des Projekts nur geringfügig ändern. Das gleiche gilt entsprechend für die vom Generator erzeugte DOL.

Zu klären bleibt, ob die Übersetzung des Metamodells in DOL von außen durchgeführt werden muss, oder ob die DOL Services die Übersetzung selbstständig durchführen. Dies stellt aber kein Hindernis für den Architekturentwurf dar.

8.4.4 DOL Services

Das Subsystem der von DOLSA bereitgestellten DOL Services dient der Verarbeitung und Erstellung von Modelldifferenzen in DOL-Repräsentation. Das Subsystem enthält die folgenden fünf Services:

1. **Calculator:** Berechnet die Differenz zwischen zwei Versionen des gleichen Modells und liefert eine Modelldifferenz in DOL

2. **Patcher:** Appliziert eine Modelldifferenz in DOL zu einem Modell und liefert das geänderte Modell zurück
3. **Tracer:** Erkennt und verbindet Modellelemente über mehrere Versionen hinweg
4. **Optimizer:** Optimiert Modelldifferenzen in Bezug auf Redundanz
5. **Adapter:** Schnittstelle zu externen Modellierungstools um Modelle zu im- und exportieren

Das **DOL-based Repository** dient im Kontext von DOLSA zur Persistierung von Modellen und Änderungen wird jedoch im Rahmen des Projekts nicht direkt als Service verwendet werden.

Für das Zielsystem sind einzig die Services Calculator und Patcher relevant. Mit ihnen lassen sich die bei Modelländerung zu versendeten Modelldifferenzen ermitteln und auf anderen Systemen auf das Modell anwenden. Somit kann mithilfe dieser zwei Services die Modellsynchronisierung zwischen mehreren Clients und dem Server umgesetzt werden.

8.4.5 DOL Application

Das Subsystem DOL Application stellt letztendlich den Versionierungsservice im Sinne der Verwaltung von Modellen und Versionen zur Verfüng. Ursprünglich ist dieses Subsystem eine Orchestrierung der DOL Services um Versionierung zu ermöglichen. Da im Rahmen dieses Projektes zwei Services der DOL Services direkt genutzt werden, ist aus der DOL Application lediglich die Persistierung von Versionen und Modellen sowie deren Abruf relevant.

8.5 KryoNet

Zur Übertragung der Daten zwischen Server und Client wird eine Technologie benötigt, die das Netzwerk aus Java heraus benutzbar macht. KryoNet [Kry14] bietet die Möglichkeit Javaobjekte von zur Kompilierzeit festgelegten Klassen über TCP oder UDP Verbindungen zu verschicken. Die Serialisierung der Objekte wird automatisch gehandhabt, kann aber vollständig oder auch nur für einzelne Klassen ausgestauscht werden, wenn dies notwendig sein sollte. Die Schnittstelle für den Programmier besteht aus mehreren Listenermethoden für das Eintreffen eines Objekts sowie den Auf- und Abbau einer Verbindung. Es bietet auch die Möglichkeit Remote Method Invocation zu verwenden. Die Einfachheit in der Benutzung wird von anderen Lösungen nicht erreicht, außerdem haben einige Projektmitglieder bereits Erfahrung in der Verwendung von KryoNet. Aus diesem Grund wird Kryonet zur Datenübertragung verwendet.

Kapitel 9

Modulsicht

Die Modulsicht baut auf die vorab angefertigte Konzeptsicht (siehe Kapitel 7) auf.

Ausgehend von den vorgestellten Funktionalitäten und Technologien, soll in diesem Kapitel die Realisierung und dabei auftretende Probleme erläutert werden.

Im Abschnitt 9.1 wird die Umsetzung der Architektur erläutert.

Im Abschnitt 9.2 werden auftretende Probleme vorgestellt und klassifiziert. Daraufhin werden in Abschnitt 9.3 und Abschnitt 9.4 Lösungsansätze zur Behebung der Probleme vorgestellt.

Im Abschnitt 9.5 wird die Realisierung der Modell- und Versionsverwaltung und im Abschnitt 9.6 das umgesetzte Parsen vorgestellt.

9.1 Model-View-Presenter

Model-View-Presenter (MVP) ist ein Architektur Entwurfsmuster in der Softwareentwicklung. Dieses gibt den groben Rahmen für die Realisierung vor.

MVP besteht aus drei Komponenten: Modell, View und Presenter. Das Modell stellt die Logik bereit. Die View bietet eine grafische Darstellung und ermöglicht es dem Anwender eingaben zu tätigen. Der Presenter verbindet das Modell und die View. Er leitet die Eingaben aus der View an das Modell und Änderungen aus dem Modell an die View weiter.

Durch die Verwendung von MVP wir die Logik von der Darstellung entkoppelt. So wird eine klare Trennung in der Umsetzung ermöglicht.

Die Umsetzung von MVP wird in der Abbildung 9.1 dargestellt. Das MVP wird zwei mal in der Architektur angewendet.

Das erste MVP umfasst die gesamte Kotelett-Komponente. Der Server ist dabei das Modell und der Client die View. Der Presenter wird auf den Client und Server zerteilt, damit ein Datenaustausch zwischen diesen beiden

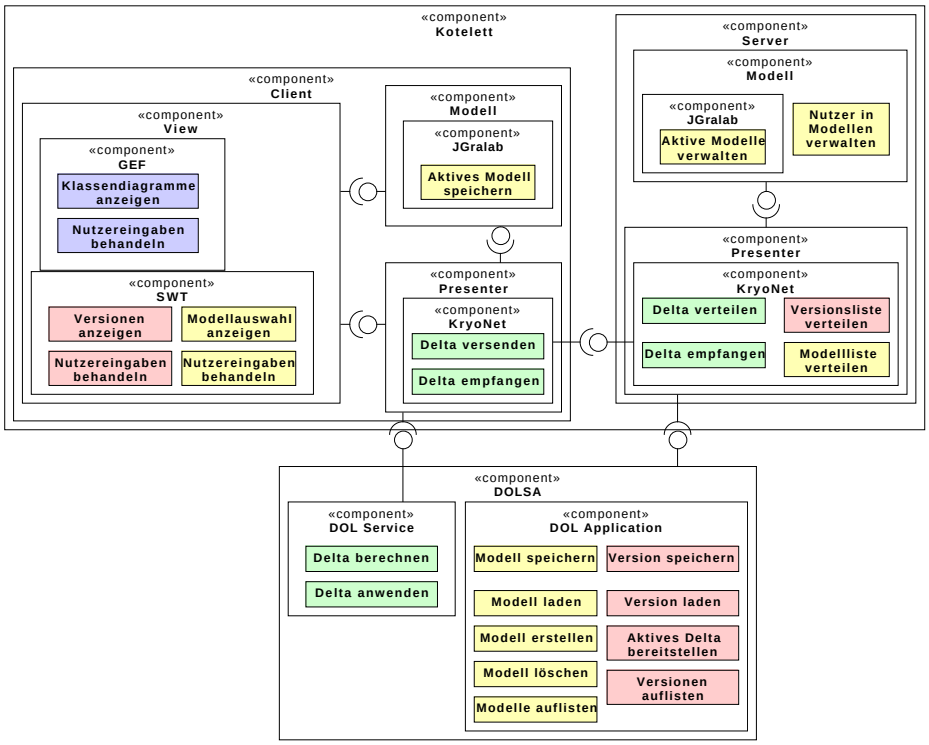

Abbildung 9.1: Verteilung der Module auf das Model-View-Presenter-Pattern

Komponenten stattfinden kann, wenn diese auf unterschiedlichen Systemen liegen.

Das zweite MVP umfasst nur die Client-Komponente. Hier sind die Unterkomponenten bereits nach MVP benannt (siehe Abbildung 9.1).

Der Presenter wird von beiden MVPs verwendet. Der Kotelett-MVP verwendet den Presenter um Nachrichten zwischen den Client und Server zu verschicken. Der Client-MVP verwendet den Presenter um Eingaben des Anwenders von der View an das Modell weiter zu leiten und angefragte Informationen von dem Modell an die View weiter zu geben.

DOLSA Kommunikation

Wie in der Abbildung 9.1 zu entnehmen ist, verwendet der Presenter aus dem Kotelett-MVP die DOLSA Komponente.

Zum verschicken der Änderungen zwischen den Clients und dem Server wird die DOL Service Komponente verwendet. Diese berechnet die Änderungen und verpackt diese in eine Nachricht. Die versendete Nachricht kann anschließend ebenfalls mit DOL Service auf das Modell angewandt werden.

Darüber hinaus verwendet der Serverteil des Presenters noch die DOL Application, um die Modelle mit den dazugehörigen Versionen zu speichern. Die DOL Application bietet auch weitere Services, damit eine vollwertige Modell und Versionsverwaltung erstellt werden kann.

Für eine schärfere Systemabgrenzung wurde eine Schnittstelle zu DOLSA mit den DOLSA Entwicklern vereinbart Abbildung 9.2. Die Schnittstelle fasst die Fähigkeiten von DOLSA bezüglich Deltaerzeugung und Anwendung, sowie Modell- und Versionsverwaltung zusammen. Die Deltaerzeugung und Anwendung ist zustandlos, DOLSA arbeitet nur auf den übergebenen Modellen. Die Modell- und Versionsverwaltung persistiert Modelle und Versionen und wird nur auf dem Server verwendet, um einen Single Point of Truth zu haben.

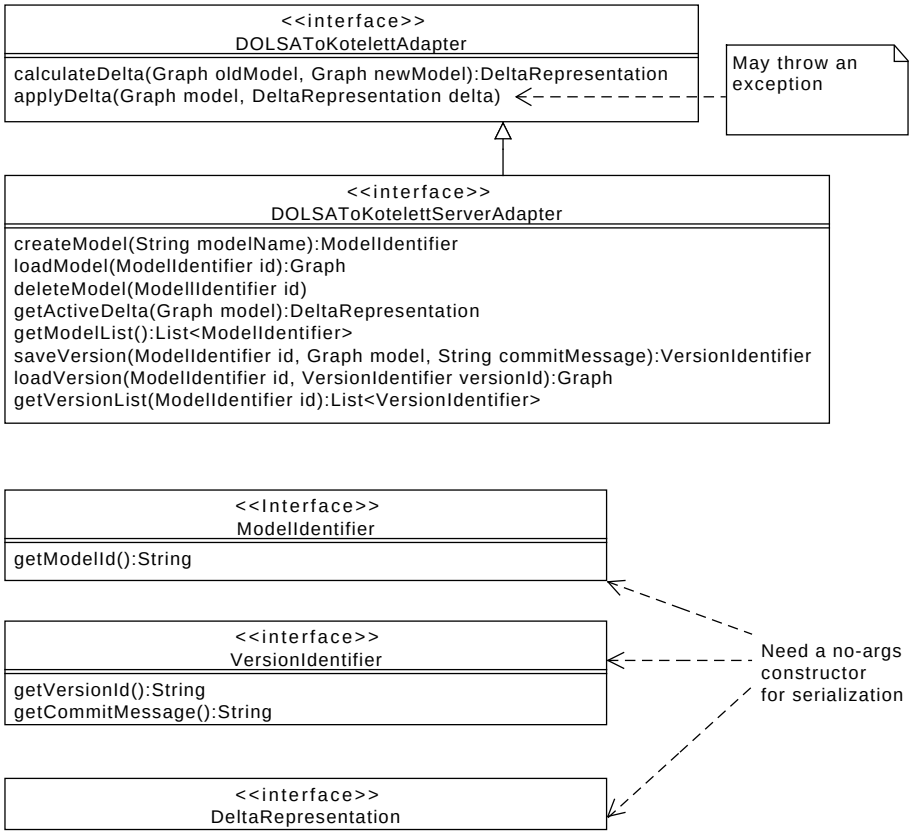

Abbildung 9.2: Vereinbarte Schnittstelle zwischen Kotelett und DOLSA

9.2 Synchronisationsprobleme des Modells

Da beim Synchronisieren der Clients Probleme auftreten können, z.B. wenn Anwender widersprüchliche Änderungen vornehmen, sollen diese in diesem Kapitel gesammelt und erläutert werden.

Die Probleme können in zwei Kategorien klassifiziert werden: Grund und Ort.

Mit Grundproblemen sind technische oder Zustandsprobleme gemeint. Technische Probleme treten auf, wenn z.B. eine verwendete Technologie interne Fehler aufweist und dadurch das Gesamtsystem gefährdet. Zustandsprobleme treten auf, wenn z.B. ungültige Änderungen auf dem Modell gemacht werden. Probleme auf dem Client treten auf, wenn z.B. der Anwender eine Änderung macht und gleichzeitig eine Änderung von dem Server ankommt. Probleme auf dem Server treten auf, wenn wenn z.B. ein Anwender ein Element löscht und ein anderer Anwender das gleiche Element ändern möchte.

9.2.1 Technische Probleme

Technische Probleme sind stark mit den verwendeten Technologien verknüpft.

Es gibt Probleme mit JGraLab (siehe Abschnitt 8.3), da dieses nicht threadsicher ist. Dies betrifft den Client und den Server gleichermaßen, da in beiden die Modellrepräsentation mit JGraLab realisiert wird. Das Problem tritt auf wenn Änderungen an dem Modell parallel geschehen. Dabei ist es irrelevant, ob beide Änderungen im Konflikt stehen. Das Modell kann einfach bei parallelen Änderungen korrumpiert werden.

Auf dem Client geschieht es, wenn der Anwender eine Änderung vornimmt und gleichzeitig eine Änderung von dem Server eingeht.

Auf dem Server geschieht es, wenn zwei Anwender gleichzeitig Änderungen zum Server schicken und dieser beide verarbeitet.

Um dieses Problem zu beheben, muss die Threadsicherheit sichergestellt werden. Die genaue Lösung wird in Abschnitt 9.3 beschrieben.

9.2.2 Zustandsprobleme

Zustandsprobleme treten auf wenn widersprüchliche bzw. ungültige Änderungen an dem Modell vorgenommen werden sollen.

Das Problem ist auf dem Client und Server nahezu gleich. Es kommen in einem sehr kurzen Abstand zwei widersprüchliche Änderungen. Auf dem Client kommt die eine Änderung von den Anwender und die anderen von dem Server. Auf dem Server kommen beide Änderungen von zwei Anwendern, die an dem selben Modell arbeiten.

Das Problem liegt darin, dass nach dem Anwenden der ersten Änderung, die zweite Änderung nicht auf das Modell angewendet werden kann. Wenn zum Beispiel die erste Änderung das Element löscht, die zweite Änderung das gleiche Element aber umbenennen möchte. Das Umbenennen kann nicht ausgeführt werden, da das Element nicht mehr existiert.

Bei der Lösung des Problems muss darauf geachtet werden, dass der Server Single Point of Truth darstellt und deswegen immer stabil sein muss.

Um dieses Problem zu beheben, müssen die Konflikte in den Änderungen aufgelöst werden. Die genaue Lösung wird in Abschnitt 9.4 beschrieben.

9.3 Serialisierung der Änderungen

Bei Mehrbenutzeranwendungen stellt die korrekte Behandlung gleichzeitiger Aktionen durch verschiedene Anwender, die sogenannte Nebenläufigkeit, eine Herausforderung dar. In diesem Projekt tritt dieses Problem insbesondere bei der Veränderung des Modells auf.

Zum einen behandelt JGralab Nebenläufigkeit nicht, so dass es hier zu undefiniertem Verhalten kommen kann. Zum anderen wird die vom Anwender durchgeführte Änderung aus der Differenz zweier Modellversionen gebildet. In diesem Fall muss sichergestellt werden, dass nur die Änderung, die der Anwender durchgeführt hat, in das Delta einfließt. Außerdem müssen vom Server eintreffende Deltas in der korrekten Reihenfolge angewandt werden.

Aus diesem Grund wurde der `ModelMonitor` geschaffen. Dieser sorgt dafür das Zugriffe auf das Modell in der richtigen Reihenfolge durchgeführt werden. Bei eingehenden Deltas nimmt er diese entgegen und speichert sie in einer Queue, damit sie in der richtigen Reihenfolge angewandt werden können. Die genau Funktionalität soll anhand zweier Sequenzdiagramme erläutert werden: Abbildung 9.3 und Abbildung 9.4.

Abbildung 9.3 zeigt die Behandlung eine Änderung auf dem Client. Zunächst wird der aktuelle Stand des Modells für die spätere Berechnung der Änderung kopiert. Nach dem Kopiervorgang wird die Änderung auf das Modell angewandt. Jetzt kann aus dem alten und neuen Modell mithilfe der DOLSA-Services das Delta gewonnen werden. Dieses Delta wird an den Server verschickt. SequenceClientServerProtokoll

Abbildung 9.4 zeigt die Behandlung einer Änderung auf dem Server. Ankommende Änderungen auf dem Server werden in eine Queue zum Arbeiten abgelegt. Der runner:Thread nimmt immer die erste Änderung aus der Queue und arbeitet diese ab. Das Delta wird mit Hilfe der DOLSA-Services auf das Modell angewandt. Daraufhin wir das Delta an dazugehörige Clients verschickt. Danach arbeitet der runner:Thread die nächste Änderung aus der Queue ab. Da DOLSA keine Fehler zurückmeldet ist der einzige Fall, wo eine Änderung nicht angewandt werden kann, wenn eine Exception von JGralab

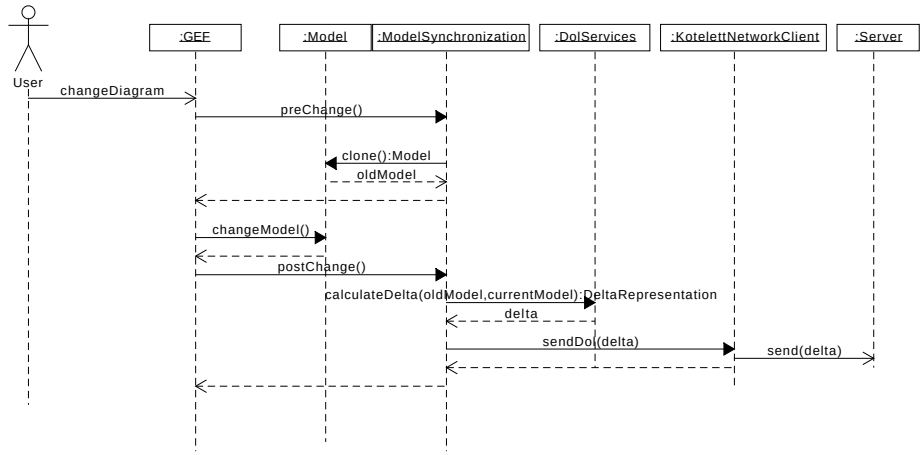

Abbildung 9.3: Send Change mit den neuen Komponenten

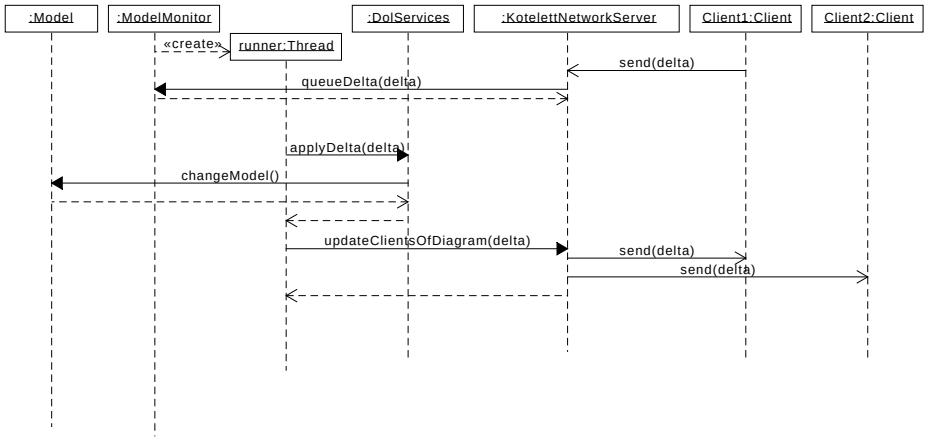

Abbildung 9.4: Server Receive Change mit den neuen Komponenten

geworfen wird. Dieser Fall wird behandelt, indem vor Anwendung des Deltas eine Kopie des Models erstellt wird und im Fehlerfall diese zurückgespielt wird, da sonst ein Delta unvollständig angewandt sein könnte.

9.4 Protokoll der Client-Server-Kommunikation

Bei der Behandlung der Nutzereingaben, in diesem Fall Modelländerungen, in einem verteilten System gibt es zwei fundamental unterschiedliche Vorgehensweisen:

1. Die Änderungen werden sofort auf dem Client angewandt und an den Server gesendet (Unterabschnitt 9.4.1).
2. Die Änderungen werden an den Server gesendet und erst angewandt, wenn dieser bestätigt hat (Unterabschnitt 9.4.2).

Diese beiden Vorgehensweisen werden im Folgenden beschrieben.

9.4.1 Sofortige Änderung

Das Protokoll der Client-Server-Kommunikation bei sofortiger Anwendung der Modeländerungen ist in Abbildung 9.5 dargestellt. Die Auswirkungen der einzelnen Nachrichten sind im Folgenden beschrieben.

- UserChangeRequest: Client sendet seine Änderungen an der Server. Enthält ein Delta in DOL.
- UserChangeMessage: Server verteilt Änderungen an die Clients. Enthält ein Delta in DOL.
- ResetRequest: Der Client fordert eine ResetModelMessage an.
- ResetModelMessage: Enthält ein ActiveDelta in DOL. Der Client verwirft sein aktuelles Modell und wendet das Delta auf ein leeres Model an.

Zur Fehlerbehandlung kann das Modell an zwei verschiedenen Stellen zurückgesetzt werden. Zum einen wenn die Änderung der Clients auf dem Server nicht angewandt werden kann. Dies tritt auf wenn eine in Konflikt stehende Änderung eines anderen Clients zuerst angewandt wurde. Zum anderen kann die Anwendung einer von Server eingehenden Änderung auf dem Client fehlschlagen. In diesem Fall treffen die beiden in Konflikt stehenden Änderung auf dem Client zusammen.

Die eben beschriebene Fehlerbehandlung legt ein Problem des Protokolls offen: Die Clients sind immer wieder asynchron. Ein deutlicher Vorteil ist allerdings, dass Nutzeränderungen sofort durchgeführt werden, so dass der Arbeitsfluss nicht unterbrochen wird. Ein wichtiger Faktor dabei ist, wie oft

die Modelle resettet werden müssen, dies kann allerdings erst am Produkt festgestellt werden.

9.4.2 Bestätigung abwarten

Das Protokoll der Client-Server-Kommunikation mit Abwarten der Bestätigung durch den Server ist in Abbildung 9.6 dargestellt. Die `RequestDenied-Message` teilt dem Client mit das seine Änderung nicht angewandt werden konnte. Die Fehlerbehandlung bei diesem Ansatz besteht nur darin, inkompatible Änderungen auf dem Server abzuweisen. Der deutliche Vorteil dieses Ansatzes ist, dass die Clients synchron bleiben, da alle Clients eine Änderung erst nach Freigabe durch den Server erhalten. Der deutliche Nachteil ist allerdings, dass die Latenz der Nutzereingaben von der Netzwerklatenz abhängt. Dadurch hängt die Benutzbarkeit des Produkts direkt von Latenz zwischen Client und Server ab.

9.4.3 Komplexere Ansätze

In der Literatur finden sich einige Ansätze die Konfliktbehandlung zu verbessern. Diese gehen vom Ansatz des Sofortanwendens aus und versuchen die Konflikte aufzulösen, indem eingehende Änderungen gepatched oder das Modell vor der Anwendung auf einen früheren Stand gesetzt wird, und danach die späteren Änderungen wiederholt werden.

9.4.4 Strategy Pattern zu Modelländerungen

In Anbetracht der verschiedenen Lösungen bietet es sich an, diese einfach austauschbar zu halten (II-NFA7). Außerdem ist es zur Erweiterbarkeit des Produktes sinnvoll, dass die Methode um die Nutzeränderung zu erfassen, austauschbar ist. Unser Ansatz für die Umsetzung des Strategy Pattern sieht folgendermaßen aus:

In Abbildung 9.7 ist dargestellt, wie eine Nutzeränderung unter Verwendung eines Strategypattern aussieht. Die auszutauschende Klasse ist der `ModelHandler`. Zunächst wird dem ModelHandler mit geteilt, dass eine Änderung ansteht. Hier kann dann entweder eine ChangeListener an das Modell gehängt oder eine Kopie erstellt werden. GEF holt dann das zu verändernde Model über den ModelHandler. Dadurch kann die Implementierung entscheiden, ob das in GEF zur Anzeige verwendete Modell verändert wird oder nicht, wodurch die zuvor beschriebenen Ansätze implementiert werden können. Nach den Änderungen am Modell, wird dem ModelHandler mitgeteilt, dass die Änderungen vollständig durchgeführt wurden. Hier wird dann das Delta aus den Daten des ChangeListeners oder per Diff erstellt und versandt.

Bei der Umsetzung ist aufgefallen, dass GEF nicht dafür ausgelegt ist das Modell auszutauschen, dies würde auch dem GEF Workflow widerspre-

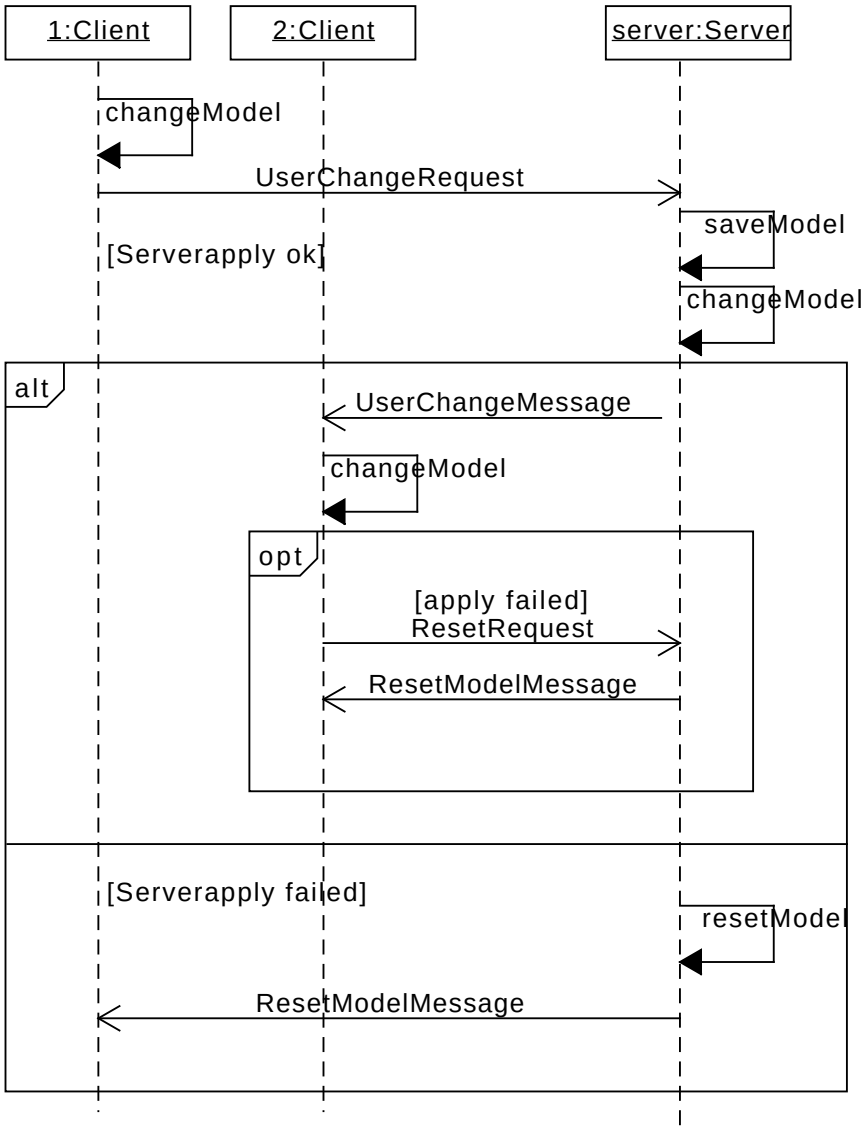

Abbildung 9.5: Client-Server-Kommunikationsprotokoll

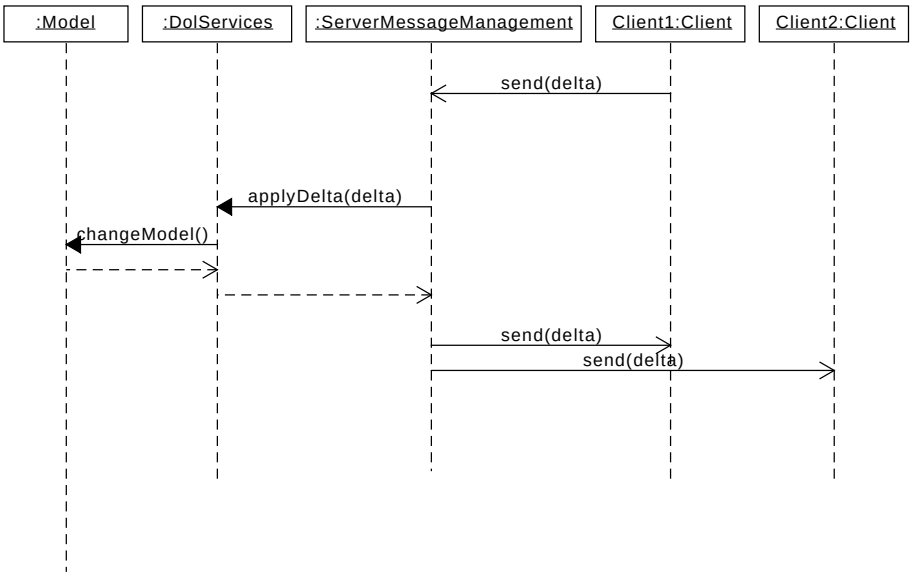

Abbildung 9.6: Client-Server-Kommunikationsprotokoll

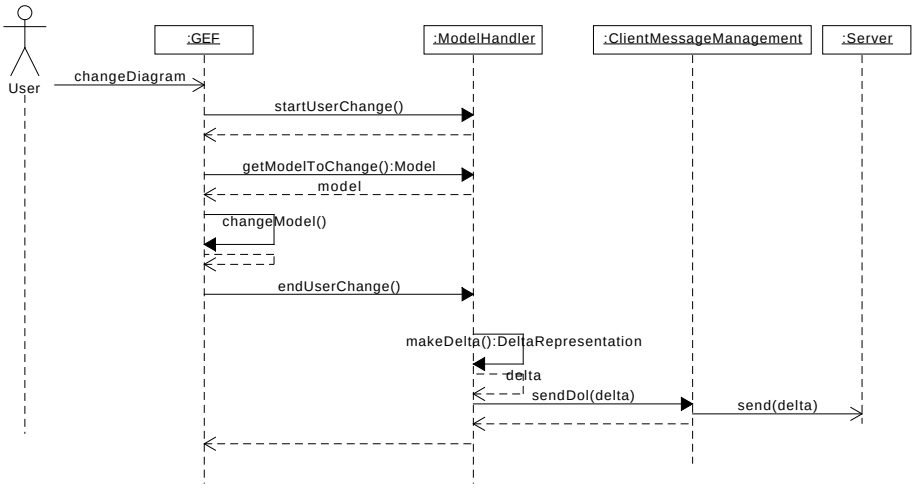

Abbildung 9.7: Ablauf einer Nutzeränderung mit austauschbarem Kommunikationsprotokoll

chen. Um die gewünschte Funktionalität zu erreichen müsste das normale Verhalten von GEF angepasst werden. Da dies mit sehr großem Aufwand verbunden und sehr fehleranfällig ist, wurde die Umsetzung des Strategy Patterns auf dem Client verworfen.

9.5 Verwaltung von Modellen, Versionen und Diagrammen

In diesem Abschnitt wird die Modellverwaltung beschrieben. Die Modellverwaltung stellt die Schnittstelle zur DOL Application dar. Sie ermöglicht es dem Benutzer damit, Modelle und Versionen zu erstellen und auszuwählen. Diese Funktionalität ist in Abbildung 7.2 und Abbildung 7.3 in Kapitel 7 gelb dargestellt. Bei der Modellverwaltung sind in erster Linie ein Teil der View des Clients (in Abbildung 9.1 als SWT bezeichnet), der Presenter vom Client und Server sowie die DOL Application involviert. In Abbildung 9.8 ist die GUI des Clients abgebildet. Zur Modellverwaltung dienen die Auswahlfelder und die Baumansicht links. Die einzelnen Editoren zum Bearbeiten der Diagramme sind in Tabs in der Mitte angeordnet.

9.5.1 Modell- und Versionsverwaltung

Der Anwender soll eine Übersicht über alle Modelle und Versionen bekommen.

Bei den Versionen ist zu beachten, dass der aktuelle Stand eines Modells gesondert von den übrigen Versionen betrachtet werden muss. Der aktuelle Stand wird als *Head*-Version bezeichnet. Bei den gespeicherten Versionen handelt es sich um Snapshots des Modells. Der aktuelle Stand ist der neuesten gespeicherten und benannten Version in der Regel voraus. Gespeicherte Versionen können nur zur Anzeige geöffnet werden, aber nicht bearbeitetet werden.

Laden von Modellen und Versionen

Der Ablauf des Ladens und Anzeigens aller Modelle und Versionen ist in Abbildung 9.9 dargestellt. Um möglichst wenig Daten auf dem Client vorhalten zu müssen, erfolgt die Auswahl einer Version eines Modells in drei Schritten:

1. Initialisierung: Beim Start des Clients werden die verfügbaren Modelle auf dem Server angefragt und deren Namen und IDs an den Client übermittelt.

2. Der Anwender wählt ein Modell aus. Die verfügbaren Versionen des Modells werden beim Server angefragt und auch hier werden Namen

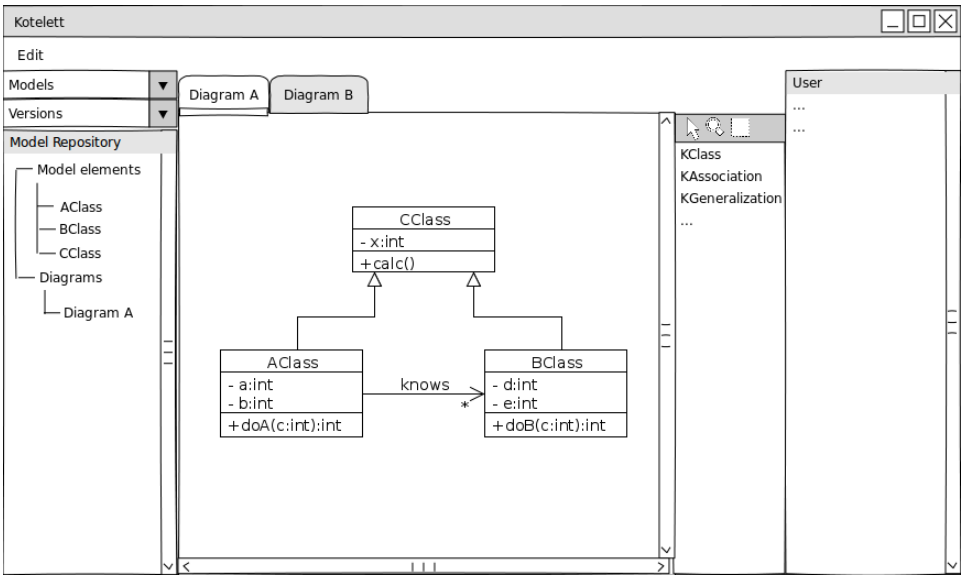

Abbildung 9.8: Mockup des Clients

und IDs zurückgegeben, die dem Anwender angezeigt werden. Als weiterer Eintrag wird die Head-Version angezeigt.

3. Der Anwender wählt eine Version aus. Je nachdem, ob die neueste Version oder eine ältere ausgewählt wird, unterscheidet sich das Laden:

   (a) Wenn die neueste Version (Head-Version) ausgewählt wird, wird das aktive Delta aus loadModel() geladen und in der Baumansicht angezeigt.

   (b) Wenn eine ältere Version ausgewählt wird, wird das aktive Delta aus loadVersion() geladen und in der Baumansicht angezeigt.

Erstellen von Modellen und Versionen

Der Anwender hat über einen Menüeintrag die Möglichkeit, neue Modelle zu erstellen. Über eine Eingabemaske wird er dabei aufgefordert, einen Namen für das Modell und das leere Diagramm, das automatisch für ein neues Modell erstellt wird, zu benennen.

Der Client sendet dann lediglich die Anfrage an den Server, ein Modell mit diesem Namen zu erstellen. Das Modell wird dann auf dem Server angelegt und alle Clients über die Existenz eines neuen Modells informiert. Dies ist in Abbildung 9.10 dargestellt.

Ebenso hat der Anwender die Möglichkeit, neue Versionen zu erstellen, d.h. den aktuellen Stand auf dem Server persistieren zu lassen. Auch hierfür gibt es einen Kontextmenü-Eintrag. Der Anwender wird aufgefordert, die

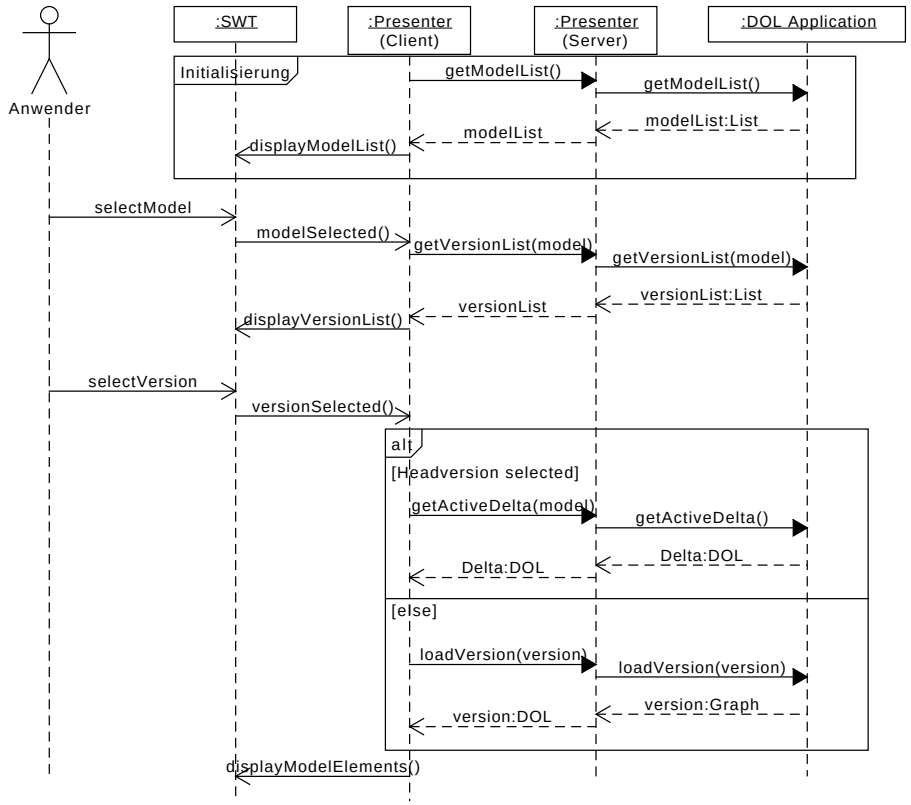

Abbildung 9.9: Sequenzdiagramm zum Anzeigen und Laden von Modellen

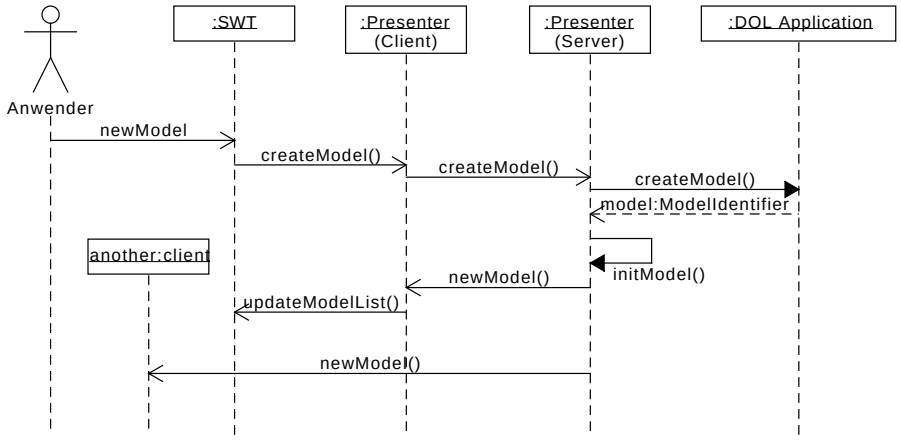

Abbildung 9.10: Sequenzdiagramm zum Erstellen neuer Modelle

Version zu benennen. Auf dem Server wird dann eine neue Version angelegt und alle Clients über die Existenz einer neuen Version informiert. Da der Editor, in dem die Bearbeitung stattfindet, nach wie vor die Head-Version referenziert, ändert sich nur in der Anzeige der Versionen etwas.

