Karlsruher Institut für Technologie

Extrahieren von Code-Änderungen aus einem Commit für kontinuierliche Integration von Leistungsmodellen

Bachelorarbeit von

Ilia Chupakhin

an der Fakultät für Informatik
Institut für Programmstrukturen und Datenorganisation (IPD)

Erstgutachter:	Prof. Dr.-Ing. Anne Koziolek
Zweitgutachter:	Prof. Dr. Ralf H. Reussner
Betreuender Mitarbeiter:	M.Sc. Manar Mazkatli
Zweiter betreuender Mitarbeiter:	M.Sc. Yves R. Kirschner

24. März 2020 – 03. September 2020

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Änderungen entnommen wurde.

Ort, Datum

. . . . . .

(Ilia Chupakhin)

Inhaltsverzeichnis

1 Zusammenfassung

Ein Performance-Modell ermöglicht den Software-Entwicklern eine frühzeitige Analyse von programmierten Komponenten in Bezug auf Leistungseigenschaften, wie zum Beispiel Speicherbedarf oder Ausführungsdauer. Das Performance-Modell muss mit allen anderen im System vorhandenen Modellen konsistent gehalten werden, aber insbesondere mit dem Code-Modell. Änderungen in dem Code-Modell beeinträchtigen seine Performance-Eigenschaften und verursachen deshalb eine Aktualisierung des Performance-Modells. Eine manuelle Aktualisierung des Performance-Modells für große Systeme ist aufwändig und fehleranfällig. Ein Ansatz für eine effiziente automatisierte Aktualisierung von Performance-Modellen wurde in [20] vorgestellt und in [21] weiterentwickelt. Wir bezeichnen diesen Ansatz als 'kontinuierliche Integration von Leistungsmodellen' oder kurz 'CIPM-Ansatz' (Englisch: Continuos Integration of Performance Model).

Als erster Schritt in dem CIPM-Ansatz werden Änderungen aus einem Git-Commit gelesen und auf die Code-Modelle angewandt. Anschließend findet eine inkrementelle Aktualisierung des Performance-Modells statt. In dieser Bachelorarbeit implementieren wir den ersten Schritt für den CIPM-Ansatz, und zwar, wir verknüpfen den CIPM-Ansatz mit Git-Repository und extrahieren Änderungen aus Git-Commits. Anhand von den extrahierten Änderungen passen wir die existierenden Code-Modelle an. Anschließend werden diese Änderungen zu den Performance-Modellen automatisch propagiert. Für diesen Zweck haben wir die existierenden Konsistenzerhaltungsregeln angepasst und einige neue Regeln implementiert.

Unsere Implementierung haben wir in einer Fallstudie evaluiert. Für ein Projekt haben wir unterschiedliche Arten von Änderungen simuliert und sie als Git-Commits in einem Git-Repository gespeichert. Danach haben wir dieses Projekt in Vitruvius integriert, Änderungen aus Commits gelesen und die entsprechenden Modelle angepasst. Anschließend haben wir die Korrektheit der aktualisierten Code- und Performance-Modelle überprüft. Unsere Ergebnisse bestätigen die korrekte Aktualisierung von Code- und Performance-Modellen in den 96,6% der durchgeführten Tests.

2 Einleitung

In einer modellgetriebenen Software-Entwicklung [33] werden alle Artefakte des zu entwickelnden Systems als Modelle gesehen und behandelt. Als Artefakte können zum Beispiel Quellcode-, Architektur- oder Performance-Modelle dienen.

Ein Performance-Modell ermöglicht Software-Entwicklern eine frühzeitige Analyse von programmierten Komponenten in Bezug auf Leistungseigenschaften, wie zum Beispiel Speicherbedarf oder Ausführungsdauer. Das Performance-Modell muss mit allen anderen im System vorhandenen Modellen konsistent gehalten werden, aber insbesondere mit dem Code-Modell. Änderungen in dem Code-Modell beeinträchtigen seine Performance-Eigenschaften und verursachen deshalb eine Aktualisierung des Performance-Modells. Eine manuelle Aktualisierung des Performance-Modells für große Systeme ist aufwändig und fehleranfällig. In einem agilen Software-Entwicklungsprozess, der dem Prinzip der kontinuierlichen Integration [32] folgt, wird das System mehrmals am Tag geändert, was eine Aktualisierung des Performance-Modelles nach jeder Änderung verursachen würde.

Ein Ansatz für eine effiziente automatisierte Aktualisierung von Performance-Modellen wurde in [20] vorgestellt und in [21] weiterentwickelt. Wir bezeichnen diesen Ansatz als 'kontinuierliche Integration von Leistungsmodellen' oder kurz 'CIPM-Ansatz' (Englisch: Continuos Integration of Performance Model). Der CIPM-Ansatz beschreibt eine kontinuierliche Integration von Performance-Modellen in einem agilen Software-Entwicklungsprozess. Wenn Änderungen in dem Code-Modell stattfinden, wird das Performance-Modell nicht für das ganze System neu aufgebaut, sondern nur für die geänderten Teile aktualisiert.

Als erster Schritt in dem CIPM-Ansatz werden Änderungen aus einem Git-Commit gelesen und auf die Code-Modelle angewandt. Anschließend findet eine inkrementelle Aktualisierung des Performance-Modells statt. In dieser Bachelorarbeit implementieren wir den ersten Schritt für den CIPM-Ansatz, und zwar, wir verknüpfen den CIPM-Ansatz mit Git-Repository und extrahieren Änderungen aus Git-Commits. Anhand von den extrahierten Änderungen passen wir das Code-Modell an.

Wenn das Code-Modell angepasst ist, kann anschließend auch das Performance-Modell aktualisiert werden. Eine Aktualisierung von Performance-Modellen findet nur dann statt, wenn die passenden Konsistenzerhaltungsregeln definiert sind. In dieser Bachelorarbeit passen wir die bereits vorhandenen Konsistenzerhaltungsregeln an und definieren neue Regeln.

Unsere Implementierung haben wir in einer Fallstudie evaluiert. Für ein Projekt haben wir unterschiedliche Arten von Änderungen simuliert und sie als Git-Commits in einem Git-Repository gespeichert. Danach haben wir dieses Projekt in Vitruvius integriert, Änderungen aus den Git-Commits gelesen und die entsprechenden Modelle angepasst. Anschließend haben wir die Korrektheit der aktualisierten Code- und Performance-

Modelle überprüft. Unsere Ergebnisse bestätigen die korrekte Aktualisierung von Code- und Performance-Modellen in den 96,6% der durchgeführten Tests.

Das Kapitel 3 gibt einen Überblick über die Grundlagen. In dem Kapitel 4 diskutieren wir verwandte Arbeiten. Das Konzept für unsere Implementierung stellen wir in dem Kapitel 5 vor. Unsere Implementierung beschreiben wir in dem Kapitel 6. Das Kapitel 7 beschreibt die Evaluierung unserer Implementierung und zeigt die Evaluierungsergebnisse. In dem Kapitel 8 diskutieren wir Limitierungen unserer Implementierung und Vorschläge für die zukünftige Arbeit. In dem Kapitel 9 ziehen wir Schlussfolgerungen zu der durchgeführten Arbeit.

3 Grundlagen

In diesem Kapitel beschreiben wir die Grundlagen für unsere Arbeit. Dazu zählen zum Beispiel Konzepte, Programme, Frameworks, Programmiersprachen, Bibliotheken und Plugins.

3.1 Modellgetriebene Software-Entwicklung

In einer modellgetriebenen Software-Entwicklung [33] werden alle Artefakte des zu entwickelnden Systems als Modelle gesehen und behandelt. Als Artefakte können zum Beispiel Quellcode-, Architektur- oder Performance-Modelle dienen. Das zu entwickelte System wird als Original betrachtet und wird durch Modelle beschrieben. Einige der Vorteile von einer modellgetriebenen Software-Entwicklung sind Steigerung der Entwicklungsgeschwindigkeit und Softwarequalität sowie Handhabbarkeit von komplexen Systemen durch Abstraktion [2]. Darüber hinaus kann aus definierten Modellen eine lauffähige Software automatisiert erzeugt werden [33]. Um Modelle zu beschreiben, kann man domänenspezifische Sprachen definieren.

3.2 Eclipse Java Developement Tool (Eclipse JDT)

Eclipse Java Developement Tool (Eclipse JDT) [6] bietet eine Sammlung von Plugins, die eine Java-Entwicklungsumgebung realisieren. Mit JDT lassen sich außer Java-Projekte auch neue Plugins erstellen, die dann neue Funktionalitäten für die Entwicklungsumgebung definieren und somit eine kontinuierliche Erweiterung der Entwicklungsumgebung ermöglichen.

JDT enthält einen Parser für den Java-Quellcode. Der geparste Java-Quellcode wird als JDT-Modell gespeichert. Das JDT-Modell repräsentiert einen abstrakten Syntaxbaum (Englisch: Abstract syntax tree (AST)) [1]. Als Knoten werden einzelne Java-Elemente (Variablen, Schleifen, Verzweigungen etc.) und als Kanten die Code-Struktur gespeichert.

JDT kann Änderungen in den erstellten Modellen detektieren. Falls sich ein Modell ändert, werden Änderungen analysiert, in Deltas umgeformt und die Deltas dann an alle Listener geschickt. Dieser Benachrichtigungsmechnismus ist allerdings nicht von dem JDT-AST-Modul, sondern von dem JDT-Core-Modul bereitgestellt. Das bedeutet, dass die ausgerechneten und abgeschickten Deltas grobgranulare Änderungen enthalten. Falls aber feingranulare Änderungen erwünscht sind, müssen die Deltas mit JDT-AST weiter verfeinert werden.

3.3 Eclipse Modeling Framework (EMF)

Eclipse Modeling Framework (EMF) [7] ist ein Framework für die Software-Modellierung. In dem Framework lassen sich Editoren erzeugen, die dann Instanzen von den erstellten Meta-Modellen produzieren können. Außerdem ermöglicht es eine Generierung des Java-Quellcodes aus den erstellten Modellen. EMF folgt dem Essential-Meta-Object-Facility-Standard (EMOF) [23]. Der EMOF-Standard wird in EMF durch Ecore implementiert. Ecore selbst ist ein Meta-Meta-Modell und somit auf der M3-Ebene in dem EMOF-Standard angesiedelt. Die Abbildung 3.1 zeigt ein Bespiel, wo alle Modellirungsebenen aus dem EMOF-Standard zu sehen sind. Wie schon erwähnt stellt Ecore das Meta-Meta-Modell dar und ist somit auch selbstbeschreibend. Als Meta-Modell kann man sich zum Beispiel ein UML-Diagramm vorstellen (M2-Ebene). Mit einem UML-Diagramm lassen sich konkrete Modelle erstellen oder anders formuliert Meta-Modell-Instanzen (M1-Ebene), die dann Objekte aus der Realität modellieren (M0-Ebene).

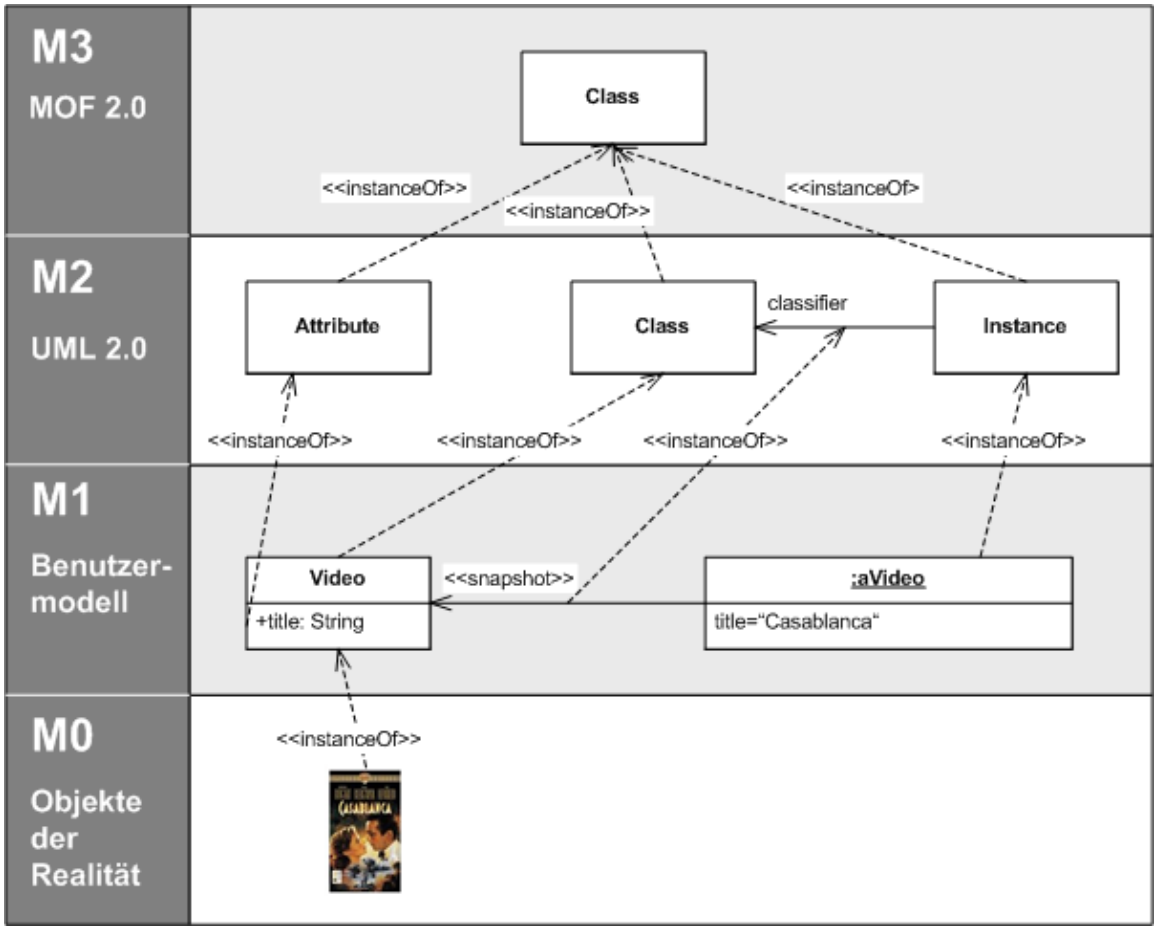

Abbildung 3.1: Modellierungsebenen in dem EMOF-Standard. Quelle: [23]

Da Ecore eine einheitliche Struktur für alle modellierten Elemente erzwingt, lassen sich auch Modell-zu-Modell-Transformationen zwischen den unterschiedlichen Modelltypen definieren. Modell-zu-Modell-Transformationen können zum Beispiel durch eine neu definierte domänenspezifische Sprache beschrieben werden.

