Ansatz einer entwicklungsprojektweiten Abhängigkeits-Konsistenz des Quellcodemodells zur Qualitätsverbesserung von Software-Entwicklungsprojekten

Zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEURS

an der KIT-Fakultät für

Elektrotechnik und Informationstechnik

des Karlsruher Instituts für Technologie (KIT)

genehmigte

DISSERTATION

von

Dipl.-Ing. Martin Eyl

geb. in: Dierdorf

Tag der mündlichen Prüfung: 22.05.2018

Hauptreferent: Prof. Dr.-Ing. Klaus D. Müller-Glaser

Korreferent: Prof. Dr. Ralf H. Reussner

Danksagung

Mein besonderer Dank geht an Herrn Prof. Dr.-Ing. Klaus D. Müller-Glaser für die Übernahme des Hauptreferats und die damit einhergehende Unterstützung, die vielen Gespräche und Anregungen, welche den erfolgreichen Abschluss der Arbeit ermöglichte.

Mein Dank gilt ebenso Herrn Prof. Dr. Ralf H. Reussner für sein Interesse und seine Bereitschaft, das Korreferat zu übernehmen.

Danken möchte ich Herrn Dr. Clemens Reichmann für das Ermöglichen dieser Arbeit, die Freiräume, die vielen Diskussionen, Ideen und Korrekturvorschläge.

Filderstadt, April 2018 Martin Eyl

Kurzfassung

Heutige Softwareentwicklungsprojekte müssen oft eine Vielzahl von Anforderungen mit hoher Komplexität in kurzen Release-Zyklen umsetzen. Daraus ergeben sich besondere Herausforderungen an Arbeitsteilung, Dokumentation, Prozesssicherheit und Qualität. Entwicklungsarbeiten müssen parallelisiert werden und Softwareentwickler müssen sich immer wieder in den Quellcode einarbeiten. Die Entwickler brauchen schnelle und präzise Rückmeldung über die Qualität ihrer durchgeführten Änderungen am Quellcode.

Über feingranulare Traceability Links in den Quellcode werden eine verbesserte Dokumentation und größere Prozesssicherheit ermöglicht. Dazu wird ein Metamodell für den Quellcode definiert und in ein Metamodell mit Anforderungsmanagement, Änderungsmanagement, Testdatenmanagement und Dokumentation eingebunden. Das gesamte Modell wird in einem Software Configuration Management (SCM) Repository abgespeichert, um die Versionierung aller Artefakte und Links zu ermöglichen. In einem Quellcode Editor können die Traceability Links erstellt und genutzt werden. Die Historie einzelner Quellcode Artefakte kann einschließlich der Traceability Links im Editor zur Anzeige gebracht werden.
Durch das Vorliegen des Quellcodes als Modell wird auch ein feingranulares pessimistisches Sperren einzelner Modell Artefakte ermöglicht. Damit ist das parallele Bearbeiten einer Klasse oder einer Methode möglich, ohne dass der Quellcode verschmolzen werden muss. Es werden durch die Sperren auch Syntaxfehler im SCM Repository verhindert. Im Quellcode Editor werden die Sperren anderer Entwickler angezeigt.
Continuous Integration wird dahingehend erweitert, dass durch Abspeichern von Class-Dateien im Repository ein schneller Produkt-Build und damit auch schnelleres Feedback für den Entwickler ermöglicht wird. Durch Testauswahlstrategien werden nur für den geänderten Quellcode relevante Tests ausgeführt. Eine Testauswahlstrategie verwendet dabei die Traceability Informationen zwischen geänderten Quellcode, Anforderung, Testspezifikation und dem Test-Quellcode. In großen Projekten entstehen auf Grund des Quellcodes sehr große Modelle, die eine Herausforderung bezüglich Speicherbedarfes und Performance darstellen. Es wurden Untersuchungen an Hand eines Projekts mit 6,5 Millionen Quellcode-Zeilen durchgeführt.

Für diese Konzepte wurde ein Prototyp auf Basis der Eclipse Entwicklungsumgebung und für Java entwickelt.

Abstract

Nowadays software development projects must develop a large number of requirements often with high complexity in short release cycles. The consequences are particular challenges for the division of labor, documentation, process reliability and quality. Development work must be parallelized, and software developers must repeatedly become familiar with the source code. The developers need fast and accurate feedback on the quality of their changes made to the source code.

An improved documentation and greater process reliability can be achieved with the help of fine granular traceability links into the source code. Therefore, a meta model for the source code has been defined. This meta model has been integrated into the meta model for the requirements management, change management, test data management and documentation. The entire model is stored in a Software Configuration Management (SCM) repository which allows versioning all artifacts and links. The traceability links can be created and viewed in a source code editor. The history of a single source code artifact including their traceability links can be displayed in the editor.
By using a model for the source code, fine granular pessimistic locks for individual model artifacts can be used. Therefore, the parallel editing of a class or even a method is possible without the need of merging source code. The locks also prevent syntax errors in the SCM repository. The source code editor displays the locks other developers in the source code text.
Continuous integration has been extended so that by storing class files in the repository a faster product build and thus faster feedback for the developer is possible. By using different test selection strategies only tests are executed which are relevant for the changed source code. One test selection strategy uses the traceability information between changed source code, requirement, test specification and the test source code.
Very large models are created in large projects because of the huge amount of source code lines. These models are a big challenge regarding performance and memory requirements. Investigations have been carried out on a project with 6.5 million source code lines.

A prototype based on the Eclipse development environment and for Java has been developed to validate these concepts.

Inhalt

1 Einleitung

1.1 Einführung

Vorgehensmodelle, Prozesse und Werkzeuge für die Softwareentwicklung wurden in den letzten Jahren und Jahrzehnten kontinuierlich weiterentwickelt, um die hohe Komplexität von Softwareprojekten besser bewältigen zu können. So werden die klassischen sequenziellen Prozesse (z. B. Wasserfall) immer mehr durch agile, leichtgewichtige Prozesse wie zum Beispiel Scrum [1] in den Firmen ersetzt. Das ist vor allem bei Softwareprodukten mit großer Komplexität und hohem Innovationsdruck der Fall. Einzelne Praktiken aus den Vorgehensmodellen werden verfeinert und erweitert. Zum Beispiel wird Continuous Integration [2] mittlerweile in vielen Projekten mit Continuous Delivery [3] ergänzt.

Die Werkzeuge haben sich beginnend mit einem Editor und einem Compiler zu mächtigen Softwareentwicklungsumgebungen weiterentwickelt, die das Schreiben von Quellcode und Navigieren im Quellcode, das Analysieren, Übersetzen, Debuggen, Testen und Speichern des Quellcodes in Software Configuration Management (SCM) Systemen erlauben. Für die Unterstützung der verschiedenen Prozessgebiete wie zum Beispiel Anforderungsmanagement, Änderungsmanagement, Testdatenmanagement oder Konfigurationsmanagement wurden viele verschiedene Softwarewerkzeuge entwickelt. Die letzten Jahre haben gezeigt, dass es noch einiges an Verbesserungspotential in der noch immer jungen Disziplin der Softwareentwicklung gibt. So werden derzeit zum Beispiel einzelne Werkzeuge in kompletten Application Lifecycle Management Lösungen mit einer gemeinsamen Datenbasis zusammengefasst.

1.2 Motivation und eigener Beitrag

Vor allem große Softwareentwicklungsprojekte und Projekte mit hoher Komplexität und hohem Innovationsdruck bringen einige Herausforderungen mit sich, die man mit verbesserter Werkzeugunterstützung und optimierten Prozessen versucht zu lösen. Unter anderem sind das folgende Herausforderungen:

Arbeitsteilung

Ein Schlüsselfaktor für den Erfolg der Software im heutigen wettbewerbsintensiven Umfeld sind kurze Produkteinführungszeiten. Der Kunde erwartet in kurzen Release-Zyklen neue Funktionalitäten für seine Software. Die Firmen müssen schnell auf geänderte Marktsituationen reagieren können. Daher muss die Anzahl der Softwareentwickler erhöht werden, die am selben Produkt und am selben Quellcode zur gleichen Zeit arbeiten [4]. Nur so können die Funktionalitäten parallel entwickelt und die Entwicklungszeiten verkürzt werden. Paralleles Bearbeiten von Quellcode kann jedoch zu Konflikten und Syntaxfehler führen, die durch Softwareentwickler gelöst werden müssen.

Dokumentation

In der Regel hat jede neu zu entwickelnde Funktionalität für die Software irgendwelche Berührungspunkte mit bereits existierendem Quellcode oder muss sich in existierenden Quellcode einfügen. In vielen Fällen muss bestehender Quellcode überarbeitet oder erweitert werden. Große Softwareprojekte bestehen aus Millionen von Quellcodezeilen. Da der Softwareentwickler mit nur einer begrenzten Anzahl von Quellcodezeilen vertraut sein kann und bereits bekannter Quellcode immer wieder von anderen Softwareentwicklern überarbeitet wird, ist eine ständige Einarbeitung in den Quellcode notwendig. Es ist notwendig für die Flexibilität der Planung des Projekts das Wissen über den Quellcode auf mehrere Entwickler zu verteilen [5]. Die Dokumentation und die historische Nachvollziehbarkeit von Änderungen am Quellcode sind daher für ein schnelles Verstehen essentiell.

Prozesssicherheit

Desto mehr Personen in einem Softwareentwicklungsprojekt involviert sind, desto wichtiger ist es Prozesse zu definieren und die Einhaltung der Prozesse über Softwarewerkzeuge zu unterstützen und sicher zu stellen. Für viele Prozessdefinitionen ist das Verlinken von Informationen fundamental wichtig, zum Beispiel das Verlinken von Änderungen am Quellcode mit der Anforderung und den dazugehörigen Tests. Dazu werden Softwarewerkzeuge mit einer durchgängigen Datenbasis inklusive des Quellcodes benötigt.

Qualität

Unglücklicherweise bedeutet Parallelität in der Softwareentwicklung eine höhere Wahrscheinlichkeit von inkonsistenten Änderungen und gefährdet daher die Qualität der Software. Die Größe der Software macht es immer schwieriger Änderungen in die Software einzubringen, ohne dass Bestandsfunktionalität „kaputt“ geht. Deswegen müssen Änderungen

am Quellcode möglichst schnell verifiziert werden. In großen Projekten ist das nur noch mit automatischen Tests möglich. Hier sind schnelle Ergebnisse wichtig für eine schnelle Behebung der Fehler in den neuen oder bereits zuvor existierenden Funktionalitäten. Dazu müssen die Zeiten für das Bauen der Software und die Ausführung der automatischen Tests möglichst kurzgehalten werden, damit der Softwareentwickler schnelle Rückmeldung bekommt.

Schlussendlich muss die entwickelte Funktionalität vom Produktmanager oder dem Kunden validiert werden, um sicherzustellen, dass der gewünschte Anwendungsfall effizient und vollständig mit der Software umgesetzt werden kann.

Die drei Punkte „Quellcode verstehen und verlinken“ (Dokumentation und Prozesssicherheit), „Quellcode parallel bearbeiten“ (Arbeitsteilung), „Quellcodeänderungen verifizieren und validieren“ (Qualität) werden in diesem Kapitel bezüglich ihrer Problematik genauer untersucht und der eigene Beitrag für eine Verbesserung dargelegt.

1.2.1 Quellcode verstehen und verlinken

1.2.1.1 Motivation

Den Quellcode zu verstehen ist essenziell für viele verschiedene Aufgaben in der Softwareentwicklung [6]. Dazu verbringt der Entwickler sehr viel Zeit in der Softwareentwicklungsumgebung zum Sichten und Debuggen von Quellcode. Oft wird für das eigentliche Lösen des Problems und die Durchführung der Änderung am Quellcode viel weniger Zeit benötigt.

Zu ihrem Informationsbedarf für ihre tägliche Arbeit wurden Softwareentwickler befragt [7] [8]. Dabei wurden unter anderem folgende Fragen von den Entwicklern gestellt [9]:

Was soll dieser Quellcode tun?

Die Anforderung, die zur Entwicklung oder zur Änderung des Quellcodes geführt hat, geben dem Entwickler wichtige Hinweise, wofür der Quellcode da ist und was er tun soll. Diese Frage ist auch für Quellcode von automatischen Tests relevant, um zu verstehen was der Test eigentlich testen soll.

Warum wurde der Quellcode auf diese Weise implementiert?

Der Kommentar anderer Entwickler bei der Abgabe des Quellcodes (englisch commit) in ein Software Configuration Management Repository kann dem Entwickler helfen zu verstehen,

warum der Quellcode auf diese Weise entwickelt wurde. Auch Designdokumente, auf Basis der Quellcode entwickelt wurde, helfen dem Entwickler den Quellcode zu verstehen.

Wer hat den Quellcode wie geändert?
Über die Historie des Quellcodes und die Information, wer die Änderung durchgeführt hat, kann der Entwickler den Quellcode besser verstehen. Der Entwickler kann dann die entsprechende Person zu ihren Änderungen befragen.

Welcher Quellcode ist bei der Implementierung dieser Funktionalität relevant?
Für den Entwickler ist es wichtig zu erkennen, welcher Quellcode für ein bestimmtes Feature relevant ist. Welche Klassen und Methoden stellen Funktionalitäten zu diesem Feature bereit?

Um dieses Feature in diesen Quellcode zu verschieben, was muss noch verschoben werden?
Um ein Feature von einem Entwicklungsprojekt in ein anderes verschieben zu können, muss bekannt sein, welcher Quellcode dazu gehört und berücksichtig werden muss.

Was wird die Auswirkung dieser Änderung sein?
Um die Auswirkung einer Änderung absehen zu können, muss der Entwickler verstehen, welche anderen Features diesen geänderten Quellcode nutzen und damit eventuell beeinträchtigt sind.

Neben der Dokumentation im Quellcode sind daher zusätzliche Informationen (zum Beispiel Anforderungen und Features) wichtig und hilfreich. In den Prozessgebieten Anforderungsmanagement, Änderungsmanagement, Testdatenmanagement, Design und Dokumentenmanagement entstehen während des Softwareentwicklungsprojekts sehr viele Inhalte die direkt mit dem Quellcode zusammenhängen. Diese Informationen stehen dem Entwickler zur Verfügung, aber es ist schwierig zu einer bestimmten Quellcodestelle möglichst direkt aus dem Quellcodeeditor die relevante Information zu finden und damit für das Verstehen des Quellcodes zu nutzen. Der unmittelbare und direkte Zugriff auf diese Informationen, ohne zum Beispiel die Softwareentwicklungsumgebung verlassen zu müssen, ist dabei für eine optimale Unterstützung des Entwicklers sehr wichtig. Das kann man durch das Verlinken der Informationen mit dem Quellcode über Traceability Links erreichen.

Traceability Links [10] ermöglichen dem Entwickler von einem Quell-Artefakt zu einem Ziel-Artefakt zu gelangen. In diesem Fall ist das Quell-Artefakt eine bestimmte Zeile im

Quellcode und das Ziel-Artefakt ein Artefakt, das während des Softwareentwicklungsprozesses entsteht. Für die verschiedenen Anwendungsfälle sind bidirektionale Traceability Links wichtig, sodass der Entwickler dem Link auch in den Quellcode folgen kann. Die Herausforderung bei diesen Links besteht darin, dass der Quellcode üblicherweise als Text in einem Software Configuration Management (SCM) Repository abgelegt wird. Links in den Text sind schwierig zu pflegen, im Besonderen wenn der Text geändert wird. Ein Traceability Link darf durch die Änderung von Quellcode Text niemals verloren gehen. Für die Bewertung der Gültigkeit eines Traceability Links ist es wichtig zu wissen, wann und von wem der Link erstellt wurde und wie die Inhalte des Quell-Artefakts und des Ziel-Artefakts sich über die Zeit verändert haben. Die Traceability Links und die Artefakte des Softwareentwicklungsprozesses müssen daher mit ihrer Historie in einem Konfigurationsmanagement Repository abgespeichert werden. Die Links müssen nicht mehr durch externes Wissen einzelner Entwickler bewertet werden, sondern die Gültigkeit ergibt sich aus dem gesamten historischen Kontext. Für eine einheitliche und durchgängige Versionierung der Artefakte und des Quellcodes ist es von Vorteil alle Artefakte und den Quellcode in einem Konfigurationsmanagement Repository abzuspeichern.

Um dem Entwickler wichtige Informationen zum Quellcode für ein besseres Verständnis des Quellcodes bereitstellen zu können, werden daher bidirektionale Traceability Links benötigt, wobei das eine Ende des Links der Quellcode ist. Diese Traceability Links erlauben eine durchgängige und vollständige Nachvollziehbarkeit und Rückverfolgbarkeit. Damit sind Auswirkungsanalyse, Abdeckungsanalyse, Projektstatusanalyse und Testoptimierungen mit Berücksichtigung des Quellcodes möglich.

Die Traceability Links müssen erstellt und gepflegt werden. Der dafür notwendige Aufwand kann sehr groß werden, wenn die Erstellung nicht automatisiert oder für den Entwickler mit wenig Aufwand und intuitiv möglich ist.

1.2.1.2 Eigener Beitrag

Um durchgängige und feingranulare Traceability Links zwischen Quellcode und Artefakten aus dem Softwareentwicklungsprozess zu ermöglichen wird ein vollständiges Modell mit Traceability Links zwischen den Artefakten auf Basis eines definierten Metamodells benötigt. Dazu wurde für den Quellcode ein Metamodell mit Hilfe des Abstract Syntax Trees (AST) definiert und in das Gesamtmetamodell integriert. Das Modell wird in einem Repository persistiert. Änderungen an Artefakten und den Traceability Links werden versioniert, damit

die vollständige Historie der Änderungen zur Verfügung steht. Die Traceability Links werden im Quellcode Editor in der Entwicklungsumgebung für einen direkten und unmittelbaren Zugriff dargestellt. Die Änderungshistorie einzelner AST Artefakte inclusive deren Links kann im Editor abgerufen werden. Für die effiziente Erstellung und Pflege der Traceability Links werden entsprechende Funktionalitäten den Softwareentwicklern im Editor zur Verfügung gestellt.

1.2.2 Parallels Bearbeiten von Quellcode

1.2.2.1 Motivation

In viele Softwareentwicklungsprojekten muss auf Grund von Termindruck die Arbeit parallelisiert werden und Softwareentwickler müssen parallel am gleichen Quellcode arbeiten [11]. Die beste Lösung ist es, die Bearbeitung des gleichen Quellcodes durch zwei oder mehr Entwickler zur gleichen Zeit zu vermeiden. Ein Ansatz ist proaktiv in Konflikt stehende Arbeitsaufgaben zu entdecken und so einzuplanen, dass sie nicht parallel bearbeitet werden [12]. Jedoch ist das in der Praxis wegen anderer Einschränkungen zum Beispiel zeitliche Einschränkungen oder Ressourcen Dispositionen nicht immer möglich. Deswegen wird eine Lösung benötigt, die das parallele Bearbeiten von Quellcode ermöglicht. Dabei geht es vor allem darum, die Änderungen anderer Softwareentwickler nicht mit den eigenen zu überschreiben. Verschieden Software Configuration Management Systeme (SCM) haben diese Probleme des parallelen Editierens adressiert und stellen entsprechende Lösungen bereit, die Vorteile und Nachteile haben [13] [14]. Der Lösungsraum spannt sich dabei vom uneingeschränkten parallelen Editieren bis hin zu erzwungenem sequentiellen Editieren des Quellcodes. Im ersten Fall muss der Quellcode im Konfliktfall verschmolzen werden, was sehr viel Aufwand bedeuten kann und fehleranfällig ist. Es können neue semantische Fehler verursacht werden, die im nicht verschmolzenen Quellcode nicht existiert haben. Ein weiteres Problem des parallelen Editierens ist, dass es zu Syntax Fehlern im Quellcode im SCM Repository kommen kann. Das hat zur Folge, dass die Software nicht gebaut und nicht automatisch getestet werden kann.

1.2.2.2 Eigener Beitrag

Es wurde ein Konzept für ein feingranulares und pessimistisches Sperren einzelner Abstract Syntax Tree (AST) Artefakte entwickelt. Damit werden Konflikte beim Editieren des Quellcodes und damit das aufwändige Verschmelzen (englisch merge) von Quellcode verhindert. Trotzdem ist das parallele Bearbeiten einer Klasse oder Methode durch mehrere

Softwareentwickler weiterhin möglich. Die Sperren sind dahingehend optimiert, dass die gegenseitige Behinderung bei der parallelen Bearbeitung des Quellcodes minimiert wird. Durch die feingranularen Sperren werden Syntaxfehler im SCM Repository verhindert. Der Quellcode kann daher zu jeder Zeit kompiliert und damit die Software gebaut und getestet werden. Die feingranularen Sperren werden während des Editierens des Quellcodes automatisch bezogen. Die Sperren anderer Entwickler sind für den Entwickler im Editor und in der Entwicklungsumgebung sichtbar und liefern ihm damit Informationen über den gerade bearbeiteten Quellcode.

1.2.3 Verifizieren und Validieren der Quellcode Änderungen

1.2.3.1 Motivation

Nachdem der Softwareentwickler die Quellcode Änderungen für die Umsetzung der Anforderungen durchgeführt und in das Software Configuration Management (SCM) Repository abgegeben hat, müssen diese Änderungen validiert und verifiziert werden [15]. Jede Änderung, die der Softwareentwickler in den Quellcode einbringt, kann neue Softwarefehler verursachen. Diese neuen Softwarefehler können nicht nur in dem neu zu entwickelten Feature auftreten, sondern auch in bereits existierenden Features und somit bestehende Funktionalität des Produkts mindern, einschränken oder unbrauchbar machen. Daher ist eine Verifikation aller Features des Produkts notwendig. Desto später der Softwarefehler entdeckt wird, desto höher ist der Aufwand den Fehler zu beheben [16]. Alle Softwarefehler, die während der Entwicklung durch den Softwareentwickler, der den Fehler verursacht hat, gefunden werden, können mit geringem Aufwand behoben werden. In diesem Moment kennt der Entwickler den Quellcode sehr genau. Es ist keine zusätzliche Dokumentation des Fehlers notwendig und es werden keine weiteren Personen involviert. Aber wenn andere Personen insbesondere der Produktmanager oder der Tester mit dem Softwarefehler konfrontiert werden, steigt der Aufwand zur Behebung des Fehlers signifikant. Wenn der Softwarefehler erst beim Kunden entdeckt wird, ist unter Umständen ein Service Pack notwendig.

Die Validierung wird in der Regel vom Produktmanager oder Kunden durchgeführt. Dazu wird eine aktuelle, stabile und lieferbare Version des Softwareprodukts mit den neuesten Änderungen des Softwareentwicklers benötigt. Deswegen müssen in regelmäßigen und kurzen Intervallen während der Entwicklung Versionen des Softwareprodukts gebaut und bereitgestellt werden. Die Anforderung bezüglich der Qualität an diese Versionen ist hoch,

damit eine Validierung überhaupt möglich ist. Eine Validierung des Features sollte schon während der Entwicklung oder spätestens unmittelbar nach Fertigstellung möglich sein, um potentielle Probleme zum Beispiel in der Bedienung der Software oder Missverständnisse zwischen Softwareentwickler und Produktmanager schnell ausräumen zu können. Wenn erst kurz vor dem finalen Release des Softwareprodukts eine falsch umgesetzte Anforderung entdeckt wird, kann die Überarbeitung der Software dazu führen, dass das Release um mehrere Wochen verschoben werden muss. Die Verschiebung eines Release Termins wird immer das Vertrauen in den Softwarelieferanten erschüttern [17].

Ziel ist es daher dem Produktmanager oder Kunden regelmäßig sehr stabile Produktversionen zur Validierung bereitzustellen (am besten täglich) um die Mehraufwände durch spät im Projekt behobene Softwarefehler zu vermeiden. Der Softwareentwickler sollte bereits kurz nachdem er den geänderten Quellcode im SCM Repository abgegeben und damit allen Projektteilnehmern zur Verfügung gestellt hat, Hinweise über neu entstandene Softwarefehler erhalten. Dieses Feedback sollte innerhalb von Minuten oder wenigen Stunden erfolgen und präzise Auskunft über die Quellcode Änderungen des Softwareentwicklers geben und nicht über die Änderungen anderer Softwareentwickler. Schnelles und akkurates Feedback ist essenziell um Probleme frühzeitig zu erkennen und Softwarefehler sofort beheben zu können, bevor der Softwareentwickler bereits durch neue Aufgaben abgelenkt wird und andere Personen mit dem Fehler konfrontiert werden. Um dieses Ziel erreichen zu können, muss der geänderte Quellcode über automatische Tests verifiziert werden. Manuelle Tests wären viel zu personalintensiv und langwierig [3].

Continuous Integration [18] ist die gängige Praxis um Produktversionen mit jeder Abgabe von Quellcode (englisch commit) zu bauen und zu testen. Die Herausforderungen in größeren Projekten liegen jedoch darin, schnell eine Produktversion bereitzustellen. Diese Projekte beinhalten sehr viel Quellcode (typischerweise mehrere Millionen von Quellcode Zeilen), dessen Übersetzung in ein ausführbares Programm entsprechend lange dauert. Prinzipiell kann man solche Produkte in mehrere Teile zerlegen und verantwortliche Entwicklerteams dafür definieren. Da jedoch neu zu entwickelnde Funktionalitäten und zu behebende Fehler oft mehrere Produktanteile betreffen, geht dabei viel an agilen Entwicklungsmöglichkeiten verloren. Ein Entwickler sollte bei notwendigen Änderungen nicht nur auf einen Teil des Quellcodes eingeschränkt sein. Das getrennte Bauen und Testen der Produktteile bedeutet

immer mehr Komplexität und Aufwand durch die zusätzlich notwendige Integration und ist wenn möglich zu vermeiden [3].

Es sollten nur die automatischen Tests ausgeführt werden, die für die Änderungen relevant sind. Die Ausführung aller Tests würde viel zu viel Zeit in Anspruch nehmen und eine schnelle Rückmeldung an den Entwickler verhindern. Es ist daher eine sinnvolle Testauswahl notwendig, die jene Tests ermittelt, die tatsächlich den geänderten Quellcode betreffen.

1.2.3.2 Eigener Beitrag

Durch das Persistieren und Wiederverwenden von Binär Dateien, die bei der Übersetzung von Quellcode-Dateien erstellt werden, wird ein beschleunigter Produkt-Build für große Software Produkte möglich. Der Produkt-Build wurde zu diesem Zweck entsprechend angepasst.

Continuous Integration wurde dahingehend erweitert, dass für die Testausführung eine dynamische Testauswahl ermöglicht wird. Es wurde ein Testauswahlframework entwickelt, das abhängig vom geänderten Quellcode möglichst passende Tests selektiert. Eine Strategie des Testauswahlframeworks analysiert dazu das Modell mit den Traceability Links in den Quellcode hinein um passende Tests zu finden. Dazu werden Traceability Links zwischen geänderten Quellcode, Anforderungen, Testfall und Test-Quellcode ausgewertet. Eine andere Strategie berücksichtigt die zuvor ermittelte Testabdeckung eines Tests.

1.3 Rahmenbedingungen

Die oben beschriebenen Beiträge wurden im Rahmen eines Projekts mit dem Namen „Morpheus“ umgesetzt. Der Aufwand für die Entwicklung einer eigenen Softwareentwicklungsumgebung wäre zu groß gewesen. Daher wurde als Entwicklungsumgebung Eclipse [19] verwendet und es wird die Programmiersprache Java unterstützt [20]. Ziel ist es dabei möglichst alle bestehende Funktionalität von Eclipse weiterhin nutzen zu können. Dies wird unter anderem dadurch erreicht, dass keine Komponente von Eclipse ersetzt wird, sondern nur einzelne Komponenten erweitert werden. Der volle Funktionsumfang der jeweiligen Komponente kommt dadurch weiterhin zum Einsatz. Deswegen kann ein neues Release von Eclipse mit relativ wenig Aufwand nach Morpheus übernommen werden.

Des Weiteren wurden verschiedene Komponenten und Teile des Metamodells von PREEvision [21] für die Umsetzung der Konzepte verwendet. PREEvision wird von der Firma Vector Informatik GmbH entwickelt und ist ein Werkzeug, das ebenfalls auf Eclipse

basiert und das vor allem in der Automobilindustrie zum Einsatz kommt. PREEvision kommt auch als Application Lifecycle Management Werkzeug zum Einsatz.

1.4 Aufbau der Arbeit

In Kapitel 2 werden für die Arbeit relevanten technischen Grundlagen vorgestellt. Das sind Themen bezüglich Software Configuration Management, Traceability, Metamodellierung, Abstract Syntax Tree, Eclipse, PREEvision und Continuous Integration.

In Kapitel 3 werden bezüglich Quellcode verstehen und verlinken bereits existierende Lösungen für Traceability Links in den Quellcode diskutiert und eine eigene Lösung und deren Umsetzung vorgestellt. Die Erstellung und Visualisierung der Links innerhalb des Java Editors von Eclipse wird präsentiert.

In Kapitel 4 werden die verschiedenen bereits existierenden Lösungsansätze für das parallele Bearbeiten von Quellcode vorgestellt. Auf Basis der in Kapitel 3 präsentierten Lösung wird eine neue Möglichkeit des feingranularen pessimistischen Sperrens von Quellcode und deren Umsetzung innerhalb des Eclipse Java Editors dargelegt.

Kapitel 5 beschäftigt sich mit dem Thema Quellcode verifizieren und validieren. Es wird eine Erweiterung von Continuous Integration präsentiert, die es erlaubt schnelleres und präziseres Feedback vom Continuous Integration Lauf zu bekommen. Dabei werden auch Traceability Informationen in den Quellcode aus Kapitel 3 genutzt.

In Kapitel 6 werden Performance und Speicherbedarf der in Kapitel 3 vorgestellten Lösung untersucht und umgesetzte Optimierungen vorgestellt und diskutiert.

Die Arbeit schließt mit Kapitel 7 mit einer Zusammenfassung und einem Ausblick.

2 Technische Grundlagen

2.1 Software Configuration Management

Software Configuration Management (SCM) kontrolliert und managt die Änderungen an großen und komplexen Softwaresystemen und deren Versionen [13]. Der Softwareentwickler wird in die Lage versetzt, koordinierte und kontrollierte Änderungen an der Software vorzunehmen. Die Wichtigkeit und Bedeutung von SCM ist heute unumstritten und findet sich in vielen Softwarevorgehensmodellen wieder.

2.1.1 Versionskontrolle

Versionskontrolle ist eine Teil-Disziplin von SCM. Hier werden die Änderungen eines spezifischen Artefakts über die Zeit verfolgt [22]. Die Version eines Softwareprodukts entspricht der Momentaufnahme aller zum Softwareprodukt gehörende Quellcode Artefakte zu einem bestimmten Zeitpunkt. Eine Revision bezieht sich auf eine Version, die durch die Änderung einer anderen Version entstanden ist [9]. Oft wird nicht zwischen Version und Revision unterschieden, weil jede Version aus der Änderung einer anderen Version entsteht (mit der Ausnahme der ersten Version). Die Versionskontrolle stellt folgende wichtige Funktionalitäten bereit [23]:

- Alle Versionen werden archiviert, sodass jede Version wiederhergestellt werden kann. So ist es zum Beispiel möglich eine ältere und bereits an den Kunden ausgelieferte Produktversion wiederherzustellen, um gemeldet Softwarefehler nachvollziehen zu können.
- Die Änderungen einer spezifischen Person zu einem spezifischen Zeitpunkt können ermittelt werden. Damit können die Änderungen über der Zeit nachvollzogen und verstanden werden.
- Fehlerhafte Änderungen am Quellcode (zum Beispiel das versehentliche Löschen einzelner Dateien) können wieder rückgängig gemacht werden, auch wenn diese Änderungen schon in das Repository abgespeichert wurden.
- Versionskontrolle erlaubt die Verwaltung von sogenannten Entwicklungszweigen. Damit ist eine parallele Entwicklung des Quellcodes möglich. Zum Beispiel wird gleichzeitig an der Entwicklung des nächsten Produktreleases gearbeitet und an Fehlerbehebungen des vorangegangenen Produktrelease für ein Service Pack.

In einfachsten Fall wird nur an einem Entwicklungszweig gearbeitet und alle Entwickler arbeiten an der letzten Version. Damit hat jede Version genau eine Nachfolgerversion. Wenn es mehrere Entwicklungszweige gibt, kann eine Version mehr als eine Nachfolgerversion besitzen. Ein Entwicklungszweig beginnt mit einer bestimmten Version eines anderen Entwicklungszweigs. Der Entwicklungszweig kann nach einiger Zeit des parallelen Entwickelns wieder in einen anderen Entwicklungszweig verschmolzen werden. Ein verschmolzener Entwicklungszweig beginnt dann mit einer Version, die zwei Vorgängerversionen besitzt.

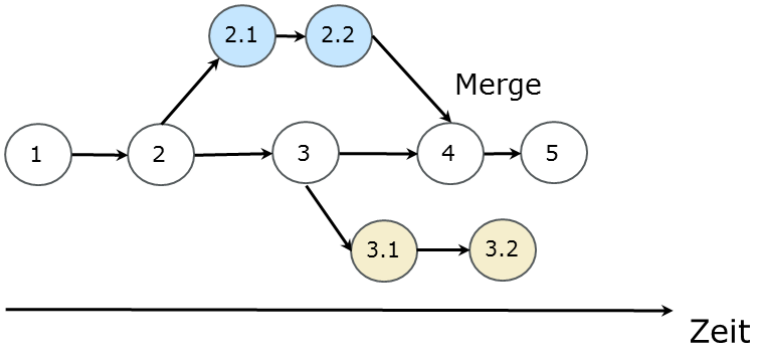


Abbildung 1: Revisionsgraph mit drei Entwicklungszweigen

Die Historie von Versionen oder Entwicklungszweigen kann durch einen Revisionsgraphen dargestellt werden. Die Versionen werden dabei durch Knoten repräsentiert und die Weiterentwicklung einer Version durch eine gerichtete Kante. In der Abbildung 1 sind drei Entwicklungszweige zu sehen. Der Entwicklungszweig (2.x) wurde in den Hauptentwicklungszweig zurück verschmolzen (englisch Merge).