Wenn der letzte Anwender das Modell schließt, legt der Server automatisch eine neue Version an.

Zurückspringen zu einer alten Version (revert)

Der Anwender hat die Möglichkeit, eine zur Ansicht geöffnete ältere Version als neue Head-Version zu übernehmen. Dabei wird die Head-Version durch jene ältere Version ersetzt. Da dies mit Zustimmung der restlichen Benutzer geschehen soll, sendet der Client lediglich eine Anfrage an den Server. Wenn die restlichen Bearbeiter des Modells zugestimmt haben, setzt der Server die Head-Version zurück. Technisch ergibt sich dies durch die Anwendung eines Deltas auf die Head-Version. Der gesamte Ablauf ist damit wie folgt:

1. Der Client des Anwenders sendet eine Anfrage an den Server, den Head zurückzusetzen. Diese beinhaltet ewine Referenz auf die Zielversion, zu der zurückgekehrt werden soll, sowie eine Commit-Message für den aktuellen Stand des Head.

2. Der Server schickt eine Anfrage um Bestätigung an die anderen Clients, die das Modell geöffnet haben. Wenn ein Client dem Revert nicht zustimmt, wird abgebrochen.

3. Wenn alle Anwender zugestimmt haben, wird eine Version des aktuellen Heads mit der Commit-Message aus 1. erstellt, sozusagen ein Backup.

4. Der Server berechnet das Delta (Rückwärts-Delta) zwischen der Head-Version und der alten Version, die diese ersetzen soll.

5. Der Server wendet dieses Delta auf die Head-Version an und sendet es als Bearbeitungsschritt an die Clients.

Löschen von Modellen

Modelle können nur gelöscht werden, wenn sie von keinem Anwender mehr geöffnet sind.

9.5.2 Diagrammverwaltung

Da Diagramme Teil des Metamodells sind, unterscheiden sie sich aus Controller-Sicht kaum von Modellelementen. Änderungen an Diagrammen (Erstellen,Umbennenen,Löschen) durch den Synchronisationsmechanismus für Modelle verteilt. (siehe Unterabschnitt 9.4.1).

Geöffnet werden Diagramme durch einen Doppelklick auf den entsprechenden Eintrag in der Baumansicht (siehe Abschnitt 9.5). In der Baumansicht können ebenso ein Kontextmenü zum Erstellen und Entfernen von Diagrammen angezeigt werden.

Beim Erstellen eines Diagramms wird der Anwender über eine simple Eingabemaske zur Benennung des Diagramms aufgefordert. Geschlossen wird ein Diagramm über das X am Editorfenster.

9.5.3 Presenter Client

In diesem Abschnitt wird der Presenter auf dem Client (Siehe auch Abbildung 9.1) und dessen Zusammenspiel mit Modell und View genauer beschrieben. Die detaillierte Architektur ist in Abbildung 9.11 abgebildet. Herzstück bildet die `ModelRegistry`, die als Singleton implementiert ist. Mit ihr werden sämtliche Informationen zu Modellen und Diagrammen auf dem Client verwaltet. Insbesondere:

- Welche Modelle stehen zur Bearbeitung zur Verfügung
  - Welche Versionen dieses Modells gibt es
- Welche Modelle sind gerade geöffnet
  - In welchen Editoren werden diese angezeigt
  - Welche Nutzer bearbeiten das Modell gerade außerdem
- Was ist die User-ID des Clients

Die `ModelSynchronization` nutzt die DOL-Services und den `KotelettNetworkClient`, um Änderungen an einem Modell zu berechnen und an den Server zu versenden. Dieselbe Komponente ist zuständig, eingehende Deltas auf den Graphen anzuwenden. Dies wird ausführlich in Abschnitt 9.4 beschrieben. Es existiert eine Instanz pro geöffnetem Modell. Dadurch, dass `ModelSynchronization` `CommandStackListener` implementiert, und als solcher bei den `Viewern` (die Komponente eines GEF-Editors, in dem das Diagramm angezeigt und vom Benutzer bearbeitet wird), die Diagramme des Modells anzeigen, registriert ist, wird jene bei Änderungen von Nutzerseite automatisch informiert und ein Delta versendet.

Im Zusammenhang damit sorgt der `ViewUpdater` dafür, dass die `Viewer`, die jeweils ein Diagramm des Modells anzeigen, bei Änderungen des Modells aktualisiert werden. Dazu implementiert auch `ViewUpdater` `CommandStackListener`.

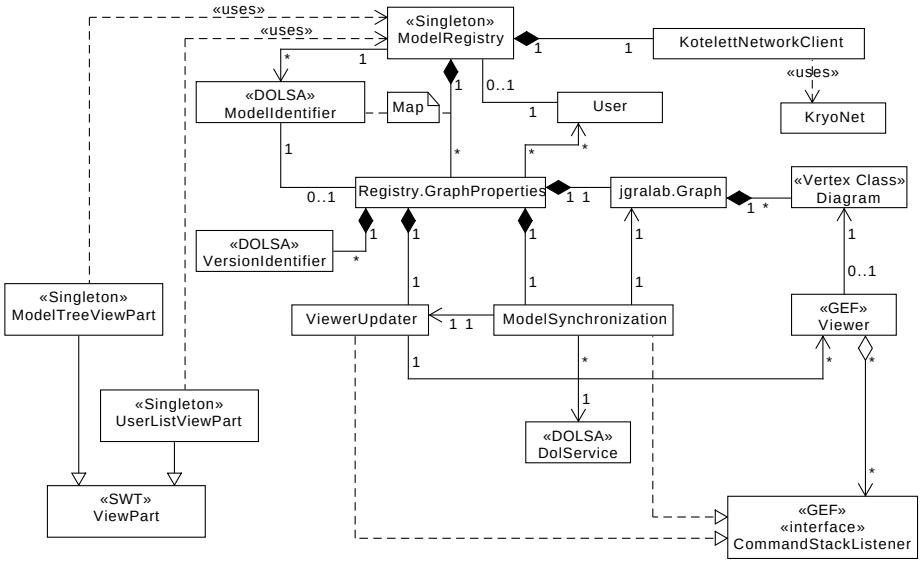

Abbildung 9.11: Klassendiagramm des Presenters auf dem Client zusammen mit dem Modell

Lifecycle für Editoren

Von der obigen Modell und Diagrammverwaltung ausgehend, kann ein Diagramm-Editor einen der Zustände „im Vordergrund“ (der aktive Editor), „im Hintergrund“ und „geschlossen“ annehmen, wie in Abbildung 9.12 dargestellt. Es ergeben sich die folgenden Übergänge: Es ist immer maximal ein Editor aktiv. Wenn ein neuer Editor aktiviert wird, wird der momentan aktive Editor in den Hintergrund verschoben. Zu einem Diagramm gibt es immer nur einen Editor. Wenn ein Diagramm geöffnet werden soll, für das bereits ein Editor existiert, wird dieser aktiviert. Ein Editor wird unter folgenden Umständen aktiviert, bzw. geöffnet:

- Wenn ein Modell geöffnet wird
- Wenn ein Diagramm per Doppelklick in der Baumansicht ausgewählt wird
- Wenn ein Modellelement unter einem Diagramm in der Baumansicht angeklickt wird.

Ein Editor wird geschlossen, wenn einer der folgenden Fälle eintritt:

- Der Benutzer schließt den Editor
- Das Diagramm, das im Editor angezeigt wird, wird gelöscht
- Das Modell, zu dem der Editor gehört, wird geschlossen

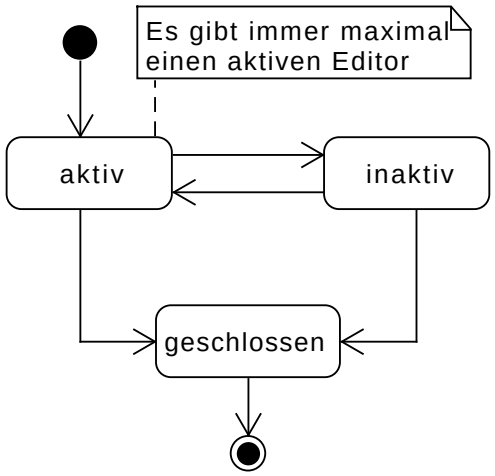

Abbildung 9.12: Lifecycle für Editoren

9.5.4 View Client

Da bei mehreren Modellen und Diagrammen die Übersicht über die erstellten Modellelemente verloren geht, wird eine `TreeView` erstellt, welche die Modellelemente einem Diagramm bzw. Modell zuordnet. In dieser `TreeView` können Modellelemente selektiert werden, sodass diese im entsprechenden Diagramm ausgewählt werden.

Die `TreeView` wird mit SWT (siehe Abschnitt 8.1) als `ModelTreeViewPart` realisiert und als ein zusätzlicher Bereich im Editor eingebettet. Die notwendigen Informationen zu dem Zustand des Modells werden von der `ModelRegistry` geholt, aufbereitet und anschließend als `TreeView` dargestellt. Bei Änderungen des Modells wird der `ModelChangeListener` aktiviert und die `TreeView` wird aktualisiert.

Die `UserListViewPart`, welche die Anwender anzeigt, die am gleichen Diagramm arbeiten, wird ebenfalls mit `SWT` als `ListView` realisiert. In dieser Liste werden Anwender mit ihren Farben dargestellt, sowie Steuerelemente, mit denen man seinen Name und seine Farbe ändern kann. Um die Farben des Diagramms auszublenden oder im Modell zu löschen existieren hier ebenso Steuerelemente.

Zudem werden alle Popup-Dialoge, welche im Programm enthalten sind, mit `SWT` realisiert.

9.5.5 Server

In diesem Abschnitt werden Presenter und Model des Servers genauer beschrieben, siehe dazu auch Abbildung 9.1. Die detaillierte Architektur ist

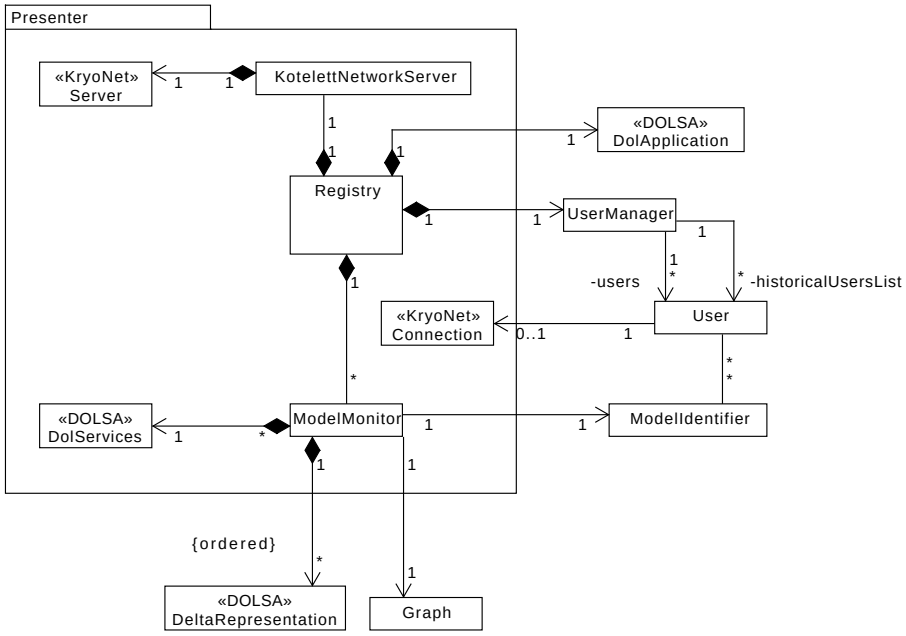

Abbildung 9.13: Der Presenter des Servers zusammen mit dem Modell

in Abbildung 9.13 abgebildet. Der Server besteht aus dem Presenter und Model. Zentral ist hierbei die `Registry`. Sie nimmt Anfragen der Clients vom `KotelettNetworkServer` entgegen und verwendet die weiteren Teile des Servers, um diese zu erfüllen. Dazu deligiert sie eingehende Deltas an den für das jeweilige Modell zuständige `ModelMonitor` und User bezogene Anfragen an den UserManager. Anfragen bezüglich des Repositories werden der DolApplication übergeben.

Der `UserManager` verwaltet alle User. Es unterscheiden sich dabei angemeldete User und historische User. Historische User werden aufgehoben, damit dem Client die zu dieser UserId gehörige Farbe hat. Einem angemeldeten User ist eine Connection zugeordnet, diese wird verwendet um dem Client des Users Nachrichten zu schicken, sowie die Menge der ModellIdentifier der Modelle an denen er arbeitet. Jedem ModellIdentifier ist umgekehrt auch die Menge der User, die auf seinem Modell arbeiten, zugeordnet.

Der `ModelMonitor` stellt sicher, dass eingehende Deltas in der korrekten Reihenfolge auf das Modell angewandt werden und dass die Deltas an alle am Modell mitarbeitenden User verteilt werden. Die Anwendung der Deltas geschiet durch Nutzung der `DolServices`

9.6 Parsen von Benutzereingaben

Bei der Modellierung von Klassendiagrammen auf Diagrammebene gibt es einige Anwendungsfälle in denen textuelle Benutzereingaben direkt in Modellinformationen umgewandelt (geparst) werden müssen:

- Attributsignaturen
- Methodensignaturen
- Rollenbezeichner
- Multiplizitäten

In all diesen Fällen wird durch die Spezifikation von Klassendiagrammen nach der UML Super Structure [Obj11] eine spezielle Syntax definiert in der Modellinformationen in Textrepräsentationen im Diagramm umgewandelt werden. Diese Syntax wird in der Anwendung als Eingabe durch den Anwender in definierten Eingabebereichen erwartet und direkt in die entsprechenden Modellinformationen umgewandelt. Hierzu wurden Parser implementiert die diese Aufgabe übernehmen und Fehler direkt an den Anwender zurückmelden. Dieser Abschnitt erklärt die verwendete und erwartete Syntax und die technische Umsetzung der Parser zum Erfüllen der Funktionalität.

9.6.1 Implementierung der Parser

Die Anwendereingaben werden mithilfe der Java-String-Standardfunktionen verarbeitet. Eine generischere Lösung mit definierten Grammatiken auf Programmebene wurde in Form von JavaCC evaluiert jedoch aufgrund der Funktionalität von JavaCC [Ora14] und der eher geringfügigen Verwendung von Parsing im Projekt verworfen. JavaCC wurde als zu mächtiges und umfangreiches Werkzeug für den umzusetzenden Task bewertet.

Um die nötigen Modellinformationen aus der Eingabe abzuleiten wird die Eingabe fortlaufend, dass heißt nach jeder erfolgreich ausgewerteten Modellinformation, mithilfe der `String.split()`-Funktion anhand eines Begrenzers zerteilt. Bspw. anhand des Separators `:` zwischen Bezeichner und Typ eines Attributs. Dieses einfache Vorgehen erlaubt genaue Rückmeldungen an den Anwender im Fehlerfall, macht jedoch die Erkennung mehrerer Fehler in der Eingabe unmöglich.

Die Parser werden in der jeweiligen EditPolicy zur Bearbeitung von Text (vgl. Unterabschnitt 8.2.2) des bearbeiteten Elements verwendet und aufgerufen. Die Eingabe des Nutzers wird überprüft und bei Erfolg in eine interne Datenrepräsentation der Informationen umgewandelt. Diese wird dann an das zu erzeugende Command zur Durchführung der Änderungen übergeben, so dass die Änderungen rückgängig gemacht und wiederholt werden können.

Schlägt die Überprüfung aufgrund eines Eingabefehlers fehl so wird keine Repräsentation und entsprechend auch kein Command erzeugt. Stattdessen wird eine entsprechende Fehlermeldung an den Anwender ausgegeben und die Bearbeitung ohne Änderung des Modells verworfen.

9.6.2 Sichtbarkeit

Wann immer eine Eingabe einen Sichtbarkeitsmodifikator enthalten kann, wird eines der gebräuchlichen Kurzschreibweisenzeichen erwartet:

- public: `+`
- protected: `~`
- private: `-`

Für default-Sichtbarkeit wird kein Zeichen erwartet.

9.6.3 Attributsignaturen

Die erwartete Syntax zur Eingabe von Attributen lautet:

```
[+|~|-] <Bezeichner> : <Typ>
```

Insbesondere wird hierbei im Parser geprüft ob der angegebene Typ bereits im Modell als primitiver Datentyp oder eigens angelegter Type existiert. Ist dies nicht der Fall wird eine entsprechende Fehlermeldung ausgegeben.

9.6.4 Methodensignaturen

Die erwartete Syntax zur Eingabe von Methoden lautet:

```
[+|~|-] <Bezeichner>({<Parameterbezeichner>:<ParameterTyp>}*)
  : <Rückgabetyp>
```

Auch hier werden im Parser sowohl Parametertypen als auch der Rückgabetyp auf Existenz im aktuellen Modell geprüft und gegebenenfalls ein Fehler zurückgemeldet.

9.6.5 Rollenbezeichner

Die erwartete Syntax für Rollenbezeichner umfasst lediglich den Sichtbarkeitsmodifikator und den Bezeichner:

```
[+|~|-] <Bezeichner>
```

9.6.6 Multiplizitäten

Die erwartete Syntax für Multiplizitäten lautet:

```
{<NatürlicheZahl>[..{<NatürlicheZahl>|*}]|*}
```

Natürliche Zahlen beinhalten in diesem Fall auch die 0.

Hierbei muss insbesondere zwischen unären Eingaben und solchen mit Ober- und Untergrenzen unterschieden werden da sowohl Ober- als auch Untergrenze ein eigenes Attribut des KAssociationEndpoint im Metamodell sind. Ebenso stellt das Literal „Unlimited Natural“ (*) einen Sonderfall dar und muss speziell in die entsprechende Modellrepräsentation umgewandelt werden.

Kapitel 10

Datenmodell

In diesem Kapitel wird das Datenmodell (Metamodell) beschrieben, das vom Modellierungstool *Kotelett* zur Repräsentation von Klassendiagrammen auf Client und Server benutzt werden soll. Es dient sowohl als Metamodell für Klassendiagramme als auch als Datenstruktur für relevante Layoutinformationen.

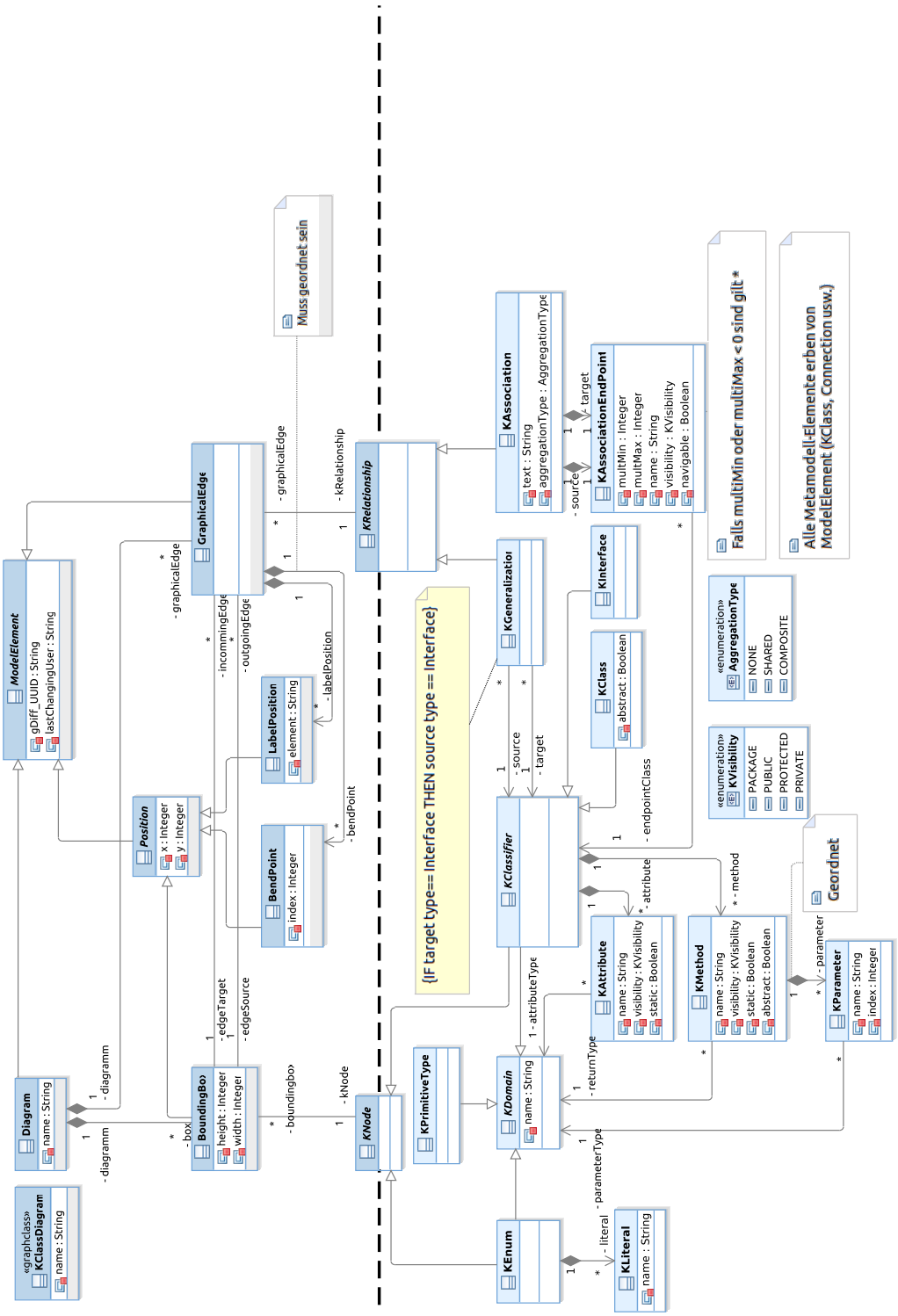

Abbildung 10.1: Metamodell für Klassendiagramme einschließlich Layoutinformationen

10.1 Grobe Struktur

Das Datenmodell beschreibt sowohl syntaktische Informationen über Klassendiagramme (das eigentliche Metamodell) als auch Layoutinformationen. Das Layout eines Klassendiagramms wird durch alle Klassen beschrieben, die oberhalb der gestrichelten Trennlinie in Abbildung 10.1 dargestellt sind, ausgenommen `ModelElement`. Die Klassen unterhalb der Trennlinie beschreiben die logische Struktur von Klassendiagrammen. Im Folgenden wird der Teil oberhalb der Trennlinie als *Layoutklassen* und der Teil unterhalb als *Metamodellklassen* bezeichnet.

`ModelElement` ist eine abstrakte Oberklasse, von der alle Klassen des Modells erben, sowohl die Klassen für Layoutinformationen als auch jene, die die logische Struktur beschreiben. Sie stellt das Attribut `gDiff_UUID` bereit, das von DOLSA zur eindeutigen Identifizierung der Elemente benötigt wird.

Der Übersicht halber sind in der Abbildung 10.1 nicht alle Generalisierungen eingezeichnet.

Das Datenmodell behandelt in der vorliegenden Version keine Kommentare und Assoziationsklassen.

10.2 Metamodellklassen

Zunächst sollen die Metamodellklassen, also jener Teil, der die logische Struktur von Klassendiagrammen beschreibt, erläutert werden.

Wesentliche Oberklassen sind `KRelationship` und `KDomain`. `KRelationship` beschreibt die Beziehungen [Generalisierungen (`KGeneralization`) und Assoziationen (`KAssociation`)] zwischen Klassen, `KDomain` ist die Oberklasse für primitive Typen (`KPrimitiveType`), Klassen und Enums (`KClass` und `KInterface` mit der gemeinsamen Oberklasse `KClassifier`) und Enums (`KEnum`). Alle Metamodellklassen haben ein K für Kotelett vorangestellt, um Überschneidungen mit Schlüsselwörter (z.B. Class) zu vermeiden.

10.2.1 Beziehungen

Es wurde entschieden, Generalisierungen wie Assoziationen durch eine Unterklasse von `KRelationship` darzustellen. Dies bietet sich in Hinblick auf die Visualisierung an.[1] `KClass` hat somit keine eigenen Attribute, die Ober- und Unterklassen beinhalten, da diese redundant zu Quelle und Ziel der Generalisierung wären (vgl. Abbildung 10.3).

Die Oberklasse `KRelationship` hat zwar keine Attribute (`source` und `target` sind jeweils nur in den Unterklassen), existiert aber konzeptionell.

[1] Die Darstellung mit GEF ist am einfachsten, wenn für jedes Element in der View (Knoten, Kanten) ein entsprechendes Modellelement existiert

`KRelationship` stellt zusammen mit `KNode`, der Oberklasse für alle Modellelemente, die als Knoten in einem Diagramm dargestellt werden können, sozusagen die Schnittstelle zwischen Modellelementen und ihrer Graphischen Repräsentation dar. Daher sind diese Klassen in Abbildung 10.1 auf der Trennungslinie positioniert.

Generalisierungen und Implementierungen

Sowohl Generalisierungen als auch Implementierungen werden durch die Klasse `KGeneralization` beschrieben. `source` und `target` von `KGeneralization` sind somit vom Typ `KClassifier`. Ob eine Instanz von `KGeneralization` eine Generalisierung oder Implementierung beschreibt, ergibt sich durch den Typ von `source` und `target`: Wenn `source` und `target` vom gleichen Typ, `KClass` oder `KInterface`, sind, wird eine Generalisierung modelliert. Wenn `source` vom Typ `KInterface` und `target` vom Typ `KClass` ist, ist es dementsprechend eine Implementierung. Eine Verbindung von `KClass` zu `KInterface` ist nicht erlaubt.

Assoziationen

Ausgehend von `KAssociation` sind `source` und `target` vom Typ `KAssociationEndPoint`, was das Ende einer Assoziation darstellt. `KAssociationEndPoint` enthält Informationen über die Rollen, Multiplizitäten und Navigierbarkeit einer Assoziation. Alternativ könnte man dies mittels einer Assoziationsklasse modellieren (siehe Abbildung 10.3), dies erschwert aber wahrscheinlich die Umsetzung in DOL.

Kompositionen und Aggregationen werden durch ein Attribut `aggregationKind` modelliert, das angibt, ob eine `KAssociation` Aggregation oder Komposition ist. So können Assoziationen, Aggregationen und kompositionen einfach ineinander umgewandelt werden.

Leserichtungen für Assoziationen sind nicht modelliert. Generell sind alle Assoziationen von `source` nach `target` gerichtet.

10.2.2 Typen

Die Oberklasse für alle Typen (primitive Typen, Klassen, Enums) ist `KDomain`.

Klassen und Interfaces

Klassen (`KClass`) und Interfaces (`KInterface`) haben die gemeinsame Oberklasse `KClassifier`. Klassen können im Unterschied zu Interfaces abstrakt sein.

Attribute und Methoden sind nicht explizit geordnet. Sie werden grundsätzlich in alphabetischer Reihenfolge angezeigt. Der Typ von Attributen,

Methoden-Rückgabewerten und Parametern kann primitiver Typ, Klasse oder Enum sein. Statische Methoden und Attribute sowie abstrakte Methoden werden unterstützt.

Primitive Typen

Primitive Typen sind Instanzen von `KPrimitiveType`. In jedem Diagramm existieren dann zur Laufzeit Instanzen für Integer, Float, Boolean, String etc.

Enums

Enums (`KEnum`) besitzen eine Menge von Literalen (`KLiteral`) Sie sind keine Klassen, können also nicht Quelle oder Ziel von Assoziationen oder Generalisierungen sein. Die Literale sind nicht geordnet.

10.3 Layoutklassen

Im Folgenden werden nun die Layoutklassen, also jene Klassen des Datenmodells, die Layoutinformationen repräsentieren, beschrieben. Die Instanzen der Layoutklassen verweisen jeweils auf die Modellelemente. Durch JGraLab können die Layoutinformationen zu einem Modellobjekt leicht gefunden werden. Das Datenmodell erlaubt für jedes Modellelement mehrere Instanzen mit Layoutinformationen, was ein erster Schritt in Richtung multiperspektivischer Modellierung ist (mehrere Diagramme für ein Modell).

10.3.1 Diagramme

Diagramme werden durch `Diagram` dargestellt. Jede `BoundingBox` und jede `GraphicalEdge` ist genau einem Diagramm zugeordnet. Da ein Modellelement aber mehreren Layoutelementen (z.B. ein `KNode` mehreren `BoundingBox`es) zugeordnet sein kann, werden mehrere Diagramme für ein Modell ermöglicht.

10.3.2 Knoten

Für jedes Element eines Klassendiagramms, für das zusätzlich Layoutinformationen verwaltet werden müssen, existiert zur Laufzeit noch mindestens eine Instanz aus den Layoutklassen, die diese Informationen speichert. Für Klassen und Enums sind Position und Größe ausreichend, die als `BoundingBox` abgebildet werden. Da diese Information für Klassen und Enums gleich ist, haben diese eine gemeinsame Oberklasse `KNode`, auf die `BoundingBox` verweist.

10.3.3 Kanten

Obwohl Generalisierungen und Assoziationen unterschiedliche Eigenschaften haben und für Assoziationen mehr Layoutinformationen verwaltet werden müssen, gibt es für Kanten in Diagrammen nur die Klasse `GraphicalEdge`. `GraphicalEdge` enthält die *Bendpoints* und die Positionen eventueller Beschriftungen der Kante.

Bendpoints sind die Punkte, durch die eine Kante im Diagramm gelegt wird. Jede kannte hat beliebig viele Bendpoints, die unbedingt geordnet sein müssen. Der Nutzer kann so eine Kante manuell routen.

Die Positionen der Beschriftungen (`LabelPosition`) haben ein Attribut `element`, welches zur Laufzeit einen Bezeichner enthält, der angibt, welche Eigenschaft des Modellelements in der Beschriftung an jener Position dargestellt wird. Bei Klassendiagrammen kann `element` z.B. angeben, ob die Position zum Namen einer Assoziation oder einem Rollenbezeichner gehört. Dadurch ergibt sich auch implizit, wie die `x`- und `y`-Koordinaten der Position interpretiert werden, z.B. ob sie relativ zu Start, Mitte oder Endpunkt der Kante zu interpretieren sind. Dadurch, dass hier String-Bezeichner zur Zuordnung der Positionen zu Eigenschaften der Modellelemente gewählt wurden, bleiben die Layoutklassen auch hier vom restlichen Metamodell unabhängig. Neben Assoziationen und Generalisierungen könnte `GraphicalEdge` z.B. genauso Layoutinformationen für Transitionen zwischen Aktionen in einem Aktivitätsdiagramm beschreiben.

Da `GraphicalEdge` eine Referenz auf das Modellelement hält, ist eine Referenz auf die `BoundingBox` von Quelle und Ziel nicht zwingend nötig. Um das Zeichnen des Diagramms zu erleichtern, wurden dennoch entsprechende Assoziationen zwischen `BoundingBox` und `GraphicalEdge` eingeführt. Hier muss selbstverständlich verstärkt auf die Konsistenz des Modells geachtet werden, dass `edgeSource` und `edgeTarget` einer `GraphicalEdge` mit den Modellelementen übereinstimmen.

10.4 Constraints

Folgende Bedingungen müssen eingehalten werden, um die Konsistenz, Validität (in Bezug auf korrektes UML) und eindeutige Darstellung eines Modells zu gewährleisten.

10.4.1 Konsistenz

[CONSISTENCY 1] Die Quelle (`edgeSource`) einer `GraphicalEdge` muss im selben Diagramm wie die `GraphicalEdge` enthalten sein.

[CONSISTENCY 2] Das Ziel (`edgeTarget`) einer `GraphicalEdge` muss im selben Diagramm wie die `GraphicalEdge` enthalten sein.

[CONSISTENCY 3] Die Quelle der `KRelationship`, auf die eine `GraphicalEdge` zeigt, muss das Modellelement (Instanz von `KNode`) sein, auf die die `BoundingBox` zeigt, die Quelle der `GraphicalEdge` ist.

[CONSISTENCY 4] Das Ziel der `KRelationship`, auf die eine `GraphicalEdge` zeigt, muss das Modellelement (Instanz von `KNode`) sein, auf die die `BoundingBox` zeigt, die Ziel der `GraphicalEdge` ist.

10.4.2 Valides UML

[UML 1] Wenn `target` einer `KGeneralization` eine Instanz von `KInterface` ist, muss `source` der `KGeneralization` ebenfalls Instanz von `KGeneralization` sein.

10.4.3 Eindeutigkeit der Darstellung

[LAYOUT 1] Die Indices (`index`) der `BendPoints` einer `GraphicalEdge` müssen verschieden sein. Sie sollten fortlaufend sein.

[LAYOUT 2] Die Indices (`index`) der `KParameter` einer `KMethod` müssen verschieden sein. Sie sollten fortlaufend sein.

[LAYOUT 3] Das Attribut `element` einer `LabelPosition` darf nur zuvor für den Kantentyp definierte Werte annehmen.

[LAYOUT 4] Das Attribut `element` aller `LabelPositions` einer `GraphicalEdge` muss verschieden sein.

10.5 Interpretation von Layoutangaben

Die in diesem Abschnitt geschilderte Darstellung ist noch nicht vollständig auf ihre Umsetzbarkeit in GEF getestet, ist aber voraussichtlich mit akzeptablem Aufwand umsetzbar.

Im folgenden wird erläutert, wie die Attribute der Layoutklassen voraussichtlich bei der Darstellung interpretiert werden. Die Interpretation zielt auf eine einfache Umsetzung in GEF.

Koordinatenursprung im Editor ist oben Links. Die Achsen verlaufen von links nach rechts bzw. oben nach unten. Somit bezeichnen die `x`- und `y`-Koordinaten einer `BoundingBox` die obere, linke Ecke des Rechtecks.

Kanten werden als Polygonzüge durch alle Eckpunkte (`BendPoints`) der Kante interpretiert. Koordinaten von Eckpunkten können sowohl absolut als auch relativ interpretiert werden. In einem früheren Prototyp hat es sich bewährt, Eckpunkte von Schlaufen relativ zum Mittelpunkt des zugehörigen

Knotens zu interpretieren, und Eckpunkte von Kanten zwischen zwei verschiedenen Knoten absolut. So verschieben sich Schlaufen automatisch mit dem verbundenen Knoten.

Zum Anknüpfungspunkt von Kanten an die Knoten gestaltet es sich am Einfachsten, den Berührungspunkt so zu wählen, dass die Kante auf den Mittelpunkt der Box zeigt, bzw. von diesem ausgeht, wie in Abbildung 10.2 oben dargestellt. Um dem Anwender aber ein Rechteckiges Layout zu ermöglichen, herrscht die Idee, bei Kanten mit mindestens einem Eckpunkt den Berührungspunkt möglichst so zu wählen, dass die Kante rechtwinklig auf der Kante der Box steht (siehe Abbildung 10.2 unten).

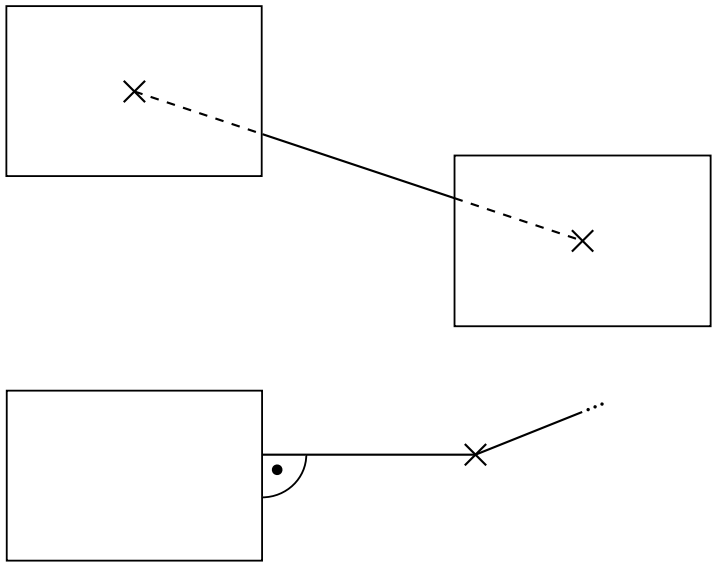

Abbildung 10.2: Auf den Mittelpunkt gerichtete Kante (oben) und senkrecht stehende Kante (unten)

10.6 Sonstiges

- *Visibility:* Das Datenmodell unterstützt für Attribute, Methoden und Rollen die Sichtbarkeiten `Package` (keine Angabe/Default, angelehnt an Java), `Public`, `Protected` und `Private`.
- *Interfaces und Constraints in grUML:* JGraLab unterstützt keine Interfaces und keine Constraints, die keine validen GreQL-Anfragen enthalten. Daher wurden im Datenmodell an Stelle von Interfaces abstrakte Klassen und anstelle von Constraints Kommentare benutzt.

- `gDiff_UUID`: Das Attribut `gDiff_UUID` der Oberklasse `ModelElement` wird von DOLSA benötigt, um die Identifier der Modelelement zu verwalten.

- `index`: Da DOL nicht von geordneten Kanten ausgeht und Deltas somit keine Information über die Reihenfolge von Modellelementen beinhalten, enthalten alle Modellelemente, deren Reihenfolge innerhalb des Modells von Bedeutung ist, ein Attribut `index`, das diese Reihenfolge bestimmt.

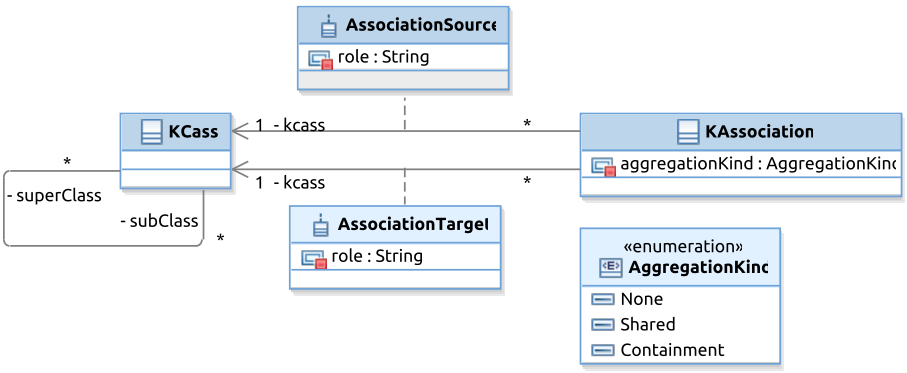

Abbildung 10.3: Alternative Modellierungsmöglichkeiten

Teil IV

Wartungshandbuch

Dieses Kapitel stellt ein Wartungshandbuch für das Projekt Kotelett dar. Ziel dieses Kapitels ist es, einen Überblick über die Package-Struktur des Projekts zu verschaffen, um die Weiterentwicklung und Wartung von Kotelett zu erleichtern.

Kapitel 11

Bauen von Kotelett

In diesem Abschnitt wird beschrieben, wie der Kotelett Client und Server gebaut wird.

Der grobe Ablauf des Builds ist in Abbildung 11.1 dargestellt. Es wird jedes mal gebaut, wenn Änderungen in das serverseitige Repository übertragen werden. Dies wird durch einen post-update hook des Git-Repositorys ermöglicht. Als Build-Server wurde während des Entwicklungsprozesses Jenkins in Verbindung mit Maven verwendet. Gebaut wird in einem lokalen Abbild des Projekt-Git. Aus den Quelldateien zusammen mit den in den Manifest-Dateien enthaltenen Abhängigkeiten werden die Eclipse-Plugin-Projekte compiliert. Detaillierte Konfigurationen sind in den `pom.xml`-Dateien in den Projekten enthalten. Benötigte dritte Eclipse-Plugins werden direkt aus der Update-Site für Eclipse Kepler geladen.

Kotelett kann aus dem Projekt-Ordner `de.uniol.pgktltt.gef` mittels `mvn clean build` gebaut werden. Es werden der Client und der Server gebaut, zusätzlich werden die Tests ausgeführt.

Die Client-Produkte befinden sich dann in `de.uniol.pgktltt.gef.client.rcp/target/products/`, es gibt ein ZIP-Archiv pro Betriebssystem bzw. Architektur.

Der gebaute Server befindet sich in `de.uniol.pgktltt.gef.client.rcp/target/de.uniol.pgktltt.server-1.0.0-SNAPSHOT.jar`, die benötigten Bibliotheken sind in `de.uniol.pgktltt.gef.client.rcp/target/dependency/`, zusätzlich sind sie in von `de.uniol.pgktltt.shared/target/de.uniol.pgktltt.shared-1.0.0-SNAPSHOT.jar` enthalten.

In den folgenden Abschnitten wird der Buildprozess detaillierter beschrieben.