3.4 Java Model Parser and Printer (JaMoPP)

Mit Java Model Parser and Printer (JaMoPP) [14] lassen sich EMF-Modelle von dem Java-Quellcode erstellen (Parser) und umgekehrt (Printer). In EMF [3.3] wird der Java-Quellcode als Text gesehen. Das bedeutet, wenn der Java-Quellcode einmal aus einem EMF-Modell generiert wurde, lässt er sich nicht mehr zurück in ein Modell umwandeln. JaMoPP füllt diese Funktionalitätslücke, sodass der Java-Quellcode als ein EMF-Modell dargestellt wird. Daraus folgt, dass Modell-zu-Modell-Transformationen zwischen dem Java-Quellcode und den anderen Modelltypen definiert und durchgeführt werden können. JaMoPP ist als ein Eclipse-Plugin [5] implementiert.

3.5 Palladio Component Model (PCM)

Palladio Component Model (PCM) ist eine domänenspezifische Sprache, die im Rahmen des Palladio-Ansatzes [25] entwickelt wurde. Mit dieser Sprache kann man eine komponentenbasierte Architektur des zu entwickelnden Systems definieren. Das PCM ermöglicht eine frühzeitige Analyse von unterschiedlichen Qualitätseigenschaften des Systems wie zum Beispiel Performance, Zuverlässigkeit und Wartbarkeit. Somit kann PCM als ein Performance-Modell dienen. Jede mit PCM definierte Komponente enthält unter anderem Service-Effect-Specifications (SEFFs). Eine SEFF stellt das Performance-Modell einer in der Komponente definierten Methode dar. Alle in einer Methode vorkommenden Statements werden in einer SEFF durch einen der vier Typen von Aktionen beschrieben:

- Internal Action - repräsentiert alle Statements, die nur auf dem Code operieren, der innerhalb einer Komponente definiert ist. Das bedeutet, es gibt keine Zugriffe auf andere Komponenten.
- External Call - repräsentiert alle Statements, die auf eine andere Komponente zugreifen.
- Loop Action - repräsentiert eine Schleife, wobei innerhalb dieser Schleife mindestens ein External Call enthalten ist.
- Branch Action - repräsentiert eine Verzweigung, wobei innerhalb dieser Verzweigung mindestens ein External Call enthalten ist.

3.6 Vitruvius

View-cenTRic engineering Using a VIrtual Underlying Single model (Virtruvius) [27] ist ein Framework, das im Kontext der modellgetriebenen Software-Entwicklung [3.1] eingesetzt werden kann. Vitruvius basiert auf dem Konzept von Single Underlying Model (SUM). Ein Beispiel für ein SUM-Metamodell ist in der Abbildung 3.2 dargestellt.

In einem SUM werden alle in dem System verfügbaren Modelle gespeichert. Ein Zugriff auf diese Modelle ist nur über spezielle Sichten möglich. In einer Sicht kann man eins

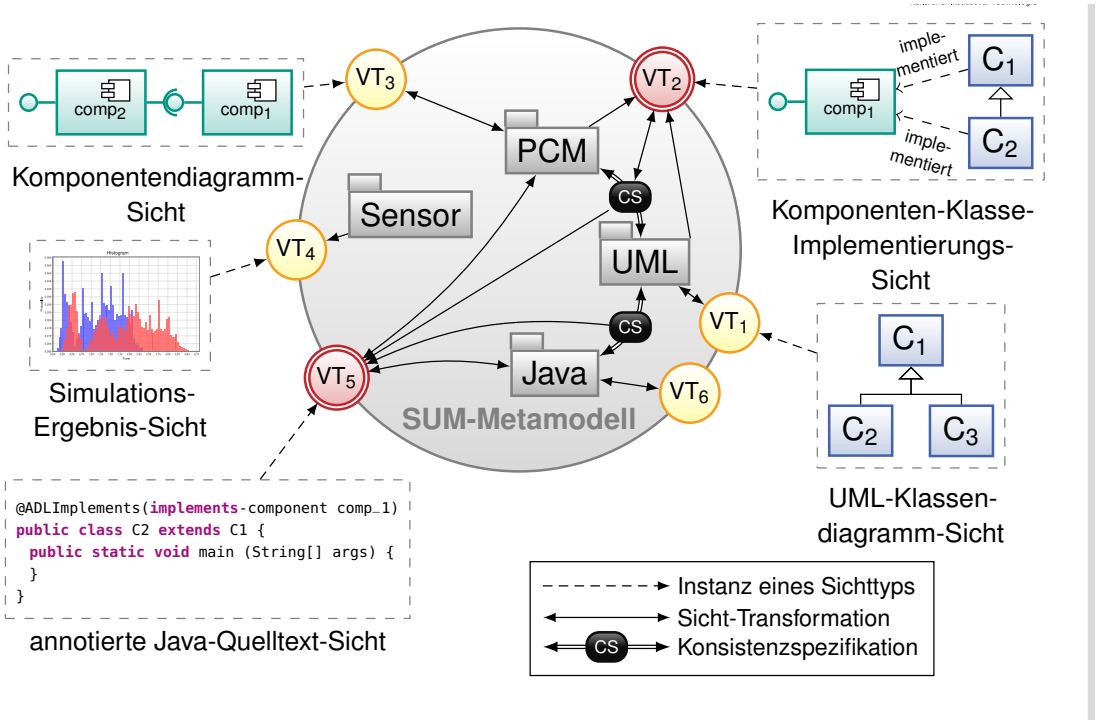

Abbildung 3.2: Ein SUM-Metamodell, Quelle: [2] Foliensatz 15 Folie 32

oder mehrere Modelle auswählen und sie manipulieren. Während mehrere Software-Entwickler an einem System über definierte Sichten arbeiten, hält Vitruvius alle in dem System definierten Modelle konsistent. Somit sieht jeder Entwickler immer den aktuellsten Zustand von dem System. Vitruvius unterstützt mehrere Modelltypen, darunter auch PCM [3.5] und JaMoPP-Modelle [3.4].

Änderungen in einem Modell werden zu den anderen Modellen mithilfe von Konsistenzerhaltungsregeln propagiert. Die Konsistenzerhaltungsregeln besagen, wie ein Modell anzupassen ist, wenn ein anderes korrespondierendes Modell geändert wurde. Eine Konsistenzerhaltungsregel ist nur für zwei bestimmte Modelle, nur für das Propagieren in eine Richtung (zum Beispiel nur von Modell A zu Modell B, aber nicht umgekehrt) und nur für einen bestimmten Typen von Änderungen definiert. Die Konsistenzerhaltungsregeln sind in der Reaction-Sprache [3.6.2] geschrieben.

3.6.1 Java Monitor

Messinger hat in seiner Masterarbeit [22] einen Java-Monitor entwickelt. Sobald der Java-Quellcode geändert und gespeichert wurde, werden Änderungen von dem Java-Monitor abgefangen, transformiert und an Vitruvius weitergeleitet. Die Funktionsweise des Java-Monitors für den Änderungtypen 'Umbenennen einer Methode' ist in der Abbildung 3.3 dargestellt. Zuerst werden Änderungen in JDT-Java-Modellen von JDT-Core detektiert und in *JavaElementDeltas*-Objekte zerlegt. Anschließend benachrichtigt JDT-Core alle Listener. *ASTChangeListener* bekommt die *JavaElementDeltas*-Objekte, analysiert die betroffenen JDT-AST-Knoten, vergleicht den neuen und den alten Zustand der JDT-AST-Knoten und

ermittelt somit noch feinere Änderungen. Die ermittelten Änderungen werden an *Change-Responder* übergeben. *ChangeResponder* transformiert die angekommenen Änderungen in *EMFModelChanges*. Diese EMF-Änderungen sind als erstellte, gelöschte oder aktualisierte JaMoPP-Elemente ausgedrückt. Anschließend werden die EMF-Änderungen an Vitruvius weitergeleitet. Vitruvius kann dann basierend auf Änderungen in JaMoPP-Modellen die korrespondierenden Modelle aktualisieren (Modell-zu-Modell-Transformation).

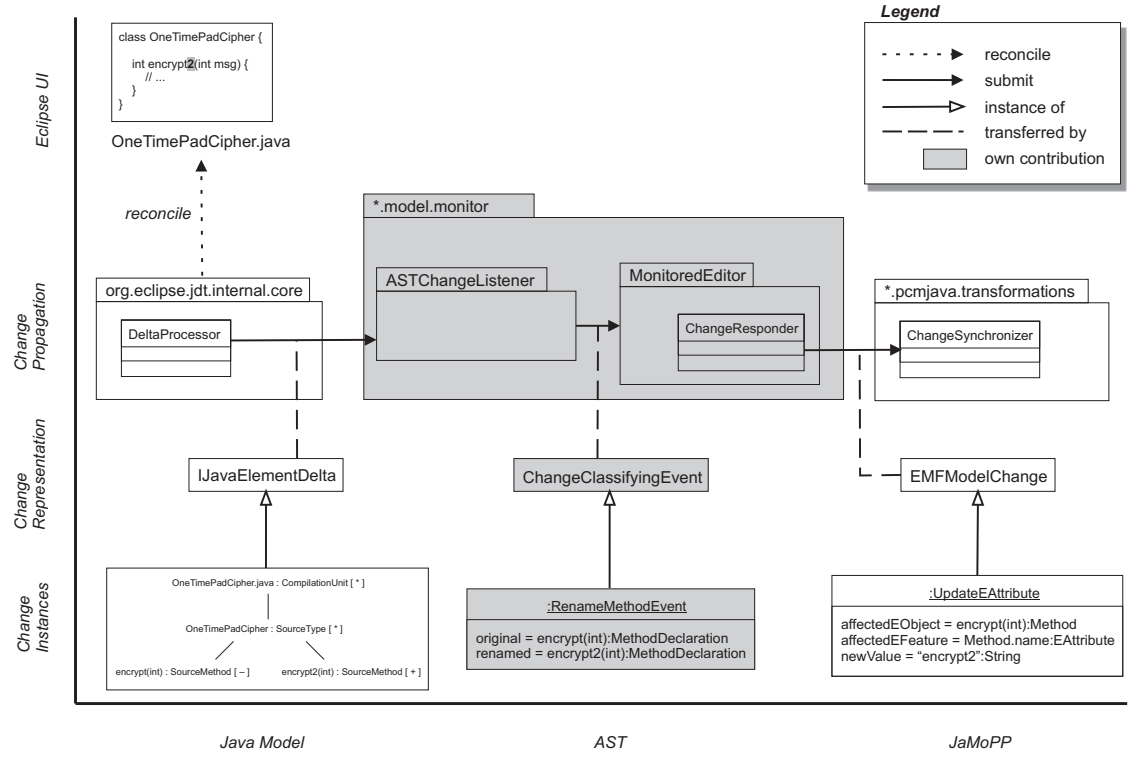

Abbildung 3.3: Funktionsweise des Java-Monitors für den Änderungtypen 'Umbenennen einer Methode'. Zuerst werden Änderungen in JDT-Java-Modellen ermittelt, sie dann anhand von JDT-AST-Knoten verfeinert und anschließend in EMF-Änderungen transformiert. Quelle: [22] Seite 47

3.6.2 Reaction-Sprache

Eine Reaction-Sprache ist eine domänenspezifische Sprache, die eine Konsistenzerhaltung zwischen zwei Meta-Modellen ermöglicht [29]. Sie wurde im Rahmen einer Dissertation von Kramer erstellt [16] und wird in Vitruvius 3.6 eingesetzt. Mit der Reaction-Sprache werden Transformationen von Änderungen in einer Meta-Modell-Instanz in Änderungen in einer anderen Meta-Modell-Instanz definiert. Die erstellten Transformationen sind unidirektional. Für ein bidirektionales Verhalten müssen zwei Transformationen definiert werden. Die Reaction-Sprache bietet zwei Arten von Funktionen an:

- reactions: Eine Reaction-Funktion kann als ein Dispatcher angesehen werden. Sie überprüft zuerst den Typen von der erfolgten Änderung. Falls die Reaction-Funktion

für diesen Änderungstypen zuständig ist, findet sie die korrespondierenden Elemente in zwei Meta-Modell-Instanzen, deren Konsistenz wieder erstellt werden muss, und ruft eine passende Routine-Funktion auf.

- routines: Eine Routine-Funktion passt die zweite Meta-Modell-Instanz an, sodass sie mit der ersten Meta-Modell-Instanz wieder konsistent ist.

3.6.3 Propagierungsstrategien

Vitruvius bietet zwei Strategien, um Änderungen in einem Modell zu dem korrespondierenden Modell zu propagieren.

- Delta-basierte Propagierungsstrategie bedeutet, dass Vitruvius Informationen über jede kleine (=feingranulare) Änderung, sowie über die Reihenfolge von den Änderungen verfügt. Basierend auf diesen Informationen werden die korrespondierenden Modelle angepasst. Die delta-basierte Propagierungsstrategie wird in Vitruvius als Standard-Mechanismus benutzt. Eine Voraussetzung für die delta-basierte Propagierungsstrategie ist, dass die Modelle nur unter 'Vitruvius-Beobachtung' geändert werden dürfen. Falls das nicht der Fall ist, kommt die zustandsbasierte Propagierungsstrategie zum Einsatz.
- Zustandsbasierte Propagierungsstrategie nimmt zwei Zustände von einem Modell als Eingabe und ermittelt alle feingranularen Änderungen zwischen den Zuständen. Anhand von diesen Änderungen können die korrespondierenden Modelle angepasst werden. Die zustandsbasierte Propagierungsstrategie wurde noch nicht in Vitruvius für Konsistenzerhaltung eingesetzt und gilt als experimentell.