2.1.2 Software Configuration Management (SCM) Repository

Ein Software Configuration Management Repository ist der Ort, wo die verschiedenen Versionen des Quellcodes eines Softwareprojekts gespeichert werden. Dort werden alle Versionen archiviert und können von dort auch wieder abgerufen werden. Deswegen muss das Repository gegen Hardwareausfälle abgesichert werden und die Daten sollten regelmäßige auf ein Backup-Medium gesichert werden. Ein Entwickler holt sich aus dem SCM Repository alle benötigten Dateien und speichert sie in einer persönlichen Arbeitskopie ab. Diese Arbeitskopie wird dann regelmäßig mit den Dateien aus dem SCM Repository aktualisiert. Nach dem die lokalen Änderungen an den Quellcode-Dateien abgeschlossen sind, werden die Änderungen in das SCM Repository abgegeben. Der Entwickler kann während der Abgabe (englisch commit) einen sogenannten Abgabe-Kommentar (englisch commit

comment) eingeben, der zusätzliche Informationen über die durchgeführten Änderungen beinhaltet (zum Beispiel den Änderungsgrund). Die Menge aller durch einen Softwareentwickler zu einem Zeitpunkt abgegebenen Dateien nennt man Change Set. Die Change Sets können vom SCM Repository sehr leicht abgefragt werden und die Änderungen, die in diesem Change Set durchgeführt wurden, können dann gesichtet werden.

In den letzten Jahren haben verteilte SCM Repositories stark an Popularität zugenommen [24] im Gegensatz zum zentralen Ansatz mit der klassischen Client / Server Architektur. Verteilte SCM Repositories speichern das komplette Repository auf dem lokalen Computer des Entwicklers ab. Damit können Versionskontroll-Operation offline ausgeführt werden und Änderungen erstmal nur in das lokale Repository abgegeben werden. Eine finale Abgabe kann aber nur online erfolgen und das lokale Repository muss aus anderen Repositories regelmäßig aktualisiert werden.

Einer der bekanntesten Vertreter für den zentralen Ansatz ist Subversion [23], für den verteilten Ansatz Git [25].

2.1.3 SCM Zusammenarbeit

Ein wichtiges Ziel von SCM Repositories ist, die Zusammenarbeit beim Editieren von Quellcode zu ermöglichen [23]. Es muss möglich sein, dass der Quellcode von mehr als einem Entwickler bearbeitet werden kann, ohne dass man die Änderungen anderer Entwickler versehentlich überschreibt. Zum Beispiel: zwei Entwickler holen sich dieselbe Quellcode-Datei aus dem SCM Repository und nehmen Änderungen zur gleichen Zeit an dieser Datei vor. Dann speichern sie diese Änderungen nacheinander in das SCM Repository ab. Ohne zusätzliche Logik würden die Änderungen des ersten Entwicklers durch die Änderungen des zweiten Entwicklers überschrieben. Es gibt zwei Strategien um das Problem zu lösen: Pessimistisches und optimistisches Sperren [26] [27].

2.1.3.1 Pessimistisches Sperren

Bei dieser Strategie erlaubt das SCM Repository nur einer Person eine Datei zur gleichen Zeit zu editieren. Das wird durch exklusive Sperren erreicht. Der Entwickler muss eine Sperre (englisch lock) für die Datei bekommen, bevor er Änderungen an dieser Datei vornehmen kann. Ein anderer Entwickler kann in dieser Zeit keine zusätzliche Sperre für diese Datei bekommen und die Datei daher nicht ändern. Die Datei kann nur gelesen und es muss auf die Freigabe der Datei gewartet werden. Nachdem der erste Entwickler seine Änderungen

vorgenommen und die Sperre wieder aufgehoben hat, muss der zweite Entwickler diese geänderte Version der Datei aus dem SCM Repository laden und kann dann nach dem Sperren der Datei mit den Änderungen beginnen. Die Änderungen werden daher sequentiell vorgenommen. Es kann zu keinen Konflikten kommen. Pessimistisches Sperren bringt folgende Nachteile mit sich:

- Durch die Sperren kann Mehraufwand in der Administration entstehen. Wenn ein Entwickler eine Datei sperrt und dann plötzlich verhindert ist (zum Beispiel krank wird), dann muss diese Sperre durch einen Administrator wieder aufgehoben werden, damit andere Entwickler an dieser Datei weiterarbeiten können.
- Durch die Sperren wird eine sequentielle Bearbeitung erzwungen auch wenn es nicht unbedingt notwendig ist. Wenn die Entwickler an ganz unterschiedlichen Stellen in der Datei Änderungen durchführen, dann spricht nichts gegen eine parallele Bearbeitung solange die Änderungen beider Entwickler am Ende in eine Datei konfliktfrei verschmolzen werden können.
- Durch die Sperren wird der Entwickler in eine falsche Sicherheit gewiegt. Durch das gleichzeitige Editieren von zwei Dateien, die semantisch voneinander abhängen, kann es zu semantischen Problemen kommen. Änderungen können dazu führen, dass die zwei Dateien nicht mehr zusammen funktionieren, weil sie semantisch nicht mehr kompatibel sind. Das kann durch die Sperren auf Dateiebene nicht verhindert werden.

2.1.3.2 Optimistisches Sperren

Bei dieser Strategie arbeiten die Entwickler simultan und unabhängig an den Dateien. Wenn die Dateien in das SCM Repository abgegeben werden, müssen die Dateien unter Umständen mit den aktuellen Dateien des SCM Repository verschmolzen werden. Das funktioniert im Einzelnen wie folgt: Der Entwickler ändert eine oder mehrere Dateien über einen gewissen Zeitraum. Nach Fertigstellung der Änderungen versucht er die Dateien in das SCM Repository abzugeben. Wenn die geänderten Dateien von keinem anderen Entwickler in der Zwischenzeit geändert wurden, wird die Abgabe durchgeführt und die Dateien werden im SCM Repository abgespeichert. Wenn es Änderung gab, muss der Entwickler zuerst diese Dateien mit den Dateien aus dem SCM Repository aktualisieren und die Änderungen werden in eine finale Datei verschmolzen. Falls die gleichen Quellcode-Zeilen geändert wurden, kommt es zu einem Konflikt und der Entwickler muss eingreifen um den Konflikt aufzulösen und muss entscheiden, wie der verschmolzene Quellcode auszusehen hat. Danach kann er

wieder eine Abgabe anstoßen, die dann durchläuft, wenn nicht jemand in der Zwischenzeit wieder neue Änderungen zu diesen Dateien abgegeben hat. Optimistisches Sperren bringt folgende Nachteile mit sich:

- Das Verschmelzen des Quellcodes kann unter Umständen sehr aufwendig werden. Wenn zum Beispiel Änderungen an einer Methode vorgenommen wurden, die inzwischen verschoben oder gelöscht wurde, kann das Verschmelzen eine signifikante und zeitaufwändige Überarbeitung des Quellcodes bedeuten. Deswegen muss der Entwickler regelmäßig seine Dateien mit den Dateien aus dem SCM Repository aktualisieren um Probleme frühzeitig zu erkennen. Desto länger der Entwickler seine lokale Arbeitskopie nicht aktualisiert, desto größer ist die Wahrscheinlichkeit, dass schwierig zu lösende Konflikte entstehen und das Verschmelzen sehr viel Zeit in Anspruch nehmen kann.
- Das parallele Bearbeiten von semantisch abhängigen Dateien wird hier nicht verhindert und kann in weiterer Folge zu Syntaxfehlern und Softwarefehlern führen.

2.2 Traceability

Die Wichtigkeit von Traceability zwischen Softwareentwicklungsartefakten, die im Zuge des Softwareentwicklungsprozesses entstehen, ist gut verstanden und wurde schon in verschiedenen Softwareentwicklungsstandards eingearbeitet [28]. Traceability, im besonderen Anforderungsverfolgung, hat schon sehr viel Aufmerksamkeit in der Forschungsgemeinschaft bekommen [29] [30] [5] [31] [32] [33].

2.2.1 Definitionen

Im Folgenden werden einige Begriffe definiert, die im Verlauf dieser Arbeit verwendet werden.

Ein Artefakt ist das Erzeugnis des Softwareentwicklungsprozesses, das sowohl ein Zwischen- als auch ein Endprodukt sein kann. Die Artefakte lassen sich dabei eindeutig einem Prozessgebiet zuordnen. Ein Prozessgebiet ist eine in sich abgeschlossene Disziplin im Entwicklungsprozess von Softwarevorgehensmodellen, wie z. B. Spice [34] oder das V-Modell [35]. Beispiele für Prozessgebiete sind Anforderungsmanagement, Design, Konfigurationsmanagement, Änderungsmanagement oder Projektmanagement. Da Traceability Links zwischen Artefakten unterschiedlicher Prozessgebiete existieren können, sind sie nicht einem Prozessgebiet eindeutig zuordenbar.

Der Traceability Link (kurz Link) wird in [10] folgendermaßen definiert:

„ …ist eine implizit oder explizit existierende und verfolgbare Verbindung zwischen Quell- und Ziel-Artefakten mit beliebiger, aber bestimmter, Semantik. Die Verbindung enthält Kontext-Informationen zur Beziehung zwischen diesen Artefakten.“

Der Traceability-Link-Typ wird in weiterer Folge so definiert [10]:

„Ein Traceability-Link-Typ (kurz: Link-Typ) ist eine Klasse an Traceability Links mit gemeinsamen Eigenschaften, also gemeinsamer Semantik und gemeinsamer Kontext-Information.“

Ein Trace ist dabei eine Folge von Traceability Links, wobei das Ziel-Artefakt des einen Links gleich dem Quell-Artefakt des nächsten Links sein muss. Die Artefakte und deren Traceability Links stellen einen gerichteten Graphen dar. Die Artefakte sind die Knoten und die Traceability Links sind die Kanten. Ein Trace ist dementsprechend ein Pfad in diesem Graphen mit beliebigen Start und End-Artefakt.

Die Artefakte und die Traceability Links können in einem Traceability Modell repräsentiert werden. Das Traceability Modell wiederum wird durch ein Traceability Metamodell beschrieben (siehe dazu Kapitel 2.3).

2.2.2 Klassifizierung von Traceability Links

Traceability Links können folgendermaßen klassifiziert werden (siehe Abbildung 2) [36]: vorwärts, rückwärts, horizontal und vertikal, Pre- und Post-Anforderungsspezifikation.

Vorwärts Traceability bedeutet den Link von dem Quell-Artefakt zu dem davon abgeleiteten Artefakt zu folgen. Rückwärts Traceability steht für den umgekehrten Weg vom abgeleiteten Artefakt zum Quell-Artefakt. Ein bidirektionaler Link unterstützt vorwärts und rückwärts Traceability.

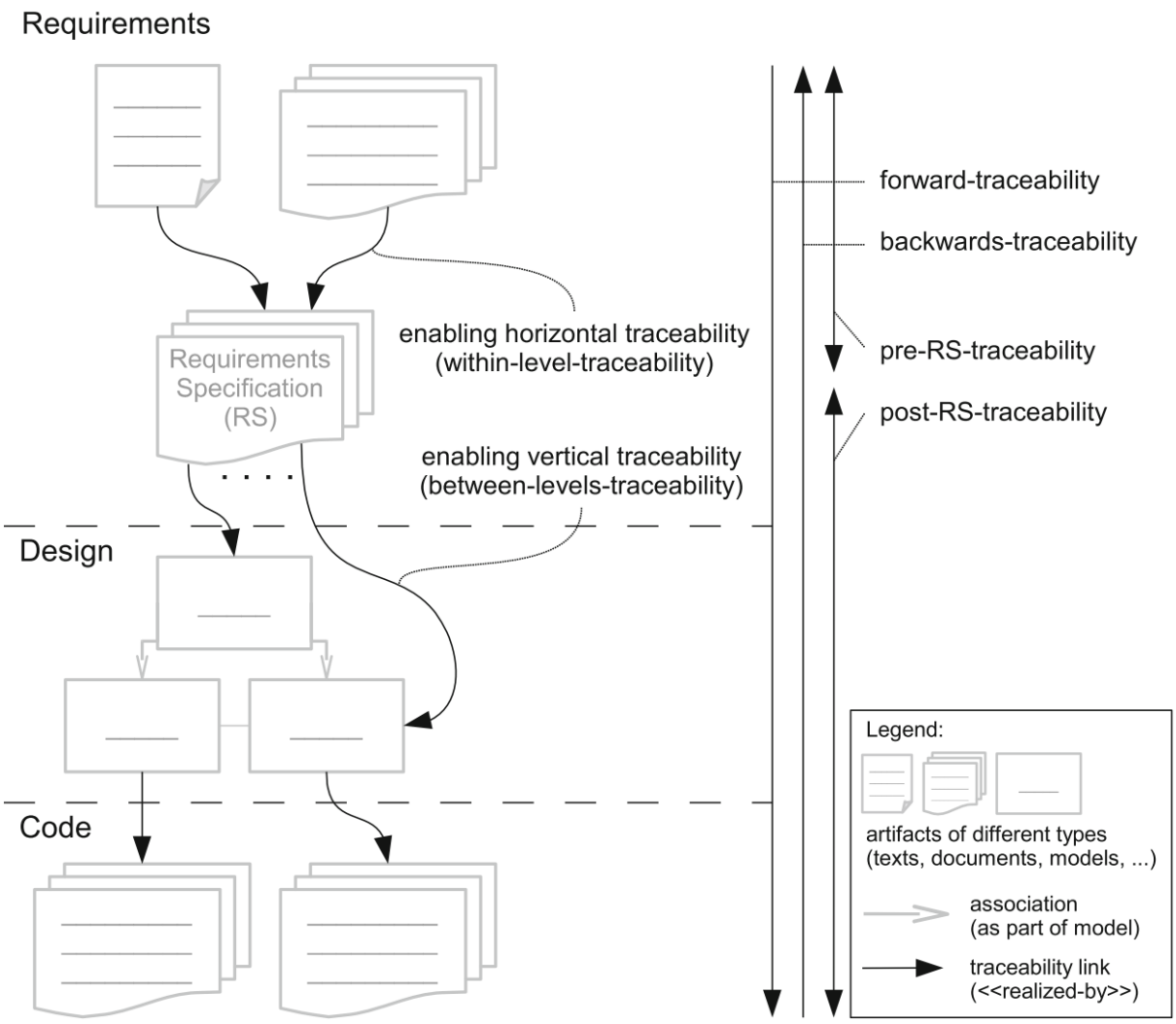


Abbildung 2: Dimensionen und Richtungen von Traceability Links [36]

Horizontale Traceability sind Links zwischen Artefakten, die zum selben Prozessgebiet oder zur selben Projektphase gehören, vertikale Traceability besteht zwischen Artefakten unterschiedlicher Abstraktionslevels.

Die Klassifizierung nach Pre-Anforderungsspezifikation [29] umfasst alle Anforderungs-Links, die vor Fertigstellung der Anforderungsspezifikation entstehen, also während der Auswahl, Diskussion und Finalisierung der Anforderungen. Die Post-Anforderungsspezifikation sind die Anforderungs-Links, die während der schrittweisen Implementierung der Anforderungen erstellt werden, also während der Design- und Codierungsphase.

2.2.3 Aktuelle Herausforderungen

Es gibt immer noch eine Reihe von Herausforderungen mit der Einführung und Anwendungen von Traceability Links [10] [9]. Die im Folgenden aufgeführten Probleme werden in dieser Arbeit angegangen:

- Es fehlt ein einheitliches und durchgängiges Modell zur Beschreibung der Traceability Informationen.
- Es gibt keine durchgängigen Werkzeuge für die Erfassung von Traceability Links mit hohem Automatisierungsgrad. Dadurch ist die Erstellung und Pflege der Links mit großem Aufwand verbunden und entsprechend fehlerträchtig.
- Wichtige Traceability Links können nur schwer genutzt werden, weil sie entweder nur implizit vorhanden sind oder über unterschiedliche Repositories hinweg existieren.
- Traceability Links sind nur dann sinnvoll, wenn sie tatsächlich die aktuellen Abhängigkeiten widerspiegeln. Aber der Aufwand für die Pflege ist oft sehr groß und daher erodieren die Links immer mehr in einen inakkuraten Zustand. Die Links müssen sich daher mit den Artefakten weiter entwickeln besonders für textuelle Artefakte.

2.3 Metamodellierung

Ein Metamodell definiert die Syntax für ein Modell [37]. Der Begriff setzt sich aus „meta“ (altgriechisch für „hinter“, „danach“ oder „jenseits“) und Modell zusammen und beschreibt daher das Modell hinter einem Modell.

Ein Modell ist die simplifizierte Repräsentation eines Systems, das analysiert oder beschrieben wird. Dabei werden Details nur berücksichtigt, wenn sie für die Verwendung des Modells wichtig sind. Durch Weglassen von Details wird eine vereinfachte Darstellung erreicht [10]. Manchmal wird mehr als nur ein Modell benötigt um verschiedene Aspekte des Systems zu beschreiben. Modelle sind stets subjektiv und für einen bestimmten Nutzen gedacht. Sie helfen das Verständnis für ein System zu verbessern.

Ein Modell kann nichts enthalten was nicht im Metamodell definiert ist. Ein Modell wird aus einem Metamodell heraus instanziiert und das Metamodell abstrahiert das Modell. Die Object Management Group (OMG) [38] hat einen Standard für die Definition von Metamodellen definiert: Meta Object Facility (MOF) [39]. Alle Metamodelle in dieser Arbeit basieren auf diesem Standard.

Die OMG hat das Vier-Schichten-Modell definiert (siehe Abbildung 3). Das Vier-Schichten-Modell spezifiziert wie komplexe Datenstrukturen metamodellbasiert formal beschrieben werden können. Es besteht aus den vier Abstraktionsschichten oder –ebenen: M0 (konkrete Schicht) bis zu M3 (abstrahierte Schicht).

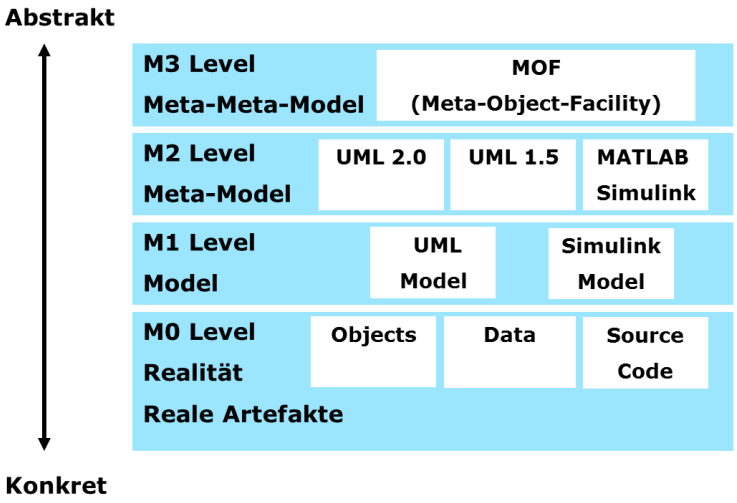


Abbildung 3: Das Vier-Schichten-Modell der OMG

Jede Schicht ist eine Abstraktion der darunterliegenden Schicht. M3 ist die höchste Schicht in der Abstraktion und jede Schicht darunter ist eine Spezialisierung und Instanziierung der darüberliegenden Schicht.

- **M0-Ebene**: Diese Ebene besteht aus tatsächlichen existierenden Systemartefakten, die abstrahiert werden sollen, wie zum Beispiel konkrete Daten oder reale Objekte.
- **M1-Ebene**: Diese Ebene ist die erste Abstraktionsebene und enthält das Modell für die Realität in der M0-Ebene. Diese Ebene wird daher auch die Modellebene genannt. In dieser Schicht sind zum Beispiel konkrete Unified Modeling Language (UML) [40] und Simulink [41] Modelle vertreten, welche die Struktur und das Verhalten von Softwaresystemen beschreiben.
- **M2-Ebene**: Diese Ebene wird auch Metamodell-Ebene genannt, weil es die Metamodelle der Modelle aus Ebene M1 enthält. Die Metamodelle definieren die Syntax und Semantik der Modelle. Das Modell ist eine Instanz des Metamodells. Vertreter der M2-Schicht ist zum Beispiel das Metamodell der UML, welches die Syntax von UML Modellen definiert.

- **M3-Ebene**: Diese Ebene wird auch Meta-Metamodell Ebene genannt und ein Metamodell wird unter Verwendung der Konstrukte aus dieser Ebene definiert. Es sind keine weiteren abstrakteren Ebenen notwendig, weil mit den vorhandenen Beschreibungsmitteln aus dieser Ebene das Meta-Metamodell beschrieben werden kann, d.h. Meta-Metamodelle können sich selbst beschreiben. Ein Vertreter dieser Ebene ist zum Beispiel MOF von der OMG.

In der MOF 2.0 wird das starre Vier-Schichten-Modell etwas aufgeweicht, indem beliebig viele Ebenen erlaubt sind. Es müssen aber mindestens zwei Ebenen zum Einsatz kommen für die folgendes gilt: eine Schicht spezifiziert die direkt darunterliegende Schicht und eine Schicht ist eine Instanz der direkt darüberliegenden Schicht.

In dieser Arbeit werden alle Metamodellklassennamen im Text kursiv geschrieben.

2.4 Abstract Syntax Tree

Der Abstract Syntax Tree (AST) entsteht während des Kompilierens von Quellcode. Ein Compiler ist ein Programm, das ein Programm einer Quellsprache in ein äquivalentes Programm einer anderen Sprache (Zielsprache) übersetzt [42]. Dabei durchläuft der Compiler in der Regel folgende Phasen (siehe auch Abbildung 4):

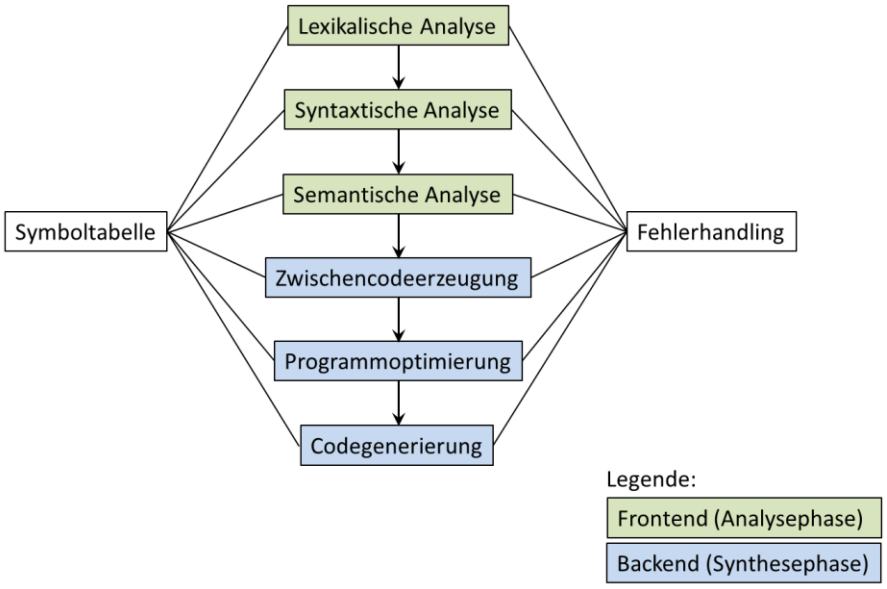


Abbildung 4: Phasen eines Compilers

1. Lexikalische Analyse: Diese Analyse zerlegt den eingelesenen Quellcode in lexikalische Einheiten (Tokens) zum Beispiel in Schlüsselwörter, Zeichenketten,

Zahlen, Operatoren und Bezeichner. Whitespaces wie zum Beispiel Leerzeichen oder Tabulatoren werden dabei übersprungen.

2. Syntaktische Analyse: Diese Analyse überprüft ob der Quellcode der kontextfreien Syntax (Grammatik) der Programmiersprache entspricht und in der korrekten Struktur vorliegt. Dabei wird aus dem Quellcode ein abstrakter Syntaxbaum erzeugt, der in den nächsten Phasen weiterverwendet wird.
3. Semantische Analyse: Diese Analyse prüft ob semantische Fehler im Quellcode vorliegen und sammelt Typinformationen für die nächsten Phasen. Eine wichtige Überprüfung ist die Typüberprüfung, zum Beispiel ob Zuweisungen mit kompatiblen Datentypen erfolgen.
4. Zwischencodeerzeugung: Aus dem abstrakten Syntaxbaum wird dann ein Zwischencode erzeugt.
5. Programmoptimierung: Der Zwischencode ist die Basis für verschiedene Programmoptimierungen.
6. Codegenerierung: Aus dem Zwischencode wird dann der Programmcode der Zielsprache erzeugt.

Die Symboltabelle speichert alle Identifier, die im Quellcode verwendet werden. Für jeden Identifier werden zusätzliche Informationen in den Phasen gesammelt, wie zum Beispiel Typ oder Geltungsbereich. In jeder Phase können Fehler auftreten und dafür wird ein entsprechendes Fehlerhandling benötigt. In Abbildung 5 ist ein Beispiel für einen abstrakten Syntaxbaum zu sehen mit dem dazugehörigen Quellcode.

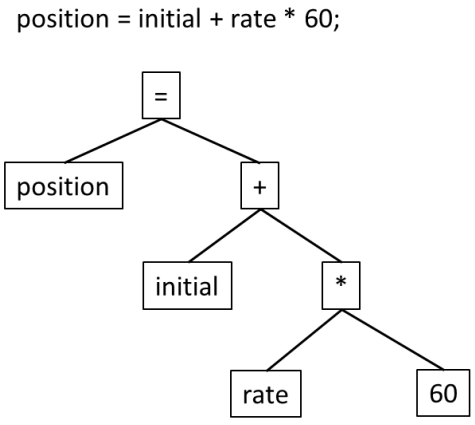


Abbildung 5: Beispiel für einen abstrakten Syntaxbaum [42]

Im Fall von Java [20] ist die Zielsprache Bytecode, der nicht direkt vom Computer ausgeführt werden kann, sondern von einer virtuellen Maschine (VM) interpretiert wird.

2.5 Eclipse

Die erste Version von Eclipse wurde von IBM im November 2001 freigegeben und der Open-Source-Gemeinschaft zur Verfügung gestellt [43]. Die Weiterentwicklung von Eclipse wird von eclipse.org verwaltet, einer unabhängigen Non-Profit-Organisation [19]. Eclipse ist eine Plattform mit einer Plugin-Architektur, die beliebige Erweiterungen erlaubt und eine Anpassung für unterschiedlichste Aufgaben ermöglicht. Die Verwendung als Integrierte Entwicklungsumgebung (kurz IDE, von englisch Integrated Development Environment) für verschiedene Programmiersprachen wie zum Beispiel Java ist nur ein möglicher Einsatz von Eclipse. Eclipse ist mittlerweile eine Plattform für viele verschiedene Werkzeuge und Anwendungen geworden. Durch die Verwendung von Java ist Eclipse weitgehend plattformunabhängig und durch den Einsatz eigener GUI-Bibliotheken (SWT und JFace) wird ein natives Look & Feel der graphischen Oberfläche der entsprechenden Plattform erreicht. Das wird durch die direkte Verwendung der jeweiligen plattformspezifischen Graphik-API ermöglicht. Über die GUI-Bibliotheken hinaus stehen weitere Klassen für Editoren, Viewer, Problembehandlung, Hilfesysteme, Ressourcenverwaltung, verschiedene Assistenten und Wizards zur Verwendung in eigenen Anwendungen bereit.

2.5.1 Plugin-Konzept

Ein sehr kleiner Eclipse-Kern hat nur die Aufgabe alle vorhandenen Plugins zur Ausführung zu bringen. Die gesamte übrige Funktionalität von Eclipse wird dann über Plugins bereitgestellt. Ein Plugin dient zur Kapselung einer bestimmten Funktionalität und ist in der Regel ein Java Archiv (JAR) mit einem Manifest. Das Manifest beschreibt die Konfiguration des Plugins und die Integration in die Plattform. Ein Kernkonzept der Eclipse-Architektur sind die Extension Points (Erweiterungspunkte). Damit spezifiziert ein Plugin Erweiterungsmöglichkeiten für andere Plugins. Mit Version 3.0 wurde Eclipse auf die offene Ablaufplattform OSGI umgestellt [44]. Damit sind alle Plugins OSGI-Bundles und können dynamisch zur Plattform hinzugefügt oder auch entfernt werden. Mit diesem Plugin-Konzept können nun Rich Client Plattform (RCP) Anwendungen durch die Bereitstellung eigener Plugins mit eigener Funktionalität erstellt werden.

In Abbildung 6 werden die wesentlichen Komponenten von Eclipse gezeigt.

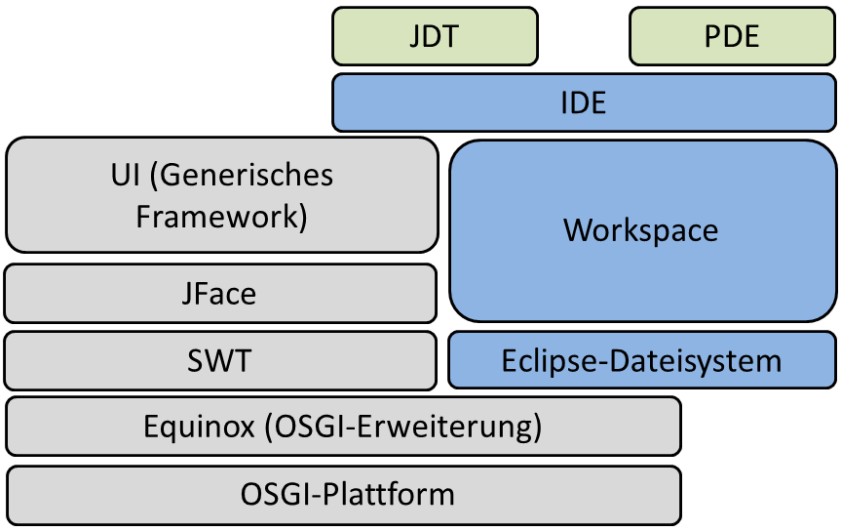


Abbildung 6: Wesentliche Eclipse Komponenten

Die Komponenten UI, JFace, SWT, Equinox und OSGI gehören zur RCP-Plattform. Die Komponenten Eclipse-Dateisystem, Workspace und IDE sind Teil der integrierten Entwicklungsumgebung. Sie erweitern die RCP-Plattform und sind erstmal programmiersprachenunabhängig. Die folgenden Komponenten gehören zur Java Entwicklungsumgebung: Java Development Tools (JDT) und Plugin Development Environment (PDE). Für die integrierte Entwicklungsumgebung gibt es für andere Sprachen wie zum Beispiel C++, Cobol oder Python ebenfalls entsprechende Erweiterungen.

2.5.2 Java Entwicklungsumgebung

Die integrierte Java Entwicklungsumgebung erlaubt die Verwaltung von einem oder mehreren Softwareprojekten. Die Entwicklungsumgebung unterstützt den Entwickler beim Erstellen und Pflegen des Quellcodes mit einem speziell auf Java zugeschnittenen Editor und mit einer entsprechenden Dateiverwaltung. Es wird Funktionalität für das Bauen, Ausführen und Testen der Anwendung und für die Fehlersuche (Debuggen) bereitgestellt. Für die Entwicklung im Team ermöglicht die Entwicklungsumgebung die Integration eines Software Configuration Management (SCM) Repositories. Die lokale Historie von Eclipse erlaubt es die letzten lokalen Änderungen an einer Java-Datei nachzuvollziehen. Mit jedem Speichern im Editor speichert Eclipse die alte Version der Java Datei lokal ab. Die verschiedenen Versionen können dann miteinander verglichen werden.

2.5.2.1 Workspace

Im Workspace werden alle Ressourcen der Java Projekte abgelegt. Es gibt drei verschiedene Ressourcentypen, die erstmal Java unabhängig sind:

- Projekte: Ein Projekt bildet jeweils das Wurzelverzeichnis für einen Verzeichnisbaum. Projekte können Verzeichnisse und Dateien enthalten, jedoch keine verschachtelten Projekte. Es gibt verschiedene Projektnaturen, die festlegen wofür das Projekt verwendet wird. In der Java Entwicklungsumgebung ist ein Projekt zum Beispiel ein Plugin.
- Verzeichnisse: Ein Verzeichnis enthält andere Verzeichnisse und Dateien. Verzeichnisse können beliebig tief verschachtelt werden.
- Dateien: Eine Datei kann keine weiteren Ressourcen enthalten. In der Java Entwicklungsumgebung ist eine Datei zum Beispiel eine Java Quellcode-Datei.

Eclipse setzt bei der Realisierung des Workspace ein eigenes Dateisystem ein. Die Standardimplementierung bildet die Ressourcen des Workspaces direkt auf das native Dateisystem ab. Projekte und Verzeichnisse werden auf Verzeichnisse und Dateien auf Dateien abgebildet. Eclipse stellt entsprechende Interfaces bereit, um eigene Dateisysteme implementieren zu können.