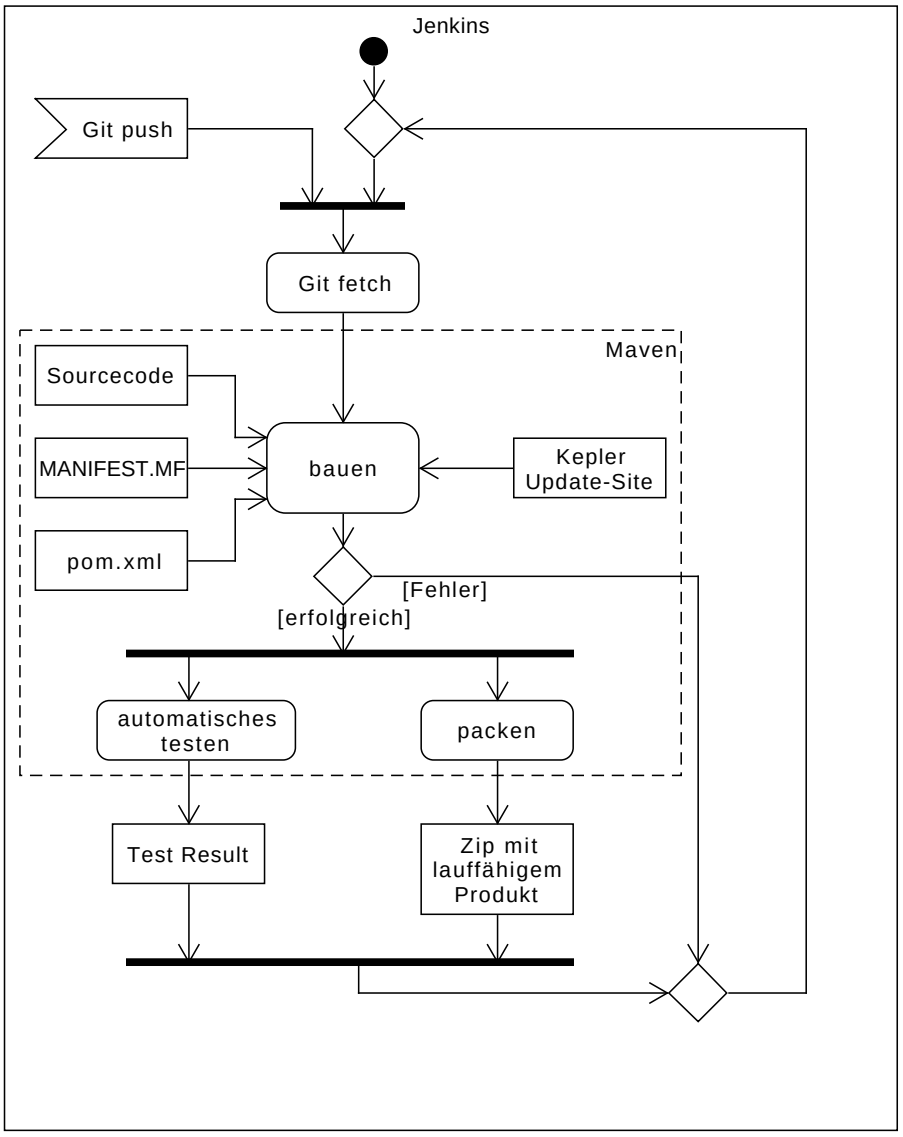

Abbildung 11.1: Grobe Darstellung des Build-Prozess

11.1 Build des Client

Die gebauten Eclipse-Plugins `de.uniol.pgktltt.gef.client` und `de.uniol.pgktltt.shared` werden zu je einem lauffähigen Produkt für Windows (32 und 64 Bit), Linux (32 und 64 Bit) und Mac Os X (64 Bit) zusammengestellt und zu einem ZIP-Archiv verpackt.

Die in `de.uniol.pgktltt.gef.test` enthaltenen Unit- und Integrationstests werden automatisch in einer Eclipse-Laufzeitumgebung ausgeführt und die Testergebnisse auf dem Server gespeichert.

Für die Maven-Konfiguration und die Aufteilung in ein extra Plugin für das RCP und die Tests dienten die Beispiele in `http://git.eclipse.org/c/tycho/org.eclipse.tycho-demo.git/tree/` als Vorbild.

11.1.1 Spezielle Anpassungen für das ARBI System

Damit automatisierte Tests auf dem virtuellen Server der Projektgruppe möglich sind, müssen ein paar spezielle Konfigurationen vorgenommen werden. Grund ist, dass das ELF-Format von FreeBSD sich von dem in Linux benutzen unterscheidet, so dass für Linux compilierte Binaries nur mit der Linux-Emulation von FreeBSD ausführbar sind. Diese unterstützt zudem nur 32-Bit-Binaries. Da Eclipse-Anwendungen viel nativen Code enthalten und diese Bibliotheken nicht für FreeBSD existieren, muss auf die Linux-Emulation zurückgegriffen werden. Dies bedeutet, dass die Tests zum einen auf einer 32-Bit-Linux-JVM ausgeführt werden müssen und zum anderen die entsprechenden nativen Bibliotheken von der Eclipse-Laufzeitumgebung geladen werden müssen.

Damit eine andere als die standard-JVM genutzt werden kann, muss diese als Maven-Toolchain[1] registriert werden.

Damit die richtigen Bibliotheken geladen werden, wird Maven so konfiguriert, dass beim Testen die JVM mit den zusätzlichen Argumenten

```
-Dosgi.arch=x86 -Dosgi.os=linux -Dosgi.ws=gtk
```

ausgeführt wird.

Um den Build sowohl auf dem ARBI-System als auch auf einem anderen System ausführen zu können, wurde für das ARBI-System ein Build-Profil in Maven mit diesen Anpassungen erstellt, das angewandt wird, wenn Maven mit dem Parameter `-P joerg-os` gestartet wird. Die Toolchain-Definition wird in `$HOME/.m2/toolchains.xml` gesucht. (`$HOME` ist das Home-Verzeichnis des Unix-Nutzers, der den Build ausführt.)

[1] siehe `http://wiki.eclipse.org/Tycho/Reference_Card#Selecting_JDK`

11.2 Build des Servers

Im Gegensatz zum Client ist der Server ein „konventionelles“ Java-Projekt. Output des Builds ist damit eine JAR-Datei. Beim Auflösen der Abhängigkeiten ergeben sich allerdings einige Probleme: Zunächst erwartet Maven, dass alle Projekt-Abhängigkeiten aus einem Repository aufgelöst werden können, d.h. dass alle Drittbibliotheken im Lokalen Maven-Repository enthalten sind. Dies ist in diesem Build-Prozess nicht der Fall. Das weitere, schwerwiegendere Problem ist, dass der gemeinsame Code für Client und Server in einer Form vorliegen muss, dass er sowohl als Eclipse-Plugin als auch als konventionelle Java-Bibliothek genutzt werden kann. Die gemeinsamen Klassen werden zum Eclipse-Plugin `de.uniol.pgktltt.gef.shared` gepackt, in dem zugleich alle Drittbibliotheken (DOLSA, JGraLab, Kryonet und Guava) enthalten sind. Diese Bibliotheken nun noch einmal getrennt in das Maven-Repository auf dem Build-Server einzupflegen würde ein zu hohes Risiko von Versions-Inkompabilitäten birgen. Stattdessen wird das Dependency-Management von Maven umgangen und das shared-Plugin in einem Unterverzeichnis im Build-Verzeichnis des Server-Projekts entpackt. Dann liegen alle Abhängigkeiten als einzelne JARs vor. Das Unterverzeicnis, in dem die JARs liegen, wird dem Java-Compiler mit dem Kommandozeilenparameter `-ext` übergeben. [2]

11.3 Build des Metamodells

Das Bauen des Metamodells wurde aus mehreren Gründen nicht in den Buildprozess integriert. Der Code-Generator hat zwar eine Schnittstelle für Ant, diese müsste aber in den Maven-Prozess integriert werden. Die generierte API müsste in das Shared-Projekt kopiert werden und bei größeren Änderungen (z.B. Umbenennungen von Packages) auch das Shared-Projekt selbst angepasst werden. Da sich während der Entwicklung das Metamodell nur selten geändert hat, wäre kein Nutzen aus einer aufwändigen Integration entstanden.

[2] Dies ist tatsächlich etwas „gehackt“. Der `-ext`-Parameter soll eigentlich auf weitere Bibliotheken der Java-Laufzeitumgebung zeigen. Hier wird er als Claspath missbraucht. Andere Maven-Mechanismen, wie z.B. System-Abhängigkeiten, die den Classpath nutzen, können nicht benutzt werden, da diese die Existenz der Bibliotheken prüfen, bevor das shared-Projekt entpackt werden kann.

Kapitel 12

Entwicklungsumgebung

Kotelett wurde mit Eclipse Kepler (Eclipse 4.3.2) entwickelt, es sollten aber auch neuere Versionen zur Weiterentwicklung verwendet werden können. Es wird die *„Eclipse IDE for Java EE Developers*[1] empfohlen. Dazu muss das *Graphical Editing Framework GEF SDK*[2] von der Eclipse-Update-Site für Kepler[3] installiert werden.

Das Metamodell kann im Rational Software Architect[IBM14] bearbeitet werden. Anweisungen, wie ein geändertes Metamodell integriert wird, finden sich in Abschnitt 16.2.

[1] `http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/keplersr2`

[2] Es wurde Version 3.9.1.201308190730 verwendet

[3] `http://download.eclipse.org/releases/kepler`

Kapitel 13

Projektstruktur

Der Sourcecode ist auf die Projekte `de.uniol.pgktltt.gef.client`, `de.uniol.pgktltt.shared` und `de.uniol.pgktltt.server` aufgeteilt, wobei die beiden ersteren Eclipse-Plugin-Projekte und letzteres ein Java-Projekt ist.

- `de.uniol.pgktltt.gef.client` enthält den Client-Code.
- `de.uniol.pgktltt.server` enthält den Server-Code
- `de.uniol.pgktltt.shared` enthält Code und Bibliotheken, die von Client und Server benötigt werden. Insbesondere Klassen zum Nachrichtenaustausch, das Metamodell (als JAR) und die Externen Bibliotheken für JGraLab, DOLSA, Kryonet und Guava.

Weitere Projekte sind

- `de.uniol.pgktltt.gef.client.rcp`: Dieses Projekt enthält die Produktkonfiguration, die benötigt wird, um Kotelett als Eclipse-Product zu bauen. Die Datei `de.uniol.pgktltt.gef.client.rcp` sollte möglichst nur manuell bearbeitet werden, um zu verhindern, dass Eclipse abhängigkeiten entfernt, die einen Build für alle unterstützen Betriebssysteme ermöglichen.
- `de.uniol.pgktltt.gef`: Dieses Projekt enthält nur eine Konfigurationsdatei für Maven, `pom.xml`, die den Hauptteil der Buildkonfiguration enthält (siehe oben). Das Projekt dient somit als Elternprojekt für die anderen fünf Projekte und als Startpunkt für den Build-Prozess.
- `de.uniol.pgktltt.test`: Dieses Projekt enthält Unit-Tests für DOLSA und kritische Komponenten von Kotelett.

13.1 Shared: detailliertere Struktur

In diesem Abschnitt wird die Struktur des Shared-Projekts genauer beschrieben. Dieses Projekt wird von Client und Server referenziert und enthält somit gemeinsam benutzte Klassen und Drittbibliotheken.

de.uni.pgktltt.shared

In diesem Package befinden sich Datenklassen für den Datenaustausch zwischen Server und Client sowie gemeinsam benutzte Utilities.

Eine wichtige Klasse ist hier der `SerializationInitializer`: Hier werden alle Klassen registriert, von denen Instanzen mittels Kryonet verschickt werden sollen. Alle neuen Klassen, die für den Nachrichtenaustausch benutzt werden, müssen registriert werden. Diese müssen zudem einen Default-Konstruktor haben, der public ist.

Eine weitere wichitige Utility-Klasse ist `GraphUtils`: Diese Klasse definiert die Methoden `addMissingUUIDs` zum Hinzufügen fehlender UUIDs in einem Graphen, `areIsomorph`, die überprüft, ob zwei Graphen bis auf Reihenfolge und JGraLab-IDs der Knoten und Kanten isomorph sind und `clone`, die einen Graphen kopiert.

de.uni.pgktltt.shared.message

Hier und in den Subpackages befinden sich die Nachrichten zur Client-Server-Kommunikation (siehe Abschnitt 9.4 und Abschnitt 9.5). Das Package `de.uni.pgktltt.shared.message` enthält die Oberklassen und Interfaces für Nachrichten. Alle Nachrichten implementieren `KotelettMessage`. Die Nachrichten, die von Client zu Server geschickt werden, implementieren zusätzlich `ClientToServerRequest` und befinden sich im Subpackage `de.uni.pgktltt.shared.message.clientToServer`. Die Nachrichten von Server zu Client implementieren `ServerToClientMessage` und befinden sich im Package `de.uni.pgktltt.shared.message.serverToClient`.

13.2 Client: detailliertere Struktur

In diesem Abschnitt wird das Client-Projekt genauer beschrieben. Aus Platzgründen beschränkt sich diese Dokumentation auf die Klassen, die für die Weiterentwicklung und Pflege am wichtigsten sind.

de.uniol.pgktltt.gef.client

Hier sind die Entrypoint-Klassen für den Client enthalten. Die Klassen `ColorHelper` und `FinalBendPoint` sind Hilfsklassen für den Klassendiagramm-Editor.

Die Klasse `OpenEditorOnStartup` initialisiert die Verbindung zum Server und die Modellverwaltung. Die Verwendung dieser Hilfsklasse ist notwendig, da sichergestellt sein muss, dass die GUI vor der Modellverwaltung und der Client-Server-Verbindung vollständig initialisiert wird.

de.uniol.pgktltt.gef.client.command

Dieses Package enthält die GEF-Commands für den Klassendiagramm-Editor. Zur Verwendung von Commands siehe Abschnitt 8.2 und [RWC11].

Alle Commands sollten von `AbstractModelCommand` erben (siehe Abschnitt 16.2).

Eine wichtige Oberklasse für Commands ist noch `CreateNodeCommand`, der die Erstellung eines KNodes (siehe Kapitel 10) im Editor einschließlich der BoundingBox kapselt. Es muss lediglich `createNode()` ausimplementiert werden. In der Methode muss dann der KNode im Modell erstellt und zurückgegeben werden.

Desweiteren ist die `GenericDeleteCommandFactory` interessant. Mit der Methode `createDeleteCommand` wird ein Command erstellt, der das übergebene Modell- oder Layoutobjekt löscht. Verbundene Elemente wie Kanten werden mit gelöscht, sodass das Modell korrekt bezüglich des Metamodells bleibt. Bei Modellobjekten werden die Layoutobjekte mit gelöscht, umgekehrt allerdings nicht. Für ein Anwendungsbeispiel siehe Abschnitt 16.2.

de.uniol.pgktltt.gef.client.control

Dieses Package enthält die ModelRegistry, wie sie in Abschnitt 9.5 beschrieben ist. Nachfolgend werden einige wichtige Methoden der Klasse `ModelRegistry` beschrieben. Die Methoden, deren Namen mit `request` beginnen, senden Nachrichten (Requests) an den Server. Die Methoden, die mit `incomming` und `handle` anfangen, behandeln eingehende Nachrichten vom Server.

- `closeDiagramEditor`: Diese Methode schließt den zu dem übergebenen Diagramm gehörigen Editor.
- `closeModel`: Diese Methode schließt das übergebene Modell: Alle zugehörigen Editoren werden geschlossen und eine Nachricht an den Server gesendet, dass der Anwender das Modell geschlossen hat.
- `createEditor`: Diese Methode öffnet einen Editor für ein Diagramm, bzw. Modell oder Version. Wenn bereits ein Editor geöffnet ist, bekommt dieser den Fokus.
- `forceOpenEditorRefresh`: Aktualisiert alle geöffneten Editoren.

- `getActiveEditor`: Gibt den Editor zurück, der aktuell den Fokus hat, bzw. als letzter Editor den Fokus hatte.
- `getAllModels`: Gibt eine Liste aller dem Client bekannten Modelle zurück, sowohl geöffnete als auch nicht geöffnete Editoren.
- `getModelList`: Gibt die Graphen zu allen geöffneten Modellen zurück.
- `getOpenModels`: Wie `getModelList`, nur dass Modell-Identifier anstelle der Graphen zurückgegeben werden.
- `getCommandStack`: Auf dem Client wird ein gemeinsamer CommandStack[1] für alle Editoren verwendet. Dieser wird von `getCommandStack` zurückgegeben.
- `setColorsVisible`: Aktiviert und deaktiviert die farblichen Hervorhebungen im Editor.

de.uniol.pgktltt.gef.client.editor

In diesem Package ist die Editor-Klasse und Hilfsklassen enthalten. `ClassDiagramEditor` erbt von `org.eclipse.gef.ui.parts.GraphicalEditorWithPalette`[2] und stellt damit die Editorklasse für Klassendiagramme dar. Der korrespondierende `EditorInput`[3] ist `ClassDiagramEditorInput`. Der `ClassDiagramEditorInput` hält eine Referenz auf das `Diagram`-Objekt (siehe Kapitel 10) sowie den zugehörigen Modell- und Versions-Identifier. (Letzterer ist bei Head-Versionen `null`). Das Zusammenspiel von `ClassDiagramEditor` und `ModelRegistry` ist in Unterabschnitt 9.5.3 beschrieben.

Für die Erstellung von Diagrammelementen ist zudem die `ClassDiagramFactory` wichtig. Diese implementiert `org.eclipse.gef.requests.CreationFactory`[4]. Instanzen dieser Klasse werden von GEF in den Paletteneinträgen (siehe Abschnitt 16.2) und den `CreationRequests`[5] verwendet. Die Methode `getNewObject()` gibt allerdings immer `null` zurück, da JGraLab-Vertices nicht ohne einen Graphen oder eine Factory instanziiert werden können.

[1] `http://help.eclipse.org/luna/topic/org.eclipse.gef.doc.isv/reference/api/org/eclipse/gef/commands/CommandStack.html`

[2] siehe `http://help.eclipse.org/luna/topic/org.eclipse.gef.doc.isv/reference/api/org/eclipse/gef/ui/parts/GraphicalEditorWithPalette.html` (12.03.2015)

[3] siehe `http://help.eclipse.org/luna/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/ui/IEditorInput.html` (12.03.2015)

[4] siehe `http://help.eclipse.org/luna/topic/org.eclipse.gef.doc.isv/reference/api/org/eclipse/gef/requests/CreationFactory.html` (12.03.2015)

[5] siehe `http://help.eclipse.org/luna/topic/org.eclipse.gef.doc.isv/reference/api/org/eclipse/gef/requests/CreateRequest.html` (12.03.2015)

Das Enum `ModelElementType` wird dazu verwendet, um Typinformationen zu Modellelementen in `CreationRequest`s zu übergeben. Jedes Literal besitzt eine Korrespondierende JGraLab-Vertex-Klasse. Da es aber zu einer Vertex-Klasse mehrere Literale geben kann, kann hier auch z.B. zwischen Assoziationen und Kompositionen unterschieden werden, die ja dieselbe Vertex-Klasse `KAssociation` haben (siehe Kapitel 10).

Die Klassen mit dem Suffix `Action` sind Aktionen[6], die im Kontextmenü des Editors verwendet werden, das bei einem Rechtsklick angezeigt wird.

de.uniol.pgktltt.gef.client.editpolicies

Dieses Package enthält Edit-Policies (siehe Abschnitt 8.2, das Tutorial in Abschnitt 16.2 und [RWC11]).

de.uniol.pgktltt.gef.client.editpolicies.inputparser

Dieses Package enthält den Parser und Hilfsklassen für Multiplizitäten, Attribute, Methoden und Rollenbezeichner. Genaueres ist in Abschnitt 9.6 beschrieben.

de.uniol.pgktltt.gef.client.parts

In dem Package `de.uniol.pgktltt.gef.client.parts` und den Sub-Packages sind die Edit-Parts für den Klassendiagrammeditor enthalten (siehe auch Abschnitt 8.2 und Abschnitt 16.2). Dies sind `GraphicalEditParts`[7] Alle Edit-Parts sollten von `KAbstractEditPart` erben, da dieser bereits wichtige Funktionalität für alle Edit-Parts in Kotelett implementiert.

de.uniol.pgktltt.gef.client.views

Hier sind weitere Teile der Kotelett-GUI implementiert. Unter anderem die Baumansicht für die Modellverwaltung im Package `de.uniol.pgktltt.gef.client.views.modeltreeview`. Die `SelectionListener` für die Menüeinträge befinden sich in `de.uniol.pgktltt.gef.client.views.modeltreeview.listener.selection`.

13.3 Server: detaillierte Struktur

Der Server hat nur zwei Packages, `de.uniol.pgktltt.server` und `de.uniol.pgktltt.server.voting`.

[6] `http://help.eclipse.org/luna/topic/org.eclipse.gef.doc.isv/reference/api/org/eclipse/gef/ui/actions/SelectionAction.html` (12.03.2015)

[7] `http://help.eclipse.org/luna/topic/org.eclipse.gef.doc.isv/reference/api/org/eclipse/gef/editparts/AbstractGraphicalEditPart.html` (12.03.2015)

Die Hauptklasse mit der `main`-Methode ist `de.uniol.pgktltt.server.KotelettServer`. Eingehende Nachrichten vom Client werden vom `KotelettNetworkServer` entgegengenommen und an `Registry` und `UserManager` weitergegeben. Der `ColorCalculator` wird benutzt um neuen Anwendern automatisch eine Farbe zuzuweisen. Genauer ist der Server in Unterabschnitt 9.5.5 beschrieben.

Das Package `de.uniol.pgktltt.server.voting` enthält die serverseitige implementierung des Voting-Systems, mit dem Anfragen an die Clients verschickt werden, bei denen alle Anwender zustimmen müssen. Die Benutzung ist sehr einfach und kann beispielsweise in der Methode `Registry.revertModel()` nachgesehen werden.

Kapitel 14

Logging

In Kotelett werden generell wenig Debug-Messages verwendet. In jedem Fall werden aber Exceptions geloggt. Sowohl auf Client als auf dem Server werden Logging-Ausgaben in den Standard-Output geschrieben.

14.1 Logging auf dem Server

Wenn das Startskript aus dem Installationshandbuch verwendet wird, wird der Output des Servers in die Datei `server.log` umgeleitet. Die Log-Messages enthalten keine Zeitstempel.

14.2 Logging auf dem Client

Unter Linux werden Log-Messages aus Kotelett automatisch ausgegeben, solange Kotelett über die Konsole gestartet wird. Auf Windows wird eine Konsole geöffnet, wenn Kotelett mit dem Parameter `-console` gestartet wird[1].

Neben den Log-Messages von Kotelett gibt es noch die Log-Messages des Eclipse-Frameworks. Diese werden standardmäßig in die Datei `workspace/.metadata/.log` im Arbeitsverzeichnis des Clients geschrieben. Wenn der Parameter `-consoleLog` verwendet wird (gegebenenfalls in Verbindung mit `-console`) werden die Eclipse-Log-Einträge auch in den Standard-Output geschrieben.

Darüber hinaus gibt es im Client eine Log-View, die die versendeten DOL-Statements gesondert anzeigt. Diese kann geöffnet werden, indem im Quick-Access-Feld oben rechts im Client „Log“ eingegeben wird.

[1] `http://help.eclipse.org/luna/topic/org.eclipse.platform.doc.isv/reference/misc/runtime-options.html` (12.03.2015)

Kapitel 15

Updates und zukünftige Nutzung

Das verwendete Framework GEF 3.x wird nicht mehr weiterentwickelt. Ebenso wurde die verwendete Eclipse-3.x-Technologie bereits von Eclipse4 abgelöst. Da GEF 3.x auf der alten Technologie basiert, in neueren Eclipse-Versionen also nur in einem Kompabilitätsmodus genutzt werden kann, und im Entwicklerteam zuvor keine Erfahrung mit Eclipse4 gemacht wurde, wurde Eclipse 3.4 als Platform beibehalten. Auf lange Sicht kann nicht sichergestellt werden, dass der Kotelett-Client auf allen Systemen unterstützt wird.

Kapitel 16

Erweiterungsmöglichkeiten

In diesem Abschnitt wird vorgestellt, wie sich das Produkt erweitern lässt, und an welchen Stellen dafür Hand angelegt werden muss. In Abschnitt Abschnitt 16.1 wird beschrieben, wie sich ein neues Modellelement in das Produkt einfügen lässt. Die Erweiterung um weitere Diagrammtypen ist erst möglich, wenn das Produkt mehrere Diagramme pro Modell vollständig unterstützt. Zu einem groben Überblick zu GEF siehe Abschnitt 8.2. Für Details muss hier auf Literatur (z.B. [Ecl14, RWC11]) verwiesen werden.

Neben dem Metamodell muss nur die GUI des Clients erweitert werden. Es müssen GEF-Edit-Parts zur Anzeige sowie GEF-Edit-Policies und GEF-Commands zum Bearbeiten der neuen Diagrammelemente erstellt werden. Zusätzlich müssen die `ClassDiagramEditPartFacory`, die Toolbar des Editors und die Baumansicht (`ModelTreeView`) erweitert werden.

16.1 Beispiel: UML-Notizen

Im Folgenden soll die Erweiterbarkeit anhand eines Beispiels beschrieben werden. Zu dem bestehenden Metamodell soll ein UML-Notzelement hinzugefügt werden, das an Klassen, Interfaces und Enumerationen angefügt werden kann.

16.1.1 Erweiterung des Metamodells

Zunächst wird das Metamodell um eine Klasse `KNote` erweitert, wie in Abbildung 16.1 angegeben. Notizen sind Knoten im Modell, weshalb `KNote` von `KNode` erben muss. Die Notizen sollen über eine Kante `KNoteConnection` mit den Knoten verbunden werden. Dazu wird eine zunächst eine weitere Abstrakte Klasse `KNoteTarget` erstellt, von der `KClassifier` und `KEnum` zusätzlich erben. Die Klasse `KNodeConnection` erbt dann von `KRelationship` und hat `Knote` und `KNoteTarget` als Source bzw. Target.

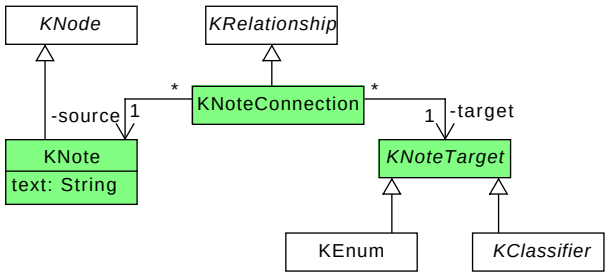

Abbildung 16.1: Ausschnitt des um `KNote` erweiterten Metamodells

16.1.2 Erweiterung der View

In der View muss dann zunächst ein `EditPart` (siehe Abschnitt 8.2) für das Notizelement erstellt werden. Die Klasse heißt hier `KNoteEditPart` und sollte von `KAbstractEditPart` (siehe Abschnitt 13.2) erben und `NodeEditPart` [Ecl14] implementieren. Bei der Implementierung kann z.B. `KEnumEditPart` (siehe Abschnitt 16.2) als Vorlage dienen. Der Editpart sollte die `GraphicalBox` (siehe Kapitel 10) der Notiz als Modellelement haben. Folgende Methoden müssen überschrieben werden [RWC11, Ecl14]:

- `createFigure()` Muss eine Notiz-Figur zurückgeben. Diese muss ggf. eigens dafür implementiert werden.
- `getModelSourceConnections()` Muss die verbundenen `KNoteConnection`-Elemente zurückgeben.
- `getSourceConnectionAnchor()` wie in `KEnumEditPart`.
- `performRequest()` für Direct-Editing. Es muss der entsprechende `DirectEditManager` instanziiert und aufgerufen werden.
- `createEditPolicies()` sollte folgende Edit-Policies [RWC11, Ecl14] installieren:
  - `ComponentEditPolicy`, die einen Delete-Command erzeugt (hierfür kann die `GenericDeleteCommandFactory` benutzt werden);
  - `DirectEditPolicy`, die einen Command zum Ändern des Notiz-Texts zurückgibt;
  - `GrahicalNodeEditPolicy`, bei der `GetConnectionCreateCommand()` überschrieben wird, um einen Command zu erzeugen, um eine `KNoteConnection` zu erstellen.

Für die `KNoteConnection` muss entsprechend ein `GraphicalConnectionEditPart` implementiert werden. Folgende Methoden müssen hier überschrieben werden:

- `createFigure()` Muss ein Draw2D-`PolylineConnection`-Objekt zurückgeben.
- `createEditPolicies()` sollte folgende Edit-Policies installieren:
  - `ConnectionEditPolicy`, die einen Delete-Command erzeugt;
  - `ConnectionEndpointEditPolicy` und
  - `KBendpointEditPolicy`, von denen keine Unterklasen erstellt werden müssen.

Folgende Commands müssen implementiert werden. Dort müssen jeweils die Methoden `execute()`, `undo()` und ggf. `redo()` überschrieben werden:

- `CreateKNoteCommand`
  Hier kann von `CreateNodeCommand` geerbt werden, sofern `KNote` von `KNode` erbt.
- `ChangeKNoteTextCommand`
- `CreateKNoteConnectionCommand`
- `ChangeKNoteConnectionCommand`

Für Delete-Commands existiert eine generische Implementierung, die benutzt werden kann. Details und Codebeispiele finden sich im nächsten Abschnitt. Existierende Klassen, die benutzt werden können sind im Wartungshandbuch, Kapitel 13 erklärt.

Für Direct-Editing müssen ein `DirectEditManager` und `CellEditorLocator` implementiert werden, um das Eingabefeld zu erzeugen und zu positionieren.

Ansonsten muss der Editor (`ClassDiagramEditor`) in der Methode `createAndFillPaletteDrawer()` um einen Eintrag zum Erstellen von Notizen und Verbindungen zu Notizen erweitert werden. Die `ClassDiagramEditPartFactory` muss um `KNoteEditPart` und `KNoteConnectionEditPart` erweitert werden. der `ModelTreeViewContentProvider` und `ModelTreeViewLabelProvider` müssen erweitert werden, sofern Notizen in der Baumansicht ebenfalls dargestellt werden sollen.

16.2 Detailliertes Beispiel: Enumerationen

Im folgenden Abschnitt wird anhand der Modellelemente `KEnum` und `KLiteral` die Erweiterung von Kotelett detaillierter beschrieben. `KEnum` und `KLiteral` sind bereits in Kotelett implementiert, die Codefragmente in diesem Abschnitt stammen daher direkt aus dem Projekt. Für die Anweisungen hier wird aber davon ausgegangen, dass diese noch nicht implementiert seien.

16.2.1 Anpassungen des Metamodells

Das Metamodell kann im Rational Software Architekt [IBM14] bearbeitet werden.[1] Es sind die Klassen `KEnum` und `KLiteral` hinzuzufügen, wie in Abbildung 16.2 angegeben. Das geänderte Modell wird im XMI-Format ex-

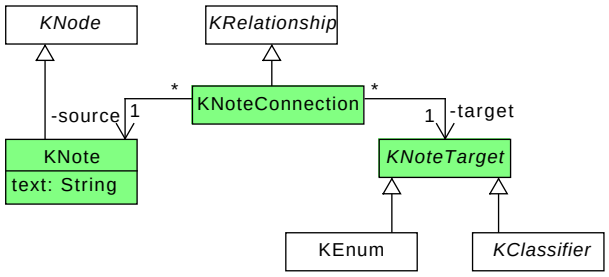

Abbildung 16.2: Ausschnitt des um `KEnum` erweiterten Metamodells

portiert. Mithilfe des im Projekt erstellten Konsolentools `JGraLabHelper` kann der Modellcode aus der XMI-Datei erstellt werden. Das Tool sollte in dem Verzeichnis aufgerufen werden, in dem die XMI-Datei liegt. Der Aufruf lautet

```
java -jar pfad/zu/JGraLabHelper.jar Modellname.xmi
```

Das Tool erstellt dann in einem Unterordner `dest/` den Modellcode sowie einen DOT-Graphen und JGraLab-Schema-Graphen im aktuellen Verzeichnis. Der Modellcode muss dann compiliert und zu einem Java-Archiv (JAR) `kotelett-model.jar` gepackt werden, das das existierende Archiv im Eclipse-Projekt `de.uniol.pgktltt.shared` ersetzt. Das Compilieren und Packen kann z.B. erfolgen, indem der Code in ein leeres Eclipse-Projekt eingefügt wird und dann von dem Projekt eine JAR exportiert wird. [2]

[1] Aufgrund geringer Unterschiede im XMI-Export sind andere Tools ungeeignet.

[2] Warum dieser Schritt nicht in den Buildprozess integriert wurde, ist im Wartungshandbuch, Abschnitt 11.3 erklärt.

16.2.2 Commands

Im Folgenden ist beschrieben welche GEF-Commands implementiert werden müssen. Alle Commands sollten von `AbstractModelCommand` erben, da dort sichergestellt wird, dass nur die Head-Version bearbeitet werden kann.

- `CreateKEnumCommand`: Dieser Command wird zum Erstellen von Enumerationen benötigt. Wir erben von `CreateNodeCommand`, der wiederum von `AbstractModelCommand` erbt, so dass nur die Methode `createNode` überschrieben werden muss.

```
1  package de.uniol.pgktltt.gef.client.commands;
2
3  import org.eclipse.draw2d.geometry.Rectangle;
4
5  import de.uniol.pgktltt.gef.model.Diagram;
6  import de.uniol.pgktltt.gef.model.KClassDiagram;
7  import de.uniol.pgktltt.gef.model.KEnum;
8
9  public class CreateKEnumCommand extends CreateNodeCommand {
10
11     private static final String DEFAULT_ENUM_NAME = "
   DefaultEnum";
12
13     public CreateKEnumCommand(Diagram diagram, Rectangle
   bounds) {
14         this.diagram = diagram;
15         this.bounds = bounds;
16         this.model = (KClassDiagram) diagram.getGraph();
17     }
18
19     @Override
20     protected KEnum createNode() {
21         KEnum kenum = model.createKEnum();
22         kenum.set_name(DEFAULT_ENUM_NAME);
23         return kenum;
24     }
25
26 }
```

Dieser Command kann jetzt in die `KClassDiagramEditPolicy` eingebaut werden. Hier muss in der Methode `getCreateCommand` ein Fall im `switch-case`-Block hinzugefügt werden.

```
45         switch (type) {
46         case CLASS:
47             return new CreateKClassCommand(diagram, bounds);
48         case ABSTRACT_CLASS:
49             return new CreateAbstractKClassCommand(diagram,
   bounds);
50         case INTERFACE:
51             return new CreateKInterfaceCommand(diagram,
   bounds);
```

```
52          case ENUM:
53              return new CreateKEnumCommand(diagram, bounds);
54          default:
55              return null;
56          }
57      }
```

Das Literal `ENUM` muss in der Form `ENUM(KEnum.VC)` dem Enum `ModelElementType` hinzugefügt werden. dies geschieht zusammen mit der Erweiterung des Editors in Unterabschnitt 16.2.4.

- `CreateKLiteralCommand`: Dieser Command wird zum Erstellen von Literalen benötigt.

  Die `execute`-Methode wird aufgerufen, wenn der Command ausgeführt wird. Hier soll das `KLiteral` erstellt werden. In Zeile 32 wird der Benutzer gesetzt, der als letztes die Enumeration bearbeitet hat. Dies ist der Benutzer dieses Clients.

  Die `redo`-Methode wird aufgerufen, wenn der Command erneut ausgeführt wird (REDO). Damit auf diesen Command folgende Commands ebenfalls wieder ausgeführt werden können, muss das alte JGraLab-Vertex wieder eingefügt werden, statt ein neues Literal zu erstellen.

  Die `undo`-Methode wird aufgerufen, wenn der Command widerrufen wird (UNDO). Das Literal wird einfach entfernt und der letzte Bearbeiter zurückgesetzt.

```
10  public class CreateKLiteralCommand extends
    AbstractModelCommand {
11
12      private KEnum kEnum;
13      private KClassDiagram model;
14      private String name;
15      KLiteral newKLiteral;
16      String newUserId, oldUserId;
17
18      public CreateKLiteralCommand(KEnum kEnum, String name) {
19          super("Create␣literal␣\"" + name + "\"", kEnum);
20          this.model = (KClassDiagram) kEnum.getGraph();
21          this.kEnum = kEnum;
22          this.name = name;
23          this.newUserId = ModelRegistry.getInstance().getUser
    ().getUserId();
24          this.oldUserId = kEnum.get_lastChangingUser();
25      }
26      @Override
27      public void execute() {
28          newKLiteral = model.createKLiteral();
29          newKLiteral.set_name(name);
30          kEnum.add_literal(newKLiteral);
31          kEnum.set_lastChangingUser(newUserId);
32          newKLiteral.set_lastChangingUser(newUserId);
```

```
33      }
34
35      @Override
36      public void redo() {
37          ((InternalVertex) newKLiteral).setId(0);
38          ((InternalGraph) model).addVertex(newKLiteral);
39          kEnum.add_literal(newKLiteral);
40          kEnum.set_lastChangingUser(newUserId);
41          newKLiteral.set_lastChangingUser(newUserId);
42      }
43      @Override
44      public void undo(){
45          kEnum.remove_literal(newKLiteral);
46          kEnum.set_lastChangingUser(oldUserId);
47      }
48  }
```

- `RenameKLiteralCommand`: Dieser Command ist für das Umbenennen eines Literals zuständig. In der `execute`-Methode wird der Name neu gesetzt und bei `undo` der alte Name wiederhergestellt.

```
14  public class RenameLiteralCommand extends
    AbstractModelCommand {
15
16      private KLiteral literal;
17      private String newName;
18      private String oldName;
19      private KEnum kEnum;
20      String newUserId, oldUserId, oldKEnumUserId;
21
22      public RenameLiteralCommand(KLiteral literal, String
    newName) {
23          super("Rename␣literal␣to␣\"" + newName + "\"",
    literal);
24          this.literal = literal;
25          this.newName = newName;
26          this.oldName = literal.get_name();
27          this.newUserId = ModelRegistry.getInstance().getUser
    ().getUserId();
28          this.oldUserId = literal.get_lastChangingUser();
29          this.kEnum = literal.getContainsLiteralIncidences().
    iterator().next().getAlpha();
30          this.oldKEnumUserId = this.kEnum.get_lastChangingUser
    ();
31      }
32
33      @Override
34      public void execute() {
35          literal.set_name(newName);
36          literal.set_lastChangingUser(newUserId);
37          kEnum.set_lastChangingUser(newUserId);
38      }
39
40      @Override
```

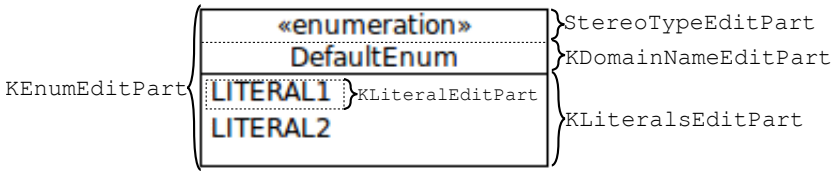

Abbildung 16.3: Darstellung und Verschachtelung der verwendeten Edit-Parts.

```
41      public void undo() {
42          this.literal.set_name(oldName);
43          this.literal.set_lastChangingUser(oldUserId);
44          this.kEnum.set_lastChangingUser(oldKEnumUserId);
45      }
46  }
```

16.2.3 Edit Parts

Es müssen mindestens zwei Editparts, je einer für Enumerationen und Literale, implementiert werden. Bei dieser Implementierung wird ein weiterer Editpart für den Teil einer Enumeration benötigt, in dem die Literale angezeigt werden. Alle Editparts sollten von `KAbstractEditPart` erben.

Enumerationen (Haupt-Part) Der `KEnumEditPart` ist der Haupt-Part für Enumerationen. Da Enumerationen in ihrer Darstellung ähnlich wie Klassen in zwei separate Bereiche gegliedert sind, dient dieser Edit-Part lediglich als Container für die inneren Parts. Dadurch kann präzise gesteuert werden, wo die eigentlichen Kinder (in diesem Fall Literale) angezeigt werden sollen.

Die Methode `getModelChildren` (Zeile 106) gibt die Kinder auf Modellebene zurück. Hier kommt die soeben erläuterte Strategie zum Tragen: Der Editpart für Enumerationen hat zunächst drei Kinder, je einen Editpart für den Text „«KEnum»“, einen für den Namen und einen, in dem die Literale hinzugefügt werden. Die ersten beiden werden in Kotelett auch für Interfaces und Klassen verwendet, sodass nur `KLiteralsEditPart` neu erstellt werden muss. Wie die verwendeten Edit-Parts dargestellt werden, ist in Abbildung 16.3 dargestellt.