3.7 EMF Compare

Mit dem EMF-Compare-Framework [8] können EMF-Modelle [7] miteinander verglichen und ihre Differenzen ausgerechnet werden. Der Vergleichsprozess lässt sich grob in sechs Phasen unterteilen (Abbildung 3.4):

- Modelle auflösen: ausgehend von der zu vergleichenden Datei alle andere Fragmente aus dem logischen Modell finden, die für den Vergleich notwendig sind
- Elemente abstimmen: Über die geladenen logischen Modelle iterieren und die zueinander korrespondierenden Elemente finden
- Differenzen finden: Differenzen zwischen den korrespondierenden Elementen ausrechnen
- Äquivalenzen finden: Differenzen, die eine Änderung repräsentieren, zusammenbinden
- Abhängigkeiten ermitteln: Abhängigkeiten zwischen Differenzen finden und die abhängigen Differenzen zusammenbinden

- Konflikte detektieren: Kollidierende Änderungen finden

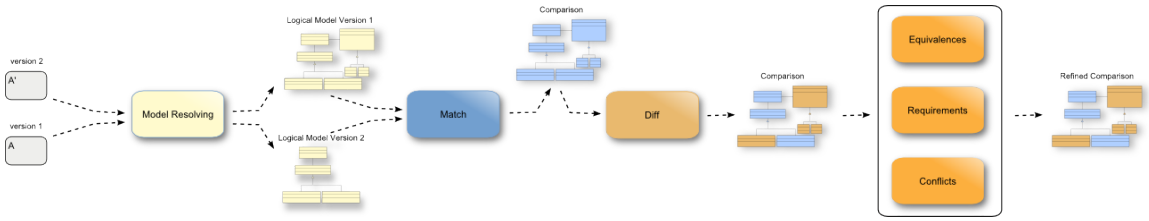

Abbildung 3.4: EMF-Compare-Vergleichsprozess. Quelle: [9]

Da JaMoPP-Modelle [3.4] auch EMF-Modelle sind, kann EMF-Compare-Framework auch zum Vergleichen von JaMoPP-Modellen benutzt werden. EMF-Compare bietet viele Einstellungen an, die die Definition der Gleichheit von Modellen steuern. Zum Beispiel, Änderungen in der Reihenfolge von Elementen oder ihre Namen können ignoriert werden.

3.7.1 Equality Helper

EqualityHelper [10] ist eine Hilfsklasse von EMF-Compare-Framework. Mit EqualityHelper lassen sich EMF-Modelle in Bezug auf ihre Struktur vergleichen. EqualityHelper bietet zwar keine große Anzahl von Einstellungenan, dafür lassen sich aber die Modelle schneller vergleichen. In manchen Situationen ist deshalb Nutzung von EqualityHelper von Vorteil, wo kein sehr tiefer Vergleich von Modellen nötig ist.

3.8 Software Model Extractor (SoMoX)

SoMoX [26] ist ein Reverse-Engineering-Ansatz [34], mit dem eine komponentenbasierte Software-Architektur von dem Quellcode extrahiert werden kann. SoMoX erkennt Komponenten (auch zusammengesetzte), die darin enthaltenen Klassen, Interfaces und Konnektoren. Zum Extrahieren einer Sofrware-Architektur können unterschiedliche Quellcode-Metriken eingestellt werden. Als Eingabe unterstützt SoMoX unter anderem JaMoPP-Modelle [3.4] und als Ausgabe die PCM-Modelle [3.5]. Darüber hinaus erstellt SoMoX ein Source-Decorator-Modell, wo alle Korrespondenzen zwischen den JaMoPP- und PCM-Elementen enthalten sind. Basierend auf diesen Korrespondenzen und den erstellten PCM-Modellen lassen sich existierende Projekte in Vitruvius [3.6] integrieren.

3.9 JGit

Git [12] ist Versionsverwaltungssystem, das zur Erfassung von Änderungen in Dokumenten oder Dateien verwendet wird [35]. Das JGit-Plugin [15] ist eine Java-Bibliothek, die das Git-Versionsverwaltungssystem in Java implementiert. Mit dieser Bibliothek kann man unter anderem ein Repository erstellen oder importieren, Commits erzeugen und auslesen und auch Git-Befehle ausführen.

3.10 Automated Coevolution of Source Code and Software Architecture Models

Langhammer hat in seiner Dissertation [17] einen Ansatz vorgeschlagen, wie der Quellcode mit komponentenbasierten Architekturmodellen konsistent gehalten werden kann. Als Quellcode hat er Java-Quellcode und als Architekturmodelle die Palladio-Performance-Modelle (PCM) [25] benutzt. Eine Konsistenerzhaltung funktioniert in seinem Ansatz bidirektional. Das bedeutet, Änderungen in dem Quellcode verursachen eine Anpassung von den korrespondierenden Architekturmodellen und auch umgekehrt. Dafür hat Langhammer änderungsgetriebene Abbildungsregeln definiert und einen Beobachter-Mechanismus erstellt, der Änderungen detektiert. Wenn Änderungen detektiert wurden, werden die entsprechenden Elemente anhand von den definierten Abbildungsregeln (sogenannte Konsistenzerhaltungsregeln) angepasst. Sein Ansatz ist im Kontext von Vitruvius [3.6] entstanden. Eine Limitierung von Vitruvius ist, dass eine Konsistenzerhaltung von Modellen nur für Projekte mit einer bestimmten Struktur gewährleistet ist. Ein Beispiel wäre, dass jedes Package '*datatypes*'- und '*contracts*'-Subpackages enthalten muss. Wenn ein Projekt von Anfang an in Vitruvius entwickelt wird, sorgt Vitruvius für die richtige Projektstruktur. Wenn aber ein existierendes Projekt mit einer beliebigen Struktur in Vitruvius integriert werden muss, gelten die definierten Konsistenzerhaltungsregeln möglicherweise nicht mehr. Aus diesem Grund werden die beiden Fälle separat in der Dissertation behandelt. Langhammer hat unterschiedliche Abbildungsregeln für die Vitruvius-Projekte (Projekte mit der von Vitruvius erwarteten Architektur) und für die integrierten Projekte (Projekte mir beliebiger Architektur) definiert.

Für seine Evaluierung der integrierten Projekte hat er mehrere Open-Source-Projekte gewählt. Er hat das ChangeReplay-Tool [3] benutzt, um Änderungen aus Commits zu extrahieren und sie auf die Code-Modelle anzuwenden. Die Ergebnisse der Evaluierung in der Abbildung 3.5 zeigen, dass für 92.1% Änderungen die definierten Reaktionen korrekt ausgeführt wurden .

Für unseren Zweck benutzen wir die von Langhammer implementierten Funktionalitäten. Wir implementieren einen anderen Ansatz für die Extrahierung und Anwendung von Änderungen aus den in [4.1] genannten Gründen. Außerdem passen wir die von Langhammer definierten Abbildungsregeln an und implementieren neue.

3.11 Continuous Integration of Performance Model (CIPM)

Mazkatli und Koziolek haben in [20] ein Konzept für eine kontinuierliche Integration von Performance-Modellen in einem agilen Software-Entwicklungsprozess vorgestellt und es in [21] weiterentwickelt. Ihr Ansatz ermöglicht, basierend auf Code-Änderungen, das Performance-Modell inkrementell zu aktualisieren statt ein neues Performance-Modell für das ganze System zu erstellen. Das Konzept von Mazkatli und Koziolek erweitert den Ansatz von Langhammer [17] und aktualisiert das Performance-Modell unter Berücksichtigung von Performance-Modellparametern. Als Performance-Modellparameter zählen zum Beispiel Anzahl der Ausführungen von einer Schleife, Sprungvorhersage oder Ressourcenbedarf. Für die Einschätzung von Performance-Modellparametern werden

nur geänderte Teile des Quellcodes betrachtet. Der CIPM-Prozess innerhalb eines agilen Software-Entwicklungsprozesses ist in der Abbildung 3.6 dargestellt. Der CIPM-Ansatz besteht aus vier Hauptaktivitäten:

- Aktualisierung des Performance-Modell und adaptive Instrumentierung: CIPM analysiert Code-Änderungen, aktualisiert das Performance-Modell und statisches Verhaltensmodell (Service Effect Specification 3.5) und instrumentiert die geänderten Teile vom Code. Eine Instrumentierung bedeutet, man fügt einen zusätzlichen Code an bestimmten Stellen in dem Quellcode hinzu, um Performance einzuschätzen. Man kann zum Beispiel eine Stoppuhr erstellen und die Ausführungsdauer von einem bestimmten Teil des Quellcodes messen.
- Monitoring: CIPM sammelt nötige Messungen während einer Test- oder Betriebsphase.
- Inkrementelle Kalibrierung von dem Performance-Modell: CIPM führt eine Dev-time- und Ops-time-Kalibrierung des Performance-Modells aus.
- Selbstvalidierung: CIPM startet eine Simulation und rechnet Differenzen zwischen den Simulationsergebnissen und den Monitoring-Daten aus, um den Schätzfehler zu ermitteln.

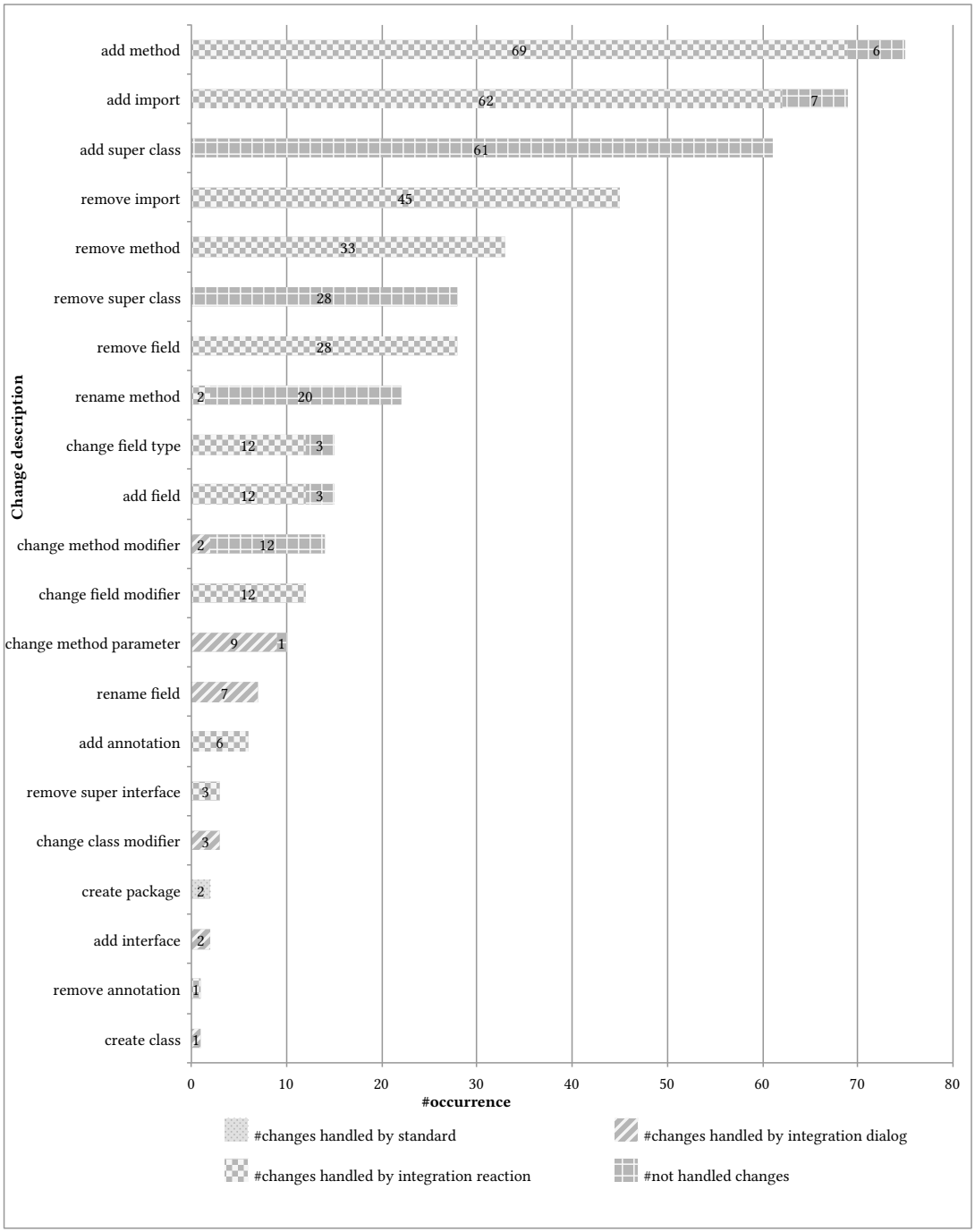

Abbildung 3.5: Evaluierungsergebnisse für ’Integrationsniveau 3’. Das Integrationsniveau 3 ermöglicht für bestimmte Änderungstypen eine automatische Anpassung von Performance-Modellen für integrierte Projekte durch definierte Integrationsreaktionen. Quelle: [17], Seite 188

