Universität Siegen

Fachbereich 12: Elektrotechnik und Informatik
Fachgruppe Praktische Informatik

Diplomarbeit DII Technische Informatik

Entwicklung von Differenzmetriken und deren Visualisierung

zur Erlangung des akademischen Grades
Diplom-Ingenieur der Technischen Informatik

Siegen, im Mai 2007

Diplomand : Jens Falk, Matrikel-Nr. 538365
1. Prüfer : Prof. Dr. Udo Kelter
2. Prüfer : Dipl.-Inform. Sven Wenzel

„Count what is countable. Measure what is measurable.
And what is not measurable, make measurable.“

— Galileo Galilei

Zusammenfassung

In dieser Diplomarbeit wird ein Werkzeug vorgestellt, mit dem der Anwender in der Lage ist, die Unterschiede zwischen strukturierten, technischen Dokumenten auf die Relevanz der Veränderungen hin zu analysieren. Es genügt damit den Anforderungen an die moderne modellgetriebene Systementwicklung.

Die Anwendung basiert auf der Berechnung von Metriken auf den Modelldifferenzen und deren Visualisierung mittels polymetrischer Sichten. Die beiden Techniken werden bereits in unterschiedlichen Bereichen der Softwaretechnik eingesetzt und wurden hier erfolgreich miteinander kombiniert.

Konkret werden eine Reihe von Definitionen für polymetrische Sichten für die Analyse der Differenzen zwischen UML-Klassendiagrammen vorgestellt, um mit deren Hilfe die gewünschten Änderungsinformationen zu erhalten.

Weiterhin wird Wert darauf gelegt, dass sich die Analysemöglichkeiten nicht nur auf diesen einzelnen Diagrammtyp beschränken. Alle Werkzeuge besitzen entsprechende Konfigurationsmöglichkeiten oder Schnittstellen, um die Fähigkeiten auch für andere Dokumenttypen, wie zum Beispiel Matlab/Simulink-Diagramme zur Verfügung zu stellen.

Danksagung

Ich möchte an dieser Stelle einigen Personen nennen, ohne die diese Arbeit so niemals hätte entstehen können.

> Allen voran meine Eltern Karin und Konrad, die in meinem ganzen Leben immer unterstützend hinter mir standen.
>
> Meine Kommilitonen Falk Bauer und Matthew Smith, mit denen ich im Studium viel erleben durfte.
>
> Kerstin, Thomas und Martin, mit denen mich mehr verbindet als unser ehemaliger Nebenjob.
>
> Mein Betreuer Sven, der mir durch seine Unterstützung gerade in der letzten Phase eine große Hilfe war.
>
> Mein Laufpartner Michael – und natürlich Marion, Mira und Maja! – der mich auch zuletzt noch immer auf Trab halten konnte.
>
> Alle Helfer, die mir durch ihre Anmerkungen zu dieser Arbeit weitergeholfen haben.

Euch allen sende ich hiermit ein herzliches Dankeschön und möge die Zukunft nur Positives für euch bereithalten!

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Neunkirchen, 14. Mai 2007 _____

Jens Falk

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1.

Einleitung und Motivation

In den vergangenen Jahren ist im Bereich der Softwareentwicklung ein neuer Trend hin zur modellgetriebenen Entwicklung auszumachen. Bedingt durch die weiterhin ständig wachsenden Anforderungen an die Entwicklung von Software in Bezug auf deren Komplexität – was zu mehr und mehr Teamarbeit führt – werden Dokumente benötigt, die über die reinen Quelltextdateien der zu entwickelnden Programme hinausgehen. Diese technischen Dokumente enthalten abstraktere Modelle der Software und ermöglichen dadurch unter anderem eine bessere Kommunikation zwischen den Teammitgliedern. Ein Beispiel für Dokumente dieser Art sind Klassendiagramme nach UML-Spezifikation [Obj03], die in der Entwurfsphase einer Software zum Einsatz kommen.

Nicht nur in der Softwareentwicklung, sondern auch in anderen Berufsfeldern existieren diese abstrahierenden Dokumente. Für die Modellierung von Steuerungssystemen etwa können Simulink-Diagramme benutzt werden und im Bereich der Fahrzeugentwicklung oder der Architektur sind das die schon seit langer Zeit benutzten CAD-Modelle.

Darüber hinaus ist für eine erfolgreiche Durchführung von Projekten wichtig, dass über den gesamten Entwicklungszeitraum hinweg, unterschiedliche Versionen der Dokumente verwaltet werden. Gerade für das Projektmanagement ist es notwendig, die Veränderungen zwischen den einzelnen Revisionen zu beobachten, um zum Beispiel den Projektfortschritt dokumentieren zu können. Um die Versionsunterschiede zu erhalten, müssen jedoch zunächst zwischen den Dokumenten die Differenzen ermittelt werden.

Die Erstellung von Differenzen zwischen reinen Textdokumenten ist aus heutiger Sicht recht einfach, da sie lediglich aus einer Aneinanderreihung von Textzeilen mit jeweils einer Menge von Zeichen bestehen. Es gibt eine Reihe von ausgereiften Werkzeugen, wie zum Beispiel *diff* für UNIX-basierte Betriebssysteme, die diese Arbeit erledigen. Die Vergleichswerkzeuge arbeiten häufig auf Kommandozeilenebene, werden aber ebenso in mächtigere grafische Anwendungen eingebunden, um den Arbeitsablauf in Entwicklungswerkzeugen zu optimieren.

Gegenüber dem textbasierten Vergleich ist die Differenzbildung von strukturierten Dokumenten wesentlich komplexer. Die Modelle, die darin beschrieben sind,

bestehen aus vielen Elementen unterschiedlicher Typen, die ineinander hierarchisch verschachtelt sein können. Hinzu kommt, dass zwischen den unterschiedlichen Elementtypen weitere Beziehungen unterschiedlicher Arten bestehen können. So befindet sich eine Klasse eines Klassendiagramms üblicherweise innerhalb eines Pakets, welches mehrere Klassen zusammenfasst. Gleichzeitig kann die Klasse bei Bedarf noch Vererbungsbeziehungen und Assoziationsbeziehungen mit anderen Klassen eingehen, die aber nicht zwingend im gleichen Paket liegen müssen.

Um die Differenz zwischen dieser Kategorie von Dokumenten zu berechnen, müssen zunächst die miteinander korrespondierenden Elemente der zu vergleichenden Dokumente gefunden werden. Erst anschließend können die genauen Unterschiede zwischen den Partnerelementen bestimmt werden. Im Rahmen des Projekts **SiDiff** [Uni07b] der Fachgruppe Praktische Informatik an der Universität Siegen, wurde bereits ein geeignetes Werkzeug entwickelt, das in der Lage ist, diese Aufgaben zu übernehmen.

Im Anschluss an die Berechnung der Modelldifferenz muss diese in einer geeigneten Form dargestellt werden. Als Alternativen gibt es die textuelle Darstellung der Unterschiede in Listenform oder auch verschiedene grafische Darstellungsarten.

Problemstellung

In allen Fällen der Darstellung ergibt sich das Problem, dass je größer die beteiligten Modelle in den zu vergleichenden Dokumenten und je umfangreicher die Änderungen zwischen den Modellen sind, desto unübersichtlicher wird deren Darstellung. Erschwerend kommt hinzu, dass eine einfache Auflistung der Änderungen vollkommen wertungsfrei ist.

Das macht zum Beispiel die Kontrolle von Entwicklungsprozessen schwierig, da die reine Anzahl der Änderungen zwischen den Modellen keine Aussagekraft hat. Fragen wie „Wie stark hat sich das Modell gegenüber der letzten Version geändert?“ oder „Wo fanden essentielle Änderungen im Modell statt?“ werden also mit wachsender Modellgröße immer schwieriger.

Damit ein Anwender Aussagen zur Qualität der Änderungen treffen kann, ist ein Gesamtüberblick über alle Änderungen notwendig. Dennoch benötigt er eine einfachere Möglichkeit, um an Informationen über die Art der Veränderungen zu gelangen, als jede Unterscheidung Schritt für Schritt einzeln zu betrachten.

Das Ziel dieser Arbeit soll es daher sein, zunächst abstrakte Daten aus den Modelldifferenzen zu gewinnen und diese in eine geeignete Darstellungsform zu bringen, die für einen Anwender möglichst intuitiv verständlich ist. Dazu werden mittels eigens entwickelter Differenzmetriken sogenannte polymetrische Sichten auf den Differenzen gebildet. Diese Diagrammform erlaubt dem Anwender, Schlussfolgerungen bezüglich der Qualität der Änderungen zu ziehen.

Die Brauchbarkeit dieser Vorgehensweise soll anschließend anhand von Sichtdefinitionen für UML-Klassendiagramme und geeigneten Testdaten, die im Verlauf der

Arbeit entstanden, bewertet werden.

Aufbau der Arbeit

Nach dieser kurzen Einleitung werden zunächst in Kapitel 2 die Grundlagen über die Berechnung und Darstellung von Differenzen vermittelt. Desweiteren werden bereits existierende Ansätze zur Gewinnung von Daten aus Softwaremodellen und die Bewertung von Veränderungen an Softwaresystemen vorgestellt. Kapitel 3 beschreibt die Konzeption und Implementierung eines Werkzeugs, welches eine Verbindung herstellt zwischen zwei Werkzeugen für die Berechnung von Modelldifferenzen und für die abstrakte Darstellung von berechneten Metriken. Das nachfolgende Kapitel 4 beinhaltet eine Reihe von Sichtdefinitionen für polymetrische Sichten auf Basis von Klassendiagrammen. Diese wurden während der Erstellung der Arbeit anhand von verschiedenen Testdaten in der Praxis entwickelt und dienen der Analyse der Differenzen. Danach wird in Kapitel 5 beschrieben, welche Schritte notwendig sind, um mit den Werkzeugen andere Diagrammtypen analysieren zu können. Abschließend erfolgt in Kapitel 6 eine Zusammenfassung der Arbeit mit einer Reihe von Vorschlägen für mögliche Weiterentwicklungen der Werkzeuge.

Kapitel 2.

Modelldifferenzen und deren Interpretation

Dieses Kapitel beschreibt, welche Möglichkeiten es gibt, Dokumente verschiedener Revisionen miteinander zu vergleichen. Der Unterschied zwischen diesen Dokumenten wird als sogenannte **Differenz** bezeichnet, die für sich alleine gesehen vollkommen wertungsfrei ist. Aussagen über die Stärke oder die Wichtigkeit der Änderungen können also zunächst nicht getroffen werden.

Es sollen daher nach einer kurzen Begriffsbestimmung, mögliche Ansätze betrachtet werden, wie mit aktuellen Werkzeugen Dokumentdifferenzen gebildet und dargestellt werden. Desweiteren werden Werkzeuge vorgestellt, die der Gewinnung von Daten aus Softwaresystemen dienen und wie aus der abstrakten Darstellung von Modellen mittels Metriken weiterführende Informationen gewonnen werden können.

Aus diesen vorgestellten Lösungen ergibt sich eine potenzielle Herangehensweise an die qualitative Analyse von Dokumentdifferenzen, die anschließend weiter verfolgt werden soll.

2.1. Was sind Differenzen?

Eine allgemeine, informelle Definition für die Differenz von Dokumenten wird in [Kel07] gegeben.

> „Die **Differenz** ist *eine* Darstellung der Unterschiede zwischen zwei Dokumenten.“

Es ist also zu beachten, dass die Differenz zwischen zwei Dokumenten nicht eindeutig ist, wie im allgemeinen Verständnis einer Subtraktion in der Mathematik. Es ist vielmehr so, dass die Unterschiede zwischen zwei komplexen Dokumentelementen auf viele verschiedene Arten beschrieben werden können. Daher spricht man üblicherweise nicht von einer Subtraktion der Dokumente, sondern von einem Dokumentvergleich. Ein einfaches Beispiel für die Mehrdeutigkeit des Vergleichs von Dokumenten ist in Abbildung 2.1 zu sehen.

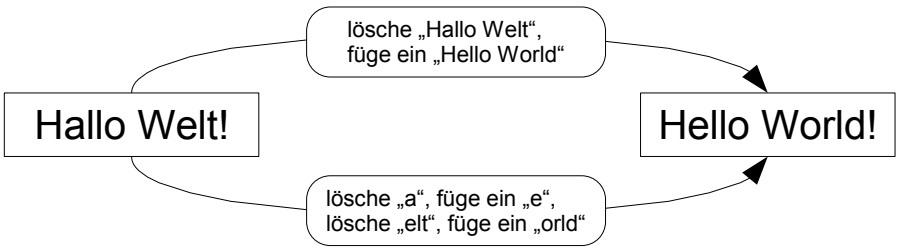


Abbildung 2.1.: Zwei von vielen möglichen Differenzbeschreibungen zwischen zwei unterschiedlichen Dokumenten.

Eine Differenz zwischen Dokumenten kann in zwei verschiedenen Varianten vorliegen. Eine **asymmetrische Differenz** entspricht einer Transformationsvorschrift, mit der sich das eine Dokument in das andere Dokument überführen lässt. Diese Form der Differenz wird üblicherweise bei versionierten Quelltexten benutzt. Die oben genannte Abbildung zeigt also zwei verschiedene asymmetrische Differenzen.

Die zweite Variante von Differenzen ist die **symmetrische Differenz**. Sie basiert auf einem mengenorientierten Ansatz und kommt bei strukturierten, technischen Dokumenten zum Einsatz. Die Differenz wird dadurch beschrieben, dass ausgehend von einer gemeinsamen Schnittmenge – den korrespondierenden Elementen beider Dokumente – die jeweiligen Ausgangsdokumente durch zwei unterschiedliche Einfügeoperationen erreicht werden können. Die beiden Einfügeoperationen haben dabei wiederum den Charakter einer asymmetrischen Differenz. Die Anforderungen an die Ermittlung und Beschreibung der Modelldifferenzen erhöht sich zusätzlich je nach Komplexität und Struktur der Elemente der beiden zu vergleichenden Dokumente. Abbildung 2.2 zeigt ein vereinfachtes Beispiel für eine symmetrische Differenz.

Abbildung 2.2.: Eine symmetrische Differenz, bestehend aus den gemeinsamen Elementen und zwei Einfügeoperationen.

2.2. Wie können Differenzen berechnet werden?

Die Berechnung von Dokumentdifferenzen unterscheiden sich je nach vorliegendem Dokumenttyp sehr stark. Verschiedene Verfahren zur Differenzberechnung sowohl für textbasierte als auch für baum- beziehungsweise graphorientierte Dokumente werden in der Diplomarbeit von J. Wehren [Weh04] ausführlich vorgestellt.

Da die vorliegende Diplomarbeit sich ausschließlich mit strukturierten Dokumenten, wie UML-Klassendiagrammen oder Matlab/Simulink-Diagrammen beschäftigt, soll hier lediglich auf Werkzeuge eingegangen werden, die mit solchen Diagrammtypen umgehen können. Nachfolgend wird daher immer von symmetrischen Differenzen ausgegangen, die zwischen den verschiedenen Modelltypen berechnet werden.

Die Erstellung von symmetrischen Differenzen hierarchischer Modelle bei der modellgetriebenen Entwicklung ist nicht trivial. Bei diesen Modellen handelt es sich um sogenannte Multimengen, was bedeutet, dass Elemente eines Typs im gleichen Modell mehrfach vorhanden sein können. Um diese mehrfachen Elemente dennoch voneinander unterscheiden zu können, gibt es für den Benutzer nicht sichtbare Attribute, die auf die Gleichheit der Elemente keine Auswirkung haben, aber deren eindeutige Identifizierung zulassen.

Zwei Werkzeuge, welche die Differenzen von Modellen mittels Korrespondenz der Elemente ermitteln sind $\mathbf{UMLDiff}_{cld}$ und $\mathbf{UMLDiff}_{sqd}$, welche in [Gir02] vorgestellt werden. Die beiden Werkzeug-Prototypen sind in der Lage, UML-Klassendiagramme beziehungsweise UML-Sequenzdiagramme zu vergleichen und die ermittelten Unterschiede anschließend zu visualisieren.

Die dort vorgestellte Lösung hat jedoch eine entscheidende Einschränkung. Es wird davon ausgegangen, dass die einzelnen Diagrammelemente zwischen den verschiedenen Modellrevisionen eindeutig durch Identifizierer (ID) referenziert werden. Haben also zwei Elemente die gleiche ID, so wird angenommen, dass es sich um korrespondierende Elemente handelt, bei denen lediglich noch überprüft werden muss, ob und welche Attribute der beiden Elemente sich zwischen den Versionen verändert haben.

Nicht immer kann jedoch garantiert werden, dass diese IDs über den kompletten Entwicklungsprozess eines Projektes identisch bleiben. Ein Grund ist zum Beispiel der Verlust der IDs bei der Konvertierung der Dokumente in ein anderes Datenformat durch die Benutzung unterschiedlicher UML-Werkzeuge. Es kann aber auch an einem einzelnen Werkzeug liegen, wenn es lediglich interne IDs zur Laufzeit benutzt, sodass bei jedem Lade-/Speicher-Zyklus der Dokumente andere IDs gespeichert werden. Beides hat zur Folge, dass die Elemente zwischen den Revisionen unterschiedliche IDs erhalten können und so die Benutzung von Differenzwerkzeugen, die auf die Unveränderlichkeit der IDs aufsetzen, nutzlos machen.

Das SiDiff-Projekt der Universität Siegen beschäftigt sich seit einigen Jahren ebenfalls mit der Berechnung von Differenzen zwischen verschiedenen Modellrevisionen. Eine erste Version des Algorithmus, der wie UMLDiff zunächst ebenfalls auf

persistenten IDs bei den Modellelementen basierte, wurde in [OWK03] vorgestellt. Der Algorithmus nimmt die beiden Modelle entgegen, berechnet die symmetrische Differenz und erstellt ein sogenanntes Mischdokument, welches die korrespondierenden Elemente und die beiden asymmetrischen Differenzen zu den Ausgangsmodellen enthält, sodass beide Modellrevisionen aus dem Ergebnis rekonstruierbar sind.

Spätere Versionen des Algorithmus benötigen die persistenten IDs nicht mehr. Die Diplomarbeit von Jürgen Wehren [Weh04] ermöglicht die Differenzberechnung zwischen Klassen- und Zustandsdiagramme der UML. Eine weitere Diplomarbeit [Lüc06], die in Zusammenarbeit mit der DaimlerChrysler AG entstand, passte den Algorithmus an die Berechnung von Differenzen auf Matlab/Simulink-Diagrammen an.

Die Anpassung an die unterschiedlichen Diagrammtypen erfolgt im wesentlichen durch eine konfigurierbare Ähnlichkeitsberechnung, die für die entsprechende Zuordnung der Diagrammelemente zwischen den Modellrevisionen sorgt. Dabei ist der Algorithmus in der Lage, identische, veränderte, neue, gelöschte und verschobene Elemente zu erkennen. Die IDs werden hier lediglich noch zur Identifikation der Elemente in einem der Dokumente benötigt. Für den Vergleichsalgorithmus selber werden sie nicht mehr hinzugezogen, um falsche Korrespondenzen zwischen Elementen mit zufällig gleicher ID auszuschließen.

2.3. Wie werden Differenzen dargestellt?

Für die Darstellung der berechneten Differenzen gibt es mehrere Ansätze, die unterschiedlich intuitiv sind. Die einfachste Form sind textuelle Ausgabeformen, wie zum Beispiel Listen, die jede Änderung Zeile für Zeile auflisten, ähnlich den Angaben wie in Abbildung 2.1. Manche Werkzeuge, wie die *Compare View* der Team-Perspektive in der Entwicklungsumgebung Eclipse, oder andere Software-Systeme zur Versionsverwaltung von Textdokumenten, benutzen eine halbgrafische Darstellung, in dem sie die beiden Textdokumente nebeneinander präsentieren und veränderte Zeilen entsprechend markieren oder durch Verbindungslinien kennzeichnen.

Bei strukturierten Modellen, die Gegenstand dieser Arbeit sind, gibt es ebenfalls die Möglichkeit der Textausgabe, so können die UMLDiff-Werkzeuge die Ergebnisse in HTML präsentieren und im SiDiff-Werkzeug erfolgt sie im XML-Format. Darüber hinaus kann die Ausgabe bei SiDiff aber auch grafisch erfolgen. Es werden dann üblicherweise die gemeinsamen, korrespondierenden Elemente in einer neutralen Farbe gezeichnet und für die speziellen Elemente der beiden Dokumente jeweils unterschiedliche Farben benutzt.

Abbildung 2.3 zeigt die Darstellung eines kleinen Mischdokuments in grafischer Form. Das Bild wurde der Internetseite des SiDiff-Projekts entnommen und ent-

stand mit einem Plugin, welches in [Lüc04] für das CASE-Tool[1] **Fujaba**[2] [Uni07a] entwickelt wurde. Die magentafarbenen Elemente befinden sich lediglich im ersten Dokument, während die grünen Elemente nur im zweiten Dokument existieren. Veränderungen an den Attributen eines Elements werden mit einem schwarzen `U` auf gelbem Grund markiert.

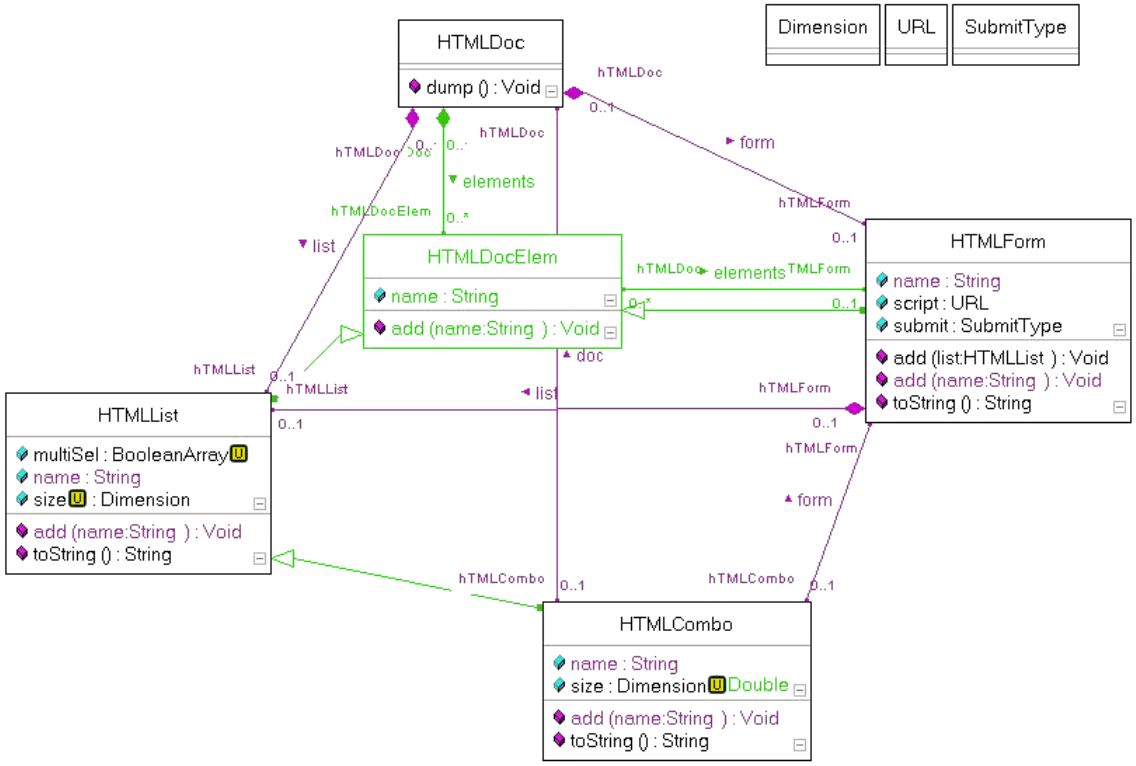


Abbildung 2.3.: Ein Mischdokument mit farblich markierten Unterschieden der beiden Ausgangsdokumente.

Wenn für die Interpretation der Ergebnisse angenommen wird, dass die beiden Dokumente zeitlich nacheinander entstanden sind, was der übliche Fall ist, dann bedeuten die Elemente in der Farbe Magenta, dass diese im Verlauf der Änderungen gelöscht wurden und die Elemente in Grün, dass sie hinzugefügt wurden.

Wie bereits in Kapitel 1 erwähnt, haben alle genannten Darstellungsformen der Differenzergebnisse für strukturierte Modelle den Nachteil, dass sie nur sehr schlecht

[1] Computer-Aided Software Engineering

[2] Mit der Fujaba Tool Suite – der Name Fujaba steht für *From UML to Java and back again* – können Softwaresystemen vollständig modellorientiert in der UML entwickelt werden und der dazugehörige Quelltext automatisch generiert werden. Wie der vollständige Name schon suggeriert, ist Fujaba auch in der Lage, vorhandenen Java-Quelltext wieder einzulesen und daraus Klassendiagramme zurückzugewinnen.

skalieren. Wie in der Abbildung zu sehen ist, gelingt die Darstellung bei den fünf Klassen – aufgrund der unterschiedlichen Beziehungskanten der Klassen untereinander – nur durch geschickte Positionierung. Schon bei wenigen zusätzlichen Klassen können die Beziehungen jedoch so vielfältig sein, dass die Darstellung gerade bei vielen Änderungen sehr komplex und unübersichtlich wird.

2.4. Anforderungen

In den folgenden Abschnitten sollen daher zunächst die Anforderungen an ein Werkzeug definiert werden, das es ermöglicht, Modelldifferenzen effizient darzustellen und die Veränderungen auf Wichtigkeit hin zu analysieren. Dazu werden auch einige bereits existierende Werkzeuge vorgestellt und daraufhin untersucht, ob deren Ansätze für eine notwendige, eigene Entwicklung genutzt werden können.

Es soll eine Lösung dafür gefunden werden, wie Änderungen sowohl an kleinen wie auch großen Dokumenten möglichst übersichtlich präsentiert werden können. Ebenfalls wird untersucht, ob sich mit dieser Form der Darstellung weitergehende Informationen über Art und Schwere der Modifikationen an den Dokumentelementen gewinnen lassen.

Ausgangspunkt für die Darstellung ist eine symmetrische Differenz, wie sie zum Beispiel der SiDiff-Algorithmus aus zwei gegebenen Dokumenten erstellt. Anstatt nun jede Änderung einfach darzustellen und dem Anwender zu präsentieren sollte eine Möglichkeit gefunden werden, von jedem Element oder jeder Änderung zu abstrahieren, um zu verhindern, dass die Präsentation mit wachsendem Umfang der Dokumente unübersichtlich wird.

Um die Präsentation des Modelldifferenz zu erzeugen, müssen zunächst aus den Modellen selber und aus der Differenz Daten ermittelt werden. Die Gewinnung dieser Daten ist ebenfalls eine Anforderung an das zu erstellende Werkzeug.

Anschließend soll die aus den Daten generierte Darstellung der Differenz eine qualitativ bewertende Aussage über die zwischen den Dokumenten vorhandenen Änderungen zulassen.

2.5. Analyse existierender Lösungen

In diesem Abschnitt werden zunächst einige schon existierende Lösungen betrachtet, die diese Anforderungen teilweise abdecken und deren Vor- und Nachteile herausgestellt. Anschließend wird darüber diskutiert, wie sich die Vorteile der Ansätze kombinieren lassen, um ein Werkzeug für die Analyse von Modelldifferenzen zu erstellen.

Das Gebiet der Modelldifferenzen und deren Präsentation ist ein relativ junges Forschungsthema der Informatik. Demzufolge gibt es nur wenige Werkzeuge, die eine

Erzeugung von Modelldifferenzen beziehungsweise der Mischdokumente zulassen. Mit UMLDiff und SiDiff wurden bereits zwei dieser Werkzeuge vorgestellt. Selbst auf der aktuellen deutschen Konferenz rund um Softwaretechnik, der **SE 2007** [Ges07a], wurden im Rahmen des VVUM-Workshops [Ges07b] keine alternativen Vergleichsalgorithmen oder Werkzeuge vorgestellt, die ohne die in Unterkapitel 2.2 beschriebenen, persistenten Identifizierer auskommen.

Da das SiDiff-Werkzeug gegenüber UMLDiff die genannten Vorteile bietet, wird im folgenden davon ausgegangen, dass eine symmetrische Differenz vorliegt, die vom SiDiff-Algorithmus erzeugt wird.

2.5.1. Erzeugung von Daten aus Softwareprojekten

Für das Projektmanagement in der Softwareentwicklung ist wichtig, zum Beispiel den aktuellen Entwicklungsstand oder die Qualität einer Software beurteilen zu können. Eine Technik, die zur Unterstützung der Beurteilung eingesetzt wird, ist die Ermittlung von sogenannten *Softwaremetriken* auf Basis der verfügbaren Quelltextdokumente. Unter einer einzelnen Metrik versteht man im wesentlichen eine Funktion, mit der bestimmte Eigenschaften der Software oder des Entwicklungsprozesses gezählt oder berechnet werden. Einfache Beispiele für solche Maße einer Software sind zum Beispiel LOC (Anzahl der Quelltextzeilen) oder NOA (Anzahl der Attribute einer Klasse).

Diverse weitere Metriken unterschiedlicher Art und Komplexität werden unter anderem in [LK94], [HS96] oder [FP96] vorgestellt. Heutzutage werden Metriken beziehungsweise die ermittelten Metrikwerte vorwiegend eingesetzt, um Informationen über den Entwicklungsprozess eines Projekts, das Produkt direkt und Aufwand, Kosten oder die Dauer zu gewinnen.

Um Aussagen über die Wichtigkeit von Änderungen an Modellen treffen zu können, reicht es nicht aus, lediglich die Ausgangsdokumente zu betrachten und darauf Metriken zu berechnen. Es müssen zusätzlich Daten über die Unterschiede der Dokumente gewonnen werden. Ein Werkzeug, welches diese Art der Datengewinnung für Softwareprojekte zur Verfügung stellt, ist der **Krakatau Project Manager** der Firma Power Software [PS07]. Die Anwendung ist in der Lage, für Projekte in unterschiedlichen Programmiersprachen, unter anderem C/C++ und Java, einige Differenzmetriken zwischen zwei verschiedenen Softwarerevisionen zu berechnen.

Die Metriken werden aus einer Kombination der Art des Veränderung und dem jeweiligen Elementtyp gebildet. Als Arten stehen *hinzugefügt, gelöscht* und *verändert* zur Verfügung und als Elementtypen *Dateien, Methoden* und einzelne *Quelltextzeilen*. Ergebnisse wie zum Beispiel „3 hinzugefügte Methoden in dieser Klasse“ erlauben dem Anwender anschließend, die Differenz der Dokumente genauer zu bewerten.

2.5.2. Abstrakte Darstellungen von Metriken

Eine Möglichkeit für die Darstellung von aus Quelltexten gewonnenen Metriken, bietet das in der Dissertation von M. Lanza [Lan03] vorgestellte Werkzeug. Es wird eine Diagrammform namens *Polymetrische Sicht* eingeführt, durch die bis zu fünf beliebige Metriken des gleichen Elementtyps eines Softwareprojektes miteinander in Beziehung gesetzt und für alle Elemente diesen Typs gleichzeitig präsentiert werden können. Eine Sicht besteht dabei aus einer Menge von Knoten, die den einzelnen Entitäten entsprechen und optional einer Menge von Kanten, welche die Beziehungen zwischen den Entitäten verdeutlichen. Eine schematische Darstellung der Sichten ist in Abbildung 2.4 zu sehen. Jeder der Knoten wird durch die fünf Attribute *Breite, Höhe, Farbe, horizontale und vertikale Position* beschrieben, denen die Metrikwerte zugeordnet werden können.