2.5.2.2 Java Development Tools (JDT)

Die JDT besteht unter anderem aus den folgenden Bestandteilen [45]:

JDT Core

JDT Core beinhaltet die Funktionalität ohne Benutzeroberfläche wie zum Beispiel den inkrementellen Java Builder, Code Assistenten oder eine indexbasierte Suchinfrastruktur. Es wird auch ein Java Modell für eine Java zentrierte Ansicht auf das Projekt bereitgestellt. Dazu gehören Artefakte wie zum Beispiel Pakete, Compilation Units, Typen, Methoden oder Felder. Es gibt auch einen vollständigen Abstract Syntax Tree (AST), der für die Refaktorierung (englisch refactoring) [46] und das Übersetzen des Quellcodes und das Finden und Anzeigen von Syntax Fehlern verwendet wird.

JDT Debug

JDT Debug implementiert den Java Debugging Support. Es stellt Funktionaltäten bereit wie zum Beispiel das Starten der virtuellen Maschine (VM) oder die Auswertung von Ausdrücken.

JDT Text

JDT Text stellt den Java Editor unter anderem mit folgende Features zur Verfügung:

- Syntaxhervorhebung verbessert die Lesbarkeit des Quellcodes.
- Syntaxfehler werden unmittelbar während des Editierens im Quellcode angezeigt.
- Hover-Info zeigt für den Text oder einem Marker über dem sich die Maus gerade befindet zusätzliche Informationen in einem Popup-Fenster an.
- Der Inhaltsassistent erlaubt es mit der Eingabe weniger Zeichen und der anschließenden Auswahl aus einer Liste ein Quellcode-Template auszuwählen und in den Text einzufügen.
- Der Korrektur-Assistent schlägt in einer Liste mögliche Korrekturen für einen Syntaxfehler vor.
- Der Quick-Assistent stellt abhängig vom Kontext nützliche Funktionen für die Programmgestaltung bereit (zum Beispiel das Einfügen eines Else-Blocks für eine If-Anweisung).
- Navigationshilfen ermöglichen den Entwickler sich schnell im Quellcode zu bewegen und abhängigen oder relevanten Quellcode zu finden (zum Beispiel die Methodendeklaration zu einem Methodenaufruf).
- Verschiedene Funktionalitäten für die Refaktorierung erlauben es den Quellcode neu zu strukturieren (zum Beispiel eine Klasse umzubenennen oder zu verschieben).
- Über die Undo- und Redo-Funktionalität können Änderungen im Editor wieder zurückgenommen bzw. wiederholt werden.

JDT UI

JDT UI stellt weitere Benutzeroberflächenkomponenten bereit:

- Der Paket-Explorer zeigt alle im Workspace vorhandenen Projekte mit deren Inhalt an.
- Die Typhierarchie-Ansicht zeigt die Ableitungshierarchie einer Klasse oder eines Interfaces an.
- Die Java Outline Ansicht zeigt den Inhalt einer Java Datei in einem Baum als Java Modell an.
- Verschiedene Wizards ermöglichen das Anlegen von Java Elementen wie zum Beispiel einer Java Klasse.

2.5.3 Mylyn

Mylyn [47] ist ein Aufgabenmanagement Tool für Softwareentwickler, das in Eclipse integriert ist. Es gestattet den Entwicklern in einer Ansicht mit einer Aufgabenliste beliebige viele Arbeitsaufgaben (Tasks) zu definieren. Mit Hilfe der Attribute Prioritäten, Anfangszeiten, Fälligkeitsdatum und geschätzte Dauer können die Arbeitsaufgaben auch zeitlich eingeplant werden. Die Arbeitsaufgaben können nach Kategorien oder Anfangszeiten gruppiert werden und können auch aktiviert werden. Während der Aktivierung werden Informationen über die Benutzung der Artefakte (wie zum Beispiel Java-Dateien) aufgezeichnet und dann entsprechend in der Benutzeroberfläche dargestellt: nicht verwendete Artefakte erscheinen grau, selten verwendete in schwarz und häufig verwendete werden fett ausgegeben.

Die Aufgaben können entweder lokal verwaltet werden oder werden aus einem Application Lifecycle Management Repository oder Bugtracker Wergzeugen importiert. Mylyn stellt eine entsprechende API zur Verfügung um eigene Systeme anbinden zu können.

2.6 PREEvision

PREEvision [21] [48] ist ein System- und Entwicklungswerkzeug (CASE-Tool) zur rechnergestützten Entwicklung von Elektrik/Elektronik-Architekturen in der Automobilindustrie. Integraler Bestandteil ist ein Meta- und Datenmodell, welches dafür genutzt wird, alle in der Entwicklung anfallenden Artefakte zu persistieren und konsistent zu halten. Ziel von PREEvision ist die Bereitstellung eines zentralen Repositories für alle Daten aus der E/E-Entwicklung. Ein wichtiger Grund dafür ist die Möglichkeit der Traceability zwischen verschiedenen Artefakten.

2.6.1 Metamodell

PREEvision stellt ein Metamodell für die E/E-Entwicklung in der Automobilbranche zur Verfügung. Das Metamodell ist in verschiedene Ebenen unterteilt (siehe Abbildung 7).

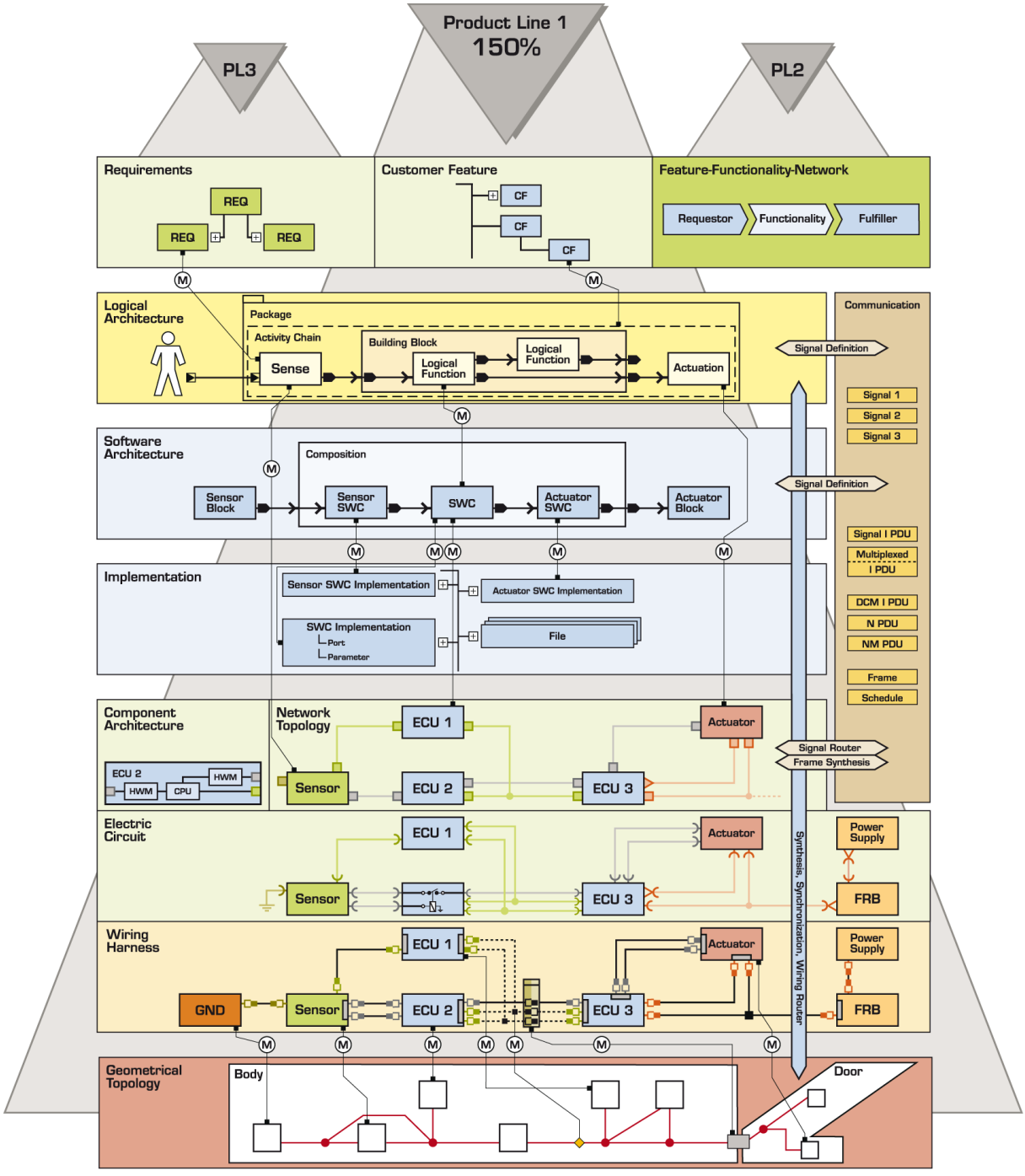


Abbildung 7: Ebenen-Modell von PREEvision

Folgende Hauptebenen sind im Metamodell von PREEvision vorhanden:

1. **Produktziele**: Diese Ebene unterstützt das Anlegen und Pflegen von Kunden erlebbaren Merkmalen (Varianten) und Anforderungen an das System. Diese Artefakte könne über ein Mapping mit anderen Artefakten verlinkt werden für eine durchgängige Traceability. Die Anforderungen und Features werden nicht als vollständige Dokumente abgespeichert, sondern in einzelne feingranulare Artefakte aufgeteilt. Über die Reportfunktionalität von PREEvision können diese dann wieder zu vollständigen Dokumenten zusammengefasst werden.
2. **Logische Architektur**: Diese Ebene erlaubt die Erstellung eines logischen System-Designs der Fahrzeugfunktionen unabhängig davon ob die Funktion in Software oder Hardware implementiert wird.
3. **Software Architektur**: Diese Ebenen spezifiziert Software Komponenten entsprechend dem AUTOSAR Standard [49]. Die Komponenten werden dann zu einem späteren Zeitpunkt den Steuergeräten zugeordnet, auf denen sie ausgeführt werden sollen.
4. **Implementierung:** In dieser Ebene werden der Quellcode und die Modelle für die Generierung des Quellcodes abgelegt.
5. **Hardware Architektur:** Diese Ebene unterstützt die Modellierung von Hardware-Komponenten und deren Verbindungen untereinander. Diese Verbindungen beinhalten logische Verbindungen auf der höchsten Abstraktionsebene bis hin zu Kabelbaumverbindungen auf der konkretesten Ebene.
6. **Geometrie**: Diese Ebene erlaubt die Modellierung des geometrischen Layouts für das System in zwei und dreidimensionalen Koordinaten.

Neben den für E/E Architektur wichtigen Anteilen gibt es weitere Metamodellanteile, die den Entwicklungsprozess unterstützen und folgende Prozessgebiete abdecken:

1. **Testdatenmanagement**: Dieses Metamodell unterstützt die Verwaltung von Testprojekten. Es können Tests spezifiziert, die Tests als manuelle oder automatische Tests implementiert und die Ausführung des Tests dokumentiert werden.
2. **Änderungsmanagement**: Über die Klasse Tickets können Fehler und Änderungswünsche im Modell abgelegt werden und die Bearbeitung und Lösung der Tickets dokumentiert werden.
3. **Projektmanagement**: Es wird die Verwaltung von Projekten mit Projektplänen unterstützt. In einem Projektplan können Meilensteine und Arbeitspakete definiert

werden. Die Arbeitspakete können dann Personen zur Abarbeitung zugeordnet werden.

Für viele dieser Klassen gibt es eine Reihe von Assoziationen und Mappings zu anderen Artefakten um die Traceability zu gewährleisten.

Da im Metamodell Mehrfachvererbung unterstützt wird, werden generische Konzepte im Metamodell von PREEvision oft in abstrakten Basisklassen definiert, wobei die entsprechende Businesslogik für das Konzept nur einmal auf diesen Basisklassen bereitgestellt werden muss. Bei der Definition einer Metaklasse können dann über die Auswahl der Basisklassen die zu unterstützenden Konzepte definiert werden. Zum Beispiel gibt es die Basisklasse *AbstraceNameAttributeArtefact*, die nur ein Attribut für den Namen eines Artefakts definiert. In der Benutzeroberfläche wird bei der Anzeige des Artefakts über die Basisklasse der Inhalt dieses Attributes genutzt. Auch im Attributeditor wird nur bei Vorhandensein der Basisklasse ein entsprechendes Eingabefeld für den Namen angeboten.

2.6.2 Metamodellierungsframework

Zum Metamodellierungsframework von PREEvision gehört das Programm MMEdit. MMEdit stellt eine graphische Benutzeroberfläche mit einem Diagrammeditor zur Verfügung um ein Metamodell (M2) in MOF (M3) definieren zu können (siehe Kapitel 2.3). Damit können Metaklassen mit ihren Basisklassen, Attributen, Kompositionen und Assoziationen definiert werden. Alle Metamodellabbildungen in dieser Arbeit wurden mit diesem Werkzeug erstellt. Zu dem Programm gehört ein Generator, der Quellcode aus dem Metamodell generiert: Meta Data Framework (MDF). MDF bildet Klassen, Attribute, Relationen, Aufzählungstypen auf Java-Techniken ab und basiert auf dem MOF Standard. Das generierte Framework mit dem Quellcode beinhaltet dabei noch eine Reihe weiterer Features:

- Jede Instanz einer Metaklasse besitzt eine XmiID und eine Universally Unique Identifier (UUID). Die XmiID ist innerhalb des Modells eindeutig und die UUID ist eine weltweit eindeutige ID.
- Alle Änderungen an dem Modell werden innerhalb von Operationen durchgeführt. Bei einem Fehler während der Ausführung der Operation werden bereits durchgeführte Änderungen wieder zurückgerollt, um sicherzustellen, dass das Modell weiterhin syntaktisch Korrekt ist. Am Ende der Operation wird auch die syntaktische Korrektheit des Modells entsprechend dem Metamodell geprüft. Die Operationen

können außerdem über eine Rückgängig-Funktion (englisch undo) vom Anwender wieder rückgängig gemacht werden.

- Bei Änderungen am Modell werden Events gefeuert. Über Event-Listeners kann auf diese Events entsprechend reagiert werden.
- Teil des Frameworks ist ein Rechtemanagement. Hier können Lese- und Schreibrechte im Modell für Personen oder Rollen vergeben werden. Es sind auch Rechte auf Metaklassen möglich, um zum Beispiel alle Instanzen einer Klasse für eine Rolle unsichtbar zu machen. Die notwendigen Informationen werden in einem Feature- und Authority-Modell verwaltet.
- Im Metamodell gibt es eine spezielle Metaklasse, die die Ablage von Dateien im Modell unterstützt. Im Modell wird die Datei über eine Instanz dieser Metaklasse repräsentiert, die Informationen über die Datei wie zum Beispiel Name, Erstellungs- und Änderungsdatum und Checksumme bereitstellt. Über dieses Artefakt kann auf den Dateiinhalt zugegriffen werden.

2.6.3 Architektur

Eine wichtige Funktion von PREEvision ist die Möglichkeit das Modell in einer kollaborativen Umgebung zur Verfügung zu stellen. Das Modell wird in einer Datenbank persistiert und an die Anwender verteilt. Der Anwender kann Änderungen am Modell durchführen und die Änderungen in die Datenbank speichern. Dies geschieht mittels eines dreischichtigen (3-tier) Systems (siehe Abbildung 8).

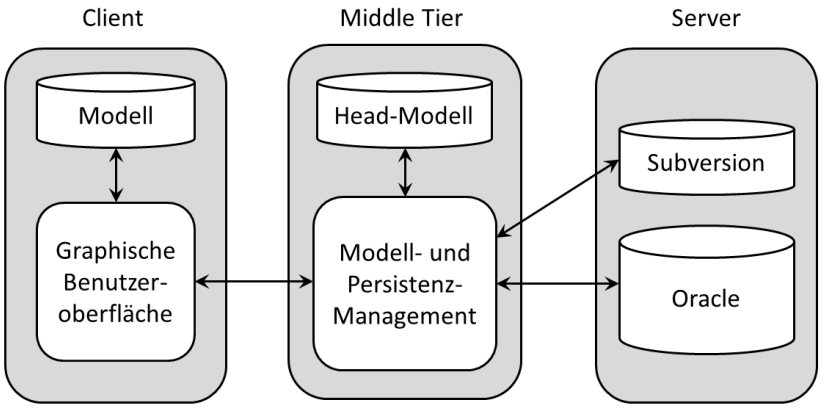


Abbildung 8: Systemübersicht PREEvision

Der Client basiert auf einer Eclipse Rich Client Plattform (RCP) (siehe Kapitel 2.5.1). Es kommen zusätzlich auch Komponenten aus der Eclipse Java Entwicklungsumgebung zum Einsatz. Der Client stellt eine graphische Benutzeroberfläche bereit, über die das Modell

gesichtet und bearbeitet werden kann. Das Modell ist vollständig oder in Teilen am Client vorhanden und wird auf der Festplatte zwischengespeichert. Bis auf die erstmalige Anmeldung müssen daher nur Änderungen am Modell (Deltas) übertragen werden. Beim Abspeichern von lokalen Änderungen in die Datenbank (Commit) und beim aktualisieren des Modells mit Änderungen anderer Anwender (Update) werden jeweils nur Deltas über das Netzwerk zwischen Client und Middle-Tier übertragen.

Auf der Middle-Tier wird die letzte Version (Head) des Modells aus Performancegründen im Arbeitsspeicher des Rechners vorgehalten. Dieses Head-Modell wird auch auf der Festplatte zwischengespeichert. Damit wird die Startzeit des Middle-Tiers deutlich verkürzt, weil das Modell nicht vollständig aus der Datenbank gelesen werden muss. Änderungen am Modell werden von der Middle-Tier in der Datenbank persistiert.

Der Server besteht aus einer Oracle Datenbank [50] für die Modelldaten und einem Subversion Server [23] für die Dateiinhalte der Dateiartefakte im Modell. Ein Subversion Client kann sich auch direkt über die Middle-Tier mit dem Subversion Server verbinden um Dateien zu lesen und zu schreiben. Änderungen wie zum Beispiel das Umbenennen oder Neuanlegen von Dateiartefakte werden dabei automatisch in das Modell synchronisiert.

2.6.4 Benutzeroberfläche

Die im Client bereitgestellte graphische Benutzeroberfläche stellt für das Sichten und Ändern des Modells eine Reihe von Ansichten und Editoren zur Verfügung. Die wichtigsten sind folgende (siehe Abbildung 9):

- **Modellansicht**: Diese Ansicht zeigt den Modellbaum mit allen aktuell geladenen Modellartefakten (1).
- **Eigenschaftsansicht**: Hier können auf mehrere Seiten verteilt die Attribute des aktuell selektierten Artefakts eingesehen und verändert werden (2).
- **Diagrammeditoren**: Es werden für verschiedene Anwendungsfälle Diagrammeditoren zur Verfügung gestellt. In der Abbildung ist ein Komponentendiagramm für die Hardware-Architektur zu sehen (3).
- **Favoritenansicht**: Hier hat der Anwender die Möglichkeit, für ihn wichtige Artefakte abzulegen und damit schneller im Zugriff zu haben (4).

- **Tabelleneditoren**: Die Modellartefakte und deren Attribute können über verschiedene Tabellen bearbeiten werden. Das Aussehen und die Inhalte der Tabellen können im Modell konfiguriert werden.

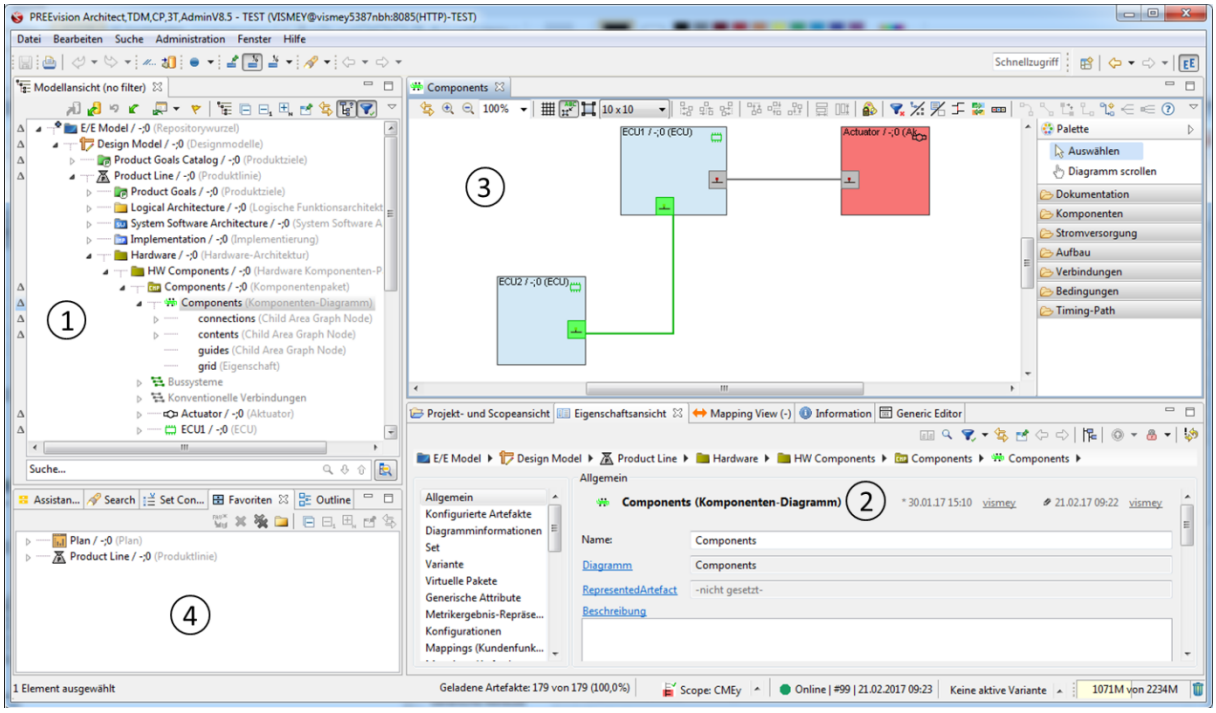


Abbildung 9: Graphische Benutzeroberfläche von PREEvision

2.6.5 Versionierung

Das Modell wird mit der vollständigen Historie in der Datenbank abgespeichert. Jede in die Datenbank abgespeichert Änderung des Modells (Commit) wird dabei als Differenz zum vorhergehenden Stand abgelegt. Mit der Abgabe wird eine sogenannte Versionsnummer für das Gesamtmodell hochgezählt. Die Änderungen im Modell zwischen zwei Versionsnummern kann sehr schnell berechnet werden zum Beispiel für die Bereitstellung eines Updates für ein lokales Modell auf dem Rechner eines Anwenders. Jedes Artefakt hat damit eine Abgabe-Historie (englisch commit history) und die Differenz zwischen zwei Abgabe-Ständen kann vom PREEvision Client für ein Artefakt abgefragt werden.

Darüber hinaus kann ein Artefakt mit allen Kind-Artefakten über ein sogenanntes Check-in eingefroren werden. Damit sind keine Änderungen an dem Artefakt mehr möglich, bis das Artefakt wieder ausgecheckt wird. Das Check-in kann dazu verwendet werden, um ein oder mehr Artefakte zu releasen. Mit dem Auschecken des Artefakts für die Weiterentwicklung bekommt das Artefakt eine neue Revisionsnummer. Das Artefakt hat damit eine Revisionshistorie. Jede Revision im Modell kann vom Client abgefragt und geladen werden.

Alternativ zum auschecken in einer neuen Revision kann das Artefakt in einem neuen Entwicklungszweig (Branch) weiterentwickelt werden. Für den neuen Entwicklungszweig wird dann ein eigener Branch-Name festgelegt. Dieser Entwicklungszweig kann durch ein- und auschecken ebenfalls weiterentwickelt werden, wobei die Revisionsnummer entsprechend hochgezählt wird.

2.6.6 Bearbeitung des Modells

Um das Modell einsehen und bearbeiten zu können muss der Anwender sich nach dem Start von PREEvision über einen Benutzernamen und ein Passwort authentifizieren. In einem eigenen Authority-Modell werden die Benutzer von PREEvision mit ihren Rechten und Rollen verwaltet. Damit ist es möglich einzelnen Anwendern oder Anwendergruppen Lese – und Schreibrechte für beliebige Teile des Modells zu entziehen. Nach dem erstmaligen Anmelden wird das Modell von der Middle-Tier entsprechend den Leserechten des Anwenders in den PREEvision Client geladen. Hier kann der Anwender das Modell über Editoren, Tabellen und Diagrammen bearbeiten. Ein Artefakt kann dabei nur bearbeitet werden, wenn es ausgecheckt ist. PREEvision unterstützt feingranulare, pessimistische Sperren (Locks) um die Notwendigkeit des Verschmelzens von Änderungen an gleichen Artefakten durch unterschiedliche Anwender zu verhindern. Ein oder mehrere Modellartefakte können gesperrt werden, um einen exklusiven Zugriff zu ermöglichen. Diese Sperren können entweder explizit durch den Anwender über das Kontextmenü angefordert werden oder sie werden automatisch vom Middle-Tier bezogen, abhängig von der gerade durchgeführten Änderung des Anwenders.

In PREEvision gibt es drei verschiedene Arten von Sperren:

- Exklusive Sperren: Nur ein Anwender kann eine exklusive Sperre auf einem Artefakt besitzen. Eine weitere exklusive Sperre eines anderen Anwenders ist nicht möglich.
- Geteilte Sperren: Ein oder mehr Anwender können eine geteilte Sperre auf einem Artefakt besitzen. Eine exklusive Sperre auf einem Artefakt mit einer geteilten Sperre ist nicht möglich.
- Exklusive Baum-Sperre: Eine exklusive Baum-Sperre sperrt einen kompletten Teil-Baum für die exklusive Bearbeitung.

Die Sperren werden durch „Bleistift“-Symbole in der Benutzeroberfläche sichtbar gemacht (siehe Abbildung 10). Die grüne Farbe steht für eine eigene Sperre und die rote Farbe für Sperren anderer Anwender (fremde Sperren).

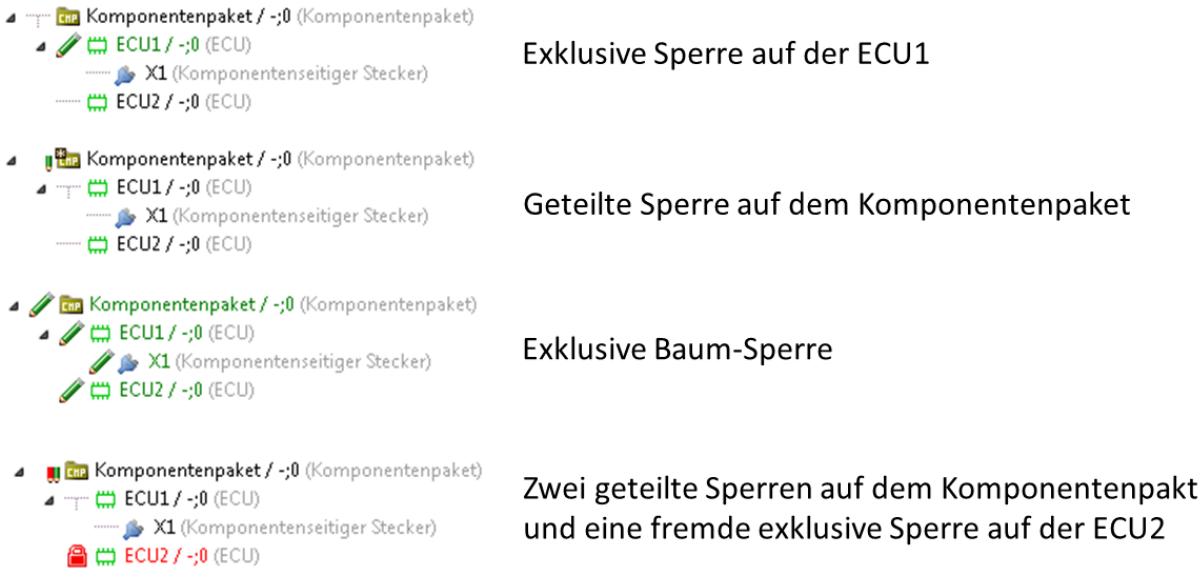


Abbildung 10: Verschieden Arten von Sperren in PREEvision

Bei der Anfrage für eine Sperre werden folgende Aktionen durchgeführt:

1. Der PREEvision Client sendet für ein oder mehr Artefakte (mittels der XmiIDs) eine Sperranfrage an die Middle-Tier.
2. Die Middle-Tier prüft ob es schon Sperren für diese Artefakte gibt. Dazu wird eine Tabelle in der Datenbank abgefragt, wo alle Sperren abgespeichert sind.
3. Wenn ein oder mehr Artefakte schon eine Sperre besitzen, wird mit einer Fehlermeldung auf die Anfrage geantwortet.
4. Wenn die Sperren möglich sind, dann wird in die Datenbank für jede Sperre folgende Informationen abgelegt: XmiID des gesperrten Artefakts, Zeitpunkt der Sperre, Anwender, dem die Sperre gehört und die Art der Sperre.
5. Das erfolgreiche Sperren wird an den Client zurückgemeldet. Außerdem werden alle aktuellen Sperren aller anderen Anwender zurückgegeben. Damit kann der Anwender sehen, welche Artefakte neben seinen eigenen noch gesperrt sind. Es wird zusätzlich ein Update auf die neueste Modellversion bereitgestellt. Das Update stellt sicher, dass die anschließenden Änderungen des Entwicklers wirklich auf der neuesten Version der Artefakte erfolgt und nicht auf einem älteren Stand. Ansonsten könnten Änderungen anderer Entwickler überschrieben werden.

Wenn das Artefakt exklusiv gesperrt ist, kann es vom Anwender bearbeitet werden. Sperren können vom Administrator wieder aufgehoben werden. Das bedeutet aber auch, dass bereits eventuell durchgeführte Änderungen auf Artefakten mit diesen Sperren für den Anwender verloren gehen. Der Anwender kann seine eigenen Sperren wieder aufheben, wodurch seine Änderungen ebenfalls zurückgerollt werden. Zu einer fremden Sperre kann die Information abgerufen werden, wem die Sperre gehört und seit wann die Sperre schon existiert.

Die aktuellen Änderungen können auf die lokale Festplatte gespeichert werden (Save) und damit kann PREEvision jederzeit beendet werden, ohne dass Änderungen verloren gehen. Beim Beenden von PREEvision wird das vollständig geladene Modell auf die Festplatte gespeichert. Beim nächsten Start muss dann das Modell nicht mehr von der Middle-Tier geladen werden, sondern kann von der Festplatte bezogen werden. Der Anwender entscheidet schlussendlich, wann er seine Änderungen in die Datenbank abspeichern will (Commit) und damit die Änderungen für alle anderen Entwickler sichtbar werden. Über einen Abgabe-Dialog kann ein Abgabe-Kommentar vergeben werden. Alle Sperren des Anwenders werden mit dem Abgeben wieder freigegeben. Beim Abgeben werden nur die durchgeführten Änderungen als Delta an die Middle-Tier übertragen (siehe Kapitel 2.6.3). In Abbildung 11 sind die einzelnen Schritte nochmals dargestellt.

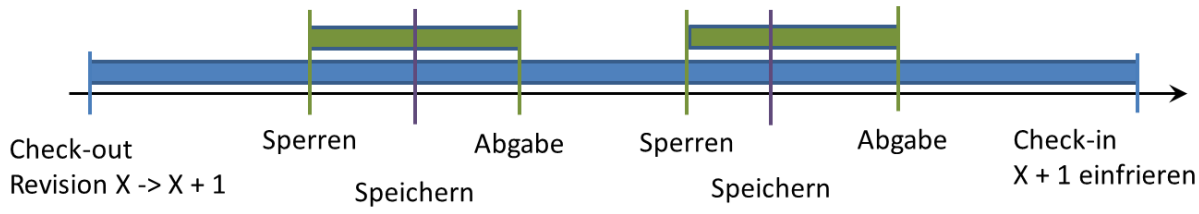


Abbildung 11: Beispiel für die Bearbeitung eines Modells in PREEvision

Eine Aktualisierung des Modells kann ebenfalls jederzeit vom Anwender angestoßen werden. Durch die automatische Aktualisierung mit den angeforderten Sperren arbeitet der Entwickler in der Regel immer mit einem relativen aktuellen Modellstand.

2.6.7 Lifecycle-Management

Im Authority-Modell gibt es die Möglichkeit ein oder mehr Lifecycles für Metaklassen zu definieren. Der Lifecycle ist ein zusätzliches Attribut auf der Instanz der Metaklasse, dass entsprechend dem im Authority-Modell definierten Zustandsautomaten auf verschiedene

Lifecycle-Zustände gesetzt werden kann. In Abbildung 12 ist ein Beispiel für einen Lifecycle für ein Ticket zu sehen.