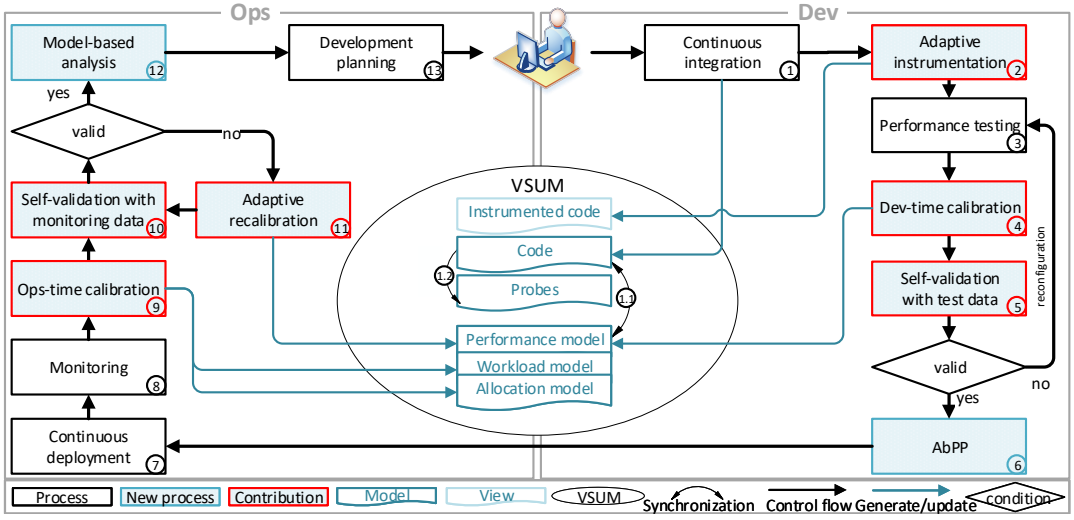

Abbildung 3.6: CIPM-Ansatz in einem agilen DevOps-Entwicklungsprozess, Quelle: [21]

4 Verwandte Arbeiten

4.1 Extending an Architecture and Code Co-Evolution Approach to Support Existing Software Projects

Petersen hat in seiner Masterarbeit [24] den Co-Evolution-Ansatz [18], [19] erweitert. Der Co-Evolution-Ansatz wird durch Vitruvius [27] realisiert. Der Kernpunkt des Ansatzes von Petersen war eine Integration von bereits existierenden Projekten in den Co-Evolution-Ansatz. Das Problem bei so einer Integration war, dass das zu integrierende Projekt, bestehend aus einem Code- und einem Architekturmodell, eine bestimmte Struktur haben muss, um in den Co-Evolution-Ansatz eingebunden werden zu können. Falls es nicht der Fall ist, funktioniert der Co-Evolution-Ansatz für dieses Projekt nicht richtig, weil keine (korrekten) Korrespondenzen zwischen den Code- und Architekturkomponenten aufgebaut werden können, was für eine Konsistenzerhaltung zwischen den Modellen notwendig ist. Petersen hat eine Lösung des Problems vorgeschlagen. Während der Integration werden alle integrierten Teile des Projekts als 'integriert' markiert. Ein Entwickler kann dann an dem Projekt arbeiten. Wenn er neue Teile zu dem Projekt hinzufügt, kommt der Standardmechanismus von Vitruvius zum Einsatz, um die neu erstellten Modelle konsistent zu halten. Wenn aber der Entwickler die als 'integriert' markierten Teile des Projekts ändert oder erweitert, wird der von Petersen implementierte Mechanismus aktiviert. Dieser Mechanismus benachrichtigt dann den Entwickler, welche Teile der Architektur manuell angepasst werden müssen. Anpassungen für bestimmte Typen von Änderungen werden in dem Architekturmodell sogar automatisch vorgenommen und müssen somit nicht manuell durchgeführt werden. Der beschriebene Vorgang ist in der Abbildung 4.1 dargestellt. Dieser Mechanismus wurde später von Langhammer in [17] benutzt.

Um die Ergebnisse seiner Masterarbeit zu evaluieren, musste Petersen unter anderem inkrementelle Code-Änderungen simulieren. Dafür hat er ein Tool namens ChangeReplay [3] implementiert. Das Tool basiert auf JGit [15] und GumTree [11]. Mit dem Tool kann man ein Commit auslesen, Code-Änderungen extrahieren und in kleine Schritte zerlegen. Als Ergebnis bekommt man eine Liste von Strings. Das erste Listenelement repräsentiert den Code vor dem Commit und das letzte den Code nach dem Commit. Alle Elemente dazwischen repräsentieren den Code in einem Zwischenzustand nach der Anwendung einer atomaren Änderung. Welche Änderung als atomar zählt, wird durch GumTree definiert.

Für unseren Zweck brauchen wir keine Zerlegung in atomare Änderungen vor der Anwendung auf die Code-Modelle. Nachdem die Code-Modelle geändert wurden, werden sie neu geparst und der Java-Monitor [3.6.1], der in Vitruvius zum Einsatz kommt, gibt Informationen über alle atomaren Änderungen wieder. Diese Informationen benutzen wir, um Performance-Modelle zu aktualisieren.

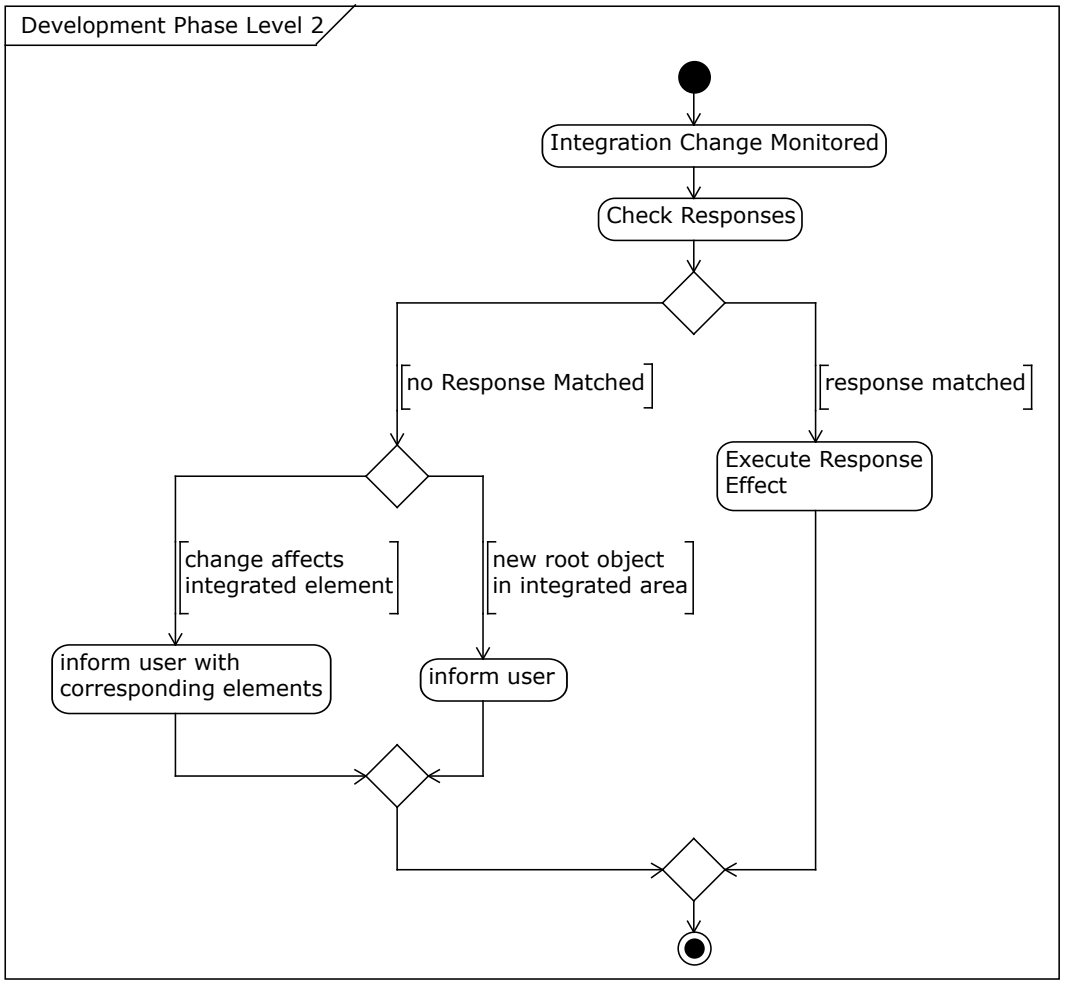

Abbildung 4.1: Reaktion auf Änderungen in dem Quellcode. Quelle: [24], Seite 34

4.2 GumTree

GumTree [11] ist ein Framework, mit dem man den Quellcode in eine Baumstruktur übersetzen und Differenzen zwischen zwei Bäumen ausrechnen kann. Ein Beispiel für einen Strukturvergleich zwischen zwei Methoden ist in der Abbildung 4.2 dargestellt. Im Vergleich zu einer rein textuellen Repräsentation des Quellcodes hat eine Baumstruktur den Vorteil, dass man Zusammenhänge zwischen den einzelnen Teilen vom Code leicht nachvollziehen kann. Wenn zum Beispiel eine neue Variable definiert wurde, kann man leicht ermitteln, ob sie innerhalb einer Schleife liegt und zu welcher Methode sie gehört. Eine Baumstruktur erleichtert auch den Vergleich zwischen zwei Quellcode-Bäumen. GumTree unterstützt mehrere Programmiersprachen, darunter auch Java. GumTree wird intern von dem ChangeReplay-Tool [3] benutzt.

Wie schon in 4.1 erklärt brauchen wir nicht die Modelle extra zu vergleichen, weil wir nötige Informationen über atomare Änderungen von dem Java-Monitor bekommen. Die

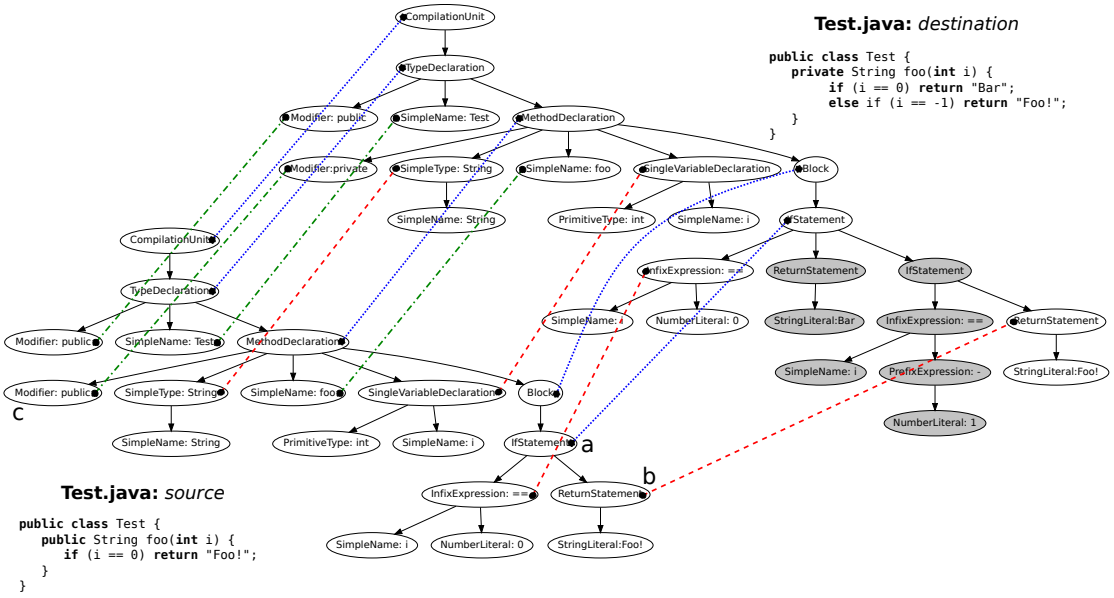

Abbildung 4.2: Strukturvergleich zwischen zwei Methoden mit GumTree. Gepunktete Linien bezeichnen die Top-Down-Vergleichsphase. Geschtrichelte Linien und Strichpunktlinien bezeichnen die Bottom-Up-Vergleichsphase. Die grauen Knoten sind Differenzen zwischen zwei Bäumen. Quelle: [11], Seite 4

Informationen werden dann genutzt, um die Code- und anschließend die Performance-Modelle anzupassen.

4.3 Arbeiten für den CIPM-Ansatz

Mehrere Teile des CIPM-Ansatzes wurden bereits in den anderen Bachelor- und Masterarbeiten implementiert. Sie beschreiben weitere Schritte des CIPM-Ansatzes und setzen deshalb voraus, dass der erste Schritte implementiert sein muss. Wir geben einen kurzen Überblick über einige von diesen Arbeiten.

Aufgrund von fehlender Implementierung des ersten Schrittes in dem CIPM-Ansatz (Aktualisierung des Code-Modells anhand von den aus Git-Commits extrahierten Änderungen) wurden in den unten beschriebenen Arbeiten Änderungen nicht aus Commits gelesen, sondern manuell simuliert. Im Rahmen dieser Bachelorarbeit befassen wir uns mit dem ersten Schritt für den CIPM-Ansatz und zwar mit der Extrahierung von Code-Änderungen aus Commits und ihre Anwendung auf das Code-Modell. Unser Beitrag vervollständigt den CIPM-Ansatz, sodass er in den Prozess der kontinuierlichen Integration und somit auch in einen agilen Software-Entwicklungsprozess eingebunden werden kann.