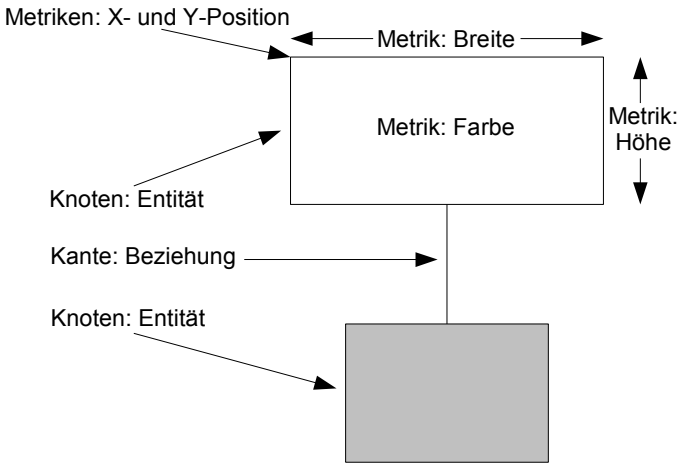


Abbildung 2.4.: Abstrakte Visualisierung von Softwaresystemen mittels Metriken und einer daraus erzeugten polymetrischen Sicht.

Polymetrische Sichten sind nach Aussage von Lanza dazu geeignet, den Reverse-Engineering-Prozess, also die Analyse von unbekannten Softwaresystemen, zu unterstützen. Dazu werden dem Anwender in dessen Arbeit einige polymetrische Sichten in Form von Sichtdefinitionen zur Verfügung gestellt. Mit dem Werkzeug namens **CodeCrawler** lassen sich diese Sichtdefinitionen anschließend auf den zu analysierenden Quelltext anwenden und die dazugehörigen polymetrischen Sichten erzeugen. Mit deren Hilfe können Aussagen über die Modellierung des Projekts, die Wichtigkeit von Modulen und ähnliche Informationen gewonnen werden. Zusätzlich lassen sich mit dem Werkzeug im sogenannten *Sichteneditor* weitere Sichtdefinitionen erzeugen, um zum Beispiel gezielt auf besondere Eigenschaften des Projekts einzugehen, oder gänzlich andere Aspekte des Systems hervorzuheben.

Eine konkrete Sichtdefinition besteht aus einer Reihe von technischen und informellen Angaben und entspricht einer Art Bauanleitung für die polymetrische Sicht.

Folgende Angaben werden aus technischer Sicht benötigt:

- Der Typ der Entitäten, die durch Knoten in der Sicht repräsentiert werden sollen.
- Die Zuordnung der Softwaremetriken zu den Attributen der Knoten.
- Ein optionaler Beziehungstyp, der in Form von Kanten dargestellt wird.
- Das Layout für die Anordnung der Knoten innerhalb der polymetrischen Sicht.
- Ein optionales Knotenattribut als Sortierreihenfolge für die Knoten.

Die informellen Angaben enthalten Empfehlungen darüber, wann die Sichtdefinition im Laufe der Softwareanalyse benutzt werden sollte, auf welche Bereiche der Software sie angewendet wird, welchem Zweck sie aus Sicht des Reverse-Engineers dient, welches Ziel sie verfolgt und wie man sie variieren kann, um auf bestimmte Aspekte einer Software einzugehen.

Durch die abstrakte Darstellungsform und den Einfluss, den man durch geschickte Wahl von Layout, Metriken oder Teile der Software darauf haben kann, ist die Skalierbarkeit der polymetrischen Sichten sehr gut. Auch tausende von Knoten nehmen unter diesen Umständen nur die Größe einer einzigen Bildschirmseite ein. Dadurch bleibt die Übersichtlichkeit gerade bei großen Projekten gewährleistet.

Um die Reverse-Engineering-Fähigkeiten der Fujaba Tool Suite zu erweitern wurde in [Fal05] das Plugin **Polymetric Views for Fujaba** (PV4F) entwickelt, welches die polymetrischen Sichten für die Analyse von eingelesenen Java-Quelltexten in Fujaba integriert. Zur Unterstützung benutzt PV4F die Metriken aus dem **Metrics-Plugin**, welches in [Rot05] ebenfalls als Fujaba-Plugin erstellt wurde und Softwaremetriken auf geparsten Quelltexten berechnen kann.

Seit dem Jahr 2005 wird die Fujaba-Plattform schrittweise in das **Eclipse-Framework**[3] [Ecl07a] integriert. Um die Funktionalität der Plugins in der neuen Umgebung weiterhin zu gewährleisten, wurden diese ebenfalls nach Eclipse migriert. Im Falle von **Polymetric Views for Eclipse** (PV4E) wurde das Plugin konzeptionell so erweitert, dass nicht nur Metriken aus dem Metrics-Plugin für Fujaba genutzt werden können, sondern auch Metriken aus beliebigen Quellen. Erreicht wird diese Funktionalität über eine Adapterschnittstelle, an der sich beliebig viele weitere Adapter an PV4E registrieren und ihre Metriken anbieten können.

[3] Bei Eclipse handelt es sich um ein quelltextoffenes Framework zur Entwicklung von Software aller Art. Geschichtlich gesehen handelte es sich bei der Plattform zunächst um eine erweiterbare Softwareentwicklungsumgebung (IDE) für Java. Mittlerweile besteht Eclipse aber nur noch aus einer Kernsoftware, die durch das Laden von Plugins um beliebige Funktionen, wie zum Beispiel die IDE-Funktionalität, erweitert werden kann. Durch diese als Rich-Client-Platform (RCP) bezeichnete Fähigkeit lassen sich funktional völlig unabhängige Werkzeuge entwickeln, die aber aufgrund der gleichen Kernfunktionalität von Eclipse ein einheitliches Look-and-Feel aufweisen.

2.5.3. Bewertung von Differenzdaten

Das Werkzeug **QualifiedDiff**[4] ist ein am Lehrstuhl für Softwaretechnik entstandenes Plugin für Eclipse. Es ermittelt – ähnlich dem Krakatau Project Manager – Metrikwerte an den Unterschieden zwischen zwei Softwarerevisionen. Anschließend werden die Ergebnisse dem Benutzer in einer Kombination aus textueller und grafischer Form präsentiert.

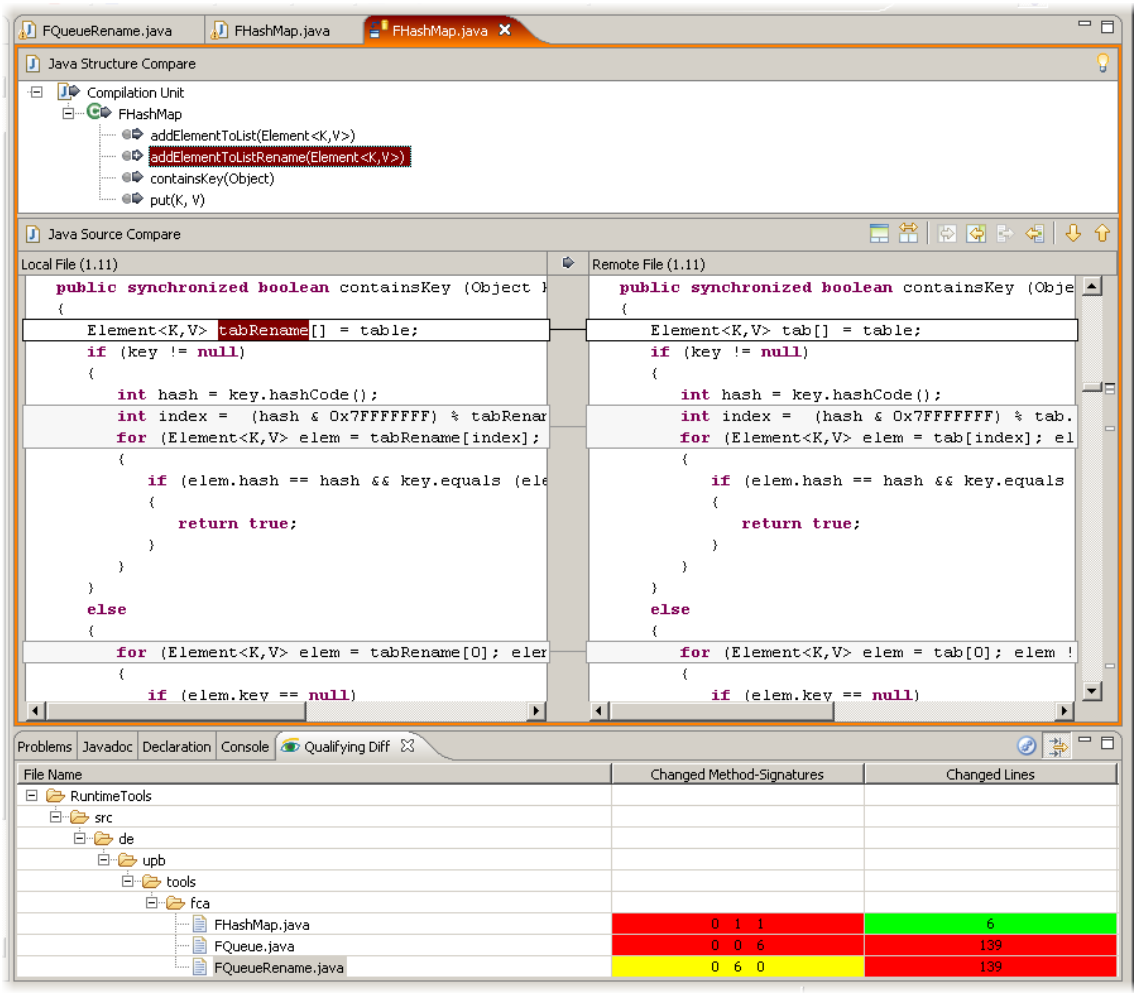


Abbildung 2.5.: Die Benutzeroberfläche des QualifiedDiff Plugins.

Für den Vergleich der Revisionen wird vorausgesetzt, dass das Projekt in einem CVS-Repository liegt. Es wird dann jeweils die aktuelle, lokal vorhandene Version des Quelltextes mit der im Repository vorliegenden Version verglichen. Für die qualitative Bewertung der Differenz wird den verschiedenen Arten der Änderungen über

[4] Eine zitierfähige Publikation dieser Arbeit liegt zum derzeitigen Zeitpunkt (Mai 2007) noch nicht vor

einen Konfigurationsdialog eine der drei Gewichtungen *trivial, mittel* und *kritisch* zugeordnet.

Zum gegenwärtigen Zeitpunkt ist das Werkzeug in der Lage, zwei Grundarten von Änderungen zu verfolgen. Das sind zum einen modifizierte Quelltextzeilen und zum anderen Modifikationen an den Methodensignaturen. Ausschlaggebend für die farbliche Darstellung ist dabei die maximale Gewichtung einer Änderung an einer Datei. Sind zum Beispiel nur triviale Änderungen vorgenommen worden so wird der Änderungstyp in grüner Färbung dargestellt. Sobald eine der Änderungen einer Änderungsart als kritisch eingestuft wird, ändert sich die Farbdarstellung nach rot. Man erhält dadurch in etwa eine Darstellung, wie sie in Abbildung 2.5 zu sehen ist.

Tabelle 2.1 gibt einen Überblick über die Standardeinstellungen, welche für die Differenzen zwischen den Revisionen in den Einstellungen von QualifiedDiff definiert sind. Diese können an die eigenen Vorstellungen und Bedürfnisse angepasst werden. Zusätzliche Änderungstypen und deren Konfigurationsdialoge können über einen eigenen Erweiterungsmechanismus hinzugefügt werden, sodass beliebige andere qualitative Aussagen über die Differenzen zwischen den Revisionen getroffen werden können.

Methodensignaturänderungen in Java-Dateien

Art der Änderung	**Gewichtung**
Parameter name has changed	Trivial
JavaDoc has changed	Trivial
Method visibility has increased	Medium
Method visibility has decreased	Critical
Method added	Medium
Method deleted	Critical
Method modifier `abstract` has changed	Medium
Method modifier `static` has changed	Critical
Method modifier `final` has changed	Critical
Method return type has changed	Critical
Method annotations have changed	Medium

Veränderung von Zeilen in beliebigen Textdateien

Art der Änderung	**Anzahl**
Geänderte Zeilen für triviale Änderung	1
Geänderte Zeilen für mittlere Änderung	10
Geänderte Zeilen für kritische Änderung	30

Tabelle 2.1.: Standardeinstellungen für die Änderungsmetriken in den Einstellungsdialogen von QualifiedDiff.

2.6. Begründung für eine eigene Entwicklung

Die Anforderungen an ein Werkzeug, welches Daten aus Modelldifferenzen erzeugen kann und diese anschließend in einer abstrakten Form visualisieren kann, sind sehr vielfältig. Zunächst muss eine symmetrische Differenz berechnet werden. Aus dieser Differenz müssen im Anschluss weitere Daten in Form von Differenzmetriken gewonnen werden. Zuletzt werden sie in einer abstrakten Darstellungsform so präsentiert, damit der Anwender Informationen über die Art der Änderungen daraus ableiten kann.

Der entscheidende Nachteil bei allen bisher vorgestellten Werkzeugen aus Unterkapitel 2.5 ist jedoch, dass sie nicht auf strukturierten Modellen arbeiten können, wie sie in der modellgetriebenen Entwicklung verlangt werden. Die Werkzeuge können lediglich mit Quelltexten auf der Implementierungsebene umgehen und wenden speziell darauf angepasste Funktionen zur Berechnung von Metrikwerten an. Die dabei zum Einsatz kommenden Versionsmanagementsysteme, die lediglich einfache Textvergleiche durchführen, sind für die komplexen strukturierten Modelle nicht geeignet.

Dennoch liefern alle Werkzeuge auch geeignete Ansätze zur Enwicklung eines eigenen Werkzeugs zur qualitativen Analyse von Differenzen auf Basis von Modelldiagrammen:

- Der Krakatau Project Manager ermittelt Differenzmetriken. Diese können auch für Modellelemente berechnet werden.
- Das PV4E-Plugin kann durch seine Erweiterungsmöglichkeit polymetrische Sichten mit Metriken aus beliebigen Quellen darstellen.
- Mit QualifiedDiff können Änderungen an einer Software qualitativ bewertet und Informationen über die Differenz daraus gewonnen werden.

Diese Ansätze zur qualitativen Differenzanalyse sollen daher in einem eigenen Werkzeug kombiniert werden, welches als Grundlage eine symmetrische Differenz auf strukturierten Diagrammen verwendet.

Kapitel 3.

Polymetrische Sichten auf Basis von Differenzmetriken

In diesem Kapitel soll beschrieben werden, wie die Anforderungen zur qualitativen Analyse von Modelldifferenzen programmiertechnisch umgesetzt werden können.

Dazu soll das Konzept für ein Werkzeug namens **Difference Metrics Plugin** (kurz DiMPl) entwickelt werden, das eine Brücke zwischen zwei bereits existierenden Werkzeugen aus der Fachgruppe Praktische Informatik bildet. Die dadurch auftretenden Rahmenbedingungen und erforderlichen Funktionalitäten sollen im folgenden Unterkapitel 3.1 vorgestellt werden.

Die Umsetzung des Konzepts auf technischer Ebene werden anschließend in Unterkapitel 3.2 anhand von konkreten Beispielen aus der Implementierung des Plugins genauer beschrieben.

3.1. Von Modelldifferenzen zu polymetrischen Sichten

Für die Berechnung von Differenzmetriken aus einer symmetrischen Differenz mit anschließender Visualisierung mittels polymetrischer Sichten, werden im wesentlichen drei Hauptkomponenten benötigt. Diese sollen die folgenden Aufgaben erfüllen:

1. Berechnung einer symmetrischen Differenz zwischen zwei Modellen.
2. Berechnung von Modell- und Differenzmetriken auf den Modellen und der Differenz.
3. Visualisierung von polymetrischen Sichten mit Hilfe der Metrikwerte.

Die erste Komponente, die Berechnung der Modelldifferenzen, existiert bereits in Form des SiDiff-Algorithmus. Benutzt werden soll die aktuelle Entwicklungsversion, die nach Abschluss der Diplomarbeiten [Weh04] und [Lüc06] einer Reihe von Verbesserungen in Bezug auf Stabilität, Geschwindigkeit und einer vereinheitlichten Benutzerführung unterlag.

Die Visualisierung von polymetrischen Sichten existiert ebenfalls und zwar als Plugin für das Eclipse-Framework. Das PV4E-Plugin ist in der Lage, unter anderem für Fujaba-Projekte mit Hilfe des Metrics-Plugins für Fujaba, Softwaremetriken in Form von polymetrischen Sichten darzustellen. PV4E enthält eine Schnittstelle für weitere Adapter, die Metriken an das Plugin liefern. Hier soll ebenfalls die aktuelle Entwicklungsversion des Plugins genommen werden.

Aus diesen Vorgaben ergibt sich, dass keine vollständig neue Anwendung entwickelt werden soll, sondern lediglich ein Werkzeug benötigt wird, welches in der Lage ist, die Berechnung der Metriken auf einer SiDiff-Differenz durchzuführen und diese an PV4E weiterleitet. Abbildung 3.1 zeigt, wie sich das geforderte Werkzeug zwischen die gegebenen Komponenten einfügen soll.

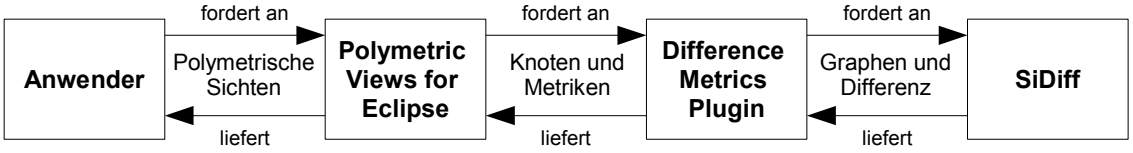


Abbildung 3.1.: Integration des Difference Metrics Plugins in die Kette der beteiligten Komponenten zur Erzeugung von Polymetrischen Sichten mit Differenzmetriken.

In den folgenden Abschnitten sollen zunächst die Rahmenbedingungen vorgestellt werden, die durch die Nutzung von PV4E und SiDiff vorgegeben werden. Anschließend wird diskutiert, wie diese Rahmenbedingungen genutzt und teilweise auch angepasst werden müssen, um die Anforderungen an das geplante Werkzeug zur Erstellung und Visualisierung von Dokumentdifferenzen zu realisieren.

3.1.1. Rahmenbedingungen durch PV4E

Das PV4E-Plugin wurde bereits bei der Integration in die Eclipse-Plattform dafür ausgelegt, dass es Dokumente und Metriken aus verschiedenen Quellen entgegennehmen und darauf polymetrische Sichten erzeugen kann. Dazu enthält das Plugin eine Schnittstelle, über die sich die Metrik-Werkzeuge am Plugin registrieren können.

Über diese Schnittstelle werden alle Aufgaben, die sich von Adapter zu Adapter unterscheiden können, von PV4E an den jeweiligen Adapter delegiert. Das zu implementierende Difference Metrics Plugin hat daher die Aufgabe, diese Schnittstelle zu bedienen. Die Schnittstelle umfasst die folgende Aufgaben, die zu unterschiedlichen Zeitpunkten während der Erstellung von polymetrischen Sichten anfallen:

- Anmelden an PV4E und mitteilen, welche Element- und Beziehungstypen in dem Adapter erzeugt werden können und welche Metriken berechenbar sind.

- Öffnen des zu analysierenden Projekts und Übermittlung des Inhalts in strukturierter Form für die Auswahl der darzustellenden Entitäten.
- Berechnen der Metriken für die ausgewählten Entitäten.
- Erzeugen der Knoten und Kanten für die ausgewählten Entitäten.

Nicht alle der Aufgaben sollen in DiMPl realisiert werden. Wie bereits in 3.1 zu sehen ist, soll ein Teil der Aufgaben wiederum an SiDiff abgegeben werden. Dazu gehört aus Sicht von PV4E unter anderem das Laden der beiden Dokumente in die interne Datenstruktur.

3.1.2. Rahmenbedingungen durch SiDiff

Genauso, wie DiMPl die geforderten Daten an das PV4E-Plugin liefert, delegiert es seinerseits einige der Anforderungen an den SiDiff-Algorithmus. Der reine SiDiff-Algorithmus ist in der Lage, zwischen zwei Dokumenten des gleichen Typs die korrespondierenden Elemente zu finden und anschließend die symmetrische Differenz der Modelle auszugeben. Der Ablauf für einen Vergleich zweier Modelle ist üblicherweise innerhalb eines kleinen Java-Programms in verschiedenen Phasen vorgegeben. Das Programm wird auf Kommandozeilenebene mit zwei Dokumenten als Parameter ausgeführt und durchläuft dabei die folgenden Phasen:

- Importieren der beiden Dokumente in die interne Graphen-Struktur.
- Laden der modellabhängigen Konfigurationen für den Vergleichsalgorithmus.
- Berechnung eindeutiger Pfade zu den Modellelementen .
- Laden optionaler, fest vorgegebener Korrespondenzen zwischen Elementen.
- Finden identischer Elemente durch einen Hashing-Algorithmus.
- Finden weiterer Korrespondenzen durch den SiDiff-Algorithmus.
- Ausgabe der Ergebnisse in der gewünschten Ausgabeform.

Dieser Ablauf erlaubt zwei mögliche Vorgehensweisen, wie das Difference Metrics Plugin an die Daten des SiDiff-Algorithmus gelangt.

Die erste Möglichkeit ist, die Ergebnisse in eine Datei zu schreiben, diese mit DiMPl wieder einzulesen und die Metrikberechnungen darauf auszuführen. Diese Variante hat jedoch entscheidende Nachteile, was die Interaktivität, Laufzeit und den Speicherverbrauch angeht.

Im PV4E-Plugin wird das zugrunde liegende Dokument für die polymetrischen Sichten interaktiv ausgewählt. Nach der Auswahl wird optional nur ein Teil der

Entitäten des Modell genutzt, um die Sicht zu erzeugen. Das Dokument muss dafür bereits vor der Kalkulation der Metriken komplett geladen sein. Wird die Berechnung einer polymetrischen Sicht angestoßen, so würde das Dokument vom SiDiff-Algorithmus ein zweites mal geladen, so dass sich sowohl die Laufzeit verlängert, als auch der Speicherverbrauch unnötig erhöht. Zusätzlich müsste ein Parser implementiert werden, der das Format der SiDiff-Ausgabe interpretieren kann und in eine eigene interne Datenstruktur umwandelt, um darauf Metriken zu berechnen.

Daher soll die zweite Variante favorisiert werden, die diese Nachteile nicht hat. Statt lediglich die fertigen Ergebnisse zu nutzen, sollen die Aufrufe der einzelnen Programmphasen von SiDiff direkt in DiMPl integriert werden. Diese Vorgehensweise hat den Vorteil, dass außer der symmetrischen Differenz auch die Zwischenergebnisse der einzelnen Phasen zur Laufzeit zur Verfügung stehen und in weiterführenden, DiMPl-spezifischen Aufgaben genutzt werden können.

3.1.3. Berechnung von Metriken

Für die qualitative Analyse der Modelldifferenz sollen zunächst so viele Daten wie möglich über die beiden Modelle und die Differenz in Form von Metriken zur Verfügung gestellt werden. Es hat sich herausgestellt, dass mittels der Datenstrukturen aus SiDiff drei verschiedene Typen von Metriken erzeugt werden können, die auf unterschiedliche Weise berechnet werden sollen:

1. **Modellmetriken**, die auf den einzelnen Modellen berechnet werden und in etwa den bekannten Softwaremetriken entsprechen.

2. **Differenzmetriken**, die auf der symmetrischen Differenz basieren.

3. **Ähnlichkeitsmetriken**, die auf den errechneten Ähnlichkeitswerten des SiDiff-Algorithmus basieren.

1. Modellmetriken

Die zu analysierenden, strukturierten Dokumente werden vor dem eigentlichen Modellvergleich in die interne Datenstruktur von SiDiff transformiert. Das dazugehörige Datenmodell ist in Abbildung 3.2 zu sehen.

Bei der Transformation wird jedes Dokument als Graph *(Graph)* aufgefasst, dessen Knoten *(Node)* aus den Element- und Beziehungstypen des Modells gebildet werden mit entsprechenden verbindenden Kanten *(Edge)* zwischen den Elementen. Jedes Graphelement hat dabei einen bestimmten Typ *(Node- bzw. EdgeType)*, dessen Name dem Elementtyp des Dokuments entspricht. Haben Elemente noch Eigenschaften, die nicht durch Knoten oder Kanten erfasst werden können, so werden diese durch Attribute *(Attribute)* in Form von *Name/Value*-Paaren modelliert.

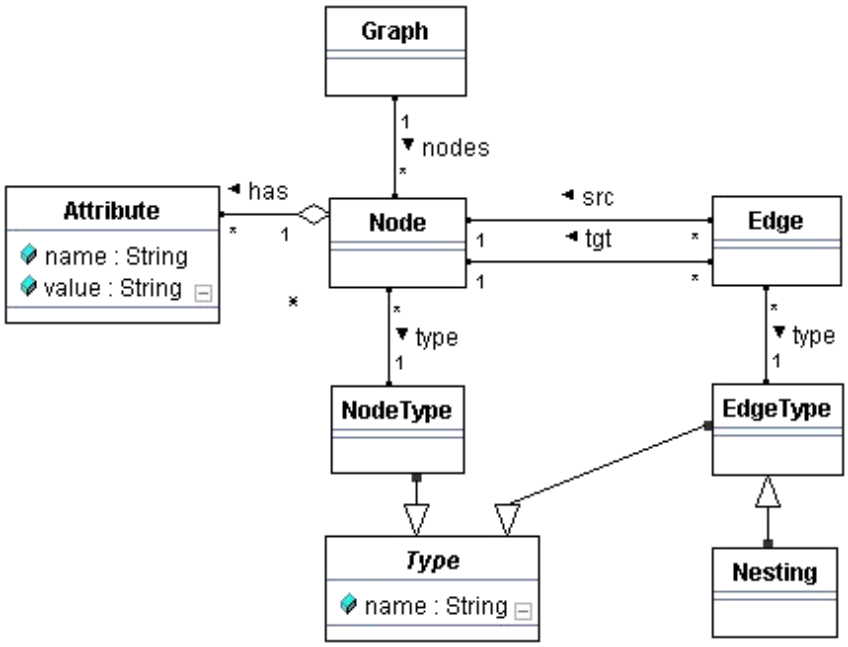


Abbildung 3.2.: Das interne Datenmodell von SiDiff für die Differenzberechnung nach [WK06].

Konkret bedeutet diese Transformation für Klassendiagramme, dass die Elementtypen wie *Modell, Paket* oder *Klasse* als Knoten aufgefasst werden und Beziehungen zwischen *Operationen* und *Parametern* als Kante.

Diese sehr allgemeine Datenstruktur erlaubt eine einfache Berechnung von Metriken für einzelne Elemente durch Abzählen der Knoten und Kanten. So müssen beispielsweise für die Ermittlung aller Klassen eines Pakets nur – ausgehend von dem Paketknoten – alle Kanten gezählt werden, an deren Ende ein Knoten vom Typ Klasse hängt. Über beliebig komplexe Abfragen auf dem Graphen lassen sich auf diese Weise viele der bekannten Softwaremetriken nachbilden. Dennoch wäre der Begriff Softwaremetriken unpassend, da für einen Teil der Softwaremetriken der Quelltext vorliegen muss, wie zum Beispiel für die *Anzahl der Quelltextzeilen einer Klasse*.

Da der SiDiff-Algorithmus lediglich die Differenz zwischen den beiden Graphen der Modelle berechnen kann, aber keine Metriken, muss geklärt werden, wo diese Berechnungen durchgeführt werden. Hierfür gibt es ebenfalls mehrere Alternativen, wobei zunächst eine Kalkulation der Metriken innerhalb von DiMPl angedacht war, um die Abhängigkeiten des Plugins gering zu halten.

Weitere Möglichkeiten sind eine Erstellung als eigenständiges Eclipse-Metrics-Plugin, ein eigenständiges SiDiff-Modul, oder eine Integration in das SiDiff.core-Modul. Hier hat den Ausschlag für die Integration in den Kern von SiDiff die Di-

plomarbeit von Christoph Treude [Tre07] gegeben. Für diese Arbeit, die sich mit der Laufzeitoptimierung des SiDiff-Algorithmus befasst, wurden ebenfalls Modellmetriken benötigt, die auf den Modellen der zu vergleichenden Dokumente berechnet werden.

Die Berechnung der Modellmetriken soll, wie bereits beschrieben, direkt auf der internen Datenstruktur von SiDiff erfolgen. Die konkreten Metrikwerte sollen in Form von Annotationen an den Elementen der Dokumente zur Verfügung gestellt werden, da in SiDiff bereits die Möglichkeit der Annotation einzelner Graphelemente existiert.

2. Differenzmetriken

Neben den Modellmetriken werden für die Erzeugung von polymetrischen Sichten auf Modelldifferenzen noch die eigentlichen Differenzmetriken benötigt, um die Unterschiede zwischen den Modellen ebenfalls durch Maße erfassen zu können.

Dazu werden nicht nur die beiden Modelle benötigt, sondern auch das Ergebnis des Vergleichs, die symmetrische Differenz. Da nach der Berechnung der symmetrischen Differenz der eigentliche SiDiff-Algorithmus beendet ist, soll die Nachverarbeitung der Ergebnisse, also die Ermittlung der zusätzlichen Metriken, in DiMPl erfolgen.

Der eigentliche SiDiff-Algorithmus sucht für jeden Knoten eines Graphs über den Vergleich der Attribute und Nachbarknoten im anderen Graph nach einem korrespondierenden Element gleichen Typs und erstellt für mögliche Unterschiede zwischen den Partnerknoten oder für den Fall, dass kein Partner gefunden wurde, einen Eintrag der folgenden Kategorien:

Equal: Das Element ist in beiden Graphen unverändert geblieben, unabhängig davon, ob darin geschachtelte Unterelemente verändert wurden.

Update: Ein Attribut des Elements hat sich zwischen den Graphen geändert (zum Beispiel der Name einer Klasse).

Structural: Das Element existiert nur in einem der beiden Dokumente, es wurde also kein geeigneter Partnerknoten gefunden (zum Beispiel eine Klasse wurde gelöscht).

Move: Der Vaterknoten des Elements hat gewechselt (Verschiebung einer Klasse in ein anderes Paket)

Reference: Eine Referenz auf einen anderen Elementtyp hat sich geändert (zum Beispiel der Rückgabetyp einer Methode).

Für die Differenzmetriken soll diese Aufteilung leicht verändert werden, da zum Beispiel aus der Sicht eines Elements unerheblich ist, ob ein Unterelement wirklich

gelöscht wurde, oder einfach nur an eine andere Stelle im System verschoben wurde. Es ergeben sich dadurch folgende Kategorien von Metriken:

Unterlement unverändert: Das Unterlement kommt aus der Kategorie *Equal*

Unterlement hinzugefügt: Das Unterlement kommt aus der Kategorie *Structural* und existiert nur im zweiten Dokument oder ist aus der Kategorie *Move* und der Ziel-Vaterknoten das Element

Unterlement entfernt: Das Unterlement kommt aus der Kategorie *Structural* und existiert nur im ersten Dokument oder ist aus der Kategorie *Move* und der Start-Vaterknoten ist das Element

Unterlement verändert: Das Unterlement kommt aus der Kategorie *Update* oder *Reference*

Neben diesen Metriken für allgemeine Informationen über die Eigenschaften der Unterelemente eines Elements, können aber noch weitere Differenzmetriken spezifiziert werden, die zusätzlich die Attribute und Referenzkanten der Elemente mit berücksichtigen.

SiDiff ist in der Lage, durch entsprechende Konfigurationsdateien, unterschiedliche Diagrammtypen zu vergleichen. In jedem dieser Diagrammtypen gibt es eine Reihe von Elementtypen, die auf verschiedene Weisen hierarchisch miteinander verknüpft sind. Um nicht für jeden neuen Elementtyp die Berechnungsfunktion der Metriken neu schreiben zu müssen, sollen die Metriken generisch aus der Dokumentstruktur erzeugt werden.

Die Dokumentstruktur soll dazu, ähnlich der Konfiguration für den Vergleich des SiDiff-Algorithmus, in einer Konfigurationsdatei beschrieben werden. Daraus sollen mittels eines Algorithmus die jeweiligen Metriken generisch erzeugt werden. Algorithmus 1 zeigt den Vorgang, mit dem die Metriken für einen beliebigen Knotentyp erzeugt werden sollen.