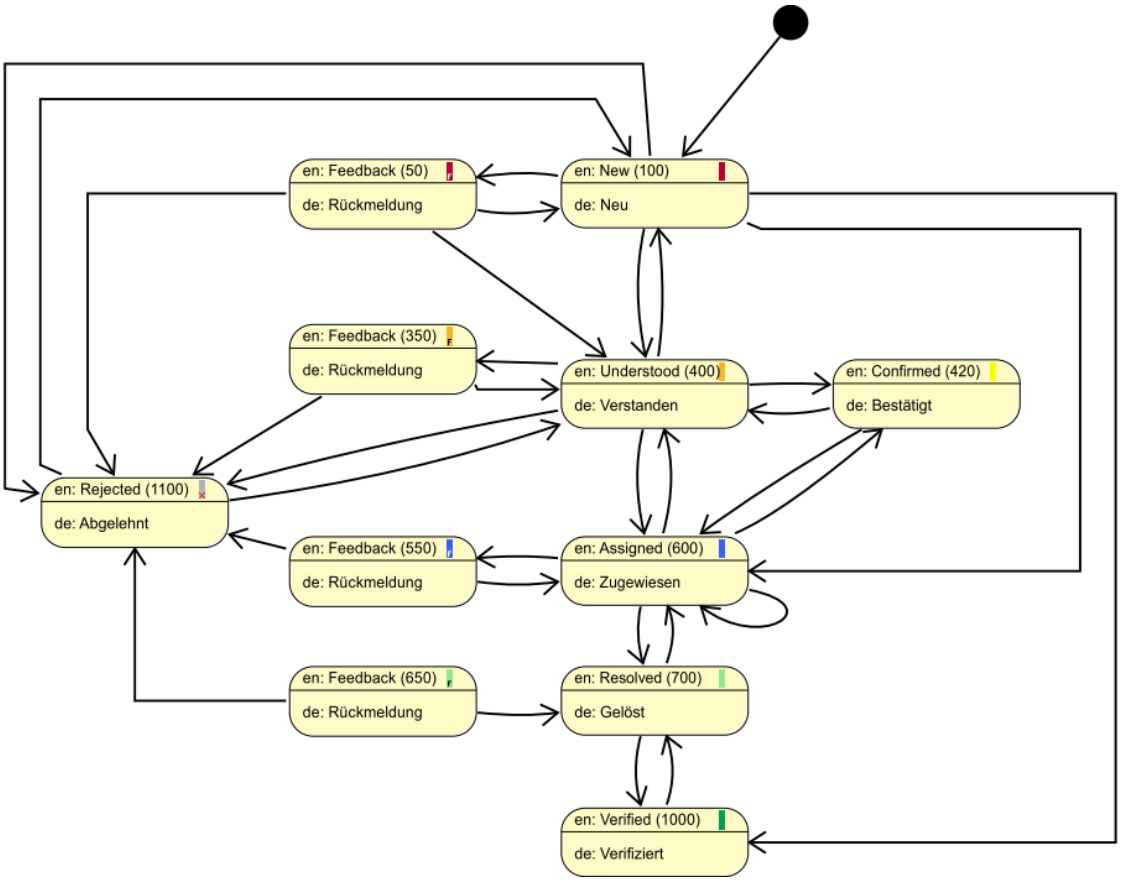


Abbildung 12: Beispiel für einen Lifecycle im PREEvision

2.7 Continuous Integration

Continuous Integration (CI) ist eine Praktik von Extreme Programming (XP) [2] [51]. XP ist ein Vorgehensmodell der Softwaretechnik und gehört zu einer weit verbreiteten agilen Methode. Wichtig sind Werte wie Simplifikation, Kommunikation und Feedback. Außerdem stehen kurze Iterationszyklen im Vordergrund. Im Gegensatz zum Wasserfall oder iterativen Vorgehen werden bei XP nur kleine Feature Umfänge definiert und zwar Feature Umfänge, die den größten Mehrwert für den Kunden bieten. Diese werden innerhalb von wenigen Wochen umgesetzt. Danach erfolgt ein Review der umgesetzten Feature durch den Kunden und so hat der Kunde die Möglichkeit korrigierend und sehr agil auf die Entwicklung Einfluss zu nehmen.

Es kommen 12 Kernpraktiken zum Einsatz (siehe Abbildung 13).

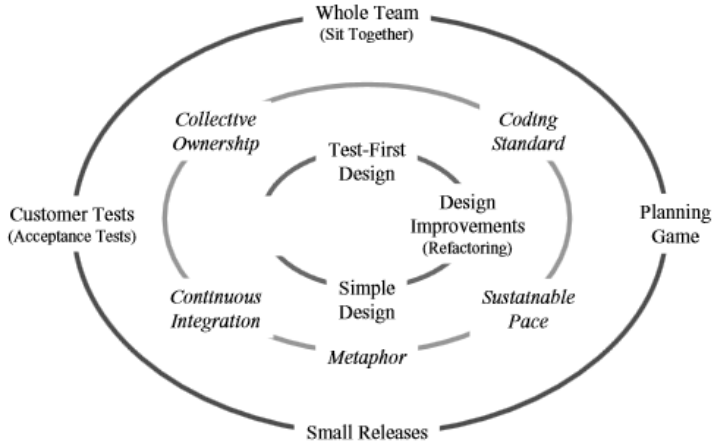


Abbildung 13: Die 12 Kernpraktiken von XP [2]

Der innere Kreis beschreibt die Aktivitäten der Programmierer. Die Praktiken des mittleren Kreises helfen der Teamkommunikation und der Koordination der Auslieferung. Im äußeren Kreis geht es um die Abstimmung zwischen Kunde und Programmierer.

CI unterstützt XP dahingehend, dass es in regelmäßigen und kurzen Intervallen Produkt-Builds zur Verfügung stellt, die validiert und durch automatische Tests getestet werden können [52] [18]. Anstatt in größeren Intervallen und dann mit großem Aufwand die Änderungen der einzelnen Softwareentwickler zu einem Produkt zu integrieren, werden die Änderungen aller Softwareentwickler täglich zusammengeführt. Softwarefehler und Konflikte können frühzeitig erkannt werden. Damit wird das Risiko minimiert in einem weit fortgeschrittenen Projekt auf Probleme zu stoßen, die womöglich den Release-Termin gefährden könnten.

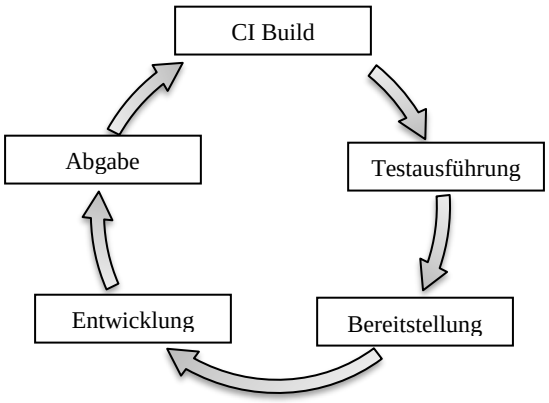


Abbildung 14: Bestandteile des CI

CI erfordert, dass die Softwareentwickler ihre Quellcodeänderungen mehrmals täglich in ein gemeinsames SCM Repository abgeben. Im Anschluss wird eine Version des Softwareprodukts erstellt und mit dieser Version automatische Tests ausgeführt. Die Übersetzung des Quellcodes und die Ausführung der automatischen Tests werden auf eigenen CI Server durchgeführt. In der Regel stehen mehrere CI Server zur Verfügung um parallel mehrere Aufgaben gleichzeitig erledigen zu können. Das Produkt kann dann zur Validierung für den Kunden und Produktmanager bereitgestellt werden (siehe Abbildung 14).

Folgende Voraussetzungen müssen für die obengenannten Schritte erfüllt sein:

Bereitstellung eines einzelnen Software Configuration Management (SCM) Repository
Ein Software Projekt beinhaltet eine Vielzahl verschiedener Dateien, die für den Bau des Softwareprodukts benötigt werden. Für das Management und die Speicherung dieser Dateien werden heutzutage SCM Systeme eingesetzt (siehe Kapitel 2.1). Damit CI funktionieren kann, müssen wirklich alle Dateien, die für das Produkt-Build benötigt werden (zum Beispiel Quellcode Dateien, Property Dateien, Skripte zur Erstellung des Datenbankschemas, Installationsskripte, zusätzliche Libraries, usw.) dort abgespeichert und die Änderungen der Dateien regelmäßig in das Repository abgegeben werden.

Automatischer Produkt-Build
Beginnend mit den Dateien im SCM Repository muss über einen einzigen Befehl ein vollständiges Produkt gebaut werden können. Um dann automatische Tests ausführen zu können, muss es möglich sein wieder über einen einzigen Befehl alle ausführbaren Programme zu starten (zum Beispiel Client und Middle-Tier) und automatisch ein Datenbankschema einzuspielen. Damit ist jeder in der Lage das komplette Produkt zu bauen und zu starten auf einem Computer, der keine besonderen Vorbedingungen erfüllen muss (zum Beispiel vorinstallierte Libraries).

Entwicklung von automatischen Tests
Um innerhalb CI das Produkt automatisch testen zu können, müssen von den Softwareentwicklern automatische Tests bereitgestellt werden, die möglichst alle Produktfeatures testen. Eine Praktik von XP ist die Test-Driven Entwicklung. Hier wird schon im Design Wert darauf gelegt, dass die erstellte Software über automatische Tests getestet werden kann. Die Softwareentwickler entwickeln zuerst den Test und schreiben dann den passenden Quellcode, sodass der Test ohne Fehler durchläuft. Häufig werden dazu Unit-Tests

und Komponententests entwickelt. Es werden zusätzlich Systemtests benötigt um das Zusammenspiel aller Komponenten testen zu können.

Softwareentwickler geben ihre Änderung jeden Tag ab

Damit ist es möglich sehr schnell Konflikte, die durch das Bearbeiten desselben Quellcodes entstehen, zu erkennen und frühzeitig zu lösen. Die Ergebnisse der automatischen Tests zeigen dem Softwareentwickler schon während der Entwicklung der neuen Funktionalität ob er bestehende Funktionalität beschädigt hat. Die regelmäßige Abgabe der Änderungen ermutigen den Entwickler seine Arbeit in kleine Einheiten von wenigen Stunden aufzuteilen.

Ausführung der Tests in einem Klon der Produkt Umgebung

Die Laufzeitumgebung kann das Verhalten der Software beeinflussen (zum Beispiel Spracheinstellung, Betriebssystemversion, Hardware, Ports). Daher sollten die Tests in einer Laufzeitumgebung ausgeführt werden, die dem Kunden möglichst nahekommt. Damit können die Tests Softwarefehler aufdecken, die nur in der Laufzeitumgebung des Kunden auftreten. Wenn die Variationen der Laufzeitumgebungen bei verschiedenen Kunden zu groß sind, kann das natürlich nur teilweise umgesetzt werden.

Einfacher Zugriff auf die vom CI erstellten Produktversionen

Damit Produktmanager und andere Stakeholder den Fortschritt der Entwicklung verfolgen und validieren können, muss ein einfacher Zugriff auf die letzte vom CI gebaute Produktversion möglich sein. Damit sind Fehlentwicklungen sehr schnell erkennbar und es können korrigierende Maßnahmen ergriffen und Spezifikationen korrigiert werden. Oft ist es viel einfacher an Hand einer lauffähigen Software an statt einer schriftlichen Spezifikation zu beurteilen, ob die Software die Anforderungen des Kunden erfüllt.

Jeder kann den Status des CI einsehen

Kommunikation ist im CI sehr wichtig und so muss jeder in der Lage sein den aktuellen Status des CI zu sehen. Meistens wird über eine Web-Seite angezeigt, ob gerade ein CI Lauf ausgeführt wird, ob es aktuell Compilerfehler gibt und wie die Ergebnisse der letzten Testläufe aussehen. Über entsprechende Farben (zum Beispiel Rot, Gelb und Grün) wird der Status des Produkts sofort sichtbar. Außerdem kann über die Webseiten eingesehen werden, welche Änderungen in der letzten Zeit durchgeführt wurden. Ein Verlauf der Anzahl der fehlgeschlagenen Tests über die Zeit liefert dem Management wichtige Hinweise über den Zustand des Produkts.

3 Traceability

3.1 Einführung

Traceability Links unterstützen den Softwareentwickler beim Verstehen des Quellcodes und beim Erkennen der Zusammenhänge zwischen prozessrelevanten Artefakten und dem Quellcode [53] (siehe Kapitel 1.2.1 und 2.2). Links von der Anforderung in den Quellcode können unter anderem folgende Fragen beantworten [32]: Wo finde ich den Quellcode, der diese Anforderung umsetzt? Haben alle funktionalen Anforderungen eine konkrete Implementierung? Warum wurde dieser Quellcode entwickelt?

Weitere Fragen, die sich im Zusammenhang mit Artefakten aus anderen Prozessgebieten stellen, sind folgende: Welche Softwarefehler wurden in dieser Anweisung im Quellcode schon gelöst (Änderungsmanagement)? Wo kann ich den Quellcode für den automatischen Test finden, der diese Anforderung testet? Welche Anforderungen testet dieser Quellcode (Testdatenmanagement)? Gibt es irgendwelche zusätzliche Dokumente oder UML Diagramme [40] für diese Klasse (Design und Dokumentenmanagement) [54]?

Neben dem Grund die Verständlichkeit des Quellcodes für den Softwareentwickler zu erhöhen, gibt es verschiedene Sicherheitsstandards, die Traceability zwischen den Artefakten im Prozess erfordern [55]. Traceability ermöglicht eine Wirkungsanalyse, die bis in den Quellcode hinein möglich ist. Es kann überprüft und nachgewiesen werden, dass jede Anforderung implementiert und getestet wurde. Diese Sicherheitsstandards (zum Beispiel ISO 26262 [56]) kommen vor allem in sicherheitskritischen Domänen wie zum Beispiel Medizintechnik, Flugzeugindustrie oder in der Automobileindustrie zum Einsatz.

Einige kommerzielle Application Lifecycle Management (ALM) Lösungen (zum Beispiel Polarion Software [57], Rational Team Concert [58] oder Microsoft Team Foundation Server [59]) integrieren bereits mehrere Prozessgebiete in einer Anwendung und in ein Repository. Diese Anwendungen unterstützen das Anlegen und Pflegen von Traceability Links zwischen Artefakten unterschiedlicher Prozessgebiete (zum Beispiel zwischen Testdatenmanagement und Anforderungsmanagement). Aber die Unterstützung von Traceability Links in den Quellcode hinein ist immer noch rudimentär. Üblich ist die Möglichkeit der Verlinkung einer Anforderung mit einer Reihe von Quellcode-Dateien, die für diese Anforderung geändert wurden. Im besten Fall werden Links zu Klassen unterstützt [9]. Ein Grund dafür ist, dass die

kleinste Einheit, die in einem Software Configuration Management (SCM) Repository abgespeichert wird, üblicherweise eine Datei ist, die wiederum eine Klasse enthält (siehe Kapitel 2.1.2). Traceability Links in einen Text sind schwierig zu pflegen, weil durch die Änderung des Texts der Link aufbrechen kann.

Ein Java-Klasse besteht aus Attributen und Methoden. Eine Methode enthält Anweisungen wie zum Beispiel eine „For-Schleife“, „If-Anweisung“ oder ein Methodenaufruf. Zumindest für all diese Artefakte sollten Traceability Links möglich sein. Zusätzlich wäre die Erwartung an ein SCM System, dass die vollständige Historie dieser Artefakte inklusive der Traceability Links dauerhaft archiviert wird und von den Softwareentwicklern abgerufen werden kann (siehe Kapitel 2.1.1). Feingranulare Traceability Links werden unter anderem aus folgenden Gründen benötigt:

- Die Traceability zwischen dem Testfall und der Methode des Unittests im Quellcode ist wichtig aus folgendem Grund: Wenn der Testfall mit der Anforderung verlinkt ist (der Testfall beschreibt wie die Anforderung zu testen ist), dann ist es möglich alle Testmethoden im Quellcode zu finden, die eine bestimmte Anforderung verifiziert.
- Üblicherweise wird nicht nur eine Anforderung in einer Java-Klasse implementiert. Das wäre zu grobgranular. Verschiedene Methoden in einer Klasse können mit unterschiedlichen Anforderungen verlinkt sein. Über die Zeit nach einigen Änderungen und Refaktorierungen [46] des Quellcodes (zum Beispiel verschieben von Anweisungen in eine andere Klasse), ergibt sich die Notwendigkeit für Traceability Links zwischen einzelnen Anweisungen und den Anforderungen. Ansonsten würde die Information verloren gehen, warum eine bestimmte Anweisung entwickelt wurde.
- Der Softwareentwickler ist nicht nur an der letzten Änderung einer Anweisung und dem Grund für diese Änderung interessiert, sondern an allen Anforderungen und Softwarefehlern, die dazu geführt haben, dass diese Anweisung geändert wurde über die vollständige Änderungshistorie. Dabei sollte es keine Rolle spielen, wie oft diese Anweisungen schon zwischen verschiedenen Methoden und Klassen verschoben worden ist.
- Dokumente (zum Beispiel die Beschreibung der Softwarearchitektur oder Konzeptentscheidungen), die mit einem Teil einer Klasse verlinkt sind, können wichtige zusätzliche Informationen für ein besseres Verständnis des Quellcodes liefern. Zum Beispiel kann eine Präsentation, die einen bestimmten Algorithmus

erklärt, direkt mit der „For-Schleife“, die diesen Algorithmus implementiert, verlinkt sein.

3.2 Stand der Technik

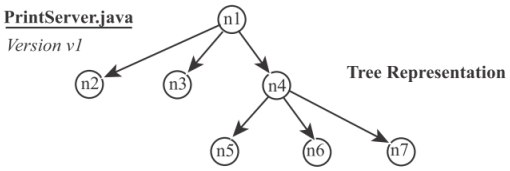
3.2.1 Feingranulare Versionskontrollsysteme

Alle populären und wichtigen Software Configuration Management (SCM) Systeme arbeiten mit Dateien. Aber es gibt einige Forschungsprojekte im Bereich von feingranularen Versionskontrollsystemen (zum Beispiel COOP/Orm [60] oder Sysiphus [61]), einige unterstützen auch Quellcode. Bei diesen Systemen wird der Quellcode in kleinere Einheiten als eine Datei versioniert und in einer Repository abgelegt. Damit besitzen diese Einheiten eine eigene Historie.

3.2.1.1 MolhadoRef

MolhadoRef [62] ist ein semantisch-basiertes und operationsorientiertes SCM für die Programmiersprache Java und für die Entwicklungsumgebung Eclipse. MolhadoRef basiert auf dem Projekt Molhoado, welches ein flexibles Datenmodell zur Verfügung stellt, dass jede Programmiersprache speichern kann. Ein Programm besteht aus einer Menge von Knoten, wobei jedem Knoten wiederum eine Menge von sogenannten Slots zugeordnet ist. Die Knoten haben eine Identität und die Slots nehmen konkrete Werte auf. Die Attribute ordnen die Knoten den Slots zu und werden in Attributtabellen abgespeichert. Dieses Datenmodell wird nicht nur verwendet um den Quellcode zu speichern, sondern auch die Versionshistorie und die Operationen, die den Quellcode verändern. Versionsverwaltung wird dem Datenmodell durch eine dritte Dimension der Attributtabelle hinzugefügt, so dass Slots versioniert werden können. Es ist möglich Entwicklungszweige zu erstellen und zu verschmelzen.

MolhadoRef ist die Spezialisierung für Java. Ein AST (Abstract Syntax Tree) Knoten wird durch einen Molhado Knoten repräsentiert, der mit zwei Attributen assoziiert wird: „parent“ definiert den Vaterknoten und „children“ beinhalten eine Liste von Referenzen auf die Kind-Knoten (siehe Kapitel 2.4). Zusätzlich gibt es noch das Attribut „NodeType“, welches die AST Knotentyp definiert zum Beispiel „MethodDecl“ für eine Methodendeklaration. Darüber hinaus gibt es weitere Attribute abhängig von dem AST Knotentyp zum Beispiel „RetType“ für den Typ des Rückgabewertes einer Methode (siehe Abbildung 15). Die AST wird nur bis zur Methodeneben feingranular abgespeichert. Der Quellcode innerhalb einer Methode wird in einem Attribut der Methode aus Effizienzgründen abgelegt.



Attribute Table

	"NodeType"	"Name"	"parent"	"children"
n1	CompiUnit	"PrintServer"	null	[n2,n3,n4]
n2	PackageDecl	"nodes"	n1	null
n3	ImportDecl	"content.Packet"	n1	null
n4	TypeDecl	"PrintServer"	n1	[n5,n6,n7]
n5	MethodDecl	"print"	n4	null
n6	MethodDecl	"getPacketInfo"	n4	null
n7	MethodDecl	"accept"	n4	null

	"SuperType"	"Modifier"	"RetType"	"Parameters"	"SourceCode"
n1	undef	undef	undef	undef	undef
n2	undef	undef	undef	undef	undef
n3	undef	undef	undef	undef	undef
n4	LANNode	"public"	undef	undef	undef
n5	undef	"public"	"void"	[(Packet, p)]	"String packet..."
n6	undef	"public"	"String"	[(Packet, p)]	"return p.cont..."
n7	undef	"public"	"void"	[(Packet, p)]	"if (p.addresse.."

Abbildung 15: MolhadoRef Repräsentation einer Java Klasse

MolhadoRef geht davon aus, dass der Editor Operationen für die Refaktorierung erkennen kann, die auf den Quellcode ausgeführt werden. In der ersten Abgabe des Quellcodes wird der vollständige AST abgespeichert. Danach werden bei einer Abgabe nur noch die aufgezeichneten Operationen für die Refaktorierung mit anderen Änderungen in das Repository abgespeichert. Bei einer Aktualisierung durch andere Entwickler werden die gespeicherten Operationen für die Refaktorierung auf dem Quellcode wiederholt. Dieser operationsbasierte Ansatz ist sehr exakt, aber eine größere Anzahl von Operationen produziert einen gewissen Overhead für die Aufzeichnung als auch für die Wiederholung der Operationen. Die Operationen können aber andererseits viele Änderungen am Quellcode zusammenfassen zum Beispiel die Umbenennung einer Methode, die dazu führt, dass der Quellcode an vielen Stellen angepasst werden muss.

3.2.1.2 Stellation

Stellation [63] [64] (früher Coven genannt) ist ein hierarchisch-basiertes SCM System, das eine Reihe von Features bereitstellt, um die Zusammenarbeit zu unterstützen. Das System stellt ein Master-Repository zur Verfügung, das für jedes Projektteam repliziert wird. Ein Sub-Repository kann wiederum in weitere Sub-Repositories repliziert werden bis hin zu einem privaten Repository für den Entwickler (hierarchisches System). Änderungen können in ein Sub-Repository abgegeben werden, ohne an andere Repositories weitergegeben zu

werden. Die Repositories tauschen jedoch Information über einen abonnierten Benachrichtigungsmechanismus untereinander aus. Zum Beispiel werden Informationen über Quellcode Änderungen an das Vater-Repository weitergereicht und dieses Repository gibt es dann an die anderen Kind-Repositories weiter. Diese Änderungsinformationen werden bei Stellation Soft-Locks genannt und dienen dazu Softwareentwickler über Änderungen am Quellcode zu informieren. Diese Soft-Locks können mit einem Kommentar versehen werden. Wenn ein anderer Entwickler versucht den Quellcode mit eine Soft-Lock zu verändern, bekommt der Entwickler einen Hinweis mit dem Soft-Lock Kommentar und kann dann selbst entscheiden, ob er die Änderung an dem Quellcode trotzdem durchführt oder ob er den Konflikt vermeiden will. Wenn der Entwickler seine Änderungen released, werden die Änderungen in die anderen Repositories kopiert und die anderen Entwickler bekommen einen Hinweis, dass neue Änderungen zur Synchronisation bereitstehen.

Stellation versioniert auf der Ebene von Fragmenten. Die Größe eines Fragments ist abhängig von der Programmiersprache. In Java und in C++ entspricht die Größe einer vollständigen Methode, Funktion und Variablendeklaration (siehe Abbildung 16).

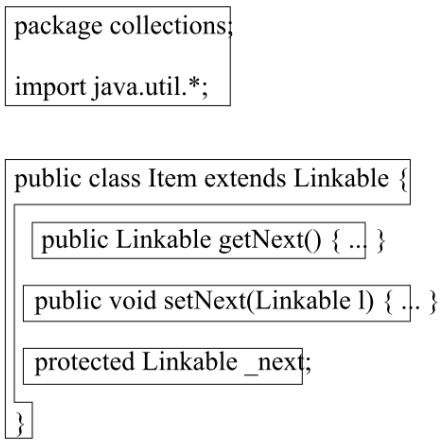


Abbildung 16: Aufteilung von Java Quellcode in versionierte Fragmente [64]

Stellation stellt eine eigene Abfragesprache für verschiedene Anwendungsfälle zur Verfügung, um eine Menge von Fragmenten zu definieren. Anstatt den Entwickler den Quellcode immer in einer strikten, gleichen und dateiorientierten Weise zu präsentieren, können über die Abfragesprache alle für eine bestimmte Aufgabe relevanten Fragmente abgefragt werden und anderer nicht relevanter Quellcode ausgeblendet werden. Damit ergeben sich unterschiedliche Sichten auf den Quellcode, die von Anwendungsfall zu Anwendungsfall unterschiedlich sein können (siehe Abbildung 17).

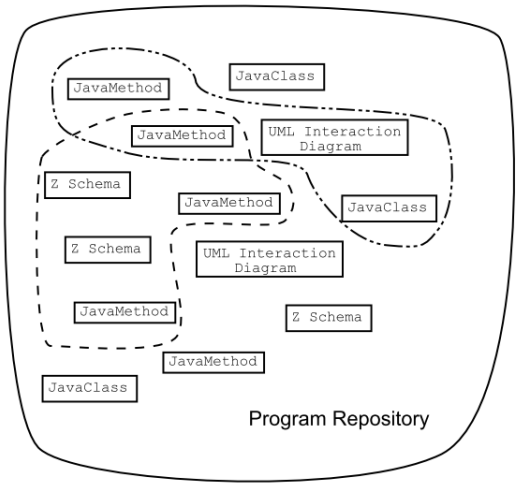


Abbildung 17: Definition von unterschiedlichen Sichten auf den Quellcode [63]

3.2.1.3 Abgrenzung

Die beschriebenen Systeme speichern den Quellcode in feingranularen Einheiten ab. Die Größe der Einheit bestimmt die Möglichkeit Traceability Links definieren zu können. Beide Systeme definieren als kleinste Einheit die Methode und nicht den Abstract Syntax Tree. Damit wird zwar die Anzahl der zu verwaltenden Artefakte deutlich kleiner, aber es sind keine Traceability Links in die Methoden möglich. Die Abspeicherung des vollständigen AST bringt bezüglich der Anzahl der Artefakte einige Herausforderungen mit sich. Außerdem wird kein übergeordnetes Metamodell für den Softwareentwicklungsprozess mit Traceability Links definiert. Damit ist kein konsistentes Modell mit Traceability Links auf feingranulare Quellcode-Artefakte möglich.

3.2.2 Traceability

In diesem Kapitel werden verschiedene Ansätze und Softwarewerkzeuge behandelt, die das verlinken von Quellcode über Traceability Links unterstützen oder Modelle mit Quellcode synchron halten.

3.2.2.1 Automatische Erstellung von Traceability Links

Beschreibung

Ein möglicher Ansatz ist Traceability Links zwischen Anforderungen und Quellcode automatisch nach verschiedenen Algorithmen zu berechnen [9].

Antoniol et al. [65] nimmt an, dass der Entwickler sinnvolle Namen für verschiedene Programmelemente verwendet zum Beispiel für Klassen, Methoden und Variablen und dass diese Namen sich in dem freien Text der Anforderungsbeschreibung wiederfinden. Durch die

Analyse des Quellcodes und der Texte der Anforderungen können dann entsprechende Links gefunden werden. Diese Methode kann vollautomatisch durchgeführt werden und es ist nur eine Bewertung der Links durch einen Menschen notwendig.

Trusttrace [66] ist ein Ansatz um die Präzession von Baseline Traceability Links zu verbessern. Es besteht aus mehreren verschiedenen Komponenten und die Komponenten „Histrace-commits“ und „Histrace-bugs“ verwenden die Abgabe-Kommentare aus dem SCM Repository um Traceability Links zwischen Quellcode und Anforderungen beziehungsweise Fehlerberichten zu erstellen. Dazu wird der Text in der Anforderungsbeschreibung und im Fehlerbericht auf Ähnlichkeiten zu dem Text des Abgabe-Kommentars hin untersucht. Wenn eine Überstimmung gefunden wurde, dann werden die mit der Abgabe veränderten Quellcode Dateien mit der Anforderung oder dem Fehlerbericht verlinkt.

Egyeds und Grünbachers [67] Ansatz geht davon aus, dass existierenden Anwendungsfälle und Anforderungen miteinander verlinkt sind. Über einen sogenannten Trace Analyzer werden alle Quellcodeartefakte aufgezeichnet, die während der Verwendung eines Anwendungsfalls ausgeführt werden. Damit können die Quellcodeartefakte mit dem Anwendungsfall und damit auch mit den Anforderungen in Verbindung gebracht werden.

Eisenberg und Volder [68] versuchen Features mit relevanten Quellcode automatisch zu verknüpfen indem untersucht wird, welcher Quellcode bei der Ausführung von Testfällen durchlaufen wird. Dazu muss ein Entwickler zuvor die Testfälle mit den Anforderungen verlinken. Während der Ausführung eines Testfalls werden die aufgerufenen Methoden mit dem Testfall und damit mit der Anforderung verlinkt. Außerdem werden heuristische Methoden verwendet um die Relevanz einer Methode für ein Feature zu berechnen.

Abgrenzung

Die oben beschriebenen Ansätze versuchen im Nachhinein den Quellcode mit den Anforderungen zu verlinken und weisen eine beschränkte Trefferquote (zwischen ca. 50% - 75%) auf [9]. Diese automatisch berechneten Traceability Links können natürlich nochmals von einer Person bewertet und korrigiert werden (bestätigt oder gelöscht werden), was aber einen erheblichen Aufwand bedeuten kann. Vor allem werden Links, die nicht gefunden wurden, weil es zum Beispiel keine Textähnlichkeiten gibt, überhaupt nicht berücksichtigt. Die Ansätze sind nicht so feingranular wie in dieser Arbeit gefordert (verlinken bis auf die

Ebene von Anweisungen). Außerdem ist die historische Rückverfolgbarkeit der Links für einzelne Programmartefakte so nicht möglich.

3.2.2.2 UNICASE Trace Client

Beschreibung

UNICASE Trace Client [9] verwendet ein Traceability Information Modell bestehend aus einem Systemmodell, Projektmodell und einem Codemodell (siehe Abbildung 18).

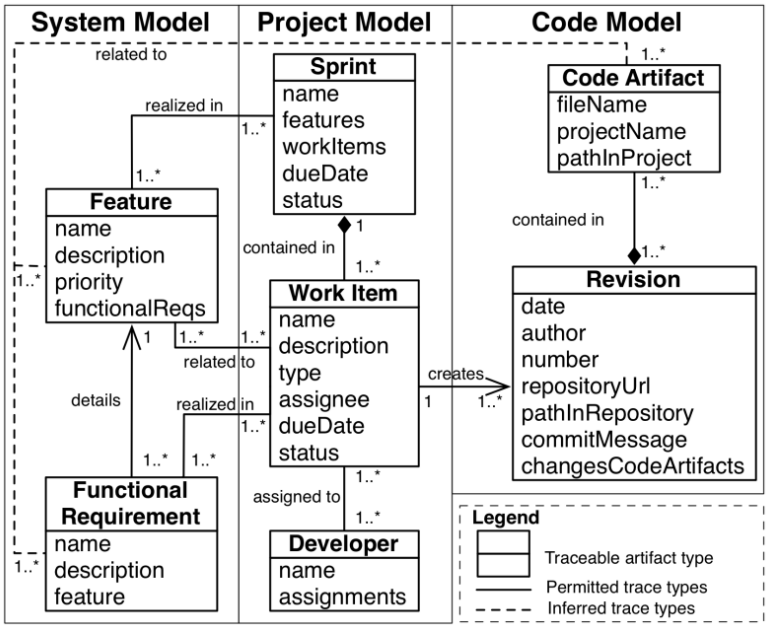


Abbildung 18: Traceability Information Modell mit Systemmodell, Projektmodell und Codemodedell [9]

Das Systemmodell beinhaltet Features und funktionale Anforderungen, welche Anforderungen mit unterschiedlicher Detaillierung repräsentieren. Ein Feature ist eine abstrakte Beschreibung der Anforderung, welches in der funktionalen Anforderung genauer beschrieben wird. Im Projektmodell werden die Artefakte Entwickler, Arbeitsaufgabe und Sprint verwendet. Die Arbeitsaufgaben besitzen eine Aufgabenbeschreibung, die dann einem konkreten Entwickler zugewiesen wird. Sprints werden verwendet um die Arbeitsaufgaben zu organisieren und legen eine Zeitspanne für die Umsetzung fest. Das Codemodell beinhaltet dateibasierte und änderungsbasierte Repräsentationen des Quellcodes (Codeartefakt und Revision). Die Artefakte werden über Traceability Links miteinander verbunden zum Beispiel zwischen Arbeitsaufgabe und Revision: Die Abarbeitung einer Arbeitsaufgabe erzeugt eine neue Revision.

Es werden drei unterschiedliche Vorgehensweisen bei der Entwicklung des Quellcodes unterstützt, um die Traceability Links semi-automatisch zu erfassen. Der Entwickler legt dabei fest für welche Arbeitsaufgabe er gerade tätig ist und es werden gesichtete Anforderungen und geändert Quellcodeartefakte als potentielle Artefakte zum verlinken dem Entwickler angeboten. Der Entwickler entscheidet, welche der Anforderungen und Quellcodeartefakte tatsächlich für die Arbeitsaufgabe relevant sind. Die Arbeitsaufgabe wird beim Abgeben des Quellcodes in das Software Configuration Management (SCM) Repository mit der dadurch erstellen Revision verlinkt. Beginnend mit den Arbeitsaufgaben lassen sich dann Traceability Links zwischen Anforderung und Quellcode ableiten. Zum Beispiel ist eine Anforderung mit zwei Arbeitsaufgaben verlinkt und jede Arbeitsaufgabe hat eine Revision mit Quellcodeartefakten. Daraus kann dann der Link zwischen Anforderung und Quellcodeartefakten abgeleitet werden. Diese Ableitung wird automatisch durchgeführt. Wenn die Anforderungen verändert wurden, muss der Projektbeteiligte entscheiden, ob die abgeleiteten Links noch korrekt sind oder nicht.