4.3.1 Adaptive Monitoring for Continuous Performance Model Integration

Dahmane hat in seiner Masterarbeit [4] eine Implementierung für ein adaptives Monitoring in dem CIPM-Ansatz vorgeschlagen. Sein Ansatz versorgt das Performance-Modell mit Monitoring-Informationen. Bevor Monitoring-Informationen gesammelt werden können, muss der Quellcode instrumentiert werden [31]. Dafür hat Dahmane das VSUM von Vitruvius [3.6] um ein Instrumentierungsmodell erweitert. Das Instrumentierungsmodell und das Quellcode-Modell werden von Vitruvius konsistent gehalten. Das Instrumentierungsmodell ermöglicht, basierend auf Änderungen im Quellcode, eine inkrementelle Instrumentierung des Quellcodes. In dem Instrumentierungsmodell werden Monitoring-Proben gespeichert. Eine Monitoring-Probe enthält den für die Instrumentierung nötigen Code und Informationen über die Stelle, wo dieser Code im Quellcode hinzugefügt werden muss. Während der Instrumentierung wird eine Kopie von dem Quellcode erstellt und die Monitoring-Proben in diese Kopie eingesetzt. Danach kann der Monitoring-Prozess gestartet werden. Monitoring-Informationen werden in einer Monitoring-Aufzeichnung gespeichert. Eine Monitoring-Aufzeichnung kann Informationen über Reaktionszeit, Anzahl der ausgeführten Schleifeniterationen, Wahrscheinlichkeit einer Sprungvorhersage oder Parameter eines Methodenaufrufes in einer externen Komponente enthalten. Wenn es genug Monitoring-Daten zu einer Monitoring-Probe gesammelt wurden, kann diese Monitoring-Probe deaktiviert werden. Anhand von den erstellten Monitoring-Aufzeichnungen kann anschließend das Performance-Modell aktualisiert werden.

4.3.2 Iterative Performance Model Parameter Estimation Considering Parametric Dependencies

Jägers hat einen Ansatz zur iterativen Abschätzung von Parametern eines Performance-Modells unter Berücksichtigung parametrischer Abhängigkeiten vorgeschlagen [13]. Als Parameter dienen Schleifeniterationen, Ressourcenanforderungen, Verzweigungsübergänge und Argumente für externe Service-Aufrufe. Sie sind ein Bestandteil des Palladio-Komponentenmodells [25]. Schätzungen von den Parametern basieren auf Überwachungsdaten. Abhängig von dem Parametertypen werden unterschiedliche Strategien zur Einschätzung benutzt. Für Verzweigungsübergänge werden Entscheidungsbäume verwendet und für Schleifeniterationen und Ressourcenanforderungen wird eine Regressionsanalyse durchgeführt. Beide Strategien erstellen Vorhersagemodelle für entsprechende Parameter. Die erzeugten Vorhersagemodelle werden anschließend in stochastische Ausdrücke umgewandelt, die das Performance-Modell anreichern.

4.3.3 Optimizing Parametric Dependencies for Incremental Performance Model Extraction

Voneva hat sich in ihrer Bachelorarbeit [30] mit Identifizierung und Optimierung von parametrischen Abhängigkeiten beschäftigt. Sie hat den Ansatz von Jägers [4.3.2] weiterentwickelt. Der Ansatz von Voneva identifiziert parametrische Abhängigkeiten für Aufrufe von externen Services. Für alle anderen Typen von Performance-Modell-Parametern werden parametrische Abhängigkeiten optimiert. Dafür werden zwei Algorithmen aus dem

Bereich 'maschinelles Lernen' sowie ein Genetic-Programming-Algorithmus benutzt. Der Ansatz erkennt auch komplexe parametrische Abhängigkeiten und berücksichtigt nicht nur Argumente von dem aktuellen Service-Aufruf, sondern auch die Ausgabewerte von den vorherigen Service-Aufrufen.

4.3.4 Enabling Consistency between Software Artefacts for Software Adaption and Evolution

Monschein beschreibt in seinem Vorschlag für eine Masterarbeit [28] ein Konzept, das mehrere Teile des CIPM-Ansatzes abdecken soll. Das sind Überwachung, Ableitung eines aktualisierten Architekturmodelles und Zuordnung zwischen den Design- und Laufzeitelementen. Außerdem soll eine Feedback-Schleife implementiert werden, die die Genauigkeit des aktualisierten Architekturmodelles regelmäßig überprüft und sie, falls nötig, anpasst. Monschein erweitert auch den Ansatz von Dahmane [4.3.1]. Aktuell ist die Masterarbeit von Monschein noch in Bearbeitung.

5 Konzeption

In diesem Kapitel stellen wir ein Konzept für unsere Implementierung vor. Zuerst beschreiben den Kontext und die Technologien für unsere Implementierung in 5.1. Dann fassen wir das Ziel unserer Implementierung in 5.2 zusammen. Anschließend erklären wir unseren Ansatz allgemein in 5.3.

5.1 Kontext und Technologien

Alle Funktionalitäten werden in der Programmiersprache Java (Version 12), in der Entwicklungsumgebung Eclipse Modeling Framework [7] und in dem Kontext des Vitruvius-Projektes [27] implementiert. Commits werden mithilfe von JGit-Plugin [15] behandelt. Code-Modelle werden in Vitruvius durch JaMoPP-Modelle [14] und Performance-Modelle durch PCM-Modelle [25] repräsentiert. JDT-Modelle sind auch Code-Modelle, die von dem Java-Parser erstellt werden. Im Vergleich zu den JaMoPP-Modellen sind sie aber keine EMF-Modelle [7]. Eine Aktualisierung von PCM-Modellen erfolgt nach den in der Reaction-Sprache [29] definierten Konsistenzerhaltungsregeln.

5.2 Ziel

Unsere Implementierung soll über die folgenden Funktionalitäten verfügen:

- Commits lesen und daraus Informationen über Code-Änderungen extrahieren
- Anhand von extrahierten Informationen die betroffenen Code-Modelle aktualisieren
- Konsistenzerhaltungsregeln bereitstellen, um eine Aktualisierung von Performance-Modellen zu ermöglichen

Dank diesen Funktionalitäten können Code- und Performance-Modelle in Vitruvius anhand von Commits angepasst werden.

5.3 Allgemeiner Ansatz

Unser Ansatz lässt sich in zwei Hauptteile unterteilen. Der erste Teil beschäftigt sich mit der Extrahierung von Änderungen aus einem Commit und Anwendung der extrahierten Änderungen auf die betroffenen JDT-Modelle. Der zweite Teil definiert die Konsistenzerhaltungsregeln. Basierend auf diesen Regeln passt Vitruvius die korrespondierenden PCM-Modelle an. In der Abbildung 5.1 ist der Ablauf einer Aktualisierung von Code- und PCM-Modellen dargestellt.

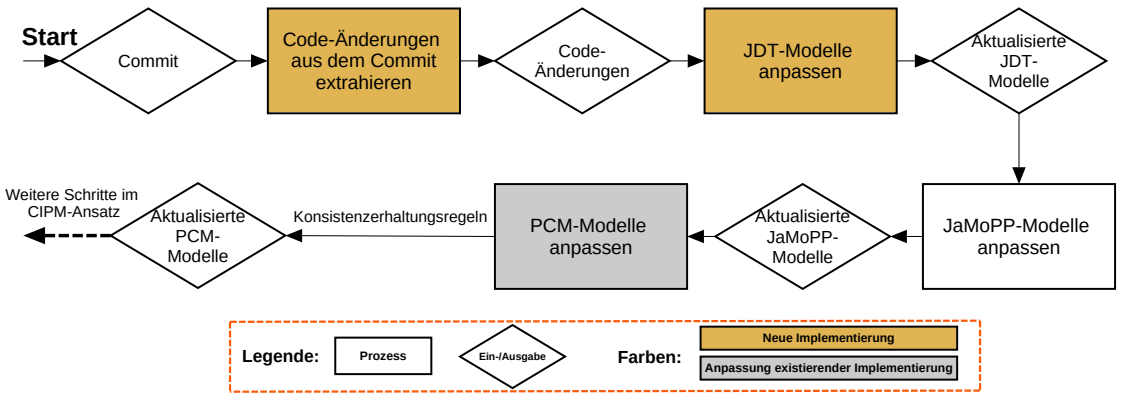

Abbildung 5.1: Ablauf einer Aktualisierung von Code- und PCM-Modellen

Wir nehmen an, das Projekt, dessen Modelle wir anhand von Commits aktualisieren werden, ist bereits in Vitruvius integriert. Das bedeutet, alle Code- und Performance-Modelle sowie ihre Korrespondenzen wurden bereits erstellt und befinden sich in Virtual Single Underlying Model (VSUM) [3.6]. In Vitruvius werden Code-Modelle durch JaMoPP-Modelle und Performance-Modelle durch PCM-Modelle repräsentiert. Die JaMoPP-Modelle hängen von den JDT-Modellen ab. Ein JDT-Modell ist ein Code-Modell und stellt den geparsten Java-Quellcode dar. Unser Ansatz verknüpft ein Git-Repository und erwartet dann ein Commit als Eingabe. Er extrahiert alle Änderungen aus diesem Commit und wendet die extrahierten Änderungen auf die JDT-Modelle an. Als Reaktion auf Änderungen in den JDT-Modellen passt Vitruvius dank dem Java-Monitor [3.6.1] die korrespondierenden JaMoPP-Modelle an. Als nächstes sucht Vitruvius nach den zu den Änderungen passenden Konsistenzerhaltungsregeln, um die korrespondierenden PCM-Modelle zu aktualisieren. Diese Konsistenzerhaltungsregeln definieren wir in unserem Ansatz. Die passenden Konsistenzerhaltungsregeln werden dann angewandt und die PCM-Modelle werden aktualisiert.

6 Implementierung

In diesem Kapitel stellen wir unsere Implementierung vor. Sie ist als Eclipse-Plugins realisiert und wir beschreiben die implementierten Plugins in 6.1.

6.1 Implementierte Plugins

Für unsere Implementierung haben wir ein neues Plugin namens *tools.vitruv.applications.pcmjava.integrationFromGit* erstellt. Die Tests für unsere Implementierung befinden sich in dem Plugin *tools.vitruv.applications.pcmjava.integrationFromGit.test*. Die Architektur der Plugins ist in den Abbildungen 6.1 und 6.2 dargestellt. Aus Gründen der Übersichtlichkeit zeigen die abgebildeten Klassendiagramme nicht alle implementierten Klassen, Methoden und Attribute.

6.1.1 Plugin tools.vitruv.applications.pcmjava.integrationFromGit

Die Klasse *GitChangeApplier* wendet extrahierte Änderungen auf die JDT-Modelle an. Die Hauptmethode *applyChangesFromCommit* nimmt als Parameter zwei Commits, zwischen denen Änderungen ausgerechnet werden müssen, sowie ein JDT-Modell von einem Java-Projekt. Die Methode ermittelt Änderungen, klassifiziert sie und ruft eine passende Routine auf. Zum Beispiel, für eine als 'ADD' klassifizierte Änderung wird die Methode *addElementToProject* aufgerufen. Nach einer erfolgreichen Ausführung sind die Änderungen auf das JDT-Modell von dem Java-Projekt angewandt. Änderungen in dem JDT-Modell lösen den JDT-Benachrichtigungsmechanismus [3.2] aus. Dieser Mechanismus schickt dann Benachrichtigungen in Form von Java-Deltas, die dann von dem Java-Monitor [3.6.1] abgefangen, verfeinert und in JaMoPP-Änderungen transformiert werden. Die JaMoPP-Änderungen werden anschließend an Vitruvius weitergeleitet. Vitruvius sucht dann nach den korrespondierenden PCM-Modellen und nach den zu den Änderungen passenden Konsistenzerhaltungsregeln. In Vitruvius sind Konsistenzerhaltungsregeln üblicherweise in einer Konsistenzerhaltungsspezifikation eingekapselt. Für unseren Ansatz haben wir eine Konsistenzerhaltungsspezifikation namens *GitIntegrationChangePropagationSpecification* definiert und dort Konsistenzerhaltungsregeln referenziert. Die Konsistenzerhaltungsregeln haben wir in der Reaction-Sprache [3.6.2] definiert. Sie befinden sich in den Dateien PackageAndClassifiers.reactions und ClassifierBody.reactions. Die meisten Konsistenzerhaltungsregeln haben wir aus den folgenden Projekten übernommen und angepasst:

- *tools.vitruv.applications.pcmjava.linkingintegration.change2command .internal.response.PackageMappingIntegration.reactions*

- *tools.vitruv.applications.pcmjava.pojotransformations.java2pcm*

Die Konsistenzerhaltungsregeln für Änderungen innerhalb der Methodenrümpfe haben wir unverändert gelassen. Sie stammen aus dem folgenden Projekt:

- *tools.vitruv.applications.pcmjava.seffstatements.pojotransformations .Java2PcmPackageMappingMethodBodyChangePreprocessor.xtend*

Außerdem haben wir auch neue Konsistenzerhaltungsregeln definiert, und zwar für das Löschen einer Klasse, einer Schnittstelle und eines Packages. Die Anpassungen in den bereits existierenden Konsistenzerhaltungsregeln waren notwendig, weil sie für unseren Ansatz nicht (korrekt) funktioniert haben. Ein Grund dafür war, dass die ursprünglichen Konsistenzerhaltungsregeln nicht für integrierte Projekte gedacht sind, sondern für die Projekte, die von Anfang an mit Vitruvius entwickelt wurden. Zum Beispiel, wenn ein neues Package erstellt wurde, wurden automatisch Subpackages '*datatypes*' und '*contracts*' erstellt. Das hat aber zu Inkonsistenzen in der Projektstruktur geführt, weil diese Subpackages in dem Git-Repository nicht exisiteren.

6.1.2 Plugin tools.vitruv.applications.pcmjava.integrationFromGit.test

Die Tests haben wir in zwei Packages unterteilt:

- *tools.vitruv.applications.pcmjava.integrationFromGit.test.integratedArea*
- *tools.vitruv.applications.pcmjava.integrationFromGit.test.nonIntegratedArea*

Die Tests in *tools.vitruv.applications.pcmjava.integrationFromGit.test.integratedArea* simulieren Änderungen auf den Teilen des Projekts, die bereits vor der Integration in Vitruv existiert haben und als ein integrierter Bereich zählen. Was als ein integrierter Bereich zählt ist projektspezifisch. In unserem Test-Projekt gelten alle Packages innerhalb des Source-Packages als integrierter Bereich. Mehr Details über integrierte und nicht integrierte Bereiche können in [17] und in [24] gefunden werden. Die Tests in *tools.vitruv.applications.pcmjava.integrationFromGit.test.nonIntegratedArea* simulieren Änderungen auf den Teilen des Projekts, die vor der Integration in Vitruv noch nicht existiert haben. Außerdem finden diese Änderungen in einem Bereich statt, der als ein nicht integrierter Bereich zählt. In unserem Test-Projekt zählt ein neues Package innerhalb des Source-Packages als ein nicht integrierter Bereich. Das neue Package darf nicht in die anderen schon existierenden Subpackages verschachtelt sein. Sonst würde es als ein integrierter Bereich erkannt.