Nachfolgend dazu ein kurzer Ausschnitt aus der Konfigurationsdatei mit deren Hilfe der Algorithmus die Metriken erzeugen soll. Dieser Teil der Datei konfiguriert die Pakete und Klassen eines UML-Klassendiagramms.

```
1  [...]
2  <nodetype name="package">
3    <nestednodetype name="association" edgetype="assocs"/>
4    <nestednodetype name="package" edgetype="packages"/>
5    <nestednodetype name="class" edgetype="classes"/>
6    <attribute name="name" isunordered="true"/>
7  </nodetype>
8  <nodetype name="class">
9    <nestednodetype name="operation" edgetype="methods"/>
10   <nestednodetype name="attribute" edgetype="attrs"/>
11   <nestednodetype name="generalization" edgetype="inheritance"/>
```

```
12   <attribute name="name" isunordered="true"/>
13   <attribute name="isAbstract" isunordered="true"/>
14   <attribute name="visibility" isunordered="false" ascendingorder=↩
         "private,package,protected,public"/>
15   <edgetype name="stereotypes" issingle="false" nodenames="↩
         interface,reference,singleton,type,JavaBean"/>
16 </nodetype>
17 [...]
```

Algorithmus 1 : Generische Metriken für einen Knotentyp knT

```
1  für alle inneren Knotentypen über Nestingkanten des knT berechne
2      // Teil 1 - direkt
3      Anzahl Elemente vom Typ iKnT unverändert
4      Anzahl Elemente vom Typ iKnT hinzugefügt
5      Anzahl Elemente vom Typ iKnT entfernt
6      Anzahl Elemente vom Typ iKnT verändert
7      // Teil 2 - Attribute
8      für alle Attribute des iKnT berechne
9          wenn Attributwerte des Attribut ungeordnet dann
10             Anzahl Elemente vom Typ iKnT mit Attributwert attrName.wert
               verändert
11         Ende
12         sonst wenn Attributwerte des Attribut geordnet dann
13             Anzahl Elemente vom Typ iKnT mit Attributwert attrName.wert
               vergrößert
14             Anzahl Elemente vom Typ iKnT mit Attributwert attrName.wert
               verkleinert
15         Ende
16     Ende
17     // Teil 3 - Referenzen
18     für alle Referenzkantentypen des iKnT berechne
19         wenn Referenz auf einzelnen Knoten dann
20             Anzahl Elemente vom Typ iKnT mit Referenzziel über rKT verändert
21         Ende
22         sonst wenn Referenz auf mehrere Knoten dann
23             für alle Knotennamen der referenzierten Knoten berechne
24                 Anzahl Elemente vom Typ iKnT mit Referenzziel über rKT hat
                   Knotenname name verändert
25             Ende
26         Ende
27     Ende
28 Ende
```

Für den konkreten Anwendungsfall lassen sich somit bereits mit Teil 1 des Algo-

rithmus jener Typ Metriken erzeugen, welche im Krakatau Project Manager enthalten sind. Beispiel für die unterschiedlichen Teile des Algorithmus sind:

1. Anzahl Klassen eines Pakets, die hinzugefügt wurden
2. Anzahl Klassen eines Pakets, deren Attribut *name* sich geändert hat
3. Anzahl Klassen eines Pakets, deren Stereotyp mit Attribut *name* und Wert *interface* sich geändert hat

3. Ähnlichkeitsmetriken

Mit Hilfe der Zwischenergebnisse des Vergleichsalgorithmus von SiDiff lässt sich noch ein weiterer Typ von Metriken erzeugen. Für alle Partnerknoten kann abgefragt werden, wie ähnlich sie zueinander sind. Dieser Ähnlichkeitswert wird durch die Konfiguration des Vergleichsalgorithmus bestimmt. Für jeden Elementtyp ist dort definiert, wie stark sich Änderungen an den Elementen und ihren Beziehungen auf das Matching, also dem Finden von Partnerknoten, auswirken. Ausgehend von einer maximalen Ähnlichkeit von `1,0` wirkt sich jede Änderung negativ auf diesen Wert aus. Wird der Schwellwert für die Ähnlichkeit bei allen Knoten des anderen Graphen unterschritten, so handelt es sich um eine strukturelle Änderung.

Beispielsweise hat die Namensänderung einer Klasse eine Gewichtung von bis zu `0,4`, sodass ein neuer Name alleine keine Auswirkung auf das Finden des idealen Partnerknoten hat. Erst in Kombination mit den Attributen und Operationen, die sich jeweils mit bis zu `0,2` auf die Ähnlichkeit auswirken, wird der Schwellwert von `0,6` bei Klassen unterschritten.

Dieser Ähnlichkeitswert im Bereich von `0,0` für nicht gematchte Knoten bis `1,0` für keine Attribut- und Beziehungsänderungen kann für den dritten Typ von Metriken verwendet werden, den sogenannten Ähnlichkeitsmetriken. Die polymetrischen Sichten sind so ausgelegt, dass je größer der Wert für eine Metrik ist, desto auffälliger ist der dargestellte Knoten. Da für eine Analyse der Modelldifferenz gerade die Unterschiede zwischen den Modellen relevant sind, soll hier der Wert $\big(1 - \mathit{Ähnlichkeit}\big) * \mathit{Skalierungsfaktor}$ für die Metrik *Unähnlichkeit* eines Knotens benutzt werden. Der Skalierungsfaktor soll ein konfigurierbarer Wert im Bereich von `10` bis `100` sein, der die Größe der Knoten in einer polymetrischen Sicht beeinflusst, wobei im Folgenden von einem Wert von `100` ausgegangen werden soll.

Problematisch sind bei dieser Vorgehensweise auch die neuen und gelöschten Knoten der Dokumente, die durch einen nicht vorhandenen Partnerknoten eine Ähnlichkeit von `0,0` aufweisen und somit eine Unähnlichkeit von `100` erhalten. Da die Schwellwerte für das Matching der Elemente in Klassendiagrammen im Bereich von `0,4` bis `0,7` liegen, klafft eine sehr große Lücke zwischen gematchten und ungematchten Knoten. Daher soll der Ähnlichkeitswert für neue und gelöschte Knoten in einem

Bereich von `0,0` bis `1,0` ebenfalls konfigurierbar sein, damit sich die strukturellen Änderungen in die polymetrischen Sichten besser integrieren.

Algorithmus 2 : Berechnung der Ähnlichkeitsmetriken für einen Knoten

```
1  Berechne Unähnlichkeit für den Knoten
2  wenn Knoten Unterelemente besitzt dann berechne
3      Durchschnittliche Unähnlichkeit direkter Unterelemente
4      Durchschnittliche Unähnlichkeit direkter Unterelemente eines Typs
5      Durchschnittliche Unähnlichkeit aller Unterelemente
6      Maximale Unähnlichkeit direkter Unterelemente
7      Maximale Unähnlichkeit direkter Unterelemente eines Typs
8      Maximale Unähnlichkeit aller Unterelemente
9      Anzahl Unterschiede direkter Unterelemente
10     Anzahl Unterschiede direkter Unterelemente eines Typs
11     Anzahl Unterschiede aller Unterelemente
12     Anzahl direkter Unterelemente
13     Anzahl direkter Unterelemente eines Typs
14     Anzahl aller Unterelemente
15 Ende
```

Der Algorithmus 2 soll, ausgehend vom Wurzelknoten der Graphen, die Ähnlichkeitsmetriken rekursiv berechnen. Als Zusatzinformation werden noch die Differenzmetriken *Anzahl der Unterschiede* und die Modellmetriken *Anzahl der Unterelemente* ohne Mehraufwand mitgezählt.

3.1.4. Darstellungsformen

Polymetrische Sichten nach Lanza werden für ein Dokument beziehungsweise Softwaremodell erstellt. Im Anwendungsfall der Modelldifferenzen liegen jedoch zwei verschiedene Dokumente und deren symmetrische Differenz vor. Da für eine Beurteilung der Qualität der Änderungen an einem Dokument eine zeitliche Abfolge der Dokumente angenommen wird, soll diese Annahme auch hier benutzt werden. Der Auswahlbaum, in dem die Elemente für die Darstellung in der polymetrischen Sicht selektiert werden, soll daher nur die Elemente aus dem zweiten Dokument enthalten. Ebenso sollen alle zu berechnenden Metriken aus der Sicht des zweiten Dokuments berechnet werden.

Für die Beurteilung der Wichtigkeit der Differenzen ist zusätzlich vorteilhaft, wenn die Elemente, die zwischen den Revisionen gelöscht wurden, ebenfalls dargestellt werden. Da jedoch im Auswahlbaum nur die Elemente des zweiten Dokuments auswählbar sind, würden die gelöschten Elemente aus dem ersten Dokument niemals gezeichnet.

Daher soll für die konkrete Ermittlung der Entitäten, die zu zeichnen sind, die symmetrische Differenz benutzt werden. Sie enthält alle Elemente beider Dokumente in den verschiedenen Kategorien, die im vorhergehenden Abschnitt beschrieben wurden. Diese Lösung alleine reicht für das Zeichnen der gelöschten Knoten noch nicht aus, da die Elemente weiterhin nicht selektierbar sind.

Eine Alternative wäre nun, einfach alle gelöschten Elemente des darzustellenden Knotentyps zu visualisieren. Das führt jedoch dazu, dass unpassend viele Elemente dargestellt werden, falls nur bestimmte Bereiche des Modells im Auswahlbaum ausgewählt wurden. Deswegen soll als zusätzliche Bedingung für die gelöschten Knoten gelten, dass nur diejenigen Elemente – und dessen Unterelemente – dargestellt werden, deren Vaterknoten einen Partnerknoten im zweiten Dokument haben, der im Auswahlbaum selektiert wurde.

Bei der Benutzung der symmetrischen Differenz als Basis ist jedoch zu beachten, dass ein Element, welches sich nicht in der Kategorie *Equal* oder *Structural* befindet, mehr als eine Änderung erfahren haben kann. So kann für ein Element gelten, dass es verschoben wurde, ein Attribut sich verändert hat und auch die Referenz auf ein anderes Element sich geändert hat. Darüber hinaus ist es auch möglich, dass sich bei einem Element mehrere Attribute geändert haben. Das führt dazu, dass dieses Element mehrfach in der Differenz auftaucht, da dort jeder Unterschied einzeln nach Typen sortiert gesammelt wird. Es ist also mit geeigneten Mitteln darauf zu achten, dass Elemente mit mehrfacher Änderung nur einfach in der polymetrischen Sicht gezeichnet werden.

Mit der Kategorisierung der Änderungen der symmetrischen Differenz existiert eine Eigenschaft, die eine wichtige Aussage über die einzelnen Elemente zulässt. Leider handelt es sich bei den Kategorien um kein echtes Maß für eine Metrik. Dennoch kann diese Eigenschaft für ein visuelles Merkmal der polymetrischen Sichten ausgenutzt werden, da lediglich fünf dieser Kategorien existieren.

In der Darstellung von polymetrischen Sichten besitzt jeder gezeichnete Knoten ursprünglich einen schwarzen Rahmen, um auch sehr helle Knoten auf dem weißen Hintergrund sichtbar zu machen. Diese immer vorhandene Rahmenfarbe soll nun genutzt werden, um die Kategorien, in der sich ein Knoten befindet, direkt zu visualisieren. Für die Darstellung der verschiedenen Kategorien wurden die folgenden deutlich unterscheidbaren Grundfarben ausgewählt:

Schwarz	für Knoten aus *Equal*
Gelb	für Knoten aus *Update*
Rot	für Knoten aus *Structural* und im ersten Dokument
Grün	für Knoten aus *Structural* und im zweiten Dokument
Blau	für Knoten aus *Move*
Magenta	für Knoten aus *Reference*
Cyan	für Knoten die mehrfach in einer oder mehreren Kategorien sind

Dadurch können in einer polymetrischen Sicht unmittelbar die Änderungskatego-

rien erkannt werden. Wenn beispielsweise die Paketstruktur eines Klassendiagramms angezeigt wird, fallen die roten und grünen Knoten für Löschungen und Einfügungen besonders auf. Zusätzlich sollen die tatsächlichen Änderungen, wie zum Beispiel die Angabe des alten und des neuen Namens bei einer Namensänderung eines Elements in die Beschreibung eines Knotens integriert werden.

3.2. Das Difference Metrics Plugin

Die in dieser Arbeit konkret zu implementierenden Komponenten sind folgende Elemente, die sich aus einer Kombination der Schnittstelle zum PV4E-Plugin und den Phasen des SiDiff-Algorithmus zusammensetzen. Sie decken den Bereich zwischen PV4E und SiDiff aus der Grafik 3.1 vollständig ab.

1. Anmeldung und Bereitstellung der bekannten Element- und Beziehungstypen und der Metriken an PV4E durch DiMPl
2. Laden der Dokumente und Bereitstellung der Graphen an DiMPl und PV4E durch SiDiff
3. Berechnung der symmetrischen Differenz und Kalkulation der Metriken durch SiDiff und DiMPl
4. Berechnung und Bereitstellung der Knoten und Kanten einer polymetrischen Sicht an PV4E durch DiMPl

Da die Benutzung des SiDiff-Algorithmus laut Konzept in das Plugin integriert werden sollte, bestand die Implementierung der Komponenten zum überwiegenden Teil in der Programmierung des Difference Metrics Plugin. Nur die bereits vorher beschriebenen Besonderheiten von Modelldifferenzen, wie die Konfiguration der relativen Metriken und der Änderung der Rahmenfarbe erforderten kleine konzeptuelle Anpassungen an PV4E. Abgesehen von der Integration der Phasen des SiDiff-Algorithmus erfolgten an SiDiff selbst lediglich Anpassungen von Konfigurationsdateien.

In den nachfolgenden Abschnitten wird nun die konkrete Implementierung des Difference Metrics Plugins beschrieben. Dabei wird unter anderem auf die verschiedene Design-Entscheidungen eingegangen. Um die Beschreibung der einzelnen Komponenten möglichst intuitiv zu gestalten, folgt sie dem Ablauf der Erzeugung einer polymetrischen Sicht für Modelldifferenzen.

Das Polymetric Views for Eclipse Plugin besitzt eine umfangreiche Adapterschnittstelle, die während der Erstellung von polymetrischen Sichten vielfältige Aufgaben an den aktuellen Adapter delegiert, wie bereits zu Beginn von Kapitel 3 beschrieben. Die komplette Schnittstelle für die Adapter ist in Abbildung 3.3 zu sehen. Diejenigen Teile, welche während den einzelnen Schritten der Sichtberechnung aktiv sind, werden in den entsprechenden Abschnitten erläutert.

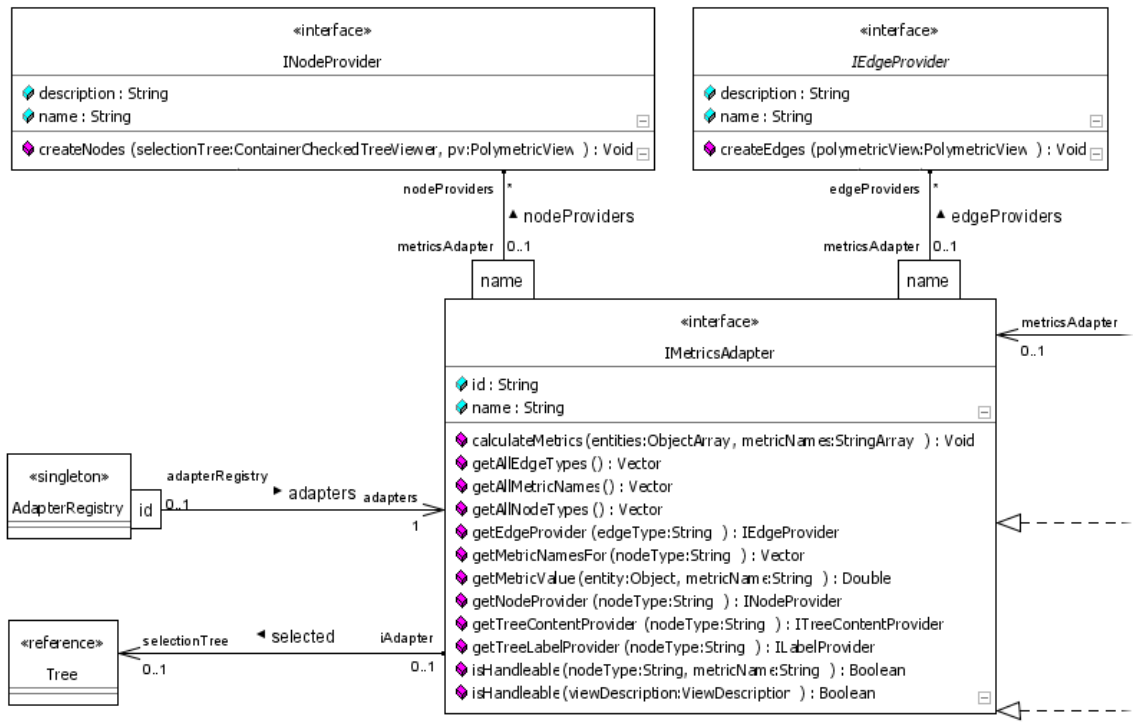


Abbildung 3.3.: Die Adapterschnittstelle des Polymetric Views for Eclipse Plugins.

3.2.1. Bereitstellung von Element- und Beziehungstypen und deren Metriken

Bevor die Berechnung einer polymetrischen Sicht oder die Erstellung einer Sichtdefinition in PV4E erfolgen kann, müssen die Adapter sich an dem Plugin registrieren und mitteilen, welche Element- und Beziehungstypen sie kennen oder genauer gesagt, mittels Knoten- und Kantenprovidern erzeugen können. Weiterhin muss mitgeteilt werden, welche Metriken bekannt sind und für welche Elementypen sie berechnet werden können.

Die Anmeldung von DiMPl an PV4E erfolgt über einen sogenannten *Extension Point* (Erweiterungspunkt) der von PV4E zur Verfügung gestellt wird. Der Erweiterungspunkt-Mechanismus ist essentieller Teil des Eclipse-Frameworks und wird in zahlreichen Einführungen und Dokumentationen wie zum Beispiel [Ecl07b] näher erläutert.

Der Erweiterungspunkt von PV4E nennt sich, wie in Abbildung 3.4 zu sehen ist, `de.usi.rcp.polymetricviews.metricsAdapter` und erwartet neben der Angabe der Eigenschaften `id` und `name` noch eine konkrete Klasse, die das Difference Metrics Plugin mit den beiden gleichlautenden Attributen initialisiert. Die beiden Attribute `id` und `name` dienen beim Start und während der Nutzung der Anwendung zur internen Identifikation des Adapters innerhalb des PV4E-Plugins.

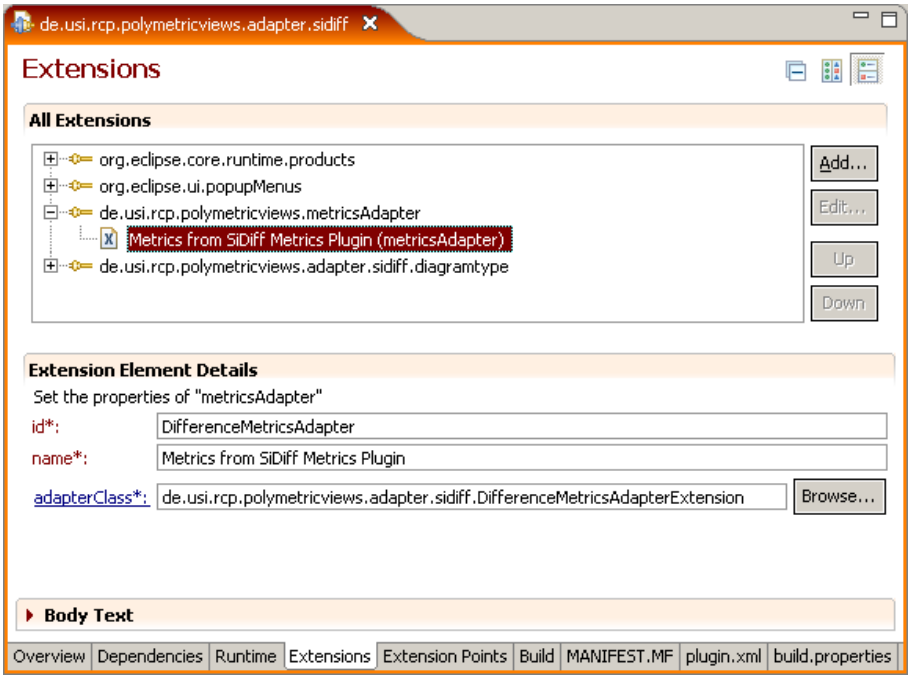


Abbildung 3.4.: Erweiterung des PV4E-Plugin um DiMPl durch Registrierung des Erweiterungspunkts.

Jeder Adapter ist in der Lage, für verschiedene Elementtypen Knoten zu erzeugen. Die erforderliche Providerklasse ist für den jeweiligen Elementtyp zu erstellen und wird in einer internen HashMap registriert. Gleiches gilt für die unterschiedlichen Beziehungstypen. PV4E bietet dazu die geeigneten Schnittstellen in Form von Interfaces und abstrakten Klassen.

Über die Methoden `getAllNodeTypes, getAllEdgeTypes` der Adapterschnittstelle erhält PV4E die entsprechenden Listen für die erzeugbaren Konten und Kanten, damit diese zum einen für die Sichtdefinitionen auswählbar sind und zum anderen, um über diesen Namen den konkreten Provider bei der Erstellung einer polymetrischen Sicht benutzen zu können.

Ebenso wird für die Erstellung von Sichtdefinitionen eine Liste der zur Verfügung stehenden Metriken benötigt. Die Modellmetriken, die in SiDiff berechnet werden können, werden über eine Konfigurationsdatei erstellt und in der Klasse `QueryRegistry` verwaltet. Die einzelnen `Query`-Klassen besitzen unter anderem einen Namen und die Information, für welche Elementtypen sie berechnet werden können. Das sind genau die Informationen, die für die Methoden `getAllMetricNames` und `getMetricNamesFor` der Adapterschnittstelle benötigt werden. Ebenso besitzen die Klassen der Modellmetriken jeweils eine `Annotator`-Klasse, die konkret für die Berechnung der Metrik und der Annotierung des Elements mit dem Metrikwert auf dem Graphen zuständig ist.

Dagegen werden die Differenz- und Ähnlichkeitsmetriken aus einer Kombination der Algorithmen 1 und 2 über die Konfiguration der Dokumentstruktur in der `plugin.xml` generiert. Dazu werden `Query`-Klassen erzeugt, die ausschließlich einen generischen Namen und den akzeptierten Elementtyp enthalten. Es wird dort keine Annotator-Klasse hinzugefügt, da die Berechnung der Metriken nach Erstellung der symmetrischen Differenz auf Basis der Differenz, beziehungsweise durch einen Top-Down-Durchlauf der Graphen, automatisch erfolgt.

Die beiden `isHandleable`-Methoden dienen dazu festzustellen, ob eine Sichtdefinition, welche mit PV4E erstellt wurde auch durch diesen Adapter darstellbar sind. So ist zum Beispiel eine Sichtdefinition, die für Simulink-Blöcke erstellt wurde, nicht kompatibel mit Adaptern, die lediglich UML-Elemente kennen. Kennen beide Adapter UML-Elemente, so ist natürlich auch die Art der Metriken, also etwa Software- und Differenzmetriken zu überprüfen, ob eine Sichtdefinition mit dem Adapter handhabbar ist.

3.2.2. Laden der Graphen

Das PV4E-Plugin erfordert, dass der Inhalt des zu analysierenden Dokumentes vor der eigentlichen Berechnung der polymetrischen Sicht zur Verfügung steht, um eventuell eine Vorauswahl der darzustellenden Entitäten treffen zu können. Es ist die Aufgabe des Adapters, eine Möglichkeit zur Verfügung zu stellen, das Dokument auszuwählen und die darin enthaltenen Elemente in Form einer Baumstruktur PV4E zu übergeben.

In den bisherigen Adaptern wurde es so gehandhabt, dass das Kontexmenü der entsprechenden Modelldatei im *Resource Navigator* von Eclipse um einen Eintrag erweitert wird, der den Ladevorgang und die Aktualisierung der *Entity Selection View* anstößt. Eine Besonderheit von DiMPl gegenüber den bisherigen Adaptern für PV4E besteht darin, dass hier zwei Modelle anstatt nur einem geladen werden müssen. Zusätzlich musste auf die verschiedenen Dokumenttypen Rücksicht genommen werden.

Die Erweiterung des Knotextmenüs musste also zum einen für jeden Dokumenttyp erfolgen und zum anderen sind dazu jeweils zwei Menüeinträge notwendig. Abbildung 3.5 zeigt die Konfiguration der Datei `plugin.xml` von DiMPl.

Beim Aufruf eines Menüeintrags im Resource Navigators wird der Adapter durch den Aufruf der Methode `loadDocument()` dazu veranlasst, das entsprechende Dokument zu laden und das Modell in die interne Graphenstruktur zu transformieren. Der nachfolgende Quelltext zeigt einen Ausschnitt aus der Lademethode von DiMPl.

```
1 private Graph loadDocument(IFile document)
2 {
3   Graph graph;
4   [...]
5   if (document.getFileExtension().equals("xmi"))
```

```
6   {
7     if (!currentDocumentType.equals(UML_CLASSDIAGRAM))
8     {
9       [...]
10      // init SymmetricDifference
11      SymmetricDifference.setAdvisor(new
            SetReferenceAsSingleReferencesAdvisor());
12      SiDiffEntityResolver.putMapping("xmi12-uml14.dtd", "xmi12-
            uml14.dtd");
13      // load configuration for uml class diagrams
14      ConfigurationLoader.loadConfiguration("umlClassDiffConfig.
            xml");
15      // load configuration for path calculation
16      AnnotationConfigurationLoader.loadConfiguration("
            SiDiffAnnotationsInstance.xml");
17      [...]
18    }
19    // load document and transform into graph
20    graph = GraphLoader.transformAndLoad("fujabaXMI.xslt", new
          File(document.getLocationURI()));
21  }
22  if (document.getFileExtension().equals("mdl"))
23  {
24    [...]
25  }
26  [...]
27  return graph;
28 }
```

Abgebildet ist der Teil der Methode, der für das Laden von Klassendiagrammen der UML zuständig ist, die aus Fujaba mit dem XMI-Export Plugin exportiert wurden. Nachdem der Modelltyp festgestellt wurde, wird der SiDiff-Algorithmus mittels einiger Konfigurationsdateien für den Vergleich der UML-Klassendiagramme initialisiert. Anschließend wird das Dokument geladen und in die interne Graphenstruktur von SiDiff transformiert. Der transformierte Graph wird dann für die weitere Verwendung an DiMPl zurückgegeben.

Dieser Vorgang entspricht den ersten beiden Phasen des SiDiff-Algorithmus wie sie normalerweise von der Kommandozeile aus aufgerufen werden. Da die Konfiguration und die Transformation für jeden Diagrammtyp individuell sind, muss die Methode für jeden Dokumenttyp entsprechend erweitert werden. Bisher ist das Laden von Matlab/Simulink-Diagrammen auf diese Weise ebenfalls möglich.

Die Konfigurationsdateien für den Vergleich der Diagrammtypen – hier die Datei `umlClassDiffConfig.xml` – sind Teil von SiDiff. In dieser XML-Datei wird für jeden Knotentyp konfiguriert, welche Änderungen an den Attributen und Beziehungen eines Elements sich in welchem Umfang auf das Matching der Knoten auswirkt. Die Konfiguration ist nicht Teil dieser Arbeit und wird unter anderem in [Weh04] näher beschrieben.

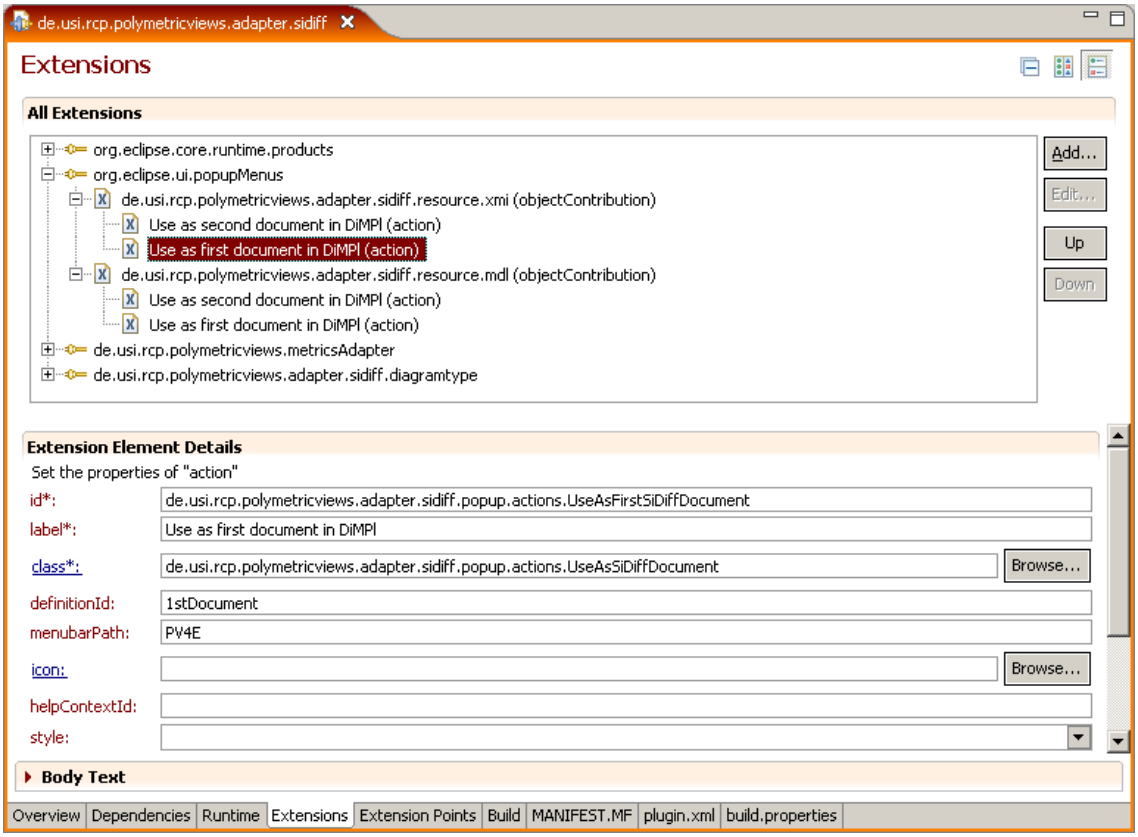


Abbildung 3.5.: Die Konfiguration der unterschiedlichen Dokumenttypen in der `plugin.xml` von DiMPl.

Nach dem Laden des zweiten zu vergleichenden Dokuments wird die Entity Selection View des PV4E-Plugins aktualisiert. Die Adapterschnittstelle von PV4E erwartet über die Methoden `getTreeContentProvider` und `getTreeLabelProvider` zwei Objekte, die dafür zuständig sind, die Struktur für den Auswahlbaum zu erstellen und zum anderen, die Namen und Icons der einzelnen Baumelemente zu erzeugen.

Die Erstellung des Baumes ist unproblematisch, da die Nesting-Kanten der SiDiff-internen Graphen eines Dokuments bereits eine Baumstruktur bilden und alle Elemente des Dokuments enthält. Diese Struktur konnte also ohne Transformation übernommen werden. Der `TreeLabelProvider` erzeugt für Baumelemente ein kleines Icon und einen Namen. Das Icon kann dabei individuell vom Knotentyp abhängig gemacht werden. So werden zum Beispiel UML-Klassen mit dem Klassensymbol von Eclipse im Baum dargestellt, wie in Abbildung 3.6 zu sehen ist. Für den Namen des Baumelements wird, sofern vorhanden das Attribut `name` des Elements im Modell benutzt. In anderen Diagrammtypen, wie zum Beispiel Simulink-Diagrammen ist der Name eines Elements nicht immer vergeben, sodass dann auf den Elementtyp in Kombination mit seiner eindeutigen ID zurückgegriffen wird.