In `createEditPolicies` (Zeile 61) werden die EditPolicies erstellt. Diese Kapseln die Behandlung von Nutzereingaben, in diesem Fall nur Löschen der Enumeration.

`refreshVisuals` (Zeile 73) wird von GEF aufgerufen, wenn die Darstellung des Editparts aktualisiert werden soll. Hier wird dies benutzt, um die Positionen der Enumeration an den übergeordneten Editpart zu übergeben, der für das Layout zuständig ist.

Die Methode `setColor` ist eine Hilfsmethode, um die Figure gemäß des letzten Bearbeiters einzufärben.

```
42 public class KEnumEditPart extends KAbstractEditPart {
43
44     private List<Object> childrenList = new ArrayList<Object>();
```

In `createFigure` wird die Draw2D-Figure erstellt um eine Enumeration zu zeichnen.

```
45     @Override
46     protected IFigure createFigure() {
47         RectangleFigure rectangleFigure = new RectangleFigure();
48         rectangleFigure.setLayoutManager(new ToolbarLayout());
49         return rectangleFigure;
50     }
51
52     public KNode getKNode() {
53         return this.getRepresentation().get_kNode();
54     }
55
56     private KEnum getKEnum() {
57         return (KEnum) getKNode();
58     }
59
60     @Override
61     protected void createEditPolicies() {
62         this.installEditPolicy(EditPolicy.COMPONENT_ROLE, new
   ComponentEditPolicy() {
63             @Override
64             protected Command getDeleteCommand(GroupRequest
   request) {
65                 return GenericDeleteCommandFactory.
   createDeleteCommand(getRepresentation());
66             }
67         });
68
69
70     }
71
72     @Override
73     protected void refreshVisuals() {
74         BoundingBox representation = this.getRepresentation();
75         if (representation == null) {
76             return;
77         }
78         Rectangle rect = new Rectangle(representation.get_x(),
   representation.get_y(), representation.get_width(), representation
   .get_height());
79         ((AbstractGraphicalEditPart) this.getParent()).
   setLayoutConstraint(this, getFigure(), rect);
80
81         setColor(getFigure());
82     }
```

```
83
84      private void setColor(IFigure figure)
85      {
86          if (!ColorHelper.getColorsVisible())
87          {
88              figure.setBackgroundColor(null);
89              return;
90          }
91
92          String lastChangedUserId =  ((ModelElement) this.
    getRepresentation().get_kNode()).get_lastChangingUser();
93
94          int[] color = ColorHelper.getColorByUserId(
    lastChangedUserId);
95          org.eclipse.swt.graphics.Color c = new org.eclipse.swt.
    graphics.Color(null, color[0], color[1], color[2]);
96
97          figure.setBackgroundColor(c);
98      }
99
100     protected BoundingBox getRepresentation() {
101         return (BoundingBox) getModel();
102     }
103
104     @SuppressWarnings("rawtypes")
105     @Override
106     public List getModelChildren() {
107         return this.childrenList;
108     }
109
110     @Override
111     public void setModel(Object model) {
112         super.setModel(model);
113         this.childrenList.clear();
114         this.childrenList.add(new GenericWrapper<KEnum>(this.
    getKEnum(), StereoTypeEditPart.class));
115         this.childrenList.add(new KDomainNameModel(this.getKEnum()
    ));
116         this.childrenList.add(new GenericWrapper<KEnum>(this.
    getKEnum(), KLiteralsEditPart.class));
117     }
118 }
```

Sub-Part für Literale Dieser Editpart, `KLiteralsEditPart`, der oben schon dem `KEnumEditPart` hinzugefügt wurde, beinhaltet die Literale. Diese werden von `getModelChildren` in Zeile 69 zurückgegeben.

Ab Zeile 36 werden wieder die Edit-Policies registriert. Das Direct-Editing wird benutzt, um neue Literale zu erstellen.

Der Editpart selbst soll nicht anwählbar sein, damit der Anwender die gesamte Enumeration besser auswählen kann. Daher wird `isSelectable` in Zeile 88 überschrieben und `false` zurückgegeben. (Der Event, um das Direct-Editing zu starten, wird vom `KEnumEditPart` bzw. einer Oberklasse

weitergeleitet. In `isSelectable false` zurückzugeben unterbindet nämlich jegliche Events.)

In Zeile 57 wird das Direct-Editing als Antwort auf einen entsprechenden Request gestartet. Es wird der `CompartmentDirectEditManager` genutzt, eine Hilfsklasse, die auch in anderen Editparts in Kotelett eingesetzt wird.

```
27 public class KLiteralsEditPart extends KAbstractEditPart {
28
29     @Override
30     protected IFigure createFigure() {
31         return new CompartmentFigure();
32     }
33
34     @Override
35     protected void createEditPolicies() {
36         this.installEditPolicy(EditPolicy.DIRECT_EDIT_ROLE, new
   DirectEditPolicy() {
37
38             @Override
39             protected Command getDirectEditCommand(
   DirectEditRequest request) {
40                 String name = (String) request.getCellEditor().
   getValue();
41                 if (!name.trim().isEmpty()) {
42                     return new CreateKLiteralCommand(getKEnum(),
   name);
43                 } else {
44                     return null;
45                 }
46             }
47
48             @Override
49             protected void showCurrentEditValue(DirectEditRequest
   request) {
50
51             }
52
53         });
54     }
55
56     @Override
57     public void performRequest(Request req) {
58         if (req.getType() == RequestConstants.REQ_DIRECT_EDIT) {
59             CompartmentDirectEditManager manager = new
   CompartmentDirectEditManager(this);
60             manager.show();
61         } else {
62             super.performRequest(req);
63         }
64     }
65
66     @SuppressWarnings("rawtypes")
67     @Override
68     protected List getModelChildren() {
```

```
69          List<KLiteral> result = Lists.newArrayList(getKEnum().
    get_literal());
70          Collections.sort(result, new Comparator<KLiteral>() {
71
72              @Override
73              public int compare(KLiteral arg0, KLiteral arg1) {
74                  return ("" + arg0.get_name()).compareTo("" + arg1.
    get_name());
75              }
76
77          });
78          return result;
79      }
80
81      @SuppressWarnings("unchecked")
82      public KEnum getKEnum() {
83          return ((GenericWrapper<KEnum>) getModel()).getModel();
84      }
85
86      @Override
87      public boolean isSelectable() {
88          return false;
89      }
90
91  }
```

Editpart für Literale Dieser Edit-Part repräsentiert nun ein einziges Literal. Die in Zeile 30 zurückgegebene Figure zeigt einfach den Namen des Literals an. Bei einer Aktualisierung muss der Text in der Figure angepasst werden, dies geschieht durch Überschreiben von `refreshVisuals` in Zeile 80.

Die Edit-Policies werden ab Zeile 42 registriert. Die erste ist für das Entfernen eines Literals zuständig, die zweite malt einen Kasten um ein selektiertes Literal und die dritte antwortet wieder auf Direct-Editing, nur das hier ein `RenameLiteralCommand` statt `CreateLiteralCommand` erstellt wird.

```
27  public class KLiteralEditPart extends AbstractNameEditPart {
28
29      @Override
30      protected IFigure createFigure() {
31          Label figure = new LeftAlignedLabel(getLiteral().get_name
    ());
32          return figure;
33      }
34
35
36      KLiteral getLiteral() {
37          return (KLiteral) getModel();
38      }
39
```

```
40      @Override
41      protected void createEditPolicies() {
42          installEditPolicy(EditPolicy.COMPONENT_ROLE, new
    ComponentEditPolicy() {
43              @Override
44              protected Command createDeleteCommand(GroupRequest
    deleteRequest) {
45                  return new DeleteKLiteralCommand(getLiteral());
46              }
47          });
48
49          installEditPolicy(EditPolicy.SELECTION_FEEDBACK_ROLE, new
    SelectionEditPolicy() {
50
51              @Override
52              protected void showSelection() {
53                  KLiteralEditPart.this.figure.setBorder(new
    LineBorder(1));
54              }
55
56              @Override
57              protected void hideSelection() {
58                  KLiteralEditPart.this.figure.setBorder(null);
59              }
60          });
61
62          installEditPolicy(EditPolicy.DIRECT_EDIT_ROLE, new
    DirectEditPolicy() {
63
64              @Override
65              protected void showCurrentEditValue(DirectEditRequest
    request) {
66                  System.out.println(request.getCellEditor().
    getValue());
67              }
68
69              @Override
70              protected Command getDirectEditCommand(
    DirectEditRequest request) {
71                  return new RenameLiteralCommand(getLiteral(),
72                          (String) request.getCellEditor().getValue
    ());
73              }
74          });
75
76      }
77
78      @Override
79      protected void refreshVisuals() {
80          super.refreshVisuals();
81          Label label = (Label) this.getFigure();
82          label.setText(this.getLiteral().get_name());
83      }
84  }
```

16.2.4 Andere Anpassungen im Code

Im Folgenden werden die weiteren Anpassungen beschrieben, die nötig sind, um Enumerationen in den Client zu integrieren. Dies sind u.a. Erweiterungen in der Toolbar des Editors und der Baumansicht.

- Im Editor `ClassDiagramEditor.java` muss in der Methode `createAndFillPaletteDrawer` ein Eintrag hinzugefügt werden, um Enumerationen erstellen zu können.

```
98          tool = new CreationToolEntry("Enum", "Enumeration",
    new ClassDiagramFactory(ModelElementType.ENUM), loadImage("
    Enum16" + imageSufix),
99                  loadImage("Enum32" + imageSufix));
```

  Für die Icons müssen Bilder `Enum16.png`, `Enum16b.png`, `Enum16o.png` in Größe 16×16 in def Farben Schwarz, Blau und Orange sowie `Enum32.png`, `Enum32b.png`, `Enum32o.png` in Größe 32×32 im Projektordner `de.uniol.pgktltt.gef.client/icons` vorhanden sein.

- Im Enum `ModelElementType.java` muss ein Literal für Enumerationen `ENUM(KEnum.VC)` hinzugefügt werden.

- Die Klasse `ClassDiagramEditPartFactory.java` muss an zwei Stellen erweitert werden, um Instanzen von `KEnumEditPart` und `KLiteralEditPart` zu erstellen. Diese sind im Folgenden farbig hervorgehoben.

```
26          } else if (layoutObject instanceof KMethod) {
27              result = new KMethodEditPart();
28          } else if (layoutObject instanceof KLiteral) {
29              result = new KLiteralEditPart();
30          } else {
31              throw new IllegalArgumentException("Unkown␣
    Layoutobject␣" + layoutObject.getClass());
32          }
33          if (modelObject instanceof Diagram) {
34              result = new KClassDiagramEditPart();
35          } else if (modelObject instanceof KClassifier) {
36              result = new KClassifierEditPart();
37          } else if (modelObject instanceof KEnum) {
38              result = new KEnumEditPart();
39          } else if (modelObject instanceof KAssociation) {
40              result = new AssociationEditPart();
```

- Für die Baumansicht muss der `ViewContentProvider` erweitert werden, um Enumerationen und Literale anzuzeigen. in den Methoden `getChildren`, `getParent` und `hasChildren` muss jeweils ein Eintrag hinzugefügt werden.

```
46      @Override
47      public ObjectObject getChildren(Object parentElement) {
```

```
48          /*
49           * Model1 Diagramme DiagrammA Anzegeigte Elemente?
    DiagrammB
50           * Angezeigte Elemente? Modellelemente Vorhandenen
    Elemente
51           *
52           * Model2 Diagramme
53           *
54           * Modellelemente
55           */
56          List<Object> list = new ArrayList<>();
57          if (parentElement instanceof KClassDiagram) {
58              KClassDiagram model = (KClassDiagram)
    parentElement;
59              list.add(new DiagramContainer(model.
    getDiagramVertices()));
60              list.add(new ModelElementsContainer(model.
    getKNodeVertices()));
61          } else if (parentElement instanceof DiagramContainer)
     {
    ...

126         } else if (parentElement instanceof KEnum) {
127             for (KLiteral literal : ((KEnum) parentElement).
    get_literal()) {
128                 list.add(literal);
129             }
130         } else if (parentElement instanceof KAssociation) {
131             KAssociationEndPoint source = ((KAssociation)
    parentElement)
132                     .get_source();
133             KAssociationEndPoint target = ((KAssociation)
    parentElement)
134                     .get_target();
135             list.add(source);
136             list.add(target);
137         } else if (parentElement instanceof KGeneralization)
    {
138             KClassifier source = ((KGeneralization)
    parentElement)
139                     .get_source();
140             KClassifier target = ((KGeneralization)
    parentElement)
141                     .get_target();
142             list.add(source);
143             list.add(target);
144         }
145         return list.toArray();
146
147     }

    ...

150     @Override
151     public Object getParent(Object element) {
```

```
152         if (element instanceof KClassDiagram) {

...

126         } else if (element instanceof KEnum) {
127             return ((KEnum) element).getGraph();
128         } else if (element instanceof KLiteral) {
129             return ((KLiteral) element).
getFirstContainsLiteralIncidence(
130                     EdgeDirection.IN).getAlpha();

...

197     @Override
198     public boolean hasChildren(Object element) {
199         if (element instanceof KClassDiagram) {

...

126         } else if (element instanceof KEnum) {
127             return this.hasMoreTheZeroChildren(element);
128         } else if (element instanceof KLiteral) {
129             return false;
130         } else if (element instanceof KPrimitiveType) {
131             return false;
132         } else if (element instanceof KAssociation) {
133             return this.hasMoreTheZeroChildren(element);
134         } else if (element instanceof KAssociationEndPoint) {
135             return false;
136         } else if (element instanceof KGeneralization) {
137             return false;
138         } else {
139             return false;
140         }
141     }
142
143     private boolean hasMoreTheZeroChildren(Object element) {
144         return this.getChildren(element).length > 0;
145     }
146
147 }
```

- In der Klasse `ViewLabelProvider` muss die Methode `getElementLabel` um Einträge für Enumerationen und Literale erweitert werden.

```
48     private String getElementLabel(Object element) {
49         String label = "";
50         if (element instanceof KClassDiagram) {

...

126         } else if (element instanceof KEnum) {
127             label += "Enum:␣" + ((KEnum) element).get_name();
```

```
128          } else if (element instanceof KLiteral) {
129              label += "Literal:␣" + ((KLiteral) element).
    get_name();
    ...
```

16.3 Probleme bei der Erweiterbarkeit

Um Kotelett um weitere Modellelemente zu erweitern muss generell nur die Nutzerschnittstelle des Clients angepasst werden; die Client-Server-Kommunikation sowie der Server sind soweit generisch, dass eine Erweiterung des Metamodells um einzelne Modellelemente keine zusätzliche Funktionalität erfordert.[3] Das die View weitaus weniger generisch ist, hat die einfache Ursache, dass ein Metamodell zu seiner graphischen Repräsentation prinzipiell nicht in Zusammenhang steht und damit jedes Element und jede Nutzerinteraktion einzelnt behandelt werden müssen. Bei der Entwicklung des Editors wurde sich an die von GEF vorgegebenen Pattern gehalten. Möglicherweise lassen sich durch ein Refactoring noch Teile finden, wo beispielsweise wiederkehrende Funktionalität in den Edit-Parts in Oberklassen ausgelagert werden kann oder die if-else-Kaskaden in einigen Klassen vermieden werden können. Ein solches Refactoring oder dies noch mehr in der Planung zu berücksichtigen, war aufgrund der verkürzten Entwicklungszeit (siehe Prozessbericht) nicht möglich.

[3] Das Entwicklerteam kann diese Aussage selbstverständlich nur bezüglich des ihm bekannten Kotelett-Codes treffen. Über die Erfahrungen diesbezüglich in der DOLSA-API wird in Abschnitt 8.4 berichtet.

Teil V

Evaluation

Kapitel 17

Einleitung

Dieses Kapitel beschäftigt sich mit der Evaluation des Produktes, welche im Rahmen des Projektes entwickelt wurde. Der Prozess des Projektes, welches ein wesentliches Ergebnis darstellt, wird im Prozessbericht evaluiert. Um die Software zu evaluieren wurden Akzeptanztests auf Basis der Anforderungen in Teil II erstellt. Aufgrund der Unvollständigkeit der Globalen Analyse in Kapitel 6 wurden ebenso Informationen aus der Architektur in Teil III verwendet. Die Traceability dieser Akzeptanzstests ist somit nicht lückenlos. Außerdem wurden die Evaluationskriterien aus der Zielvereinbarung ausgewertet (Abschnitt 1.6).

Dieser Evaluierungsprozess hat zum Ziel neben der Evaluierung des Produkts eine möglichst vollständige Dokumentation der Fehlerfälle für die Weiterentwicklung zu liefern.

Die getestete Software baut auf Eclipse auf welches SWT 4.3 als Widget-Toolkit-Framework verwendet. Da SWT 4.3 ein Wrapper für die nativen Widget-Toolkits ist, führt dies unter jedem der folgenden nativen Widget Toolkits zu unterschiedlichem Verhalten [Ecl15]:

- win32 (Windows)
- GTK2 (Linux / Unix / Windows / OS X)
- Cocoa (32-bit OS X)
- Cocoa (64-bit OS X)
- Motif (Unix / Linux)

Kapitel 18

Fehlgeschlagene Akzeptanztests

Dieser Abschnitt beinhaltet jene Akzeptanztests, welche fehlgeschlagen sind mit dem unerwarteten Verhalten.

18.1 Klassenattribut mit protected Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Klassenattributes mit protected Sichtbarkeit mit Leerzeichen ab. Dieser Fehler tritt bei Abstrakten Klassen und Interfaces ebenfalls auf.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Klassennamen klicken.
2. Im Texteingabefeld „ hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Klassenattribut angezeigt mit Leerzeichen.
2. Links in der ModelTreeView wird das neue Klassenattribut in der Klasse angezeigt mit Leerzeichen.

Tatsächliches Ergebnis

1. Eine Fehlermeldung erscheint mit der Meldung „Unrecognized modifier in hello : Void“.

18.2 Bearbeiten des Quell-Rollenbezeichners an Assoziationen mit Protected-Sichtbarkeit

Dieser Test deckt das Bearbeiten des Quell-Rollenbezeichners mit Public-Sichtbarkeit ab. Dieser Fehler tritt bei Ziel-Rollenbezeichnern ebenfalls auf.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation erstellt.
3. Es wurde der Quell-Rollenbezeichner selektiert.

Test

1. Im Editor-Bereich auf das selektierte Textfeld klicken.
2. „ hallo“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird „ hallo“ im selektierten Textfeld angezeigt.
2. In der ModelTreeView links wurde der Wert des Textfelds and der Assoziation aktualisiert.
3. Die Sichtbarkeit des Rollenbezeichners wird in der ModelTreeView nicht angezeigt.

Tatsächliches Ergebnis

1. Das Sichtbarkeitssymbol wird als Beginn des Namens interpretiert.
2. Der Name des Rollenbezeichners lautet „ hallo“

18.3 Bearbeiten des Quell-Rollenbezeichners an Assoziationen mit Protected-Sichtbarkeit (mit Leerzeichen)

Dieser Test deckt das Bearbeiten des Quell-Rollenbezeichners mit Public-Sichtbarkeit ab. Dieser Test wurde aufgrund des Fehlers beim Parsen der Sichtbarkeit bei Klassenattributen (Abschnitt 18.1) durchgeführt. Dieser Fehler tritt auch bei Ziel-Rollenbezeichnern auf.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation erstellt.
3. Es wurde der Quell-Rollenbezeichner selektiert.

Test

1. Im Editor-Bereich auf das selektierte Textfeld klicken.
2. „ hallo“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird „ hallo“ im selektierten Textfeld angezeigt.
2. In der ModelTreeView links wurde der Wert des Textfelds and der Assoziation aktualisiert.
3. Die Sichtbarkeit des Rollenbezeichners wird in der ModelTreeView nicht angezeigt.

Tatsächliches Ergebnis

1. Das Sichtbarkeitssymbol wird als Beginn des Namens interpretiert.
2. Der Name des Rollenbezeichners lautet „ hallo“

18.4 Undo der Erstellung einer Generalisierung

Dieser Test deckt das Undo der Erstellung einer Generalisierung von einer Klasse zu einer Klasse ab.

Vorbedingung

1. Es wurden 2 Klassen erstellt.
2. Es wurde eine Generalisierung zwischen diesen erstellt.

Test

1. Strg+Z drücken (Undo).

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die erstellte Generalisierung nicht mehr angezeigt.
2. Links in der ModelTreeView wird die Generalisierung nicht mehr angezeigt.

Tatsächliches Ergebnis

1. Die Funktion, die Erstellung der Generalisierung Rückgängig zu machen, ist nicht verfügbar.
2. Dieses Fehlverhalten tritt für alle möglichen Generalisierung zwi

18.5 Umhängen des Startpunktes einer Generalisierung

Dieser Test deckt das Umhängen des Startpunktes einer Generalisierung ab. Dieser Test ist generisch formuliert, da das Problem hierbei eine fehlende Überprüfung der Constraints beim Umhängen ist. Diese Constraints sind bei der Erstellung von Generalisierungen eingehalten.

Vorbedingung

1. Es wurden 3 Knoten erstellt.
2. Es wurde eine Generalisierung zwischen diesen erstellt.
3. Die Generalisierung wurde selektiert.

Test

1. Der Startpunkt der Generalisierung wird per Drag and Drop auf einen anderen Knoten gezogen.

Erwartetes Ergebnis

1. Der Startpunkt der Generalisierung wird nur dann verändert, wenn die daraus resultierende Generalisierung gültig ist.

Tatsächliches Ergebnis

1. Die Constraints der Generalisierung werden beim Umhängen des Startpunktes nicht überprüft.

2. Die möglichen ungültige Resultate sind (Diese Menge ist vollständig):

   (a) Generalisierung von Interface zu Klasse

   (b) Generalisierung von Interface zu Abstrakter Klasse

   (c) Generalisierung mit dem selben Knoten als Start- und Endpunkt (Selbst-Transition)

18.6 Umhängen des Endpunktes einer Generalisierung

Dieser Test deckt das Umhängen des Endpunktes einer Generalisierung ab. Dieser Test ist generisch formuliert, da das Problem hierbei eine fehlende Überprüfung der Constraints beim Umhängen ist. Diese Constraints sind bei der Erstellung von Generalisierungen eingehalten.

Vorbedingung

1. Es wurden 3 Knoten erstellt.

2. Es wurde eine Generalisierung zwischen diesen erstellt.

3. Die Generalisierung wurde selektiert.

Test

1. Der Endpunkt der Generalisierung wird per Drag and Drop auf einen anderen Knoten gelegt.

Erwartetes Ergebnis

1. Der Endpunkt der Generalisierung wird nur dann verändert, wenn die daraus resultierende Generalisierung gültig ist.

Tatsächliches Ergebnis

1. Die Constraints der Generalisierung werden beim Umhängen des Endpunktes nicht überprüft.

2. Die möglichen ungültige Resultate sind (Diese Menge ist vollständig):

   (a) Generalisierung von Interface zu Klasse

   (b) Generalisierung von Interface zu Abstrakter Klasse

   (c) Generalisierung mit dem selben Knoten als Start- und Endpunkt (Selbst-Transition)

18.7 Redo vom Umhängen einer Assoziation, nachdem ein anderer Anwender die Zielklasse gelöscht hat

Dieser Test deckt das Redo vom Umhängen einer Assoziation, nachdem ein anderer Anwender die Zielklasse gelöscht hat ab.

Vorbedingung

1. Zwei Anwender sind am System angemeldet.

2. Es wurden drei Klassen erstellt.

3. Es wurde eine Assoziation zwischen 2 Klassen erstellt.

4. Das Ziel der Assoziation wurde geändert.

5. Das Ändern des Zieles der Assoziation wurde rückgängig gemacht (Strg+Z).

6. Der andere Anwender löscht das vorherige Ziel der Assoziation (mit dem die Assoziation jetzt nicht mehr verbunden ist)

Test

1. Der Anwender, welcher das Ändern des Zieles der Assoziation rückgängig gemacht hat, führt ein „Redo“ durch (Strg+Shift+Z).

Erwartetes Ergebnis

1. Das Redo ist nicht durchführbar, da die Zielklasse nicht mehr existiert.

Tatsächliches Ergebnis

1. Eine NullPointerException tritt auf und das Modell wird inkonsistent.
2. Hierauf folgt nicht definiertes Verhalten.

18.8 Leeren der Userliste beim Schließen des letzten Editors

Dieser Test deckt das Leeren der Userliste beim Schließen des letzten Editors ab.

Vorbedingung

1. Es ist ein Modell geöffnet.
2. Zu diesem Modell ist ein Editor geöffnet.

Test

1. Der Anwender schließt den Editor.

Erwartetes Ergebnis

1. Da kein Editor geöffnet ist, ist die Userliste leer.

Tatsächliches Ergebnis

1. Die Userliste wird nicht geleert.
2. Wenn man jetzt z.B. seine Farbe ändert, kann der Client nicht mehr mit dem Server kommunizieren.

Kapitel 19

Evaluation nach Zielvereinbarung

In diesem Kapitel werden die in der Zielvereinbarung definierten Evaluationskriterien ausgewertet (siehe Abschnitt 1.6).

FA-I **Das System muss dem Anwender die Modellierung von Klassendiagrammen ermöglichen.** Die verfügbaren Sprachmittel sind dabei durch das Metamodell definiert. Die grapische Repräsentation soll sich dabei am UML 2.4.1 Standard orientieren [Obj11]. Als Akzeptanztest soll der Nutzer in der Lage sein die StClock zu modellieren. Dabei soll das Ergebniss semantisch äquivalent zu dem Klassendiagramm in Abbildung 1.2 sein.

Folgende Sprachmittel unterstützt Kotelett nicht:

- Default Werte für Attribute
- Leere Rollenbezeichner
- Konstruktoren
- Arrays
- Namespaces in Klassennamen
- long, int und alle JDK Klassen fehlen. Es existiert allerdings Integer.

In Abbildung 19.1 sind die Fehler in Rot markiert. Die leeren Kästen stehen dabei für leere Rollenbezeichner, die Kotelett nicht unterstützt. Den größten Aufwand hierbei stellen Arrays dar, da sie weitere Datentypen für jeden vorhandenen Datentyp darstellen.

FA-II **Das System muss den Zustand eines Modells unter allen bearbeitenden Anwendern in Echtzeit synchronisieren.** Um kollaboratives Arbeiten zu ermöglichen, müssen die Clients der Anwender

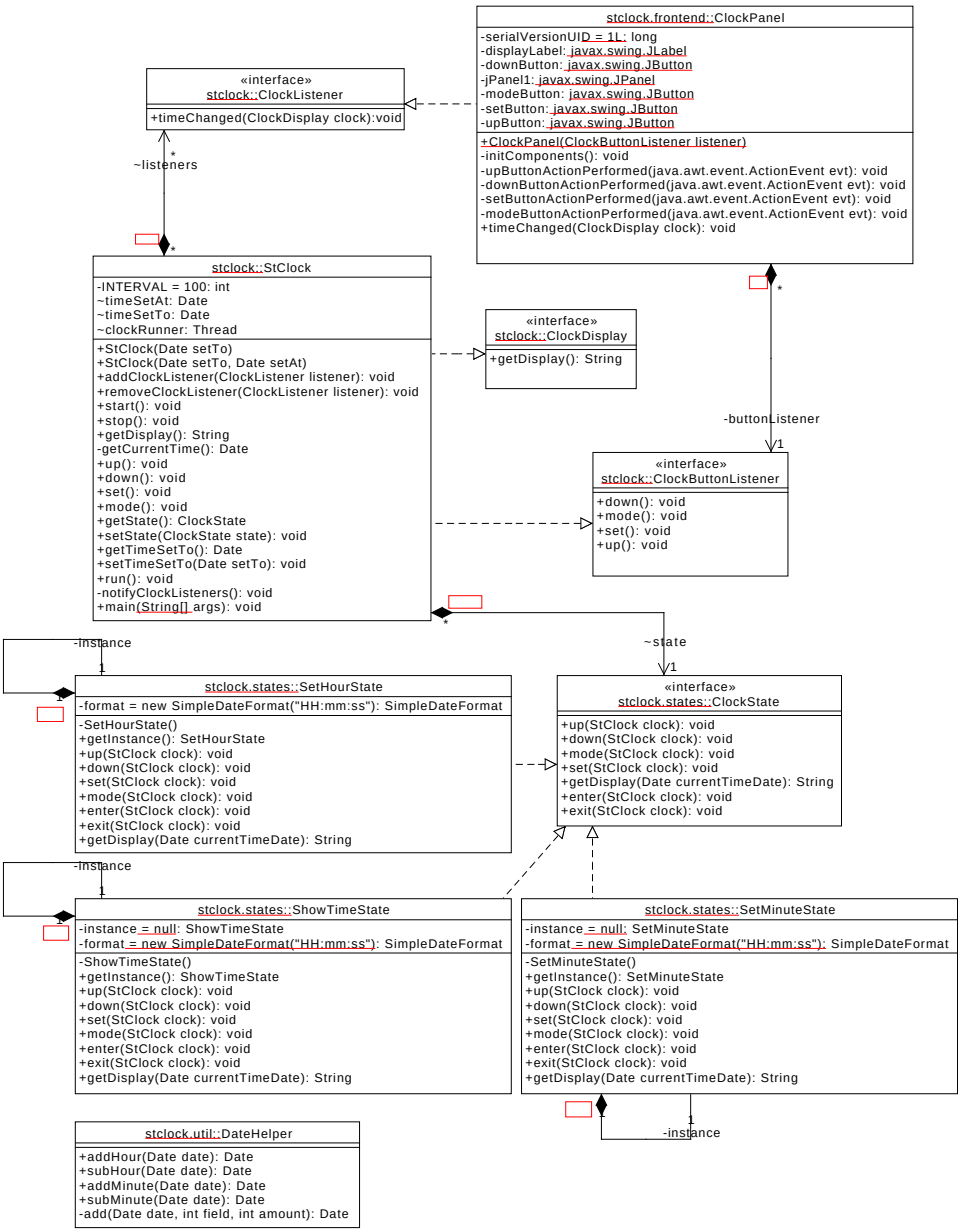

Abbildung 19.1: StClock

in Echtzeit synchronisiert werden. Ein asynchroner Arbeitsablauf, wie in Git und SVN umgesetzt, genügt für den gewünschten Grad der Kollaborativität nicht. In klassischen Versionskontrollsystemen fügt der Benutzer seine Änderungen manuell dem Repository hinzu, und er muss auch manuell eine neue Version des Repositories abrufen. Beides soll vom System automatisiert geschehen, so dass Änderungen sofort

hinzugefügt werden, und dann sofort an alle anderen Bearbeiter verteilt werden.

Die Anforderung ist erfüllt. Das Tool wurde zum Beispiel beim Schülerinformationstag (siehe Prozessbericht) eingesetzt.

FA-III **Das System muss es dem Anwender ermöglichen, mehrere Modelle zu verwalten.** Anwender wollen mehrere Modelle erstellen, löschen und bearbeiten können, da bereits verhältnismäßig kleine Softwaresysteme mehrere Modelle benötigen. Diese Anforderung ist zudem für die Erweiterbarkeit des Systems sehr wichtig.

Die Anforderung ist erfüllt.
Siehe dazu die Akzeptanztests (Abschnitt 22.1).

FA-IV **Das System muss es dem Anwender ermöglichen, zu früheren Versionen eines Modells zurückzuspringen.** Anwender in der Softwareentwicklung benötigen Versionierung um auf frühere Versionen zurückzugreifen. Dies kann genutzt werden, um in Fehlerfällen frühere Versionen zu verwenden oder die Modellhistorie nachzuvollziehen.

Die Anforderung ist erfüllt.
Siehe dazu die Akzeptanztests (Abschnitt 22.3).

NFA-I **Die Synchronisierung aus Anforderung FA-II muss unter Verwendung der DOL-Services geschehen.** Der Auftraggeber will, dass die DOL als Austauschformat zur Synchronisierung verwendet wird. Um dies zu realisieren, wird dem Team die „DOL-Service“ Komponente von DOLSA zur Verfügung gestellt.

Die Anforderung ist erfüllt. Dies ist in Abschnitt 8.4 dokumentiert.

NFA-II **Die Modellversionierung aus Anforderung FA-IV muss durch die Einbindung von Versioning erfolgen.** Da Modellversionierung ein aufwendiges Feature ist, wird eine Schnittstelle zum Modellversionierungssystem Versioning entwickelt, womit die PG die Versionierung realisieren darf. Dieses System wird von der ST-Abteilung bereitgestellt.

Die Anforderung ist erfüllt. Dies ist in Abschnitt 8.4 dokumentiert.

Kapitel 20

Bekannte Fehler

Dieses Kapitel beschreibt offene bekannte Fehler, die aufgrund Zeitmangels nicht während des Projektes behoben werden konnten.

20.1 Fehlgeschlagene Akzeptanztests

Die fehlgeschlagenen Akzeptanztests in Kapitel 18 dokumentieren meist kleinere Fehler reproduzierbare Fehler mit einem Testfall, sodass diese in einer potenzieller Weiterentwicklung durch eingearbeitete Entwickler behoben werden können.

20.2 Verbindungstabilität ist nicht gewährleistet

Die Verbindung zum Server kann unvorhersehbar abbrechen. Dies ist vermutlich ein Problem im Zusammenspiel zwischen KryoNet und der Eclipseumgebung, da KryoNet schon in reinen Javaprojekten sehr zuverlässig eingesetzt wurde.

20.3 Gleichzeitige Bearbeitung eines Elements

Wenn zwei Benutzer ein Element zugleich bearbeiten, ist das Verhalten von Kotelett undefiniert. Es kann passieren, dass die Deltas auf einem Client vertauscht werden, wodurch die jeweilige Änderung des Users überschrieben wird. Um dies zu beheben müssten die Deltas zum Beispiel mit Zeitstempeln versehen sein.

20.4 Löschen von Modellen

In wenigen nicht-reproduzierbaren Fällen kann das Löschen von Modellen zum Absturz des Servers führen. In diesem Fall wird an den Nutzer eine er-

folgreiche Löschung des Modells gemeldet, aber das Modell taucht weiterhin in der Liste verfügbarer Modelle auf.

Leider führt dieser Fehler zu einem korrumpierten Modellrepository, weshalb die einzige mögliche Lösung ein Herunterfahren des Servers und manuelles Löschen des Modells im Repository ist. Um das korrumpierte Modellrepository zu reparieren, muss im Arbeitsverzeichnis des Servers das Unterverzeichnis `repository` geöffnet werden. Darin befinden sich mehrere Unterverzeichnisse, die jeweils mehrere Dateien enthalten. In einem der Verzeichnisse sollte sich *lediglich eine XML-Datei* befinden. Dieses Verzeichnis muss gelöscht werden. Danach kann der Server wieder gestartet werden.

20.5 Aktualisierung der Userliste

Die Userliste soll immer die Nutzer des aktuell ausgewählten Editors anzeigen. Dies funktioniert nicht immer richtig. Bei mehreren Modellen kommt es immer wieder vor, dass eine falsche Userliste angezeigt wird. Dies tritt bei User Beitritten zu nicht aktiven Modellen und auch bei Namens- sowie Farbänderungen auf. Außerdem wird die Userliste nicht gelöscht, wenn der letzte Editor geschlossen wird (s. Abschnitt 18.8).

20.6 Undo/Redo-Problematik

Ein bekannter Fehler ist falsches Verhalten durch verschiedene Kombinationen von Undo/Redo und Änderungen anderer Anwender. Auf Client-Seite kann es hier zu einer falschen UI-Darstellung kommen. Dies passiert z.B. wenn eine Kante umgehangen wird, die alte Klasse von einem anderen Anwender gelöscht wird und dann das Umhängen rückgängig gemacht wird. Außerdem tritt das Problem auch bei Redo-Funktionen auf. In den meisten Fällen wird ein Redo angeboten, wobei nichts passiert oder Änderungen am Modell durchgeführt werden, da das dazugehörige Diagramm-Element von einem anderen Anwender gelöscht wurde. Die Problematik wurde eine Woche vor Abschluss der Implementierung entdeckt. Beim Versuch der Korrektur hatte sich herausgestellt, dass dabei noch sehr viele weitere Randfälle betrachtet werden müssten, sodass nur die wesentlichen Undo-Fälle behoben werden konnten.

20.7 Treeview Problem

Das Problem ist, dass in der TreeView unter dem Diagramm-Knoten Modellelemente angezeigt werden, die in dem Diagramm keine Graphische-Repräsentation besitzen.

Das Problem tritt auf, wenn sich im Diagramm ein Knoten befindet, der im Modell eine Kante zu anderen Knoten hat, diese Kante aber nicht in dem Diagramm vorhanden ist.

Das Problem wird durch das verschachtelte Aufbauen der TreeView erzeugt. Dabei wird immer von einem Element ausgegangen und geprüft, was seine direkten Kinder sind. Wenn er welche hat werden diese in der TreeView dargestellt. Kinder der tieferen Ebenen werden ausgehend von diesem Element nicht betrachtet.

Erläuterung an einem kleinen Beispiel: Wenn man von dem Diagramm ausgeht, kommt der Knoten rein, da dieser das einzige direkte Kind des Diagramms ist. Dieser hat auch eine Graphische-Repräsentation, ansonsten wäre er kein Kind des Diagramms. Als nächstes wird die TreeView weiter, ausgehen von dem Knoten, aufgebaut. Die Kinder des Knoten sind seine Kanten zu anderen Knoten, dabei wird aber nicht betrachtet, ob sich die Kanten in dem gleichen Diagramm befinden, welches das Root-Parent dieses Zweiges in der TreeView darstellt. Diese Überprüfung kann leider auch nicht ohne große Änderungen eingefügt werden.

Die TreeView wird mit einem ContentProvider aufgebaut. Dieser nimmt das Modell und erzeugt daraus die eigentliche TreeView mit den einzelnen TreeView-Elementen. Mit den TreeView-Elementen kann man in beide Richtungen navigieren. Man kann von jedem Element auf die Eltern und die Kinder zugreifen. Dieser kennt auch immer sein dazugehöriges ModellElement aus welchem er erzeugt wurde.

Während der Aufbau des ContentProviders ist der Zugriff auf die Eltern und die Kinder nicht möglich. Denn der ContenProvider merkt sich nicht welches das Vorgängerelement war. Somit kann von dem Knoten aus nicht sein TreeView Parent beim Aufbau der TreeView ermittelt werden. Es fehlt der Zugriff auf die TreeView-Elemente. Direkt auf dem Modell kann dies nicht bestimmt werden, da ein Knoten sich in mehreren Digrammen befinden kann.

Um dieses Problem zu lösen, müsste der Aufbau der TreeView unter den Diagramm-Knoten von den Modell-Repräsentationen auf Graphische-Repräsentationen umgeschrieben werden. Dabei würden dort bei Aufbau keine Knoten sondern BoundingBoxen vorliegen und anstatt Kanten GraphicalEdges. Diese haben eine 1 zu 1 Verbindung zum Diagramm und zum Modell-Element. Dadurch ist die benötigte Navigierbarkeit auch ohne die TreeView Elemente gegeben.

Da an den TreeView Elementen bereits Funktionalität anliegt, wie z.B. Klick-Listener, würde diese Änderung eine Reihe weiterer Anpassungen nach sich ziehen. Da dieser Fehler erst kurz vor dem finalen Codefreeze aufgefallen ist, konnte diese Änderung wegen Zeitmangel nicht mehr umgesetzt werden.

20.8 Modellkorruption beim SIT

Beim Schülerinformationstag haben Schüler an 4 Arbeitsrechnern das Produkt ausprobiert. Hierbei ist beim modellieren im Laufe des Tages zwei mal ein Fehler aufgetreten, welches eine vollständige Korruption des bearbeiteten Modells zur folge hatte. Der Server ist hierbei nicht abgestürzt, wodurch Arbeiten an anderen Modellen nicht gestört wurden.