In unseren Tests wollten wir zeigen, dass die von uns angepassten Konsistenzerhaltungsregeln sich gleich verhalten ungeachtet darauf, ob Änderungen in einem integrierten oder nicht integrierten Bereich stattgefunden haben. In der Zukunft sollte diese Unterscheidung komplett irrelevant sein, weil der CIPM-Ansatz [20] auf den Java-Projekten mit beliebigen Strukturen anwendbar sein soll.

In dem Package *tools.vitruv.applications.pcmjava.integrationFromGit.test.commits* sind die Namen von den Git-Branches und Commit-Hashes für alle Tests gespeichert.

In dem Ordner *tools.vitruv.applications.pcmjava.integrationFromGit.test/testProjects* ist unser Test-Projekt gespeichert.

In der Klasse *ApplyingChangesTestUtil.java* sind die Hilfsmethoden für alle Tests enthalten. Die Methoden sind statisch und können deshalb auch in anderen Projekten benutzt werden.

Jeder Test überprüft die Korrektheit unserer Implementierung für einen bestimmten Änderungstypen. Zum Beispiel der Test *IAChangeClassHeaderTest.java* simuliert Änderungen eines Klassenkopfes. Das Präfix 'IA' steht für einen integrierten Bereich und 'NIA' für einen nicht integrierten Bereich.

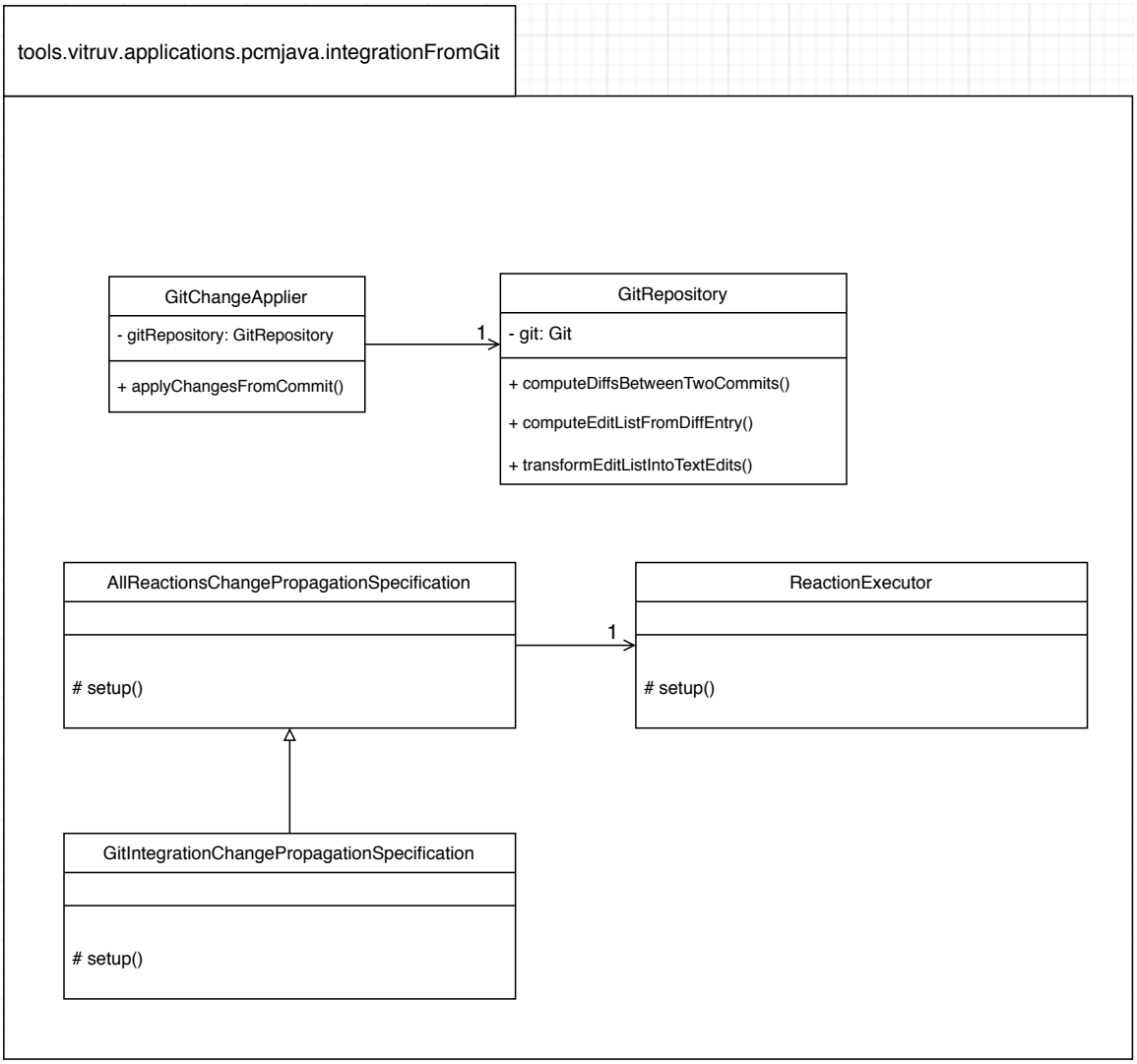

Abbildung 6.1: Klassendiagram für das Plugin *tools.vitruv.applications.pcmjava.integrationFromGit*. Das Klassendiagramm ist unvollständig und dient nur einer Darstellung der einzelnen Klassen, Methoden und Variablen.

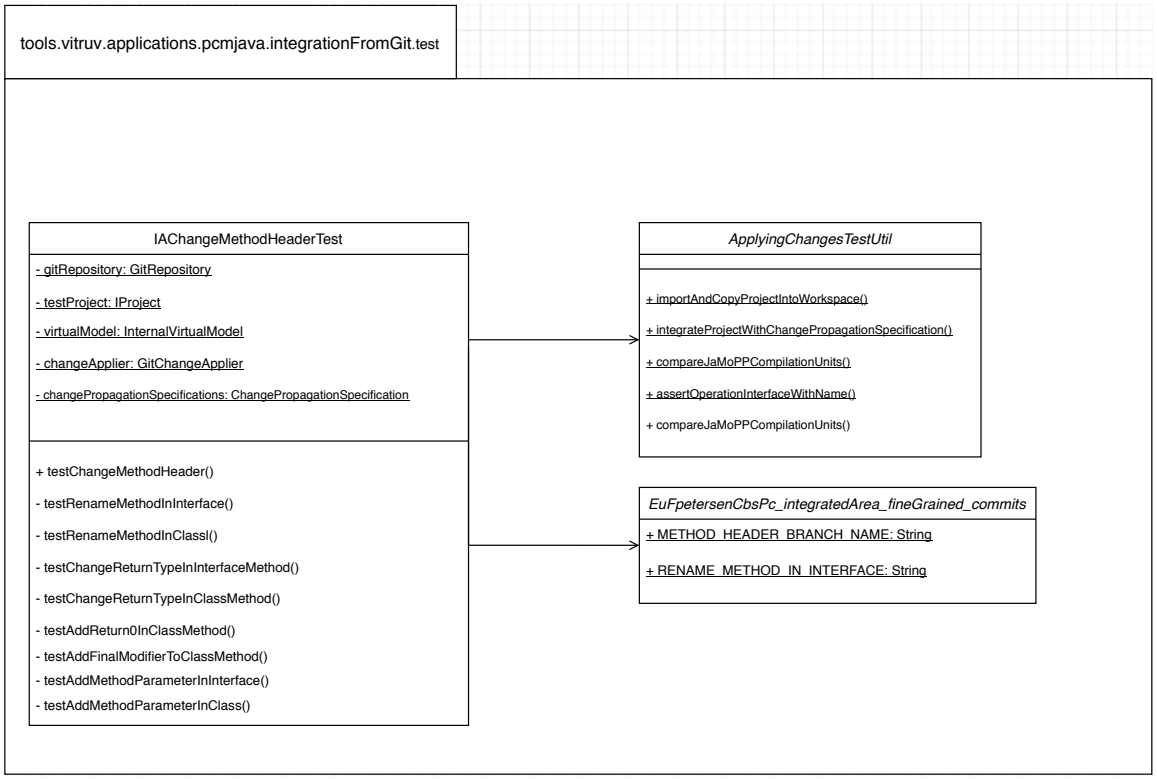

Abbildung 6.2: Klassendiagram für das Plugin *tools.vitruv.applications.pcmjava.integrationFromGit.test*. Das Klassendiagramm ist unvollständig und dient nur einer Darstellung der einzelnen Klassen, Methoden und Variablen.

7 Evaluierung

In diesem Kapitel beschreiben wir den Evaluierungsprozess für unsere Implementierung. Wir begründen unsere Wahl eines Testprojektes in 7.1. Einen allgemeinen Evaluierungsansatz erklären wir in 7.2. Eine konkrete Implementierung unseres Evaluierungsansatzes in Form von Tests beschreiben wir in 7.3. Die Evaluierungsergebnisse diskutieren wir in 7.4.

7.1 Projektauswahl

Für unsere Evaluierung mussten wir ein existierendes Java-Project in Vitruvius integrieren. Für dieses Projekt soll außerdem ein Git-Repository mit Commits vorhanden sein. Ursprünglich haben wir geplant, ein Open-Source-Projekt zu benutzen. Als Kandidaten haben wir drei Projekte ausgewählt: *Apache Any23*[1], *Apache Gora*[2] und *mRUBiS*[3]. Leider konnten wir keins von diesen Projekten in Vitruvius integrieren. Der Grund liegt in der JaMoPP-Bibliothek[14]. Sie wird seit einigen Jahren nicht mehr aktualisiert und ist nicht mehr komplett kompatibel mit der aktuellen Vitruvius-Version. Kleine und gezielte Anpassungen in der JaMoPP-Bibliothek haben auch nicht zum Erfolg geführt. Deshalb haben wir das Projekt namens '*eu.fpetersen.cbs.pc*' [4] benutzt. Das ist ein kleines Projekt, das innerhalb der Master-Arbeit [24] von Frederik Petersen entstanden ist. Für dieses Projekt haben wir ein Git-Repository initialisiert und manuell Commits erstellt. Die Commits umfassen unterschiedliche Arten von feingranularen Änderungen.

7.2 Evaluierungsansatz

In unserer Evaluierung wollten wir für unterschiedliche Arten von feingranularen Änderungen die Korrektheit der angepassten Code- und Performance-Modelle überprüfen. Wir haben ein existierendes Java-Projekt in Vitruvius integriert. Aus dem zu diesem Projekt gehörenden Git-Repository haben wir Commits gelesen, Änderungen aus den Commits extrahiert und auf die Code-Modelle angewandt. Anschließend haben wir die angepassten Code-Modelle mit den Referenz-Code-Modellen verglichen. Die Referenz-Code-Modelle haben wir erzielt, indem wir den Git-Checkout-Befehl an einem bestimmten Commit ausgeführt und neue Code-Modelle von dem Java-Projekt in dem neuen Zustand erstellt haben. Um es leichter zu verstehen, kann man sich das folgende Beispiel vorstellen: Angenommen, das in Vitruvius integrierte Projekt befindet sich in dem Zustand 'A', der

[1] Apache Any23, Core-Subprojekt, Version 0.9
[2] Apache Gora, Core-Subprojekt, Version 0.6
[3] mRUBiS
[4] eu.fpetersen.cbs.pc

dem Commit 'A' entspricht. Dann bekommen wir ein weiteres Commit 'B', lesen alle Änderungen aus dem Commit 'B' und wenden sie auf das integrierte Projekt an. Die Code-Modelle sind jetzt aktualisiert worden und sollten sich in dem Zustand 'B' befinden, der dem Commit 'B' entspricht. Die Referenz-Code-Modelle erstellen wir aber ohne Vitruvius und direkt von dem Projekt in dem Zustand 'B'. Falls die Aktualisierung der Code-Modelle korrekt stattgefunden hat, müssen die aktualisierten Code-Modelle und die erstellten Referenz-Code-Modelle gleich sein. Als letztes haben wir auch die aktualisierten Performance-Modelle überprüft. Anders als bei den Code-Modellen haben wir keine Referenz-Performance-Modelle erstellt. Das war unmöglich wegen technischer Probleme in SoMoX [26]. Stattdessen haben wir die aktualisierten Teile des Performance-Modells manuell untersucht. Zum Beispiel, falls eine Interface-Methode umbenannt wurde, haben wir sichergestellt, dass ein Performance-Modell für die Methode mit dem neuen Namen erstellt und für die Methode mit dem alten Namen gelöscht wurde.

7.3 Tests

Unsere Evaluierung haben wir als JUnit-Tests implementiert. Alle Tests haben eine ähnliche Struktur und unterscheiden sich abhängig davon:

- welches Java-Element von der Änderung betroffen ist (zum Beispiel Klassenkopf, Klassenvariable, Methodenkopf, Methodenrumpf etc.)
- ob das zu ändernde Element in einem integrierten oder nicht integrierten Bereich liegt

Was als ein integrierter Bereich zählt ist projektspezifisch. In unserem Test-Projekt gelten alle Packages innerhalb des Source-Packages als integrierte Bereiche. Mehr Details über integrierte und nicht integrierte Bereiche können in [17] und in [24] gefunden werden. In unserer Implementierung haben wir aber die Konsistenzerhaltungsregeln so angepasst, dass sie sich für integrierte und nicht integrierte Bereiche gleich verhalten. Deshalb ist eine Unterscheidung von integrierten und nicht integrierten Bereichen nicht notwendig. Eine separate Implementierung von Tests für integrierte und nicht integrierte Bereiche soll das gleiche Verhalten von Konsistenzerhaltungsregeln beweisen.