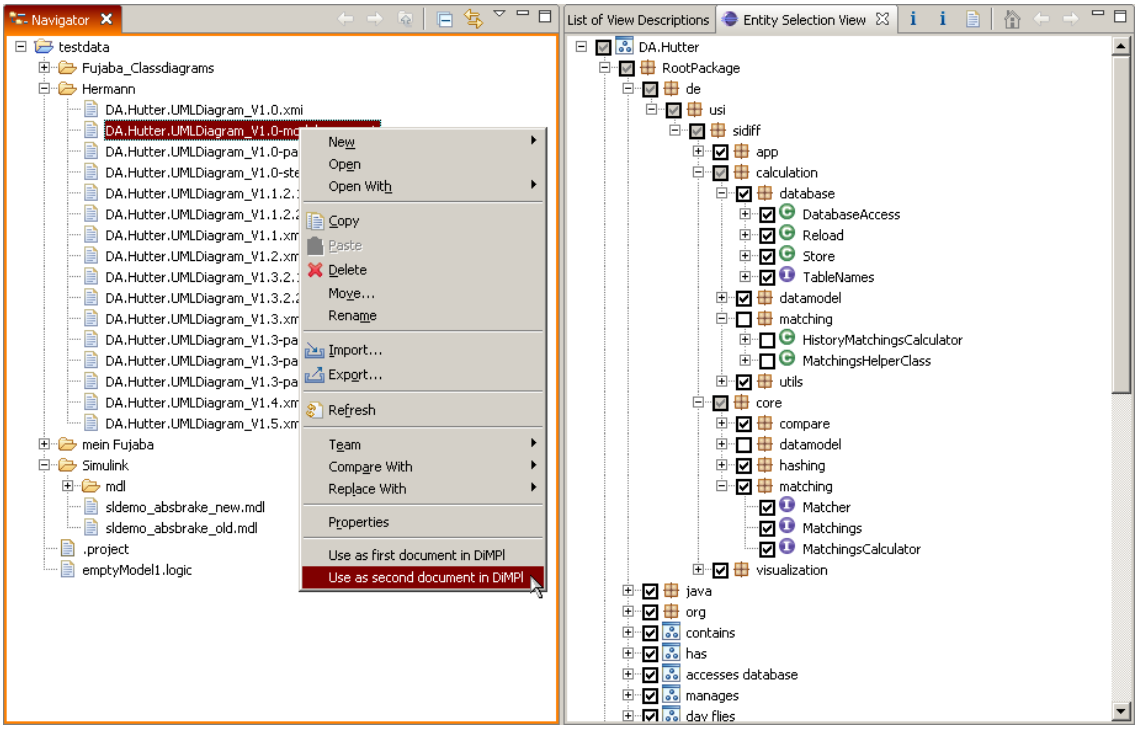


Abbildung 3.6.: Auswahl der Dokumente im *Resource Navigator* und Darstellung des zweiten Modells in der *Entity Selection View* des DiMPl.

Nachdem der Auswahlbaum erstellt wurde, können die Elemente dort nach Wunsch selektiert werden und in einer polymetrischen Sicht angezeigt werden.

3.2.3. Berechnung der Differenz und Kalkulation der Metriken

Ist die Auswahl der Dokumente und Elemente erfolgt, kann die Berechnung einer polymetrischen Sicht über die *List of View Descriptions* angestoßen werden. Dazu müssen die restlichen Phasen von SiDiff ausgeführt werden und die Berechnung der Metriken erfolgen. Dies wurde in der Schnittstellenmethode `calculateMetrics()` im Adapter realisiert. Ein Ausschnitt aus der Methode ist im nachfolgenden Quelltext abgebildet.

```
1  public void calculateMetrics(Object[] entities, String[] ↩
       metricNames)
2  {
3    [...]
4    // calculate path values
5    Annotations.getInstance().annotateGraph(firstGraph);
6    calculatePaths(firstGraph);
7    Annotations.getInstance().annotateGraph(secondGraph);
8    calculatePaths(secondGraph);
9
10   // calculate hash values
11   HashCalculator calculator = new TreeHashCalculator();
12   calculator.calculate(firstGraph);
13   calculator.calculate(secondGraph);
14
15   [...]
16
17   // match nodes by hash values
18   List<Matching> matchings = new HashMatchingsCalculator().↩
         calculate(firstGraph, secondGraph);
19   Matchings.removeOverlappingMatchings(matchings);
20   Matchings.getInstance(firstGraph, secondGraph).addMatchings(↩
         matchings);
21
22   // store hash-matched nodes
23   HashSet<Node> hashedNodes = new HashSet<Node>();
24   for (Matching matching : matchings)
25   {
26     hashedNodes.add(matching.getNode1());
27     hashedNodes.add(matching.getNode2());
28   }
29
30   // match nodes by iteration
31   new IterativeMatchingsCalculator().calculateMatchings(firstGraph↩
         , secondGraph);
32
33   // store symmetric difference
34   symmetricDifference = SiDiffDifferences.getSymmetricDifference(↩
         firstGraph, secondGraph);
35
36   // store similarity values
```

```
37    similarities = Similarities.getInstance(firstGraph, secondGraph)
          ;
38
39    // calculate difference metrics
40    DifferenceMetricsCalculationEngine dmce = new
          DifferenceMetricsCalculationEngine(symmetricDifference,
          firstGraph, secondGraph);
41    dmce.setNodeTypesToCalculate(currentDocumentType);
42    dmce.calculateDifferenceMetrics();
43
44    // calculate similarity metrics
45    [...]
46    dmce.setHashedNodes(hashedNodes);
47    dmce.setSimilarities(similarities);
48    dmce.setNodeTypeMapping(nodeTypeMapping);
49    dmce.calculateSimilarityMetrics(secondGraph.getRoot());
50    dmce.calculateSimilarityMetrics(firstGraph.getRoot());
51    [...]
52  }
```

Zunächst werden die eindeutigen Pfade zur Lokalisierung der Elemente und Hashwerte für die Elemente in den Graphen berechnet. Elemente mit gleichem Hashwert können so zwecks Laufzeitoptimierung vom Vergleichsalgorithmus ausgenommen werden. Danach erst wird der eigentliche Vergleich gestartet und die symmetrische Differenz berechnet. Die Zwischenergebnisse der SiDiff-Phasen werden zusätzlich gespeichert, da sie bei der Berechnung der Metriken in der `DifferenceMetricsCalculationEngine` (DMCE) noch benötigt werden.

Mit den verschiedenen Ergebnissen wird die DMCE initialisiert und zunächst die Differenzmetriken auf Basis der symmetrischen Differenz berechnet und anschließend die Ähnlichkeitsmetriken auf den Modellen. Ein explizites Anstoßen der Berechnung der Modellmetriken ist nicht vonnöten, da diese automatisch in SiDiff berechnet werden, wenn für einen Knoten der Wert aus einer Annotation angefordert wird, der dort noch nicht vorhanden ist. In allen Fällen werden die Metrikwerte zusammen mit einem kurzen Akronym für die Metrik in der Annotation des Knoten gespeichert.

Nachfolgend wird jeweils ein konkretes Beispiel für die Metrikberechnungen der drei Metriktypen aufgeführt. Weitere Metrikberechnungen können dem der Diplomarbeit beigefügten Quelltext oder direkt aus dem SVN-Repository des SiDiff-Projekts entnommen werden:

1. Modellmetrik

Zur Berechnung von *Anzahl Klassen in Paket*. Die Initialisierung der Methode erfolgte mit dem Parameter *class* und wird durch die Initialisierung der dazugehörigen *Query*-Klasse mit *package* als Knotentyp auf allen Paketknoten ausgeführt.

```
1 public class CountChildNodesTypes extends Annotator
```

```
2 {
3   public CountChildNodesTypes(String auxKey, String param, String
        docType, String name, int traverseOrder)
4   {
5     super(auxKey, param, docType, name, traverseOrder);
6   }
7
8   public Object getAnnotationValue(Node node)
9   {
10    Type type = TypeRegistry.getInstance().getType(this.
          getParameter());
11    List<Node> list = TreeAccess.getChildren(node);
12    Float count = 0f;
13    for (Node n: list)
14    {
15      if (n.getNodeType() == type) count += 1;
16    }
17    return count;
18  }
19 }
```

2. Differenzmetrik

Zur Berechnung von *Anzahl Klassen in Paket die Attribut Name geändert haben.* Die Initialisierung der Methode erfolgte mit den Parametern *name*, *class* und *classes* und erhöht bei allen Klassen, auf die die Bedingung zutrifft den Metrikwert für diese Metrik beim Vaterelement um 1.

```
1 private void calculateNOYChangedAttr(String unorderedAttrName,
      NodeType nestedNodeType, EdgeType nestingEdgeType)
2 {
3   // NO[nestedNodeType]Changed[AttrName] - Alle Subelemente von
        Element aus SymmetricDifference.Updates UND in Graph 2 UND
        Graph2.Y.AttrName != Graph1.Y.AttrName
4
5   // ...aus SymmetricDifference.Updates...
6   Set<Update> updates = symmetricDifference.getUpdates();
7
8   for (Update update : updates)
9   {
10    Node aNode;
11    // ...UND in Graph 2
12    aNode = update.getNodeInSetB();
13
14    // Alle Subelemente...
15    if (aNode.getNodeType() == nestedNodeType)
16    {
17      // ...UND Graph2.Y.AttrName != Graph1.Y.AttrName
18      if (update.getAttributeName().equals(unorderedAttrName))
```

```
19        {
20          // ...erhöhe beim Vater des Elements den Metrikwert um 1
21          String acronym = "NO" + nestedNodeType.getName() + "
                Changed" + unorderedAttrName;
22          increaseParentMetricValue(aNode,nestingEdgeType, acronym);
23        }
24      }
25    }
26  }
```

3. Ähnlichkeitsmetrik

Zur Berechnung von *Maximale Unähnlichkeit direkter Kinder vom Typ Klasse in Paket*. Die Übergabeparameter sind in diesem Fall unter anderem das *Paket*. Per Rekursion wird der jeweilige Unähnlichkeitswert der Kindelemente berechnet und der Maximalwert aller Klassen als Metrikwert für das Paket übernommen.

```
1  public void calculateSimilarityMetrics(Node node, ComplexNodeType
       complexNodeType)
2  {
3    [...]
4    // get all nesting children
5    List<Node> neighbors = GraphAccess.getOutgoingNodeNeighbors(node
         , Nesting.class);
6    [...]
7    for (Node nestingNode : neighbors)
8    {
9      // for each child calculate the similarity metrics recursivly
10     calculateSimilarityMetrics(nestingNode, nodeTypeMapping.get(
           nestingNode.getNodeType()));
11     [...]
12     // the dissimilarity value of nesting node is needed for...
13     dissValue = getValueAsFloat(nestingNode, "DIFF");
14     [...]
15     // ...the maximum difference of direct descendant children of
           a specific type
16     String acronym="MDDCOT"+nestingNode.getNodeType().getName();
17     nodesAux.setNodeInformation(node, acronym, Math.max(
           getValueAsFloat(node, acronym), dissValue));
18     [...]
19   }
20   // calculate difference value for the current node
21   nodesAux.setNodeInformation(node, "DIFF",
         calculateDifferenceValue(node));
22 }
```

Abschließend sei angemerkt, dass die symmetrische Differenz der beiden Dokumente respektive die unterschiedlichen Metriken auf den Graphelementen, per geeigneter Fallunterscheidung nur berechnet wird, wenn mindestens eines der zu vergle-

chenden Dokumente neu ausgewählt wurde. Ansonsten würde bei jeder einzelnen Sicht, die auf den gleichen Modellen berechnet würde, jedes mal die Modelldifferenz neu berechnet.

3.2.4. Erzeugung der Knoten und Kanten anhand der Metrikwerte

Nach der Berechnung aller benötigten Metriken werden die Knoten und Kanten der polymetrischen Sicht in den entsprechenden Knoten- und Kantenprovidern erzeugt. Wie bereits in 3.1.4 beschrieben, sollte für die Darstellung einer polymetrischen Sicht auf Modelldifferenzen, die symmetrische Differenz benutzt werden, sodass auch die gelöschten Elemente angezeigt werden können.

Aus Sicht von DiMPl musste dazu lediglich die Methode `getMetricValue` implementiert werden. Sie sorgt dafür, dass die Metrikwerte aus dem Graphen geholt werden können und in den Providern für die Erzeugung der Knoten zur Verfügung stehen. Die Methoden `getNodeProvider, getEdgeProvider` sind bereits in ihrer abstrakten Form ausreichend, da lediglich anhand des Elementtyps entschieden werden muss, welcher Provider übergeben wird.

Zuständig für die Erzeugung von Knoten für die Graphelemente aus der SiDiff-Datenstruktur ist im wesentlichen der *SiDiffNodeProvider*, der nach Algorithmus 3 vorgeht.

Algorithmus 3 : SiDiffNodeProvider für Knotentyp nodeType

```
1  für alle Kategorien der symmetrischen Differenz tue
2      für alle Elemente der Kategorie tue
3          wenn Element vom Typ nodeType dann
4              wenn Element im erstem Graph dann
5                  solange Vaterelement keinen Partnerelement hat tue
6                      Suche Vaterelement
7                  Ende
8                  wenn Partnerelement des Vaterelements im Baum selektiert dann
9                      Erzeuge Knoten
10                 Ende
11             sonst
12                 wenn Element im Baum selektiert dann
13                     Erzeuge Knoten
14                 Ende
15             Ende
16         Ende
17     Ende
18 Ende
```

Die Unterscheidung bei gelöschten Elementen ist notwendig, damit diese wie im Konzept vorgesehen gezeichnet werden. Kurz gesagt wird bei einem gelöschtes Element für dessen ersten nicht gelöschten Vaterknoten überprüft, ob der Partnerknoten im Auswahlbaum selektiert wurde.

Wenn festgellt wird, dass ein Element gezeichnet werden soll, wird ein neues Knotenelement für die polymetrische Sicht erzeugt. Es wird mit dem Graphelement verknüpft und erhält neben dem Namen des Graphelements, die Eigenschaften gemäß den zugewiesenen Metriken. Die Metrikwerte werden dazu über die Methode `getMetricValue` des Adapters aus der Annotation des Graphelements ausgelesen und als Größe, Position und Farbe des Knoten benutzt.

Als Besonderheit des Knotenproviders für Modelldifferenzen wird an dieser Stelle die Rahmenfarbe gesetzt, entsprechend der Kategorie der symmetrischen Differenz in der sich der Knoten befindet. Zusätzlich wird in das Tooltip der grafischen Knotenelemente der Sicht eine Beschreibung der Änderung geschrieben. Befindet sich ein Knoten in mehreren oder mehrfach in einer der Kategorien, so wird die Rahmenfarbe auf die Farbe Cyan gesetzt und das Tooltip um die Änderung ergänzt.

Der Anwender von PV4E erhält also neben dem zusätzlichen visuellen Merkmal der Rahmenfarbe noch die genaue Information darüber, was sich an dem Graphelement geändert hat. Jedes Element besitzt also mindestens einen der folgenden Kommentare:

Equal: 'operation' is unchanged

Update: 'parameter' updated attribute 'kind' from 'in' to 'inout'

Structural: 'association' has been created *oder* 'association' has been deleted

Move: 'package' has moved from 'package: core' to 'package: kern'

Reference: 'class' changed reference from 'null' to 'stereotype: reference'

Der `SiDiffNodeProvider` benutzt standardmäßig für den Namen eines grafischen Knotens den von der Pfadberechnung erzeugten Pfad. Dieser entspricht nicht immer dem bekannten Namensschema der Dokumentelemente. Im Anwendungsfall von Klassendiagrammen wird für die Identifizierung einer Klasse üblicherweise der voll qualifizierte Klassenname benutzt. Daher wurde für die unterschiedlichen Elementtypen eines Klassendiagramms der `SiDiffNodeProvider` erweitert und die Methode für die Vergabe des Namens des grafischen Knotens überschrieben. Als Beispiel sei hier der Quelltext des `UMLClassNodeProvider` aufgeführt, der den voll qualifizierenden Klassennamen aus dem von der Pfadberechnung erzeugten Namen transformiert.

```
1 public class UMLClassNodeProvider extends SiDiffNodeProvider
2 {
```

```
3   public UMLClassNodeProvider(String name)
4   {
5     super(name, "class");
6   }
7
8   protected String getTransformedNodeName(String name)
9   {
10    return "Class:" + name.substring(name.indexOf("RootPackage/")
          + 12).replace("/", ".");
11  }
12 }
```

Die Erzeugung der Kanten in den polymetrischen Sichten ist oftmals diagrammabhängig. So existieren Vererbungsbeziehungen, die zwei *class*-Elemente über ein *generalization*-Element verbinden nur in UML-Klassendiagrammen. Die Implementierung musste also individuell für die unterschiedlichen Beziehungstypen erfolgen, die durch die Dokumentstruktur vorgegeben wurden.

Im Falle der Klassendiagramme wurden implementiert:

- Assoziationen zwischen Elementen vom Typ *class* über Elemente vom Typ *associationEnd*.
- Assoziationen zwischen Elementen vom Typ *associationEnd*.
- Vererbungen zwischen Elementen vom Typ *class* über Elemente vom Typ *generalization*, wahlweise nur Interface-Implementierungen, nur Klassen-Erweiterungen oder beides.
- Die baumartige Hierarchie zwischen Elementen vom Typ *packet*

Die hierarchische Paket-Subpaket-Beziehung ergibt sich dadurch, dass Pakete in UML-Klassendiagrammen Containerelemente sind, die wiederum Elemente des gleichen Typs enthalten können. Dies wurde ausgenutzt, um einen generischen `NodeTypeHierachyEdgeProvider` zu implementieren, der mit dem entsprechenden Typ des Containerelements initialisiert wird. Dies kann zum Beispiel für Simulink-Diagramme ausgenutzt werden, bei denen System-Subsystem-Beziehungen bestehen.

Mit dem Difference Metrics Plugin steht nun an dieser Stelle ein Werkzeug zur Verfügung, welches die Analyse von Modelldifferenzen ermöglicht. Dies soll im nachfolgenden Kapitel am Beispiel von Klassendiagrammen durchgeführt werden.

Kapitel 4.

Sichtdefinitionen zur Analyse von Klassendiagrammen

In diesem Kapitel werden eine Reihe von Sichtdefinitionen vorgestellt, die zur Analyse der Modelldifferenzen zwischen UML-Klassendiagrammen geeignet sind. Diese Sichten entstanden im Rahmen der Diplomarbeit während der Analyse diverser Testdaten, die innerhalb der Fachgruppe zur Verfügung stehen.

Die Definitionen sind nach einem bestimmten Schema aufgebaut, welches an die Definitionen der Sichten nach Lanza [Lan03] angelehnt ist.

Zunächst folgt nach dem Namen der polymetrischen Sicht deren genaue Definition in tabellarischer Form. Die Tabelle enthält unter anderem die Eingaben, welche im Sichteneditor von PV4E vorzunehmen sind. Darüber hinaus enthält sie eine Angabe, für welchen Bereich des Modells sich diese Sicht am besten eignet und einen Verweis auf Beispiele aus der praktischen Anwendung. Die Beispiele sind meist im Anschluss an die Definition zu finden. Die Funktionsweise der in der Tabelle genannten Layouter aus PV4E wird in Anhang C näher erläutert.

Die der Tabelle nachfolgenden Abschnitte enthalten eine Reihe von Erläuterungen, die den Einsatzzweck der Sicht beschreiben und speziell auf die Anwendung für die Analyse von Modelldifferenzen zugeschnitten sind.

Der Abschnitt **Beschreibung** gibt einen Überblick darüber, wann und unter welchen Voraussetzungen die polymetrische Sicht im Analyseablauf eingesetzt werden sollte.

Der **Informationsgewinn** beschreibt, wie der Name schon sagt, die Information, die aus der Visualisierung der ermittelten Daten aus der Modelldifferenz zu gewinnen ist. Er ist unterteilt in vier verschiedene Kategorien, die weiter unten genauer beschrieben werden.

Mit den **Symptomen** werden eine Reihe von einzelnen oder kombinierten, sichtabhängigen Eigenschaften beschrieben, anhand denen bestimmte Charakteristiken der Differenz direkt ablesbar sind.

Die angegebenen **Varianten** einer Sichtdefinition sind dazu geeignet, spezielle Aspekte in der Darstellung der Differenz der Modelle gesondert hervorzuheben.

Es hat sich gezeigt, dass mittels einer polymetrischen Sicht verschiedene, bewertende Aussagen über die Relevanz der Modelländerungen getroffen werden können – der sogenannte Informationsgewinn. Er ergibt sich aus den verschiedenen visuellen Knoten- und Kantenmerkmalen und deren textuellen Anmerkungen in dieser Sicht. Die Bewertungen sind dabei direkt abhängig von den Eigenschaften die sich an Veränderungen feststellen lassen. Diese Eigenschaften können in verschiedene Stufen eingeteilt werden, die im Folgenden mit **Veränderungskategorie** (VK) bezeichnet werden.

VK1 - Es gibt Änderungen ja/nein: Diese Eigenschaft zeigt sich in den Knoten durch einen Metrikwert, der von `0` verschieden ist. Das bedeutet konkret, der Knoten *hat* eine Breite beziehungsweise Höhe – die größer ist als die Mindestknotengröße – er befindet sich abseits des Ursprungs in horizontaler oder vertikaler Position, er besitzt eine dunklere Schattierung als weiß oder die Rahmenfarbe ist nicht schwarz.

VK2 - Menge von Änderungen: Diese Form der Veränderung kann ermittelt werden, in dem der genaue Wert der verknüpften Metrik aus dem Tooltip des Knotens abgelesen wird. Ist die Anzahl der Knoten einer Sicht überschaubar, kann auch das Abzählen der Knoten für eine Ermittlung dieser Eigenschaft in Frage kommen.

VK3 - Stärke von Änderungen: Die Stärke von Änderungen kann zum Teil ebenfalls aus dem Tooltip in Form des Unähnlichkeitswertes abgelesen werden. Sie kann aber auch bereits durch eine Interpretation der präsentierten Knoten gewonnen werden. Dies kann das Verhältnis von Breite zu Höhe eines Knoten sein, die Position eines Knoten zu einer gedachten Linie, die Länge von nebeneinander gestapelten Knoten, oder weitere Kombinationen der Knotenattribute in Verbindung mit dem Layout einer polymetrischen Sicht sein.

VK4 - Wichtigkeit von Änderungen: Diese Eigenschaft von Änderungen ist nicht allein aus den Merkmalen einer Sicht zu ermitteln. Die Ermittlung erfordert zum einen gewisse Kenntnisse in der Struktur der zu analysierenden Modelle und zum anderen von den Inhalten der Modelle selber. Manche Änderungen können auch nur durch das Wissen über die Projektvorgaben bei der Entwicklung der Modelle beurteilt werden. Beispielsweise ist die Wichtigkeit der Erkenntnis, dass der Stereotyp *Singleton* in der zweiten Revision eines Klassendiagramms nicht mehr benutzt wird abhängig von einem äußeren Faktor. War die Auflage der Projektleitung, dass das dazugehörige Entwurfsmuster nicht mehr benutzt werden soll, so ist dies für den Anwender des Werkzeugs eine wichtige Erkenntnis. Die Interpretation der Wichtigkeit von Änderungen ist also in vielen Fällen das Ergebnis von Kombinationen mehrerer Attribute einer polymetrischen Sicht ergänzt um zusätzliche Informationen von außen.

Allgemein hat sich herausgestellt, dass das Vorgehen bei der Analyse einer neu zu untersuchenden Modelldifferenz gewissermaßen von oben nach unten bezüglich der Hierarchie der Modellelemente erfolgen sollte. Daher ist eine gewisse Kenntnis der Dokumentstruktur vonnöten.

Es sollte zunächst auf der Ebene des Wurzelelement begonnen werden. Dieses Element sollte anhand von Sichten dieses Elementtyps auf Änderungen untersucht werden. Danach die Elementtypen der direkten Kindelemente aus der Sicht des Wurzelelements und anschließend die Kindelemente direkt mit deren Elementtypen als Knoten. Dies sollte so weitergeführt werden, bis die hierarchisch tiefsten Elemente der Struktur der Elementtypen erreicht ist. Als sehr nutzvoll für die Analyse haben sich dabei diejenigen Sichten erwiesen, bei denen die Knoten der Entitäten durch Beziehungstypen miteinander verbunden sind.

Diese Top-Down-Vorgehensweise hat den Vorteil, dass wenn sich herausstellt, dass an einem Elementtyp keine Änderungen stattgefunden haben, die polymetrischen Sichten aus der Sicht dieses Typs nicht notwendig sind und bei der weiteren Analyse übersprungen werden können.

Nachfolgend die Sichten für Klassendiagramme, die sich auf die beschriebene Weise von oben nach unten durch die Dokumentstruktur hangeln und über die Modelländerungen Aufschluss geben. Für einen besseren Überblick ist die interne Datenstruktur von Klassendiagrammen in Anhang A angegeben.

Um die Reihenfolge der Analyse auch im PV4E-Plugin deutlich zu machen, ist ein Namensschema mit einer großzügigen Nummerierung benutzt worden. Der Teil **cd** steht dabei für die Abkürzung des Begriffs *class diagram*. Danach folgt eine **Nummer**, die mit Absicht so gewählt wurde, dass weitere Sichten dazwischengefügt werden können. Abschließend der zu untersuchende **Elementtyp**, beziehungsweise der zu betrachtenden Elementtypen nach dem Bindestrich. Die Abkürzung **DDC** steht für *direct decendant children*, **ADC** für *all descendant children*.

Die Sichten gehen nach dem oben beschriebenen Verfahren nur bis zu einer allgemeinen Analyse der Kindelemente und verweisen dann auf eine weitere Analyse anhand der Originaldiagramme, um die Veränderungen genau zu betrachten. Die weitere Analyse kann auch mit Hilfe der polymetrischen Sichten weitergeführt werden, jedoch erfordert dies speziell auf die Änderungen zugeschnittene Sichten. Diese speziellen Sichten sollten projektabhängig individuell erstellt werden, um beispielsweise Fragen zu beantworten, wie „Anzahl Methoden, deren Sichtbarkeit größer wurde in der Klasse“.

Anzumerken ist, dass die Sichtdefinitionen in deutscher Sprache angegeben sind, um die Verständlichkeit der einzelnen Beschreibungen zu erhöhen. Die Projektsprache für die Quelltexte und auch die Bedienungsoberfläche von PV4E sind dagegen komplett in Englisch gehalten, was sich in den diversen Bildschirmfotos wiederspiegelt.

4.1. Sichten für Modelle

cd 100 model

Layout	Ohne Layout
Knoten	Modell
Kanten	—
Auswahl	Gesamtes System
Metriken	
Breite	Anz. aller Kindelemente
Höhe	Anz. Unterschiede aller Kindelemente
X-Position	Anz. Unterschiede direkter Kindelemente vom Typ Datentyp
Y-Position	Anz. Unterschiede direkter Kindelemente vom Typ Stereotyp
Farbe	Durchschnittliche Unähnlichkeit aller Kindelemente
Sortierung	—
Beispiel	Abbildung 4.1

Beschreibung

Diese sehr einfache Sicht bietet einen allerersten Überblick über die Menge der Elemente und die Größe der Änderungen dazwischen. Obwohl nur ein einziger Knoten gezeichnet wird, lassen sich sehr schnell einige Aussagen über die zu analysierende Modelldifferenz treffen.

Informationsgewinn

VK1: Die Knotenhöhe und der Abstand zum Bildschirmrand zeigen an, dass es Änderungen an den Kindelementen beziehungsweise den Stereo- und Datentypen gibt.

VK2: Die Metriken für Höhe und Position im Tooltip enthalten die genauen Werte für die Anzahl der Veränderungen.

VK3: Das Verhältnis von Breite zu Höhe des Knoten und der Wert für die Farbe, sind ein Hinweis auf die Stärke der Veränderungen zwischen den Modellrevisionen.

VK4: Eine Änderung an den Stereo- und Datentypen ist aus Entwicklersicht wichtig und erfordert eine genauere Untersuchung der Änderungen. Dazu sind die beiden folgenden Sichtdefinitionen geeignet und auch die weiterführenden in Abschnitt 4.2 und 4.3.

Symptome

Größe: Je breiter der Knoten, desto größer die verglichenen Modelle.

Höhe: Je höher der Knoten, desto stärker die Veränderung zwischen den beiden Modellen.

Position: Je weiter der Knoten vom Rand entfernt ist, desto mehr Änderungen gab es an Stereo- und Datentypen.

Tooltip: Der Wert für die Farbmetrik gibt ebenfalls Auskunft über die Stärke der Änderungen und kann mit anderen Modelldifferenzen verglichen werden.

Varianten

Var. 1: Wenn für das Layout der *Scatterplot Layouter* benutzt wird, befindet sich der Knoten bei Änderungen an den Stereo- und Datentypen deutlich weiter vom Rand weg.

Anmerkung

Um die Stärke der Veränderung bei verschiedenen Modelldifferenzen zu betrachten muss derzeit jeder Versionssprung einzeln betrachtet werden. Diese Sichtdefinition könnte aber auch in einem Werkzeug nützlich sein, welches in der Lage ist, mehre Differenzen miteinander zu vergleichen. Siehe dazu auch Kapitel 6.

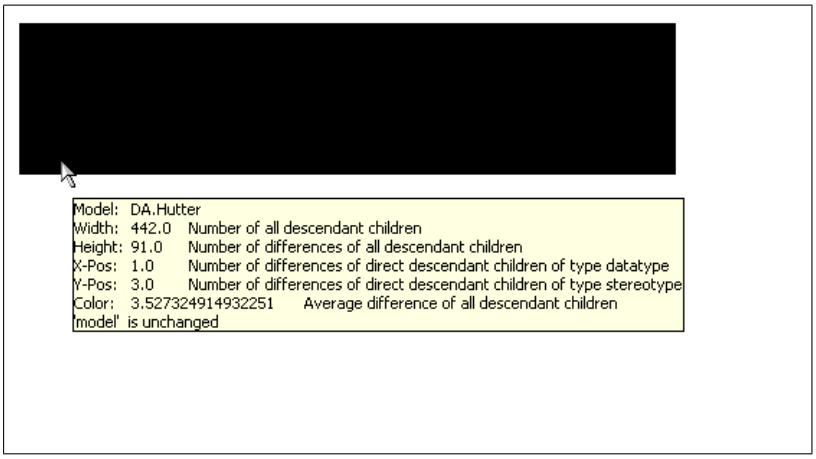


Abbildung 4.1.: Ein einzelner großer Modellknoten, der aber bereits Aussagen über die Stärke der Veränderungen am Gesamtsystem zulässt.

cd 110 model-stereotype

Layout	Scatterplot Layout
Knoten	Modell
Kanten	—
Auswahl	Gesamtes System
Metriken	
Breite	Anz. Stereotypen im Modell
Höhe	Anz. Stereotypen unverändert
X-Position	Anz. Stereotypen hinzugefügt
Y-Position	Anz. Stereotypen entfernt
Farbe	Anz. Stereotypen die ihren Namen geändert haben
Sortierung	—
Beispiel	Abbildung 4.2

Beschreibung

Eine weitere einfache Sicht, die genauere Informationen über die Art der Veränderungen an den Stereotypen der Dokumente gibt. Sie sollte im Anschluss an die Sicht 4.1 benutzt werden, falls Änderungen an Stereotypen festgestellt wurden.

Informationsgewinn

VK1: Die Knotenhöhe und der Abstand zum Bildschirmrand zeigen an, dass es Änderungen an den Stereotypen gibt.

VK2: Die Metriken für Position und Farbe im Tooltip enthalten die genauen Werte für die Anzahl der Veränderungen an den Stereotypen.

VK3: Eine geringe Höhe des Knoten ist ein Hinweis auf starke Veränderungen an den Stereotypen zwischen den Modellrevisionen.