Bei der Ableitung der Traceability Links zwischen Anforderungen und Quellcode werden SCM-Operationen berücksichtigt: Hinzufügen von Quellcode-Dateien, Ändern von Namen oder Pfad einer Quellcode-Datei, Änderung des Inhalts oder Löschen von Quellcode-Dateien. Wenn zum Beispiel der Name einer Datei für eine bestimmte Arbeitsaufgabe während des SCM Abgabe verändert wurde, wird bei der Ableitung der Links alle bereits existierende Links auf den neuen Namen aktualisiert. Beim Löschen einer Datei werden bestehende Links gelöscht. Zusätzlich werden zwei Operationen für die Refaktorierung unterstützt (siehe Abbildung 19).

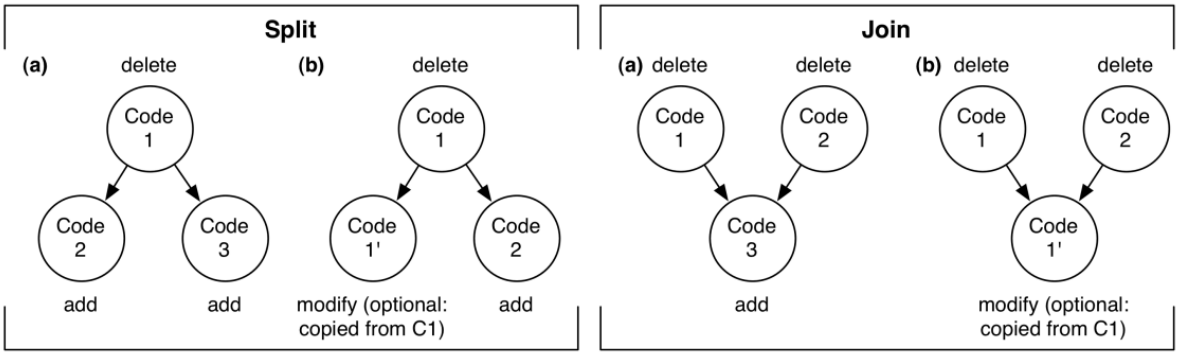


Abbildung 19: Operationen für die Refaktorierung [9]

Dabei wird das Aufspalten einer Klasse in mehrere Klassen oder die Zusammenführung mehrerer Klassen in einer Klasse unterstützt.

UNICASE Trace Client unterstützt eine graphische Visualisierung der Traceability Links, wobei die Artefakte als Knoten und die Traceability Links als Kanten dargestellt werden. Der Anwender kann dabei einzelne Artefakte, eine Gruppe von Artefakten oder ein komplettes Projekt für die Darstellung auswählen.

Abgrenzung

Das Modell des UNICASE Trace Clients beinhaltet die folgenden Prozessgebiete: Anforderungsmanagement, Projektmanagement und Quellcode. Testdatenmanagement und deren Traceability Links sind nicht berücksichtigt. Außerdem verwendet UNICASE Trace Client ein dateibasiertes SCM Repository. Damit sind die Möglichkeiten von Traceability Links in den Quellcode beschränkt. Im Wesentlichen werden Links zu Klassen unterstützt und einige Refaktorierungen auf Klassen-Ebene wie zum Beispiel umbenennen, zusammenführen und aufspalten von Klassen. Traceability Links auf Methodenebene oder noch feingranularer werden nicht unterstützt. Die Aufzeichnung der Historie von Traceability Links wird nicht unterstützt. Die Strategie ist Links, die ungültig geworden sind, zu löschen. Für die Erstellung der Traceability Links zwischen Anforderung und Quellcode während des Abgebens von Quellcode in das Repository wurde eine ähnliche Lösung entwickelt, wie in dieser Arbeit.

3.2.2.3 Software Architektur Modelle und Quellcode

Beschreibung

Langhammer [69] entwickelte einen Ansatz, um komponentenbasierte und verhaltensbeschreibende Architekturmodelle konsistent mit dem Quellcode zu halten. Als Architektur Modell wird das Palladio Component Model (PCM) und als Quellcode Java unterstützt. Um die Konsistenz sicherzustellen, werden Abbildungsregeln zwischen dem Modell und dem Quellcode definiert. Dazu wurden drei Dimensionen für die Konsistenzregeln festgelegt: eine technologie-spezifische, eine projekt-spezifische und eine element-spezifische Dimension. Wenn die Konsistenz nicht automatisch hergestellt werden kann, dann interagiert der Anwender mit dem System und teilt seine Absicht mit der durchgeführten Änderung mit. Der vorgestellte Ansatz ist änderungsgetrieben, das heißt alle Änderungen am Architekturmodell und am Quellcode werden über Beobachter in den Editoren aufgezeichnet und an das System für die Anwendung der Abbildungsregeln weitergereicht. Dabei sollen die für den Anwender gewohnten Editoren unterstützt werden, also der Editor für das PCM Modell und der Eclipse Java Quellcodeeditor. Für bereits bestehende Architekturmodelle und bestehenden Quellcode wird die Erstellung der Modelle

simuliert und die Änderungen werden aufgezeichnet. Dazu stehen zwei Integrationsstrategien zur Verfügung. Für die Reconstructive Integration Strategy (RIS) wird die Erstellung des Architektur Modells simuliert und durch die aufgezeichneten Änderungen die Quellcode-Elemente generiert. Bei der Linking Integration Strategy (LIS) werden Model-to-Model (M2M) oder Model-to-Text (M2T) Transformationen verwendet, um die zu integrierende Modellanteile zu erzeugen. Für die Evaluierung der Lösungskonzepte wurden mehrere Open Source Projekte verwendet.

Abgrenzung

In diesem Ansatz gibt es keine expliziten Links zwischen den Modellen, sondern es wird die Konsistenz der Modelle über Abbildungsregeln sichergestellt. Anforderungen, Tickets und Testdaten werden nicht berücksichtigt und die Historie der Änderungen lässt sich nicht nachvollziehen. Der Ansatz die Änderungen aufzuzeichnen, um die Konsistenz sicherzustellen, ist ähnlich zu dem Ansatz in dieser Arbeit für die Erhaltung der Traceability Links bei Änderungen am Quellcode.

3.2.2.4 Capra

Beschreibung

Capra [55] unterstützt Traceability zwischen beliebigen Artefakten und besitzt einen größeren Grad an Anpassbarkeit im Vergleich zu anderen Werkzeugen. Es werden benutzerdefinierte Linktypen unterstützt, die von einem Projektmanager für das Projekt definiert werden können. Der Anwender kann Traceability Links anlegen, aktualisieren und in verschiedenen Ansichten visualisieren.

Capra ist ein Eclipse-Plugin und verwendet das Eclipse Modelling Framework (EMF). Die Traceability Links werden in einem EMF Modell gespeichert. Es werden über mehrere Erweiterungspunkte eine benutzerdefinierte Anpassung der Anwendung ermöglicht (siehe Abbildung 20).

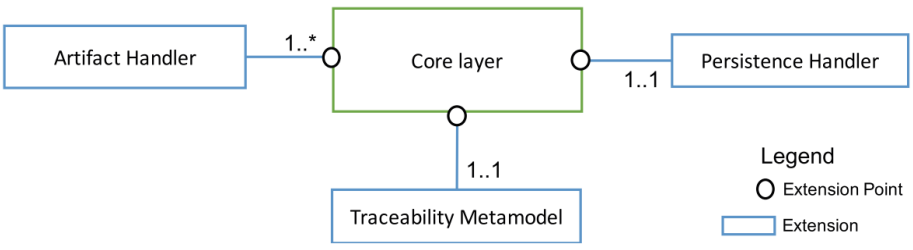


Abbildung 20: Erweiterungspunkte von Capra [55]

Für die Definition der unterschiedlichen Linktypen stellt das Werkzeug den Erweiterungspunkt „Traceability Metamodel“ zur Verfügung. Hier kann der Endanwender die Linktypen in einem Metamodell definieren, die dann in der Anwendung zur Verfügung stehen. Mögliche Linktypen wären zum Beispiel „verifiziert“, „implementiert“, „verfeinert“ oder „steht in Beziehung zu“. Der Erweiterungspunkt „Artifact Handler“ erlaubt die Integration von Artefakten unterschiedlicher Anwendungen aus verschiedenen Prozessgebieten wie zum Beispiel Anforderungsmanagement, Design, Implementierung und Testdatenmanagement. Da die Traceability Links in einem EMF Modell gespeichert werden, müssen über diesen Erweiterungspunkt die Artefakte aus unterschiedlichen Formaten in ein EMF Format überführt werden. Der Erweiterungspunkt „Persistence Handler“ erlaubt die Definition unterschiedlicher Speicherorte für das Traceability Modell und die Integration von Versionierungssystemen.

Ohne eine konkrete Konfiguration kann die Anwendung nicht verwendet werden. Daher gibt es eine Default-Konfiguration, das ein einfaches Traceability Modell zur Verfügung stellt. Damit sind Traceability Links zwischen folgenden Artefakt-Typen möglich: Java Quellcode (bis zu Methodenebene), C/C++ Quellcode (bis zur Funktionsebene), Dateien (wie zum Beispiel PDF oder MS Word), Aufgaben aus Mylyn [47] und Testausführungen aus einem Continuous Integration Werkzeug. Alle Traceability Links werden im Workspace in einem Verzeichnis gespeichert.

Das Werkzeug stellt Funktionalität für die Visualisierung der Traceability Links zur Verfügung. Die Links können in einer Matrix dargestellt werden. Außerdem wird ein graphisches Format mit den Artefakten als Knoten und die Traceability Links als Kanten unterstützt. Das in Abbildung 21 gezeigte Beispiel verknüpft verschiedene Artefakte miteinander: eine Anforderungsspezifikation, ein Feature, eine Komponente, ein PDF Datei und einen Zustandsautomaten.

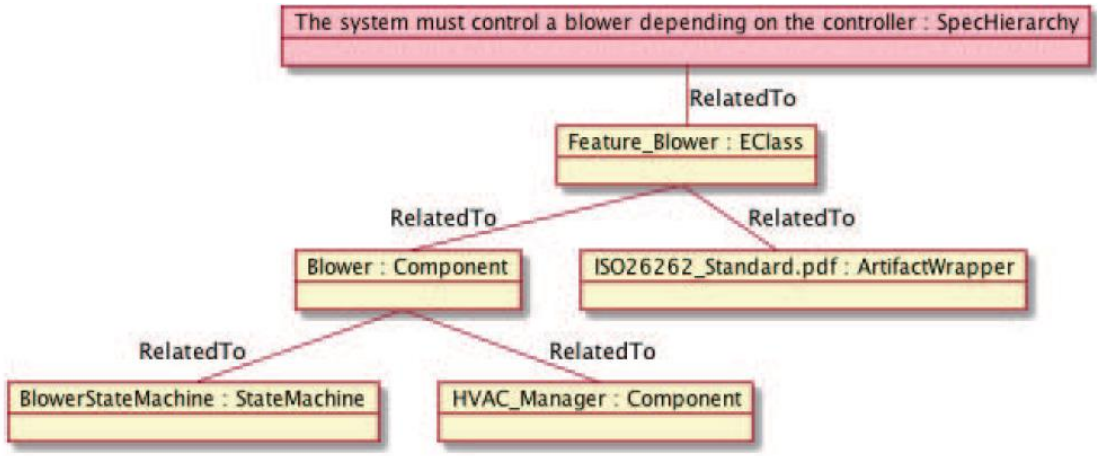


Abbildung 21: Graphische Repräsentation von Artefakten mit Traceability Links [55]

Abgrenzung

Capra unterstützt Traceability nur bis auf Methodenebene und nicht für einzelne AST Artefakte. Es wird keine Versionierung bzw. Historie einzelner AST Artefakte unterstützt. Die Flexibilität bezüglich des Metamodells ermöglicht zwar eine genaue Anpassung an die Vorstellung des Endanwenders. Damit sind dann aber keine maßgeschneiderten Editoren und Ansichten möglich, sondern es muss immer über eine generische Benutzeroberfläche gearbeitet werden. Businesslogik kann erstmal nicht bereitgestellt werden oder muss aufwändig an das Metamodell angepasst werden. Es gibt keine starke Integration in den Java Editor für die Erstellung und Visualisierung der Traceability Links. Damit ist eine einfache und intuitive Bedienung nur bedingt möglich und die Aufwände für das Erstellen und Pflegen von Traceability Links bleiben hoch.

3.2.2.5 Application Lifecycle Management Lösungen

Es gibt einige kommerzielle Application Lifecycle Management (ALM) Lösungen (zum Beispiel Polarion Software [57], Rational Team Concert [58] oder Microsoft Team Foundation Server [59]), die die Idee eines Datenbackbones mit allen Daten aus dem Softwareentwicklungsprozess mit Traceability Möglichkeiten und die Idee einer Benutzeroberfläche, um auf die Daten zugreifen zu können, in Ansätzen schon sehr gut umsetzen.

Microsoft hat in den letzten Jahren viel Aufwand in ihre Entwicklungsumgebung Visual Studio und dem Datenbackbone Microsoft Team Foundation Server [70] gesteckt und hat große Fortschritte gemacht. Microsoft gehört daher zu den führenden Firmen auf diesem

Gebiet. Daher sollen hier stellvertretend für die anderen ALM Systeme die Möglichkeiten von Visual Studio und dem Microsoft Team Foundation Server aufgezeigt werden.

Beschreibung

Visual Studio Team Services und der Team Fundation Server unterstützen den Softwareentwicklerprozess in folgenden Punkten:

- Verwaltung von Quellcode
  Microsoft unterstützt zwei verschiedene Versionsverwaltungssysteme: Git [25] und Microsofts Team Foundation Version Control (TFVC). Git unterstützt ein verteiltes Arbeiten, das heißt jeder Entwickler hat seine eigene Kopie des Quellcode-Repositories auf seiner lokalen Maschine. Änderungen können erst in das lokale Repository abgegeben werden und verschiedene Versionsverwaltungsoperationen können offline durchgeführt werden (zum Beispiel Abfrage der Historie einer Datei). TFVC ist der zentralisierte Ansatz für die Versionsverwaltung. Typischerweise haben die Entwickler immer nur eine Version ihrer Datei auf ihrer lokalen Maschine und die verschiedenen Versionsverwaltungsoperationen können nur auf dem Server durchgeführt werden. Ansonsten werden die sonst üblichen Versionsverwaltungsoperationen auf den Quellcode-Dateien unterstützt wie zum Beispiel auschecken, editieren und check in (oder auch Abgabe) der Änderungen, vergleichen von Quellcode und Konfliktlösung, Benennung von Versionen, Erstellen von Entwicklungszweigen, suchen und einsehen von Change-Sets, zurückrollen von Change-Sets, sichten der Historie einer Datei.
- Unterstützung des Projektmanagements durch Agile Werkzeuge
  Microsoft unterstützt dabei verschiedene Prozesse: Agile, Scrum [1] oder Capability Maturity Model Integration (CMMI) [71]. Der sogenannte Backlog entspricht dem Projektplan mit den geplanten Lieferumfängen für den Kunden (siehe Abbildung 22). Mit dem Backlog hat man eine priorisierte Liste von Anforderungen und Features, die umgesetzt werden sollen. Daraus werden dann die Arbeitsaufgaben erstellt und an die Entwickler zugewiesen. Unter der Angabe der Arbeitsaufgabe wird der geänderte Quellcode abgegeben und damit verlinkt. Die Verwaltung von Softwarefehlern wird ebenfalls unterstützt.
- Unterstützung von Continuous Integration [2] und Continuous Delivery [3]
  Mit Hilfe von Continuous Integration und Delivery kann das Produkt regelmäßig

gebaut und für die Validierung und Verifizierung durch andere Projektbeteiligte auf unterschiedlichen Umgebungen zur Verfügung gestellt werden. Außerdem wird die Verwaltung von Testdaten unterstützt: Erstellung, einplanen und Ausführung von Testfällen.

- Dashboards
  Reportberichte zeigen den Status des Projektes, überwachen den Fortschritt und teilen Informationen zwischen allen Projektbeteiligten. Dazu können Web-Seiten individuell eingerichtet werden.

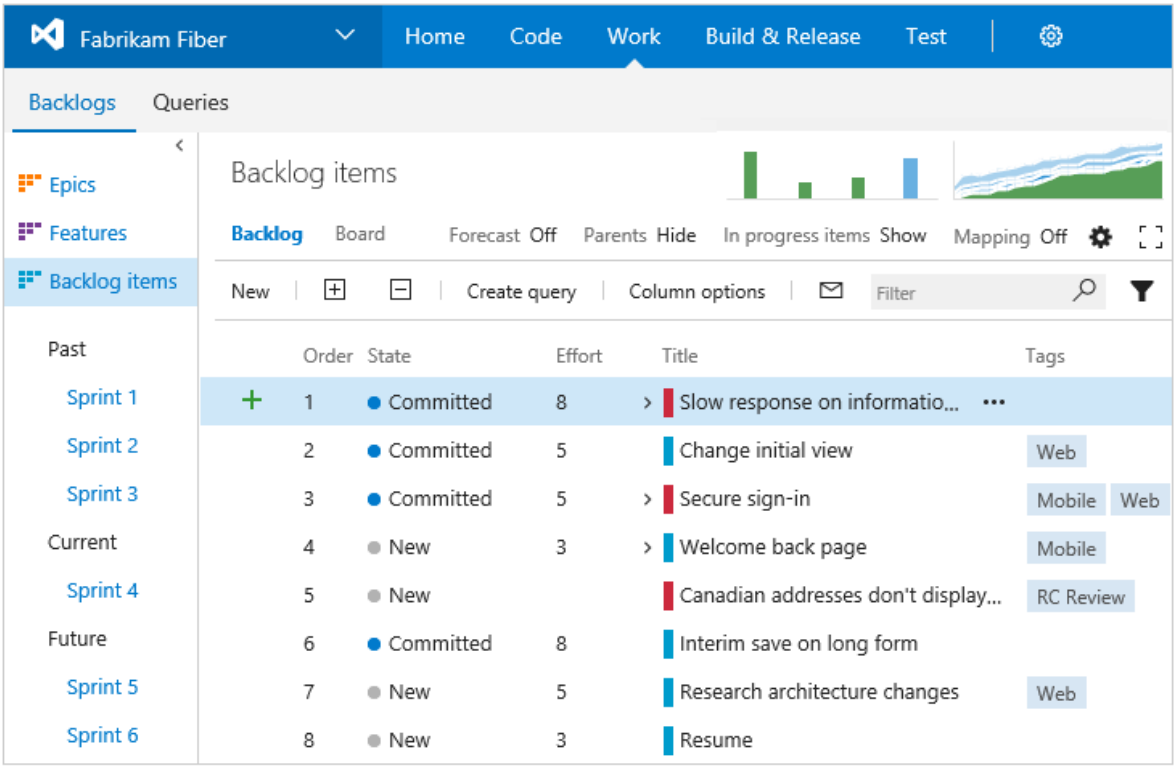


Abbildung 22: Beispiel für ein Backlog

Die Visual Studio Team Services können zum einen über eine Web-Oberfläche genutzt werden und integrieren sich aber auch in verschiedene Entwicklungsumgebungen: Microsoft Visual Studio, Eclipse [19] und IntelliJ [72].

Visual Studio ist die integrierte Entwicklungsumgebung (IDE) von Microsoft. Mit ihr lassen sich Anwendungen für verschiedene Betriebssysteme erstellen. Sie stellt Funktionalität bereit für das Entwickeln von Quellcode, das Navigieren und Sichten von Quellcode, das Debuggen und Testen der entwickelten Anwendung. Visual Studio unterstützt die Zusammenarbeit der Entwickler durch zum Beispiel die Integration in die Visual Studio Team Services und der

sogenannten CodeLens, die in den Editor integriert ist (siehe Abbildung 23). Die CodeLens stellt zu einer Methode verschiedene Informationen bereit: Verweise (wo wird diese Methode verwendet), Informationen über Änderungen an dieser Methode (wer, wann und welche Änderungen wurden durchgeführt), Verknüpfte Fehler und Arbeitsaufgaben zu dieser Methode.

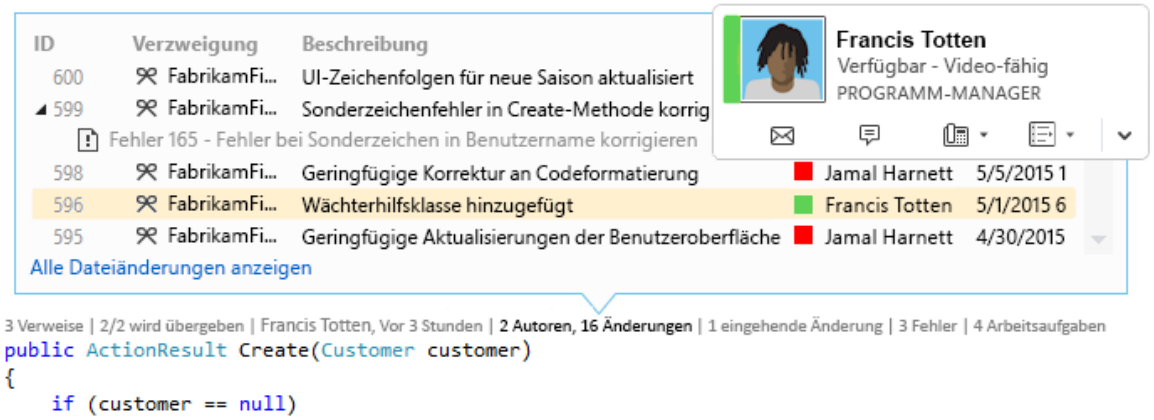


Abbildung 23: Editor in der Visual Studio IDE mit der CodeLens

Abgrenzung

Diese ALM Lösungen arbeiten alle mit den üblichen und weitverbreiteten dateibasierten SCM Repositories zusammen, womit die Verlinkung mit Quellcode nur eingeschränkt möglich ist. Wenn der Quellcode als Text abgelegt wird, kann die Historie einzelner AST Artefakte nicht mehr zuverlässig ermittelt werden und sind Traceability Links auf einzelnen AST Artefakten nur beschränkt möglich. Eine Refaktorierung einer Methode (zum Beispiel Umbenennung oder Verschieben) ist nachträglich im Text nicht mehr erkennbar. Um die Historie und Traceability Links zu unterstützen müssen die AST Artefakte mit eigener Identität (ID) in einer Datenbank abgelegt werden und über einen Editor, der mit AST Artefakten umgehen kann, bearbeitet werden. Visual Studio stellt zumindest auf Methodenebene über die CodeLens detaillierte Informationen zur Methode bereit, aber nicht für andere AST Artefakte. Nach der Umbenennung einer Methode, auch wenn die Funktionalität für die Refaktorierung von der Entwicklungsumgebung genutzt wird, kann nicht mehr auf die Informationen der Methode vor der Umbenennung zugegriffen werden, weil durch die Umbenennung eine neue Methode entsteht.

3.2.3 Abstract Syntax Tree (AST)

Es gibt eine Reihe von Softwarewerkzeugen, die sich auf das Arbeiten mit dem AST spezialisiert haben. Die zwei wichtigsten für diese Arbeit werden in diesem Kapitel vorgestellt.

3.2.3.1 Domain-Specific Language (DSL) Development Environment

Beschreibung

JetBrains Meta Programming System (MPS) [72] ist eine Umgebung für Sprachdatenverarbeitung, die es ermöglicht eine eigene domänenspezifische Sprache zu kreieren oder eine bestehende zu erweitern. Das Problem einer erweiterten Sprachsyntax ist die textuelle Repräsentation. Daher wird der Quellcode in MPS immer als Abstract Syntax Tree gepflegt. MPS erlaubt es trotzdem den Quellcode in einer textähnlichen Art zu editieren. Dieser Editor wird Projectional-Editor genannt und die Editormöglichkeiten orientieren sich immer an dem für den Editor konfigurierten AST (siehe Abbildung 24). MPS bring eine universelle Sprache mit namens „BaseLanguage“ mit, die noch zusätzlich erweitert werden kann. Diese Sprache kann als Referenz zur Definition eigener Sprachen verwendet werden. Verschieden Werkzeuge werden hierfür bereitgestellt.

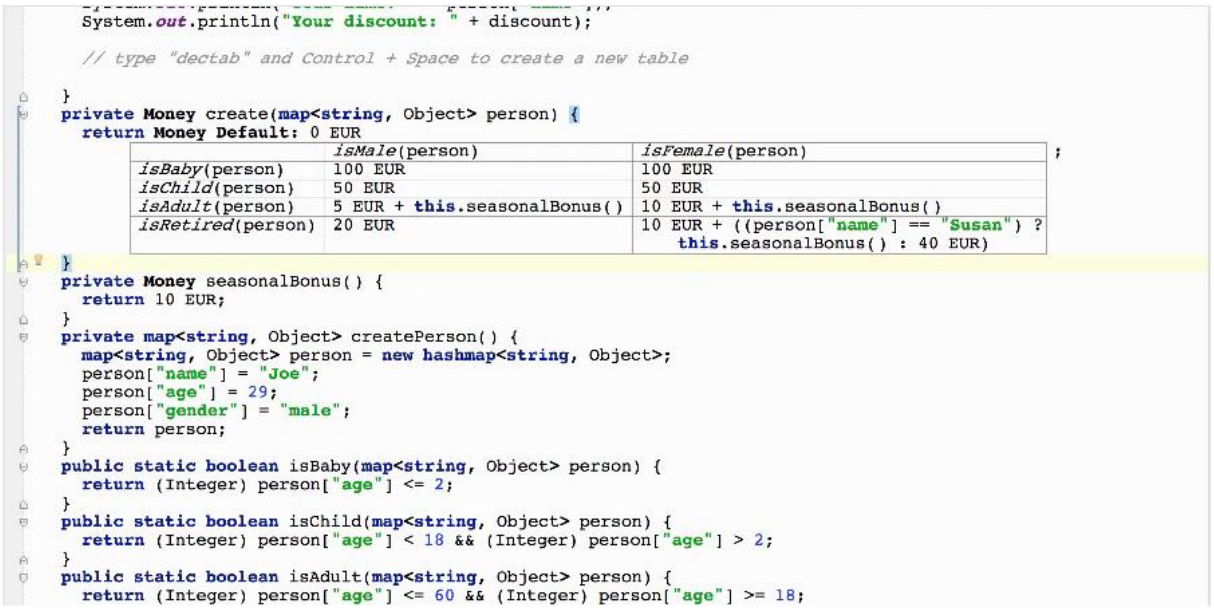


Abbildung 24: Projectional-Editor von MPS [72]

Der Projectional-Editor von MPS ist in eine Entwicklungsumgebung integriert, mit der Softwareprojekte und deren Quellcode verwaltet werden können. Wenn der AST ein Java AST ist, dann kann die Entwicklungsumgebung als Java Entwicklungsumgebung verwendet

werden. Das Java Projekt kann gebaut, ausgeführt und debugged werden. Der Quellcode wird nicht mehr in Textdateien abgelegt, sondern in XML Dateien, die den AST beinhalten (siehe Abbildung 25). Die Dateien und Projekte können in einem Versionskontrollsystem abgelegt werden zum Beispiel in Subversion [23]. Die AST Artefakte haben eine ID, die bei verschiedenen Operationen erhalten bleibt. So kann zum Beispiel der Name einer Methode geändert werden oder eine Methode verschoben werden, ohne dass sich die ID verändert. Auch feingranulare AST Artefakte wie eine If-Anweisung oder eine For-Schleife haben eine ID.

```
<?xml version="1.0" encoding="UTF-8"?>
<model ref="r:f7db11ae-d0f8-47e1-b7ab-0786ce192dc6(vi.test2)">
  <persistence version="9" />
  <languages>
    <use id="f3061a53-9226-4cc5-a443-f952ceaf5816" name="jetbrains.mps.baseLanguage" version="-1" />
  </languages>
  <imports>
    <import index="4qxd" ref="r:fcc26a65-4dc9-471b-a00b-09fddbda4ad2(vi.test)" />
  </imports>
  <registry>
    <language id="f3061a53-9226-4cc5-a443-f952ceaf5816" name="jetbrains.mps.baseLanguage">
      <concept id="1215693861676" name="jetbrains.mps.baseLanguage.structure.BaseAssignmentExpression" flags="nn" index="d038R">
        <child id="1068498886297" name="rValue" index="37vLTx" />
        <child id="1068498886295" name="lValue" index="37vLTJ" />
      </concept>
      <concept id="1202948039474" name="jetbrains.mps.baseLanguage.structure.InstanceMethodCallOperation" flags="nn" index="liA8E" />
      <concept id="1465982738277781862" name="jetbrains.mps.baseLanguage.structure.PlaceholderMember" flags="ng" index="2tJIrI" />
      <concept id="2820489544401957797" name="jetbrains.mps.baseLanguage.structure.DefaultClassCreator" flags="nn" index="HV5vD">
        <reference id="2820489544401957798" name="classifier" index="HV5vE" />
      </concept>
      <concept id="1197027756228" name="jetbrains.mps.baseLanguage.structure.DotExpression" flags="nn" index="2OqwBi">
        <child id="1197027771414" name="operand" index="2Oq$k0" />
        <child id="1197027833540" name="operation" index="2OqNvi" />
      </concept>
      <concept id="1145552977093" name="jetbrains.mps.baseLanguage.structure.GenericNewExpression" flags="nn" index="2ShNRf">
        <child id="1145553007750" name="creator" index="2ShVmc" />
      </concept>
      <concept id="1070534370425" name="jetbrains.mps.baseLanguage.structure.IntegerType" flags="in" index="10Oyi0" />
      <concept id="1068390468200" name="jetbrains.mps.baseLanguage.structure.FieldDeclaration" flags="ig" index="312cEg">
        <property id="8606350594693632173" name="isTransient" index="eg7rD" />
        <property id="1240249534625" name="isVolatile" index="34CwA1" />
      </concept>
      <concept id="1068390468198" name="jetbrains.mps.baseLanguage.structure.ClassConcept" flags="ig" index="312cEu" />
      <concept id="1068431474542" name="jetbrains.mps.baseLanguage.structure.VariableDeclaration" flags="ng" index="33uBYm">
        <property id="1176718929932" name="isFinal" index="3TUv4t" />
        <child id="1068431790190" name="initializer" index="33vP2m" />
      </concept>
      <concept id="1068498886296" name="jetbrains.mps.baseLanguage.structure.VariableReference" flags="nn" index="37vLTw">
        <reference id="1068581517664" name="variableDeclaration" index="3cqZAo" />
      </concept>
```

Abbildung 25: MPS AST in einer XML Datei

Abgrenzung

Mit dem Projectional-Editor von MPS wurde ein wichtiges Feature für stabile Traceability Links in den Quellcode bereitgestellt. Damit ist eine Refaktorierung von Quellcode möglich, ohne dass die Identität der AST Artefakte verloren gehen. Jedoch wird bei der Kopie von Quellcode die ID beibehalten, was zu mehreren AST Artefakte mit gleicher ID führen kann. Wenn man die ID um die Modell-ID erweitert, was zu eindeutigen IDs führen würde, geht dann jedoch die Identität durch Verschieben verloren, weil sich dann die ID des AST

Artefakts ändert (siehe Abbildung 25). MPS stellt kein Gesamtmetamodell für den Softwareentwicklungsprozess mit Traceability Links bereit. Dadurch können keine anderen Artefakte mit dem Quellcode verknüpft werden. Außerdem ist die Ablage der AST Artefakte innerhalb der XML Dateien schwierig. Es ist nicht klar in welcher XML Datei nach einer bestimmten ID zu suchen ist und die XML Datei muss geparst und durchsucht werden, um das AST Artefakt zu finden. Die Historie eines AST Artefakts ist schwierig und aufwendig zu berechnen, weil hier nicht klar ist, welche Dateien in welcher Revision wirklich relevant sind. Und so bestehen sehr ähnliche Probleme wie bei der Referenzierung über den vollqualifizierten Namen mit dem Unterschied, dass eine Refaktorierung des Quellcodes unterstützt wird.

3.2.3.2 Aufzeichnung der Änderungshistorie eines ASTs

Beschreibung

Spyware [73] ist eine Anwendung zur Aufzeichnung aller vom Entwickler durchgeführten Änderungen am Quellcode. Dazu wird in die Entwicklungsumgebung um ein Überwachungs-Plugin erweitert, das nicht die Änderung des Quellcode-Texts, sondern die Änderungen am AST aufzeichnen (inklusive Refaktorierungen). Änderungen werden für spätere Auswertungen in ein eigenes Repository abgespeichert. Damit können Änderungen an jedem Package, Klasse, Variable, Methode oder Anweisung nachvollzogen werden. Die Unterschiede zwischen zwei Quellcode Versionen können mittels Farbkodierung für die Art der Änderung graphisch dargestellt werden. Außerdem kann das Ausmaß der Änderungen über Metriken berechnet werden.

Abgrenzung

Der Schwerpunkt von Spyware ist die wissenschaftliche Untersuchung von Programmänderungen durch den Softwareentwickler. Es kann zwar die Historie von AST Artefakten nachvollzogen werden, es werden jedoch keine Traceability Links unterstützt und es gibt kein übergeordnetes Metamodell für den Softwareentwicklungsprozess. Außerdem wird jede Änderung im Editor gespeichert und nicht nur die Änderungen zwischen zwei Abgaben. Der Prototyp dieser Arbeit zielt auf große Entwicklungsprojekte ab mit mehreren Millionen Zeilen Quellcode. Da würde die Aufzeichnung jeder Änderung zu einer ungewünscht großen Datenmenge führen. Das Überwachungs-Plugin für Eclipse und Java, Eclipseye [74], kann nur bis auf Methodenebene Änderungen aufzeichnen.