Das korrupte Modell konnte nicht mehr bearbeitet werden und stellt somit einen schweren Datenverlust dar. Aus diesem Grund haben wir den Stacktrace analysiert und haben den Ausschnitt in Abbildung 20.1 bei beiden Vorkommnissen identifizieren können. Die Auslöser für den Fehler konnte nicht identifiziert werden und der Fehler konnte durch die Projektgruppe nicht reproduziert werden.

```
INFO de.unioldenburg.kotlett.impl.GMoVerSToKotelettServerAdapterImpl.
     getActiveDelta: Creating active delta for the given model ...
INFO de.unioldenburg.dol.services.patcher.impl.PatcherImpl.applyDelta:
     Please wait, applying operations to the model ...
INFO de.unioldenburg.dol.services.patcher.impl.PatcherImpl.applyDelta:
     Please wait, applying operations to the model ...
INFO de.unioldenburg.kotlett.impl.GMoVerSToKotelettServerAdapterImpl.
     saveVersion: Saving the model version into the repository ...
SEVERE de.unioldenburg.dol.config.impl.RepositoryFacadeImpl.loadDelta:
     Delta representation doesn't exist for repository/4/active.gmvs
INFO de.unioldenburg.dol.services.patcher.impl.PatcherImpl.applyDelta:
     Please wait, applying operations to the model ...
INFO de.unioldenburg.dol.services.calculator.impl.CalculatorImpl.
     calculateDelta: Fetching model elements ...
INFO de.unioldenburg.dol.services.calculator.impl.CalculatorImpl.
     calculateDelta: Calculating differences ...
INFO de.unioldenburg.dol.services.calculator.impl.CalculatorImpl.
     calculateDelta: Optimizing modeling delta ...
INFO de.unioldenburg.dol.config.impl.LogFileFacadeImpl.writeLogMessage:
     Writing the commit message to log file ...
INFO de.unioldenburg.kotlett.impl.GMoVerSToKotelettServerAdapterImpl.
     saveVersion: The model Version is successfully saved into the
     repository!
```

Abbildung 20.1: Stacktrace des Fehlers

Kapitel 21

Fazit

Die fehlgeschlagenen Tests in Kapitel 18 zeigen, dass das Produkt noch nicht für den produktiven Einsatz in der Softwareentwicklung bereit ist. Auch wurde die Usability des Produktes im Rahmen dieses Projektes nicht hoch priorisiert, wie in Abschnitt 1.3 beschrieben. Die erfolgreich durchgeführten Akzeptanztests in Kapitel 22 zeigen jedoch, dass die Hauptfunktionalität des Produktes gewährleistet wird. Zu Forschungszwecken ist das Projekt somit einsetzbar und erfüllt die in der Zielvereinbarung in Kapitel 1 definierten Kriterien abgesehen von dem Modellierungsumfang der StClock. Gerade als praktisches Anwendungsbeispiel der Delta Operation Language der Abteilung Softwaretechnik der Universität Oldenburg ist das Produkt geeignet. Anhand der LogView im Client und der Protokollierung auf dem Server ist es auch für Demonstrationen der DOLSA Synchronisation gut geeignet.

Kapitel 22

Erfolgreiche Akzeptanztests

Die Akzeptanztests in diesem Kapitel zielen auf eine vollständige Abdeckung der Benutzereingabe ab, um die Funktionalität des Systems auf UI-Ebene zu evaluieren und für die weitere Entwicklung zu dokumentieren. Die folgenden Akzeptanzstests wurden fehlerfrei zum Ende des Projektes am 18.03.2015 unter der Linux-Distribution Ubuntu 14.04.2 mit dem nativen Widget Toolkit GTK2 in der Version 2.24 durchgeführt.

Aufgrund der exzessiven Menge von Tests wurde auf das dokumentieren der Undo/Redo von allen Commands, sowie der Kreuzprodukte von aufeinander folgenden Aktionen verzichtet. Alle Undo/Redo-Operationen wurden getestet und der einzige fehlgeschlagene Test wurde in Kapitel 18 dokumentiert.

Wo keine Vorbedingungen definiert sind, wird von einem neuen, leeren Modell ausgegangen.

22.1 Tests für die Modellverwaltung

Die Tests in diesem Abschnitt decken den Funktionsumfang der Modellverwaltung ab.

22.1.1 Modellerstellung

Dieser Test deckt die Erstellung und das Öffnen eines neuen Modells ab.

Test

1. Oben im Menü den Menüeintrag „File“->„New Model“ anklicken.
2. Im Popup-Fenster den Namen des neuen Modells eintragen und auf „Ok“ klicken.
3. Oben im Menü den Menüeintrag „File“->„Open Model“ anklicken.

4. Das Listenelement des neuen Modells anklicken und auf „Ok“ klicken.

Erwartetes Ergebnis

1. Es ist das neu erstellte Modell geöffnet und links in der ModelTreeView angezeigt.
2. Es ist ein Editor des ersten Diagramms des neuen Modells geöffnet.

22.1.2 Modell öffnen

Dieser Test deckt das Öffnen eines Modells ab.

Test

1. Oben im Menü den Menüeintrag „File“->„Open Model“ anklicken.
2. Das Listenelement des neuen Modells anklicken und auf „Ok“ klicken.

Erwartetes Ergebnis

1. Es ist das geöffnete Modell links in der ModelTreeView angezeigt.
2. Es ist ein Editor des ersten Diagramms des geöffneten Modells geöffnet.

22.1.3 Modell schließen

Dieser Test deckt das Schließen eines Modells ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.

Test

1. Links in der TreeView auf das Modell (oberstes Element) rechtsklicken.
2. „Close Model“ anklicken.

Erwartetes Ergebnis

1. Die ModelTreeView links ist leer.
2. Es ist kein Editor geöffnet.

22.1.4 Modell löschen

Dieser Test deckt das Löschen eines Modells ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.

Test

1. Links in der TreeView auf das Modell (oberstes Element) rechtsklicken.
2. „Delete Model“ anklicken.
3. Im Popup das Löschen des Modells bestätigen.

Erwartetes Ergebnis

1. Die ModelTreeView links ist leer.
2. Es ist kein Editor geöffnet.
3. Es erscheint das Dialog zum Öffnen eines neuen Modells, indem das gelöschte Modell nicht enthalten ist.

22.1.5 Diagramm erstellen

Dieser Test deckt das Erstellen eines Diagramms ab.

Test

1. Links in der ModelTreeView rechtsklick auf das oberste Element (das Modell).
2. „Create new Diagram“ anklicken.
3. Im Popup-Fenster den Namen des neuen Diagramms eintragen und auf „Ok“ klicken.

Erwartetes Ergebnis

1. Links in der ModelTreeView wird das neue Diagramm unter Diagrams angezeigt.

22.1.6 Diagramm Umbenennen

Dieser Test deckt das Umbenennen eines Diagramms ab.

Test

1. Links in der ModelTreeView rechtsklick auf das umzubenennende Diagramm.
2. „Rename Diagram“ anklicken.
3. Im Popup-Fenster den neuen Namen des Diagramms eintragen und auf „Ok“ klicken.

Erwartetes Ergebnis

1. Links in der ModelTreeView wird der neue Name des Diagramms angezeigt.

22.2 Tests für SWT-GUI

Die Tests in diesem Abschnitt decken die SWT-GUI des Produktes ab.

22.2.1 Öffnen der LogView

Dieser Test deckt das Öffnen der LogView ab.

Test

1. Oben rechts im „Quick Access“ Textfeld „Log“ eingeben.
2. Enter drücken.

Erwartetes Ergebnis

1. Im unteren Bereich erscheint ein „Log“ Tab neben dem „Users“ Tab.

22.2.2 Aktualisierung der LogView

Dieser Test deckt das Aktualisierung der LogView ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.
2. Es wurde die LogView geöffnet.

Test

1. Eine Klasse erstellen.

Erwartetes Ergebnis

1. In der LogView unten erscheinen die Zeilen „begin send delta“, dann die versandtlen DOL Operationen und dann „end send delta“.

22.2.3 Hervorhebung des eigenen Namens

Dieser Test deckt die Hervorhebung des eigenen Namens in der Userliste ab.

Test

1. Ein Modell öffnen.

Erwartetes Ergebnis

1. Unten in der Userliste wird der eigene Nutzername hervorgehoben.

22.2.4 Name ändern

Dieser Test deckt das Ändern des eigenen Namens ab.

Test

1. Unten rechts auf den „Change Name“ Button klicken.
2. Im Popup-Fenster den neuen Nutzernamen eintragen und auf „Ok“ klicken.

Erwartetes Ergebnis

1. Unten in der Userliste wird der neue Nutzername angezeigt.

22.2.5 Farbe ändern

Dieser Test deckt das Ändern der eigenen Farbe ab.

Test

1. Unten rechts auf den „Change Color“ Button klicken.
2. Im Popup-Fenster die neue Farbe auswählen und bestätigen.

Erwartetes Ergebnis

1. Unten in der Userliste wird die neue Farbe angezeigt.
2. In allen Diagrammen werden die Elemente des Nutzers in der neuen Farbe angezeigt.

22.2.6 Farben ausblenden

Dieser Test deckt das temporäre Ausblenden der Farben im Diagrammeditor.

Vorbedingung

1. Es wurde eine Klasse erstellt.

Test

1. Unten rechts auf „Colors Visible“ klicken.

Erwartetes Ergebnis

1. In allen Diagrammen werden alle Elemente mit weißem Hintergrund angezeigt.

22.2.7 Farben wieder einblenden

Dieser Test deckt das Einblenden der Farben im Diagrammeditor.

Vorbedingung

1. Es wurde eine Klasse erstellt.
2. Die Farben im Diagramm wurden temporär ausgeblendet.

Test

1. Unten rechts auf „Colors Visible“ klicken.

Erwartetes Ergebnis

1. In allen Diagrammen werden alle Elemente mit der Farbe des Anwenders als Hintergrund angezeigt.

22.2.8 Farben im Modell permanent löschen.

Dieser Test deckt das Löschen aller Farben aus dem Modell ab.

Vorbedingung

1. Es wurde eine Klasse erstellt.

Test

1. Unten rechts auf „Clear Colors“ klicken.

Erwartetes Ergebnis

1. In allen Diagrammen werden alle Elemente mit weißem Hintergrund angezeigt.

22.2.9 Diagramm Drucken.

Dieser Test deckt das Drucken eines Diagramms ab.

Vorbedingung

1. Es wurde eine Klasse erstellt.

Test

1. Im Editor-Bereich in der Zeichenfläche rechtsklicken.
2. „Print Diagram“ anklicken.

Erwartetes Ergebnis

1. Es wird das Drucken-Dialog des Betriebssystems aufgerufen.

22.3 Tests für die Versionierung

Die Tests in diesem Abschnitt decken den Funktionsumfang der Versionierung ab.

22.3.1 Anlegen und Laden einer Version

Dieser Test deckt das Anlegen und das Laden einer Version ab.

Test

1. Oben im Menü den Menüeintrag „File“->„Save Version“ anklicken.
2. Im Popup-Fenster den Namen der neuen Version eintragen und auf „Ok“ klicken.
3. Links über der ModelTreeView auf die ComboBox klicken
4. Die neue Version anklicken

Erwartetes Ergebnis

1. Die neu angelegte Version wurde geladen.

22.4 Tests für Klassen

Die Tests in diesem Abschnitt decken das Erstellen, Bearbeiten und Löschen von Klassen ab.

22.4.1 Klasse erstellen

Dieser Test deckt das Erstellen einer Klasse ab.

Test

1. Im Editor-Bereich links auf „Class“ klicken.
2. Im Editor-Bereich in der Zeichenfläche klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klasse angezeigt.
2. Links in der ModelTreeView wird die neue Klasse im firstDiagram angezeigt.

22.4.2 Klasse erstellen mit Größenangabe

Dieser Test deckt das Erstellen einer Klasse mit Größenangabe ab.

Test

1. Im Editor-Bereich links auf „Class“ klicken.
2. Im Editor-Bereich in der Zeichenfläche klicken und nach unten rechts ziehen.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klasse mit angezeigt.
2. Die neue Klasse füllt den Bereich vom Drücken der Maustaste bis zum Loslassen der Maustaste.
3. Links in der ModelTreeView wird die neue Klasse im firstDiagram angezeigt.

22.4.3 Klasse auswählen

Dieser Test deckt das Auswählen einer Klasse ab.

Vorbedingung

1. Es wurde eine Klasse erstellt.

Test

1. Im Editor-Bereich die erstellte Klasse anklicken.

Erwartetes Ergebnis

1. Im Editor-Bereich wird die ausgewählte Klasse umrandet angezeigt.

22.4.4 Mehrfachauswahl von Klassen

Dieser Test deckt das Auswählen mehrerer Klassen ab.

Vorbedingung

1. Es wurden zwei Klassen erstellt.

Test

1. Im Editor-Bereich oben links von den beiden Klassen klicken und gedrückt halten.
2. Die Maus nach unten rechts ziehen bis die Auswahlanzeige beide Elemente komplett umfasst.
3. Maustaste loslassen.

Erwartetes Ergebnis

1. Im Editor-Bereich werden die beiden ausgewählte Klasse umrandet angezeigt.

22.4.5 Klasse umbenennen

Dieser Test deckt das Umbenennen einer Klasse ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich den Namen der ausgewählten Klasse anklicken.
2. Im Eingabefeld den neuen Namen „MyTestClass“ eingeben.
3. Im Editor-Bereich auf ein beliebigen Punkt klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird der neue Name der Klasse angezeigt.
2. Links in der ModelTreeView wird der neue Name der Klasse angezeigt.

22.4.6 Klasse umbenennen (Bestätigung mit Enter)

Dieser Test deckt das Umbenennen einer Klasse ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich den Namen der ausgewählten Klasse anklicken.
2. Im Eingabefeld den neuen Namen „MyTestClass“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird der neue Name der Klasse angezeigt.
2. Links in der ModelTreeView wird der neue Name der Klasse angezeigt.

22.4.7 Klasse verschieben

Dieser Test deckt das Verschieben einer Klasse ab.

Vorbedingung

1. Es wurde eine Klasse erstellt.

Test

1. Im Editor-Bereich die erstellte Klasse anklicken und gedrückt halten.
2. Den Mauszeiger innerhalb des Editor-Bereichs an die neue Position der Klasse bewegen.
3. Maustaste loslassen.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche befindet sich die verschobene Klasse an der neuen Position.

22.4.8 Klasse aus Diagramm löschen

Dieser Test deckt das Löschen einer Klasse aus einem Diagramm ab.

Vorbedingung

1. Es wurde eine Klasse erstellt.

Test

1. Im Editor-Bereich die erstellte Klasse anklicken.
2. Entfernen-Taste drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die gelöschte Klasse nicht mehr angezeigt.

22.4.9 Klasse aus Modell löschen

Dieser Test deckt das Löschen einer Klasse aus einem Modell ab.

Vorbedingung

1. Es wurde eine Klasse erstellt.

Test

1. Links in der ModelTreeView rechtsklick auf die Klasse unter 'Model-Elements'.
2. „Delete Class from model“ anklicken.

Erwartetes Ergebnis

1. Links in der ModelTreeView wird das gelöschte Element nicht mehr angezeigt.

22.4.10 Klasse aus Modell in Diagramm einfügen

Dieser Test deckt das Einfügen einer Klasse aus einem Modell in ein Diagramm ab.

Vorbedingung

1. Es wurde eine Klasse erstellt.
2. Die erstellte Klasse wurde aus dem Diagramm gelöscht.

Test

1. Links in der ModelTreeView auf die Klasse unter 'ModelElements' doppelklicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die eingefügte Klasse angezeigt.

22.4.11 Refactoring eines Klassennamens

Dieser Test deckt das Umbenennen einer Klasse ab, welche durch ein Klassenattribut referenziert wird. Hierbei muss das Klassenattribut seinen Typ korrekt anpassen.

Vorbedingung

1. Es wurde eine Klasse erstellt.
2. Diese Klasse wurde „myClass“ genannt.
3. Dieser Klasse wurde der Klassenattribut „myAttribute : myClass“ hinzugefügt.

Test

1. Die Klasse umbenennen zu „refactoredClass“.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das Klassenattribut mit dem Typ „refactoredClass“ angezeigt.
2. Links in der ModelTreeView wird das Klassenattribut mit dem Typ „refactoredClass“ in der Klasse angezeigt.

22.5 Tests für Klassenattribute

Die Tests in diesem Abschnitt decken das Erstellen, Bearbeiten und Löschen von Klassenattributen ab.

22.5.1 Klassenattribut mit Void Primitiv-Typ

Dieser Test deckt das Erstellen eines Klassenattributes vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Klassennamen klicken.
2. Im Texteingabefeld „hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Klassenattribut in der Klasse angezeigt.

22.5.2 Klassenattribut mit Integer Primitiv-Typ

Dieser Test deckt das Erstellen eines Klassenattributes vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Klassennamen klicken.
2. Im Texteingabefeld „hello : Integer“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Klassenattribut in der Klasse angezeigt.

22.5.3 Klassenattribut mit String Primitiv-Typ

Dieser Test deckt das Erstellen eines Klassenattributes vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Klassennamen klicken.
2. Im Texteingabefeld „hello : String“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Klassenattribut in der Klasse angezeigt.

22.5.4 Klassenattribut mit Boolean Primitiv-Typ

Dieser Test deckt das Erstellen eines Klassenattributes vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Klassennamen klicken.
2. Im Texteingabefeld „hello : Boolean“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Klassenattribut in der Klasse angezeigt.

22.5.5 Klassenattribut mit Float Primitiv-Typ

Dieser Test deckt das Erstellen eines Klassenattributes vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Klassennamen klicken.
2. Im Texteingabefeld „hello : Float“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Klassenattribut in der Klasse angezeigt.

22.5.6 Klassenattribut mit UnlimitedNatural Primitiv-Typ

Dieser Test deckt das Erstellen eines Klassenattributes vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Klassennamen klicken.
2. Im Texteingabefeld „hello : UnlimitedNatural“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Klassenattribut in der Klasse angezeigt.

22.5.7 Klassenattribut mit default Sichtbarkeit

Dieser Test deckt das Erstellen eines Klassenattributes mit default Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Klassennamen klicken.
2. Im Texteingabefeld „hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Klassenattribut mit protected Sichtbarkeit in der Klasse angezeigt.

22.5.8 Klassenattribut mit public Sichtbarkeit

Dieser Test deckt das Erstellen eines Klassenattributes mit public Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Klassennamen klicken.
2. Im Texteingabefeld „+hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Klassenattribut in der Klasse angezeigt.

22.5.9 Klassenattribut mit protected Sichtbarkeit

Dieser Test deckt das Erstellen eines Klassenattributes mit protected Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Klassennamen klicken.
2. Im Texteingabefeld „ hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Klassenattribut in der Klasse angezeigt.

22.5.10 Klassenattribut mit package-protection Sichtbarkeit

Dieser Test deckt das Erstellen eines Klassenattributes mit package-protection Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Klassennamen klicken.
2. Im Texteingabefeld „#hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Klassenattribut in der Klasse angezeigt.

22.5.11 Klassenattribut mit privater Sichtbarkeit

Dieser Test deckt das Erstellen eines Klassenattributes mit privater Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Klassennamen klicken.
2. Im Texteingabefeld „-hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Klassenattribut in der Klasse angezeigt.

22.5.12 Klassenattribut mit default Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Klassenattributes mit default Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Klassennamen klicken.
2. Im Texteingabefeld „ hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Klassenattribut mit protected Sichtbarkeit in der Klasse angezeigt.

22.5.13 Klassenattribut mit public Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Klassenattributes mit public Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Klassennamen klicken.
2. Im Texteingabefeld „+ hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Klassenattribut in der Klasse angezeigt.

22.5.14 Klassenattribut mit protected Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Klassenattributes mit protected Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Klassennamen klicken.
2. Im Texteingabefeld „ hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Klassenattribut angezeigt mit Leerzeichen.

2. Links in der ModelTreeView wird das neue Klassenattribut in der Klasse angezeigt mit Leerzeichen.

22.5.15 Klassenattribut mit package-protection Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Klassenattributes mit package-protection Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Klassennamen klicken.

2. Im Texteingabefeld „# hello : Void“ eingeben.

3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Klassenattribut angezeigt.

2. Links in der ModelTreeView wird das neue Klassenattribut in der Klasse angezeigt.

22.5.16 Klassenattribut mit privater Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Klassenattributes mit privater Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Klassennamen klicken.
2. Im Texteingabefeld „- hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Klassenattribut in der Klasse angezeigt.

22.5.17 Klassenattribut mit Datentyp aus dem Modell

Dieser Test deckt das Erstellen eines Klassenattributes mit Datentyp aus dem Modell ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.
2. Der Name dieser Klasse lautet „DefaultClass“

Test

1. Im Editor-Bereich im Bereich unter dem Klassennamen klicken.
2. Im Texteingabefeld „hello : DefaultClass“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Klassenattribut in der Klasse angezeigt.

22.5.18 Klassenattribut mit ungültigem Datentyp

Dieser Test deckt das Erstellen eines Klassenattributes mit ungültigem Datentyp ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.
2. Der Name dieser Klasse lautet „DefaultClass“

Test

1. Im Editor-Bereich im Bereich unter dem Klassennamen klicken.
2. Im Texteingabefeld „hello : NonExistantClass“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Eine Fehlermeldung erscheint mit der Meldung „The current model does not contain a type with the requested name 'NonExistantClass'.“.

22.5.19 Klassenattribut löschen

Dieser Test deckt das Löschen eines Klassenattributes ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Klassennamen klicken.
2. Im Texteingabefeld „- hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Klassenattribut in der Klasse angezeigt.

22.6 Tests für Klassenmethoden

Die Tests in diesem Abschnitt decken das Erstellen, Bearbeiten und Löschen von Klassenmethoden ab.

22.6.1 Klassenmethode mit Void Primitiv-Typ

Dieser Test deckt das Erstellen eines Klassenmethode vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.
2. Die erstellte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.
2. Im Texteingabefeld „hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Klassenmethode in der Klasse angezeigt.

22.6.2 Klassenmethode mit Integer Primitiv-Typ

Dieser Test deckt das Erstellen eines Klassenmethode vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.
2. Die erstellte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.
2. Im Texteingabefeld „hello() : Integer“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Klassenmethode in der Klasse angezeigt.

22.6.3 Klassenmethode mit String Primitiv-Typ

Dieser Test deckt das Erstellen eines Klassenmethode vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.
2. Die erstellte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.
2. Im Texteingabefeld „hello() : String“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Klassenmethode in der Klasse angezeigt.

22.6.4 Klassenmethode mit Boolean Primitiv-Typ

Dieser Test deckt das Erstellen eines Klassenmethode vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.
2. Die erstellte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.
2. Im Texteingabefeld „hello() : Boolean“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Klassenmethode in der Klasse angezeigt.

22.6.5 Klassenmethode mit Float Primitiv-Typ

Dieser Test deckt das Erstellen eines Klassenmethode vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.
2. Die erstellte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.
2. Im Texteingabefeld „hello() : Float“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Klassenmethode in der Klasse angezeigt.

22.6.6 Klassenmethode mit UnlimitedNatural Primitiv-Typ

Dieser Test deckt das Erstellen eines Klassenmethode vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.
2. Die erstellte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.
2. Im Texteingabefeld „hello() : UnlimitedNatural“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Klassenmethode in der Klasse angezeigt.

22.6.7 Klassenmethode mit default Sichtbarkeit

Dieser Test deckt das Erstellen eines Klassenmethode mit default Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.
2. Die erstellte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.
2. Im Texteingabefeld „hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Klassenmethode mit protected Sichtbarkeit in der Klasse angezeigt.

22.6.8 Klassenmethode mit public Sichtbarkeit

Dieser Test deckt das Erstellen eines Klassenmethode mit public Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.
2. Die erstellte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.
2. Im Texteingabefeld „+hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Klassenmethode in der Klasse angezeigt.

22.6.9 Klassenmethode mit protected Sichtbarkeit

Dieser Test deckt das Erstellen eines Klassenmethode mit protected Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.
2. Die erstellte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.
2. Im Texteingabefeld „ hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Klassenmethode in der Klasse angezeigt.

22.6.10 Klassenmethode mit package-protection Sichtbarkeit

Dieser Test deckt das Erstellen eines Klassenmethode mit package-protection Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.
2. Die erstellte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.
2. Im Texteingabefeld „#hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Klassenmethode in der Klasse angezeigt.

22.6.11 Klassenmethode mit privater Sichtbarkeit

Dieser Test deckt das Erstellen eines Klassenmethode mit privater Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.
2. Die erstellte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.
2. Im Texteingabefeld „-hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Klassenmethode in der Klasse angezeigt.

22.6.12 Klassenmethode mit default Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Klassenmethode mit default Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.
2. Die erstellte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.
2. Im Texteingabefeld „ hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Klassenmethode mit protected Sichtbarkeit in der Klasse angezeigt.

22.6.13 Klassenmethode mit public Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Klassenmethode mit public Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.

2. Die erstellte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.

2. Im Texteingabefeld „+ hello() : Void“ eingeben.

3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klassenmethode angezeigt.

2. Links in der ModelTreeView wird die neue Klassenmethode in der Klasse angezeigt.

22.6.14 Klassenmethode mit protected Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Klassenmethode mit protected Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.

2. Die erstellte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.

2. Im Texteingabefeld „ hello() : Void“ eingeben.

3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klassenmethode angezeigt mit Leerzeichen.
2. Links in der ModelTreeView wird die neue Klassenmethode in der Klasse angezeigt mit Leerzeichen.

22.6.15 Klassenmethode mit package-protection Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Klassenmethode mit package-protection Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.
2. Die erstellte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.
2. Im Texteingabefeld „# hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Klassenmethode in der Klasse angezeigt.

22.6.16 Klassenmethode mit privater Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Klassenmethode mit privater Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.
2. Die erstellte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.
2. Im Texteingabefeld „- hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Klassenmethode in der Klasse angezeigt.

22.6.17 Klassenmethode mit Datentyp aus dem Modell

Dieser Test deckt das Erstellen einer Klassenmethode mit Datentyp aus dem Modell ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.
2. Der Name dieser Klasse lautet „DefaultClass“
3. Die erstellte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.
2. Im Texteingabefeld „hello() : DefaultClass“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Klassenmethode in der Klasse angezeigt.

22.6.18 Klassenmethode mit ungültigem Datentyp

Dieser Test deckt das Erstellen einer Klassenmethode mit ungültigem Datentyp ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.
2. Die erstellte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.
2. Im Texteingabefeld „hello : NonExistantClass“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Eine Fehlermeldung erscheint mit der Meldung „The current model does not contain a type with the requested name 'NonExistantClass'.“.

22.6.19 Klassenmethode löschen

Dieser Test deckt das Löschen einer Klassenmethode ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.
2. Im Texteingabefeld „- hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die Klassenmethode nicht mehr angezeigt.
2. Links in der ModelTreeView wird die Klassenmethod in der Klasse nicht mehr angezeigt.

22.6.20 Klassenmethode mit Parameter vom Integer Primitiv-Typ

Dieser Test deckt das Erstellen eines Klassenmethode mit Parameter vom Integer Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.

2. Die erstellte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.

2. Im Texteingabefeld „hello(parameter : Integer) : Void“ eingeben.

3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klassenmethode angezeigt.

2. Links in der ModelTreeView wird die neue Klassenmethode in der Klasse angezeigt.

22.6.21 Klassenmethode mit Parameter vom String Primitiv-Typ

Dieser Test deckt das Erstellen eines Klassenmethode mit Parameter vom String Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.

2. Die erstellte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.

2. Im Texteingabefeld „hello(parameter : String) : Void“ eingeben.

3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Klassenmethode in der Klasse angezeigt.

22.6.22 Klassenmethode mit Parameter vom Boolean Primitiv-Typ

Dieser Test deckt das Erstellen eines Klassenmethode mit Parameter vom Boolean Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.
2. Die erstellte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.
2. Im Texteingabefeld „hello(parameter : Boolean) : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Klassenmethode in der Klasse angezeigt.

22.6.23 Klassenmethode mit Parameter vom Float Primitiv-Typ

Dieser Test deckt das Erstellen eines Klassenmethode mit Parameter vom Float Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.
2. Die erstellte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.
2. Im Texteingabefeld „hello(parameter : Float) : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Klassenmethode in der Klasse angezeigt.

22.6.24 Klassenmethode mit Parameter vom UnlimitedNatural Primitiv-Typ

Dieser Test deckt das Erstellen eines Klassenmethode mit Parameter vom UnlimitedNatural Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.
2. Die erstellte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.
2. Im Texteingabefeld „hello(parameter : UnlimitedNatural) : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Klassenmethode in der Klasse angezeigt.

22.6.25 Klassenmethode mit Parameter vom Datentyp aus dem Modell

Dieser Test deckt das Erstellen eines Klassenmethode mit Parameter vom Datentyp aus dem Modell ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.
2. Der Name der Klasse wurde zu „myClass“ geändert.
3. Die erstellte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.
2. Im Texteingabefeld „hello(parameter : myClass) : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Klassenmethode in der Klasse angezeigt.

22.6.26 Klassenmethode mit zwei Parametern

Dieser Test deckt das Erstellen eines Klassenmethode mit zwei Parametern ab.

Vorbedingung

1. Es wurde eine Klasse erstellt und ausgewählt.
2. Die erstellte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Klasse klicken.
2. Im Texteingabefeld „hello(parameter1 : Integer, parameter2 : Integer) : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Klassenmethode in der Klasse angezeigt.

22.7 Tests für Abstrakte Klassen

Die Tests in diesem Abschnitt decken das Erstellen, Bearbeiten und Löschen von Abstrakte Klassen ab.

22.7.1 Abstrakte Klasse erstellen

Dieser Test deckt das Erstellen einer Abstrakten Klasse ab.

Test

1. Im Editor-Bereich links auf „Class“ klicken.
2. Im Editor-Bereich in der Zeichenfläche klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klasse angezeigt.
2. Links in der ModelTreeView wird die neue Abstrakte Klasse im first-Diagram angezeigt.

22.7.2 Abstrakte Klasse erstellen mit Größenangabe

Dieser Test deckt das Erstellen einer Abstrakten Klasse mit Größenangabe ab.

Test

1. Im Editor-Bereich links auf „Class“ klicken.

2. Im Editor-Bereich in der Zeichenfläche klicken und nach unten rechts ziehen.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klasse mit angezeigt.

2. Die neue Abstrakte Klasse füllt den Bereich vom Drücken der Maustaste bis zum Loslassen der Maustaste.

3. Links in der ModelTreeView wird die neue Abstrakte Klasse im first-Diagram angezeigt.

22.7.3 Abstrakte Klasse auswählen

Dieser Test deckt das Auswählen einer Abstrakten Klasse ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt.

Test

1. Im Editor-Bereich die erstellte Abstrakte Klasse anklicken.

Erwartetes Ergebnis

1. Im Editor-Bereich wird die ausgewählte Abstrakte Klasse umrandet angezeigt.

22.7.4 Mehrfachauswahl von Abstrakte Klassen

Dieser Test deckt das Auswählen mehrerer Abstrakte Klassen ab.

Vorbedingung

1. Es wurden zwei Abstrakte Klassen erstellt.

Test

1. Im Editor-Bereich oben links von den beiden Abstrakte Klassen klicken und gedrückt halten.
2. Die Maus nach unten rechts ziehen bis die Auswahlanzeige beide Elemente komplett umfasst.
3. Maustaste loslassen.

Erwartetes Ergebnis

1. Im Editor-Bereich werden die beiden ausgewählte Abstrakte Klasse umrandet angezeigt.

22.7.5 Abstrakte Klasse umbenennen

Dieser Test deckt das Umbenennen einer Abstrakten Klasse ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich den Namen der ausgewählten Abstrakte Klasse anklicken.
2. Im Eingabefeld den neuen Namen „MyTestClass“ eingeben.
3. Im Editor-Bereich auf ein beliebigen Punkt klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird der neue Name der Abstrakte Klasse angezeigt.
2. Links in der ModelTreeView wird der neue Name der Abstrakte Klasse angezeigt.

22.7.6 Abstrakte Klasse umbenennen (Bestätigung mit Enter)

Dieser Test deckt das Umbenennen einer Abstrakten Klasse ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich den Namen der ausgewählten Abstrakte Klasse anklicken.
2. Im Eingabefeld den neuen Namen „MyTestClass“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird der neue Name der Abstrakte Klasse angezeigt.
2. Links in der ModelTreeView wird der neue Name der Abstrakte Klasse angezeigt.

22.7.7 Abstrakte Klasse verschieben

Dieser Test deckt das Verschieben einer Abstrakten Klasse ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt.

Test

1. Im Editor-Bereich die erstellte Abstrakte Klasse anklicken und gedrückt halten.
2. Den Mauszeiger innerhalb des Editor-Bereichs an die neue Position der Abstrakte Klasse bewegen.
3. Maustaste loslassen.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche befindet sich die verschobene Abstrakte Klasse an der neuen Position.

22.7.8 Abstrakte Klasse aus Diagramm löschen

Dieser Test deckt das Löschen einer Abstrakten Klasse aus einem Diagramm ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt.

Test

1. Im Editor-Bereich die erstellte Abstrakte Klasse anklicken.
2. Entfernen-Taste drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die gelöschte Abstrakte Klasse nicht mehr angezeigt.

22.7.9 Abstrakte Klasse aus Modell löschen

Dieser Test deckt das Löschen einer Abstrakten Klasse aus einem Modell ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt.

Test

1. Links in der ModelTreeView rechtsklick auf die Abstrakte Klasse unter 'ModelElements'.
2. „Delete Class from model“ anklicken.

Erwartetes Ergebnis

1. Links in der ModelTreeView wird das gelöschte Element nicht mehr angezeigt.

22.7.10 Abstrakte Klasse aus Modell in Diagramm einfügen

Dieser Test deckt das Einfügen einer Abstrakten Klasse aus einem Modell in ein Diagramm ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt.
2. Die erstellte Abstrakte Klasse wurde aus dem Diagramm gelöscht.

Test

1. Links in der ModelTreeView auf die Abstrakte Klasse unter 'Model-Elements' doppelklicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die eingefügte Abstrakte Klasse angezeigt.

22.7.11 Refactoring eines Abstrakte Klassennamens

Dieser Test deckt das Umbenennen einer Abstrakten Klasse ab, welche durch ein Abstrakte Klassenattribut referenziert wird. Hierbei muss das Abstrakte Klassenattribut seinen Typ korrekt anpassen.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt.
2. Diese Abstrakte Klasse wurde „myAbstractClass“ genannt.
3. Dieser Abstrakte Klasse wurde der Abstrakte Klassenattribut „myAttribute : myAbstractClass“ hinzugefügt.

Test

1. Die Abstrakte Klasse umbenennen zu „refactoredClass“.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das Abstrakte Klassenattribut mit dem Typ „refactoredClass“ angezeigt.
2. Links in der ModelTreeView wird das Abstrakte Klassenattribut mit dem Typ „refactoredClass“ in der Abstrakte Klasse angezeigt.

22.8 Tests für Abstrakte Klassenattribute

Die Tests in diesem Abschnitt decken das Erstellen, Bearbeiten und Löschen von Abstrakte Klassenattributen ab.

22.8.1 Abstrakte Klassenattribut mit Void Primitiv-Typ

Dieser Test deckt das Erstellen eines Abstrakte Klassenattributes vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Abstrakte Klassennamen klicken.
2. Im Texteingabefeld „hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Abstrakte Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Abstrakte Klassenattribut in der Abstrakten Klasse angezeigt.

22.8.2 Abstrakte Klassenattribut mit Integer Primitiv-Typ

Dieser Test deckt das Erstellen eines Abstrakte Klassenattributes vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Abstrakte Klassennamen klicken.
2. Im Texteingabefeld „hello : Integer“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Abstrakte Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Abstrakte Klassenattribut in der Abstrakten Klasse angezeigt.

22.8.3 Abstrakte Klassenattribut mit String Primitiv-Typ

Dieser Test deckt das Erstellen eines Abstrakte Klassenattributes vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Abstrakte Klassennamen klicken.
2. Im Texteingabefeld „hello : String“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Abstrakte Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Abstrakte Klassenattribut in der Abstrakten Klasse angezeigt.

22.8.4 Abstrakte Klassenattribut mit Boolean Primitiv-Typ

Dieser Test deckt das Erstellen eines Abstrakte Klassenattributes vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Abstrakte Klassennamen klicken.
2. Im Texteingabefeld „hello : Boolean“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Abstrakte Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Abstrakte Klassenattribut in der Abstrakten Klasse angezeigt.

22.8.5 Abstrakte Klassenattribut mit Float Primitiv-Typ

Dieser Test deckt das Erstellen eines Abstrakte Klassenattributes vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Abstrakte Klassennamen klicken.
2. Im Texteingabefeld „hello : Float“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Abstrakte Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Abstrakte Klassenattribut in der Abstrakten Klasse angezeigt.

22.8.6 Abstrakte Klassenattribut mit UnlimitedNatural Primitiv-Typ

Dieser Test deckt das Erstellen eines Abstrakte Klassenattributes vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Abstrakte Klassennamen klicken.
2. Im Texteingabefeld „hello : UnlimitedNatural“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Abstrakte Klassenattribut angezeigt.

2. Links in der ModelTreeView wird das neue Abstrakte Klassenattribut in der Abstrakten Klasse angezeigt.

22.8.7 Abstrakte Klassenattribut mit default Sichtbarkeit

Dieser Test deckt das Erstellen eines Abstrakte Klassenattributes mit default Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Abstrakte Klassennamen klicken.

2. Im Texteingabefeld „hello : Void“ eingeben.

3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Abstrakte Klassenattribut angezeigt.

2. Links in der ModelTreeView wird das neue Abstrakte Klassenattribut mit protected Sichtbarkeit in der Abstrakten Klasse angezeigt.

22.8.8 Abstrakte Klassenattribut mit public Sichtbarkeit

Dieser Test deckt das Erstellen eines Abstrakte Klassenattributes mit public Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Abstrakte Klassennamen klicken.
2. Im Texteingabefeld „+hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Abstrakte Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Abstrakte Klassenattribut in der Abstrakten Klasse angezeigt.

22.8.9 Abstrakte Klassenattribut mit protected Sichtbarkeit

Dieser Test deckt das Erstellen eines Abstrakte Klassenattributes mit protected Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Abstrakte Klassennamen klicken.
2. Im Texteingabefeld „ hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Abstrakte Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Abstrakte Klassenattribut in der Abstrakten Klasse angezeigt.

22.8.10 Abstrakte Klassenattribut mit package-protection Sichtbarkeit

Dieser Test deckt das Erstellen eines Abstrakte Klassenattributes mit package-protection Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Abstrakte Klassennamen klicken.
2. Im Texteingabefeld „#hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Abstrakte Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Abstrakte Klassenattribut in der Abstrakten Klasse angezeigt.

22.8.11 Abstrakte Klassenattribut mit privater Sichtbarkeit

Dieser Test deckt das Erstellen eines Abstrakte Klassenattributes mit privater Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Abstrakte Klassennamen klicken.
2. Im Texteingabefeld „-hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Abstrakte Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Abstrakte Klassenattribut in der Abstrakten Klasse angezeigt.

22.8.12 Abstrakte Klassenattribut mit default Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Abstrakte Klassenattributes mit default Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Abstrakte Klassennamen klicken.
2. Im Texteingabefeld „ hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Abstrakte Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Abstrakte Klassenattribut mit protected Sichtbarkeit in der Abstrakten Klasse angezeigt.

22.8.13 Abstrakte Klassenattribut mit public Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Abstrakte Klassenattributes mit public Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Abstrakte Klassennamen klicken.
2. Im Texteingabefeld „+ hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Abstrakte Klassenattribut angezeigt.

2. Links in der ModelTreeView wird das neue Abstrakte Klassenattribut in der Abstrakten Klasse angezeigt.

22.8.14 Abstrakte Klassenattribut mit protected Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Abstrakte Klassenattributes mit protected Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Abstrakte Klassennamen klicken.

2. Im Texteingabefeld „ hello : Void“ eingeben.

3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Abstrakte Klassenattribut angezeigt mit Leerzeichen.

2. Links in der ModelTreeView wird das neue Abstrakte Klassenattribut in der Abstrakten Klasse angezeigt mit Leerzeichen.

22.8.15 Abstrakte Klassenattribut mit package-protection Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Abstrakte Klassenattributes mit package-protection Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Abstrakte Klassennamen klicken.
2. Im Texteingabefeld „# hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Abstrakte Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Abstrakte Klassenattribut in der Abstrakten Klasse angezeigt.

22.8.16 Abstrakte Klassenattribut mit privater Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Abstrakte Klassenattributes mit privater Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Abstrakte Klassennamen klicken.
2. Im Texteingabefeld „- hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Abstrakte Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Abstrakte Klassenattribut in der Abstrakten Klasse angezeigt.

22.8.17 Abstrakte Klassenattribut mit Datentyp aus dem Modell

Dieser Test deckt das Erstellen eines Abstrakte Klassenattributes mit Datentyp aus dem Modell ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.
2. Der Name dieser Abstrakte Klasse lautet „DefaultClass“

Test

1. Im Editor-Bereich im Bereich unter dem Abstrakte Klassennamen klicken.
2. Im Texteingabefeld „hello : DefaultClass“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Abstrakte Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Abstrakte Klassenattribut in der Abstrakten Klasse angezeigt.

22.8.18 Abstrakte Klassenattribut mit ungültigem Datentyp

Dieser Test deckt das Erstellen eines Abstrakte Klassenattributes mit ungültigem Datentyp ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.
2. Der Name dieser Abstrakte Klasse lautet „DefaultClass“

Test

1. Im Editor-Bereich im Bereich unter dem Abstrakte Klassennamen klicken.
2. Im Texteingabefeld „hello : NonExistantClass“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Eine Fehlermeldung erscheint mit der Meldung „The current model does not contain a type with the requested name ’NonExistantClass’.“.