VK4: Veränderungen an den Stereotypen sind aus Entwicklersicht wichtig und erfordern eine genauere Untersuchung der Änderungen. Um konkrete Informationen über die betroffenen Stereotypen zu erhalten und deren Einfluss auf das System sollten die Sichtdefinitionen in Abschnitt 4.2 genutzt werden.

Symptome

Größe: Je breiter der Knoten, desto mehr Stereotypen enthält das System.

Höhe: Je niedriger der Knoten, desto stärker die Veränderung an den Stereotypen zwischen den beiden Modellen.

Position: Je weiter der Knoten vom Rand entfernt ist, desto mehr Stereotypen wurden gelöscht oder hinzugefügt.

Tooltip: Der Wert für die Farbmetrik gibt Auskunft über Namensänderungen der Stereotypen.

Varianten

Var. 1: Je nachdem, ob der Anwender bei der Analyse der Stereotypen mehr Wert auf die Position oder die Form des Knoten legt, können die entsprechenden Metriken paarweise miteinander vertauscht werden.

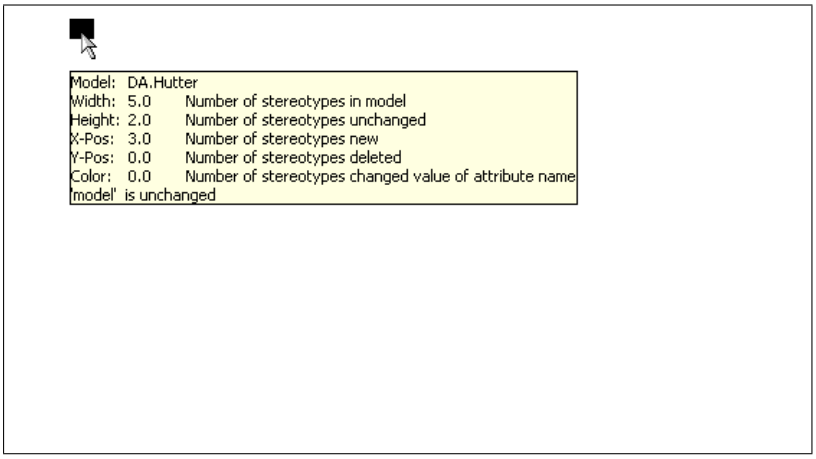


Abbildung 4.2.: Ein einzelner Modellknoten, der Aussagen über die Stärke der Veränderungen an Stereotypen zulässt.

cd 120 model-datatype

Layout	Scatterplot Layout
Knoten	Modell
Kanten	—
Auswahl	Gesamtes System
Metriken	
Breite	Anz. Datentypen im Modell
Höhe	Anz. Datentypen unverändert
X-Position	Anz. Datentypen hinzugefügt
Y-Position	Anz. Datentypen entfernt
Farbe	Anz. Datentypen die ihren Namen geändert haben
Sortierung	—
Beispiel	Abbildung 4.3

Beschreibung

Eine weitere einfache Sicht, die genauere Informationen über die Art der Veränderungen an den Datentypen der Dokumente gibt. Sie sollte im Anschluss an die Sicht 4.1 benutzt werden, falls Änderungen an Datentypen festgestellt wurden.

Informationsgewinn

VK1: Die Knotenhöhe und der Abstand zum Bildschirmrand zeigen an, dass es Änderungen an den Datentypen gibt.

VK2: Die Metriken für Position und Farbe im Tooltip enthalten die genauen Werte für die Anzahl der Veränderungen an den Datentypen.

VK3: Eine geringe Höhe des Knoten sind ein Hinweis auf starke Veränderungen an den Datentypen zwischen den Modellrevisionen.

VK4: Veränderungen an den Datentypen sind aus Entwicklersicht wichtig und erfordern eine genauere Untersuchung der Änderungen. Um konkrete Informationen über die betroffenen Datentypen zu erhalten und deren Einfluss auf das System sollten die Sichtdefinitionen in Abschnitt 4.3 genutzt werden.

Symptome

Größe: Je breiter der Knoten, desto mehr Datentypen enthält das System.

Höhe: Je niedriger der Knoten, desto stärker die Veränderung an den Datentypen zwischen den beiden Modellen.

Position: Je weiter der Knoten vom Rand entfernt ist, desto mehr Datentypen wurden gelöscht oder hinzugefügt.

Tooltip: Der Wert für die Farbmetrik gibt Auskunft über Namensänderungen der Datentypen.

Varianten

Var. 1: Je nachdem, ob der Anwender bei der Analyse der Datentypen mehr Wert auf die Position oder die Form des Knoten legt, können die entsprechenden Metriken paarweise miteinander vertauscht werden.

Anmerkung

Der Fall, dass Datentypen umbenannt werden sollte nach der Standardkonfiguration des SiDiff-Algorithmus nicht eintreten.

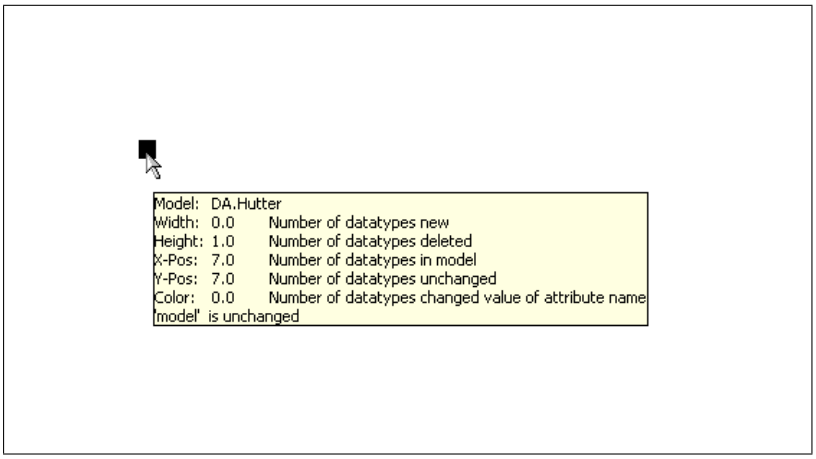


Abbildung 4.3.: Ein einzelner Modellknoten, der Aussagen über die Stärke der Veränderungen an Datentypen zulässt.

4.2. Sichten für Stereotypen

cd 130 stereotype

Layout	Checker Board Layout
Knoten	Stereotypen
Kanten	—
Auswahl	Gesamtes System
Metriken	
Breite	Anzahl Unterschiede
Höhe	Anzahl Unterschiede
X-Position	—
Y-Position	—
Farbe	Unähnlichkeit
Sortierung	Farbe
Beispiel	Abbildung 4.4

Beschreibung

Diese Sicht zeigt dem Benutzer, welche Änderungen speziell an den Stereotypen eines Klassendiagramms vorgenommen wurden. Er ist dadurch in der Lage sofort zu erkennen, wie sich die Nutzung der Stereotypen zwischen den Revisionen der Dokumente geändert hat.

Informationsgewinn

VK1: Die Breite, Höhe und Rahmenfarbe eines Knotens zeigen direkt an, dass der Stereotyp verändert wurde.

VK2: Die Metrikwerte für Breite oder Höhe geben Aufschluss über die Anzahl der Änderungen an einem Stereotyp. Die Anzahl der geänderten Stereotypen ergibt sich durch Abzählen oder ist bereits aus 4.1 bekannt.

VK3: Das Verhältnis aus unveränderten (schwarzen) und veränderten (bunten) Knoten bestimmt die Stärke der Änderungen an den Stereotypen.

VK4: Die Wichtigkeit der Änderungen an den Stereotypen ergibt sich aus dem Namen des Stereotyps und eventuell vorhandenen Projektvorgaben. Wurde etwa im Laufe der Entwicklung gefordert, dass das Entwurfsmuster „Singleton“ nicht mehr benutzt werden soll, und der Knoten für diesen Stereotyp ist rot umrandet, dann wurde das Entwicklungsziel erreicht.

Symptome

Position: Die geänderten Stereotypen befinden sich am rechten unteren Rand der polymetrischen Sicht.

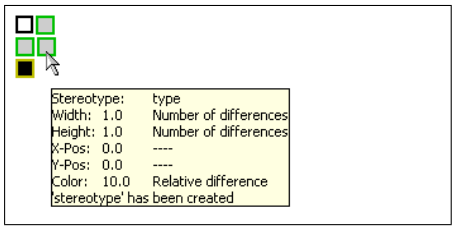
Tooltip: Der Name des Stereotyps wird für die Beurteilung der Wichtigkeit der Änderungen benötigt.

Farbe: Je dunkler der Knoten, desto stärker die Änderung an dem Stereotyp.

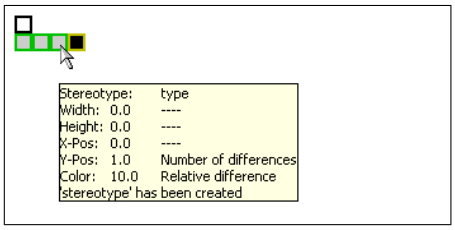
Varianten

Var. 1: Wenn die *Anzahl Unterschiede* statt für die Größe eines Knoten für die vertikale Position in Kombination mit dem *Vertical Histogram Layout* benutzt wird, dann wird die Beurteilung der Stärke der Änderungen vereinfacht. Wie in Abbildung 4.4(b) zu sehen ist, befinden sich unveränderte Knoten in der ersten Zeile und alle veränderten in der zweiten.

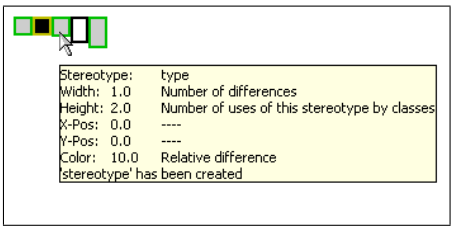
Var. 2: Wird das *Horizontal Stapled Layout* mit einer Sortierung nach der *Höhe* der Knoten und für deren Höhe die *Anzahl Nutzungen des Stereotyps durch eine Klasse* benutzt, so wird dadurch die Beurteilung der Wichtigkeit des Stereotyps für das System vereinfacht. Häufig benutzte Stereotypen befinden sich dann weiter rechts in der Sicht, wie in Abbildung 4.4(c) zu sehen.



(a) cd 130 stereotype



(b) cd 130 stereotype var1



(c) cd 130 stereotype var2

Abbildung 4.4.: Diese Sichten sind für die Beurteilung der Änderungen an den Stereotypen gut geeignet.

4.3. Sichten für Datentypen

cd 160 datatype

Layout	Checker Board Layout
Knoten	Datentypen
Kanten	—
Auswahl	Gesamtes System
Metriken	
Breite	Anzahl Unterschiede
Höhe	Anzahl Unterschiede
X-Position	—
Y-Position	—
Farbe	Unähnlichkeit
Sortierung	Farbe
Beispiel	Abbildung 4.5

Beschreibung

Diese Sicht zeigt dem Benutzer, welche Änderungen speziell an den Datentypen eines Klassendiagramms vorgenommen wurden. Er ist dadurch in der Lage sofort zu erkennen, wie sich die Nutzung der Datentypen zwischen den Revisionen der Dokumente geändert hat.

Informationsgewinn

VK1: Die Breite, Höhe und Rahmenfarbe eines Knotens zeigen direkt an, dass der Datentyp verändert wurde.

VK2: Die Metrikwerte für Breite oder Höhe geben Aufschluss über die Anzahl der Änderungen an einem Datentyp. Die Anzahl der geänderten Datentypen ergibt sich durch Abzählen oder ist bereits aus 4.1 bekannt.

VK3: Das Verhältnis aus unveränderten (schwarzen) und veränderten (bunten) Knoten bestimmt die Stärke der Änderungen an den Datentypen.

VK4: Die Wichtigkeit der Änderungen an den Datentypen ergibt sich wie schon bei den Stereotypen aus dem Namen des Datentyps und eventuell vorhandenen Projektvorgaben.

Symptome

Position: Die geänderten Datentypen befinden sich am rechten unteren Rand der polymetrischen Sicht.

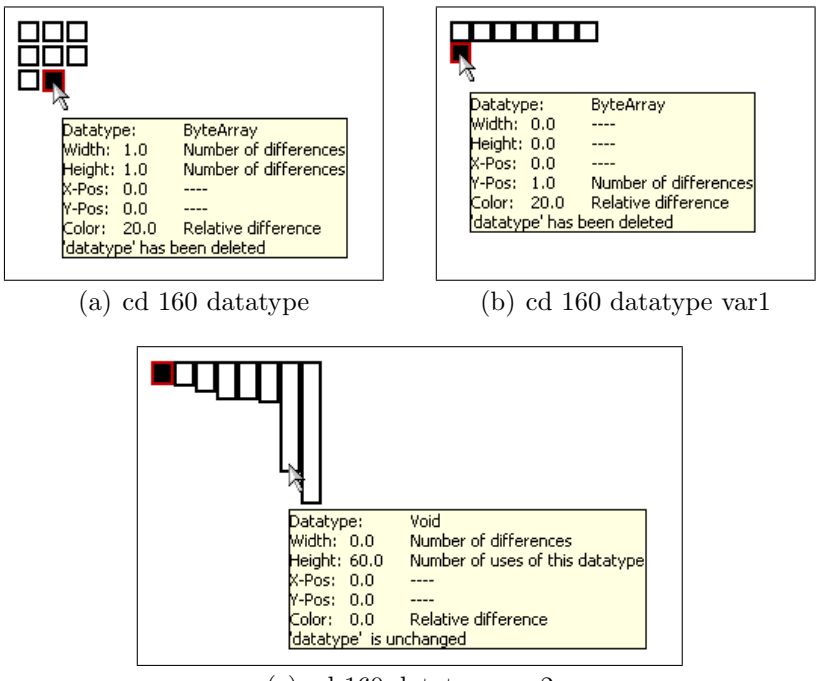
Tooltip: Der Name des Datentyps wird für die Beurteilung der Wichtigkeit der Änderungen benötigt.

Farbe: Je dunkler der Knoten, desto stärker die Änderung an dem Datentyp.

Varianten

Var. 1: Wenn die *Anzahl Unterschiede* statt für die Größe eines Knoten für die vertikale Position in Kombination mit dem *Vertical Histogram Layout* benutzt wird, dann wird die Beurteilung der Stärke der Änderungen vereinfacht. Wie in Abbildung 4.5(b) zu sehen ist, befinden sich unveränderte Datentypen in der ersten Zeile und alle veränderten in der zweiten.

Var. 2: Wird das *Horizontal Stapled Layout* mit einer Sortierung nach der *Höhe* der Knoten und für deren Höhe die *Anzahl Nutzungen des Datentyps durch eine Klasse* benutzt, so wird dadurch die Beurteilung der Wichtigkeit des Datentyps für das System vereinfacht. Häufig benutzte Datentypen befinden sich dann weiter rechts in der Sicht, wie in Abbildung 4.5(c) zu sehen.



(a) cd 160 datatype (b) cd 160 datatype var1

(c) cd 160 datatype var2

Abbildung 4.5.: Diese Sichten sind für die Beurteilung der Änderungen an den Datentypen gut geeignet.

4.4. Sichten für Pakete

cd 200 package

Layout	Vertical Tree Layout
Knoten	Pakete
Kanten	Unterpaket zu Paket
Auswahl	Gesamtes System
Metriken	
Breite	Unähnlichkeit
Höhe	Unähnlichkeit
X-Position	—
Y-Position	—
Farbe	Unähnlichkeit
Sortierung	Name
Beispiel	Abbildung 4.6

Beschreibung

Diese polymetrische Sicht ist dazu geeignet, Änderungen an der Paketstruktur eines Softwaresystems zu erkennen. Durch die hierarchische Darstellung der enthaltenen Pakete fallen die Unterschiede unmittelbar durch eine eindeutige Rahmenfarbe auf.

Informationsgewinn

VK1: Die Rahmenfarbe zeigt auf den ersten Blick, dass eine Veränderung am Paket stattgefunden hat.

VK2: Die Rahmenfarbe und der Tooltip geben Auskunft über die Anzahl der Veränderungen am Paket.

VK3: Die Größe und Farbe des Knotens zeigen die Stärke der Änderungen am Paket an.

VK4: Position und Name des Pakets, zusammen mit der Art der Änderung, sind nützlich für die Beurteilung der Wichtigkeit der Änderungen für das Gesamtsystem.

Symptome

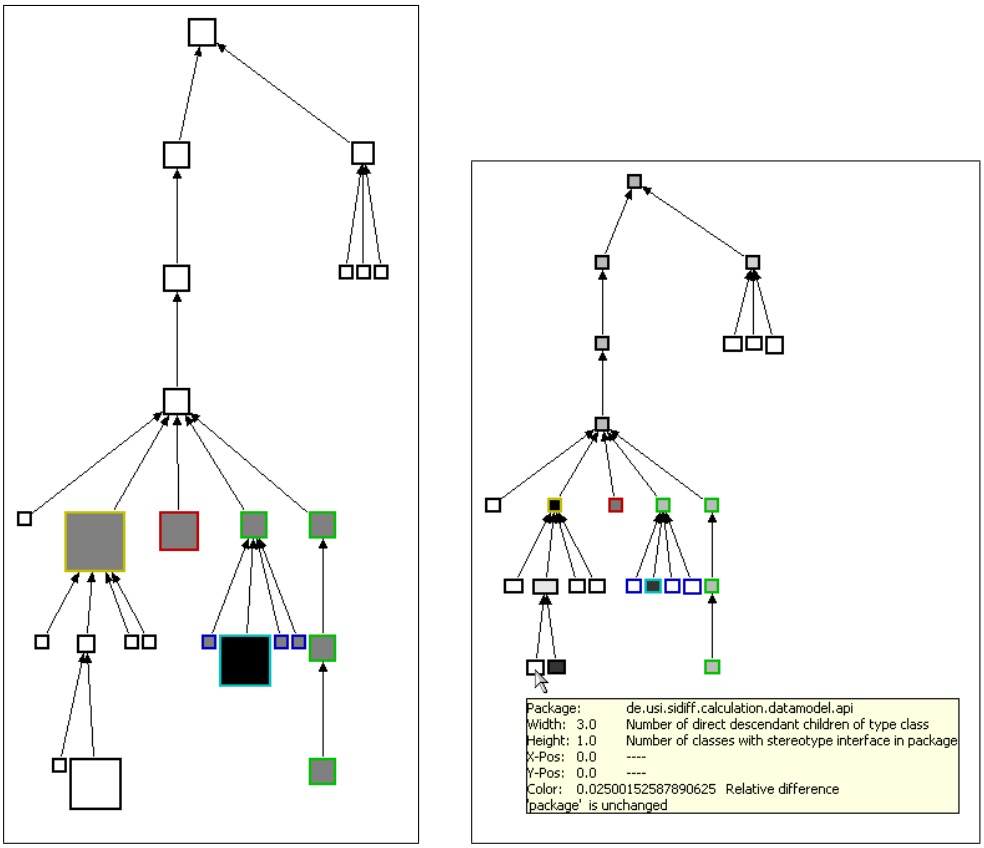
Größe und Farbe: Je größer und dunkler das Paket, desto stärker sind die Änderungen am Paket. Für die Beurteilung der Unähnlichkeit fließen jedoch nicht nur Änderungen am Paket selber, sondern auch Unterpakete, Assoziationen und enthaltene Klassen ein.

Position: Je höher ein geändertes Paket in der Hierarchie liegt, desto wichtiger ist diese Änderung für das Gesamtsystem.

Varianten

Var. 1: Durch die Benutzung von *Anzahl Unterschiede* für die Farbe eines Pakets, werden die direkten Änderungen, also die Änderungen, die nicht die Unterpakete und Klassen betreffen explizit durch die Schattierung hervorgehoben.

Var. 2: Wird für die Breite der Knoten *Anzahl Klassen im Paket* und für die Höhe *Anzahl Interfaces im Paket* benutzt, so kann die Wichtigkeit einer Änderung am Paket besser beurteilt werden, da der Benutzer die genaue Anzahl der für Klassendiagramme wichtigsten betroffenen Elemente erfährt.



(a) cd 200 package var1 (b) cd 200 package var2

Abbildung 4.6.: Nützliche Sichten, um die Veränderungen an der Paketstruktur offenzulegen. Im rechten Bild wird zusätzlich der Einfluss auf die Klassen des Systems verdeutlicht.

cd 220 package-DDC

Layout	Vertical Tree Layout
Knoten	Pakete
Kanten	Unterpaket zu Paket
Auswahl	Gesamtes System
Metriken	
Breite	Anz. direkter Kindelemente
Höhe	Anz. Unterschiede direkter Kindelemente
X-Position	—
Y-Position	—
Farbe	Durchschn. Unähnlichkeit direkter Kindelemente
Sortierung	Name
Beispiel	Abbildung 4.7

Beschreibung

Mit Hilfe dieser Sicht sieht der Anwender, in welchen Paketen sich deren Kindelemente geändert haben und wie stark diese Änderungen im Verhältnis zu allen Kindelementen durchschnittlich ausfallen. Die Anordnung der Pakete folgt der hierarchischen Struktur innerhalb des Modells, so dass der Überblick ständig gewährleistet ist.

Informationsgewinn

VK1: Ist die Höhe von 0 verschieden und der Knoten schattiert, so wurde mindestens ein Kindelement geändert.

VK2: Der Metrikwert für Höhe des Knoten zeigt die exakte Zahl der Veränderungen an den Kindelementen an.

VK3: Ein dunkler Farbton und auch ein großes Verhältnis zwischen Höhe und Breite der Knoten zeigen an, dass in dem Paket eine relativ starke Änderungen an den Kindelementen stattgefunden hat.

VK4: Stärke und Menge der Änderungen in einem Paket liefern einen ersten Anhaltspunkt für die Wichtigkeit der Änderungen. Darüber hinaus ist für die Beurteilung aber auch wichtig, welche konkreten Elementtypen von den Änderungen betroffen sind. Daher sollten im Anschluss an diese Sicht, die Sichten für die verschiedenen Elementtypen generiert werden, die in einem Paket vorkommen können. Diese Serie von Sichten wird nachfolgend in 4.4 vorgestellt.

Symptome

Breite: Je breiter der Knoten, desto mehr Kindelemente enthält das Paket. Dies gilt insbesondere für das „Rootpackage“, denn dort werden alle Assoziationen zwischen den Klassen angehängt.

Höhe: Je höher der Knoten, desto mehr Änderungen wurden an den vorhandenen Kindelementen vorgenommen.

Form: Je mehr sich die Form einem Quadrat annähert, desto umfangreicher die Änderungen an den Kindelementen.

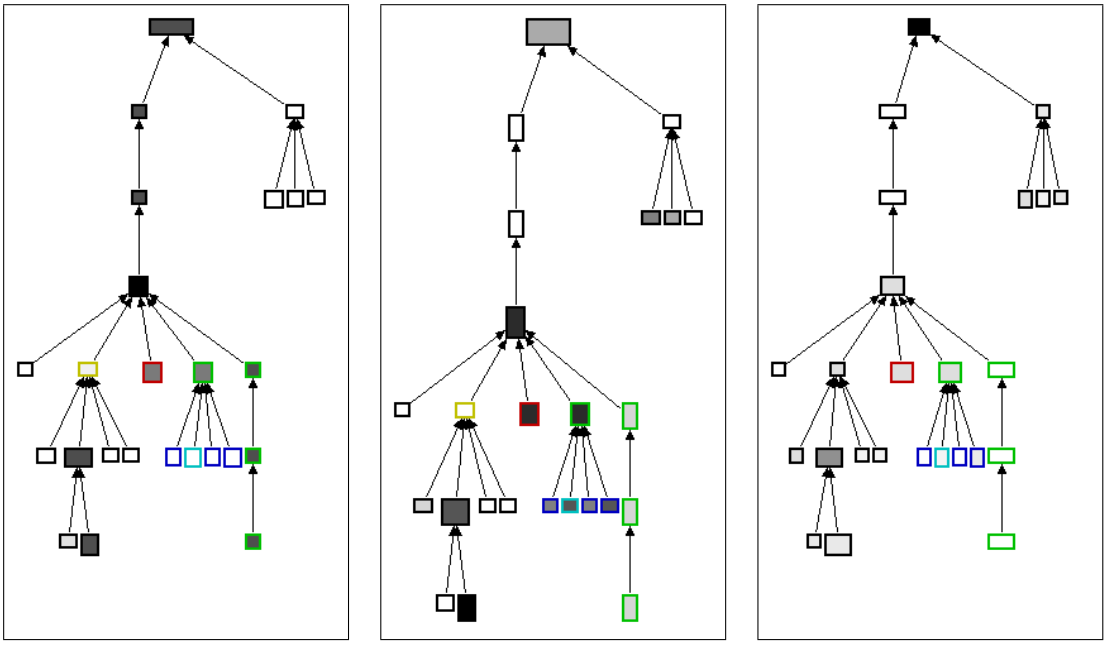
Position: Je höher ein Knoten mit großer Höhe in der Hierarchie sitzt, desto stärker können die Auswirkungen auf das System sein, da hierarchisch höher gelegene Elemente oftmals weitreichendere Assoziationen im System haben als tiefer gelegene.

Farbe: Je dunkler der Knoten, desto stärker fallen die Änderungen an den Kindelementen aus.

Varianten

Var. 1: Werden die Metriken für die Höhe und die Farbe der Knoten vertauscht, so werden alle Knoten mit Änderungen an den Kindelementen schattiert dargestellt. Dies vereinfacht die Auswahl der Elemente für die weitere Analyse der Differenz.

Var. 2: Werden die Metriken für die Breite und die Farbe der Knoten vertauscht, so bedeuten breite Knoten eine recht stärke Änderung der Kindelemente. Weiterhin gilt, dass je dunkler der Knoten ist, desto mehr Elemente sind betroffen. Dies sollte sich auch in der Höhe der Knoten wiederspiegeln, denn für eine starke Änderung von vielen Elementen sind auch mehr Änderungen nötig.



(a) cd 220 package-ddc (b) cd 220 package-ddc var1 (c) cd 220 package-ddc var2

Abbildung 4.7.: Die verschiedenen Varianten der Sichtdefinition für Pakete machen jeweils andere Aspekte der geänderten Kindelemente deutlich.

cd 220 package-nestingtype

Layout	Vertical Tree Layout
Knoten	Pakete
Kanten	Unterpaket zu Paket
Auswahl	Gesamtes System
Metriken	
Breite	Anz. direkter Kindelemente vom Typ Paket/Klasse/Assoziation
Höhe	Anz. Unterschiede direkter Kindelemente vom Typ Paket/Klasse/Assoziation
X-Position	—
Y-Position	—
Farbe	Durchschn. Unähnlichkeit direkter Kindelemente vom Typ Paket/Klasse/Assoziation
Sortierung	Name
Beispiel	Abbildung 4.8

Beschreibung

Diese Reihe von Sichten sollte erzeugt werden, wenn mittels der Sicht aus 4.4 festgestellt wurde, dass es Änderungen an den Kindelementen von Paketen gibt. Der Benutzer erhält dadurch Aufschluss darüber, welche Elementtypen an den jeweiligen Änderungen eines Pakets beteiligt sind und wie stark sich diese Änderungen auf das Paket auswirken. Die konkreten Kindelementtypen von Pakten bei Klassendiagrammen sind *Pakete*, also sogenannte Unterpakete, die für den Softwareentwickler wichtigsten Elemente, die *Klassen* und *Assoziationen*, die aber ausschließlich am „Rootpackage“ hängen. Von höchstem Interesse ist dabei vor allem die klassenbezogene Sicht, da diese bestens dazu geeignet ist, Kandidaten für die Untersuchung in Sichten mit Klassen als Elementtyp auszuwählen (Unterkapitel 4.5).

Informationsgewinn

VK1: Ist die Höhe von 0 verschieden und der Knoten schattiert, so wurde mindestens ein Kindelement dieses Typs geändert.

VK2: Der Metrikwert für Höhe des Knoten zeigt die exakte Zahl der Veränderungen an den Kindelementen dieses Typs an.

VK3: Ein dunkler Farbton und auch ein großes Verhältnis zwischen Höhe und Breite der Knoten zeigen an, dass in dem Paket eine relativ starke Änderungen an den Kindelementen dieses Typs stattgefunden hat.

VK4: Alle drei Sichten zusammengenommen ergeben die Sicht aus 4.4. Die paketbezogene Sicht hat dabei eher informellen Charakter, da die Änderungen sich knozeptionell bedingt auch in der Rahmenfarbe wiederspiegeln. Änderungen

an den Assoziationen sollten mit den Sichten in 4.10 oder 4.11 weiter analysiert werden, während die Änderungen an Klassen im nachfolgene Unterkapitel 4.5 behandelt werden.

Symptome

Breite: Je breiter der Knoten, desto mehr Kindelemente des entsprechenden Typs enthält das Paket. Alle Assoziationen hängen am „Rootpackage“, so dass nur dieses für die Betrachtung von Assoziationen von Interesse ist

Höhe: Je höher der Knoten, desto mehr Änderungen wurden an den vorhandenen Kindelementen dieses Typs vorgenommen.

Form: Je mehr sich die Form einem Quadrat annähert, desto umfangreicher die Änderungen an den Kindelementen dieses Typs.

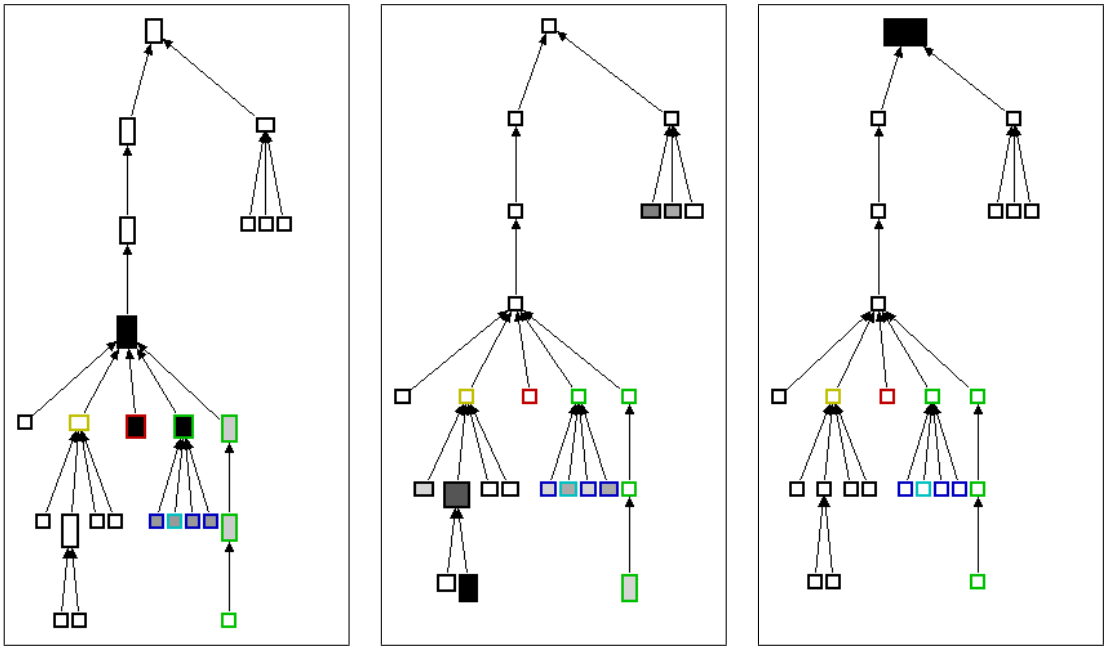
Position: Je höher ein Knoten mit großer Höhe in der Hierarchie sitzt, desto stärker können die Auswirkungen auf das System sein, da hierarchisch höher gelegene Elemente oftmals weitreichendere Assoziationen im System haben als tiefer gelegene. (Gilt nur für Pakete und Klassen)

Farbe: Je dunkler der Knoten, desto stärker fallen die Änderungen an den Kindelementen dieses Typs aus.

Varianten

Var. 1: Werden die Metriken für die Höhe und die Farbe der Knoten vertauscht, so werden alle Knoten mit Änderungen an den Kindelementen dieses Typs schattiert dargestellt. Dies vereinfacht die Auswahl der Elemente für die weitere Analyse der Differenzen des entsprechenden Types.