3.3 Konzeptidee und Alleinstellungsmerkmale

Der Quellcode wird in der Regel als Text in einem Repository abgespeichert. Folgende drei Lösungen wären möglich, um einen bidirektionalen Link zwischen Quellcodetext und einem Artefakt zu realisieren um Traceability zu ermöglichen:

1. Der vollqualifizierte Name, zum Beispiel in Java der Packagename, Klassenname und Methodenname werden mit dem Artefakt gespeichert. Wenn der Softwareentwickler an der entsprechenden Quellcode-Stelle eine Änderung vornimmt, zum Beispiel die Methode umbenennt, müssen alle Traceability Links angepasst werden. In diesem Fall müssen die Links zu dieser Methode gesucht und aktualisiert werden. Beim Löschen des Quellcodes müssen die Links ebenfalls gesucht und gelöscht werden. Das ist sehr aufwändig und komplex. Außerdem ist es nicht möglich Links zu Quellcode herzustellen, die keinen expliziten Namen haben, zum Beispiel eine „For-Schleife“ oder eine „If-Anweisung“. Es ist ebenfalls sehr aufwändig und schwierig dem Link vom Quellcode zum Artefakt zu folgen.
2. Der Verzeichnisname, Dateiname und die Zeilennummer werden mit dem Artefakt gespeichert. Diese Information ist jedoch nur für eine ganz bestimmte Version der Datei gültig, die mit dem Artefakt gespeichert werden muss. Jede Änderung an der Datei kann dazu führen, dass die Zeilennummer nicht mehr gültig ist. Es ist deswegen nur möglich dem Traceability Link zu einer bestimmten Version der Datei in der Historie zu folgen und es ist nicht definiert wohin der Link in der aktuellen Version der Datei zeigt.
3. Der Link könnte im Quellcode hinterlegt werden durch zum Beispiel das Abspeichern einer eindeutigen ID (Identifier) oder einer URL (Uniform Resource Locator) des Artefakts in einem Kommentar vor dem verlinkten Quellcode (Methoden- oder Klassenkommentar). Ein Verschieben und Umbenennen von Quellcode würde so funktionieren. Dabei ist man jedoch auf den Softwareentwickler angewiesen, dass der Kommentar nicht aus Versehen gelöscht wird oder durch Kopieren von Quellcode an falsche Stellen gelangt. Damit wäre es möglich dem Link aus dem Quellcode zum Artefakte zu folgen, aber der umgekehrte Weg wäre wieder deutliche aufwändiger. Links zu einzelnen Anweisungen wären nicht wirklich praktikabel, weil die Bindung von Kommentar zu der entsprechenden Anweisung nicht eindeutig genug ist: Steht die Link Information vor oder hinter der Anweisung und auf welchen Teil der Anweisung

bezieht sich die Link-Information? Der Quellcode würde durch diese zusätzlichen Kommentare unübersichtlich werden.

Viele bisherige Ansätze nutzen eines dieser Lösungskonzepte mit den daraus resultierenden Nachteilen. Diese Probleme können nur konsequent gelöst werden, wenn der Text feingranular verarbeitet wird zum Beispiel als Abstract Syntax Tree. Einige feingranulare Versionskontrollsysteme verfolgen diesen Ansatz. Es wird jedoch ein Teil des Quellcodes immer noch als Text abgelegt, was die Möglichkeiten von Tracebilitiy Links einschränkt. Zudem fehlen die Konzepte für Traceability Links. Es muss beachtet werden, dass Softwareentwickler es gewohnt sind Quellcodetext in einem Texteditor zu bearbeiten und nicht AST-Datenstrukturen.

Um feingranulare Traceability Links in den Quellcode hinein konsequent und vollständig zu unterstützen, sind daher folgende Konzepte notwendig:

1. Ein konsistentes und vollständiges Metamodell mit allen Prozessgebieten der Softwareentwicklung (siehe Kapitel 2.3), das es erlaubt Traceability Links zwischen verschiedenen Artefakten und dem Quellcode zu definieren. Das kann durch die Definition eines Metamodells für den Quellcode erreicht werden.
2. Ein Editor, der es erlaubt den Quellcode als Text zu bearbeiten ohne dass die Traceability Links in den Quellcode verloren gehen.
3. Unterstützung für das Anlegen und Pflegen der Traceability Links um die Aufwände für den Softwareentwickler gering zu halten.
4. Visualisierung der Traceability Links während der Bearbeitung des Quellcodes im Quellcode Editor.

Darüber hinaus wurden noch folgende Ziele, die sich teilweise aus den Rahmenbedingungen (siehe Kapitel 1.3) ergeben, in dieser Arbeit verfolgt:

1. Der Softwareentwickler soll mit einer einheitlichen und durchgängigen graphischen Benutzeroberfläche arbeiten können ohne zwischen verschiedenen Anwendungen wechseln zu müssen (z. B. um Anforderungen sichten oder Softwarefehler einstellen zu können). Damit muss der Softwareentwickler nicht unterschiedliche Bedienkonzepte von unterschiedlichen Anwendungen erlernen und es kann eine Effizienzsteigerung in der Softwareentwicklung erreicht werden.

2. Alle Daten und Dateien sollen in einem gemeinsamen SCM Repository abgelegt werden: die verschiedenen Artefakte aus den Prozessgebieten, der Quellcode, die Projekte, in denen der Quellcode organisiert wird und alle zusätzlichen Daten und Dateien (zum Beispiel Manifest-Dateien). Damit kommt es zu keinem Bruch zwischen unterschiedlichen Repositorien mit unterschiedlichen Versionierungskonzepten.
3. Alle Features und Funktionen von Eclipse (siehe Kapitel 2.5) sollen weiterhin wie gewohnt vom Softwareentwickler genutzt werden können.

Damit lassen sich folgende Produktmerkmale bezüglich der Traceability Links für eine Darstellung der Alleinstellungsmerkmale definieren:

A) Feingranulare SCM

Der Quellcode wird nicht mehr als Text, sondern feingranular in einem Repository abgespeichert. Die Historie der Artefakte ist verfügbar, jedoch wird nicht unbedingt der vollständige Abstract Syntax Tree verwendet. Teile des Quellcodes können weiter als Text abgelegt werden.

B) SCM mit AST

Der Quellcode wird vollständig als Abstract Syntax Tree im Software Configuration Management (SCM) Repository abgespeichert. Jedes AST Artefakt hat seine eigene Historie, hat eine eindeutige ID und kann referenziert werden.

C) Stabile Traceability Links in den Code

Es werden Traceability Links in den Quellcode unterstützt. Beim Bearbeiten des Quellcodes gehen diese Links nicht verloren und es werden auch Refaktorierungen zum Beispiel umbenennen und verschieben unterstützt.

D) Ein Repository für alle Daten

Es werden alle Daten, die im Zuge der Entwicklung einer Software entstehen, in einem Repository abgelegt (zum Beispiel Anforderungsmanagement, Änderungsmanagement und Testdatenmanagement). Es gibt auch eine Benutzeroberfläche um diese Daten zu bearbeiten. Traceability Links zwischen den Artefakten wird unterstützt.

E) Editieren AST

Es wird das Editieren des ASTs ermöglicht, wobei die Objektidentität durch Verschieben oder Umbenennen nicht verloren geht.

F) Editieren AST und Traceability Links

Es wird das Editieren des ASTs unterstützt und das Erstellen von Traceability Links auf einzelne AST Artefakte. Bereits bestehende Traceability Links werden im Editor direkt im Quellcode angezeigt.

	Morpheus	MolhadoRef	Stellation	UNICASE Trace Client	Capra	Microsoft Visual Studio	Meta Programming System	Spyware
A) Feingranulare SCM	✓	✓	✓					✓
B) SCM mit AST	✓							
C) Traceability Links in den Code	✓			✓[1]	✓[2]			
D) Ein Repository für alle Daten	✓			✓		✓		
E) Editieren AST	✓						✓	
F) Editieren AST und Traceability Links	✓							

Tabelle 1: Alleinstellungsmerkmale bezüglich Traceability Links

3.4 Metamodell

3.4.1 Abstract Syntax Tree Metamodell

Für die verschiedenen Prozessgebiete kann ein Metamodell definiert werden mit Metaklassen wie zum Beispiel Anforderung, Testspezifikation, Testfall oder Ticket und Traceability Links zwischen diesen Metaklassen (siehe Kapitel 2.3) [75]. Quellcode wird üblicherweise als Text verarbeitet und gespeichert. Eine Lösung bezüglich der Traceability Links ist alles, inklusive des Quellcodes, in einem Metamodell zu definieren. Dann könnten alle Links gleich verarbeitet und behandelt werden, egal ob der Link auf Quellcode oder ein anderes Artefakt zeigt. Das kann man erreichen in dem man den Abstract Syntax Tree (AST) von Java in das Metamodell integriert (siehe Kapitel 2.4). Das komplette Modell einschließlich Anforderungen, Testfälle, Tickets und alle AST Artefakte kann somit in einem Datenbackbone abgelegt werden. Das Ziel ist **ein** Metamodell mit Traceability Links

[1] Teilweise: Traceability Links nur bis zur Klasse
[2] Teilweise: Traceability Links nur bis zur Methode

zwischen den Metaklassen für **alle** Aspekte des Software Entwicklungsprozesses zu definieren. Die Traceability Links sind bidirektional und so ist es möglich mit geringem Aufwand von einem Artefakt zum anderen zu gelangen.

Für diese Arbeit kam das Meta Data Framework (MDF) von PREEvision zum Einsatz (siehe Kapitel 2.6.2). Der Datenbackbone von PREEvision wird verwendet um das Modell zu persistieren (siehe Kapitel 2.6.3). Die Client-Anwendung von PREEvision basiert auf Eclipse und so kommen einige Plugins von PREEvision zur Anwendung um das Modell zu Laden und zu Speichern und für die Visualisierung des Modells in der graphischen Benutzeroberfläche. Auch wurden Teile des Metamodells von PREEvision wiederverwendet (siehe Kapitel 2.6.1).

3.4.2 Metamodell Generator

Das Java AST Metamodell kann aus der Java Sprachspezifikation von Oracle abgeleitet werden [20]. Die manuelle Erstellung des Metamodells wäre sehr aufwändig und zu fehlerträchtig. Daher wurde ein Metamodellgenerator im Rahmen dieser Arbeit entwickelt, der es erlaubt ein MDF Metamodell aus einer Backus Naur Form (BNF) zu generieren [76]. Damit kann ein Metamodell für jede Programmiersprache generiert werden, bei der die Definition der Sprache als BNF vorliegt. Zusätzlich zum Metamodell werden noch einige Hilfsklassen generiert, die den Export, Import und das Durchlaufen des Modells mittels des Besucher-Entwurfsmusters erlaubt [77].

Außerdem unterstützt der Generator das XML Schema Definition (XSD) Format für XML Dateien [78]. Der Grund für die Unterstützung von XSD ist, dass es innerhalb der Java-Projekte in Eclipse auch XML Dateien gibt. In einer späteren Ausbaustufe sollen diese Dateiinhalte feingranular als Modell abgespeichert werden. Das Ziel ist alle Dateiinhalte des Projekts im Modell abzulegen, so dass es keine Notwendigkeit mehr gibt Informationen oder Daten in Dateien abzuspeichern. Damit sind die Inhalte für Traceability Links referenzierbar und die Inhalte können unmittelbar ausgewertet werden ohne die Dateien zuerst in ein Modell übersetzen zu müssen. Außerdem können Links, die heute über Namenskonventionen existieren, explizit gemacht werden. Zum Beispiel werden OSGI Services in einer eigenen XML Datei definiert, die den vollständigen Klassennamen (Klasse, die den Service implementiert) enthält [44]. Hier wird über den Namen der Klasse ein Link zwischen Quellcode und der OSGI Servicedefinition definiert.

Eclipse stellte viel Funktionalität zum Parsen und Prozessieren von Java Quellcode zur Verfügung. Es gibt einen Java AST und einen Parser, der aus einer Java-Datei einen AST erstellt (siehe Kapitel 2.5.2.2). Um dieses und andere Funktionalitäten nutzen zu können, ist es von Vorteil, wenn das Metamodell sehr ähnlich zum Eclipse Java AST ist. Deswegen wurde der Generator dahingehend erweitert, dass er als Input die Eclipse Java AST Klassen verarbeiten und ein entsprechendes Metamodell erstellen kann. Dazu werden die Java AST Klassen über Reflection bezüglich ihrer Relationen und Attribute analysiert und dann für jede AST Klasse eine Metamodell Klasse generiert mit den entsprechenden Relationen und Attributen. Mit diesem zusätzlichen Feature des Generators ist es nun möglich das MDF Modell mit Eclipse Funktionalitäten bezüglich AST nutzen zu können.

Abbildung 26 und Abbildung 27 zeigen ein einfaches Beispiel für eine Klasse und den dazugehörigen MDF AST.

```
package vi.text;

import java.util.Collection;

/**
 * Commit Dialog
 */
public class CommitDlg {
  private Collection<Object> changedArtefacts;

  public CommitDlg(Collection<Object> artefacts) {
    this.changedArtefacts = artefacts;
  }

  public void openDialog(String title, String message) {
    if (title == null || message == null || changedArtefacts == null || changedArtefacts.size() == 0) {
      return;
    }
  }
}
```

Abbildung 26: Einfacher Beispielcode für die Illustration eines MDF AST

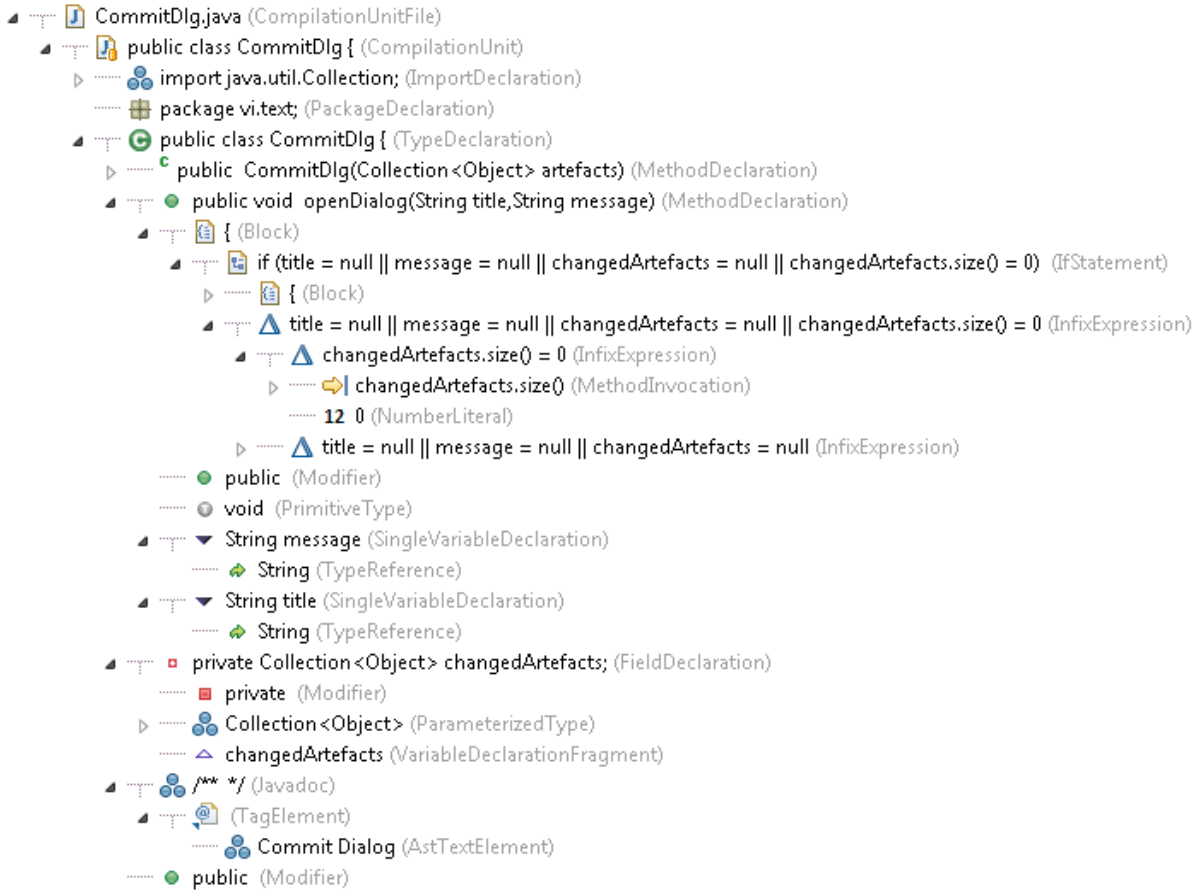


Abbildung 27: Der zum Beispielcode gehörende MDF AST

3.4.3 Traceability in den Quellcode

In diesem Kapitel werden die Metamodelle der unterstützten Prozessgebiete vorgestellt. Dabei werden Teile des PREEvision Metamodells wiederverwendet und erweitert (siehe Kapitel 2.6.1).

3.4.3.1 Testdatenmanagement

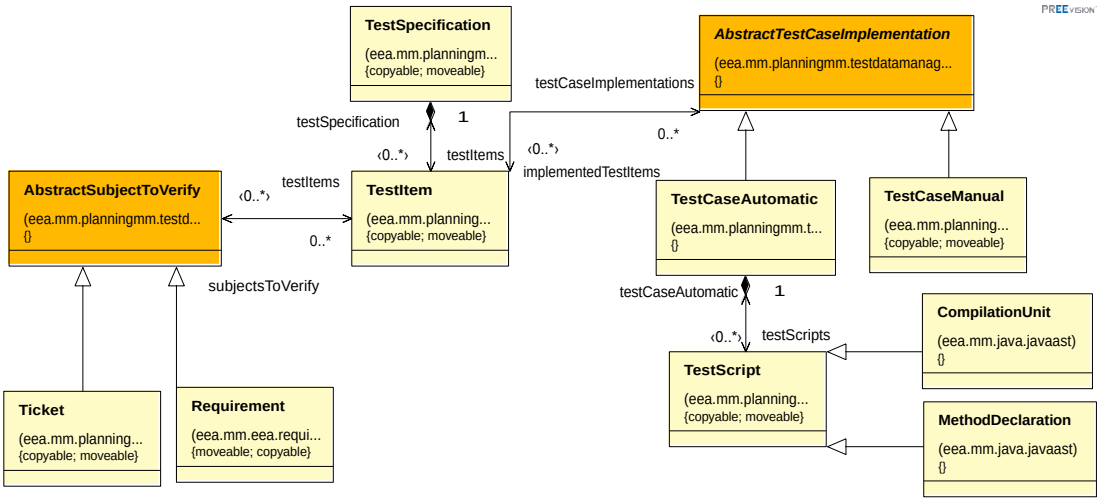


Abbildung 28: Metamodell für Testdatenmanagement

Die *TestSpecification* beschreibt wie funktionale Anforderungen getestet werden müssen, um die Qualität der Software sicherstellen zu können (siehe Abbildung 28). Die Spezifikation wird in natürlicher Sprache geschrieben unabhängig von der Implementierung der Tests und beinhaltet ein oder mehrere *TestItem*s. Typischerweise berücksichtigen die *TestItem*s unterschiedliche Anwendungsfälle und sie werden mit den *Requirement*s und *Ticket*s verlinkt, die das *TestItem* verifiziert.

Ein *Ticket* kann ein Änderungsauftrag oder ein Softwarefehler sein. Um sicherzustellen, dass ein Softwarefehler nicht wieder in einem späteren Release auftaucht und um die Testabdeckung der automatischen Tests zu erhöhen, macht es Sinn einen automatischen Test pro Softwarefehler anzulegen, der auf diesen Fehler testet. Ein *Requirement* repräsentiert nicht eine vollständige Anforderungsspezifikation, sondern eine einzelne Anforderung innerhalb der Spezifikation. Die Anforderung muss spezifisch sein und wird in natürlicher Sprache verfasst. Typischerweise ist diese Anforderung eine funktionale Anforderung, die unmittelbar in Quellcode umgesetzt werden kann. Die *Requirement*s werden hierarchisch in Anforderungspaketen organisiert, die zusammen eine vollständige Anforderungsspezifikation bilden.

Für jedes *TestItem* wird eine oder mehrere Testimplementierungen entwickelt, die ausgeführt werden können. Eine Testimplementierung kann ein manueller Test (*TestCaseManual*) oder ein automatischer Test (*TestCaseAutomatic*) sein. Der automatische Test kann mit einer Compilation Unit (zum Beispiel eine Unittest-Klasse) oder mit einer Methode der

Compilation Unit verlinkt werden. Die Metaklassen *CompilationUnit* und *MethodDeclaration* sind AST Artefakte. Damit existieren direkt in den Quellcode hinein bidirektionale Traceability Links. Die *MethodDeclaration* befindet sich unterhalb einer *CompilationUnit.* Wenn der Softwareentwickler die Testmethode umbenennt und oder in eine andere Testklasse verschiebt, so bleibt der Traceability Link zum Testfall und damit auch zum *Requirement* oder dem *Ticket* erhalten. Die Metaklasse *TestCaseAutomatic* ist verlinkt mit der *CompilationUnit* oder mit der *MethodDeclaration,* weil entweder die komplette Testklasse oder nur die Testmethode für den Testfall *TestCaseAutomatic* ausgeführt werden soll.

Traceability Links können in beide Richtungen gefolgt werden und so ist es möglich allen automatischen Test-Quellcode für eine bestimmte Anforderung und Ticket zu finden und der Softwareentwickler kann sofort erkennen welche Anforderungen oder Tickets eine bestimmte Testmethode oder Testklasse testet.

3.4.3.2 Anforderungs- und Änderungsmanagement

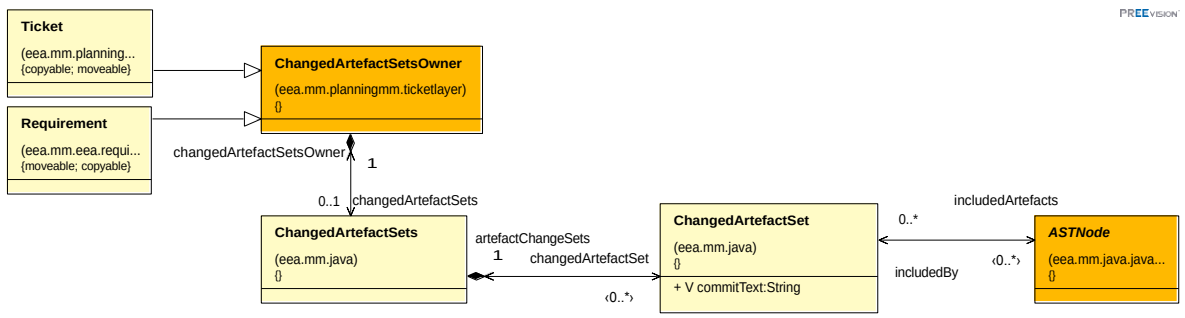


Abbildung 29: Metamodell für Anforderungs- und Änderungsmanagement

Die Metaklasse *ASTNode* ist die Basisklasse aller AST Artefakte im Metamodell (siehe Abbildung 29). Wenn der Softwareentwickler den Quellcode ändert, werden bereits existierende AST Artefakte geändert oder gelöscht und neue AST Artefakte werden angelegt. Diese Menge der geänderten AST Artefakte wird dem *ChangedArtefactSet* hinzugefügt. Das *ChangedArtefactSet* ist wiederum mit dem *Requirement* oder dem *Ticket* verknüpft. Damit ist der Änderungsgrund (Anforderung oder Ticket) mit dem geänderten Quellcode verlinkt und Traceability ist in beide Richtungen möglich.

Mit jedem geändertem AST Artefakt wird ein Traceability Link zu einer Anforderung oder einem Ticket angelegt. Über die Zeit können Links zu unterschiedlichen Anforderungen entstehen, weil der Quellcode für verschiedene Anforderungen relevant ist. Für die Anforderungen können alle Testklassen und Testmethoden ermittelt werden. All diese

Information zusammen (Quellcode – Anforderung – Testmethode) können dazu verwendet werden, um die automatischen Tests für geänderten Quellcode (AST Artefakte) zu ermitteln. Diese Testauswahlstrategie kann zum Beispiel während des Continuous Integration (CI) Lauf genutzt werden [15] (siehe Kapitel 5.5.1).

3.4.3.3 Dokumentation

Der Quellcode kann durch Kommentare direkt im Quellcode dokumentiert werden. Dabei kann der Softwareentwickler aber nur Text verwenden. Es gibt einige Formatierungsmöglichkeiten unter Verwendung von HTML Tags (zum Beispiel mit Javadoc), aber das Editieren ist schwierig. Es ist nicht möglich Bilder, Diagramme, Kalkulationstabellen oder Präsentationen zu verwenden.

Dateianhang

Die Unterstützung von Traceability Links zwischen Dokumenten jeglicher Art mit Informationen zum Quellcode und dem Quellcode selber ist eine wichtige Verbesserung. Diese Dokumente können an jeder Stelle im Quellcode platziert werden, weil Traceability Links zu jedem AST Artefakt möglich sind. Auch Design Dokumente können mit allen relevanten AST Artefakten verlinkt werden.

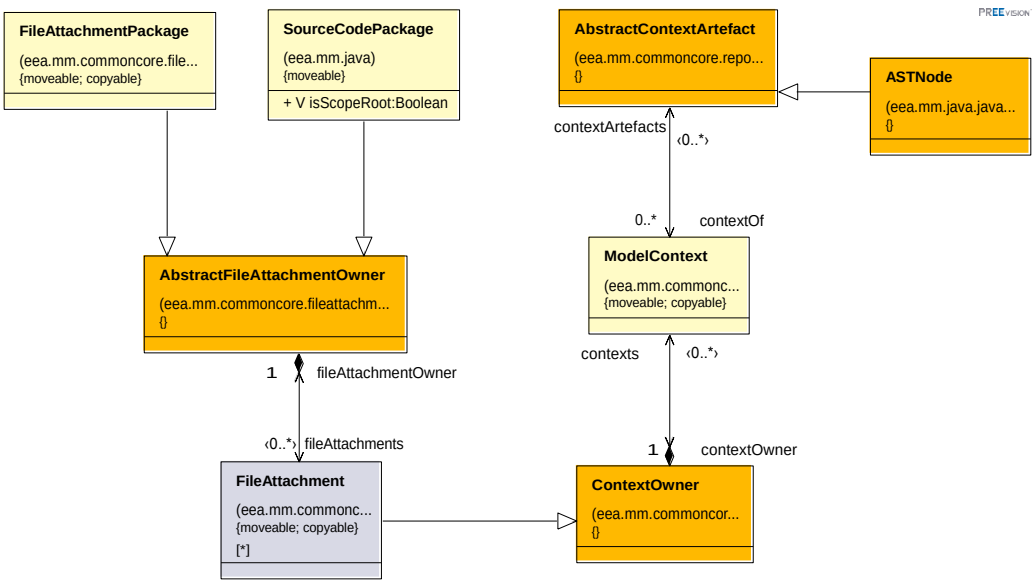


Abbildung 30: Metamodell für Dateianhang

Im Metamodell gibt es eine spezielle Metamodellklasse mit dem Namen *FileAttachment*, die es erlaubt Dateien mit dem Modell im Datenbackbone abzulegen (siehe Abbildung 30 und Kapitel 2.6.2). Die Metamodellklasse *FileAttachment* unterstützt jedes Dokumentenformat zum Beispiel WinWord-, Powerpoint- oder Excel-Dokumente. Der Dateianhang kann in

einem *SourceCodePackage* oder in einem *FileAttachmentPackage* platziert werden unabhängig von den AST Artefakten. Die *FileAttachmentPackage*s können wiederum *FileAttachmentPackage*s enthalten und Instanzen davon können an fast allen Stellen im Modell angelegt werden. Damit kann eine beliebige Ablagestruktur an der passenden Stelle im Modell erzeugt werden. Der Softwareentwickler kann den Dateianhang mit einem AST Artefakt über die Metamodellklasse *ModelContext* verlinken, um die Relevanz der Datei für diesen Quellcode zu dokumentieren. Zusätzliche Dokumentation in einem Dateianhang kann für jedes AST Artefakt sinnvoll sind, zum Beispiel eine Klasse, eine Methode, eine Anweisung oder eine Variablendeklaration. Wenn der Quellcode in eine andere Klasse verschoben wird, bleibt der Traceability Link trotzdem erhalten.

Kommentare

Neben dem Dateianhang gibt es eine weitere Möglichkeit zusätzliche Dokumentation im Quellcode zu hinterlegen.

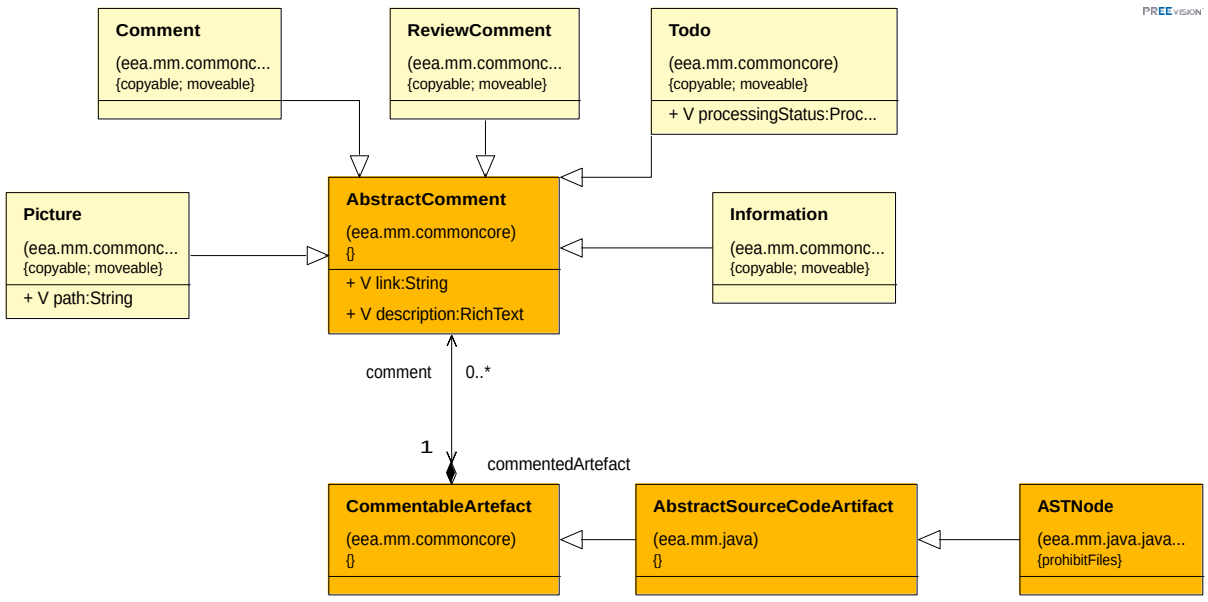


Abbildung 31: Metamodell für Kommentare

Die Basisklasse *AbstractComment* besitzt ein Attribut „description“ vom Typ Rich-Text und hat mehrere konkrete abgeleitete Klassen für unterschiedliche Anwendungsfälle: *Comment*, *ReviewComment*, *ToDo*, *Information*, *Picture* (siehe Abbildung 31). Der Rich-Text kann innerhalb der Anwendung über einen Rich-Text Editor bearbeitet werden und erlaubt unterschiedlichste Formatierungsmöglichkeiten inklusive dem Einfügen von Bildern und Tabellen im Text. Im Gegensatz zum Dateianhang wird das Kommentar-Artefakt über eine Kompositionsbeziehung an das AST-Artefakt gebunden. Das bedeutet, wenn das AST

Artefakt gelöscht wird, wird auch das Kommentar-Artefakt gelöscht. Im Gegensatz dazu verliert der Dateianhang nur den Traceability Link zum AST-Artefakt, wenn das AST Artefakt gelöscht wird und der Dateianhang bleibt weiterhin erhalten. Das Kommentar-Artefakt hat daher mehr Ähnlichkeit zum Kommentar direkt im Quellcode.

3.4.4 Implikationen eines AST Models

Die folgenden Implikationen ergeben sich aus der Verwendung eines Modells zum Speichern des Quellcodes anstelle von Text.

1. Für das Editieren des AST Modells wird ein spezieller AST Editor benötigt. Es ist nicht möglich aus dem AST Text zu generieren, den Text mit einem beliebigen Texteditor zu bearbeiten und dann den Text wieder in einen AST umzuwandeln, weil so Traceability Links in den Quellcode und die Historie einzelner AST Artefakte verloren gehen (für mehr Details siehe Kapitel 3.5). Deswegen wird ein spezieller Texteditor oder ein komplett neuer AST-Editor benötigt.
2. Die Syntax kann sich mit jeder neuen Java Version verändern und dann wird ein neues Metamodell für die neue Java Version benötigt. Der Quellcode von Java 1.8 lässt sich nicht in einem Metamodell für Java 1.7 abspeichern. Normalerweise sind die Änderungen in Java rückwärts kompatibel, so dass der Quellcode nicht angepasst werden muss. Daher ist eine aufwändige Migration des Modells mittels einer Modell-zu-Modell Transformation nicht notwendig [79]. Das Metamodell wird dann nur erweitert.
3. Was auch immer der Softwareentwickler in den Editor eingibt, muss in einen AST umgewandelt werden können, so dass es im Modell abgespeichert werden kann. Deswegen ist es nicht möglich Syntaxfehler im Modell abzuspeichern.
4. Wenn der Anwender temporär Quellcode auskommentiert zum Beispiel für Testzwecke, dann können Traceability Links verloren gehen. Es ist natürlich möglich die Kommentare im AST zu speichern, aber die ursprünglichen AST Artefakte mit den Traceability Links werden dann gelöscht. Nach dem Entfernen des Kommentars und damit der Wiederherstellung des ursprünglichen Quellcodes werden die AST Artefakte neu angelegt ohne die Traceability Links. Für diesen Anwendungsfall ist eine spezielle Funktionalität und Unterstützung notwendig.
5. Spezielle Formatierungen wie zum Beispiel eine zusätzliche Leerzeile oder zusätzliche Tabulatoren gehen verloren. Der Quellcode Text wird aus dem AST generiert und es

kommen in Eclipse hinterlegte Formatierungsregeln zur Anwendung. Damit wird eine einheitliche Formatierung unterstützt und der Quellcode für die Softwareentwickler einfacher zu lesen. In vielen Firmen gibt es Coding-Styleguides für das Schreiben von Quellcode, die solche Formatierungsregeln festlegen.