22.8.19 Abstrakte Klassenattribut löschen

Dieser Test deckt das Löschen eines Abstrakte Klassenattributes ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Abstrakte Klassennamen klicken.
2. Im Texteingabefeld „- hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Abstrakte Klassenattribut angezeigt.
2. Links in der ModelTreeView wird das neue Abstrakte Klassenattribut in der Abstrakten Klasse angezeigt.

22.9 Tests für Abstrakte Klassenmethoden

Die Tests in diesem Abschnitt decken das Erstellen, Bearbeiten und Löschen von Abstrakte Klassenmethoden ab.

22.9.1 Abstrakte Klassenmethode mit Void Primitiv-Typ

Dieser Test deckt das Erstellen eines Abstrakte Klassenmethode vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.
2. Die erstellte Abstrakte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.
2. Im Texteingabefeld „hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klassenmethode angezeigt.

2. Links in der ModelTreeView wird die neue Abstrakte Klassenmethode in der Abstrakten Klasse angezeigt.

22.9.2 Abstrakte Klassenmethode mit Integer Primitiv-Typ

Dieser Test deckt das Erstellen eines Abstrakte Klassenmethode vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

2. Die erstellte Abstrakte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.

2. Im Texteingabefeld „hello() : Integer“ eingeben.

3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klassenmethode angezeigt.

2. Links in der ModelTreeView wird die neue Abstrakte Klassenmethode in der Abstrakten Klasse angezeigt.

22.9.3 Abstrakte Klassenmethode mit String Primitiv-Typ

Dieser Test deckt das Erstellen eines Abstrakte Klassenmethode vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

2. Die erstellte Abstrakte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.
2. Im Texteingabefeld „hello() : String“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Abstrakte Klassenmethode in der Abstrakten Klasse angezeigt.

22.9.4 Abstrakte Klassenmethode mit Boolean Primitiv-Typ

Dieser Test deckt das Erstellen eines Abstrakte Klassenmethode vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.
2. Die erstellte Abstrakte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.
2. Im Texteingabefeld „hello() : Boolean“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Abstrakte Klassenmethode in der Abstrakten Klasse angezeigt.

22.9.5 Abstrakte Klassenmethode mit Float Primitiv-Typ

Dieser Test deckt das Erstellen eines Abstrakte Klassenmethode vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

2. Die erstellte Abstrakte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.

2. Im Texteingabefeld „hello() : Float“ eingeben.

3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klassenmethode angezeigt.

2. Links in der ModelTreeView wird die neue Abstrakte Klassenmethode in der Abstrakten Klasse angezeigt.

22.9.6 Abstrakte Klassenmethode mit UnlimitedNatural Primitiv-Typ

Dieser Test deckt das Erstellen eines Abstrakte Klassenmethode vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

2. Die erstellte Abstrakte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.

2. Im Texteingabefeld „hello() : UnlimitedNatural“ eingeben.

3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klassenmethode angezeigt.

2. Links in der ModelTreeView wird die neue Abstrakte Klassenmethode in der Abstrakten Klasse angezeigt.

22.9.7 Abstrakte Klassenmethode mit default Sichtbarkeit

Dieser Test deckt das Erstellen eines Abstrakte Klassenmethode mit default Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

2. Die erstellte Abstrakte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.

2. Im Texteingabefeld „hello() : Void“ eingeben.

3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klassenmethode angezeigt.

2. Links in der ModelTreeView wird die neue Abstrakte Klassenmethode mit protected Sichtbarkeit in der Abstrakten Klasse angezeigt.

22.9.8 Abstrakte Klassenmethode mit public Sichtbarkeit

Dieser Test deckt das Erstellen eines Abstrakte Klassenmethode mit public Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

2. Die erstellte Abstrakte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.
2. Im Texteingabefeld „+hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Abstrakte Klassenmethode in der Abstrakten Klasse angezeigt.

22.9.9 Abstrakte Klassenmethode mit protected Sichtbarkeit

Dieser Test deckt das Erstellen eines Abstrakte Klassenmethode mit protected Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.
2. Die erstellte Abstrakte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.
2. Im Texteingabefeld „ hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Abstrakte Klassenmethode in der Abstrakten Klasse angezeigt.

22.9.10 Abstrakte Klassenmethode mit package-protection Sichtbarkeit

Dieser Test deckt das Erstellen eines Abstrakte Klassenmethode mit package-protection Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.
2. Die erstellte Abstrakte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.
2. Im Texteingabefeld „#hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Abstrakte Klassenmethode in der Abstrakten Klasse angezeigt.

22.9.11 Abstrakte Klassenmethode mit privater Sichtbarkeit

Dieser Test deckt das Erstellen eines Abstrakte Klassenmethode mit privater Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.
2. Die erstellte Abstrakte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.
2. Im Texteingabefeld „-hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Abstrakte Klassenmethode in der Abstrakten Klasse angezeigt.

22.9.12 Abstrakte Klassenmethode mit default Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Abstrakte Klassenmethode mit default Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.
2. Die erstellte Abstrakte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.
2. Im Texteingabefeld „ hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Abstrakte Klassenmethode mit protected Sichtbarkeit in der Abstrakten Klasse angezeigt.

22.9.13 Abstrakte Klassenmethode mit public Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Abstrakte Klassenmethode mit public Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.
2. Die erstellte Abstrakte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.
2. Im Texteingabefeld „+ hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Abstrakte Klassenmethode in der Abstrakten Klasse angezeigt.

22.9.14 Abstrakte Klassenmethode mit protected Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Abstrakte Klassenmethode mit protected Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.
2. Die erstellte Abstrakte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.
2. Im Texteingabefeld „ hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klassenmethode angezeigt mit Leerzeichen.
2. Links in der ModelTreeView wird die neue Abstrakte Klassenmethode in der Abstrakten Klasse angezeigt mit Leerzeichen.

22.9.15 Abstrakte Klassenmethode mit package-protection Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Abstrakte Klassenmethode mit package-protection Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.
2. Die erstellte Abstrakte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.
2. Im Texteingabefeld „# hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Abstrakte Klassenmethode in der Abstrakten Klasse angezeigt.

22.9.16 Abstrakte Klassenmethode mit privater Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Abstrakte Klassenmethode mit privater Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.
2. Die erstellte Abstrakte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.
2. Im Texteingabefeld „- hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Abstrakte Klassenmethode in der Abstrakten Klasse angezeigt.

22.9.17 Abstrakte Klassenmethode mit Datentyp aus dem Modell

Dieser Test deckt das Erstellen einer Abstrakte Klassenmethode mit Datentyp aus dem Modell ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.
2. Der Name dieser Abstrakte Klasse lautet „DefaultClass“
3. Die erstellte Abstrakte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.
2. Im Texteingabefeld „hello() : DefaultClass“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Abstrakte Klassenmethode in der Abstrakten Klasse angezeigt.

22.9.18 Abstrakte Klassenmethode mit ungültigem Datentyp

Dieser Test deckt das Erstellen einer Abstrakte Klassenmethode mit ungültigem Datentyp ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.
2. Die erstellte Abstrakte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.
2. Im Texteingabefeld „hello : NonExistantClass“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Eine Fehlermeldung erscheint mit der Meldung „The current model does not contain a type with the requested name ’NonExistantClass’.“.

22.9.19 Abstrakte Klassenmethode löschen

Dieser Test deckt das Löschen einer Abstrakte Klassenmethode ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.
2. Im Texteingabefeld „- hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die Abstrakte Klassenmethode nicht mehr angezeigt.
2. Links in der ModelTreeView wird die Abstrakte Klassenmethod in der Abstrakten Klasse nicht mehr angezeigt.

22.9.20 Abstrakte Klassenmethode mit Parameter vom Integer Primitiv-Typ

Dieser Test deckt das Erstellen eines Abstrakte Klassenmethode mit Parameter vom Integer Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.
2. Die erstellte Abstrakte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.
2. Im Texteingabefeld „hello(parameter : Integer) : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Abstrakte Klassenmethode in der Abstrakten Klasse angezeigt.

22.9.21 Abstrakte Klassenmethode mit Parameter vom String Primitiv-Typ

Dieser Test deckt das Erstellen eines Abstrakte Klassenmethode mit Parameter vom String Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.
2. Die erstellte Abstrakte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.
2. Im Texteingabefeld „hello(parameter : String) : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Abstrakte Klassenmethode in der Abstrakten Klasse angezeigt.

22.9.22 Abstrakte Klassenmethode mit Parameter vom Boolean Primitiv-Typ

Dieser Test deckt das Erstellen eines Abstrakte Klassenmethode mit Parameter vom Boolean Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.
2. Die erstellte Abstrakte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.
2. Im Texteingabefeld „hello(parameter : Boolean) : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Abstrakte Klassenmethode in der Abstrakten Klasse angezeigt.

22.9.23 Abstrakte Klassenmethode mit Parameter vom Float Primitiv-Typ

Dieser Test deckt das Erstellen eines Abstrakte Klassenmethode mit Parameter vom Float Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.
2. Die erstellte Abstrakte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.
2. Im Texteingabefeld „hello(parameter : Float) : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Abstrakte Klassenmethode in der Abstrakten Klasse angezeigt.

22.9.24 Abstrakte Klassenmethode mit Parameter vom UnlimitedNatural Primitiv-Typ

Dieser Test deckt das Erstellen eines Abstrakte Klassenmethode mit Parameter vom UnlimitedNatural Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.
2. Die erstellte Abstrakte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.
2. Im Texteingabefeld „hello(parameter : UnlimitedNatural) : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klassenmethode angezeigt.
2. Links in der ModelTreeView wird die neue Abstrakte Klassenmethode in der Abstrakten Klasse angezeigt.

22.9.25 Abstrakte Klassenmethode mit Parameter vom Datentyp aus dem Modell

Dieser Test deckt das Erstellen eines Abstrakte Klassenmethode mit Parameter vom Datentyp aus dem Modell ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.
2. Der Name der Abstrakte Klasse wurde zu „myAbstractClass“ geändert.
3. Die erstellte Abstrakte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.
2. Im Texteingabefeld „hello(parameter : myAbstractClass) : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klassenmethode angezeigt.

2. Links in der ModelTreeView wird die neue Abstrakte Klassenmethode in der Abstrakten Klasse angezeigt.

22.9.26 Abstrakte Klassenmethode mit zwei Parametern

Dieser Test deckt das Erstellen eines Abstrakte Klassenmethode mit zwei Parametern ab.

Vorbedingung

1. Es wurde eine Abstrakte Klasse erstellt und ausgewählt.

2. Die erstellte Abstrakte Klasse wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich der Abstrakte Klasse klicken.

2. Im Texteingabefeld „hello(parameter1 : Integer, parameter2 : Integer) : Void“ eingeben.

3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Abstrakte Klassenmethode angezeigt.

2. Links in der ModelTreeView wird die neue Abstrakte Klassenmethode in der Abstrakten Klasse angezeigt.

22.10 Tests für Interfaces

Die Tests in diesem Abschnitt decken das Erstellen, Bearbeiten und Löschen von Interfaces ab.

22.10.1 Interface erstellen

Dieser Test deckt das Erstellen eines Interface ab.

Test

1. Im Editor-Bereich links auf „Interface“ klicken.
2. Im Editor-Bereich in der Zeichenfläche klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Interface angezeigt.
2. Links in der ModelTreeView wird das neue Interface im firstDiagram angezeigt.

22.10.2 Interface erstellen mit Größenangabe

Dieser Test deckt das Erstellen eines Interface mit Größenangabe ab.

Test

1. Im Editor-Bereich links auf „Interface“ klicken.
2. Im Editor-Bereich in der Zeichenfläche klicken und nach unten rechts ziehen.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Interface mit angezeigt.
2. Das neue Interface füllt den Bereich vom Drücken der Maustaste bis zum Loslassen der Maustaste.
3. Links in der ModelTreeView wird das neue Interface im firstDiagram angezeigt.

22.10.3 Interface auswählen

Dieser Test deckt das Auswählen eines Interface ab.

Vorbedingung

1. Es wurde ein Interface erstellt.

Test

1. Im Editor-Bereich das erstellte Interface anklicken.

Erwartetes Ergebnis

1. Im Editor-Bereich wird das erstellte Interface umrandet angezeigt.

22.10.4 Mehrfachauswahl von Interfaces

Dieser Test deckt das Auswählen mehrerer Interfaces ab.

Vorbedingung

1. Es wurden zwei Interfaces erstellt.

Test

1. Im Editor-Bereich oben links von den beiden Interfaces klicken und gedrückt halten.
2. Die Maus nach unten rechts ziehen bis die Auswahlanzeige beide Elemente komplett umfasst.
3. Maustaste loslassen.

Erwartetes Ergebnis

1. Im Editor-Bereich werden die beiden ausgewählten Interfaces umrandet angezeigt.

22.10.5 Interface umbenennen

Dieser Test deckt das Umbenennen eines Interfaces ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.

Test

1. Im Editor-Bereich den Namen des ausgewählten Interfaces anklicken.
2. Im Eingabefeld den neuen Namen „MyTestInterface“ eingeben.
3. Im Editor-Bereich auf ein beliebigen Punkt klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird der neue Name des Interfaces angezeigt.
2. Links in der ModelTreeView wird der neue Name des Interfaces angezeigt.

22.10.6 Interface umbenennen (Bestätigung mit Enter)

Dieser Test deckt das Umbenennen eines Interfaces ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.

Test

1. Im Editor-Bereich den Namen des ausgewählten Interfaces anklicken.
2. Im Eingabefeld den neuen Namen „MyTestInterfaces“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird der neue Name des Interfaces angezeigt.
2. Links in der ModelTreeView wird der neue Name des Interfaces angezeigt.

22.10.7 Interface verschieben

Dieser Test deckt das Verschieben eines Interfaces ab.

Vorbedingung

1. Es wurde ein Interface erstellt.

Test

1. Im Editor-Bereich das erstellte Interface anklicken und gedrückt halten.
2. Den Mauszeiger innerhalb des Editor-Bereichs an die neue Position des Interfaces bewegen.
3. Maustaste loslassen.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche befindet sich das verschobene Interface an der neuen Position.

22.10.8 Interface aus Diagramm löschen

Dieser Test deckt das Löschen eines Interfaces aus einem Diagramm ab.

Vorbedingung

1. Es wurde ein Interface erstellt.

Test

1. Im Editor-Bereich das erstellte Interface anklicken.
2. Entfernen-Taste drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das gelöschte Interface nicht mehr angezeigt.

22.10.9 Interface aus Modell löschen

Dieser Test deckt das Löschen eines Interfaces aus einem Modell ab.

Vorbedingung

1. Es wurde ein Interface erstellt.

Test

1. Links in der ModelTreeView rechtsklick auf das Interface unter 'ModelElements'.
2. „Delete Interface from model“ anklicken.

Erwartetes Ergebnis

1. Links in der ModelTreeView wird das gelöschte Element nicht mehr angezeigt.

22.10.10 Interface aus Modell in Diagramm einfügen

Dieser Test deckt das Einfügen eines Interfaces aus einem Modell in ein Diagramm ab.

Vorbedingung

1. Es wurde ein Interface erstellt.
2. Das erstellte Interface wurde aus dem Diagramm gelöscht.

Test

1. Links in der ModelTreeView auf das Interface unter 'ModelElements' doppelklicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das eingefügte Interface angezeigt.

22.10.11 Refactoring eines Interfacenamens

Dieser Test deckt das Umbenennen eines Interfaces ab, welche durch ein Interfaceattribut referenziert wird. Hierbei muss das Interfaceattribut seinen Typ korrekt anpassen.

Vorbedingung

1. Es wurde ein Interface erstellt.
2. Dieses Interface wurde „myInterface“ genannt.
3. Diesem Interface wurde das Interfaceattribut „myAttribute : myInterface“ hinzugefügt.

Test

1. Das Interface umbenennen zu „refactoredInterface“.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das Interfaceattribut mit dem Typ „refactoredInterface“ angezeigt.
2. Links in der ModelTreeView wird das Interfaceattribut mit dem Typ „refactoredInterface“ im Interface angezeigt.

22.11 Tests für Interface-Attribute

Die Tests in diesem Abschnitt decken das Erstellen, Bearbeiten und Löschen von Interface-Attributen ab.

22.11.1 Interface-Attribut mit Void Primitiv-Typ

Dieser Test deckt das Erstellen eines Interface-Attributes vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Interface erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Interfacenamen klicken.
2. Im Texteingabefeld „hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Interface-Attribut angezeigt.
2. Links in der ModelTreeView wird das neue Interface-Attribut im Interface angezeigt.

22.11.2 Interface-Attribut mit Integer Primitiv-Typ

Dieser Test deckt das Erstellen eines Interface-Attributes vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Interface erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Interfacenamen klicken.
2. Im Texteingabefeld „hello : Integer“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Interface-Attribut angezeigt.
2. Links in der ModelTreeView wird das neue Interface-Attribut im Interface angezeigt.

22.11.3 Interface-Attribut mit String Primitiv-Typ

Dieser Test deckt das Erstellen eines Interface-Attributes vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Interface erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Interfacenamen klicken.
2. Im Texteingabefeld „hello : String“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Interface-Attribut angezeigt.
2. Links in der ModelTreeView wird das neue Interface-Attribut im Interface angezeigt.

22.11.4 Interface-Attribut mit Boolean Primitiv-Typ

Dieser Test deckt das Erstellen eines Interface-Attributes vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Interface erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Interfacenamen klicken.
2. Im Texteingabefeld „hello : Boolean“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Interface-Attribut angezeigt.
2. Links in der ModelTreeView wird das neue Interface-Attribut im Interface angezeigt.

22.11.5 Interface-Attribut mit Float Primitiv-Typ

Dieser Test deckt das Erstellen eines Interface-Attributes vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Interface erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Interfacenamen klicken.
2. Im Texteingabefeld „hello : Float“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Interface-Attribut angezeigt.
2. Links in der ModelTreeView wird das neue Interface-Attribut im Interface angezeigt.

22.11.6 Interface-Attribut mit UnlimitedNatural Primitiv-Typ

Dieser Test deckt das Erstellen eines Interface-Attributes vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde eine Interface erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Interfacenamen klicken.
2. Im Texteingabefeld „hello : UnlimitedNatural“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Interface-Attribut angezeigt.
2. Links in der ModelTreeView wird das neue Interface-Attribut im Interface angezeigt.

22.11.7 Interface-Attribut mit default Sichtbarkeit

Dieser Test deckt das Erstellen eines Interface-Attributes mit default Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Interface erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Interfacenamen klicken.
2. Im Texteingabefeld „hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Interface-Attribut angezeigt.
2. Links in der ModelTreeView wird das neue Interface-Attribut mit protected Sichtbarkeit im Interface angezeigt.

22.11.8 Interface-Attribut mit public Sichtbarkeit

Dieser Test deckt das Erstellen eines Interface-Attributes mit public Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Interface erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Interfacenamen klicken.
2. Im Texteingabefeld „+hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Interface-Attribut angezeigt.
2. Links in der ModelTreeView wird das neue Interface-Attribut im Interface angezeigt.

22.11.9 Interface-Attribut mit protected Sichtbarkeit

Dieser Test deckt das Erstellen eines Interface-Attributes mit protected Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Interface erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Interfacenamen klicken.
2. Im Texteingabefeld „ hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Interface-Attribut angezeigt.
2. Links in der ModelTreeView wird das neue Interface-Attribut im Interface angezeigt.

22.11.10 Interface-Attribut mit package-protection Sichtbarkeit

Dieser Test deckt das Erstellen eines Interface-Attributes mit package-protection Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Interface erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Interfacenamen klicken.
2. Im Texteingabefeld „#hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Interface-Attribut angezeigt.
2. Links in der ModelTreeView wird das neue Interface-Attribut im Interface angezeigt.

22.11.11 Interface-Attribut mit privater Sichtbarkeit

Dieser Test deckt das Erstellen eines Interface-Attributes mit privater Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Interface erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Interfacenamen klicken.
2. Im Texteingabefeld „-hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Interface-Attribut angezeigt.
2. Links in der ModelTreeView wird das neue Interface-Attribut im Interface angezeigt.

22.11.12 Interface-Attribut mit default Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Interface-Attributes mit default Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde eine Interface erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Interfacenamen klicken.
2. Im Texteingabefeld „ hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Interface-Attribut angezeigt.
2. Links in der ModelTreeView wird das neue Interface-Attribut mit protected Sichtbarkeit im Interface angezeigt.

22.11.13 Interface-Attribut mit public Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Interface-Attributes mit public Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde eine Interface erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Interfacenamen klicken.
2. Im Texteingabefeld „+ hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Interface-Attribut angezeigt.
2. Links in der ModelTreeView wird das neue Interface-Attribut im Interface angezeigt.

22.11.14 Interface-Attribut mit protected Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Interface-Attributes mit protected Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde eine Interface erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Interfacenamen klicken.
2. Im Texteingabefeld „ hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Interface-Attribut angezeigt mit Leerzeichen.

2. Links in der ModelTreeView wird das neue Interface-Attribut im Interface angezeigt mit Leerzeichen.

22.11.15 Interface-Attribut mit package-protection Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Interface-Attributes mit package-protection Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde eine Interface erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Interfacenamen klicken.

2. Im Texteingabefeld „# hello : Void“ eingeben.

3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Interface-Attribut angezeigt.

2. Links in der ModelTreeView wird das neue Interface-Attribut im Interface angezeigt.

22.11.16 Interface-Attribut mit privater Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen eines Interface-Attributes mit privater Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde eine Interface erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Interfacenamen klicken.
2. Im Texteingabefeld „- hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Interface-Attribut angezeigt.
2. Links in der ModelTreeView wird das neue Interface-Attribut im Interface angezeigt.

22.11.17 Interface-Attribut mit Datentyp aus dem Modell

Dieser Test deckt das Erstellen eines Interface-Attributes mit Datentyp aus dem Modell ab.

Vorbedingung

1. Es wurde eine Interface erstellt und ausgewählt.
2. Der Name dieser Interface lautet „DefaultInterface“

Test

1. Im Editor-Bereich im Bereich unter dem Interfacenamen klicken.
2. Im Texteingabefeld „hello : DefaultInterface“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Interface-Attribut angezeigt.
2. Links in der ModelTreeView wird das neue Interface-Attribut im Interface angezeigt.

22.11.18 Interface-Attribut mit ungültigem Datentyp

Dieser Test deckt das Erstellen eines Interface-Attributes mit ungültigem Datentyp ab.

Vorbedingung

1. Es wurde eine Interface erstellt und ausgewählt.
2. Der Name dieser Interface lautet „DefaultInterface“

Test

1. Im Editor-Bereich im Bereich unter dem Interfacenamen klicken.
2. Im Texteingabefeld „hello : NonExistantInterface“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Eine Fehlermeldung erscheint mit der Meldung „The current model does not contain a type with the requested name 'NonExistantInterface'.“.

22.11.19 Interface-Attribut löschen

Dieser Test deckt das Löschen eines Interface-Attributes ab.

Vorbedingung

1. Es wurde eine Interface erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Interfacenamen klicken.
2. Im Texteingabefeld „- hello : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Interface-Attribut angezeigt.
2. Links in der ModelTreeView wird das neue Interface-Attribut im Interface angezeigt.

22.12 Tests für Interfacemethoden

Die Tests in diesem Abschnitt decken das Erstellen, Bearbeiten und Löschen von Interfacemethoden ab.

22.12.1 Interfacemethode mit Void Primitiv-Typ

Dieser Test deckt das Erstellen einer Interfacemethode vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.
2. Das erstellte Interface wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.
2. Im Texteingabefeld „hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Interfacemethode angezeigt.
2. Links in der ModelTreeView wird die neue Interfacemethode in des Interfaces angezeigt.

22.12.2 Interfacemethode mit Integer Primitiv-Typ

Dieser Test deckt das Erstellen einer Interfacemethode vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.
2. Das erstellte Interface wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.
2. Im Texteingabefeld „hello() : Integer“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Interfacemethode angezeigt.
2. Links in der ModelTreeView wird die neue Interfacemethode in des Interfaces angezeigt.

22.12.3 Interfacemethode mit String Primitiv-Typ

Dieser Test deckt das Erstellen einer Interfacemethode vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.
2. Das erstellte Interface wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.
2. Im Texteingabefeld „hello() : String“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Interfacemethode angezeigt.
2. Links in der ModelTreeView wird die neue Interfacemethode in des Interfaces angezeigt.

22.12.4 Interfacemethode mit Boolean Primitiv-Typ

Dieser Test deckt das Erstellen einer Interfacemethode vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.
2. Das erstellte Interface wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.
2. Im Texteingabefeld „hello() : Boolean“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Interfacemethode angezeigt.
2. Links in der ModelTreeView wird die neue Interfacemethode in des Interfaces angezeigt.

22.12.5 Interfacemethode mit Float Primitiv-Typ

Dieser Test deckt das Erstellen einer Interfacemethode vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.
2. Das erstellte Interface wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.
2. Im Texteingabefeld „hello() : Float“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Interfacemethode angezeigt.
2. Links in der ModelTreeView wird die neue Interfacemethode in des Interfaces angezeigt.

22.12.6 Interfacemethode mit UnlimitedNatural Primitiv-Typ

Dieser Test deckt das Erstellen einer Interfacemethode vom Void Primitiv-Typ ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.
2. Das erstellte Interface wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.
2. Im Texteingabefeld „hello() : UnlimitedNatural“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Interfacemethode angezeigt.
2. Links in der ModelTreeView wird die neue Interfacemethode in des Interfaces angezeigt.

22.12.7 Interfacemethode mit default Sichtbarkeit

Dieser Test deckt das Erstellen einer Interfacemethode mit default Sichtbarkeit ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.
2. Das erstellte Interface wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.
2. Im Texteingabefeld „hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Interfacemethode angezeigt.
2. Links in der ModelTreeView wird die neue Interfacemethode mit protected Sichtbarkeit in des Interfaces angezeigt.

22.12.8 Interfacemethode mit public Sichtbarkeit

Dieser Test deckt das Erstellen einer Interfacemethode mit public Sichtbarkeit ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.
2. Das erstellte Interface wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.
2. Im Texteingabefeld „+hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Interfacemethode angezeigt.
2. Links in der ModelTreeView wird die neue Interfacemethode in des Interfaces angezeigt.

22.12.9 Interfacemethode mit protected Sichtbarkeit

Dieser Test deckt das Erstellen einer Interfacemethode mit protected Sichtbarkeit ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.
2. Das erstellte Interface wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.
2. Im Texteingabefeld „ hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Interfacemethode angezeigt.
2. Links in der ModelTreeView wird die neue Interfacemethode in des Interfaces angezeigt.

22.12.10 Interfacemethode mit package-protection Sichtbarkeit

Dieser Test deckt das Erstellen einer Interfacemethode mit package-protection Sichtbarkeit ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.
2. Das erstellte Interface wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.
2. Im Texteingabefeld „#hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Interfacemethode angezeigt.
2. Links in der ModelTreeView wird die neue Interfacemethode in des Interfaces angezeigt.

22.12.11 Interfacemethode mit privater Sichtbarkeit

Dieser Test deckt das Erstellen einer Interfacemethode mit privater Sichtbarkeit ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.
2. Das erstellte Interface wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.
2. Im Texteingabefeld „-hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Interfacemethode angezeigt.
2. Links in der ModelTreeView wird die neue Interfacemethode in des Interfaces angezeigt.

22.12.12 Interfacemethode mit default Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen einer Interfacemethode mit default Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.
2. Das erstellte Interface wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.
2. Im Texteingabefeld „ hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Interfacemethode angezeigt.
2. Links in der ModelTreeView wird die neue Interfacemethode mit protected Sichtbarkeit in des Interfaces angezeigt.

22.12.13 Interfacemethode mit public Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen einer Interfacemethode mit public Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.

2. Das erstellte Interface wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.

2. Im Texteingabefeld „+ hello() : Void“ eingeben.

3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Interfacemethode angezeigt.

2. Links in der ModelTreeView wird die neue Interfacemethode in des Interfaces angezeigt.

22.12.14 Interfacemethode mit protected Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen einer Interfacemethode mit protected Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.

2. Das erstellte Interface wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.

2. Im Texteingabefeld „ hello() : Void“ eingeben.

3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Interfacemethode angezeigt mit Leerzeichen.
2. Links in der ModelTreeView wird die neue Interfacemethode in des Interfaces angezeigt mit Leerzeichen.

22.12.15 Interfacemethode mit package-protection Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen einer Interfacemethode mit package-protection Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.
2. Das erstellte Interface wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.
2. Im Texteingabefeld „# hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Interfacemethode angezeigt.
2. Links in der ModelTreeView wird die neue Interfacemethode in des Interfaces angezeigt.

22.12.16 Interfacemethode mit privater Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Erstellen einer Interfacemethode mit privater Sichtbarkeit mit Leerzeichen ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.
2. Das erstellte Interface wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.
2. Im Texteingabefeld „- hello() : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Interfacemethode angezeigt.
2. Links in der ModelTreeView wird die neue Interfacemethode in des Interfaces angezeigt.

22.12.17 Interfacemethode mit Datentyp aus dem Modell

Dieser Test deckt das Erstellen einer Interfacemethode mit Datentyp aus dem Modell ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.
2. Der Name dieses Interfaces lautet „DefaultClass“
3. Das erstellte Interface wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.
2. Im Texteingabefeld „hello() : DefaultClass“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Interfacemethode angezeigt.
2. Links in der ModelTreeView wird die neue Interfacemethode in des Interfaces angezeigt.

22.12.18 Interfacemethode mit ungültigem Datentyp

Dieser Test deckt das Erstellen einer Interfacemethode mit ungültigem Datentyp ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.
2. Das erstellte Interface wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.
2. Im Texteingabefeld „hello : NonExistantClass“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Eine Fehlermeldung erscheint mit der Meldung „The current model does not contain a type with the requested name 'NonExistantClass'.“.

22.12.19 Interfacemethode löschen

Dieser Test deckt das Löschen einer Interfacemethode ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.
2. Entfernen drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die Interfacemethode nicht mehr angezeigt.
2. Links in der ModelTreeView wird die Interfacemethode in des Interfaces nicht angezeigt.

22.12.20 Interfacemethode mit Parameter vom Integer Primitiv-Typ

Dieser Test deckt das Erstellen einer Interfacemethode mit Parameter vom Integer Primitiv-Typ ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.

2. Das erstellte Interface wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.

2. Im Texteingabefeld „hello(parameter : Integer) : Void“ eingeben.

3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Interfacemethode angezeigt.

2. Links in der ModelTreeView wird die neue Interfacemethode in des Interfaces angezeigt.

22.12.21 Interfacemethode mit Parameter vom String Primitiv-Typ

Dieser Test deckt das Erstellen einer Interfacemethode mit Parameter vom String Primitiv-Typ ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.

2. Das erstellte Interface wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.

2. Im Texteingabefeld „hello(parameter : String) : Void“ eingeben.

3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Interfacemethode angezeigt.
2. Links in der ModelTreeView wird die neue Interfacemethode in des Interfaces angezeigt.

22.12.22 Interfacemethode mit Parameter vom Boolean Primitiv-Typ

Dieser Test deckt das Erstellen einer Interfacemethode mit Parameter vom Boolean Primitiv-Typ ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.
2. Das erstellte Interface wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.
2. Im Texteingabefeld „hello(parameter : Boolean) : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Interfacemethode angezeigt.
2. Links in der ModelTreeView wird die neue Interfacemethode in des Interfaces angezeigt.

22.12.23 Interfacemethode mit Parameter vom Float Primitiv-Typ

Dieser Test deckt das Erstellen einer Interfacemethode mit Parameter vom Float Primitiv-Typ ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.
2. Das erstellte Interface wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.
2. Im Texteingabefeld „hello(parameter : Float) : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Interfacemethode angezeigt.
2. Links in der ModelTreeView wird die neue Interfacemethode in des Interfaces angezeigt.

22.12.24 Interfacemethode mit Parameter vom Unlimited-Natural Primitiv-Typ

Dieser Test deckt das Erstellen einer Interfacemethode mit Parameter vom UnlimitedNatural Primitiv-Typ ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.
2. Das erstellte Interface wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.
2. Im Texteingabefeld „hello(parameter : UnlimitedNatural) : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Interfacemethode angezeigt.
2. Links in der ModelTreeView wird die neue Interfacemethode in des Interfaces angezeigt.

22.12.25 Interfacemethode mit Parameter vom Datentyp aus dem Modell

Dieser Test deckt das Erstellen einer Interfacemethode mit Parameter vom Datentyp aus dem Modell ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.
2. Der Name des Interfaces wurde zu „myClass“ geändert.
3. Das erstellte Interface wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.
2. Im Texteingabefeld „hello(parameter : myClass) : Void“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Interfacemethode angezeigt.
2. Links in der ModelTreeView wird die neue Interfacemethode in des Interfaces angezeigt.

22.12.26 Interfacemethode mit zwei Parametern

Dieser Test deckt das Erstellen einer Interfacemethode mit zwei Parametern ab.

Vorbedingung

1. Es wurde ein Interface erstellt und ausgewählt.
2. Das erstellte Interface wurde vergrößert, sodass der Methodenbereich sichtbar ist.

Test

1. Im Editor-Bereich im Methodenbereich des Interfaces klicken.

2. Im Texteingabefeld „hello(parameter1 : Integer, parameter2 : Integer) : Void“ eingeben.

3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Interfacemethode angezeigt.

2. Links in der ModelTreeView wird die neue Interfacemethode in des Interfaces angezeigt.

22.13 Tests für Enumerationen

Die Tests in diesem Abschnitt decken das Erstellen, Bearbeiten und Löschen von Enumerationen ab.

22.13.1 Enumeration erstellen

Dieser Test deckt das Erstellen einer Enumeration ab.

Test

1. Im Editor-Bereich links auf „Enum“ klicken.

2. Im Editor-Bereich in der Zeichenfläche klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Enumeration angezeigt.

2. Links in der ModelTreeView wird die neue Enumeration im firstDiagram angezeigt.

22.13.2 Enumeration erstellen mit Größenangabe

Dieser Test deckt das Erstellen einer Enumeration mit Größenangabe ab.

Test

1. Im Editor-Bereich links auf „Enum“ klicken.

2. Im Editor-Bereich in der Zeichenfläche klicken und nach unten rechts ziehen.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die neue Enumeration mit angezeigt.

2. Die neue Enumeration füllt den Bereich vom Drücken der Maustaste bis zum Loslassen der Maustaste.

3. Links in der ModelTreeView wird die neue Enumeration im firstDiagram angezeigt.

22.13.3 Enumeration auswählen

Dieser Test deckt das Auswählen einer Enumeration ab.

Vorbedingung

1. Es wurde eine Enumeration erstellt.

Test

1. Im Editor-Bereich die erstellte Enumeration anklicken.

Erwartetes Ergebnis

1. Im Editor-Bereich wird die ausgewählte Enumeration umrandet angezeigt.

22.13.4 Mehrfachauswahl von Enumerationn

Dieser Test deckt das Auswählen mehrerer Enumerationn ab.

Vorbedingung

1. Es wurden zwei Enumerationn erstellt.

Test

1. Im Editor-Bereich oben links von den beiden Enumerationn klicken und gedrückt halten.
2. Die Maus nach unten rechts ziehen bis die Auswahlanzeige beide Elemente komplett umfasst.
3. Maustaste loslassen.

Erwartetes Ergebnis

1. Im Editor-Bereich werden die beiden ausgewählte Enumeration umrandet angezeigt.

22.13.5 Enumeration umbenennen

Dieser Test deckt das Umbenennen einer Enumeration ab.

Vorbedingung

1. Es wurde eine Enumeration erstellt und ausgewählt.

Test

1. Im Editor-Bereich den Namen der ausgewählten Enumeration anklicken.
2. Im Eingabefeld den neuen Namen „MyTestEnum“ eingeben.
3. Im Editor-Bereich auf ein beliebigen Punkt klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird der neue Name der Enumeration angezeigt.
2. Links in der ModelTreeView wird der neue Name der Enumeration angezeigt.

22.13.6 Enumeration umbenennen (Bestätigung mit Enter)

Dieser Test deckt das Umbenennen einer Enumeration ab.

Vorbedingung

1. Es wurde eine Enumeration erstellt und ausgewählt.

Test

1. Im Editor-Bereich den Namen der ausgewählten Enumeration anklicken.
2. Im Eingabefeld den neuen Namen „MyTestEnum“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird der neue Name der Enumeration angezeigt.
2. Links in der ModelTreeView wird der neue Name der Enumeration angezeigt.

22.13.7 Enumeration verschieben

Dieser Test deckt das Verschieben einer Enumeration ab.

Vorbedingung

1. Es wurde eine Enumeration erstellt.

Test

1. Im Editor-Bereich die erstellte Enumeration anklicken und gedrückt halten.
2. Den Mauszeiger innerhalb des Editor-Bereichs an die neue Position der Enumeration bewegen.
3. Maustaste loslassen.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche befindet sich die verschobene Enumeration an der neuen Position.

22.13.8 Enumeration aus Diagramm löschen

Dieser Test deckt das Löschen einer Enumeration aus einem Diagramm ab.

Vorbedingung

1. Es wurde eine Enumeration erstellt.

Test

1. Im Editor-Bereich die erstellte Enumeration anklicken.
2. Entfernen-Taste drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die gelöschte Enumeration nicht mehr angezeigt.

22.13.9 Enumeration aus Modell löschen

Dieser Test deckt das Löschen einer Enumeration aus einem Modell ab.

Vorbedingung

1. Es wurde eine Enumeration erstellt.

Test

1. Links in der ModelTreeView rechtsklick auf die Enumeration unter 'ModelElements'.
2. „Delete Enum from model“ anklicken.

Erwartetes Ergebnis

1. Links in der ModelTreeView wird das gelöschte Element nicht mehr angezeigt.

22.13.10 Enumeration aus Modell in Diagramm einfügen

Dieser Test deckt das Einfügen einer Enumeration aus einem Modell in ein Diagramm ab.

Vorbedingung

1. Es wurde eine Enumeration erstellt.
2. Die erstellte Enumeration wurde aus dem Diagramm gelöscht.

Test

1. Links in der ModelTreeView auf die Enumeration unter 'ModelElements' doppelklicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die eingefügte Enumeration angezeigt.

22.14 Tests für Enumliterale

Die Tests in diesem Abschnitt decken das Erstellen, Bearbeiten und Löschen von Enumliteralen ab.

22.14.1 Enumliteral erstellen

Dieser Test deckt das Erstellen eines Enumliterals ab.

Vorbedingung

1. Es wurde eine Enumeration erstellt und ausgewählt.

Test

1. Im Editor-Bereich im Bereich unter dem Enumerationsnamen klicken.
2. Im Texteingabefeld „myLiteral“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue Enumliteral angezeigt.
2. Links in der ModelTreeView wird das neue Enumliteral im Enum angezeigt.

22.14.2 Enumliteral umbenennen

Dieser Test deckt das Umbenennen eines Enumliterals ab.

Vorbedingung

1. Es wurde eine Enumeration erstellt und ausgewählt.
2. Es wurde ein Enumliteral in diesem Enum erstellt.

Test

1. Im Editor-Bereich im Enum auf das Enumliteral klicken.
2. Im Texteingabefeld „newLiteralName“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird der neue Name des Enumliterals angezeigt.
2. Links in der ModelTreeView wird der neue Name des Enumliterals im Enum angezeigt.

22.14.3 Enumliteral umbenennen

Dieser Test deckt das Löschen eines Klassenattributes mit default Sichtbarkeit ab.

Vorbedingung

1. Es wurde eine Enumeration erstellt und ausgewählt.
2. Es wurde ein Enumliteral in diesem Enum erstellt.

Test

1. Im Editor-Bereich im Enum auf das Enumliteral klicken.
2. Entfernen-Taste drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das Enumliteral nicht mehr angezeigt.
2. Links in der ModelTreeView wird das Enumliteral nicht mehr im Enum angezeigt.

22.15 Tests für Generalisierungen

Die Tests in diesem Abschnitt decken das Erstellen, Bearbeiten und Löschen von Generalisierungen/Implementierungen ab.

22.15.1 Generalisierung von einer Klasse zu einer Klasse

Dieser Test deckt das Erstellen einer Generalisierung von einer Klasse zu einer Klasse ab.

Vorbedingung

1. Es wurden 2 Klassen erstellt.

Test

1. Links im Editor-Bereich auf Generalization klicken.
2. Auf die erste Klasse klicken.
3. Auf die zweite Klasse klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Generalisierung von der ersten zur zweiten Klasse angezeigt.
2. Links in der ModelTreeView wird die neue Generalisierung unter der ersten Klasse im firstDiagram angezeigt.

22.15.2 Generalisierung von einer Klasse zu einer Abstrakten Klasse

Dieser Test deckt das Erstellen einer Generalisierung von einer Klasse zu einer Abstrakten Klasse ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.
2. Es wurde 1 Klasse erstellt.
3. Es wurde 1 Abstrakte Klasse erstellt.

Test

1. Links im Editor-Bereich auf Generalization klicken.
2. Auf die Klasse klicken.
3. Auf die Abstrakte Klasse klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Generalisierung von der Klasse zur Abstrakten Klasse angezeigt.