Var. 2: Werden die Metriken für die Breite und die Farbe der Knoten vertauscht, so bedeuten breite Knoten eine recht stärke Änderung der Kindelemente dieses Typs. Weiterhin gilt, dass je dunkler der Knoten ist, desto mehr Elemente sind betroffen. Dies sollte sich auch in der Höhe der Knoten wiederspiegeln, denn für eine starke Änderung von vielen Elementen sind auch mehr Änderungen nötig.



(a) cd 230 package-package var1 (b) cd 240 package-class var1 (c) cd 250 package-association var1

Abbildung 4.8.: Die Sichtenserie für die Kindelementtypen von Paketen. Sie entspricht einer Aufspaltung der Sicht 4.7(b) in die Elemente Paket, Klasse und Assoziationen auf und ermöglicht dadurch eine genauere Analyse über die Herkunft der Änderungen.

4.5. Sichten für Klassen

cd 400 class

Layout	Vertical Tree Layout
Knoten	Klassen
Kanten	Vererbung
Auswahl	Gesamtes System oder nur Pakete mit veränderten Klassen
Metriken	
Breite	Unähnlichkeit
Höhe	Unähnlichkeit
X-Position	—
Y-Position	—
Farbe	Unähnlichkeit
Sortierung	Name
Beispiel	Abbildung 4.9

Beschreibung

Diese polymetrische Sicht ist dazu geeignet, Änderungen an der Vererbungsstruktur eines Softwaresystems zu erkennen. Durch die hierarchische Darstellung der Klassen fallen eventuelle Unterschiede unmittelbar durch eine eindeutige Rahmenfarbe auf.

Informationsgewinn

VK1: Die Rahmenfarbe zeigt auf den ersten Blick, dass eine Veränderung an der Klasse stattgefunden hat.

VK2: Die Rahmenfarbe und der Tooltip geben Auskunft über die Anzahl der Veränderungen an der Klasse.

VK3: Die Größe und Farbe des Knotens zeigen die Stärke der Änderungen an der Klasse an.

VK4: Position und Name der Klasse, zusammen mit der Art der Änderung, sind nützlich für die Beurteilung der Wichtigkeit der Änderungen für diesen Bereich des Systems und weitere, per Assoziation verbundene Klassen.

Symptome

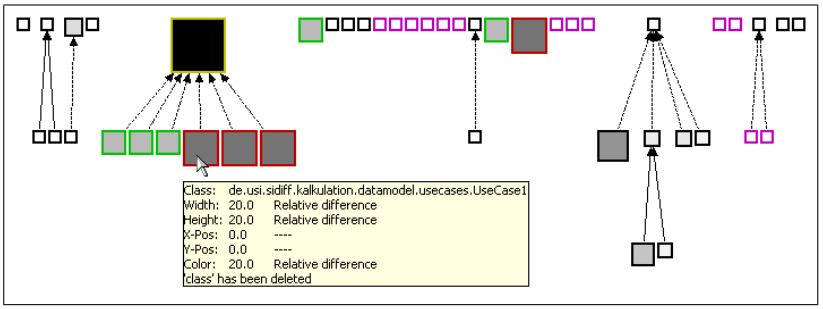
Größe und Farbe: Je größer und dunkler die Klasse, desto stärker sind die Änderungen an ihr. Für die Beurteilung der Unähnlichkeit fließen jedoch nicht nur Änderungen an der Klasse selber, sondern auch deren Operationen, Attribute und Generalisierungen.

Position: Je höher eine geänderte Klasse in der Hierarchie liegt, desto wichtiger ist diese Änderung für Unterklassen und assoziierte Klassen.

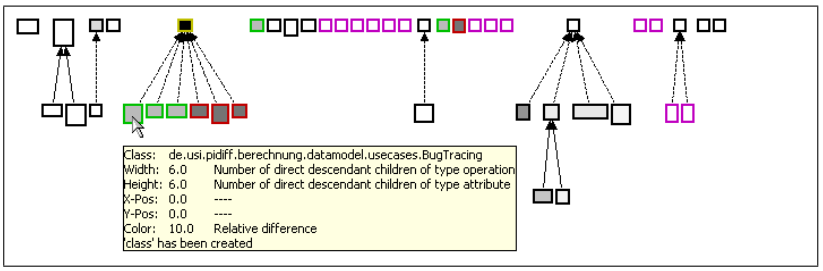
Varianten

Var. 1: Durch die Benutzung von *Anzahl Unterschiede* für die Farbe einer Klasse, werden die direkten Änderungen, also die Änderungen, die nicht durch Operationen, Attribute und Vererbung hervorgerufen werden, explizit durch die Schattierung hervorgehoben.

Var. 2: Wird für die Breite der Knoten *Anzahl Operationen in der Klasse* und für die Höhe *Anzahl Attribute in der Klasse* benutzt, so kann die Wichtigkeit einer Änderung an der Klasse besser beurteilt werden, da der Benutzer die genaue Anzahl der, für eine Klasse, wichtigsten betroffenen Elemente erhält.



(a) cd 400 class



(b) cd 400 class var2

Abbildung 4.9.: Zwei verschiedene Varianten einer Sicht, die Veränderungen an der Vererbungshierarchie eines Systems aufzeigt. Die linke Abbildung legt den Wert auf die Stärke der Veränderung einer Klasse und die rechte zeigt die Menge der betroffenen Operationen und Attribute an.

cd 420 class-DDC

Layout	Vertical Tree Layout
Knoten	Klassen
Kanten	Vererbung
Auswahl	Gesamtes System oder nur Pakete mit veränderten Klassen
Metriken	
Breite	Anz. direkter Kindelemente
Höhe	Anz. Unterschiede direkter Kindelemente
X-Position	—
Y-Position	—
Farbe	Durchschn. Unähnlichkeit direkter Kindelemente
Sortierung	Name
Beispiel	Abbildung 4.10

Beschreibung

Mit Hilfe dieser Sicht sieht der Anwender, in welchen Klassen sich deren Kindelemente, also Operationen, Attribute und die Vererbung geändert haben und wie stark diese Änderungen im Verhältnis zu allen Kindelementen durchschnittlich ausfallen. Die Anordnung der Klassen folgt der Vererbungsstruktur.

Informationsgewinn

VK1: Ist die Höhe von 0 verschieden und der Knoten schattiert, so wurde mindestens ein Kindelement der Klasse geändert.

VK2: Der Metrikwert für Höhe des Knoten zeigt die exakte Zahl der Veränderungen an den Kindelementen an.

VK3: Ein dunkler Farbton und auch ein großes Verhältnis zwischen Höhe und Breite der Knoten zeigen an, dass in dem Paket eine relativ starke Änderungen an den Kindelementen stattgefunden hat.

VK4: Stärke und Menge der Änderungen in einer Klasse liefern einen ersten Anhaltspunkt für die Wichtigkeit der Änderungen. Darüber hinaus ist für die Beurteilung ist aber auch wichtig, welche konkreten Elementtypen von den Änderungen betroffen sind. Daher sollten im Anschluss an diese Sicht, die Sichten für die verschiedenen Elementtypen generiert werden, die in einem Paket vorkommen können. Diese Serie von Sichten wird nachfolgend in 4.5 vorgestellt.

Symptome

Breite: Je breiter der Knoten, desto mehr Kindelemente enthält das Paket.

Höhe: Je höher der Knoten, desto mehr Änderungen wurden an den vorhandenen Kindelementen vorgenommen.

Form: Je mehr sich die Form einem Quadrat annähert, desto umfangreicher die Änderungen an den Kindelementen.

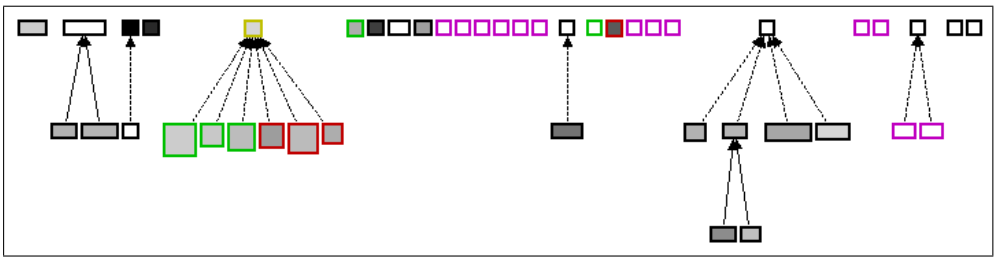
Position: Je höher ein Knoten mit großer Höhe in der Hierarchie sitzt, desto stärker sind die Auswirkungen auf Unterklassen und per Assoziationen verbundene Klassen.

Farbe: Je dunkler der Knoten, desto stärker fallen die Änderungen an den Kindelementen aus.

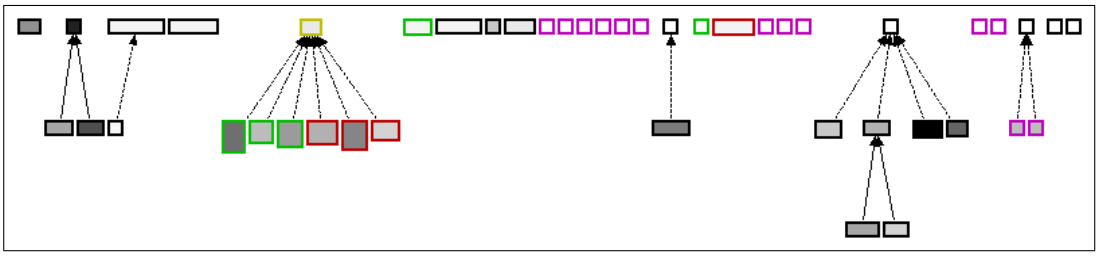
Varianten

Var. 1: Werden die Metriken für die Höhe und die Farbe der Knoten vertauscht, so werden alle Knoten mit Änderungen an den Kindelementen schattiert dargestellt. Dies vereinfacht die Auswahl der Elemente für die weitere Analyse der Differenz.

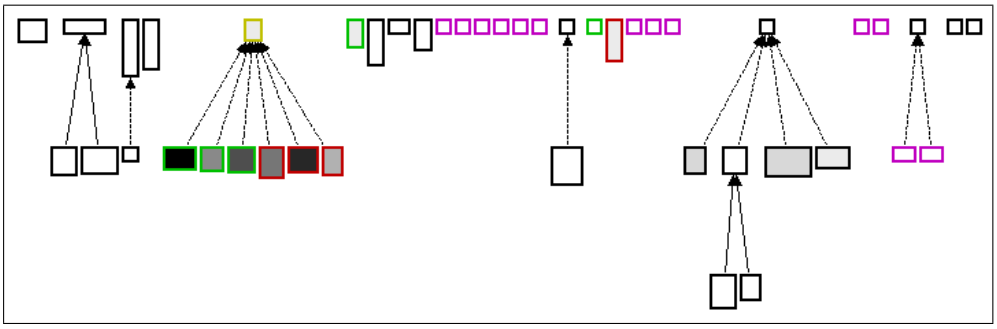
Var. 2: Werden die Metriken für die Breite und die Farbe der Knoten vertauscht, so bedeuten breite Knoten eine recht stärke Änderung der Kindelemente. Weiterhin gilt, dass je dunkler der Knoten ist, desto mehr Elemente sind betroffen. Dies sollte sich auch in der Höhe der Knoten wiederspiegeln, denn für eine starke Änderung von vielen Elementen sind auch mehr Änderungen nötig.



(a) cd 420 package-ddc



(b) cd 420 package-ddc var2



(c) cd 420 package-ddc var1

Abbildung 4.10.: Die verschiedenen Varianten der Sichtdefinition für Klassen machen jeweils andere Aspekte der geänderten Kindelemente deutlich.

cd 420 class-nestingtype

Layout	Vertical Tree Layout
Knoten	Klassen
Kanten	Vererbung
Auswahl	Gesamtes System oder nur Pakete mit veränderten Klassen
Metriken	
Breite	Anz. direkter Kindelem. vom Typ Operation/Attribut/Generalisierung
Höhe	Anz. Unterschiede direkter Kindelemente vom Typ Operation/Attribut/Generalisierung
X-Position	—
Y-Position	—
Farbe	Durchschn. Unähnlichkeit direkter Kindelemente vom Typ Operation/Attribut/Generalisierung
Sortierung	Name
Beispiel	Abbildung 4.11

Beschreibung

Diese Sichtenreihe sollte erzeugt werden, wenn mittels der Sicht aus 4.5 festgestellt wurde, dass es Änderungen an den Kindelementen von Klassen gibt. Der Benutzer erhält dadurch Aufschluss darüber, welche Elementtypen an den jeweiligen Änderungen einer Klasse beteiligt sind und wie stark sich diese Änderungen auf die Klasse auswirken. Die konkreten Kindelementtypen von Klassen bei Klassendiagrammen sind *Operationen*, oder auch Methoden genannt, *Attribute* und *Generalisierungen*, also Vererbungsbeziehungen, die an der erbenden Klasse hängen. Alle Sichten sind dazu geeignet, gezielt Kandidaten für weitere Untersuchungen der jeweiligen Elementtypen auszuwählen.

Informationsgewinn

VK1: Ist die Höhe von 0 verschieden und der Knoten schattiert, so wurde mindestens ein Kindelement dieses Typs geändert.

VK2: Der Metrikwert für Höhe des Knoten zeigt die exakte Zahl der Veränderungen an den Kindelementen dieses Typs an.

VK3: Ein dunkler Farbton und auch ein großes Verhältnis zwischen Höhe und Breite der Knoten zeigen an, dass in der Klasse eine relativ starke Änderungen an den Kindelementen dieses Typs stattgefunden hat.

VK4: Alle drei Sichten zusammengenommen ergeben die Sicht aus 4.5. Diese einzelnen Sichten erlauben also bereits genauere Aussagen über die Wichtigkeit der

Änderungen in Bezug auf die Klassen und Subklassen. Die Sichten für Operationen (Unterkapitel 4.6), Attribute (4.8) und Vererbungen (4.9) können noch zusätzlich zur Informationsgewinnung hinzugezogen werden.

Symptome

Breite: Je breiter der Knoten, desto mehr Kindelemente des entsprechenden Typs enthält die Klasse.

Höhe: Je höher der Knoten, desto mehr Änderungen wurden an den vorhandenen Kindelementen dieses Typs vorgenommen.

Form: Je mehr sich die Form einem Quadrat annähert, desto umfangreicher die Änderungen an den Kindelementen dieses Typs.

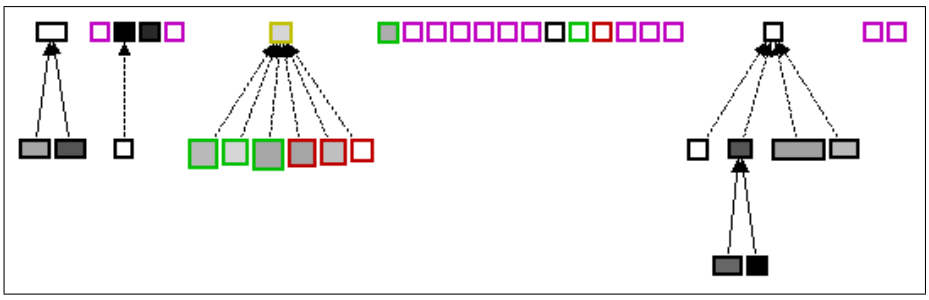
Position: Je höher ein Knoten mit großer Höhe in der Hierarchie sitzt, desto stärker sind die Auswirkungen auf tiefer in der Vererbungsstruktur gelegene Klassen.

Farbe: Je dunkler der Knoten, desto stärker fallen die Änderungen an den Kindelementen dieses Typs aus.

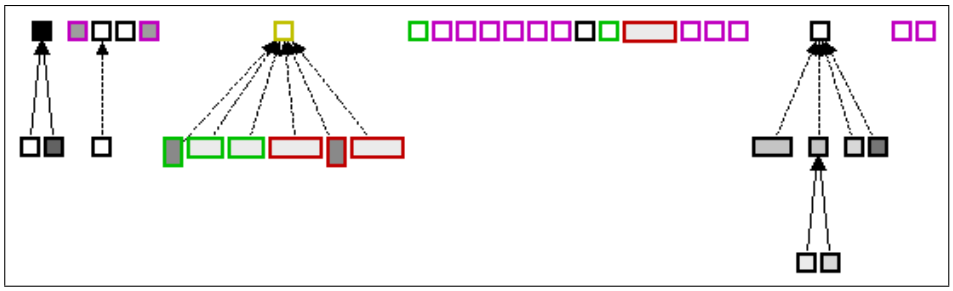
Varianten

Var. 1: Werden die Metriken für die Höhe und die Farbe der Knoten vertauscht, so werden alle Knoten mit Änderungen an den Kindelementen dieses Typs schattiert dargestellt. Dies vereinfacht die Auswahl der Elemente für die weitere Analyse der Differenzen des entsprechenden Types.

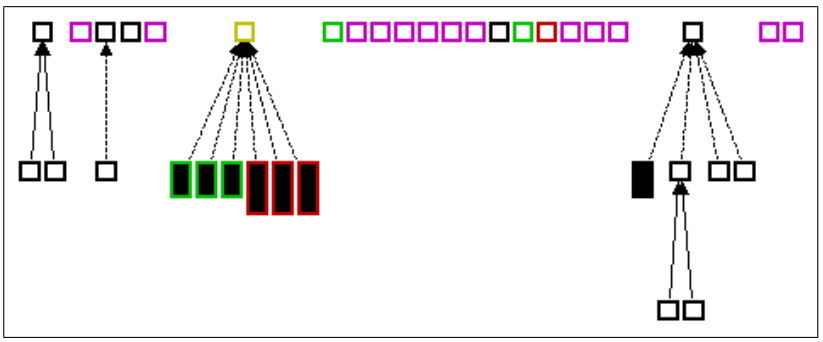
Var. 2: Werden die Metriken für die Breite und die Farbe der Knoten vertauscht, so bedeuten breite Knoten eine recht stärke Änderung der Kindelemente dieses Typs. Weiterhin gilt, dass je dunkler der Knoten ist, desto mehr Elemente sind betroffen. Dies sollte sich auch in der Höhe der Knoten wiederspiegeln, denn für eine starke Änderung von vielen Elementen sind auch mehr Änderungen nötig.



(a) cd 430 class-operation



(b) cd 440 class-operation var2



(c) cd 450 class-generalization var1

Abbildung 4.11.: Die Sichtenserie für die Kindelementtypen von Klassen, hier jeweils eine der Varianten für jeden Elementtyp.

4.6. Sichten für Methoden

cd 700 operation

Layout	Checker Board Layout
Knoten	Operationen
Kanten	—
Auswahl	Klassen mit geänderten Operationen aus 4.5
Metriken	
Breite	Anzahl Unterschiede
Höhe	Anzahl Unterschiede
X-Position	—
Y-Position	—
Farbe	Unähnlichkeit
Sortierung	Breite
Beispiel	Abbildung 4.12

Beschreibung

Diese Sicht zeigt dem Benutzer, welche Änderungen an den Operationen der Klassen eines Klassendiagramms vorgenommen wurden. Idealerweise wurden vorher die Klassen, bei denen sich Operationen geändert haben mittels der operationenbezogenen Sicht aus Unterkapitel 4.5 ausgewählt. Der Anwender ist durch diese Sicht in der Lage zu erkennen, welche Operationen sich in den Klassen geändert haben.

Informationsgewinn

VK1: Die Breite, Höhe und Rahmenfarbe eines Knotens zeigen direkt an, dass die Operation verändert wurde.

VK2: Die Metrikwerte für Breite oder Höhe geben Aufschluss über die Anzahl der Änderungen an der Operation. Die genaue Anzahl der geänderten Operationen einer Klasse ist aus Unterkapitel 4.5 bekannt.

VK3: Das Verhältnis aus unveränderten (schwarzen) und veränderten (bunten) Operationen bestimmt die Stärke der Änderungen an den Klassen.

VK4: Die Wichtigkeit der Änderungen an den Operationen lässt sich aus dem Namen der Operation ableiten. Möglicherweise muss aber zusätzlich die gesamte Klasse im Klassendiagramm betrachtet werden.

Symptome

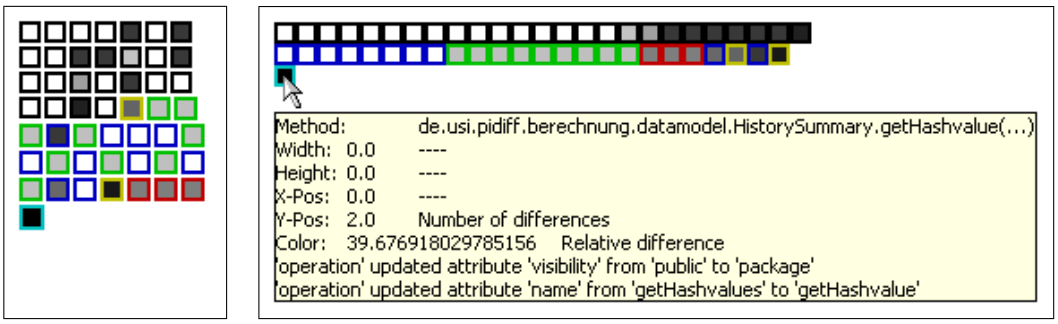
Position: Die geänderten Operationen befinden sich am rechten unteren Rand der polymetrischen Sicht.

Tooltip: Der Name der Operation wird für die Beurteilung der Wichtigkeit der Änderungen benötigt.

Farbe: Je dunkler der Knoten, desto stärker die Änderung an der Operation.

Varianten

Var. 1: Wenn die *Anzahl Unterschiede* statt für die Größe eines Knoten für die vertikale Position benutzt wird in Kombination mit dem *Vertical Histogram Layout*, dann wird die Beurteilung der Stärke der Änderungen vereinfacht. Wie in Abbildung 4.12(b) zu sehen ist, befinden sich unveränderte Operationen in der ersten Zeile und die veränderten, je nach Anzahl der Veränderungen, in den Zeilen darunter.



(a) cd 700 operation (b) cd 700 operation var1

Abbildung 4.12.: Zwei Sichtvarianten, um die Änderungen an den Operationen aufzuzeigen.

cd 710 operation-parameter

Layout	Checker Board Layout
Knoten	Operationen
Kanten	—
Auswahl	Gesamtes System
Metriken	
Breite	Anz. direkter Kindelemente vom Typ Parameter
Höhe	Anz. Unterschiede direkter Kindelemente vom Typ Parameter
X-Position	—
Y-Position	—
Farbe	Durchschn. Unähnlichkeit direkter Kindelemente vom Typ Parameter
Sortierung	Höhe
Beispiel	Abbildung 4.13

Beschreibung

Diese Sicht dient dazu, Operationen zu lokalisieren, deren Parameter sich verändert haben. Die Operationen, auf die eine Parameteränderung zutrifft, befinden sich recht unten in der polymetrischen Sicht und können einfach für weitere Untersuchungen an den Parametern in Unterkapitel 4.7 ausgewählt werden.

Informationsgewinn

VK1: Ist die Höhe von 0 verschieden, so wurde mindestens ein Parameter dieser Operation geändert.

VK2: Der Metrikwert für Höhe des Knoten zeigt die exakte Zahl der Veränderungen an den Parametern dieser Operation an.

VK3: Ein dunkler Farbton und auch ein großes Verhältnis zwischen Höhe und Breite der Knoten zeigen an, dass in der Operation eine relativ starke Änderungen an den Parametern dieser Operation stattgefunden hat.

VK4: Diese Sicht erlaubt, Kandidaten für eine weitere Untersuchung der Parameter auszuwählen. Mit Hilfe einer Sicht für Parameter kann anschließend herausgefunden werden, wie wichtig diese Änderungen für die Operation oder auch für die umgebende Klasse sind.

Symptome

Breite: Je breiter der Knoten, desto mehr Parameter hat die Operation.

Höhe: Je höher der Knoten, desto mehr Änderungen wurden an den Parametern dieser Operation vorgenommen.

Form: Je mehr sich die Form einem Quadrat annähert, desto umfangreicher die Änderungen an den Parametern dieser Operation.

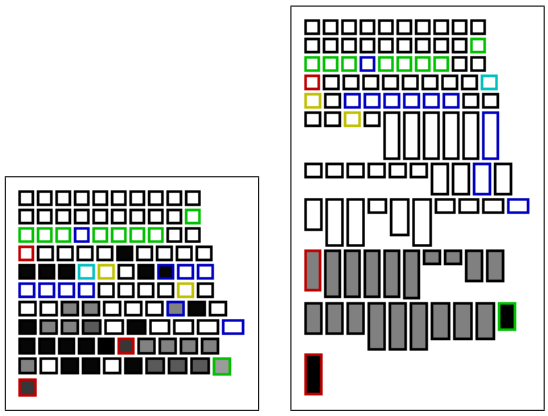
Position: Je weiter rechts unten sich der Knoten befindet, desto mehr Änderungen fanden an den Parametern der Operation statt.

Farbe: Je dunkler der Knoten, desto stärker fallen die Änderungen an den Parametern dieser Operation aus.

Varianten

Var. 1: Werden die Metriken für die Höhe und die Farbe der Knoten vertauscht, so werden alle Knoten mit Änderungen an den Parametern der Operation schattiert dargestellt. Dies vereinfacht die Auswahl der Operationen für die weitere Analyse der Unterschiede der Parameter.

Var. 2: Werden die Metriken für die Breite und die Farbe der Knoten vertauscht, so bedeuten breite Knoten eine recht stärke Änderung der Parametern dieser Operation. Weiterhin gilt, dass je dunkler ein breiter Knoten ist, desto mehr Parameter sind betroffen. Dies sollte sich auch in der Höhe der Knoten wiederspiegeln, denn für eine starke Änderung an vielen Parametern sind auch mehr Änderungen nötig.



(a) cd 710 operation-parameter (b) cd 710 operation-parameter var1

Abbildung 4.13.: Zwei Sichtvarianten, um Operationen für die Untersuchung der Änderungen an den Parametern auszuwählen.

4.7. Sichten für Parameter

cd 730 parameter

Layout	Checker Board Layout
Knoten	Parameter
Kanten	—
Auswahl	Operationen mit geänderten Parametern aus 4.6
Metriken	
Breite	Anzahl Unterschiede
Höhe	Anzahl Unterschiede
X-Position	—
Y-Position	—
Farbe	Unähnlichkeit
Sortierung	Breite
Beispiel	Abbildung 4.14

Beschreibung

Diese Sicht zeigt dem Benutzer, welche Änderungen an den Parametern der Operationen eines Klassendiagramms vorgenommen wurden. Idealerweise wurden vorher die Operationen, bei denen sich Parameter geändert haben mittels der parameterbezogenen Sicht aus Unterkapitel 4.6 ausgewählt. Der Anwender ist durch diese Sicht in der Lage zu erkennen, welche Parameter sich in den Operationen geändert haben.

Informationsgewinn

VK1: Die Breite, Höhe und Rahmenfarbe eines Knotens zeigen direkt an, dass der Parameter verändert wurde.

VK2: Die Metrikwerte für Breite oder Höhe geben Aufschluss über die Anzahl der Änderungen an der Operation. Die Anzahl der geänderten Parameter einer Operation ist aus Unterkapitel 4.6 bekannt.

VK3: Das Verhältnis aus unveränderten (schwarzen) und veränderten (bunten) Parameter bestimmt die Stärke der Änderungen an den Operationen.

VK4: Die Wichtigkeit der Änderungen an den Parametern lässt sich aus dem Namen des Parameters ableiten. Zusätzlich sollte die Operation, oder sogar die gesamte Klasse im geänderten Klassendiagramm betrachtet werden.

Symptome

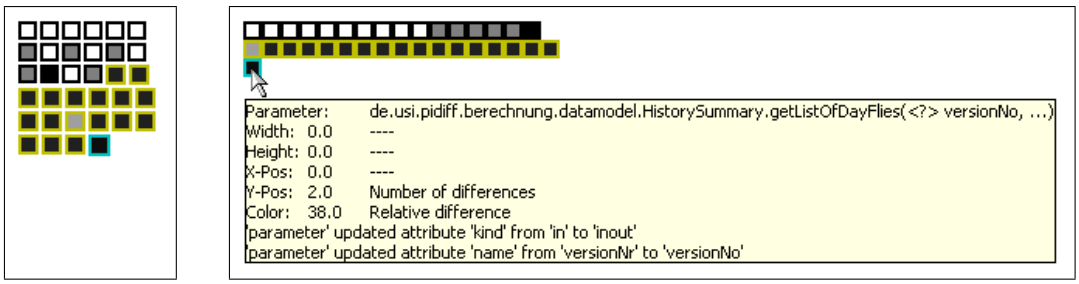
Position: Die geänderten Parameter befinden sich am rechten unteren Rand der polymetrischen Sicht.

Tooltip: Der Name des Parameters wird für die Beurteilung der Wichtigkeit der Änderungen benötigt.

Farbe: Je dunkler der Knoten, desto stärker die Änderung an dem Parameter.

Varianten

Var. 1: Wenn die *Anzahl Unterschiede* statt für die Größe eines Knoten für die vertikale Position benutzt wird in Kombination mit dem *Vertical Histogram Layout*, dann wird die Beurteilung der Stärke der Änderungen vereinfacht. Wie in Abbildung 4.14(b) zu sehen ist, befinden sich unveränderte Parameter in der ersten Zeile und die veränderten, je nach Anzahl der Veränderungen, in den Zeilen darunter.



(a) cd 730 parameter

(b) cd 730 parameter var1

Abbildung 4.14.: Zwei Sichtvarianten, um die Änderungen an den Parametern aufzuzeigen.

4.8. Sichten für Attribute

cd 760 attribute

Layout	Checker Board Layout
Knoten	Parameter
Kanten	—
Auswahl	Klassen mit geänderten Attributen aus 4.5
Metriken	
Breite	Anzahl Unterschiede
Höhe	Anzahl Unterschiede
X-Position	—
Y-Position	—
Farbe	Unähnlichkeit
Sortierung	Breite
Beispiel	Abbildung 4.15

Beschreibung

Diese Sicht zeigt dem Benutzer, welche Änderungen an den Attributen der Klassen eines Klassendiagramms vorgenommen wurden. Idealerweise wurden vorher die Klassen, bei denen sich Attribute geändert haben, mittels der attributbezogenen Sicht aus Unterkapitel 4.5 ausgewählt. Der Anwender ist durch diese Sicht in der Lage zu erkennen, welche Attribute sich in den Klassen geändert haben.

Informationsgewinn

VK1: Die Breite, Höhe und Rahmenfarbe eines Knotens zeigen direkt an, dass das Attribut verändert wurde.

VK2: Die Metrikwerte für Breite oder Höhe geben Aufschluss über die Anzahl der Änderungen an dem Attribut. Die Anzahl der geänderten Attribut einer Operation ist aus Unterkapitel 4.5 bekannt.

VK3: Das Verhältnis aus unveränderten (schwarzen) und veränderten (bunten) Attributen bestimmt die Stärke der Änderungen an den Klassen.

VK4: Die Wichtigkeit der Änderungen an den Attributen lässt sich aus dem Namen des Attributs ableiten. Zusätzlich sollte die Klasse im geänderten Klassendiagramm betrachtet werden.

Symptome

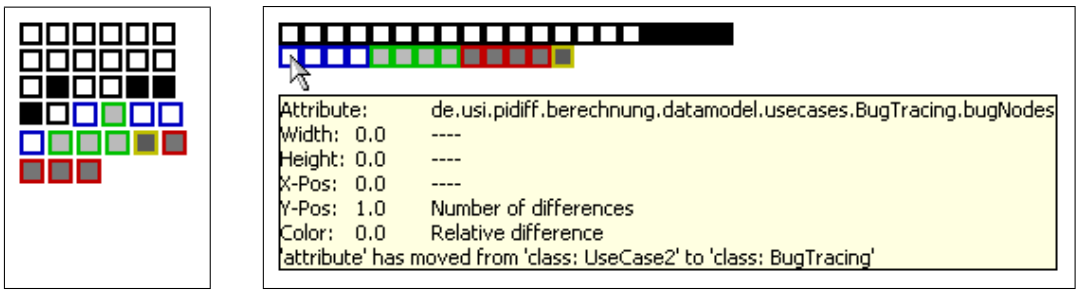
Position: Die geänderten Attribute befinden sich am rechten unteren Rand der polymetrischen Sicht.