3.5 Java AST Editor

Für das Editieren des AST Modells wird, wie bereits erwähnt, ein spezieller AST Editor benötigt. Der Java Editor von Eclipse ist ein sehr mächtiger Editor mit vielen Features wie zum Beispiel Syntaxhervorhebung, Korrektur-Assistent, Quick-Assistent, integriertes Debuggen und einiges mehr (siehe Kapitel 2.5.2.2). Das Ziel ist diesen Editor weiterhin nutzen zu können und den Editor um die zusätzliche Funktionalität zum Editieren des AST Modells zu erweitern. Dieser Editor wird in weiterer Folge „Java AST Editor“ genannt [80].

Auch wenn der Eclipse Java Editor intern viel mit AST arbeitet (zum Beispiel für die Syntaxhervorhebung), so ist der Input und Output für den Editor weiterhin Text. Deswegen muss der AST aus dem Modell in Text umgewandelt werden vor dem Editieren und der Text muss in einen AST konvertiert werden nach dem Editieren bzw. beim Speichern des Texts. Dabei muss folgendes Problem gelöst werden (gezeigt am Beispiel der Umbenennung einer Methode): Wenn der Entwickler die Methode umbenennt und den Quellcode in das AST Modell speichert, wird eine neue Methodendeklaration angelegt und die bestehende Methodendeklaration gelöscht mit allen Traceability Links zu dieser Methodendeklaration. Während des Speicherns ist es unmöglich zu wissen, ob der Entwickler die Methode unbenannt hat oder die Methode gelöscht hat, weil sie nicht länger benötigt wird und eine neue angelegt hat. Das wird dadurch bestimmt, wie der Entwickler den Text editiert hat: Der Entwickler ändern den Namen der Methode im Text oder der Entwickler löscht den Text der Methode und beginnt eine neue Methode zu schreiben. Deswegen muss die Positionen der AST Artefakte im Text immer bekannt sein und bei jeder Textänderung korrigiert werden. Das wird durch folgende Features des Java AST Editors erreicht:

- Beim Öffnen der Java AST Editor werden die AST Artefakte aus dem Modell in Text konvertiert. Für jedes AST Artefakt wird die Startposition und Länge im erzeugten Text bestimmt und im Editor als Mapping-Information hinterlegt. Damit weiß der Editor für jede Position im Text welche Artefakte dort platziert sind. An einer konkreten Stelle können mehrere AST Artefakte vorhanden sein, die sich überlagern.

Das AST Artefakt, das in der Hierarchie des Abstract Syntax Baums auf der tiefsten Ebene sitzt, ist das Artefakt, welches im Editor sichtbar ist (siehe Abbildung 32).

- Wenn der Entwickler neue Zeichen eingibt oder bestehende Zeichen löscht, muss die Startposition und Länge der AST Artefakte korrigiert werden. Wenn der geänderte Text sich an der Stelle des AST Artefakts befindet, muss die Länge korrigiert werden. Für alle AST Artefakte hinter der geänderten Position muss die Startposition angepasst werden.
- Wenn der Entwickler Text innerhalb des Editors über Drag und Drop verschiebt, Text über die Zwischenablage ausschneidet und den Text wieder über die Zwischenablage einfügt, muss die Startposition und Länge aller relevanten AST Artefakte angepasst werden.
- Wenn der Entwickler Quellcode Text in die Zwischenablage über den Befehl „Ausschneiden“ übernimmt, wird nicht nur der Text in die Zwischenablage abgelegt, sondern alle AST Artefakte, die in diesem Textabschnitt vorhanden sind und deren Startposition und Länge. Beim Einfügen des Texts zurück in den Editor wird die Mapping-Information im Editor mit den Informationen über die AST Artefakte aus der Zwischenablage aktualisiert. Damit ist es möglich Quellcode von einer Klasse zu einer anderen zu verschieben ohne irgendwelche Traceability Links auf die verschobenen AST Artefakte zu verlieren.

Abbildung 32: Java AST Editor mit dem Mapping zu den AST Artefakten

Wenn der Text aus dem Editor gespeichert wird, dann werden aus dem Text wieder AST Artefakte erzeugt. Die AST Artefakte aus dem Editor besitzen keine Traceability Links, die eventuell im Modell existieren und deswegen müssen diese Traceability Links verschmolzen werden. MDF stellt eine mächtige Komponente zum Verschmelzen von Modellen zur Verfügung, die für diesen Zweck genutzt wird. Diese Komponente kann die Mapping-Informationen aus dem Editor nutzen, um die AST Artefakte richtig zuordnen zu können. Das Mapping liefert die Information, wo die AST Artefakte aus dem Modell im aktuellen Quellcode-Text positioniert sind. Die AST Artefakte aus dem Editor werden den AST Artefakten aus dem aktuellen Modell zugeordnet und können verschmolzen werden. Die AST Artefakte aus dem Modell werden dadurch mit den Änderungen, die sich aus der Bearbeitung des Quellcode-Text ergeben haben, aktualisiert.

Mit diesem Java AST Editor kann der Softwareentwickler den Quellcode bearbeiten wie bisher ohne zu wissen, dass der Quellcode als AST Artefakte im Modell abgespeichert wird. Aber der Java AST Editor kann nicht erkennen, dass der Entwickler nur eine Methode ändern will, obwohl der Entwickler den Text der Methode löscht und eine neue Methode mit dem gleichen Namen anlegt. Deswegen sollte der Softwareentwickler sich bewusst sein, dass er tatsächlich AST Artefakte mit Traceability Links editiert und der Entwickler muss verstehen, wann eine Methode neu erzeugt, geändert oder gelöscht wird: So lange der Entwickler den gesamten Text der Methode nicht löscht und nur den Text der Methode ändert, wird kein neues AST Artefakt für die Methode angelegt, sondern nur das bestehende AST Artefakt der Methode abgeändert. Mit diesem Wissen kann der Entwickler den Quellcode entsprechend seinen Absichten abändern.

3.6 Traceability Link Erzeugung und Visualisierung

Ein Fokus in dieser Arbeit liegt auf der Unterstützung des Softwareentwicklers für das Erstellen und Visualizern der Traceability Links. Für andere Rollen im Softwareentwicklungsprozess wie zum Beispiel Testmanager, Produktmanager oder Projektmanager sind noch zusätzliche Features und Reports denkbar und notwendig.

3.6.1 Testdatenmanagement und Dokumentation

3.6.1.1 Erstellen von Traceability Links

PREEvision stellt eine sogenannte „Modellansicht“ (1) zur Verfügung, die alle Artefakte des aktuell geladenen Modells darstellt (siehe Abbildung 33 und Kapitel 2.6.4).

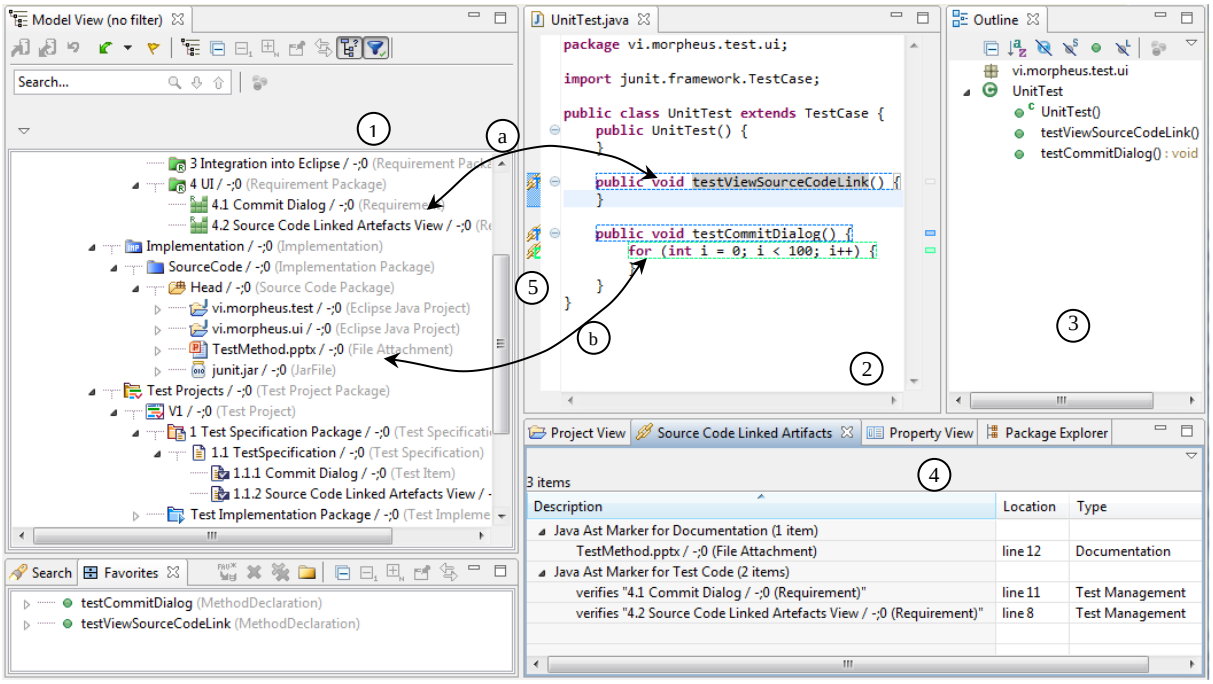


Abbildung 33: Erstellen und Visualisierern von Traceability Links

Die Ansicht ist in die Eclipse Entwicklungsumgebung integriert. Der Java AST Editor (2) kennt zu jeder Cursor Position das entsprechende AST Artefakt. Deswegen ist Drag und Drop in den Editor und aus dem Editor möglich.

Wenn der Softwareentwickler dokumentieren will, dass die neu angelegte Testmethode „testViewSourceCodeLink“ die Anforderung „Source Code Linked Artefacts View“ verifiziert, kann der Entwickler die Anforderung aus der Modellansicht in den Java AST Editor ziehen und auf der Testmethode fallen lassen (a). Alternativ kann der Entwickler den Text des Methodenamens aus der Editor ziehen und auf der Anforderung in der Modelansicht fallen lassen. Es ist möglich die „Outline“-Ansicht (3) als Drop Ziel oder als Startpunkt für die Drag und Drop Operation zu nutzen. Nach dem die Anforderung auf der Testmethode fallengelassen wurde, wird automatisch eine *TestSpezification* (wenn sie nicht bereits existiert), ein *TestItem* und ein *TestCaseAutomatic* im Modell angelegt und alle Artefakte inklusive der Testmethode und der Anforderung werden miteinander verlinkt. Natürlich kann der Entwickler auch ein bereits existierendes *TestItem* oder *TestCaseAutomatic* nutzen.

Eine ähnliche Drag und Drop Operation kann mit dem Dateianhang ausgeführt werden (b) mit dem Unterschied, dass der Dateianhang auf jedem AST Artefakt fallengelassen werden kann und nicht nur auf einer Methode wie bei einer Anforderung. Es sind beide Richtungen möglich von der Modellansicht in der Java AST Editor oder vom Java AST Editor in die

Modellansicht. Ein Kommentar kann über das Kontextmenü im Java AST Editor auf einem AST Artefakt angelegt werden.

3.6.1.2 Visualizern von Traceability Links

Für die Darstellung der Traceability Links innerhalb des Java AST Editor wurde die Marker-Funktionalität von Eclipse genutzt. Die Traceability Link Marker werden in der „Source Code Linked Arefakts“-Ansicht (4) gezeigt und auf der Marker-Leiste (5) im Editorbereich (siehe Abbildung 33). Außerdem wird der Text des AST Artefakts im Editor mit einem gestrichelten Rechteck umrandet. Die genaue Darstellung kann in den Eclipse Einstellungen festgelegt werden.

In der „Source Code Linked Arefakts“-Ansicht wird nach dem Traceability Link Typ gruppiert alle Traceability Links in dem aktuell geöffneten und selektierten Editor angezeigt. Es wird das verlinkte Artefakt und die Zeilennummer im Editor angezeigt. Durch einen Doppelklick auf eine Traceability Link Marker in der Ansicht wird der entsprechende Quellcode-Text hervorgehoben und das verlinkte Artefakt in der Modellansicht selektiert. Durch einen Mausklick auf das Icon in der Marker-Leiste werden alle verlinkten Artefakte in einem Popup-Fenster dargestellt und ein Tooltip für jedes Artefakt in dem Popup-Fenster zeigt zusätzliche, detaillierte Informationen zu dem Artefakt an (siehe Abbildung 34). Weil in einer Zeile mehrere Traceability Links vorhanden sein können, wird im Popup-Fenster eine Liste von Artefakten angezeigt.

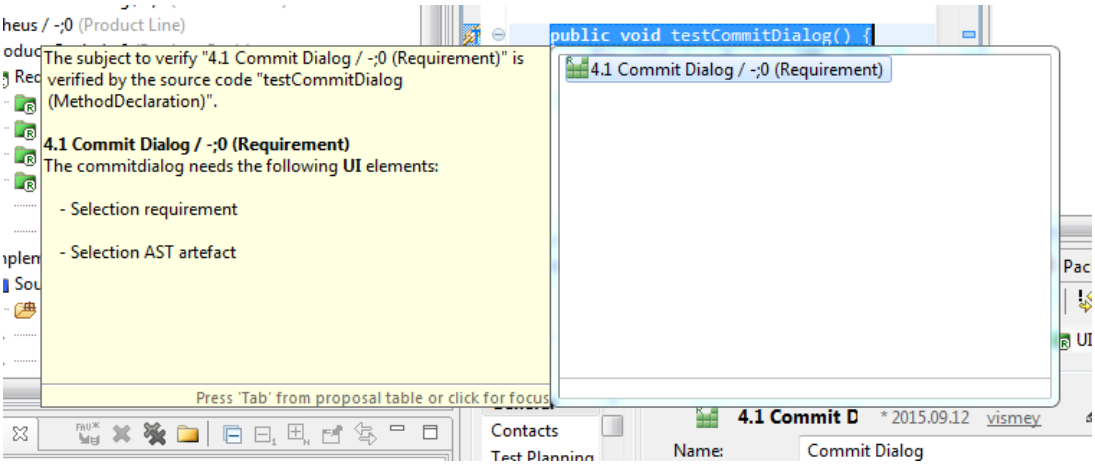


Abbildung 34: Popup-Fenster zeigt verlinkte Anforderungen

Ein Doppelklick auf dem verlinkten Artefakt im Popup-Fenster selektiert das Artefakt in der Modellansicht und wenn das Artefakt ein Dateianhang ist, wird die Datei von Server geladen und mit dem entsprechenden Editor geöffnet. Damit erkennt der Softwareentwickler während

des Sichtens des Quellcodes, dass zusätzliche Informationen für eine Quellcode-Zeile vorhanden ist (Anforderung, Ticket, Dokument oder Kommentar) und über einen oder höchstens zwei Klicks kann er die Zusatzinformation öffnen ohne den Editor verlassen und nach dieser Information suchen zu müssen. Das unterstützt den Entwickler beim Verstehen des Quellcodes.

Das Löschen von Traceability Links kann über ein Kontextmenü in der „Source Code Linked Artefacts“-Ansicht durchgeführt werden.

3.6.2 Anforderungs- und Änderungsmanagement

3.6.2.1 Erstellen von Traceability Links

Für die Nachverfolgbarkeit zwischen der Anforderung und dem Quellcode, der die Anforderung implementiert, werden Links zwischen diesen Artefakten benötigt. Des Weiteren können Traceability Links zwischen Tickets und dem Quellcode, der das Ticket löst, erstellt werden. Anforderungen und Tickets werden an dieser Stelle gleichbehandelt. Deswegen kann in diesem Kapitel der Ausdruck Anforderung immer mit Ticket ersetzt werden.

Der Aufwand für das Verlinken der Artefakte kann enorm sein, wenn das nach dem Abschluss der Entwicklungsaufgabe im Nachhinein versucht wird. Die beste Zeit, um diese Artefakte zu verlinken, ist während der Abgabe des geänderten Quellcodes [9]. Bei der Abgabe werden die Änderungen in den Datenbackbone gespeichert und so werden die Änderungen für alle Softwareentwickler sichtbar (siehe Kapitel 2.6.5). Der Entwickler muss einen Grund für die Änderungen angeben. Das erfolgt über die Auswahl der Anforderung, für die die Änderungen durchgeführt wurden. Dazu wurde die Abgabe innerhalb von PREEvision um einen zusätzlichen Dialog erweitert, wo der Entwickler die Anforderung auswählen kann (siehe Abbildung 35). Zusätzlich kann ein Abgabe-Kommentar eingegeben werden. Wenn der Entwickler parallel an mehreren Anforderungen arbeitet, muss der Entwickler entsprechend alle Anforderungen auswählen und muss dann entscheiden, welches geänderte AST Artefakt zu welcher Anforderung gehört. Das kann in diesem Dialog durchgeführt werden.

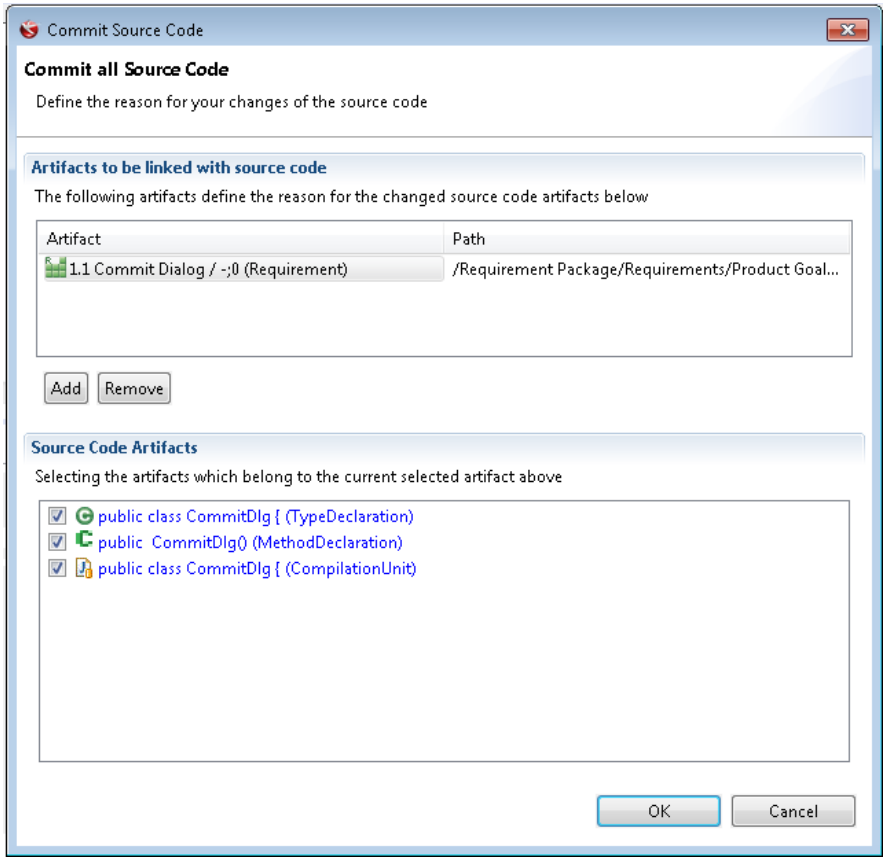


Abbildung 35: Abgabe Dialog für Zuordnung von Anforderungen zu AST Artefakten

Mit jeder Abgabe von Quellcode wird ein *ChangedArtefactSet* angelegt mit folgenden Informationen: Autor, Datum der Abgabe, Abgabe-Kommentar und alle geänderten AST Artefakte. Das *ChangedArtefactSet* wird mit der selektierten Anforderung oder Ticket verlinkt.

Mylyn

Die Anforderungen und Tickets können über die von PREEvision bereitgestellten Lifecycle-Management Funktionalität Softwareentwickler zugewiesen werden (siehe Kapitel 2.6.7). Der Lifecycle der Anforderung oder des Tickets wird dabei auf einen Zustand weitergeschaltet, der dem Softwareentwickler signalisiert, dass das Artefakt jetzt für die Abarbeitung freigegen ist (zum Beispiel der Lifecycle Zustand „Assigned“). Dem Lifecycle-Zustand wird dann eine Person zugeordnet, die dafür verantwortlich ist (zum Beispiel der Entwickler, der das Ticket lösen soll). Die Anforderungen oder das Tickets werden damit zu einer Aufgabe für die zugewiesene Person.

Mylyn [47] kann den Softwareentwickler bei der Verwaltung dieser zugeordneten Aufgaben unterstützen (siehe Kapitel 2.5.3). Die Anforderungen und Tickets werden dabei als

Aufgaben nach Mylyn importiert. Dazu wurde ein neuer Mylyn Repository Connector entwickelt, der es erlaubt sich mit dem PREEvision Modell zu verbinden. Mit diesem neuen Connector kann der Entwickler in Mylyn ein PREEvision Task Repository anlegen (siehe Abbildung 36).

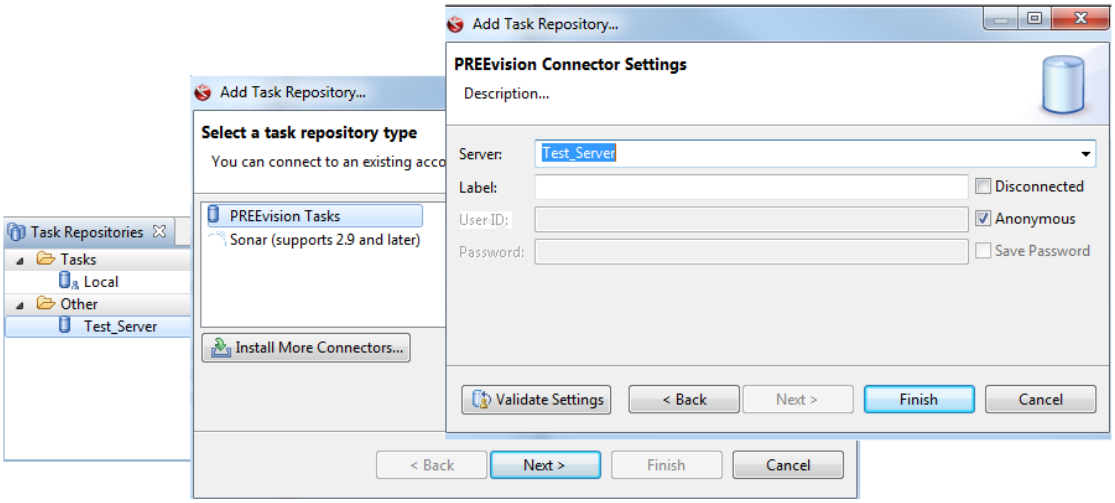


Abbildung 36: Erstellung eines neuen PREEvision Task Repository

Der PREEvision Repository Connector durchsucht dann das PREEvision Modell nach allen Anforderungen und Tickets, die dem aktuell angemeldeten Anwender zugeordnet sind und die wichtigsten Daten (Identifikationsnummer, Titel, Beschreibung und Typ) werden nach Mylyn übernommen.

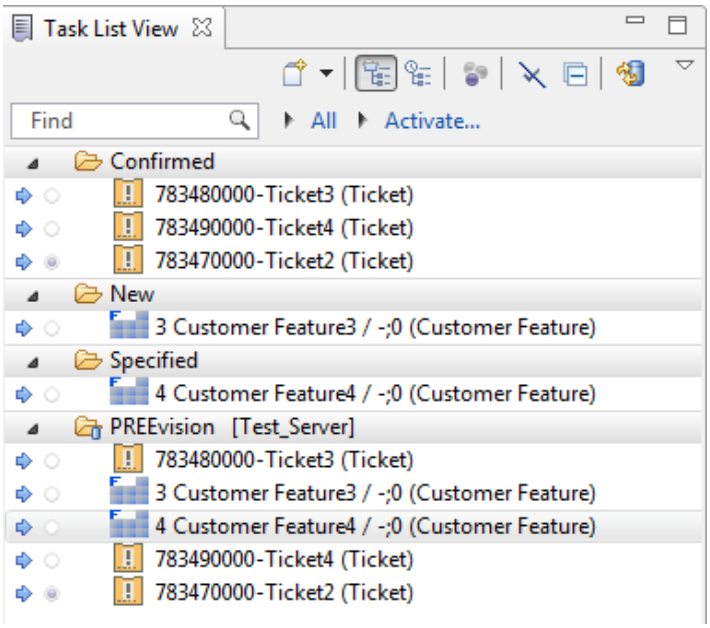


Abbildung 37: Mylyn Aufgabenliste mit Tickets und Features

Diese importierten Aufgaben sind in einer Mylyn Aufgabenliste für den Entwickler sichtbar und werden nach dem Lifecycle Zustand gruppiert. In Abbildung 37 sieht man sowohl Tickets als auch Features in der Liste. Features können ähnlich wie Anforderungen ebenfalls einem Entwickler zur Bearbeitung zugeordnet werden. Der Entwickler kann in dieser Liste den Lifecycle Zustand des Artefakts über ein Kontextmenü weiterschalten, wenn zum Beispiel die Aufgabe abgeschlossen ist.

Der Entwickler aktiviert dann die Aufgabe in der Aufgabenliste bevor er die ersten Änderungen für diese Aufgabe im Quellcode durchführt. Mit der Aktivierung wird ein sogenannter Kontext für die Aufgabe angelegt. Mylyn speichert alle angefassten und veränderten Artefakte in diesem Kontext bis die Aufgabe wieder deaktiviert wird. Der Kontext wird zusätzlich im Modell von PREEvision zur Anforderung oder zum Ticket in einem Dateianhang abgespeichert und steht damit anderen Softwareentwicklern zur Verfügung. Folgende Möglichkeiten ergeben sich dadurch:

- Während der Abgabe des Quellcodes und der damit notwendigen Zuordnung der Anforderung oder des Tickets zu den veränderten AST Artefakten, kann der Entwickler den Kontext nutzen, um zu sehen welche AST Artefakte mit welcher Anforderung oder Ticket verändert wurden.
- Ein Entwickler kann beim Sichten von Quellcode über die Traceability Links die dazugehörigen Anforderungen und Tickets finden. Über diese Artefakte kann er auf den Kontext zugreifen und sieht welcher zusätzliche Quellcode in diesem Bereich oder zu diesem Thema noch relevant ist. Bei großen Softwareprojekten mit tausenden von Klassen kann das eine wichtige Hilfestellung sein, den relevanten Quellcode zu finden.
- Wenn ein Entwickler die Umsetzung einer Anforderung oder eines Tickets von einem anderen Entwickler übernimmt, bekommt er damit auch den Kontext des anderen über das Modell und hat sofort den relevanten Quellcode im Zugriff, um die Aufgabe fortzusetzen.

3.6.2.2 Visualizern von Traceability Links

Eclipse stellt die Funktionalität bereit, um eine sogenannte Annotation Leiste im Editor Bereich darzustellen. Im Java Editor wird diese Leiste dazu verwendet, um Informationen über die letzte Änderung in der entsprechenden Quellcode-Zeile anzuzeigen. Diese Information wird aus dem Software Configuration Management (SCM) Repository geladen.

Neben dem Autor und dem Änderungsdatum wird der Abgabe-Kommentar in einem Popup-Fenster angezeigt. Die Farbe der Annotation repräsentiert entweder den Autor (unterschiedliche Farben für unterschiedliche Autoren) oder das Datum (neuere Änderungen werden in helleren Farben dargestellt).

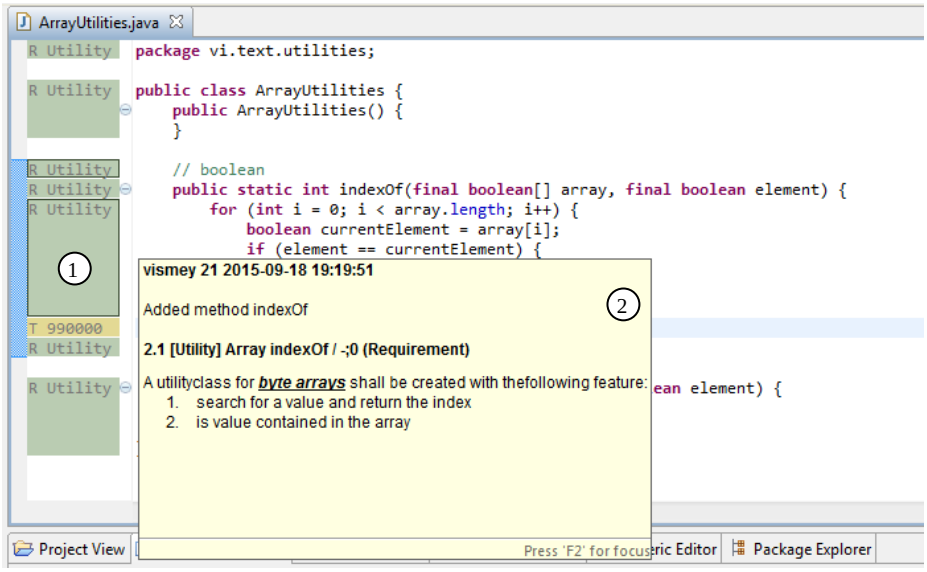


Abbildung 38: Annotation Leiste zeigt den Änderungsgrund für die Quellcode-Zeilen

Für diese Arbeit wurde die Annotation Leiste dahingehend verändert, dass die notwendigen Informationen jetzt aus dem Modell gelesen werden. Der Traceability Link vom Quellcode zur Anforderung oder zum Ticket kann nun dazu verwendet werden, um den Titel und den Inhalt dieser Artefakte zu erhalten. Titel und Inhalt werden mit dem Namen des Autors, dem Änderungsdatum und dem Abgabe-Kommentar in der Annotation Leiste (1) und im Popup-Fenster (2) angezeigt (siehe Abbildung 38). Bezüglich der Farben in der Annotation Leiste werden die bereist von Eclipse bekannten Modi (Autor und Datum) weiterhin unterstützt und noch ein neuer Modus eingeführt: Über die Farbe wird die Anforderung oder das Ticket repräsentiert. Dabei wird jeder Anforderung und jedem Ticket eine Farbe zugeordnet und in der Annotation Leiste für jede Quellcode-Zeile angezeigt. Damit bekommt der Entwickler einen sehr schnellen Überblick, welche Zeilen für welche Anforderungen oder Tickets geändert wurden.

Die Annotation Leiste des Java Editors kann nur Informationen über die letzte Änderung einer Quellcode-Zeile anzeigen. Obwohl es auch möglich ist, zwei beliebige Versionen der Quellcode-Datei zu vergleichen, kann ein einzelnes AST Artefakt in den zwei Versionen der Datei nicht verglichen werden. Eclipse ist nicht in der Lage die zwei passenden AST

Artefakte in den zwei Versionen der Datei zu finden, weil Position und/oder Name der AST Artefakte sich verändert haben könnten. Eine typische Situation ist folgende: der Entwickler will wissen ob und wenn ja, wann, wer und warum eine „For-Schleife“ über die letzten zwei Jahren geändert wurde. Durch den Vergleich verschiedener Versionen der Datei kann der Entwickler versuchen die „For- Schleife“ in den alten Versionen der Datei zu finden. Das kann sehr zeitintensiv und schwierig werden, weil die „For- Schleife“ sich an einer ganz anderen Stelle in der Datei befinden und sich bezüglich der beinhalteten Anweisungen verändert haben kann. Wenn die „For- Schleife“ von einer Datei in eine andere Datei verschoben worden ist, dann must der Entwickler alle in dieser Abgabe geänderten Dateien überprüfen um die „For-Schleife“ zu finden. Desto mehr Änderungen in den verschiedenen Versionen der Datei neben der Änderung der „For-Schleife“ durchgeführt wurden, desto schwieriger wird es die relevanten Quellcode Stellen überhaupt zu finden. Die Aufgabe kann damit beliebig kompliziert und aufwändig werden.

Mit der Speicherung der AST Artefakte im PREEvision Datenbackbone ist die genaue Historie jedes einzelnen AST Artefakts bekannt, weil nicht nur die letzte Version archiviert wird, sondern mit jeder Abgabe eine neue Version des AST Artefakts erzeugt wird (siehe Kapitel 2.6.5). Diese Versionen können vom Server abgefragt werden. Über ein Kontextmenü in der Annotation Leiste oder direkt im Editor kann der Entwickler detaillierte Informationen über die Historie eines AST Artefakts abrufen (siehe Abbildung 39). In einem Popup-Fenster wird jede Abgabe, indem das AST Artefakt verändert wurde, in einer Liste dargestellt (in diesem Beispiel die „return“ Anweisung) mit den Informationen wer und wann die Änderung durchgeführt hat. Ein Tooltip-Fenster beinhaltet Informationen über die verlinkten Anforderungen und Tickets und dem Abgabe-Kommentar (Änderungsgrund).