2. Links in der ModelTreeView wird die neue Generalisierung unter der Klasse im firstDiagram angezeigt.

22.15.3 Implementierung von einer Klasse zu einem Interface

Dieser Test deckt das Erstellen einer Generalisierung von einer Klasse zu einem Interface ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.

2. Es wurde 1 Klasse erstellt.

3. Es wurde 1 Interface erstellt.

Test

1. Links im Editor-Bereich auf Generalization klicken.

2. Auf die Klasse klicken.

3. Auf das Interace klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Implementierung von der Klasse zum Interace angezeigt.

2. Links in der ModelTreeView wird die neue Implementierung unter der Klasse im firstDiagram angezeigt.

22.15.4 Generalisierung von einer Abstrakten Klasse zu einer Abstrakten Klasse

Dieser Test deckt das Erstellen einer Generalisierung von einer Abstrakten Klasse zu einer Abstrakten Klasse ab.

Vorbedingung

1. Es wurden 2 Abstrakte Klassen erstellt.

Test

1. Links im Editor-Bereich auf Generalization klicken.
2. Auf die erste Abstrakte Klasse klicken.
3. Auf die zweite Abstrakte Klasse klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Generalisierung von der ersten zur zweiten Abstrakten Klasse angezeigt.
2. Links in der ModelTreeView wird die neue Generalisierung unter der ersten Abstrakten Klasse im firstDiagram angezeigt.

22.15.5 Generalisierung von einer Abstrakten Klasse zu einer Klasse

Dieser Test deckt das Erstellen einer Generalisierung von einer Abstrakten Klasse zu einer Klasse ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.
2. Es wurde 1 Abstrakte Klasse erstellt.
3. Es wurde 1 Klasse erstellt.

Test

1. Links im Editor-Bereich auf Generalization klicken.
2. Auf die Abstrakte Klasse klicken.
3. Auf die Klasse klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Generalisierung von der Abstrakten Klasse zur Klasse angezeigt.
2. Links in der ModelTreeView wird die neue Generalisierung unter der Abstrakten Klasse im firstDiagram angezeigt.

22.15.6 Generalisierung von einer Abstrakten Klasse zu einem Interface

Dieser Test deckt das Erstellen einer Generalisierung von einer Abstrakten Klasse zu einem Interface ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.
2. Es wurde 1 Abstrakte Klasse erstellt.
3. Es wurde 1 Interface erstellt.

Test

1. Links im Editor-Bereich auf Generalization klicken.
2. Auf die Abstrakte Klasse klicken.
3. Auf das Interface klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Implementierung von der Abstrakten Klasse zum Interface angezeigt.
2. Links in der ModelTreeView wird die neue Implementierung unter der Abstrakten Klasse im firstDiagram angezeigt.

22.15.7 Generalisierung von einem Interface zu einem Interface

Dieser Test deckt das Erstellen einer Generalisierung von einem Interface zu einem Interface ab.

Vorbedingung

1. Es wurden 2 Interfaces erstellt.

Test

1. Links im Editor-Bereich auf Generalization klicken.
2. Auf das erste Interface klicken.
3. Auf das zweite Interface klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Generalisierung vom ersten zum zweiten Interface angezeigt.
2. Links in der ModelTreeView wird die neue Generalisierung unter dem ersten Interface im firstDiagram angezeigt.

22.15.8 Selektion einer Generalisierung

Dieser Test deckt das Löschen einer Generalisierung ab.

Vorbedingung

1. Es wurden 2 Klassen erstellt.
2. Es wurde 1 Generalisierung zwischen diesen erstellt.

Test

1. Im Editor-Bereich auf die Generalisierung klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche ist die Generalisierung hervorgehoben.

22.15.9 Löschen einer Generalisierung aus dem Diagramm

Dieser Test deckt das Löschen einer Generalisierung aus dem Diagramm ab.

Vorbedingung

1. Es wurden 2 Klassen erstellt.
2. Es wurde 1 Generalisierung zwischen diesen erstellt.
3. Die Generalisierung wurde selektiert.

Test

1. Entfernen drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die gelöschte Generalisierung nicht mehr angezeigt.
2. Links in der ModelTreeView wird die gelöschte Generalisierung noch angezeigt.

22.15.10 Löschen einer Generalisierung aus dem Modell

Dieser Test deckt das Löschen einer Generalisierung aus dem Modell ab.

Vorbedingung

1. Es wurden 2 Klassen erstellt.
2. Es wurde 1 Generalisierung zwischen diesen erstellt.

Test

1. Links in der ModelTreeView rechtsklick auf die Generalisierung.
2. „Delete Generalization from model“ anklicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die gelöschte Generalisierung nicht mehr angezeigt.
2. Links in der ModelTreeView wird die gelöschte Generalisierung nicht mehr angezeigt.

22.15.11 Keine Generalisierung von einem Interface zu einer Klasse

Dieser Test deckt das Fehlschlagen einer Generalisierung von einem Interface zu einer Klasse ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.
2. Es wurde 1 Interface erstellt.
3. Es wurde 1 Klasse erstellt.

Test

1. Links im Editor-Bereich auf Generalization klicken.
2. Auf das Interface klicken.
3. Auf die Klasse klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird keine Generalisierung vom Interface zur Klasse angezeigt.

2. Links in der ModelTreeView wird keine neue Generalisierung unter dem Interface im firstDiagram angezeigt.

22.15.12 Keine Generalisierung von einem Interface zu einer Abstrakten Klasse

Dieser Test deckt das Fehlschlagen einer Generalisierung von einem Interface zu einer Abstrakten Klasse ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.

2. Es wurde 1 Interface erstellt.

3. Es wurde 1 Abstrakte Klasse erstellt.

Test

1. Links im Editor-Bereich auf Generalization klicken.

2. Auf das Interface klicken.

3. Auf die Abstrakte Klasse klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird keine Generalisierung vom Interface zur Abstrakten Klasse angezeigt.

2. Links in der ModelTreeView wird keine neue Generalisierung unter dem Interface im firstDiagram angezeigt.

22.15.13 Keine Generalisierung mit einem Enum als Startelement

Dieser Test deckt das Fehlschlagen einer Generalisierung von einem Enum zu einem Enum ab.

Vorbedingung

1. Es wurden 2 Enums erstellt.

Test

1. Links im Editor-Bereich auf Generalization klicken.
2. Auf das erste Enum klicken.

Erwartetes Ergebnis

1. Das Generalisierungstool kann nicht auf das Enum als Startelement angewandt werden, wodurch das Auswählen eines Ziels unmöglich ist.

22.15.14 Keine Generalisierung von einer Klasse zu einem Enum

Dieser Test deckt das Fehlschlagen einer Generalisierung von einer Klasse zu einem Enum ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.
2. Es wurde 1 Klasse erstellt.
3. Es wurde 1 Enum erstellt.

Test

1. Links im Editor-Bereich auf Generalization klicken.
2. Auf die Klasse klicken.
3. Auf das Enum klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird keine Generalisierung von der Klasse zur Abstrakten Klasse angezeigt.
2. Links in der ModelTreeView wird keine neue Generalisierung unter der Klasse im firstDiagram angezeigt.

22.15.15 Keine Generalisierung von einer Abstrakten Klasse zu einem Enum

Dieser Test deckt das Fehlschlagen einer Generalisierung von einer Abstrakten Klasse zu einem Enum ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.
2. Es wurde 1 Abstrakte Klasse erstellt.
3. Es wurde 1 Enum erstellt.

Test

1. Links im Editor-Bereich auf Generalization klicken.
2. Auf die Abstrakte Klasse klicken.
3. Auf das Enum klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird keine Generalisierung von der Abstrakten Klasse zur Abstrakten Klasse angezeigt.
2. Links in der ModelTreeView wird keine neue Generalisierung unter der Abstrakten Klasse im firstDiagram angezeigt.

22.15.16 Keine Generalisierung von einem Interface zu einem Enum

Dieser Test deckt das Fehlschlagen einer Generalisierung von einem Interface zu einem Enum ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.
2. Es wurde 1 Interface erstellt.
3. Es wurde 1 Enum erstellt.

Test

1. Links im Editor-Bereich auf Generalization klicken.
2. Auf das Interface klicken.
3. Auf das Enum klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird keine Generalisierung vom Interface zum Enum angezeigt.

2. Links in der ModelTreeView wird keine neue Generalisierung unter dem Interface im firstDiagram angezeigt.

22.16 Tests für Assoziationen

Die Tests in diesem Abschnitt decken das Erstellen, Bearbeiten und Löschen von Assoziationen ab.

22.16.1 Assoziation von einer Klasse zu einer Klasse

Dieser Test deckt das Erstellen einer Assoziation von einer Klasse zu einer Klasse ab.

Vorbedingung

1. Es wurden 2 Klassen erstellt.

Test

1. Links im Editor-Bereich auf Association klicken.

2. Auf die erste Klasse klicken.

3. Auf die zweite Klasse klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Assoziation von der ersten zur zweiten Klasse angezeigt.

2. Links in der ModelTreeView wird die neue Assoziation unter der ersten Klasse im firstDiagram angezeigt.

22.16.2 Assoziation von einer Klasse zu einer Abstrakten Klasse

Dieser Test deckt das Erstellen einer Assoziation von einer Klasse zu einer Abstrakten Klasse ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.
2. Es wurde 1 Klasse erstellt.
3. Es wurde 1 Abstrakte Klasse erstellt.

Test

1. Links im Editor-Bereich auf Association klicken.
2. Auf die Klasse klicken.
3. Auf die Abstrakte Klasse klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Assoziation von der Klasse zur Abstrakten Klasse angezeigt.
2. Links in der ModelTreeView wird die neue Assoziation unter der Klasse im firstDiagram angezeigt.

22.16.3 Assoziation von einer Klasse zu einem Interface

Dieser Test deckt das Erstellen einer Assoziation von einer Klasse zu einem Interface ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.
2. Es wurde 1 Klasse erstellt.
3. Es wurde 1 Interface erstellt.

Test

1. Links im Editor-Bereich auf Association klicken.
2. Auf die Klasse klicken.
3. Auf das Interace klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Assoziation von der Klasse zum Interace angezeigt.

2. Links in der ModelTreeView wird die neue Assoziation unter der Klasse im firstDiagram angezeigt.

22.16.4 Assoziation von einer Abstrakten Klasse zu einer Abstrakten Klasse

Dieser Test deckt das Erstellen einer Assoziation von einer Abstrakten Klasse zu einer Abstrakten Klasse ab.

Vorbedingung

1. Es wurden 2 Abstrakte Klassen erstellt.

Test

1. Links im Editor-Bereich auf Association klicken.

2. Auf die erste Abstrakte Klasse klicken.

3. Auf die zweite Abstrakte Klasse klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Assoziation von der ersten zur zweiten Abstrakten Klasse angezeigt.

2. Links in der ModelTreeView wird die neue Assoziation unter der ersten Abstrakten Klasse im firstDiagram angezeigt.

22.16.5 Assoziation von einer Abstrakten Klasse zu einer Klasse

Dieser Test deckt das Erstellen einer Assoziation von einer Abstrakten Klasse zu einer Klasse ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.

2. Es wurde 1 Abstrakte Klasse erstellt.

3. Es wurde 1 Klasse erstellt.

Test

1. Links im Editor-Bereich auf Association klicken.
2. Auf die Abstrakte Klasse klicken.
3. Auf die Klasse klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Assoziation von der Abstrakten Klasse zur Klasse angezeigt.
2. Links in der ModelTreeView wird die neue Assoziation unter der Abstrakten Klasse im firstDiagram angezeigt.

22.16.6 Assoziation von einer Abstrakten Klasse zu einem Interface

Dieser Test deckt das Erstellen einer Assoziation von einer Abstrakten Klasse zu einem Interface ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.
2. Es wurde 1 Abstrakte Klasse erstellt.
3. Es wurde 1 Interface erstellt.

Test

1. Links im Editor-Bereich auf Association klicken.
2. Auf die Abstrakte Klasse klicken.
3. Auf das Interface klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Assoziation von der Abstrakten Klasse zum Interface angezeigt.
2. Links in der ModelTreeView wird die neue Assoziation unter der Abstrakten Klasse im firstDiagram angezeigt.

22.16.7 Assoziation von einem Interface zu einem Interface

Dieser Test deckt das Erstellen einer Assoziation von einem Interface zu einem Interface ab.

Vorbedingung

1. Es wurden 2 Interfaces erstellt.

Test

1. Links im Editor-Bereich auf Association klicken.
2. Auf das erste Interface klicken.
3. Auf das zweite Interface klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Assoziation vom ersten zum zweiten Interface angezeigt.
2. Links in der ModelTreeView wird die neue Assoziation unter dem ersten Interface im firstDiagram angezeigt.

22.16.8 Assoziation von einem Interface zu einer Klasse

Dieser Test deckt das Erstellen einer Assoziation von einem Interface zu einer Klasse ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.
2. Es wurde 1 Interface erstellt.
3. Es wurde 1 Klasse erstellt.

Test

1. Links im Editor-Bereich auf Association klicken.
2. Auf das Interface klicken.
3. Auf die Klasse klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Assoziation vom Interface zur Klasse angezeigt.
2. Links in der ModelTreeView wird die neue Assoziation unter dem Interface im firstDiagram angezeigt.

22.16.9 Assoziation von einem Interface zu einer Abstrakten Klasse

Dieser Test deckt das Erstellen einer Assoziation von einem Interface zu einer Abstrakten Klasse ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.
2. Es wurde 1 Interface erstellt.
3. Es wurde 1 Abstrakte Klasse erstellt.

Test

1. Links im Editor-Bereich auf Association klicken.
2. Auf das Interface klicken.
3. Auf die Abstrakte Klasse klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Assoziation vom Interface zur Abstrakten Klasse angezeigt.
2. Links in der ModelTreeView wird die neue Assoziation unter dem Interface im firstDiagram angezeigt.

22.16.10 Selektion einer Assoziation

Dieser Test deckt das Löschen einer Assoziation ab.

Vorbedingung

1. Es wurden 2 Klassen erstellt.
2. Es wurde 1 Assoziation zwischen diesen erstellt.

Test

1. Im Editor-Bereich auf die Assoziation klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche ist die Assoziation hervorgehoben.

22.16.11 Löschen einer Assoziation aus dem Diagramm

Dieser Test deckt das Löschen einer Assoziation aus dem Diagramm ab.

Vorbedingung

1. Es wurden 2 Klassen erstellt.
2. Es wurde 1 Assoziation zwischen diesen erstellt.
3. Die Assoziation wurde selektiert.

Test

1. Entfernen drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die gelöschte Assoziation nicht mehr angezeigt.
2. Links in der ModelTreeView wird die gelöschte Assoziation noch angezeigt.

22.16.12 Löschen einer Assoziation aus dem Modell

Dieser Test deckt das Löschen einer Assoziation aus dem Modell ab.

Vorbedingung

1. Es wurden 2 Klassen erstellt.
2. Es wurde 1 Assoziation zwischen diesen erstellt.

Test

1. Links in der ModelTreeView rechtsklick auf die Assoziation.
2. „Delete Association from model“ anklicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die gelöschte Assoziation nicht mehr angezeigt.
2. Links in der ModelTreeView wird die gelöschte Assoziation nicht mehr angezeigt.

22.16.13 Umhängen des Startpunktes einer Assoziation

Dieser Test deckt das Umhängen des Startpunktes einer Generalisierung ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 3 Knoten erstellt.
2. Es wurde eine Assoziationen zwischen diesen erstellt.
3. Die Assoziation wurde selektiert.

Test

1. Der Startpunkt der Assoziation per Drag and Drop auf ein anderes Knoten verändern.

Erwartetes Ergebnis

1. Der Startpunkt der Assoziation wird nur dann verändert, wenn die daraus resultierende Assoziation gültig ist.

22.16.14 Umhängen des Endpunktes einer Assoziation

Dieser Test deckt das Umhängen des Endpunktes einer Generalisierung ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 3 Knoten erstellt.
2. Es wurde eine Assoziationen zwischen diesen erstellt.
3. Die Assoziation wurde selektiert.

Test

1. Der Endpunkt der Assoziation per Drag and Drop auf ein anderes Knoten verändern.

Erwartetes Ergebnis

1. Der Endpunkt der Assoziation wird nur dann verändert, wenn die daraus resultierende Assoziation gültig ist.

22.16.15 Sichtbarkeit des Startpunktes einer Assoziation aktivieren

Dieser Test deckt das Aktivieren der Sichtbarkeit des Startpunktes einer Assoziation ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation zwischen diesen erstellt.
3. Die Assoziation wurde selektiert.

Test

1. Rechtsklick auf die Assoziation.
2. Klick auf „Toggle source navigability“.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird ein Pfeil am Startpunkt der Assoziation angezeigt.

22.16.16 Sichtbarkeit des Startpunktes einer Assoziation deaktivieren

Dieser Test deckt das Aktivieren der Sichtbarkeit des Startpunktes einer Assoziation ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation zwischen diesen erstellt.
3. Die Sichtbarkeit des Startpunktes der Assoziation wurde aktiviert.
4. Die Assoziation wurde selektiert.

Test

1. Rechtsklick auf die Assoziation.
2. Klick auf „Toggle source navigability“.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird kein Pfeil am Startpunkt der Assoziation angezeigt.

22.16.17 Sichtbarkeit des Endpunktes einer Assoziation aktivieren

Dieser Test deckt das Aktivieren der Sichtbarkeit des Endpunktes einer Assoziation ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation zwischen diesen erstellt.
3. Die Assoziation wurde selektiert.

Test

1. Rechtsklick auf die Assoziation.
2. Klick auf „Toggle source navigability“.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird ein Pfeil am Endpunkt der Assoziation angezeigt.

22.16.18 Sichtbarkeit des Endpunktes einer Assoziation deaktivieren

Dieser Test deckt das Aktivieren der Sichtbarkeit des Endpunktes einer Assoziation ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation zwischen diesen erstellt.
3. Die Sichtbarkeit des Endpunktes der Assoziation wurde aktiviert.
4. Die Assoziation wurde selektiert.

Test

1. Rechtsklick auf die Assoziation.
2. Klick auf „Toggle source navigability“.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird kein Pfeil am Endpunkt der Assoziation angezeigt.

22.16.19 Umwandlung einer Assoziation zu einer Aggregation

Dieser Test deckt das Umwandeln einer Assoziation zu einer Aggregation ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation zwischen diesen erstellt.
3. Die Assoziation wurde selektiert.

Test

1. Rechtsklick auf die Assoziation.
2. Klick auf „Change to aggregation“.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird anstelle der Assoziation eine Aggregation angezeigt.
2. Links in der ModelTreeView wird anstelle der Assoziation eine Aggregation angezeigt.

22.16.20 Umwandlung einer Assoziation zu einer Komposition

Dieser Test deckt das Umwandeln einer Assoziation zu einer Komposition ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation zwischen diesen erstellt.
3. Die Assoziation wurde selektiert.

Test

1. Rechtsklick auf die Assoziation.
2. Klick auf „Change to composition“.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird anstelle der Assoziation eine Komposition angezeigt.
2. Links in der ModelTreeView wird anstelle der Assoziation eine Komposition angezeigt.

22.16.21 Erstellen eines Bendpoints in einer Assoziation

Dieser Test deckt das Erstellen eines Bendpoints in einer Assoziation ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation zwischen diesen erstellt.
3. Die Assoziation wurde selektiert.

Test

1. Klick auf den mittleren markierten Punkt auf der Assoziation.
2. Verschieben des Bendpoints per Drag and Drop.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die Assoziation über den Bendpoint umgeleitet angezeigt.

22.16.22 Verschieben eines Bendpoints in einer Assoziation

Dieser Test deckt das Verschieben eines Bendpoints in einer Assoziation ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation zwischen diesen erstellt.
3. Es wurde an der Assoziation ein Bendpoint erstellt.
4. Die Assoziation wurde selektiert.

Test

1. Klick auf den Bendpoint auf der Assoziation.
2. Verschieben des Bendpoints per Drag and Drop an eine Position, sodass die Assoziation nicht gerade ist.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die Assoziation über den verschobenen Bendpoint umgeleitet angezeigt.

22.16.23 Löschen eines Bendpoints in einer Assoziation

Dieser Test deckt das Löschen eines Bendpoints in einer Assoziation ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation zwischen diesen erstellt.
3. Es wurde an der Assoziation ein Bendpoint erstellt.
4. Die Assoziation wurde selektiert.

Test

1. Klick auf den Bendpoint auf der Assoziation.
2. Verschieben des Bendpoints per Drag and Drop, sodass die Assoziation wieder gerade ist.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die Assoziation ohne Bendpoint angezeigt.

22.17 Tests für Aggregationen

Die Tests in diesem Abschnitt decken das Erstellen, Bearbeiten und Löschen von Aggregationen ab.

22.17.1 Aggregation von einer Klasse zu einer Klasse

Dieser Test deckt das Erstellen einer Aggregation von einer Klasse zu einer Klasse ab.

Vorbedingung

1. Es wurden 2 Klassen erstellt.

Test

1. Links im Editor-Bereich auf Aggregation klicken.
2. Auf die erste Klasse klicken.
3. Auf die zweite Klasse klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Aggregation von der ersten zur zweiten Klasse angezeigt.
2. Links in der ModelTreeView wird die neue Aggregation unter der ersten Klasse im firstDiagram angezeigt.

22.17.2 Aggregation von einer Klasse zu einer Abstrakten Klasse

Dieser Test deckt das Erstellen einer Aggregation von einer Klasse zu einer Abstrakten Klasse ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.
2. Es wurde 1 Klasse erstellt.
3. Es wurde 1 Abstrakte Klasse erstellt.

Test

1. Links im Editor-Bereich auf Aggregation klicken.
2. Auf die Klasse klicken.
3. Auf die Abstrakte Klasse klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Aggregation von der Klasse zur Abstrakten Klasse angezeigt.
2. Links in der ModelTreeView wird die neue Aggregation unter der Klasse im firstDiagram angezeigt.

22.17.3 Aggregation von einer Klasse zu einem Interface

Dieser Test deckt das Erstellen einer Aggregation von einer Klasse zu einem Interface ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.
2. Es wurde 1 Klasse erstellt.
3. Es wurde 1 Interface erstellt.

Test

1. Links im Editor-Bereich auf Aggregation klicken.
2. Auf die Klasse klicken.
3. Auf das Interace klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Aggregation von der Klasse zum Interace angezeigt.

2. Links in der ModelTreeView wird die neue Aggregation unter der Klasse im firstDiagram angezeigt.

22.17.4 Aggregation von einer Abstrakten Klasse zu einer Abstrakten Klasse

Dieser Test deckt das Erstellen einer Aggregation von einer Abstrakten Klasse zu einer Abstrakten Klasse ab.

Vorbedingung

1. Es wurden 2 Abstrakte Klassen erstellt.

Test

1. Links im Editor-Bereich auf Aggregation klicken.

2. Auf die erste Abstrakte Klasse klicken.

3. Auf die zweite Abstrakte Klasse klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Aggregation von der ersten zur zweiten Abstrakten Klasse angezeigt.

2. Links in der ModelTreeView wird die neue Aggregation unter der ersten Abstrakten Klasse im firstDiagram angezeigt.

22.17.5 Aggregation von einer Abstrakten Klasse zu einer Klasse

Dieser Test deckt das Erstellen einer Aggregation von einer Abstrakten Klasse zu einer Klasse ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.

2. Es wurde 1 Abstrakte Klasse erstellt.

3. Es wurde 1 Klasse erstellt.

Test

1. Links im Editor-Bereich auf Aggregation klicken.
2. Auf die Abstrakte Klasse klicken.
3. Auf die Klasse klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Aggregation von der Abstrakten Klasse zur Klasse angezeigt.
2. Links in der ModelTreeView wird die neue Aggregation unter der Abstrakten Klasse im firstDiagram angezeigt.

22.17.6 Aggregation von einer Abstrakten Klasse zu einem Interface

Dieser Test deckt das Erstellen einer Aggregation von einer Abstrakten Klasse zu einem Interface ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.
2. Es wurde 1 Abstrakte Klasse erstellt.
3. Es wurde 1 Interface erstellt.

Test

1. Links im Editor-Bereich auf Aggregation klicken.
2. Auf die Abstrakte Klasse klicken.
3. Auf das Interface klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Aggregation von der Abstrakten Klasse zum Interface angezeigt.
2. Links in der ModelTreeView wird die neue Aggregation unter der Abstrakten Klasse im firstDiagram angezeigt.

22.17.7 Aggregation von einem Interface zu einem Interface

Dieser Test deckt das Erstellen einer Aggregation von einem Interface zu einem Interface ab.

Vorbedingung

1. Es wurden 2 Interfaces erstellt.

Test

1. Links im Editor-Bereich auf Aggregation klicken.
2. Auf das erste Interface klicken.
3. Auf das zweite Interface klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Aggregation vom ersten zum zweiten Interface angezeigt.
2. Links in der ModelTreeView wird die neue Aggregation unter dem ersten Interface im firstDiagram angezeigt.

22.17.8 Aggregation von einem Interface zu einer Klasse

Dieser Test deckt das Erstellen einer Aggregation von einem Interface zu einer Klasse ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.
2. Es wurde 1 Interface erstellt.
3. Es wurde 1 Klasse erstellt.

Test

1. Links im Editor-Bereich auf Aggregation klicken.
2. Auf das Interface klicken.
3. Auf die Klasse klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Aggregation vom Interface zur Klasse angezeigt.
2. Links in der ModelTreeView wird die neue Aggregation unter dem Interface im firstDiagram angezeigt.

22.17.9 Aggregation von einem Interface zu einer Abstrakten Klasse

Dieser Test deckt das Erstellen einer Aggregation von einem Interface zu einer Abstrakten Klasse ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.
2. Es wurde 1 Interface erstellt.
3. Es wurde 1 Abstrakte Klasse erstellt.

Test

1. Links im Editor-Bereich auf Aggregation klicken.
2. Auf das Interface klicken.
3. Auf die Abstrakte Klasse klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Aggregation vom Interface zur Abstrakten Klasse angezeigt.
2. Links in der ModelTreeView wird die neue Aggregation unter dem Interface im firstDiagram angezeigt.

22.17.10 Selektion einer Aggregation

Dieser Test deckt das Löschen einer Aggregation ab.

Vorbedingung

1. Es wurden 2 Klassen erstellt.
2. Es wurde 1 Aggregation zwischen diesen erstellt.

Test

1. Im Editor-Bereich auf die Aggregation klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche ist die Aggregation hervorgehoben.

22.17.11 Löschen einer Aggregation aus dem Diagramm

Dieser Test deckt das Löschen einer Aggregation aus dem Diagramm ab.

Vorbedingung

1. Es wurden 2 Klassen erstellt.
2. Es wurde 1 Aggregation zwischen diesen erstellt.
3. Die Aggregation wurde selektiert.

Test

1. Entfernen drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die gelöschte Aggregation nicht mehr angezeigt.
2. Links in der ModelTreeView wird die gelöschte Aggregation noch angezeigt.

22.17.12 Löschen einer Aggregation aus dem Modell

Dieser Test deckt das Löschen einer Aggregation aus dem Modell ab.

Vorbedingung

1. Es wurden 2 Klassen erstellt.
2. Es wurde 1 Aggregation zwischen diesen erstellt.

Test

1. Links in der ModelTreeView rechtsklick auf die Aggregation.
2. „Delete Aggregation from model“ anklicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die gelöschte Aggregation nicht mehr angezeigt.
2. Links in der ModelTreeView wird die gelöschte Aggregation nicht mehr angezeigt.

22.17.13 Umhängen des Startpunktes einer Aggregation

Dieser Test deckt das Umhängen des Startpunktes einer Generalisierung ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 3 Knoten erstellt.
2. Es wurde eine Aggregationen zwischen diesen erstellt.
3. Die Aggregation wurde selektiert.

Test

1. Der Startpunkt der Aggregation per Drag and Drop auf ein anderes Knoten verändern.

Erwartetes Ergebnis

1. Der Startpunkt der Aggregation wird nur dann verändert, wenn die daraus resultierende Aggregation gültig ist.

22.17.14 Umhängen des Endpunktes einer Aggregation

Dieser Test deckt das Umhängen des Endpunktes einer Generalisierung ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 3 Knoten erstellt.
2. Es wurde eine Aggregationen zwischen diesen erstellt.
3. Die Aggregation wurde selektiert.

Test

1. Der Endpunkt der Aggregation per Drag and Drop auf ein anderes Knoten verändern.

Erwartetes Ergebnis

1. Der Endpunkt der Aggregation wird nur dann verändert, wenn die daraus resultierende Aggregation gültig ist.

22.17.15 Sichtbarkeit des Startpunktes einer Aggregation aktivieren

Dieser Test deckt das Aktivieren der Sichtbarkeit des Startpunktes einer Aggregation ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Aggregation zwischen diesen erstellt.
3. Die Aggregation wurde selektiert.

Test

1. Rechtsklick auf die Aggregation.
2. Klick auf „Toggle source navigability“.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird ein Pfeil am Startpunkt der Aggregation angezeigt.

22.17.16 Sichtbarkeit des Startpunktes einer Aggregation deaktivieren

Dieser Test deckt das Aktivieren der Sichtbarkeit des Startpunktes einer Aggregation ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Aggregation zwischen diesen erstellt.
3. Die Sichtbarkeit des Startpunktes der Aggregation wurde aktiviert.
4. Die Aggregation wurde selektiert.

Test

1. Rechtsklick auf die Aggregation.
2. Klick auf „Toggle source navigability“.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird kein Pfeil am Startpunkt der Aggregation angezeigt.

22.17.17 Sichtbarkeit des Endpunktes einer Aggregation aktivieren

Dieser Test deckt das Aktivieren der Sichtbarkeit des Endpunktes einer Aggregation ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Aggregation zwischen diesen erstellt.
3. Die Aggregation wurde selektiert.

Test

1. Rechtsklick auf die Aggregation.
2. Klick auf „Toggle source navigability“.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird ein Pfeil am Endpunkt der Aggregation angezeigt.

22.17.18 Sichtbarkeit des Endpunktes einer Aggregation deaktivieren

Dieser Test deckt das Aktivieren der Sichtbarkeit des Endpunktes einer Aggregation ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Aggregation zwischen diesen erstellt.
3. Die Sichtbarkeit des Endpunktes der Aggregation wurde aktiviert.
4. Die Aggregation wurde selektiert.

Test

1. Rechtsklick auf die Aggregation.
2. Klick auf „Toggle source navigability“.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird kein Pfeil am Endpunkt der Aggregation angezeigt.

22.17.19 Umwandlung einer Aggregation zu einer Assoziation

Dieser Test deckt das Umwandeln einer Aggregation zu einer Assoziation ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Aggregation zwischen diesen erstellt.
3. Die Aggregation wurde selektiert.

Test

1. Rechtsklick auf die Aggregation.
2. Klick auf „Change to association“.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird anstelle der Aggregation eine Assoziation angezeigt.
2. Links in der ModelTreeView wird anstelle der Aggregation eine Assoziation angezeigt.

22.17.20 Umwandlung einer Aggregation zu einer Komposition

Dieser Test deckt das Umwandeln einer Aggregation zu einer Komposition ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Aggregation zwischen diesen erstellt.
3. Die Aggregation wurde selektiert.

Test

1. Rechtsklick auf die Aggregation.
2. Klick auf „Change to composition“.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird anstelle der Aggregation eine Komposition angezeigt.
2. Links in der ModelTreeView wird anstelle der Aggregation eine Komposition angezeigt.

22.17.21 Erstellen eines Bendpoints in einer Aggregation

Dieser Test deckt das Erstellen eines Bendpoints in einer Aggregation ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Aggregation zwischen diesen erstellt.
3. Die Aggregation wurde selektiert.

Test

1. Klick auf den mittleren markierten Punkt auf der Aggregation.
2. Verschieben des Bendpoints per Drag and Drop.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die Aggregation über den Bendpoint umgeleitet angezeigt.

22.17.22 Verschieben eines Bendpoints in einer Aggregation

Dieser Test deckt das Verschieben eines Bendpoints in einer Aggregation ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Aggregation zwischen diesen erstellt.
3. Es wurde an der Aggregation ein Bendpoint erstellt.
4. Die Aggregation wurde selektiert.

Test

1. Klick auf den Bendpoint auf der Aggregation.
2. Verschieben des Bendpoints per Drag and Drop an eine Position, sodass die Aggregation nicht gerade ist.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die Aggregation über den verschobenen Bendpoint umgeleitet angezeigt.

22.17.23 Löschen eines Bendpoints in einer Aggregation

Dieser Test deckt das Löschen eines Bendpoints in einer Aggregation ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Aggregation zwischen diesen erstellt.
3. Es wurde an der Aggregation ein Bendpoint erstellt.
4. Die Aggregation wurde selektiert.

Test

1. Klick auf den Bendpoint auf der Aggregation.
2. Verschieben des Bendpoints per Drag and Drop, sodass die Aggregation wieder gerade ist.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die Aggregation ohne Bendpoint angezeigt.

22.18 Tests für Kompositionen

Die Tests in diesem Abschnitt decken das Erstellen, Bearbeiten und Löschen von Kompositionen ab.

22.18.1 Komposition von einer Klasse zu einer Klasse

Dieser Test deckt das Erstellen einer Komposition von einer Klasse zu einer Klasse ab.

Vorbedingung

1. Es wurden 2 Klassen erstellt.

Test

1. Links im Editor-Bereich auf Composition klicken.
2. Auf die erste Klasse klicken.
3. Auf die zweite Klasse klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Komposition von der ersten zur zweiten Klasse angezeigt.
2. Links in der ModelTreeView wird die neue Komposition unter der ersten Klasse im firstDiagram angezeigt.

22.18.2 Komposition von einer Klasse zu einer Abstrakten Klasse

Dieser Test deckt das Erstellen einer Komposition von einer Klasse zu einer Abstrakten Klasse ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.
2. Es wurde 1 Klasse erstellt.
3. Es wurde 1 Abstrakte Klasse erstellt.

Test

1. Links im Editor-Bereich auf Composition klicken.
2. Auf die Klasse klicken.
3. Auf die Abstrakte Klasse klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Komposition von der Klasse zur Abstrakten Klasse angezeigt.
2. Links in der ModelTreeView wird die neue Komposition unter der Klasse im firstDiagram angezeigt.

22.18.3 Komposition von einer Klasse zu einem Interface

Dieser Test deckt das Erstellen einer Komposition von einer Klasse zu einem Interface ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.
2. Es wurde 1 Klasse erstellt.
3. Es wurde 1 Interface erstellt.

Test

1. Links im Editor-Bereich auf Composition klicken.
2. Auf die Klasse klicken.
3. Auf das Interace klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Komposition von der Klasse zum Interace angezeigt.
2. Links in der ModelTreeView wird die neue Komposition unter der Klasse im firstDiagram angezeigt.

22.18.4 Komposition von einer Abstrakten Klasse zu einer Abstrakten Klasse

Dieser Test deckt das Erstellen einer Komposition von einer Abstrakten Klasse zu einer Abstrakten Klasse ab.

Vorbedingung

1. Es wurden 2 Abstrakte Klassen erstellt.

Test

1. Links im Editor-Bereich auf Composition klicken.
2. Auf die erste Abstrakte Klasse klicken.
3. Auf die zweite Abstrakte Klasse klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Komposition von der ersten zur zweiten Abstrakten Klasse angezeigt.
2. Links in der ModelTreeView wird die neue Komposition unter der ersten Abstrakten Klasse im firstDiagram angezeigt.

22.18.5 Komposition von einer Abstrakten Klasse zu einer Klasse

Dieser Test deckt das Erstellen einer Komposition von einer Abstrakten Klasse zu einer Klasse ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.
2. Es wurde 1 Abstrakte Klasse erstellt.
3. Es wurde 1 Klasse erstellt.

Test

1. Links im Editor-Bereich auf Composition klicken.
2. Auf die Abstrakte Klasse klicken.
3. Auf die Klasse klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Komposition von der Abstrakten Klasse zur Klasse angezeigt.
2. Links in der ModelTreeView wird die neue Komposition unter der Abstrakten Klasse im firstDiagram angezeigt.

22.18.6 Komposition von einer Abstrakten Klasse zu einem Interface

Dieser Test deckt das Erstellen einer Komposition von einer Abstrakten Klasse zu einem Interface ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.
2. Es wurde 1 Abstrakte Klasse erstellt.
3. Es wurde 1 Interface erstellt.

Test

1. Links im Editor-Bereich auf Composition klicken.
2. Auf die Abstrakte Klasse klicken.
3. Auf das Interface klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Komposition von der Abstrakten Klasse zum Interface angezeigt.

2. Links in der ModelTreeView wird die neue Komposition unter der Abstrakten Klasse im firstDiagram angezeigt.

22.18.7 Komposition von einem Interface zu einem Interface

Dieser Test deckt das Erstellen einer Komposition von einem Interface zu einem Interface ab.

Vorbedingung

1. Es wurden 2 Interfaces erstellt.

Test

1. Links im Editor-Bereich auf Composition klicken.

2. Auf das erste Interface klicken.

3. Auf das zweite Interface klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Komposition vom ersten zum zweiten Interface angezeigt.

2. Links in der ModelTreeView wird die neue Komposition unter dem ersten Interface im firstDiagram angezeigt.

22.18.8 Komposition von einem Interface zu einer Klasse

Dieser Test deckt das Erstellen einer Komposition von einem Interface zu einer Klasse ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.

2. Es wurde 1 Interface erstellt.

3. Es wurde 1 Klasse erstellt.

Test

1. Links im Editor-Bereich auf Composition klicken.
2. Auf das Interface klicken.
3. Auf die Klasse klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Komposition vom Interface zur Klasse angezeigt.
2. Links in der ModelTreeView wird die neue Komposition unter dem Interface im firstDiagram angezeigt.

22.18.9 Komposition von einem Interface zu einer Abstrakten Klasse

Dieser Test deckt das Erstellen einer Komposition von einem Interface zu einer Abstrakten Klasse ab.

Vorbedingung

1. Es wurde ein Modell geöffnet.
2. Es wurde 1 Interface erstellt.
3. Es wurde 1 Abstrakte Klasse erstellt.

Test

1. Links im Editor-Bereich auf Composition klicken.
2. Auf das Interface klicken.
3. Auf die Abstrakte Klasse klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird eine Komposition vom Interface zur Abstrakten Klasse angezeigt.
2. Links in der ModelTreeView wird die neue Komposition unter dem Interface im firstDiagram angezeigt.

22.18.10 Selektion einer Komposition

Dieser Test deckt das Löschen einer Komposition ab.

Vorbedingung

1. Es wurden 2 Klassen erstellt.
2. Es wurde 1 Komposition zwischen diesen erstellt.

Test

1. Im Editor-Bereich auf die Komposition klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche ist die Komposition hervorgehoben.

22.18.11 Löschen einer Komposition aus dem Diagramm

Dieser Test deckt das Löschen einer Komposition aus dem Diagramm ab.

Vorbedingung

1. Es wurden 2 Klassen erstellt.
2. Es wurde 1 Komposition zwischen diesen erstellt.
3. Die Komposition wurde selektiert.

Test

1. Entfernen drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die gelöschte Komposition nicht mehr angezeigt.
2. Links in der ModelTreeView wird die gelöschte Komposition noch angezeigt.

22.18.12 Löschen einer Komposition aus dem Modell

Dieser Test deckt das Löschen einer Komposition aus dem Modell ab.

Vorbedingung

1. Es wurden 2 Klassen erstellt.
2. Es wurde 1 Komposition zwischen diesen erstellt.

Test

1. Links in der ModelTreeView rechtsklick auf die Komposition.
2. „Delete Composition from model“ anklicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die gelöschte Komposition nicht mehr angezeigt.
2. Links in der ModelTreeView wird die gelöschte Komposition nicht mehr angezeigt.

22.18.13 Umhängen des Startpunktes einer Komposition

Dieser Test deckt das Umhängen des Startpunktes einer Generalisierung ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 3 Knoten erstellt.
2. Es wurde eine Kompositionen zwischen diesen erstellt.
3. Die Komposition wurde selektiert.

Test

1. Der Startpunkt der Komposition per Drag and Drop auf ein anderes Knoten verändern.

Erwartetes Ergebnis

1. Der Startpunkt der Komposition wird nur dann verändert, wenn die daraus resultierende Komposition gültig ist.

22.18.14 Umhängen des Endpunktes einer Komposition

Dieser Test deckt das Umhängen des Endpunktes einer Generalisierung ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 3 Knoten erstellt.
2. Es wurde eine Kompositionen zwischen diesen erstellt.
3. Die Komposition wurde selektiert.