Tooltip: Der Name des Attributs wird für die Beurteilung der Wichtigkeit der Änderungen benötigt.

Farbe: Je dunkler der Knoten, desto stärker die Änderung an dem Attribut.

Varianten

Var. 1: Wenn die *Anzahl Unterschiede* statt für die Größe eines Knoten für die vertikale Position benutzt wird in Kombination mit dem *Vertical Histogram Layout*, dann wird die Beurteilung der Stärke der Änderungen vereinfacht. Wie in Abbildung 4.15(b) zu sehen ist, befinden sich unveränderte Attribute in der ersten Zeile und die veränderten, je nach Anzahl der Veränderungen, in den Zeilen darunter.



(a) cd 760 attribute (b) cd 760 attribute var1

Abbildung 4.15.: Zwei Sichtvarianten, um die Änderungen an den Attributen aufzuzeigen.

4.9. Sichten für Generalisierungen

cd 790 generalization

Layout	Checker Board Layout
Knoten	Parameter
Kanten	—
Auswahl	Klassen mit geänderter Generalisierung aus 4.5
Metriken	
Breite	Anzahl Unterschiede
Höhe	Anzahl Unterschiede
X-Position	—
Y-Position	—
Farbe	Unähnlichkeit
Sortierung	Breite
Beispiel	Abbildung 4.16

Beschreibung

Diese Sicht zeigt dem Benutzer, welche Änderungen an der Vererbungsstruktur der Klassen eines Klassendiagramms vorgenommen wurden. Idealerweise wurden vorher die Klassen, bei denen sich die Generalisierung geändert hat, mittels der generalisierungsbezogenen Sicht aus Unterkapitel 4.5 ausgewählt. Der Anwender ist durch diese Sicht in der Lage zu erkennen, wie sich die Generalisierung einzelner Klassen verändert hat.

Informationsgewinn

VK1: Die Breite, Höhe und Rahmenfarbe eines Knotens zeigen direkt an, dass die Generalisierung verändert wurde.

VK2: Die Metrikwerte für Breite oder Höhe geben Aufschluss über die Anzahl der Änderungen an der Generalisierung. Dies sollten in der Standardkonfiguration lediglich neue oder gelöschte Generalisierungen sein.

VK3: Das Verhältnis aus unveränderten (schwarzen) und veränderten (bunten) Generalisierungen bestimmt die Stärke der Änderungen an den Klassen.

VK4: Die Wichtigkeit der Änderungen an den Generalisierungen lässt sich aus dem Namen der Generalisierung ableiten. Zusätzlich sollte die Klasse im geänderten Klassendiagramm betrachtet werden.

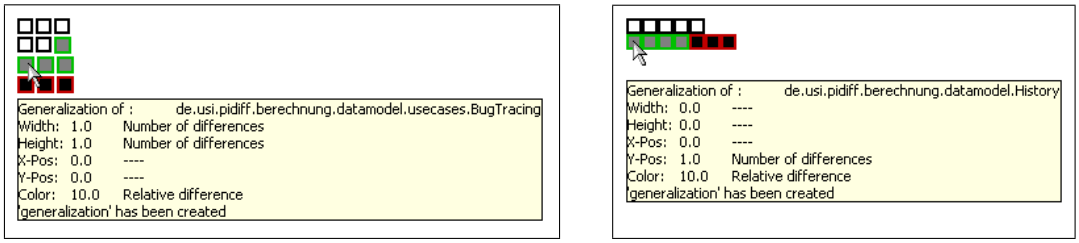
Symptome

Position: Die geänderten Generalisierungen befinden sich am rechten unteren Rand der polymetrischen Sicht.

Tooltip: Der Name der Generalisierung wird für die Beurteilung der Wichtigkeit der Änderung benötigt.

Varianten

Var. 1: Wenn die *Anzahl Unterschiede* statt für die Größe eines Knoten für die vertikale Position benutzt wird in Kombination mit dem *Vertical Histogram Layout*, dann wird die Beurteilung der Stärke der Änderungen vereinfacht. Wie in Abbildung 4.16(b) zu sehen ist, befinden sich unveränderte Generalisierungen in der ersten Zeile und die neuen beziehungsweise gelöschten in der Zeile darunter.



(a) cd 790 generalization (b) cd 790 generalization var1

Abbildung 4.16.: Zwei Sichtvarianten, um die Änderungen an den Generalisierungen aufzuzeigen.

4.10. Sichten für Assoziationen

cd 900 assoc

Layout	Checker Board Layout
Knoten	Assoziationen
Kanten	—
Auswahl	Gesamtes System
Metriken	
Breite	Anzahl Unterschiede
Höhe	Anzahl Unterschiede
X-Position	—
Y-Position	—
Farbe	Unähnlichkeit
Sortierung	Breite
Beispiel	Abbildung 4.17

Beschreibung

Diese polymetrische Sicht ist dazu geeignet, Änderungen an den Assoziationen eines Softwaresystems zu erkennen. Sowohl durch die Position, als auch durch die Rahmenfarbe fallen veränderte Assoziationen unmittelbar auf.

Informationsgewinn

VK1: Die Rahmenfarbe zeigt auf den ersten Blick, dass eine Veränderung an der Assoziation stattgefunden hat.

VK2: Die Rahmenfarbe und der Tooltip geben Auskunft über die Anzahl der Veränderungen an der Assoziation.

VK3: Die Größe und Farbe des Knotens zeigen die Stärke der Änderungen an der Assoziation an.

VK4: Der Name der Assoziation, zusammen mit der Art der Änderung, sind nützlich für die Beurteilung der Wichtigkeit der Änderungen dieser Assoziation und die über diese Assoziation verbundenen Klassen.

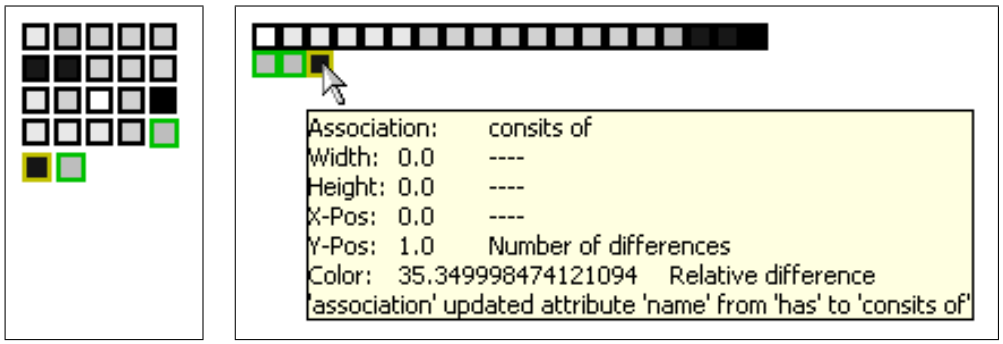
Symptome

Position: Die geänderten Assoziationen befinden sich am rechten unteren Rand der polymetrischen Sicht.

Tooltip: Der Name der Assoziation wird für die Beurteilung der Wichtigkeit der Änderung benötigt.

Varianten

Var. 1: Wenn die *Anzahl Unterschiede* statt für die Größe eines Knoten für die vertikale Position benutzt wird in Kombination mit dem *Vertical Histogram Layout*, dann wird die Beurteilung der Stärke der Änderungen vereinfacht. Wie in Abbildung 4.17(b) zu sehen ist, befinden sich unveränderte Assoziationen in der ersten Zeile und die veränderten, je nach Anzahl der Veränderungen, in den Zeilen darunter.



(a) cd 900 assoc (b) cd 900 assoc var1

Abbildung 4.17.: Zwei Sichtvarianten, um die Änderungen an den Assoziationen aufzuzeigen.

cd 910 assoc-assocend

Layout	Checker Board Layout
Knoten	Assoziationen
Kanten	—
Auswahl	Gesamtes System
Metriken	
Breite	Anz. direkter Kindelemente vom Typ Assoziationsende
Höhe	Anz. Unterschiede direkter Kindelemente vom Typ Assoziationsende
X-Position	—
Y-Position	—
Farbe	Durchschn. Unähnlichkeit direkter Kindelemente vom Typ Assoziationsende
Sortierung	Höhe
Beispiel	Abbildung 4.18

Beschreibung

Diese Sicht dient dazu, Assoziationen zu lokalisieren, deren Assoziationsenden sich verändert haben. Die Assoziationen, auf die eine Änderung an den Assoziationsenden zutrifft, befinden sich rechts unten in der polymetrischen Sicht und können einfach für weitere Untersuchungen an den Parametern in Unterkapitel 4.11 ausgewählt werden.

Informationsgewinn

VK1: Ist die Höhe von 0 verschieden, so wurde mindestens ein Assoziationsende dieser Assoziation geändert.

VK2: Der Metrikwert für Höhe des Knoten zeigt die exakte Zahl der Veränderungen an den Assoziationsenden dieser Assoziation an.

VK3: Ein dunkler Farbton und auch ein großes Verhältnis zwischen Höhe und Breite der Knoten zeigen an, dass in der Assoziation eine relativ starke Änderungen an den Assoziationsenden stattgefunden hat.

VK4: Diese Sicht erlaubt, Kandidaten für eine weitere Untersuchung der Assoziationsenden auszuwählen. Mit Hilfe einer Sicht für Assoziationsenden kann anschließend herausgefunden werden, wie wichtig diese Änderungen für die Assoziation oder auch für die assoziierten Klassen sind.

Symptome

Breite: Je breiter der Knoten, desto mehr Assoziationsenden hat die Assoziation.

Höhe: Je höher der Knoten, desto mehr Änderungen wurden an den Assoziationsenden dieser Assoziation vorgenommen.

Form: Je mehr sich die Form einem Quadrat annähert, desto umfangreicher die Änderungen an den Assoziationsenden dieser Assoziation.

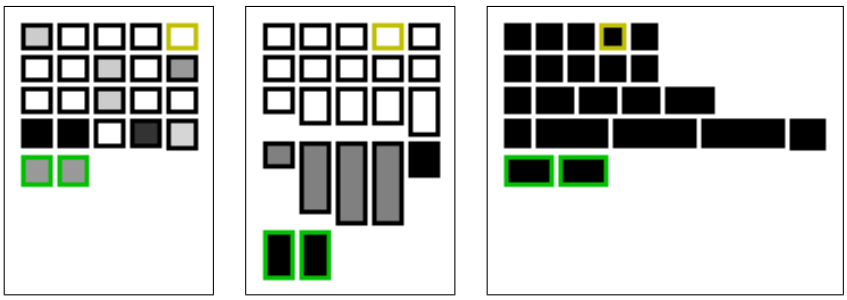
Position: Je weiter rechts unten sich der Knoten befindet, desto mehr Änderungen fanden an den Assoziationsenden der Assoziation statt.

Farbe: Je dunkler der Knoten, desto stärker fallen die Änderungen an den Assoziationsenden dieser Assoziation aus.

Varianten

Var. 1: Werden die Metriken für die Höhe und die Farbe der Knoten vertauscht, so werden alle Knoten mit Änderungen an den Parametern der Operation schattiert dargestellt. Dies vereinfacht die Auswahl der Assoziationen für die weitere Analyse der Unterschiede der Assoziationsenden.

Var. 2: Werden die Metriken für die Breite und die Farbe der Knoten vertauscht, so bedeuten breite Knoten eine recht stärke Änderung der Assoziationsenden dieser Assoziation. Weiterhin gilt, dass je dunkler ein breiter Knoten ist, desto mehr Assoziationsenden sind betroffen. Dies sollte sich auch in der Höhe der Knoten wiederspiegeln, denn für eine starke Änderung an vielen Assoziationsenden sind auch mehr Änderungen nötig.



(a) cd 910 assoc-assocend (b) cd 910 assoc-assocend var1 (c) cd 910 assoc-assocend var2

Abbildung 4.18.: Zwei Sichtvarianten, um Assoziationen für die Untersuchung der Änderungen an den Assoziationsenden auszuwählen.

4.11. Sichten für Assoziationsenden

cd 950 assoc end

Layout	Horizantal Tree Layout
Knoten	Assoziationsenden
Kanten	Assoziationen zwischen Assoziationsenden
Auswahl	Assoziationen mit geänderten Assoziationsenden aus 4.10
Metriken	
Breite	Anzahl Unterschiede
Höhe	Unähnlichkeit
X-Position	—
Y-Position	—
Farbe	Unähnlichkeit
Sortierung	Farbe
Beispiel	Abbildung 4.19

Beschreibung

Diese Sicht zeigt dem Benutzer, welche Änderungen an den Assoziationsenden der Assoziationen eines Klassendiagramms vorgenommen wurden. Idealerweise wurden vorher die Assoziationen, bei denen sich Assoziationsenden geändert haben, mittels der assoziationsendenbezogenen Sicht aus Unterkapitel 4.10 ausgewählt. Der Anwender ist durch diese Sicht in der Lage zu erkennen, welche Assoziationsenden sich in den Assoziationen geändert haben.

Informationsgewinn

VK1: Die Rahmenfarbe zeigt direkt an, ob es Änderungen an dem Assoziationsende gibt.

VK2: Der Metrikwert für die Breite zeigt an, wie viele Änderungen es an dem Assoziationsende gibt.

VK3: Höhe und Farbe des Knotens zeigen an, wie stark sich das Assoziationsende verändert hat.

VK4: Die Wichtigkeit einer Assoziation hängt von verschiedenen Faktoren ab, wie zum Beispiel den Klassen, die mit der Assoziation verbunden sind.

Symptome

Breite: Je breiter der Knoten, desto mehr Änderungen an diesem Assoziationsende

Höhe & Farbe Je höher beziehungsweise dunkler, desto stärker die Änderung an diesem Assoziationsende

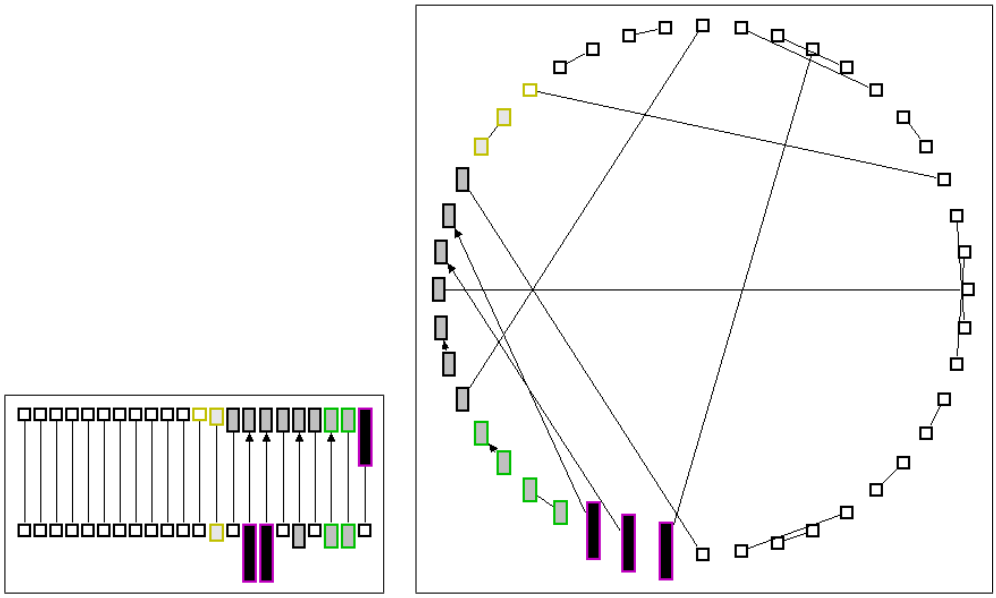
Tooltip: Der Name und die Beschreibung der Änderung sind hilfreich für die Beurteilung der Wichtigkeit der Änderungen.

Varianten

Var. 1: Alternativ kann auch das *Circle Layout* benutzt werden. Die Übersicht ist dann noch weitestgehend erhalten und man es den Vorteil, dass alle veränderten Knoten hintereinander gezeichnet werden.

Anmerkung

Die Auswahl der Assoziationsenden kann leider nicht über die zugehörigen Klassen erfolgen, da alle Assoziationen mit den zugehörigen Enden am „Rootpackage“ hängen.



(a) cd 950 assoc end (b) cd 950 assoc end var1

Abbildung 4.19.: Zwei Sichtvarianten, um die Änderungen an den Assoziationsenden zu erkennen.

Kapitel 5.

Anpassungen an andere Modelltypen

In diesem Kapitel soll beschrieben werden, welche Schritte notwendig sind, um mit den benutzten Werkzeugen die qualitative Analyse von Modelldifferenzen für einen weiteren Dokumenttyp zu ermöglichen. Die dazu erforderlichen Anpassungen werden in einer möglichst allgemeinen Form beschrieben. Weiterhin wird für jeden Schritt jeweils ein Beispiel angegeben, um ihn zusätzlich zu verdeutlichen.

Durch die Kombination der drei benutzten Werkzeuge – SiDiff, DiMPl und PV4E – teilen sich die Erweiterungen in drei Bereiche auf, die in den nachfolgenden Unterkapiteln beschrieben werden.

5.1. Anpassungen an SiDiff

Zunächst muss der SiDiff-Algorithmus entsprechend vorbereitet werden, mit dem neuen Dokumenttyp umzugehen.

5.1.1. Transformation des Dateiformats

Um zwei strukturierte Modelle miteinander vergleichen zu können, muss der Inhalt der beiden Dokumente durch einen Parser in die SiDiff-interne Datenstruktur transformiert werden. Dabei ist zwischen zwei verschiedenen Dateiformaten zu unterscheiden.

XML-basierte Dateiformate

Für XML-basierte Dateiformate existiert im SiDiff.core-Modul eine Klasse namens `GraphLoader`, die über die Methode `transformAndLoad()` dazu veranlasst wird, die Datei zu öffnen und mittels einer XSLT-Datei in die interne Graphenstruktur zu transformieren. Anschließend wird die aufbereitete XML-Datei in die Graphenstruktur geladen.

Beispiel: In SiDiff existieren die Transformationsdateien `fujabaXMI.xslt` und `ArgoUML.xslt` für UML-Klassendiagramme aus Fujaba und ArgoUML.

Binäre Dateiformate

Bei Dateiformaten, die in binärer Form vorliegen und für die es keine Möglichkeit für einen Export in XML existiert kann die `GraphLoader`-Klasse nicht genutzt werden. Hier wird ein neues Parser-Frontend benötigt, welches das spezielle Dateiformat in die Graphenstruktur transformiert.

Beispiel: Für Simulink-Diagramme heißt die entsprechende `Loader`-Klasse `JMIMatlabModelLoader`, mit der die MDL-Dateien geöffnet und transformiert werden.

5.1.2. Konfigurationsdatei für den Vergleich

Der Vergleichsalgorithmus von SiDiff benötigt eine XML-Datei um den Elementvergleich zu konfigurieren. Der Aufbau dieser Datei ist in der DTD `SiDiffConfig.dtd` des SiDiff-Projekts beschrieben. Für jeden Elementtyp wird dort anhand von sogenannten *CompareItems* festgelegt, welche Attribute, Subelemente und Beziehungen für eine Korrespondenz des Knoten relevant sind und wie stark diese Gewichtet werden.

Beispiel: Ausschnitt aus der Datei `TUDJmiSimulinkBalancedDiffConfig.xml` in der das Elemente vom Typ *DataStoreMemoryBlock* für Simulink-Diagramme konfiguriert wird.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE SiDiff SYSTEM "http://pi.informatik.uni-siegen.de/SiDiff
       /SiDiffConfig.dtd">
3  <SiDiff>
4    <Configurations>
5      [...]
6      <NodeType name="DataStoreMemoryBlock">
7        <Configuration qualifyingAttribute="Name" threshold="0.45"
             ordered="false" moveAllowed="false" parentForceMatch="
             false"/>
8        <CompareItems>
9          <If weight="1.0" condition="IsParentMatched" parameter="
               true">
10           <Then>
11             <CompareItem class="CNIAttributeLCS" weight="0.2"
                   parameter="Name"/>
12             <CompareItem class="CNIAttributeLCS" weight="0.4"
                   parameter="DataStoreName"/>
13             <CompareItem class="CNIAttributeEQUALS" weight="0.2"
                   parameter="InitialValue"/>
14             <CompareItem class="CNIAttributeEQUALS" weight="0.2"
                   parameter="RTWStateStorageClass"/>
```

```
15            </Then>
16            <Else>
17              <CompareItem class="CNIAttributeEQUALS" weight="0.4" ↩
                    parameter="Name"/>
18              <CompareItem class="CNIAttributeEQUALS" weight="0.4" ↩
                    parameter="DataStoreName"/>
19              <CompareItem class="CNIAttributeEQUALS" weight="0.1" ↩
                    parameter="InitialValue"/>
20              <CompareItem class="CNIAttributeEQUALS" weight="0.1" ↩
                    parameter="RTWStateStorageClass"/>
21            </Else>
22          </If>
23        </CompareItems>
24      </NodeType>
25      [...]
26    </Configurations>
27    [...]
28  </SiDiff>
```

5.1.3. Konfiguration der Modellmetriken

Sollen Modellmetriken berechnet werden, welche über die generisch erzeugten Modellmetriken der `DifferenceMetricsCalculationEngine` hinausgehen, so können diese in der Datei `SiDiffMetricsInstance.xml` im SiDiff-Projekt konfiguriert werden. Sie werden vom `QueryConfigurationLoader` in vollständige `Query`-Objekte umgesetzt und an der `QueryRegistry` angemeldet. Über das Attribut *operation* wird die konkrete `Annotator`-Klasse angegeben, die mit den in *parameter* angegebenen Übergabewerten initialisiert wird und die eigentliche Berechnung durchführt.

Beispiel: Die beiden hier gezeigten Einträge konfigurieren zwei Metriken für den Elementtyp *Paket* eines Klassendiagramms. Die konkrete Berechnungsfunktion für die erste der beiden angegebenen Metriken ist in Abschnitt 3.2.3 angegeben.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <QueryDefinitions>
3    [...]
4    <Query
5      name="NOclass"
6      description="Number of classes in package"
7      nodeType="package"
8      order="preorder"
9      operation="CountChildNodesTypes"
10     parameter="class"/>
11   [...]
12   <Query
13     name="NOclassvisibilitypublic"
14     description="Number of classes with visibility public in ↩
           package"
```

```
15      nodeType="package"
16      order="preorder"
17      operation="CountChildNodesByValue"
18      parameter="class,visibility,public"/>
19    [...]
20  </QueryDefinitions>
```

5.2. Anpassungen an DiMPl

Nach der Erweiterung des SiDiff-Algorithmus auf den neuen Dokumenttyp muss die Konfiguration von DiMPl entsprechend erweitert werden und eventuell spezielle Komponenten implementiert werden.

5.2.1. Erweiterung der plugin.xml

Die Dokumentstruktur

Damit sowohl die Differenz- als auch die Ähnlichkeitsmetriken generisch erzeugt werden können, muss die Dokumentstruktur des neuen Dokumenttyps in der `plugin.xml` von DiMPl eingetragen werden. Die Dokumentstruktur leitet sich aus der internen Objektstruktur ab, welche die Elementtypen bei der Transformation in die SiDiff-Datenstruktur bilden.

Beispiel: Ausschnitt aus der Konfiguration der `plugin.xml` für Simulink-Diagramme.

```
1  [...]
2  <diagram type="SimuLinkDiagram">
3    [...]
4    <nodetype name="Block">
5      <attribute name="Name" isunordered="true"/>
6      <nestednodetype name="Port" edgetype="BlockContainsPort"/>
7      <nestednodetype name="Inport" edgetype="BlockContainsInport"/>
8    </nodetype>
9    <nodetype name="Subsystem">
10     <attribute name="Name" isunordered="true"/>
11   </nodetype>
12   <nodetype name="SubsystemBlock">
13     <attribute name="Name" isunordered="true"/>
14     <nestednodetype name="Subsystem" edgetype="
           SubsystemBlockContainsSubsystem"/>
15   </nodetype>
16   <nodetype name="System">
17     <attribute name="Name" isunordered="true"/>
18     <nestednodetype name="Block" edgetype="SystemContainsBlock"/>
19     <nestednodetype name="Line" edgetype="SystemContainsLine"/>
```

```
20      <nestednodetype name="Annotation" edgetype="
            SystemContainsAnnotation"/>
21    </nodetype>
22    [...]
23  </diagram>
```

Kontextmenü des Navigator

Falls die Dokumente des neuen Dokumenttyps eine neue Dateiendung mit sich bringen, muss das Popup-Menü des *Navigators* in PV4E entsprechend erweitert werden. Dies kann entweder über die Formularoberfläche der `plugin.xml` erfolgen, wie in Abbildung 3.4 bereits zu sehen, oder auch per Hand direkt in der Datei editiert werden.

Beispiel: Ausschnitt aus der `plugin.xml` von DiMPl für die MDL-Dokumente von Matlab/Simulink-Diagrammen.

```
1  <extension point="org.eclipse.ui.popupMenus">
2    [...]
3    <objectContribution
4        id="de.usi.rcp.polymetricviews.adapter.sidiff.resource.mdl"
5        nameFilter="*.mdl"
6        objectClass="org.eclipse.core.resources.IFile">
7      <action
8          class="de.usi.rcp.polymetricviews.adapter.sidiff.popup.
               actions.UseAsSiDiffDocument"
9          definitionId="2ndDocument"
10         enablesFor="1"
11         id="de.usi.rcp.polymetricviews.adapter.sidiff.popup.
               actions.UseAsSecondSiDiffDocument"
12         label="Use as second document in DiMPl"
13         menubarPath="PV4E"/>
14     <action
15         class="de.usi.rcp.polymetricviews.adapter.sidiff.popup.
               actions.UseAsSiDiffDocument"
16         definitionId="1stDocument"
17         enablesFor="1"
18         id="de.usi.rcp.polymetricviews.adapter.sidiff.popup.
               actions.UseAsFirstSiDiffDocument"
19         label="Use as first document in DiMPl"
20         menubarPath="PV4E"/>
21   </objectContribution>
22 </extension>
```

5.2.2. Integration der SiDiff-Phasen

Damit die Modelle korrekt geladen und die symmetrische Differenz berechnet werden kann, müssen die Konfigurations- und Lade-Phasen des SiDiff-Algorithmus individuell für den neuen Dokumenttyp in DiMPl integriert werden. Per geeigneter Fallunterscheidung muss entschieden werden, welche `Loader`-Klasse benutzt wird beziehungsweise, welche Transformationsdatei für den `GraphLoader` benötigt wird. Die anschließenden Phasen des Algorithmus, die nach dem Start der Berechnung einer polymetrischen Sicht benötigt werden, sind universell für alle Dokumente und daher bereits integriert.

Beispiel: Fallunterscheidung zwischen MDL-Dateien für Simulink-Diagramme und XMI-Dateien für UML-Klassendiagramme. Für Simulink-Diagramme ist zusätzlich die Konfigurationsphase angegeben.

```
1  private Graph loadDocument(IFile document)
2  {
3    if (document.getFileExtension() != null)
4    {
5      if (document.getFileExtension().equals("xmi"))
6      {
7        if (!currentDocumentType.equals(UML_CLASSDIAGRAM))
8        {
9          // load configuration UML class diagram
10         [...]
11         currentDocumentType = UML_CLASSDIAGRAM;
12       }
13       // transform document into graph
14       Graph graph = GraphLoader.transformAndLoad("fujabaXMI.xslt",
               new File(document.getLocationURI()));
15     }
16     if (document.getFileExtension().equals("mdl"))
17     {
18       if (!currentDocumentType.equals(SIMULINK_DIAGRAM))
19       {
20         [...]
21         // init SymmetricDifference
22         SymmetricDifference.setAdvisor(new
               SetReferenceGreedyAdvisor());
23         // load configuration for simulink diagrams
24         ConfigurationLoader.loadConfiguration("
               TUDJmiSimulinkBalancedDiffConfig.xml");
25
26         currentDocumentType = SIMULINK_DIAGRAM;
27       }
28       // transform document into graph
29       Graph graph = MDLLOADER.load(document.getLocation().toString
             ());
```

```
30      }
31    }
32    return graph;
33  }
```

5.2.3. Spezielle Elemente und Beziehungen

Der `SiDiffNodeProvider` ist in der Lage für beliebige Elementtypen der Modelle die entsprechenden Knoten zu erzeugen. Für die Angabe des Elementnamens wird jedoch der Pfad aus der Phase der Pfadberechnung benutzt. Daher ist es sinnvoll, die Provider-Klasse für die Elementtypen zu erweitern und den Namen in ein Namensschema umzuwandeln, welches dem Anwender bekannt ist.

Für Beziehungstypen gibt es wie in 3.2.4 beschrieben den generischen `NodeTypeHierachyEdgeProvider`, der mit dem Elementtyp des Containerelements initialisiert werden kann. Für dokumentabhängige Beziehungstypen muss das entsprechende `IEdgeProvider`-Interface implementiert werden.

Weiterhin müssen alle Provider im DiMPl registriert werden, damit für PV4E die Liste der verfügbaren Element- und Beziehungstypen entsprechend erweitert wird.

Beispiel: Ausschnitt aus den Methoden, in denen die Knoten- und Kantenprovider initialisiert werden. Zu sehen sind einige spezielle Provider, die mit verschiedenen Parametern initialisiert werden, aber auch Standard-Provider.

```
1  protected void initNodeProviders()
2  {
3    [...]
4    // UML
5    addToNodeProviders(new UMLClassNodeProvider("Classes"));
6    addToNodeProviders(new UMLAssociationNodeProvider("Associations"
         ));
7    addToNodeProviders(new UMLParameterNodeProvider("Parameters"));
8    [...]
9
10   // Matlab/SimuLink
11   String[] systems = { "System", "Subsystem", "Rootsystem" };
12
13   addToNodeProviders(new SiDiffNodeProvider("SL Lines", "Line"));
14   addToNodeProviders(new SimulinkNodeProvider("SL Systems", "
         System", Arrays.asList(systems)));
15 }
16
17 protected void initEdgeProviders()
18 {
19   // No Edges
20   addToEdgeProviders(new NoEdgeProvider(NO_EDGES));
21
```

```
22    // UML
23    addToEdgeProviders(new InheritanceEdgeProvider("Inheritance"));
24    addToEdgeProviders(new NodeTypeHierachyEdgeProvider("Package is
          subpackage of package", "package", "packages"));
25
26    // Matlab/SimuLink
27    addToEdgeProviders(new NodeTypeHierachyEdgeProvider("System is
          subsystem of system", "System", "subsystems"));
28  }
```

5.3. Anpassungen an PV4E

Für die Anpassung des neuen Modelltyps an PV4E sind keine technischen Erweiterungen mehr notwendig. PV4E benutzt lediglich alle Komponenten, die vorher erweitert wurden. Es erhält automatisch alle Informationen über die neuen Elementtypen und deren Metriken.

5.3.1. Sichtdefinitionen erstellen

Damit die Modelldifferenzen für den neuen Dokumenttyp analysiert werden können, müssen eine Reihe von Sichtdefinitionen erstellt werden. Die Definition der neuen Sichten erfolgt dabei im *View Editor* von PV4E. Der ideale Ablauf der Analyse von Modelldifferenzen wurde bereits in Kapitel 4 allgemein und anhand den UML-Klassendiagrammen beschrieben.

Beispiel: Konfiguration (Abbildung 5.1) und Visualisierung (Abbildung 5.2) einer polymetrischen Sicht für Systemelemente einer Modelldifferenz für Matlab/Simulink-Diagramme.