Listing 7.1 zeigt einen Test für das Umbenennen einer Methode in einem Interface, das in einem integrierten Bereich liegt. Zuerst werden alle Änderungen aus einem Commit gelesen und auf das in Vitruvius integrierte Projekt angewandt (Zeile 3). Das geänderte JDT-Modell wird in *compUnitChange* gespeichert (Zeile 12). Dank Vitruvius werden dann die korrespondierenden JaMoPP- und PCM-Modelle in VSUM aktualisiert. Dann wird der Git-Checkout-Befehl an dem gleichen Commit ausgeführt (Zeile 6). Dadurch übergeht das Projekt in den neuen Zustand. Ein temporäres JDT-Modell von dem Projekt und anschließend von dem Interface (dient als Referenzmodell) werden erstellt (Zeilen 8, 11). Anschließend werden die JaMoPP-Modelle für die beiden Interfaces (*compUnitChanged* und *compUnitFromGit*) verglichen (Zeile 14). Innerhalb der Methode *compareJaMoPPCompilationUnits* wird das JaMoPP-Modell für *compUnitFromGit* neu erstellt während das JaMoPP-Modell für *compUnitChanged* aus VSUM genommen. Dann wird es sichergestellt, dass für die umbenannte Methode ein Performance-Modell existiert (Zeile 16). Die Existenz einer Korrespondenz zwischen dem JaMoPP- und PCM-Modell wird implizit in der Methode *assertInterfaceMethodWithName* kontrolliert. Als letztes muss die Abwesenheit

des Performance-Modells für die Methode mit dem alten Namen überprüft werden (Zeile 18).

Listing 7.1: Test für das Umbenennen einer Methode in einem Interface, das in einem integrierten Bereich liegt

```
1  private void testRenameMethodInInterface() {
       //Apply changes
3      changeApplier.applyChangesFromCommit(commits.get(EuFpetersenCbsPc_integratedArea_fineGrained_commits.INIT),
           commits.get(EuFpetersenCbsPc_integratedArea_fineGrained_commits.RENAME_METHOD_IN_INTERFACE), testProject);
5      //Checkout the repository on the certain commit
       gitRepository.checkoutFromCommitId(EuFpetersenCbsPc_integratedArea_fineGrained_commits.RENAME_METHOD_IN_INTERFACE);
7      //Create temporary model from project from git repository. It does NOT add the created project to the workspace.
       projectFromGitRepository = ApplyingChangesTestUtil.createIProject(workspace,
9          workspace.getRoot().getLocation().toString() + "/clonedGitRepositories/" + testProjectName + ".withGit");
       //Get the changed compilation unit and the compilation unit from git repository to compare
11     ICompilationUnit compUnitFromGit = CompilationUnitManipulatorHelper
           .findICompilationUnitWithClassName("IDisplay.java", projectFromGitRepository);
13     ICompilationUnit compUnitChanged = CompilationUnitManipulatorHelper
           .findICompilationUnitWithClassName("IDisplay.java", testProject);
15     //Compare JaMoPP-Models
       boolean jamoppClassifiersAreEqual = ApplyingChangesTestUtil
17         .compareJaMoPPCompilationUnits(compUnitChanged, compUnitFromGit, virtualModel);
       //Ensure that there is a corresponding PCM model to the compUnitChanged.
19     boolean pcmExists = ApplyingChangesTestUtil.assertInterfaceMethodWithName("drawFrameRenamed",
           compUnitChanged, virtualModel);
21     //Ensure that there is a corresponding PCM model to the compUnitChanged.
       boolean noPcmExists = ApplyingChangesTestUtil.assertNoInterfaceMethodWithName("drawFrame",
23         compUnitChanged, virtualModel);

25     assertTrue("In testRenameMethodInInterface() the JaMoPP-models are NOT equal, but they should be",
           jamoppClassifiersAreEqual);
27     assertTrue("In testRenameMethodInInterface() corresponding PCM model does not exist, but it should exist",
           pcmExists);
29     assertTrue("In testRenameMethodInInterface() corresponding PCM model exists, but it should not exist",
           noPcmExists);
31 }
```

7.4 Ergebnisse

Die Ergebnisse unserer Evaluierung sind in der Tabelle 7.1 abgebildet. Die erste Spalte enthält die Nummern von Tests und dient lediglich der Bequemlichkeit. Die zweite Spalte enthält die Namen von Tests. Die dritte Spalte enthält die Namen von Klassen, wo die entsprechenden Tests implementiert wurden. Mit der Methode *print* aus der JaMoPP-Bibliothek lassen sich die JaMoPP-Modelle als *OutputStream*-Objekte ausgeben. Wir haben die Methode *print* an den geänderten JaMoPP-Modellen und an den JaMoPP-Referenzmodellen ausgeführt, die erhaltenen *OutputStream*-Objekte in *String*-Objekte konvertiert und sie dann zeichenbasiert (mit der Methode *compare*) miteinander verglichen. Die Ergebnisse von den Vergleichen sind in der Spalte 'JaMoPP-OutputStream' aufgeführt. Die JaMoPP-Modelle haben wir mithilfe von EMF Compare [8] verglichen. Falls ein Vergleich mit EMF Compare ein negatives Ergebnis zurückgegeben hat, haben wir dann die JaMoPP-Modelle mit EMF-EqualityHelper [5] verglichen. Die Ergebnisse dafür sind in der Spalte 'JaMoPP-Modelle'. Die Korrektheit der korrespondierenden PCM-Modelle wurde ebenso überprüft und die Ergebnisse dafür sind in der Spalte 'PCM-Modelle' enthalten. Manche

[5] Dokumentation für EqualityHelper

Tests konnten wir nicht ausführen oder sie haben ein falsches Ergebnis zurückgegeben. Über die entdeckten Probleme berichten wir in der Spalte 'Probleme'.

Insgesamt haben wir 88 Tests erstellt. 85 Tests wurden richtig ausgeführt. Die restlichen drei Tests (Test Nr. 6, 38 und 39) sind fehlgeschlagen. In den fehlgeschlagenen Tests wird eine Java-Datei gelöscht. Über Windows-System-Explorer haben wir sichergestellt, das sie von der Festplatte tatsächlich gelöscht wurden. Die Testausführung wird aber durch eine Ausnahme unterbrochen. Der Grund liegt in dem Plugin *tools.vitruv.domains.java.monitorededitor*. Die Methoden *ChangeResponder.visit(DeleteClassEvent)* und *ChangeResponder.visit(DeleteInterfaceEvent)* greifen auf die bereits gelöschten JDT-Modelle von den Java-Dateien. Wir vermuten, dass diese Methoden nicht korrekt funktionierten. Aus zeitlichen Gründen konnten wir aber diese Methoden nicht korrigieren. Somit wurden 96,6% der erstellten Tests richtig ausgeführt.

Test Nr.	Test	Testklasse	JaMoPP-Output-Stream	JaMoPP-Modell	PCM-Modell	Probleme
1	testAddClassAnnotation	IAChangeClassAnnotationTest	korrekt	korrekt	nicht betroffen	
2	testChangeClassAnnotation	IAChangeClassAnnotationTest	korrekt	korrekt	nicht betroffen	
3	testRemoveClassAnnotation	IAChangeClassAnnotationTest	korrekt	korrekt	nicht betroffen	
4	testAddAbstract	IAChangeClassHeaderTest	korrekt	korrekt	nicht betroffen	
5	testChangeAbstractToFinal	IAChangeClassHeaderTest	korrekt	korrekt	nicht betroffen	
6	testRenameClass	IAChangeClassHeaderTest	nicht überprüft	nicht überprüft	nicht überprüft	Vitruvius wirft eine Exception wenn eine compilation unit gelöscht ist. Umbennenen einer compilation unit ist Löschen der compilation unit mit dem alten Namen und Erstellen einer mit dem neuen Namen. tools.vitruv.domains.java.monitorededitor.ChangeResponder.visit(DeleteClassEvent)-Methode versucht auf die bereits gelöschte compilation unit zuzugreifen, was eine exception verursacht.
7	testAddExtends	IAChangeExtendsTest	korrekt	korrekt	keine CPR implementiert	
8	testRemoveExtends	IAChangeExtendsTest	korrekt	korrekt	keine CPR implementiert	
9	testAddField	IAChangeFieldTest	korrekt	korrekt	korrekt	
10	testRenameField	IAChangeFieldTest	korrekt	korrekt	korrekt	
11	testAddFieldModifier	IAChangeFieldTest	korrekt	korrekt	nicht betroffen	
12	testChangeFieldModifier	IAChangeFieldTest	korrekt	korrekt	nicht betroffen	
13	testChangeFieldType	IAChangeFieldTest	korrekt	korrekt	korrekt	
14	testRemoveField	IAChangeFieldTest	korrekt	korrekt	korrekt	
15	testAddImplements	IAChangeImplementsTest	korrekt	korrekt	korrekt	
16	testRemoveImplements	IAChangeImplementsTest	korrekt	korrekt	korrekt	
17	testRenameMethodInInterface	IAChangeMethodHeaderTest	korrekt	korrekt	korrekt	
18	testRenameMethodInClass	IAChangeMethodHeaderTest	korrekt	korrekt	korrekt	
19	testChangeReturnTypeInInterfaceMethod	IAChangeMethodHeaderTest	korrekt	korrekt	korrekt	
20	testChangeReturnTypeInClassMethod	IAChangeMethodHeaderTest	korrekt	korrekt	nicht betroffen	
21	testAddReturn0InClassMethod	IAChangeMethodHeaderTest	korrekt	korrekt	nicht betroffen	
22	testAddFinalModifierToClassMethod	IAChangeMethodHeaderTest	korrekt	korrekt	nicht betroffen	
23	testAddMethodParameterInInterface	IAChangeMethodHeaderTest	korrekt	korrekt	korrekt	
24	testAddMethodParameterInClass	IAChangeMethodHeaderTest	korrekt	korrekt	nicht betroffen	
25	testAddInternalAction	IAChangeMethodImplementationTest	korrekt	korrekt	korrekt	
26	testAddForLoop	IAChangeMethodImplementationTest	korrekt	korrekt	korrekt	
27	testAddIfElse	IAChangeMethodImplementationTest	korrekt	korrekt	korrekt	
28	testRemoveIfElse	IAChangeMethodImplementationTest	korrekt	korrekt	korrekt	
29	testRemoveForLoop	IAChangeMethodImplementationTest	korrekt	korrekt	korrekt	
30	testRemoveInternalAction	IAChangeMethodImplementationTest	korrekt	korrekt	korrekt	
31	testCreateFolderAndFile	IAChangeNonJavaFileTest	nicht betroffen	nicht betroffen	nicht betroffen	
32	testRenameFile	IAChangeNonJavaFileTest	nicht betroffen	nicht betroffen	nicht betroffen	

Test Nr.	Test	Testklasse	JaMoPP-Output-Stream	JaMoPP-Modell	PCM-Modell	Probleme
33	testChangeFileContent	IAChangeNonJavaFileTest	File-Inhalt korrekt	nicht betroffen	nicht betroffen	
34	testCopyFile	IAChangeNonJavaFileTest	File-Inhalt korrekt	nicht betroffen	nicht betroffen	
35	testRemoveFile	IAChangeNonJavaFileTest	nicht betroffen	nicht betroffen	nicht betroffen	
36	testCreateClass	IACreateDeleteCompilationUnitTest	korrekt	korrekt	korrekt	
37	testCreateInterface	IACreateDeleteCompilationUnitTest	korrekt	korrekt	korrekt	
38	testRemoveClass	IACreateDeleteCompilationUnitTest	nicht betroffen	nicht korrekt	nicht korrekt	Vitruvius wirft eine Exception wenn eine compilation unit gelöscht ist. Umbennenen einer compilation unit ist Löschen der compilation unit mit dem alten Namen und Erstellen einer mit dem neuen Namen. tools.vitruv.domains.java.monitored editor.ChangeResponder.visit(DeleteClassEvent)-Methode versucht auf die bereits gelöschte compilation unit zuzugreifen, was eine exception verursacht.
39	testRemoveInterface	IACreateDeleteCompilationUnitTest	nicht betroffen	nicht korrekt	nicht korrekt	Vitruvius wirft eine Exception wenn eine compilation unit gelöscht ist. Umbennenen einer compilation unit ist Löschen der compilation unit mit dem alten Namen und Erstellen einer mit dem neuen Namen. tools.vitruv.domains.java.monitored editor.ChangeResponder.visit(DeleteInterfaceEvent)-Methode versucht auf die bereits gelöschte compilation unit zuzugreifen, was eine exception verursacht.
40	testAddField	IACreateDeleteFieldTest	korrekt	korrekt	korrekt	
41	testRenameField	IACreateDeleteFieldTest	korrekt	korrekt	korrekt	
42	testRemoveCreatedField	IACreateDeleteFieldTest	korrekt	korrekt	korrekt	
43	testCreateMethodInInterface	IACreateDeleteMethodTest	korrekt	korrekt	korrekt	
44	testCreateMethodInClass	IACreateDeleteMethodTest	korrekt	korrekt	korrekt	
45	testRemoveMethodInClass	IACreateDeleteMethodTest	korrekt	korrekt	korrekt	
46	testRemoveMethodInInterface	IACreateDeleteMethodTest	korrekt	korrekt	korrekt	
47	testCreatePackage	IACreateDeletePackageTest	korrekt	korrekt	korrekt	
48	testRenameCreatedPackage	IACreateDeletePackageTest	korrekt	korrekt	korrekt	
49	testRemoveCreatedPackage	IACreateDeletePackageTest	korrekt	korrekt	korrekt	
50	testAddClassAnnotation	NIAChangeClassAnnotationTest	korrekt	korrekt	nicht betroffen	
51	testChangeClassAnnotation	NIAChangeClassAnnotationTest	korrekt	korrekt	nicht betroffen	
52	testRemoveClassAnnotation	NIAChangeClassAnnotationTest	korrekt	korrekt	nicht betroffen	
53	testRemovePublicClassModifier	NIAChangeClassHeaderTest	korrekt	korrekt	nicht betroffen	
54	testAddFinalClassModifier	NIAChangeClassHeaderTest	korrekt	korrekt	nicht betroffen	
55	testChangeFinalToAbstractClassModifier	NIAChangeClassHeaderTest	korrekt	korrekt	nicht betroffen	
56	testChangeAbstractToPublicClassModifier	NIAChangeClassHeaderTest	korrekt	korrekt	nicht betroffen	
57	testAddFirstImportForExtends	NIAChangeExtendsTest	korrekt	korrekt	nicht betroffen	
58	testAddExtends	NIAChangeExtendsTest	korrekt	korrekt	keine CPR implementiert	
59	testAddSecondImportForExtends	NIAChangeExtendsTest	korrekt	korrekt	nicht betroffen	
60	testChangeExtends	NIAChangeExtendsTest	korrekt	korrekt	keine CPR implementiert	
61	testRemoveExtends	NIAChangeExtendsTest	korrekt	korrekt	keine CPR implementiert	
62	testRemoveSecondImportForExtends	NIAChangeExtendsTest	korrekt	korrekt	nicht betroffen	
63	testRemoveFirstImportForExtends	NIAChangeExtendsTest	korrekt	korrekt	nicht betroffen	
64	testAddField	NIAChangeFieldTest	korrekt	korrekt	korrekt	
65	testRenameField	NIAChangeFieldTest	korrekt	korrekt	korrekt	
66	testAddFieldModifier	NIAChangeFieldTest	korrekt	korrekt	nicht betroffen	
67	testChangeFieldModifier	NIAChangeFieldTest	korrekt	korrekt	nicht betroffen	
68	testChangeFieldType	NIAChangeFieldTest	korrekt	korrekt	korrekt	
69	testRemoveField	NIAChangeFieldTest	korrekt	korrekt	korrekt	
70	testAddImplementsAndMethod	NIAChangeImplementsTest	korrekt	korrekt	korrekt	
71	testChangeImplementsAndAddMethod	NIAChangeImplementsTest	korrekt	korrekt	korrekt	
72	testRemoveImplements	NIAChangeImplementsTest	korrekt	korrekt	korrekt	
73	testRenameMethodInInterface	NIAChangeMethodHeaderTest	korrekt	korrekt	korrekt	