Test

1. Der Endpunkt der Komposition per Drag and Drop auf ein anderes Knoten verändern.

Erwartetes Ergebnis

1. Der Endpunkt der Komposition wird nur dann verändert, wenn die daraus resultierende Komposition gültig ist.

22.18.15 Sichtbarkeit des Startpunktes einer Komposition aktivieren

Dieser Test deckt das Aktivieren der Sichtbarkeit des Startpunktes einer Komposition ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Komposition zwischen diesen erstellt.
3. Die Komposition wurde selektiert.

Test

1. Rechtsklick auf die Komposition.
2. Klick auf „Toggle source navigability“.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird ein Pfeil am Startpunkt der Komposition angezeigt.

22.18.16 Sichtbarkeit des Startpunktes einer Komposition deaktivieren

Dieser Test deckt das Aktivieren der Sichtbarkeit des Startpunktes einer Komposition ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Komposition zwischen diesen erstellt.
3. Die Sichtbarkeit des Startpunktes der Komposition wurde aktiviert.
4. Die Komposition wurde selektiert.

Test

1. Rechtsklick auf die Komposition.
2. Klick auf „Toggle source navigability“.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird kein Pfeil am Startpunkt der Komposition angezeigt.

22.18.17 Sichtbarkeit des Endpunktes einer Komposition aktivieren

Dieser Test deckt das Aktivieren der Sichtbarkeit des Endpunktes einer Komposition ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Komposition zwischen diesen erstellt.
3. Die Komposition wurde selektiert.

Test

1. Rechtsklick auf die Komposition.
2. Klick auf „Toggle source navigability“.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird ein Pfeil am Endpunkt der Komposition angezeigt.

22.18.18 Sichtbarkeit des Endpunktes einer Komposition deaktivieren

Dieser Test deckt das Aktivieren der Sichtbarkeit des Endpunktes einer Komposition ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Komposition zwischen diesen erstellt.
3. Die Sichtbarkeit des Endpunktes der Komposition wurde aktiviert.
4. Die Komposition wurde selektiert.

Test

1. Rechtsklick auf die Komposition.
2. Klick auf „Toggle source navigability“.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird kein Pfeil am Endpunkt der Komposition angezeigt.

22.18.19 Umwandlung einer Komposition zu einer Aggregation

Dieser Test deckt das Umwandeln einer Komposition zu einer Aggregation ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Komposition zwischen diesen erstellt.
3. Die Komposition wurde selektiert.

Test

1. Rechtsklick auf die Komposition.
2. Klick auf „Change to aggregation“.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird anstelle der Komposition eine Aggregation angezeigt.
2. Links in der ModelTreeView wird anstelle der Komposition eine Aggregation angezeigt.

22.18.20 Umwandlung einer Komposition zu einer Assoziation

Dieser Test deckt das Umwandeln einer Komposition zu einer Assoziation ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Komposition zwischen diesen erstellt.
3. Die Komposition wurde selektiert.

Test

1. Rechtsklick auf die Komposition.
2. Klick auf „Change to association“.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird anstelle der Komposition eine Assoziation angezeigt.
2. Links in der ModelTreeView wird anstelle der Komposition eine Assoziation angezeigt.

22.18.21 Erstellen eines Bendpoints in einer Komposition

Dieser Test deckt das Erstellen eines Bendpoints in einer Komposition ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Komposition zwischen diesen erstellt.
3. Die Komposition wurde selektiert.

Test

1. Klick auf den mittleren markierten Punkt auf der Komposition.
2. Verschieben des Bendpoints per Drag and Drop.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die Komposition über den Bendpoint umgeleitet angezeigt.

22.18.22 Verschieben eines Bendpoints in einer Komposition

Dieser Test deckt das Verschieben eines Bendpoints in einer Komposition ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Komposition zwischen diesen erstellt.
3. Es wurde an der Komposition ein Bendpoint erstellt.
4. Die Komposition wurde selektiert.

Test

1. Klick auf den Bendpoint auf der Komposition.
2. Verschieben des Bendpoints per Drag and Drop an eine Position, sodass die Komposition nicht gerade ist.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die Komposition über den verschobenen Bendpoint umgeleitet angezeigt.

22.18.23 Löschen eines Bendpoints in einer Komposition

Dieser Test deckt das Löschen eines Bendpoints in einer Komposition ab. Dieser Test ist generisch dokumentiert, da das Kreuzprodukt der Knoten-Kombinationen sehr groß ist und die Tests über dieses Kreuzprodukt fehlerfrei durchgeführt wurde.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Komposition zwischen diesen erstellt.
3. Es wurde an der Komposition ein Bendpoint erstellt.
4. Die Komposition wurde selektiert.

Test

1. Klick auf den Bendpoint auf der Komposition.
2. Verschieben des Bendpoints per Drag and Drop, sodass die Komposition wieder gerade ist.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird die Komposition ohne Bendpoint angezeigt.

22.19 Tests für Assoziationslabels

Die Tests in diesem Abschnitt decken das Interagieren und Bearbeiten von Assoziationslabels ab. Da diese Tests Fehlerfrei für Assoziationen, Aggregationen und Komposition durchgeführt wurden, sind diese nur für Assoziationen dokumentiert. Auch die verschiedenen Knotentypen brachten keine

Fehler hervor, was auf eine vollkommen analoge Abhandlung der Assoziationslabels hindeutet. Bei allen durchgeführten Tests gab es kein mal Unterschiede zwischen Quell- und Ziel-Rollenbezeichner. Ebenso gab es keine Unterschiede zwischen Quell- und Ziel-Multiplizitäten.

22.19.1 Selektion der Quellmultiplizität

Dieser Test deckt das Selektieren der Quellmultiplizität ab.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation erstellt.

Test

1. Im Editor-Bereich auf das „0“ Textfeld am Startpunkt der Assoziation klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue angeklickte Feld hervorgehoben angezeigt.

22.19.2 Selektion der Zielmultiplizität

Dieser Test deckt das Selektieren der Zielmultiplizität ab.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation erstellt.

Test

1. Im Editor-Bereich auf das „0“ Textfeld am Endpunkt der Assoziation klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue angeklickte Feld hervorgehoben angezeigt.

22.19.3 Selektion des Quell-Rollenbezeichners

Dieser Test deckt das Selektieren der Quell-Rollenbezeichners ab.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation erstellt.

Test

1. Im Editor-Bereich auf das „# null“ Textfeld am Startpunkt der Assoziation klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue angeklickte Feld hervorgehoben angezeigt.

22.19.4 Selektion des Ziel-Rollenbezeichners

Dieser Test deckt das Selektieren der Ziel-Rollenbezeichners ab.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation erstellt.

Test

1. Im Editor-Bereich auf das „# null“ Textfeld am Endpunkt der Assoziation klicken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue angeklickte Feld hervorgehoben angezeigt.

22.19.5 Selektion des Assoziationsbezeichners

Dieser Test deckt das Selektieren des Assoziationsbezeichners ab.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation erstellt.

Test

1. Im Editor-Bereich auf das „ “ Textfeld in der Mitte der Assoziation klicken (Mit einer hellen Nutzerfarbe schwer sichtbar).

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird das neue angeklickte Feld hervorgehoben angezeigt.

22.19.6 Bearbeiten des Assoziationsbezeichners

Dieser Test deckt das Bearbeiten des Assoziationsbezeichners ab.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation erstellt.
3. Es wurde der Assoziationsbezeichner selektiert.

Test

1. Im Editor-Bereich auf das selektierte Textfeld klicken.
2. „hallo“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird „hallo“ im selektierten Textfeld angezeigt.
2. In der ModelTreeView links wurde der Wert des Textfelds and der Assoziation aktualisiert.

22.19.7 Bearbeiten des Quell-Rollenbezeichners mit Default-Sichtbarkeit

Dieser Test deckt das Bearbeiten des Quell-Rollenbezeichners mit Default-Sichtbarkeit ab.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation erstellt.
3. Es wurde der Quell-Rollenbezeichner selektiert.

Test

1. Im Editor-Bereich auf das selektierte Textfeld klicken.
2. „hallo“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird „hallo“ im selektierten Textfeld angezeigt.
2. In der ModelTreeView links wurde der Wert des Textfelds and der Assoziation aktualisiert.
3. Die Sichtbarkeit des Rollenbezeichners wird in der ModelTreeView nicht angezeigt.

22.19.8 Bearbeiten des Quell-Rollenbezeichners mit Private-Sichtbarkeit

Dieser Test deckt das Bearbeiten des Quell-Rollenbezeichners mit Private-Sichtbarkeit ab.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation erstellt.
3. Es wurde der Quell-Rollenbezeichner selektiert.

Test

1. Im Editor-Bereich auf das selektierte Textfeld klicken.
2. „-hallo“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird „- hallo“ im selektierten Textfeld angezeigt.
2. In der ModelTreeView links wurde der Wert des Textfelds and der Assoziation aktualisiert.
3. Die Sichtbarkeit des Rollenbezeichners wird in der ModelTreeView nicht angezeigt.

22.19.9 Bearbeiten des Quell-Rollenbezeichners mit Package-Sichtbarkeit

Dieser Test deckt das Bearbeiten des Quell-Rollenbezeichners mit Package-Sichtbarkeit ab.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation erstellt.
3. Es wurde der Quell-Rollenbezeichner selektiert.

Test

1. Im Editor-Bereich auf das selektierte Textfeld klicken.
2. „#hallo“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird „# hallo“ im selektierten Textfeld angezeigt.
2. In der ModelTreeView links wurde der Wert des Textfelds and der Assoziation aktualisiert.
3. Die Sichtbarkeit des Rollenbezeichners wird in der ModelTreeView nicht angezeigt.

22.19.10 Bearbeiten des Quell-Rollenbezeichners mit Public-Sichtbarkeit

Dieser Test deckt das Bearbeiten des Quell-Rollenbezeichners mit Public-Sichtbarkeit ab.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation erstellt.
3. Es wurde der Quell-Rollenbezeichner selektiert.

Test

1. Im Editor-Bereich auf das selektierte Textfeld klicken.
2. „+hallo“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird „+ hallo“ im selektierten Textfeld angezeigt.
2. In der ModelTreeView links wurde der Wert des Textfelds and der Assoziation aktualisiert.
3. Die Sichtbarkeit des Rollenbezeichners wird in der ModelTreeView nicht angezeigt.

22.19.11 Bearbeiten des Quell-Rollenbezeichners mit Default-Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Bearbeiten des Quell-Rollenbezeichners mit Default-Sichtbarkeit mit Leerzeichen ab. Dieser Test wurde aufgrund des Fehlers beim Parsen der Sichtbarkeit bei Klassenattributen (Abschnitt 18.1 durchgeführt.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation erstellt.
3. Es wurde der Quell-Rollenbezeichner selektiert.

Test

1. Im Editor-Bereich auf das selektierte Textfeld klicken.
2. „ hallo“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird „hallo“ im selektierten Textfeld angezeigt.

2. In der ModelTreeView links wurde der Wert des Textfelds and der Assoziation aktualisiert.

3. Die Sichtbarkeit des Rollenbezeichners wird in der ModelTreeView nicht angezeigt.

22.19.12 Bearbeiten des Quell-Rollenbezeichners mit Private-Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Bearbeiten des Quell-Rollenbezeichners mit Private-Sichtbarkeit mit Leerzeichen ab. Dieser Test wurde aufgrund des Fehlers beim Parsen der Sichtbarkeit bei Klassenattributen (Abschnitt 18.1 durchgeführt.

Vorbedingung

1. Es wurden 2 Knoten erstellt.

2. Es wurde 1 Assoziation erstellt.

3. Es wurde der Quell-Rollenbezeichner selektiert.

Test

1. Im Editor-Bereich auf das selektierte Textfeld klicken.

2. „- hallo“ eingeben.

3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird „- hallo“ im selektierten Textfeld angezeigt.

2. In der ModelTreeView links wurde der Wert des Textfelds and der Assoziation aktualisiert.

3. Die Sichtbarkeit des Rollenbezeichners wird in der ModelTreeView nicht angezeigt.

22.19.13 Bearbeiten des Quell-Rollenbezeichners mit Package-Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Bearbeiten des Quell-Rollenbezeichners mit Package-Sichtbarkeit mit Leerzeichen ab. Dieser Test wurde aufgrund des Fehlers beim Parsen der Sichtbarkeit bei Klassenattributen (Abschnitt 18.1 durchgeführt.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation erstellt.
3. Es wurde der Quell-Rollenbezeichner selektiert.

Test

1. Im Editor-Bereich auf das selektierte Textfeld klicken.
2. „# hallo“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird „# hallo“ im selektierten Textfeld angezeigt.
2. In der ModelTreeView links wurde der Wert des Textfelds and der Assoziation aktualisiert.
3. Die Sichtbarkeit des Rollenbezeichners wird in der ModelTreeView nicht angezeigt.

22.19.14 Bearbeiten des Quell-Rollenbezeichners mit Public-Sichtbarkeit mit Leerzeichen

Dieser Test deckt das Bearbeiten des Quell-Rollenbezeichners mit Public-Sichtbarkeit mit Leerzeichen ab. Dieser Test wurde aufgrund des Fehlers beim Parsen der Sichtbarkeit bei Klassenattributen (Abschnitt 18.1 durchgeführt.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation erstellt.
3. Es wurde der Quell-Rollenbezeichner selektiert.

Test

1. Im Editor-Bereich auf das selektierte Textfeld klicken.
2. „+ hallo“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird „+ hallo“ im selektierten Textfeld angezeigt.
2. In der ModelTreeView links wurde der Wert des Textfelds and der Assoziation aktualisiert.
3. Die Sichtbarkeit des Rollenbezeichners wird in der ModelTreeView nicht angezeigt.

22.19.15 Setzen von 0 als Quell-Multiplizität

Dieser Test deckt das Setzen der Quell-Multiplizität auf 0 ab.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation erstellt.
3. Es wurde die Quell-Multiplizität selektiert.

Test

1. Im Editor-Bereich auf das selektierte Textfeld klicken.
2. „0“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird „0“ im selektierten Textfeld angezeigt.
2. In der ModelTreeView links wurde der Wert des Textfelds and der Assoziation aktualisiert.

22.19.16 Setzen von 1 als Quell-Multiplizität

Dieser Test deckt das Setzen der Quell-Multiplizität auf 1 ab.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation erstellt.
3. Es wurde die Quell-Multiplizität selektiert.

Test

1. Im Editor-Bereich auf das selektierte Textfeld klicken.
2. „1“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird „1“ im selektierten Textfeld angezeigt.
2. In der ModelTreeView links wurde der Wert des Textfelds and der Assoziation aktualisiert.

22.19.17 Setzen von 0..1 als Quell-Multiplizität

Dieser Test deckt das Setzen der Quell-Multiplizität auf 0..1 ab.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation erstellt.
3. Es wurde die Quell-Multiplizität selektiert.

Test

1. Im Editor-Bereich auf das selektierte Textfeld klicken.
2. „0..1“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird „0..1“ im selektierten Textfeld angezeigt.
2. In der ModelTreeView links wurde der Wert des Textfelds and der Assoziation aktualisiert.

22.19.18 Setzen von * als Quell-Multiplizität

Dieser Test deckt das Setzen der Quell-Multiplizität auf * ab.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation erstellt.
3. Es wurde die Quell-Multiplizität selektiert.

Test

1. Im Editor-Bereich auf das selektierte Textfeld klicken.
2. „*“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird „*“ im selektierten Textfeld angezeigt.
2. In der ModelTreeView links wurde der Wert des Textfelds and der Assoziation aktualisiert.

22.19.19 Setzen von 1..* als Quell-Multiplizität

Dieser Test deckt das Setzen der Quell-Multiplizität auf 1..* ab.

Vorbedingung

1. Es wurden 2 Knoten erstellt.
2. Es wurde 1 Assoziation erstellt.
3. Es wurde die Quell-Multiplizität selektiert.

Test

1. Im Editor-Bereich auf das selektierte Textfeld klicken.
2. „1..*“ eingeben.
3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird „1..*“ im selektierten Textfeld angezeigt.

2. In der ModelTreeView links wurde der Wert des Textfelds and der Assoziation aktualisiert.

22.19.20 Setzen von 25 als Quell-Multiplizität

Dieser Test deckt das Setzen der Quell-Multiplizität auf eine beliebige aber feste Zahl ab.

Vorbedingung

1. Es wurden 2 Knoten erstellt.

2. Es wurde 1 Assoziation erstellt.

3. Es wurde die Quell-Multiplizität selektiert.

Test

1. Im Editor-Bereich auf das selektierte Textfeld klicken.

2. „25“ eingeben.

3. Enter drücken.

Erwartetes Ergebnis

1. Im Editor-Bereich in der Zeichenfläche wird „25“ im selektierten Textfeld angezeigt.

2. In der ModelTreeView links wurde der Wert des Textfelds and der Assoziation aktualisiert.

22.20 Tests für Kollaboration

Die Tests in diesem Abschnitt decken die Kollaborativen Featues des Produktes ab.

22.20.1 Laden der Userliste beim beitreten

Dieser Test deckt das Aktualisieren der Userliste beim Betreten eines Models Benutzer ab.

Vorbedingung

1. Ein zweiter Nutzer ist in einem Modell.

Test

1. Der Nutzer tritt diesem Modell bei.

Erwartetes Ergebnis

1. In der Userliste werden jetzt beide Nutzer angezeigt.

22.20.2 Aktualisieren der Userliste wenn ein Benutzer dem Modell beitritt

Dieser Test deckt das Aktualisieren der Userliste beim Beitreten neuer Benutzer ab.

Vorbedingung

1. Es wurde ein Model geladen.
2. In diesem Modell ist kein Nutzer außer dem Tester.

Test

1. Ein zweiter Nutzer tritt dem Modell bei.

Erwartetes Ergebnis

1. In der Userliste werden jetzt zwei Nutzer angezeigt.

22.20.3 Aktualisieren der Userliste beim Modell wechseln - I

Dieser Test deckt das Aktualisieren der Userliste beim Auswählen eines Editors aus einem Modell mit mehr Nutzern ab.

Vorbedingung

1. Es wurden zwei Modelle geladen.
2. Von jedem Modell ist genau ein Editor geöffnet.
3. Im ersten Modell ist kein Nutzer außer dem Tester.
4. Im zweiten Modell ist auch noch ein weiterer Nutzer.
5. Der Editor des ersten Modells ist aktiv.

Test

1. Der Nutzer wechselt zum Editor des zweiten Modells.

Erwartetes Ergebnis

1. In der Userliste werden jetzt zwei Nutzer angezeigt.

22.20.4 Aktualisieren der Userliste beim Modell wechseln - II

Dieser Test deckt das Aktualisieren der Userliste beim Auswählen eines Editors eines Modells mit weniger Nutzern ab.

Vorbedingung

1. Es wurden zwei Modelle geladen.
2. Von jedem Modell ist genau ein Editor geöffnet.
3. Im ersten Modell ist kein Nutzer außer dem Tester.
4. Im zweiten Modell ist auch noch ein weiterer Nutzer.
5. Der Editor des zweiten Modells ist aktiv.

Test

1. Der Nutzer wechselt zum Editor des ersten Modells.

Erwartetes Ergebnis

1. In der Userliste wird jetzt ein Nutzer angezeigt.

22.20.5 Aktualisieren der Userliste bei Namesänderung

Dieser Test deckt das Aktualisieren der Userliste bei Namensänderung eines Nutzers ab.

Vorbedingung

1. Es wurde ein Modell geladen.
2. Zu diesem Modell ist genau ein Editor geöffnet.
3. Im diesem Modell befinden sich zwei Benutzer.

Test

1. Der Nutzer ändert seinen Namen.

Erwartetes Ergebnis

1. In der Userliste beider Clients wird der neue Name des Benutzers angezeigt.

22.20.6 Aktualisieren der Userliste bei Farbänderung

Dieser Test deckt das Aktualisieren der Userliste bei Farbänderung eines Nutzers ab.

Vorbedingung

1. Es wurde ein Modell geladen.
2. Zu diesem Modell ist genau ein Editor geöffnet.
3. Im diesem Modell befinden sich zwei Benutzer.

Test

1. Der Nutzer ändert seine Farbe.

Erwartetes Ergebnis

1. In der Userliste beider Clients wird die neue Farbe des Benutzers angezeigt.

22.20.7 Synchronisierung von Modelldifferenzen

Dieser Test deckt das Synchronisierung der Modelldifferenzen bei Nutzeränderungen ab. Da für die Differenzerzeugung eine Metamodellgenerische Drittsoftware verwendet wird, und die Umsetzung von Nutzereingaben in Modelländerungen in den vorangehenden Akzeptanztest ausführlich durchgeführt wurden, wird hier nur ein einziger Test durchgeführt.

Vorbedingung

1. Es wurde ein Modell geladen.
2. Zu diesem Modell ist genau ein Editor geöffnet.
3. Im diesem Modell befinden sich zwei Benutzer.

Test

1. Der Nutzer führt eine Modelländerung durch.

Erwartetes Ergebnis

1. Das versandte Delta entspricht dem empfangen Delta. Dies kann mittels der LogView überprüft werden.

22.20.8 Farbe löschen wenn alle Mitbenutzer zustimmen

Dieser Test stellt sicher das die Farben aus einem Mehrbenutzermodell nur bei einstimmiger Zustimmung gelöscht werden.

Vorbedingung

1. Es wurde ein Modell geladen.
2. Zu diesem Modell ist genau ein Editor geöffnet.
3. Im diesem Modell befinden sich zwei Benutzer.
4. Im Model ist von jedem Benutzer ein Element erzeugt worden.

Test

1. Der Nutzer klickt auf „Clear Colors“.
2. Der andere Nutzer stimmt der Farblöschung zu.

Erwartetes Ergebnis

1. Die Elemente werden wieder in weiß dargestellt.

22.20.9 Farbe nicht löschen wenn nicht alle Mitbenutzer zustimmen

Dieser Test stellt sicher das die Farben aus einem Mehrbenutzermodell nur bei einstimmiger Zustimmung gelöscht werden.

Vorbedingung

1. Es wurde ein Modell geladen.
2. Zu diesem Modell ist genau ein Editor geöffnet.
3. Im diesem Modell befinden sich zwei Benutzer.
4. Im Model ist von jedem Benutzer ein Element erzeugt worden.

Test

1. Der Nutzer klickt auf „Clear Colors“.
2. Der andere Nutzer lehnt die Farblöschung ab.

Erwartetes Ergebnis

1. Die Elemente werden weiterhin farbig dargestellt.
2. Beide Nutzer erhalten eine Nachricht, dass die Farblöschung abgelehnt wurde.

22.20.10 Version zurücksetzen nur wenn alle Mitbenutzer zugestimmt haben

Dieser Test stellt sicher, dass ein Mehrbenutzermodell nur bei einstimmiger Zustimmung auf eine frühere Version zurückgesetzt werden kann.

Vorbedingung

1. Es wurde ein Modell geladen.
2. Zu diesem Modell ist genau ein Editor geöffnet.
3. Im diesem Modell befinden sich zwei Benutzer.
4. Vom Model ist eine zweite Version gespeichert.
5. Gegenüber der zweiten Version enthält HEAD eine Änderung.

Test

1. Der Nutzer wählt die zweite Version aus der Combobox aus.
2. Der Nutzer rechtsklickt auf ClassDiagram und wählt „Revert HEAD to this version“.
3. Der andere Nutzer stimmt dem Revert zu.

Erwartetes Ergebnis

1. Die Änderung in HEAD gegenüber der zweiten Version verschwindet und HEAD ist gleich Version zwei.

22.20.11 Version nicht zurücksetzen wenn nicht alle Mitbenutzer zugestimmt haben

Dieser Test stellt sicher, dass ein Mehrbenutzermodell nur bei einstimmiger Zustimmung auf eine frühere Version zurückgesetzt werden kann.

Vorbedingung

1. Es wurde ein Modell geladen.
2. Zu diesem Modell ist genau ein Editor geöffnet.
3. Im diesem Modell befinden sich zwei Benutzer.
4. Vom Model ist eine zweite Version gespeichert.
5. Gegenüber der zweiten Version enthält HEAD eine Änderung.

Test

1. Der Nutzer wählt die zweite Version aus der Combobox aus.
2. Der Nutzer rechtsklickt auf ClassDiagram und wählt „Revert HEAD to this version“.
3. Der andere Nutzer lehnt den Revert ab.

Erwartetes Ergebnis

1. Die Änderung in HEAD gegenüber der zweiten Version bleibt erhalten.
2. Beide Benutzer erhalten eine Nachricht, dass der Revert abgelehnt wurde.

Teil VI

Softwarelizenzen

In diesem Teil werden die Software Lizenzen, der im Projekt verwendeten Bibliotheken aufgelistet. Dies ist notwendig, weil alle Lizenzen fordern eine Kopie der Lizenz mit dem Produkt auszuliefern.

Kapitel 23

Apache 2.0 Lizenz

Im Projekt Kotelett werden die Google Guava und Apache Commons IO Bibliotheken verwendet. Diese stehen unter der Apache 2.0 Lizenz.

Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to

communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor,

Kapitel 24

BSD Lizenz

Die im Projekt Kotelett verwendete Bibliothek KryoNet steht unter einer BSD-artigen Lizenz.

```
Copyright (c) 2008, Nathan Sweet
All rights reserved.
Redistribution and use in source and binary forms, with or without modification,
     are permitted provided that the following conditions are met:
* Redistributions of source code must retain the above copyright notice, this
     list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above copyright notice, this
     list of conditions and the following disclaimer in the documentation and/or
     other materials provided with the distribution.
* Neither the name of Esoteric Software nor the names of its contributors may be
     used to endorse or promote products derived from this software without
     specific prior written permission.
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
     ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
     WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
     DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
     FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
     DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
     SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
     CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
     OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE
     USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

Kapitel 25

Eclipse Public License

Eclipse, sowie SWT und GEF stehen unter Eclipse Public License.

```
Eclipse Public License - v 1.0

THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS ECLIPSE PUBLIC
     LICENSE ("AGREEMENT"). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM
     CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.

1. DEFINITIONS

"Contribution" means:

a) in the case of the initial Contributor, the initial code and documentation
     distributed under this Agreement, and

b) in the case of each subsequent Contributor:

i) changes to the Program, and

ii) additions to the Program;

where such changes and/or additions to the Program originate from and are
     distributed by that particular Contributor. A Contribution 'originates' from
     a Contributor if it was added to the Program by such Contributor itself or
     anyone acting on such Contributor's behalf. Contributions do not include
     additions to the Program which: (i) are separate modules of software
     distributed in conjunction with the Program under their own license agreement
     , and (ii) are not derivative works of the Program.

"Contributor" means any person or entity that distributes the Program.

"Licensed Patents" mean patent claims licensable by a Contributor which are
     necessarily infringed by the use or sale of its Contribution alone or when
     combined with the Program.

"Program" means the Contributions distributed in accordance with this Agreement.

"Recipient" means anyone who receives the Program under this Agreement, including
      all Contributors.

2. GRANT OF RIGHTS

a) Subject to the terms of this Agreement, each Contributor hereby grants
     Recipient a non-exclusive, worldwide, royalty-free copyright license to
     reproduce, prepare derivative works of, publicly display, publicly perform,
     distribute and sublicense the Contribution of such Contributor, if any, and
     such derivative works, in source code and object code form.

b) Subject to the terms of this Agreement, each Contributor hereby grants
     Recipient a non-exclusive, worldwide, royalty-free patent license under
     Licensed Patents to make, use, sell, offer to sell, import and otherwise
     transfer the Contribution of such Contributor, if any, in source code and
     object code form. This patent license shall apply to the combination of the
     Contribution and the Program if, at the time the Contribution is added by the
      Contributor, such addition of the Contribution causes such combination to be
      covered by the Licensed Patents. The patent license shall not apply to any
     other combinations which include the Contribution. No hardware per se is
     licensed hereunder.
```

c) Recipient understands that although each Contributor grants the licenses to its Contributions set forth herein, no assurances are provided by any Contributor that the Program does not infringe the patent or other intellectual property rights of any other entity. Each Contributor disclaims any liability to Recipient for claims brought by any other entity based on infringement of intellectual property rights or otherwise. As a condition to exercising the rights and licenses granted hereunder, each Recipient hereby assumes sole responsibility to secure any other intellectual property rights needed, if any. For example, if a third party patent license is required to allow Recipient to distribute the Program, it is Recipient's responsibility to acquire that license before distributing the Program.

d) Each Contributor represents that to its knowledge it has sufficient copyright rights in its Contribution, if any, to grant the copyright license set forth in this Agreement.

3. REQUIREMENTS

A Contributor may choose to distribute the Program in object code form under its own license agreement, provided that:

a) it complies with the terms and conditions of this Agreement; and

b) its license agreement:

i) effectively disclaims on behalf of all Contributors all warranties and conditions, express and implied, including warranties or conditions of title and non-infringement, and implied warranties or conditions of merchantability and fitness for a particular purpose;

ii) effectively excludes on behalf of all Contributors all liability for damages, including direct, indirect, special, incidental and consequential damages, such as lost profits;

iii) states that any provisions which differ from this Agreement are offered by that Contributor alone and not by any other party; and

iv) states that source code for the Program is available from such Contributor, and informs licensees how to obtain it in a reasonable manner on or through a medium customarily used for software exchange.

When the Program is made available in source code form:

a) it must be made available under this Agreement; and

b) a copy of this Agreement must be included with each copy of the Program.

Contributors may not remove or alter any copyright notices contained within the Program.

Each Contributor must identify itself as the originator of its Contribution, if any, in a manner that reasonably allows subsequent Recipients to identify the originator of the Contribution.

4. COMMERCIAL DISTRIBUTION

Commercial distributors of software may accept certain responsibilities with respect to end users, business partners and the like. While this license is intended to facilitate the commercial use of the Program, the Contributor who includes the Program in a commercial product offering should do so in a manner which does not create potential liability for other Contributors. Therefore, if a Contributor includes the Program in a commercial product offering, such Contributor ("Commercial Contributor") hereby agrees to defend and indemnify every other Contributor ("Indemnified Contributor") against any losses, damages and costs (collectively "Losses") arising from claims, lawsuits and other legal actions brought by a third party against the Indemnified Contributor to the extent caused by the acts or omissions of such Commercial Contributor in connection with its distribution of the Program in a commercial product offering. The obligations in this section do not apply to any claims or Losses relating to any actual or alleged intellectual property infringement. In order to qualify, an Indemnified Contributor must: a) promptly notify the Commercial Contributor in writing of such claim, and b) allow the Commercial Contributor to control, and cooperate with the Commercial Contributor in, the defense and any related settlement negotiations. The Indemnified Contributor may participate in any such claim at its own expense.

For example, a Contributor might include the Program in a commercial product offering, Product X. That Contributor is then a Commercial Contributor. If that Commercial Contributor then makes performance claims, or offers warranties related to Product X, those performance claims and warranties are such Commercial Contributor's responsibility alone. Under this section, the Commercial Contributor would have to defend claims against the other

Contributors related to those performance claims and warranties, and if a court requires any other Contributor to pay any damages as a result, the Commercial Contributor must pay those damages.

5. NO WARRANTY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, THE PROGRAM IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Each Recipient is solely responsible for determining the appropriateness of using and distributing the Program and assumes all risks associated with its exercise of rights under this Agreement, including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and unavailability or interruption of operations.

6. DISCLAIMER OF LIABILITY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, NEITHER RECIPIENT NOR ANY CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE PROGRAM OR THE EXERCISE OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. GENERAL

If any provision of this Agreement is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this Agreement, and without further action by the parties hereto, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

If Recipient institutes patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Program itself (excluding combinations of the Program with other software or hardware) infringes such Recipient's patent(s), then such Recipient's rights granted under Section 2(b) shall terminate as of the date such litigation is filed.

All Recipient's rights under this Agreement shall terminate if it fails to comply with any of the material terms or conditions of this Agreement and does not cure such failure in a reasonable period of time after becoming aware of such noncompliance. If all Recipient's rights under this Agreement terminate, Recipient agrees to cease use and distribution of the Program as soon as reasonably practicable. However, Recipient's obligations under this Agreement and any licenses granted by Recipient relating to the Program shall continue and survive.

Everyone is permitted to copy and distribute copies of this Agreement, but in order to avoid inconsistency the Agreement is copyrighted and may only be modified in the following manner. The Agreement Steward reserves the right to publish new versions (including revisions) of this Agreement from time to time. No one other than the Agreement Steward has the right to modify this Agreement. The Eclipse Foundation is the initial Agreement Steward. The Eclipse Foundation may assign the responsibility to serve as the Agreement Steward to a suitable separate entity. Each new version of the Agreement will be given a distinguishing version number. The Program (including Contributions) may always be distributed subject to the version of the Agreement under which it was received. In addition, after a new version of the Agreement is published, Contributor may elect to distribute the Program (including its Contributions) under the new version. Except as expressly stated in Sections 2(a) and 2(b) above, Recipient receives no rights or licenses to the intellectual property of any Contributor under this Agreement, whether expressly, by implication, estoppel or otherwise. All rights in the Program not expressly granted under this Agreement are reserved.

This Agreement is governed by the laws of the State of New York and the intellectual property laws of the United States of America. No party to this Agreement will bring a legal action under this Agreement more than one year after the cause of action arose. Each party waives its rights to a jury trial in any resulting litigation.

Kapitel 26

GNU General Public License Version 3

In Kotelett wird das Java Graphenlabor (jGralab) verwendet, welches unter der GPLv3 mit einem zusätzlichen Artikel steht.

```
  JGraLab - The Java Graph Laboratory
Copyright (C) 2006-2014 Institute for Software Technology
University of Koblenz-Landau, Germany
ist@uni-koblenz.de
For bug reports, documentation and further information, visit
https://github.com/jgralab/jgralab
This program is free software; you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by the
Free Software Foundation; either version 3 of the License, or (at your
option) any later version.
This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General
Public License for more details.
You should have received a copy of the GNU General Public License along
with this program; if not, see <http://www.gnu.org/licenses>.
Additional permission under GNU GPL version 3 section 7
If you modify this Program, or any covered work, by linking or combining
it with Eclipse (or a modified version of that program or an Eclipse
plugin), containing parts covered by the terms of the Eclipse Public
License (EPL), the licensors of this Program grant you additional
permission to convey the resulting work. Corresponding Source for a
non-source form of such a combination shall include the source code for
the parts of JGraLab used as well as that of the covered work.


                    GNU GENERAL PUBLIC LICENSE
                       Version 3, 29 June 2007

 Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble

  The GNU General Public License is a free, copyleft license for
software and other kinds of works.

  The licenses for most software and other practical works are designed
to take away your freedom to share and change the works.  By contrast,
the GNU General Public License is intended to guarantee your freedom to
share and change all versions of a program--to make sure it remains free
software for all its users.  We, the Free Software Foundation, use the
GNU General Public License for most of our software; it applies also to
any other work released this way by its authors.  You can apply it to
your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
them if you wish), that you receive source code or can get it if you
```

want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a

menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no

further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you

add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory

patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

Kapitel 27

JDOM Lizenz

Die von DOLSA verwendete JDOM Bibliothek steht unter einer Apache ähnlichen Lizenz.

```
$Id: LICENSE.txt,v 1.11 2004/02/06 09:32:57 jhunter Exp $

Copyright (C) 2000-2004 Jason Hunter & Brett McLaughlin.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions, and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions, and the disclaimer that follows
   these conditions in the documentation and/or other materials
   provided with the distribution.

3. The name "JDOM" must not be used to endorse or promote products
   derived from this software without prior written permission.  For
   written permission, please contact <request_AT_jdom_DOT_org>.

4. Products derived from this software may not be called "JDOM", nor
   may "JDOM" appear in their name, without prior written permission
   from the JDOM Project Management <request_AT_jdom_DOT_org>.

In addition, we request (but do not require) that you include in the
end-user documentation provided with the redistribution and/or in the
software itself an acknowledgement equivalent to the following:
    "This product includes software developed by the
     JDOM Project (http://www.jdom.org/)."
Alternatively, the acknowledgment may be graphical using the logos
available at http://www.jdom.org/images/logos.

THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED
WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED.  IN NO EVENT SHALL THE JDOM AUTHORS OR THE PROJECT
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.

This software consists of voluntary contributions made by many
individuals on behalf of the JDOM Project and was originally
created by Jason Hunter <jhunter_AT_jdom_DOT_org> and
Brett McLaughlin <brett_AT_jdom_DOT_org>.  For more information
on the JDOM Project, please see <http://www.jdom.org/>.
```

Literaturverzeichnis

[Bla14] Black Duck Software Inc.: *OpenHub - Compare Repositories*. https://www.openhub.net/repositories/compare. Version: 2014, Abruf: 2014-11-07

[Ecl13] Eclipse contributors: *Platform Plug-in Developer Guide*. 2013. – Eclipse Hilfe

[Ecl14] Eclipse Foundation: *GEF Programmer's Guide*. http://help.eclipse.org/luna/topic/org.eclipse.gef.doc.isv/guide/guide.html. Version: 2014, Abruf: 2015-01-25

[Ecl15] Eclipse Foundation: *The SWT FAQ*. http://www.eclipse.org/swt/faq.php. Version: 2015, Abruf: 2015-03-11

[EL11] Elaasar, Maged ; Labiche, Yvan: Diagram Definition: A Case Study with the UML Class Diagram. Version: 2011. http://dx.doi.org/10.1007/978-3-642-24485-8_26. In: Whittle, Jon (Hrsg.) ; Clark, Tony (Hrsg.) ; Kühne, Thomas (Hrsg.): *Model Driven Engineering Languages and Systems* Bd. 6981. Springer Berlin Heidelberg, 2011. – DOI 10.1007/978–3–642–24485–8_26. – ISBN 978–3–642–24484–1, 364-378

[Git14] Git: *Git - git-diff Documentation*. http://git-scm.com/docs/git-diff. Version: 2014, Abruf: 2014-11-07

[IBM14] IBM: *IBM Rational Software Architect*. http://www-03.ibm.com/software/products/en/ratisoftarch. Version: 2014, Abruf: 2014-06-11

[Kah06] Kahle, Steffen: *JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen*, Universität Koblenz-Landau, Institut für Softwaretechnik, Diplomarbeit, 2006

[Kry14] KryoNet: *KryoNet, TCP/UDP Client/Server library for Java*. https://github.com/EsotericSoftware/kryonet. Version: 2014, Abruf: 2014-11-16

[Kur14] Kuryazov, Dilshodbek: Delta Operations Language for Model Difference Representation. In: Plödereder, Erhard (Hrsg.) ; Grunske, Lars (Hrsg.) ; Schneider Ull, Eric D. (Hrsg.): *44. Jahrestagung der Gesellschaft für Informatik e.V. (GI), ISBN 978-3-88579-626-8* Bd. 232. Stuttgart, Germany : Gesellschaft für Informatik, 22-26 September 2014, S. 2221–2232

[KW14] Kuryazov, Dilshodbek ; Winter, Andreas: *Software Engineering Group - Publications*. `http://www.se.uni-oldenburg.de/60716.html`. Version: 2014

[MFBC12] Muller, Pierre-Alain ; Fondement, Frédéric ; Baudry, Benoît ; Combemale, Benoît: Modeling modeling modeling. In: *Software & Systems Modeling* 11 (2012), Nr. 3, 347-359. `http://dx.doi.org/10.1007/s10270-010-0172-x`. – DOI 10.1007/s10270–010–0172–x. – ISSN 1619–1366

[MLA10] McAffer, Jeff (Hrsg.) ; Lemieux, Jean-Michel (Hrsg.) ; Aniszcyk, Chris (Hrsg.): *Eclipse Rich Client Platform*. 2nd Edition. Addison-Wesley, 2010

[Nul14] Nulab Inc.: *Cacoo: Online Diagram Software for Flow Chart & UML and More*. `https://cacoo.com/`. Version: 2014, Abruf: 2014-06-11

[Obj11] Object Managment Group: *OMG Unified Modelung Lanugage^{TM}(OMG UML), Superstructure. Version 2.4.1*. `http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/`. Version: 2011, Abruf: 2014-06-11

[Ora14] Oracle Corporation: *Java Compiler Compiler (JavaCC) - The Java Parser Generator*. `https://java.net/projects/javacc`. Version: 2014, Abruf: 2015-03-16

[PR11] Pohl, K. ; Rupp, C.: *Requirements Engineering Fundamentals: A Study Guide for the Certified Professional for Requirements Engineering Exam - Foundation Level - IREB compliant*. Rocky Nook Computing, 2011

[RWC11] Rubel, Dan ; Wren, Jaime ; Clayberg, Eric: *The Eclipse Graphical Editor Framework (GEF)*. Addison Wesley, 2011

[WLS+13] Wieland, Konrad ; Langer, Philip ; Seidl, Martina ; Wimmer, Manuel ; Kappel, Gerti: Turning Conflicts into Collaboration. In: *Computer Supported Cooperative Work (CSCW)* 22 (2013), Nr. 2, S. 181–240