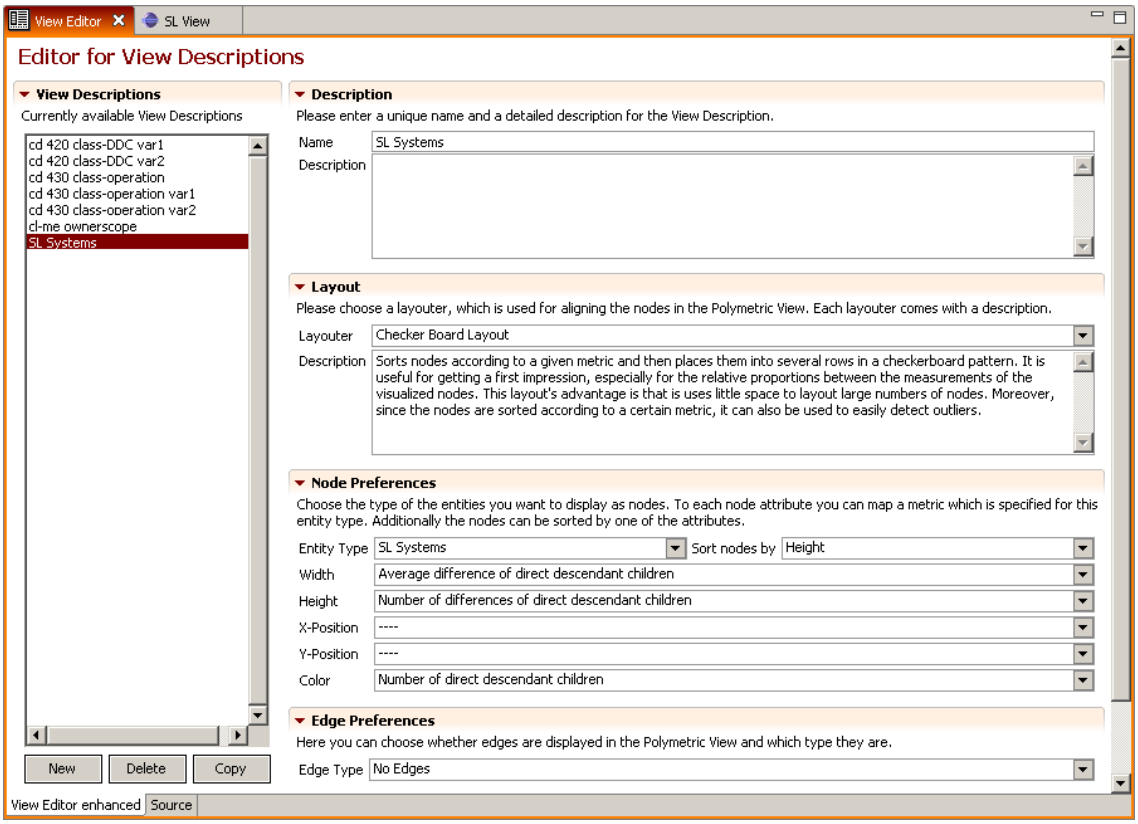


Abbildung 5.1.: Konfiguration einer Sichtdefinition im *View Editor*.

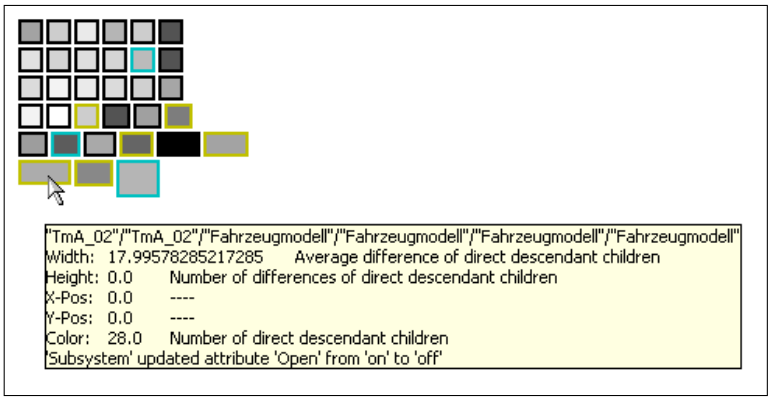


Abbildung 5.2.: Eine polymetrische Sicht für Simulink Systemelemente.

Kapitel 6.

Zusammenfassung und Ausblick

In der vorliegenden Diplomarbeit wurde gezeigt, dass polymetrische Sichten dazu eingesetzt werden können, um die Differenzen von strukturierten Modellen hinsichtlich ihrer Relevanz zu überprüfen.

Als Ansatz für die qualitative Analyse wurde die Technik der polymetrischen Sichten gewählt, die im Bereich des Reverse-Engineering eingesetzt werden, um Informationen über unbekannte Softwaresysteme zu erhalten. Für die Darstellung der polymetrischen Sichten auf Softwaresystemen werden eine Reihe von Softwaremetriken benötigt, die durch Berechnungsfunktionen auf den Softwareelementen realisiert sind. Neben dieser Einsatzmöglichkeit werden Softwaremetriken überwiegend zur Qualitätssicherung in der Softwareentwicklung eingesetzt. Die ermittelten Werte der Metriken werden anschließend in die visuellen Eigenschaften der Knotenelemente einer polymetrischen Sichten umgesetzt.

Um den Einsatz von polymetrischen Sichten zur Analyse von Modelldifferenzen zu ermöglichen, wurde das Konzept der Softwaremetriken zunächst auf strukturierte Modelle und deren Differenzen übertragen. Dazu wurde ein Werkzeug namens Difference Metrics Plugin entwickelt, welches die generische Berechnung der Metriken auf Modelldifferenzen unterstützt. Die dafür erforderliche Modelldifferenzen erzeugt der im Rahmen des SiDiff-Projekts entwickelte Vergleichsalgorithmus.

Für die Visualisierung der polymetrischen Sichten steht ein weiteres Werkzeug namens Polymetric Views for Eclipse zur Verfügung, das mit geringfügigen Änderungen für die Besonderheiten von Sichten auf Modelldifferenzen angepasst werden konnte. Die im Difference Metrics Plugin entstandenen Modell-, Differenz- und Ähnlichkeitsmetriken können daher als Grundlage für Sichtdefinitionen genutzt werden.

Über die Verknüpfung der drei genannten Werkzeuge über die gemeinsamen Schnittstellen ist ein Gesamtwerkzeug entstanden, welches die Berechnung der Modelldifferenzen und deren qualitative Analyse in einem Arbeitsfluss ermöglicht. Anhand einer Reihe von vorgestellten Sichtdefinitionen wird dies am Beispiel von UML-Klassendiagrammen gezeigt.

Darüber hinaus wurden alle beteiligten Werkzeuge durch die Benutzung von generischen Strukturen so implementiert, dass sie mit geringem Aufwand an beliebige andere Dokumenttypen, wie zum Beispiel Simulink-Diagramme angepasst werden

können. Mit Kenntnis der Dokumentstruktur können anschließend für den neuen Dokumenttyp weitere Sichtdefinitionen für polymetrische Sichten erzeugt werden, die dessen Analyse zulassen. Die dafür benötigten Metriken werden ebenfalls generisch vom Difference Metrics Plugin erzeugt.

Auftretende Schwierigkeiten

Bei der Bearbeitung der Diplomarbeit traten einige Probleme auf, deren Lösungen nicht Gegenstand der Arbeit waren oder deren Umfang unverhältnismäßig vergrößert hätten.

Das PV4E-Plugin ist so konzipiert, dass eine polymetrische Sicht zur Analyse von UML-Klassendiagrammen mit allen Adaptern berechnet werden, welche in der Lage sind, die in der Sichtdefinition zugewiesenen Metriken, zu berechnen. Diese Zuordnung erfolgt über die Namen der jeweiligen Metriken. Die Namen der in DiMPl erzeugten Metriken, werden jedoch alle generisch erzeugt, sodass die Namen mit denen aus den anderen Adaptern nicht übereinstimmen. Die Folge ist, dass mehrere Sichtdefinitionen für ein und dieselbe Analyse erzeugt werden müssen. Dies ist ein Problem, welches nur unzureichend durch manuelle Umbenennung der Metriken zu lösen ist.

Die Analyse der Veränderungen an Assoziationen von Fujaba-Klassendiagrammen ist nur schwer durchzuführen, da alle Assoziationen an den Knoten des *RootPackage* gehängt werden. Dies ist ein konzeptionelles Problem von Fujaba und mit den hier vorgestellten Werkzeugen nicht zu lösen.

Während der Erarbeitung der Sichtdefinitionen hat sich herausgestellt, dass manche Aspekte von Klassendiagrammen über spezielle Metriken eventuell noch besser herausgestellt werden könnten. Diese Metriken, deren Berechnung ebenfalls auf Modellen und der symmetrischen Differenz basieren, laufen entgegen dem generischen Konzept der Differenz- und Ähnlichkeitsmetriken. Sie ließen sich aber mit einer konzeptuellen Erweiterung in DiMPl integrieren. Der Vorteil dieser Metriken könnte sein, dass für die Analyse einer Modelldifferenz, die zugrunde liegende Datenstruktur dem Anwender nicht so gut bekannt sein muss. Die Kenntnis des internen Aufbaus der Dokumente ist für die derzeitige Analyse noch recht wichtig.

Weiterführende Arbeiten

Die vorliegende Arbeit hat gezeigt, dass die polymetrischen Sichten eine qualitative Analyse der Modelldifferenzen zulassen und diese Technik mit dem PV4E-Plugin anwendbar ist. Es bieten sich darüber hinaus noch eine Reihe von konzeptionellen und implementierungstechnischen Erweiterungen an, die in nachfolgende Arbeiten integriert werden können.

Konzeptionelle Erweiterungen

In [Hut07] wird zur Zeit eine Diplomarbeit entwickelt, welche die Nachverfolgbarkeit von Modellelementen in Versionshistorien behandelt. Eine Verknüpfung der beiden Arbeiten könnte die Analyse der Modelldifferenzen gegenseitig positiv beeinflussen, da zum Beispiel mehrere Revisionssprünge gleichzeitig in einer polymetrischen Sicht beobachtet werden könnten.

Die Nutzbarkeit der polymetrischen Sichten auf Modelldifferenzen konnte anhand der präsentierten Sichtdefinitionen für Klassendiagramme gezeigt werden. Eine empirische Analyse, welche die konkrete Zeitersparnis bei der Differenzanalyse oder den Nutzen einzelner Sichten genauer betrachtet, könnte für eine Optimierung der Sichten oder die Weiterentwicklung der Werkzeuge aus Benutzersicht genutzt werden.

Das Prinzip der polymetrischen Sichten sieht vor, dass in einer einzelnen Sicht nur ein Elementtyp betrachtet wird. Bei Simulink-Diagrammen gibt es mit den Blockelementen verschiedene Containerelemente, die sich gegenseitig enthalten können. Daher wäre es gut, wenn in einer Sicht alle Blockelementtypen gleichzeitig dargestellt werden könnten. Dies wurde bereits experimentell in Form der `SimulinkNodeProvider`-Klasse realisiert. Diese Änderung an der Darstellung erfordert aber auch bei der Berechnung der Differenz- und Ähnlichkeitsmetriken noch konzeptionelle Änderungen.

Im Zeitraum der Erstellung dieser Arbeit wurde die knotenbasierte Abfrage der Änderungen aus der symmetrischen Sicht seitens der Fachgruppe erst möglich gemacht. Dies machte die Nutzung von generischen Ähnlichkeitsmetriken erst sinnvoll nutzbar. Eventuell könnten auch die Differenzmetriken auf dieser Basis erzeugt werden und so die Metrikberechnung vereinheitlicht werden.

Die Metrikwerte für Änderungen an Elementen stehen bisher nur den direkten Vaterelementen zur Verfügung. Diese Werte könnten über Propagation in Richtung des Wurzelelements auch in der Hierarchie höher gelegenen Elementen zur Verfügung gestellt werden. Damit wären Metriken wie *Anzahl Änderungen an Attributen im Modell* verfügbar.

Erweiterung der Implementierung

Da die Werkzeuge teilweise experimentellen Status besitzen, ist die Benutzerführung an einigen Stellen etwas hakelig und könnte verbessert werden, um die Analyse der Modelldifferenzen flüssiger zu gestalten. Dies sind meist kleinere Probleme, die im wesentlichen auf Implementierungsebene gelöst werden können.

Polymetrische Sichten, bei denen die Anzahl der Änderungen an Elementen auf die Größe der Knoten angewendet werden, sind teilweise recht schwierig von Elementen ohne Änderung zu unterscheiden, wenn die Mindestgröße der Knoten relativ hoch eingestellt ist. Eine nichtlineare Skalierung der Knoteneigenschaften, um kleine

Werte deutlicher zu machen, könnte hilfreich sein.

Die Assoziationsbeziehungen zwischen Klassen bilden mit hoher Wahrscheinlichkeit einen Graphen. Die verfügbaren Tree Layouter können zwar nach einer Anpassung benutzt werden, sind aber für eine übersichtliche Darstellung eher ungeeignet. Ein (interaktiver) Graph Layouter könnte hier Abhilfe schaffen.

Derzeit existieren in PV4E sechs verschiedene Provider für Beziehungstypen zwischen unterschiedlichen Elementtypen. Diese Provider sind aber alle unabhängig vom ausgewählten Knotentyp im Sichteneditor auswählbar. Es sollte durch eine Erweiterung des Provider Interface wie bei den Metriken noch bestimmt werden können, für welchen Elementtyp die Beziehungen gelten.

In einer polymetrischen Sicht lassen sich per Popup-Menü die ausgewählten Knoten im Baum exklusiv oder zusätzlich selektieren. Hier könnten noch wesentlich mehr Selektionsmöglichkeiten hinzugefügt werden. Zum Beispiel alle Knoten aus der Kategorie *Update*, alle Elemente mit bestimmten Metrikwerten, oder alle Elemente einer Hierarchieebene.

Zur Zeit werden die verschobenen Knoten im Tree Layouter per Beziehungskante an das Vaterelement des zweiten Dokuments gezeichnet und im Tooltip ein Kommentar angefügt, welcher Knoten das Vaterelement im ersten Dokument war. Hier gibt es zum einen die Möglichkeit eine zweite Beziehungskante zum ersten Vaterelement zu zeichnen oder einen zweiten Knoten an das erste Vaterelement zu zeichnen und die Kindelemente miteinander zu verbinden.

Bei der Auswahl einer Untermenge der Entitäten bleiben die Durchnitts- und Maximalwerte der Ähnlichkeitsmetriken unverändert. Hier wäre zu überprüfen, ob eine Neuberechnung der Werte vorteilhaft für die Analyse der Modelldifferenzen ist.

Der SiDiff-Algorithmus ist noch nicht offiziell als Plugin in Eclipse integriert worden. Daher wurde die Integration nur provisorisch vorgenommen, um ständig die aktuelle Entwicklungsversion benutzen zu können. Allgemein gesehen kann eine korrekte Integration der Werkzeuge in das Eclipse-Framework und die Nutzung des Erweiterungspunkt-Mechanismus die Kommunikation und Benutzerführung der Komponenten untereinander verbessern.

Anhang A.

Struktur der Fujaba Klassendiagramme

Nachfolgend ist die Struktur der Fujaba-Klassendiagramme angegeben, wie sie in die interne Datenstruktur von SiDiff aus Abbildung 3.2 umgesetzt wird.

Die benutzten Symbole haben dabei folgende Bedeutung:

- Klassensymbole entsprechen *Nodes*, deren Name dem *NodeType* entspricht.
- Attribute der Klassen entsprechen *Attributes*.
- Kompositionen entsprechen *Edges*, deren Name dem *EdgeType* des Typs *Nesting* entspricht.
- Gerichtete Assoziationen entsprechen normalen *Edges*, deren Name dem *EdgeType* entspricht.
- Kommentare geben Informationen über den Wertebereich der Attribute

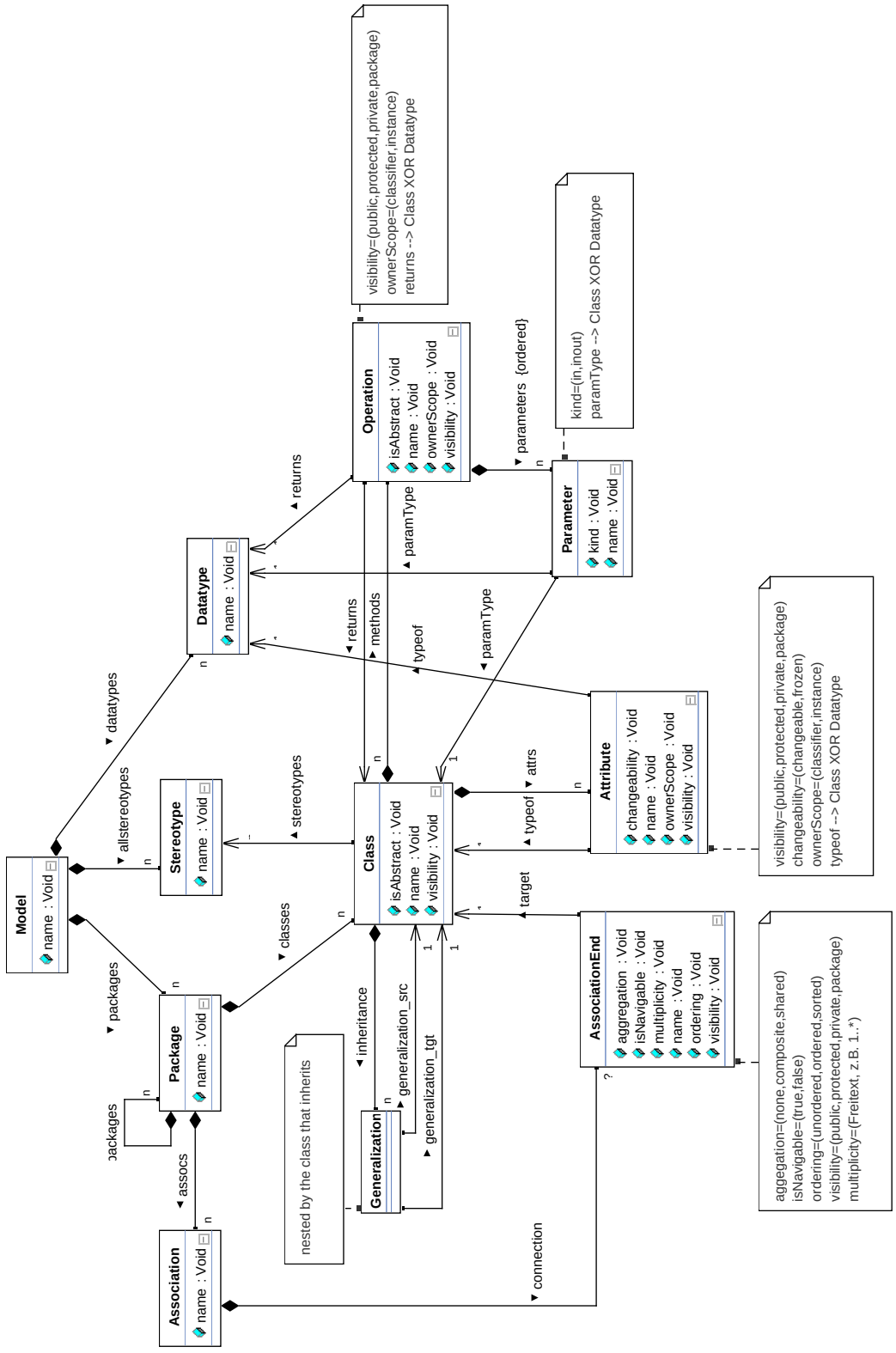


Abbildung A.1.: Die Elementstruktur der UML-Klassendiagramme in Fujaba.

Anhang B.

Konfiguration für Fujaba-Klassendiagramme

Nachfolgend die Konfiguration für Fujaba-UML-Klassendiagramme in der `plugin.xml` des Polymetric Views for Eclipse Plugins. Sie basiert auf der internen Datenstruktur für Klassendiagramme aus Anhang A und bildet die Grundlage für die automatisch generierten Metriken.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
  <extension point="de.usi.rcp.polymetricviews.adapter.sidiff.diagramtype">
    <diagram type="UMLClassdiagram">
      <nodetype name="parameter">
        <attribute isunordered="true" name="name"/>
        <attribute isunordered="true" name="kind"/>
        <edgetype issingle="true" name="paramType"/>
      </nodetype>
      <nodetype name="operation">
        <attribute isunordered="true" name="name"/>
        <attribute isunordered="true" name="isAbstract"/>
        <attribute isunordered="true" name="ownerScope"/>
        <attribute ascendingorder="private,package,protected,public" isunordered="false" name="visibility"/>
        <edgetype issingle="true" name="returns"/>
        <nestednodetype edgetype="parameters" name="parameter"/>
      </nodetype>
      <nodetype name="attribute">
        <attribute name="name" isunordered="true"/>
        <attribute name="changeability" isunordered="true"/>
        <attribute name="ownerScope" isunordered="true"/>
        <attribute name="visibility" isunordered="false" ascendingorder="private,package,protected,public"/>
        <edgetype name="typeof" issingle="true"/>
      </nodetype>
      <nodetype name="generalization">
        <edgetype name="generalization_src" issingle="true"/>
        <edgetype name="generalization_tgt" issingle="true"/>
```

```
28      </nodetype>
29      <nodetype name="class">
30        <attribute name="name" isunordered="true"/>
31        <attribute name="isAbstract" isunordered="true"/>
32        <attribute name="visibility" isunordered="false" ↩
              ascendingorder="private,package,protected,public"/>
33        <edgetype name="stereotypes" issingle="false" nodenames="↩
              interface,reference,singleton,type"/>
34        <nestednodetype name="operation" edgetype="methods"/>
35        <nestednodetype name="attribute" edgetype="attrs"/>
36        <nestednodetype name="generalization" edgetype="↩
              inheritance"/>
37      </nodetype>
38      <nodetype name="associationEnd">
39        <attribute name="name" isunordered="true"/>
40        <attribute name="aggregation" isunordered="true"/>
41        <attribute name="isNavigable" isunordered="true"/>
42        <attribute name="multiplicity" isunordered="true"/>
43        <attribute name="ordering" isunordered="true"/>
44        <attribute name="visibility" isunordered="false" ↩
              ascendingorder="private,package,protected,public"/>
45        <edgetype name="target" issingle="true"/>
46      </nodetype>
47      <nodetype name="association">
48        <attribute name="name" isunordered="true"/>
49        <nestednodetype name="associationEnd" edgetype="connection↩
              "/>
50      </nodetype>
51      <nodetype name="package">
52        <attribute name="name" isunordered="true"/>
53        <nestednodetype name="association" edgetype="assocs"/>
54        <nestednodetype name="package" edgetype="packages"/>
55        <nestednodetype name="class" edgetype="classes"/>
56      </nodetype>
57      <nodetype name="stereotype">
58        <attribute name="name" isunordered="true"/>
59      </nodetype>
60      <nodetype name="datatype">
61        <attribute name="name" isunordered="true"/>
62      </nodetype>
63      <nodetype name="model">
64        <attribute name="name" isunordered="true"/>
65        <nestednodetype name="package" edgetype="packages"/>
66        <nestednodetype name="stereotype" edgetype="allstereotypes↩
              "/>
67        <nestednodetype name="datatype" edgetype="datatypes"/>
68      </nodetype>
69    </diagram>
70  </extension>
71 </plugin>
```

Anhang C.

Verfügbare Layouter in PV4E

Nachfolgend werden die in PV4E verfügbaren Layouter beschrieben. Es handelt sich um die in [Lan03] vorgestellten Layouter. Hinzugefügt wurde noch der Circle Layouter, der aufgrund von Problemen bei der Übersichtlichkeit nur selten zu einem Einsatz kommt. Bei einigen der Layoutern existiert eine horizontale und eine vertikale Variante. Die Beschreibung erfolgt dabei immer aus Sicht der horizontalen Variante. Prinzipbedingt haben je nach Layout nicht immer alle Metrikzuweisungen Auswirkung auf die jeweilige Knoteneigenschaften, sodass sich die Anzahl der nutzbaren Metriken entsprechend reduziert.

Stapled: Alle Knoten werden in einer langen Reihe am oberen Bildschirmrand angeordnet. Jeder Knoten wird dabei unabhängig von der Breite ohne Abstand direkt an den Nachbarknoten gezeichnet. Die Höhe der Knoten ergibt sich aus einer zweiten Metrik. Korrelieren diese Metriken und werden die Knoten nach der Breite sortiert, erhält man eine treppenartige Struktur, die nur von eventuellen Ausreißern durchbrochen wird. Dieser Layouter steht in einer horizontalen und vertikalen Variante zur Verfügung.

Histogram: Die Knoten werden an einer vertikalen Achse abhängig von der entsprechenden Metrik positioniert. Alle Knoten mit dem gleichen Metrikwert werden anschließend nebeneinander in der gleichen Reihe gezeichnet. Dieses Layout wird hauptsächlich zur Analyse der Verteilung von Knoten bezüglich dieser Metrik benutzt. Dieser Layouter steht in einer horizontalen und vertikalen Variante zur Verfügung.

Scatterplot: Die Knoten werden in einem orthogonalen Gitter angeordnet abhängig von zwei Metriken angeordnet. Elemente mit den gleichen Metrikwerten werden übereinander gezeichnet. In großen Knotenmengen ist dieser Layouter dazu geeignet, zwei verschiedene Metriken miteinander zu vergleichen. Die Skalierbarkeit ist sehr hoch, da die Größe des Diagramms nicht von der Anzahl der Knoten, sondern nur von den Wertebereichen der Metriken abhängig ist.

Checker: Die Sortierung der Knoten erfolgt nach einer gegebenen Metrik. Anschließend werden die Knoten schachbrettartig Reihen und Spalten gezeichnet. Dadurch bieten sie einen ersten Eindruck von den relativen Größen der Metrikwerte. Der Platzverbrauch der Sicht ist moderat und ist ebenfalls gut geeignet zur Identifizierung von Ausreißern.

Tree: Dieser Layouter erfordert die Verfügbarkeit von Beziehungskanten zwischen den darzustellenden Knoten. Anschließend werden die Knoten anhand der hierarchischen Struktur in mehreren Ebenen angeordnet. Gerade in strukturierten Dokumenten ist dieser Layouter essentiell. Im Falle von Klassendiagrammen können so zum Beispiel Vererbungsbeziehungen sichtbar gemacht werden. Dieser Layouter steht in einer horizontalen und vertikalen Variante zur Verfügung.

Circle: Alle darzustellenden Knoten werden einfach entlang eines Kreises angeordnet. Dadurch, dass die Knoten relativ weit voneinander entfernt sind, wird die Benutzung von Beziehungskanten begünstigt.

Anhang D.

Technologien und Benutzung

Während der Erstellung der Diplomarbeit kamen folgende Hilfsmittel zum Einsatz:

Eclipse SDK in der Version `3.2.2` vom 12.02.2007

Java SDK in der Version `1.5.0_11` von Mitte Dezember 2006

Fujaba Tool Suite in der Version `4.3.2` vom 24.August 2006

XMI Export Plugin angepasste Version vom 22.03.2005

Die Hilfsmittel in den aufgeführten Versionen befinden sich in der Variante für Windows im Unterverzeichnis **tools** auf der beigefügten CD. Weiterhin befinden sich in den verschiedenen Unterverzeichnissen die folgenden Programme und Dokumente:

pv4e Die lauffähige Entwicklungsversion von PV4E mit DiMPl und SiDiff

repository Die Quelltexte und weitere Dokumentation aus dem SVN-Repository

thesis Diese Diplomarbeit als PDF-Dokument

presentation Präsentationsgrafiken, die auf der Hannover-Messe eingesetzt wurden

PV4E kann unter Windows über die EXE-Datei im Ordner PV4E gestartet werden. Eine Linux-Version kann über die in der Entwicklungsversion enthaltenen `.product`-Dateien erzeugt werden.

Literaturverzeichnis

[Ecl07a] Eclipse Foundation, Inc., Kanada: *Eclipse - an open development platform*, im Internet unter http://www.eclipse.org/, 2007.

[Ecl07b] Eclipse Foundation, Inc., Kanada: *The Official Eclipse FAQs*, im Internet unter http://wiki.eclipse.org/index.php/The_Official_Eclipse_FAQs, 2007.

[Fal05] Falk, Jens: *Polymetric Views - Ein Plugin für Fujaba*. Studienarbeit, Universität Siegen, Deutschland, 2005.

[FP96] Fenton, N. E. und S. L. Pfleeger: *Software Metrics - A Rigorous & Practical Approach (Englisch)*. International Thompson Computer Press, Second Edition Auflage, 1996.

[Ges07a] Gesellschaft für Informatik e.V., Deutschland: *SE 2007 - die Konferenz rund um Softwaretechnik*, im Internet unter http://www.se2007.de/, 2007.

[Ges07b] Gesellschaft für Informatik e.V., Deutschland: *Vergleich und Versionierung von UML-Modellen, ein Workshop im Rahmen der SE 2007*, im Internet unter http://pi.informatik.uni-siegen.de/gi/fg211/VVUM07/, 2007.

[Gir02] Girschick, Martin: *UMLDiff - Erkennung und Analyse von Unterschieden in Klassendiagrammen und Sequenzdiagrammen*. Diplomarbeit, Technische Universität Darmstadt, Deutschland, März 2002.

[HS96] Henderson-Sellers, B.: *Object-Oriented Metrics: Measures of Complexity (Englisch)*. Prentice-Hall, 1996.

[Hut07] Hutter, Hermann: *Nachverfolgbarkeit von Modellelementen in Versionshistorien*. Diplomarbeit, Universität Siegen, Deutschland, Mai 2007.

[Kel07] Kelter, Udo: *Lehrmodul Dokumentdifferenzen*, im Internet unter http://pi.informatik.uni-siegen.de/lehre/2006w/LM/lm_dif_20070322_info.html. Universität Siegen, Deutschland, März 2007.

[Lan03] Lanza, Michele: *Object-Oriented Reverse Engineering (Englisch).* Doktorarbeit, Universität Bern, Schweiz, 2003.

[Lüc04] Lück, Stephan: *Ein Differenzanzeige-Plugin für Klassendiagramme in Fujaba.* Studienarbeit, Universität Siegen, Deutschland, 2004.

[Lüc06] Lück, Stephan: *Differenzberechnung hierarchischer Softwareentwicklungsmodelle.* Diplomarbeit, Universität Siegen, Deutschland, 2006.

[LK94] Lorenz, M. und J. Kidd: *0bject-Oriented Software Metrics: A Practical Guide (Englisch).* Prentice-Hall, 1994.

[Obj03] Object Management Group: *UML v1.5 Specification*, im Internet unter http://www.omg.org/cgi-bin/doc?formal/03-03-01, March 2003.

[OWK03] Ohst, Dirk, Michael Welle und Udo Kelter: *Differences between versions of UML diagrams (Englisch).* In: *Proc. of the 9th European software engineering conference, Helsinki, Finland*, Seiten 227–236, 2003.

[PS07] Power Software, Schottland: *Krakatau Project Manager*, im Internet unter http://www.powersoftware.com/, 2007.

[Rot05] Roth, Lukas: *Metrics-Plugin.* Studienarbeit, Universität Paderborn, Deutschland, 2005.

[Tre07] Treude, Christoph: *Einsatz multidimensionaler Suchstrukturen zur Optimierung der Bestimmung von Dokumentdifferenzen.* Diplomarbeit, Universität Siegen, Deutschland, März 2007.

[Uni07a] Universität Paderborn, Deutschland: *Fujaba Tool Suite*, im Internet unter http://www.fujaba.de/, 2007.

[Uni07b] Universität Siegen, Deutschland: *The SiDiff Project*, im Internet unter http://www.sidiff.org/, 2007.

[Weh04] Wehren, Jürgen: *Ein XMI-basiertes Differenzwerkzeug für UML-Diagramme.* Diplomarbeit, Universität Siegen, Deutschland, September 2004.

[WK06] Wenzel, Sven und Udo Kelter: *Model-Driven Design Pattern Detection Using Difference Calculation (Englisch).* In: *Proc. of the 1st International Workshop on Pattern Detection For Reverse Engineering (DPD4RE), co-located with 13th Working Conference on Reverse Engineering (WCRE'06), Benevento, Italy*, October 2006.

Alle Internetseiten wurden zuletzt am 13. Mai 2007 auf ihre Existenz überprüft.