Test Nr.	Test	Testklasse	JaMoPP-Output-Stream	JaMoPP-Modell	PCM-Modell	Probleme
74	testRenameMethodInClass	NIAChangeMethodHeaderTest	korrekt	korrekt	korrekt	
75	testChangeReturnTypeInInterfaceMethod	NIAChangeMethodHeaderTest	korrekt	korrekt	korrekt	
76	testChangeReturnTypeInClassMethod	NIAChangeMethodHeaderTest	korrekt	korrekt	nicht betroffen	
77	testAddReturn0InClassMethod	NIAChangeMethodHeaderTest	korrekt	korrekt	nicht betroffen	
78	testAddFinalModifierToClassMethod	NIAChangeMethodHeaderTest	korrekt	korrekt	nicht betroffen	
79	testAddMethodParameterInInterface	NIAChangeMethodHeaderTest	korrekt	korrekt	korrekt	
80	testAddMethodParameterInClass	NIAChangeMethodHeaderTest	korrekt	korrekt	nicht betroffen	
81	testAddExternalCall	NIAChangeMethodImplementationTest	korrekt	korrekt	korrekt	
82	testAddInternalAction	NIAChangeMethodImplementationTest	korrekt	korrekt	korrekt	
83	testAddFor	NIAChangeMethodImplementationTest	korrekt	korrekt	korrekt	
84	testAddIfElse	NIAChangeMethodImplementationTest	korrekt	korrekt	korrekt	
85	testRemoveIfElse	NIAChangeMethodImplementationTest	korrekt	korrekt	korrekt	
86	testRemoveFor	NIAChangeMethodImplementationTest	korrekt	korrekt	korrekt	
87	testRemoveInternalAction	NIAChangeMethodImplementationTest	korrekt	korrekt	korrekt	
88	testRemoveExternalCall	NIAChangeMethodImplementationTest	korrekt	korrekt	korrekt	

Tabelle 7.1: Evaluationsergebnisse. Abkürzungen: IA - integrierter Bereich, NIA - nicht integrierter Bereich, CPR - Change-Propagation-Regeln.

8 Limitierungen und zukünftige Arbeit

In diesem Kapitel diskutieren wir Limitierungen unserer Implementierung und die Gründe für diese Limitierungen. Außerdem sprechen wir über die Stellen, an welchen unser Ansatz in einer zukünftigen Arbeit erweitert werden kann.

8.1 Tests mit anderen Projekten

Die JaMoPP-Bibliothek [14] wird seit einigen Jahren nicht mehr aktualisiert und ist deshalb nicht (komplett) kompatibel mit der aktuellen Verison von Vitruvius. Aus diesem Grund konnten wir kein Open-Source-Projekt mit bereits existierenden Commits in Vitruvius integrieren. Deshalb haben wir unsere Implementierung nur auf einem Projekt mit manuell erstellten Commits getestet. Sobald die Probleme in der JaMoPP-Bibliothek behoben sind, kann unsere Implementierung auf anderen Projekten getestet werden. Die Tests für zwei Open-Source-Projekte [1] [2] haben wir bereits erstellt.

Die Integration von den Open-Source-Projekten Apache Any23 und Apache Gora in Vitruvius nimmt sehr viel Zeit in Anspruch. Für Apache Any23 hat sie mehr als sieben Stunden und für Apache Gora mehr als anderthalb Stunden gedauert. Dabei haben wir den Integrationsfortschritt beobachtet. Abgesehen davon, dass die Integration von beiden Projekten am Ende fehlgeschlagen ist, haben wir festgestellt, dass die Integration einer Java-Datei bis zu zwei Minuten dauern kann [3]. In einer zukünftigen Arbeit sollte der Integrationsansatz optimiert werden.

8.2 Anpassung für die zustandsbasierte Propagierungsstrategie

Aktuell kann Vitruvius eine Konsistenzerhaltung für feingranulare Änderungen erstellen. Als wir unseren Ansatz für Commits mit grobgranularen Code-Änderungen getestet haben, hat es zu Fehlern und falschen Modellaktualisierungen in Vitruvius geführt. Eine zustandsbasierte Propagierungsstrategie wurde in Vitruvius zwar implementiert, aber noch nicht komplett integriert. Bei einer zustandsbasierten Propagierungsstrategie bekommt Vitruvius als Eingabe ein Modell in einem neuen Zustand und ermittelt feingranulare Änderungen zwischen dem neuen und dem alten Zustand. Die zustandsbasierte Propagierungsstrategie würde für den CIPM-Ansatz sehr gut passen, weil Commits keine Informationen

[1] Apache Any23, Core-Subprojekt, Version 0.9

[2] Apache Gora, Core-Subprojekt, Version 0.6

[3] Die Integration wurde auf einem Laptop ausgeführt mit den folgenden Spezifikationen: Prozessor Intel Core i7 9750H, 8 GB Arbeitsspeicher, NVMe SSD Festplatte

über feingranulare Änderungen enthalten. Sobald die zustandsbasierte Propagierungsstrategie in Vitruvius vollständig integriert ist, kann unser Ansatz für die Nutzung der zustandsbasierten Propagierungsstrategie angepasst werden.

9 Schlussfolgerungen

Im Rahmen dieser Bachelorarbeit haben wir einen Ansatz implementiert, der es ermöglicht, die Code- und Performance-Modelle anhand von den Informationen aus Commits anzupassen. Unser Ansatz verbindet ein Git-Repository mit den in Vitruvius gespeicherten Code- und Performance-Modellen, extrahiert Änderungen aus Commits und wendet sie auf die Code-Modelle an. Für eine automatische Anpassung von Performance-Modellen haben wir die existierenden Konsistezerhaltungsregeln angepasst. Außerdem haben wir einige neue Konsistezerhaltungsregeln erstellt. Unsere Implementierung haben wir auf einem Projekt evaluiert. Für dieses Projekt haben wir ein Git-Repository und Commits erzeugt, Änderungen aus den Commits gelesen und auf die Code- und Performance-Modelle angewandt.

Leider konnten wir aufgrund von technischen Problemen in den anderen Plugins nicht alle gesetzten Ziele erreichen. Darüber haben wir in dem Kapitel 8 berichtet. Wenn diese Probleme behoben sind, kann unsere Implementierung mit anderen größeren Projekten getestet werden. Außerdem kann unsere Implementierung für die zustandsbasierte Propagierungsstrategie angepasst werden, sobald diese Strategie in Vitruvius vollständig eingesetzt ist. Dadurch könnte der CIPM-Ansatz[21] in den Prozess der kontinuierlichen Integration und somit auch in einen agilen Software-Entwicklungsprozess eingebunden werden.

Literatur

[1] *Abstrakte Syntaxbäume*. URL: https://de.wikipedia.org/wiki/Syntaxbaum#Abstrakte_Syntaxb%C3%A4ume.

[2] Erik Burger. "Vorlesung Modellgetriebene Software-Entwicklung". Karlsruher Institut für Technologie, 2020.

[3] *Changereplay Tool*. URL: https://sdqweb.ipd.kit.edu/wiki/Changereplay.

[4] Noureddine Dahmane. "Adaptive Monitoring for Continuous Performance Model Integration". Magisterarb. Karlsruher Institut für Technologie, 2019.

[5] *Eclipse Foundation*. URL: https://www.eclipse.org/.

[6] *Eclipse Java development tools (JDT)*. URL: https://www.eclipse.org/jdt/.

[7] *Eclipse Modeling Framework (EMF)*. URL: https://www.eclipse.org/modeling/emf/.

[8] *EMF Compare*. URL: https://www.eclipse.org/emf/compare/.

[9] *EMF Compare Developer Guide*. URL: https://www.eclipse.org/emf/compare/documentation/latest/developer/developer-guide.html#Default_Behavior_and_Extensibility.

[10] *EqualityHelper Manual*. URL: https://download.eclipse.org/modeling/emf/emf/javadoc/2.6.0/org/eclipse/emf/ecore/util/EcoreUtil.EqualityHelper.html.

[11] Jean-Rémy Falleri u. a. "Fine-grained and accurate source code differencing". In: *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 2014, S. 313–324. DOI: 10.1145/2642937.2642982. URL: http://doi.acm.org/10.1145/2642937.2642982.

[12] *Git*. URL: https://git-scm.com/.

[13] Jan-Philipp Jägers. "Iterative Performance Model Parameter Estimation Considering Parametric Dependencies". Magisterarb. Karlsruher Institut für Technologie, 2018.

[14] *JaMoPP Java Model Parser and Printer*. URL: https://www.jamopp.org/.

[15] *JGit-Plugin*. URL: https://www.eclipse.org/jgit/.

[16] Max Kramer. "Specification Languages for Preserving Consistency between Models of Different Languages". Diss. Karlsruher Institut für Technologie, 2017.

[17] Michael Langhammer. "Automated Coevolution of Source Code and Software Architecture Models". Diss. Karlsruher Institut für Technologie, 2017.

[18] Michael Langhammer. "Co-evolution of Component-based Architecture-model and Object-oriented Source Code". In: (2013).

[19] Sven Leonhardt u. a. “Integration of Existing Software Artifacts into a View- and Change-Driven Development Approach”. In: (2015).

[20] Manar Mazkatli und Anne Koziolek. “Continuous Integration of Performance Model”. In: (2018).

[21] Manar Mazkatli u. a. “Incremental Calibration of Architectural Performance Models with Parametric Dependencies”. In: (2020).

[22] Dominik Messinger. “Incremental Code Architecture Consistency Support through Change Monitoring and Intent Clarification”. Magisterarb. Karlsruher Institut für Technologie, 2014.

[23] *Meta-Object Facility (MOF).* URL: https://en.wikipedia.org/wiki/Meta-Object_Facility.

[24] Frederik Petersen. “Extending an Architecture and Code Co-Evolution Approach to Support Existing Software Projects”. Magisterarb. Karlsruher Institut für Technologie, 2017.

[25] Ralf H. Reussner u. a. *Modelling and Simulating Software Architectures The Palladio Approach*. MIT Press, 2016.

[26] *SoMoX Software Model Extractor.* URL: https://sdqweb.ipd.kit.edu/wiki/SoMoX.

[27] *Vitruvius Project.* URL: https://github.com/vitruv-tools/Vitruv/wiki.

[28] *Vitruvius Project.* URL: https://svnserver.informatik.kit.edu/i43/svn/theses/Abschlussarbeiten/Proposals/MA_David_Monschein/.

[29] *Vitruvius Reaction Language.* URL: https://github.com/vitruv-tools/Vitruv/wiki/The-Reactions-Language.

[30] Sonya Voneva. “Optimizing Parametric Dependencies for Incremental Performance Model Extraction”. Magisterarb. Karlsruher Institut für Technologie, 2020.

[31] *Wikipedia Code-Instrumentierung.* URL: https://de.wikipedia.org/wiki/Instrumentierung_(Softwareentwicklung).

[32] *Wikipedia Kontinuierliche Integration.* URL: https://de.wikipedia.org/wiki/Kontinuierliche_Integration.

[33] *Wikipedia modellgetriebene Software-Entwicklung.* URL: https://de.wikipedia.org/wiki/Modellgetriebene_Softwareentwicklung.

[34] *Wikipedia Reverse Engineering.* URL: https://de.wikipedia.org/wiki/Reverse_Engineering.

[35] *Wikipedia Versionsverwaltung.* URL: https://de.wikipedia.org/wiki/Versionsverwaltung.