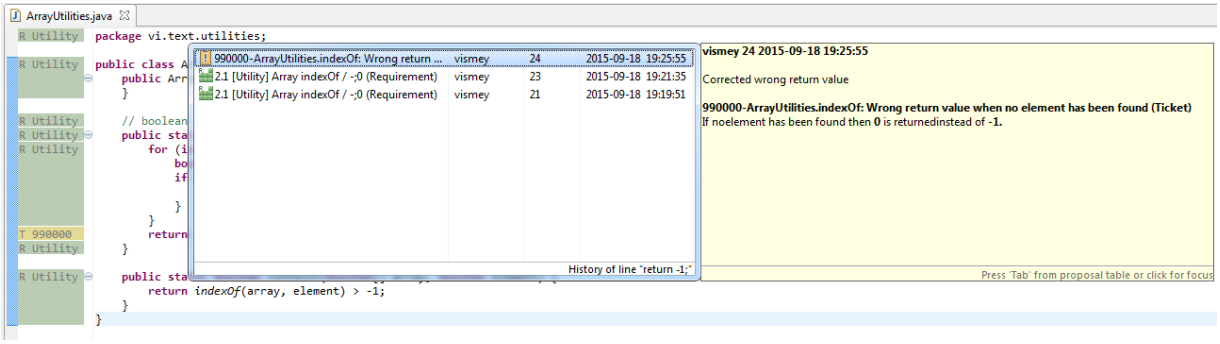


Abbildung 39: Popup-Fenster zeigt detaillierte Historie des AST Artefakts "return -1"

Zwei Einträge in der Liste (entsprechen zwei Versionen des AST Artefakts) können selektiert und verglichen werden. Dafür wird die komplette Compilation Unit, wo die Version des AST Artefakts enthalten ist, aus dem Datenbackbone geladen. Die zwei Versionen der Compilation Unit werden in Quellcode Text umgewandelt und werden dann in einem Vergleichseditor von Eclipse dargestellt. Der Vergleichseditor zeigt die beiden Quellcode Texte nebeneinander an, um einen direkten Vergleich der Texte zu ermöglichen. Wenn das AST Artefakt in der Historie verschoben wurde von einer Compilation Unit zu einer anderen (zum Beispiel eine Methode 1 wird von Compilation Unit 1 nach Compilation Unit 2 verschoben), dann werden zwei verschiedene Compilation Units geladen und verglichen mit zwei verschiedenen Versionen desselben AST Artefakts (siehe Abbildung 40).

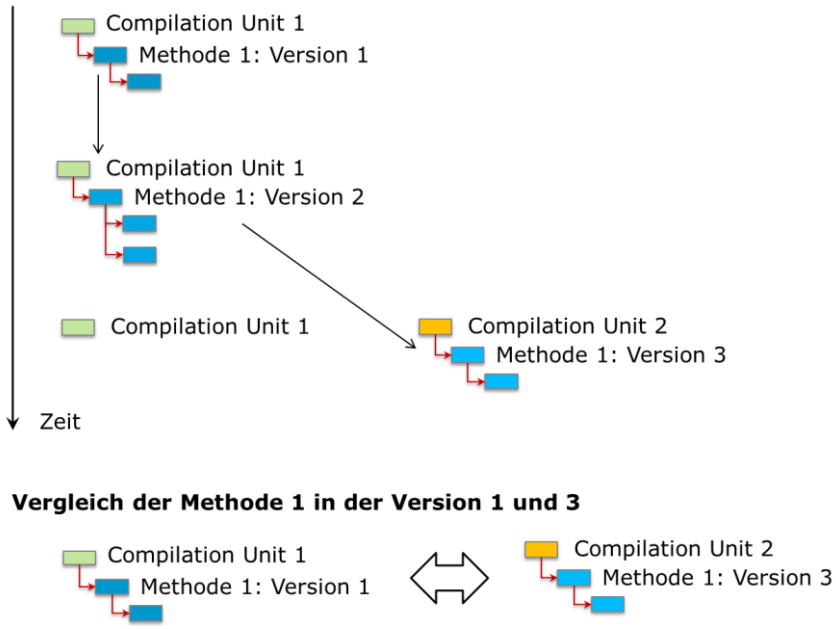


Abbildung 40: Vergleich von zwei Versionen in unterschiedlichen Compilation Units

Die verlinkte Anforderung oder das verlinkte Ticket haben auch eine Historie. Ein Ticket, das eine Softwarefehler beschreibt, wird sich in der Regel inhaltlich nicht grundlegend ändern (höchstens mit zusätzlichen Informationen angereichert), aber ein Ticket mit einem Änderungsauftrag oder eine Anforderung können über die Zeit substantiell geändert werden. Die Strategie ist dabei nicht Traceability Links zu löschen, weil sich der Inhalt geändert hat und damit der Link eventuell nicht mehr gültig ist, sondern die vollständige Historie bleibt erhalten und kann vom Softwareentwickler eingesehen werden. Das heißt, der Softwareentwickler kann den aktuellen Inhalt der Anforderung zu dem AST Artefakt einsehen als auch den Inhalt zu dem Zeitpunkt als der Traceability Link gesetzt wurde und natürlich für jede Zwischenversion. Der Entwickler kann damit erkennen, ob der Inhalt sich signifikant

geändert hat und was die ursprüngliche Vorgabe an den Entwickler beziehungsweise was die ursprüngliche Absicht bei der Entwicklung des Quellcodes war.

3.7 Eclipse Integration

Obwohl Eclipse intern den Abstract Syntax Tree (AST) für verschiedene Features nutzt (zum Beispiel für Refaktorierungen), wird schlussendlich aller Quellcode in Textdateien auf der Festplatte gespeichert. Außerdem werden noch eine ganze Reihe von weiteren Dateien in Projekten und Verzeichnissen in einem sogenannten Workspace auf der Festplatte abgelegt, die alle für den Build des Projekts notwendig sind (siehe Kapitel 2.5.2.1). Mit dieser Arbeit soll nicht nur der Quellcode in einem AST Modell gespeichert werden, sondern der komplette Workspace von Eclipse soll schlussendlich aus dem Modell geladen und in das Modell gespeichert werden. Damit besteht keine Notwendigkeit mehr weitere Verzeichnisse oder Dateien in einem anderen Software Configuration Management (SCM) abzulegen. Alle Daten werden konsequent in einem Modell und damit auch in einem Repository abgespeichert inklusive des Quellcodes mit seinen Projekten und den Artefakten aus den verschiedenen Prozessgebieten wie Tickets, Anforderungen, Testfällen und so weiter. Es können weitere Traceability Links im Metamodell definiert werden, wobei eben potenziell alle Artefakte als Link-Targets genutzt werden können, auch alle Artefakte, die innerhalb des Eclipse Workspaces liegen. Die Eclipse Funktionalität soll dabei wie bisher weiter genutzt werden können und es sollen Änderungen am Quellcode von Eclipse vermieden, sondern nur die bestehenden Erweiterungsmöglichkeiten von Eclipse genutzt werden. Damit ist ein Umstieg auf eine neuere Eclipse Version mit adäquatem Aufwand möglich.

3.7.1 Workspace Metamodell

Für die Ablage der Eclipse Projekte im Modell wird natürlich ein Metamodell benötigt. Das Metamodell orientiert sich dabei sehr stark an der Struktur der Verzeichnisse im Eclipse Workspace, um ein einfaches Mapping zu ermöglichen. Eclipse kennt ein generisches Projektkonzept mit Ressource-Verzeichnissen und –Dateien (siehe Kapitel 2.5.2). Diese generischen Projekte sind die Basis für die spezialisierten Projekte wie zum Beispiel für die Verwaltung von Java Projekten. Dementsprechend wurde zuerst ein Metamodell für das generische Projektkonzept definiert (siehe Abbildung 41).

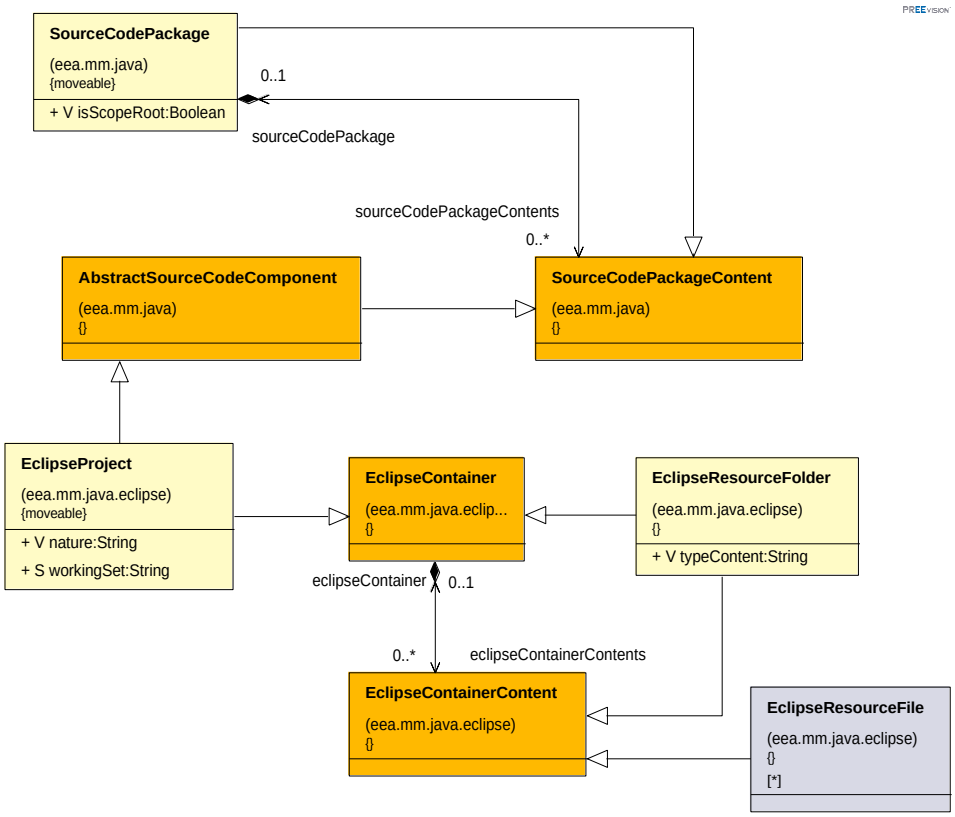


Abbildung 41: Metamodell für ein generisches Projekt

Die *SourceCodePackage*s können *SourceCodePackage*s enthalten und damit kann eine beliebige hierarchische Struktur aufgebaut werden. Die Eclipse-Projekte können unter *SourceCodePackage*s angelegt werden. Innerhalb des *EclipseProject*s können dann *EclipseResourceFolder*s und *EclipseResourceFile*s angelegt werden.

In Eclipse gibt es mittels der sogenannten Working Sets die Möglichkeit über nur eine Hierarchieebene Projekte zu gruppieren. Bei einer großen Anzahl von Projekten (zum Beispiel in großen Produkten mit über 1000 Projekten) ergeben sich dadurch sehr lange und unübersichtliche Listen von Projekten. Das macht das Finden von Projekten und Quellcode für den Softwareentwickler sehr kompliziert und es ist sehr schwierig ein Überblick über alle Projekte zu bekommen. Über die *SourceCodePackage*s kann eine viel tiefere und aussagekräftigere Hierarchie aufgebaut werden mit viel kürzeren Listen. Da die Projekte als Modellartefakte vorliegen, können sie über Traceability Links referenziert werden und so mit zusätzlicher Dokumentation verlinkt werden. Denkbar sind weitere zusätzliche Gruppierungen der Projekte nach unterschiedlichen Kriterien. All das kann den Softwareentwickler im Verstehen des Quellcodes unterstützen.

Für die Ablage von Java Projekten wurde das Metamodell erweitert (siehe Abbildung 42). Ein Java Projekt wird in der Regel für eine Komponente im Gesamtprodukt angelegt zum Beispiel für ein Java Archive (JAR) oder ein Executable (EXE).

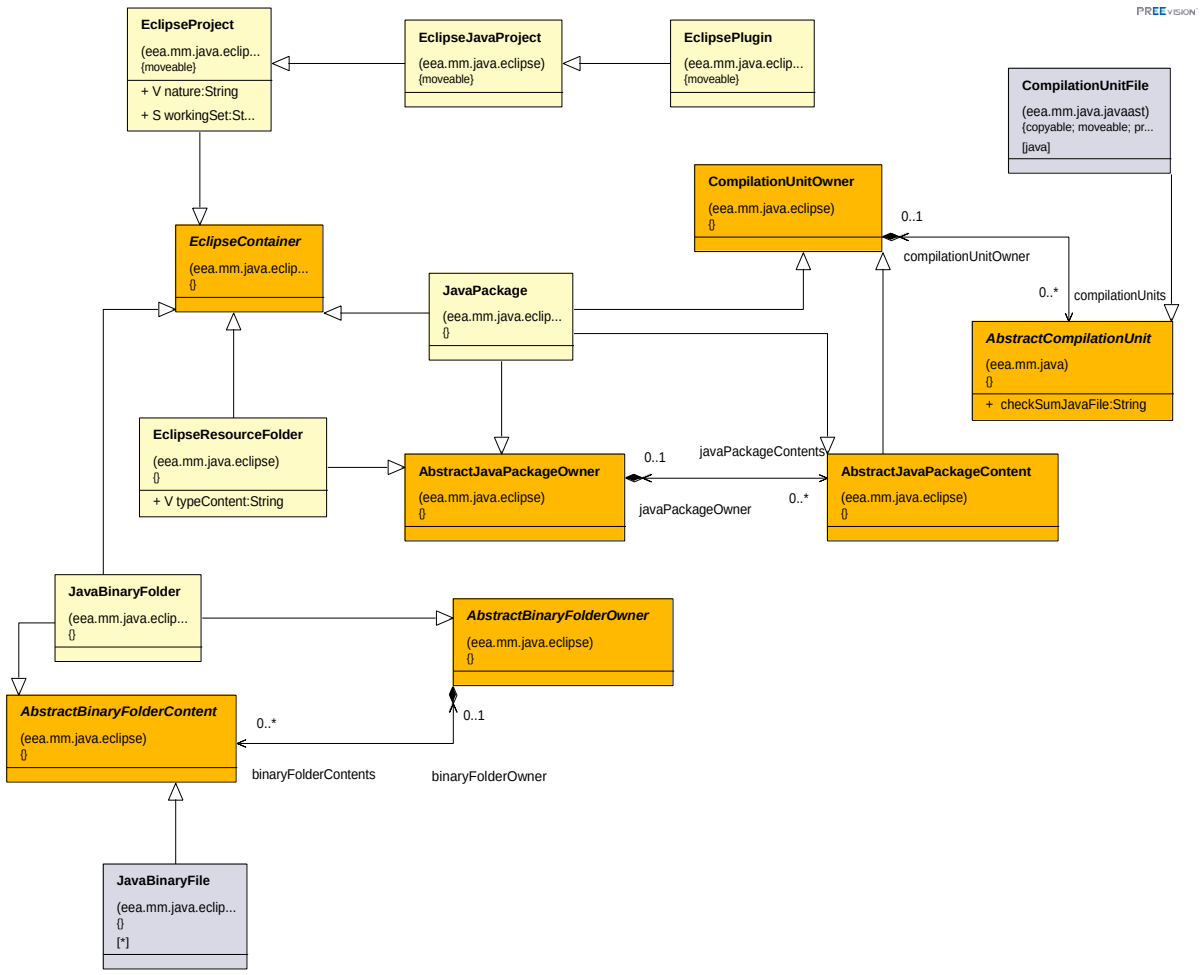


Abbildung 42: Metamodell für die Ablage von Java Projekten

Das *EclipseProject* kennt zwei Spezialisierungen: *EclipseJavaProject* für einfache Java Projekte und *EclipsePlugin* für ein Eclipse Plugin zur Erweiterungen einer Eclipse Rich Client Platform (RCP) (siehe auch Kapitel 2.5.1). Das *EclipseProject* kann neben *EclipseResouceFolder* und *EclipseResourceFile* auch *JavaBinaryFolder*s mit *JavaBinaryFile*s und *JavaPackage*s mit C*ompilationUnitFile*s enthalten. Unter der Metadatenklasse C*ompilationUnitFile* wird der Abstract Syntax Tree (AST) der jeweiligen Compilation Unit abgelegt. Es gibt zwei Besonderheiten in diesem Metamodell:

1. Es werden mit der Metadatenklasse *JavaBinaryFile*s auch Binary-Dateien im Modell abgelegt (in Java die Class-Dateien). Der Grund dafür ist, dass der Continuous Integration Build dadurch beschleunigt werden kann (siehe für Details Kapitel 5.4).

2. Die Metadatenklasse C*ompilationUnitFile* ist offensichtlich eine Datei obwohl der Quellcode doch als AST abgespeichert wird. Grundsätzlich wäre an dieser Stelle keine Datei notwendig. Für einen ersten Prototyp der Anwendung ist es jedoch sinnvoll den Quellcode nochmals redundant in einer Datei abzulegen, bis die Anwendung stabil funktioniert.

In Abbildung 43 ist ein Beispiel für ein Eclipse Java Projekt im Modell dargestellt.

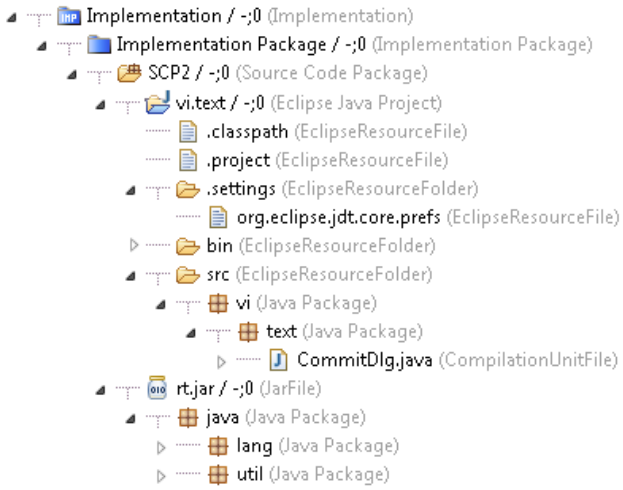


Abbildung 43: Beispiel für ein Eclipse Java Projekt im Modell

3.7.2 MDF Dateisystem

3.7.2.1 Konzept des MDF Dateisystems

Eclipse muss nun so angepasst werden, dass der Workspace aus dem Modell gelesen wird anstatt von der Festplatte ohne dabei größere Komponenten von Eclipse austauschen zu müssen und damit unter Umständen Funktionalität von Eclipse zu verlieren. Eclipse stellt einen Extension Point zur Verfügung, der es erlaubt ein eigenes Dateisystem bereitzustellen. Für das MDF Modell wurde daher ein MDF Dateisystem entwickelt und in Eclipse als Erweiterung registriert. Damit wird jeder Lese- und Schreibzugriff von Eclipse auf eine Datei oder ein Verzeichnis vom MDF Dateisystem verarbeitet. Das MDF Dateisystem liest dann die Datei und die Verzeichnisstruktur nicht von der Festplatte, sondern aus dem Modell und schreibt die Dateiinhalte und Verzeichnisse in das Modell.

Wenn der Entwickler zum Beispiel ein neues Verzeichnis oder eine neue Datei über den Package Explorer von Eclipse (über die Benutzeroberfläche von Eclipse) anlegt, dann wird der Aufruf von Eclipse an das MDF Dateisystem weitergeleitet, welches die Artefakte im Modell anlegt. Wenn eine Datei oder ein Verzeichnis umbenannt wird, wird über das MDF

Dateisystem der Name des Artefakts im Modell umbenannt. Nach dem Löschen von Modellartefakten zum Beispiel durch ein Aktualisieren aus dem Repository und der anschließenden Aktualisierung der Ansichten in Eclipse, werden die Dateien und Verzeichnisse in Eclipse verschwinden, weil Eclipse über das MDF Dateisystem die aktualisierten Daten ausliest. Im Falle von Quellcode-Dateien (Java-Dateien) zum Beispiel beim Öffnen einer Java-Datei über den Java AST Editor liest das MDF Dateisystem die AST Artefakte aus dem Modell aus, konvertiert die Artefakte zu Text und reicht den Text an Eclipse als Dateiinhalt weiter. Während des Speicherns der Datei gibt Eclipse den geänderten Text an das MDF Dateisystem weiter, welches den Text wieder in AST Artefakte umwandelt und in das aktuelle Modell integriert (siehe Kapitel 3.5). Mit der Verwendung des MDF Dateisystems bleiben alle Features von Eclipse wie gewohnt erhalten wie zum Beispiel die Wizards, der Build, die Ansichten, die Editoren oder die lokale Historie einer Datei. Für die lokale Historie speichert Eclipse sich die verschiedenen Änderungen an den Dateien lokal ab, die dann über ein Kontextmenü als Änderungshistorie abgerufen werden und die lokalen Versionen auch miteinander verglichen werden können.

3.7.2.2 Implementierung des MDF Dateisystems

Für die Bereitstellung eines eigenen Dateisystems müssen zwei Interfaces von Eclipse implementiert werden (siehe Abbildung 44).

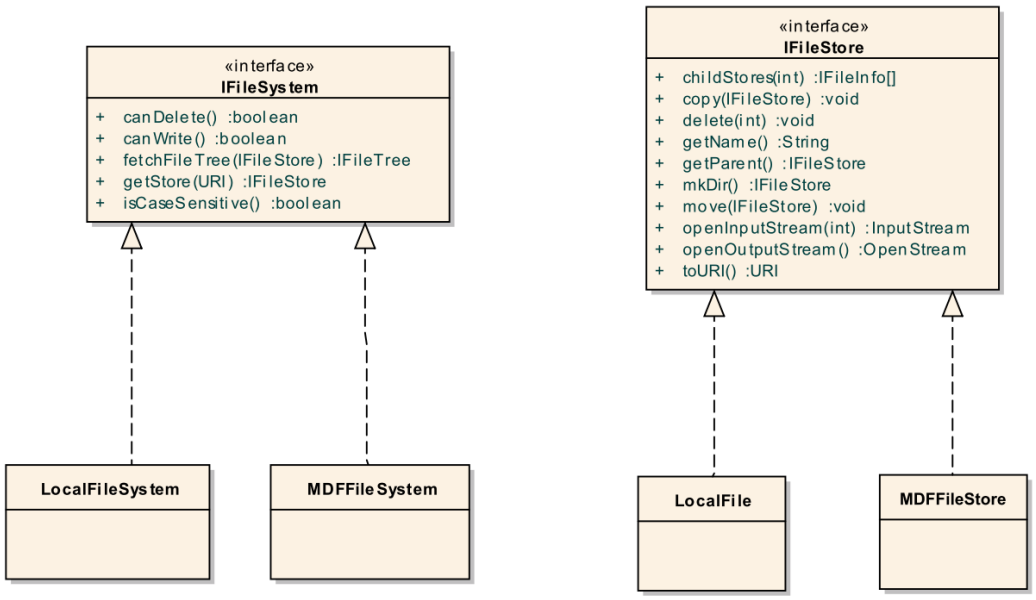


Abbildung 44: Interfaces für die Implementierung eines Dateisystems

Für die Interfaces gibt es eine Default-Implementierung von Eclipse für das Lesen von und Schreiben auf die Festplatte: LocalFileSystem und LocalFile. Die Implementierung für das MDF Dateisystem ist in den Klassen MDFileSystem und MDFFileStore umgesetzt. Über das Interface IFileSystem werden allgemeine Informationen über das Dateisystem bereitgestellt, wie zum Beispiel ob auf das Dateisystem geschrieben werden kann. Über die Methode getStore dieses Interfaces kann zu einer Uniform Resource Identifier (URI) ein IFileStore abgefragt werden (siehe Kapitel 3.7.2.3). IFileStore repräsentiert in der Default-Implementierung von Eclipse eine konkrete Datei oder ein konkretes Verzeichnis auf der Festplatte und in der MDFFileStore Implementierung ein Artefakt aus dem Modell.

Das IFileStore Interface definiert einige Operationen, die von der jeweiligen Implementierung bereitgestellt werden müssen: „move“, „copy“, „delete“, „openInputStream“, „openOutputStream“. LocalFile führt diese Operation auf der Festplatte mit den entsprechenden Dateien und Verzeichnissen aus. MDFFileStore wendet diese Operationen auf das Modell an. Wenn zum Beispiel die Methode „move“ auf dem MDFFileStore aufgerufen wird, dann wird das Artefakt im Modell an die durch den Parameter der „move“-Methode vorgegebene neue Stelle im Modell verschoben. Durch die „delete“-Operation wird das Artefakt im Modell gelöscht. Die Open-Stream-Methoden öffnen Streams zum Lesen und Schreiben der Dateiinhalte. Die MDFFileStore-Implementierung greift hierfür auf die Dateiartefakte im Modell zu beziehungsweise für die Java-Dateien auf die AST Artefakte.

3.7.2.3 MDF Uniform Resource Identifier (URI)

Für das MDF Dateisystem wurde ein eigener Uniform Resource Identifier (URI) Schema definiert um Dateien und Verzeichnisse im MDF Dateisystem eindeutig identifizieren zu können. Die MDF URI ist dabei folgendermaßen aufgebaut: mdf://<UUID des *SourceCodePackage*s>/<Pfad zum Artefakt>. Jedes MDF Artefakt besitzt eine Universally Unique Identifier (UUID), also eine ID die weltweit eindeutig ist und das Artefakt eindeutig identifiziert (siehe Kapitel 2.6.6). Über diese UUID des *SourceCodePackage*s kann das Artefakt im Modell gefunden werden. Der Pfad zum Artefakt setzt sich aus den einzelnen Namen der Artefakte im Modell entsprechen der Hierarchie zusammen. Für das in Abbildung 43 gezeigte Projekt ist in Abbildung 45 für die Datei CommitDlg.java die Eigenschaften und unter Location die entsprechende URI dargestellt:

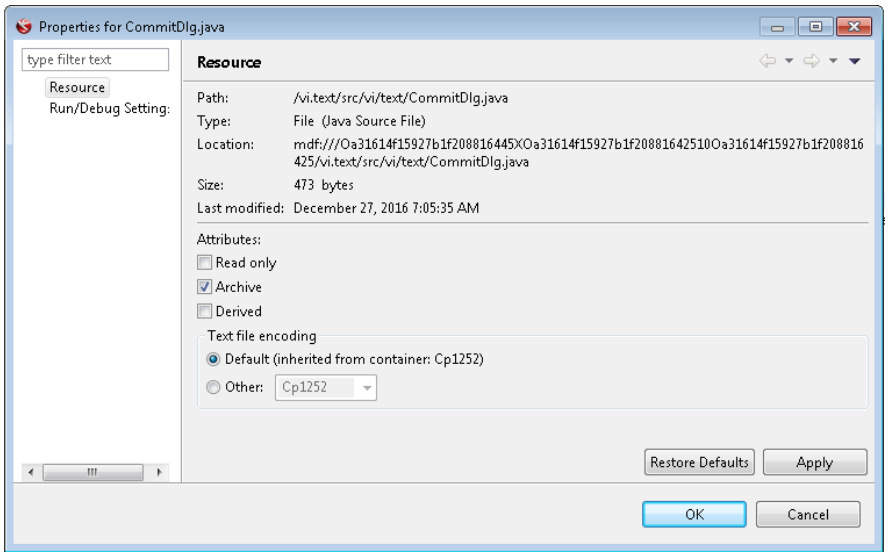


Abbildung 45: Eigenschaften einer Datei im MDF Dateisystem

Die in der URI enthalten UUID gehört dabei zu dem *SourceCodePackage* SCP2.

3.7.2.4 Erweiterung des MDF Dateisystems für zusätzlichen Dateitypen

Bei der Umsetzung des MDF Dateisystems wurde auf eine klare Trennung zwischen allgemeiner Funktionalität, die sich auf das generische Projektkonzept von Eclipse mit Ressource-Verzeichnisse und –Dateien anwenden lässt und der spezialisierten Funktionalität für die Java-Projekte und Java-Dateien. Diese Trennung ist auch im Metamodell vorhanden (siehe Kapitel 3.7.1). Dafür wird die Funktionalität des MDF Dateisystems auf zwei verschiedene Plugins aufgeteilt: ein generisches Plugin und ein Java-spezifisches Plugin. Damit ist es möglich ohne großen Implementierungsaufwand eine Unterstützung für andere Programmiersprachen und Dateitypen zu erreichen. Um weitere Dateitypen zu unterstützen (zum Beispiel die Manifest-Datei oder die Plugin-Datei), müssen folgende Schritte durchgeführt werden:

1. Für den Dateiinhalt muss ein Metamodell definiert werden. Das zum Metamodell passende Modell muss in der Lage sein jeden erlaubten Inhalt der Datei abspeichern zu können. Der in Kapitel 3.4.2 beschriebene Metamodellgenerator kann für XML Dateien mit einer XML Schema Definition (XSD) und für Programmiersprachen mit einer BNF Definition ein passendes Metamodell generieren [78] [76].
2. Es müssen zwei Interfaces implementiert werden: IStructuredFileInfoProvider und IStructuredFileTransformer (siehe Abbildung 46). Die Implementierung des Interfaces IStructuredFileInfoProvider stellt Informationen zu einem Dateityp bereit: die Dateiendung und die Metamodellklasse für das Wurzelartefakt des Teilmodells, das

den Dateiinhalt speichert. MDFObject ist die Basisklasse aller Metamodellklassen des Meta Data Frameworks (MDF). Außerdem stellt das Interface eine Implementierung für das Interface IStructuredFileTransformer zur Verfügung. Über dieses Interface kann aus einem Text ein Modell und aus einem Modell der Text generiert werden.

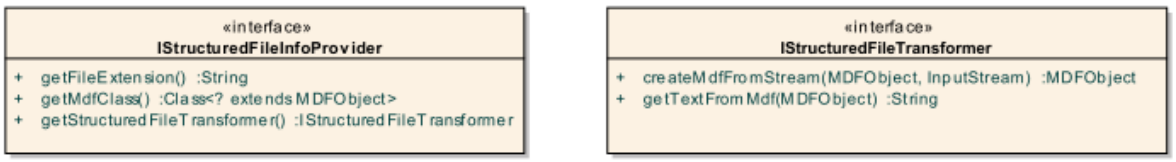


Abbildung 46: Interfaces für die Unterstützung eines neuen Dateityps

Beim Neuanlegen, Laden oder Speichern einer Datei geht das MDF Dateisystem alle registrierten Structured-File-Info-Provider durch. Wenn eine passende Implementierung anhand der Dateiendung oder des Wurzelartefakts des Teilmodells gefunden wurde, wird die dazugehörige Implementierung des IStructuredFileTransformer dazu verwendet den Text oder das Modell zu generieren. Wenn kein Structured-File-Info-Provider gefunden wurde, dann wird eine Instanz der Klasse *EclipseResourceFile* verwendet und der Dateiinhalt in dieses Artefakt abgelegt oder von dort ausgelesen.

3.7.2.5 Einschränkungen

Das ursprüngliche Ziel war ohne irgendwelche Dateien auf der Festplatte im Workspace auskommen zu können und Eclipse sollte alle Information aus dem Modell laden. Dabei ergeben sich jedoch folgende Probleme:

- Beim Starten von Eclipse greift Eclipse sehr früh auf alle Projekte beziehungsweise auf die Projekt-Dateien (.project) im Workspace zu. Zu diesem Zeitpunkt ist jedoch der Anwender noch überhaupt nicht am Modell angemeldet und das Modell ist noch nicht geladen. Um das Modell laden und bearbeiten zu können, muss der Anwender sich authentifizieren. Daher musste hier zusätzliche Logik vorgesehen werden, so dass in diesem Fall ohne Modell auf zwischengespeicherte Dateien zugriefen werden kann.
- Wenn die zu entwickelnde Anwendung aus Debug- und Testgründen gestartet wird, werden die Class-Dateien nicht mehr über das Interface IFileStore geladen, sondern direkt von der Festplatte. Daher müssen die Dateien auf der Festplatte an der richtigen Stelle vorliegen.

- Eclipse greift relative oft über das IFileStore-Interface auf die Dateiinhalte zu. Das kann bei vielen Dateien zu Performanceproblemen führen, wenn jedes Mal aus dem Modell der Text generiert werden muss. Daher muss eventuell der Dateiinhalt in einem Cache gespeichert und vorgehalten werden, um eine wiederholtes Generieren des Texts aus dem Modell zu verhindern.

4 Paralleles Editieren von Quellcode

4.1 Einführung

Damit mehr als ein Softwareentwickler parallel an einer Softwarekomponente arbeiten kann, muss ein Software Configuration Management (SCM) System das parallele Bearbeiten von Source Code unterstützen (siehe Kapitel 1.2.2). Dadurch kann die Arbeit an einem Softwareprodukt parallelisiert und die Entwicklungszeiten können deutlich verkürzt werden. Im Wesentlichen gibt es zwei mögliche Lösungsansätze: pessimistisches und optimistisches Sperren.

4.1.1 Pessimistisches und Optimistisches Sperren

Pessimistisches und Optimistisches Sperren wurde bereits in Kapitel 2.1.3 ausgeführt. Die Konflikte, die beim optimistischen Sperren während der Abgabe gelöst werden müssen, können trivial und einfach sein, aber in anderen Fällen auch sehr kompliziert und fehlerträchtig [81] [82]. Eine Möglichkeit um dieses Problem anzugehen ist es den Softwareentwickler auf mögliche Konflikte mit Änderungen anderer Softwareentwickler während des Bearbeitens des Quellcodes aufmerksam zu machen [83]. Damit weiß der Softwareentwickler bereits vor der Abgabe über den Konflikt Bescheid. Es gibt mehrere Werkzeuge, die diese Art von Funktionalität bereitstellen. Jedoch verhindern diese Werkzeuge das parallele Editieren der gleichen Quellcode-Zeilen nicht und die Konflikte müssen gelöst werden.

Die meisten bekannten SCM Systeme unterstützen pessimistische Sperren. Es wird in der Regel für die Dateien, die nicht verschmolzen werden können (zum Beispiel binäre Dateien) oder in anderen Ausnahmefällen verwendet. Die kleinste Einheit, die in einem SCM Repository abgelegt wird, ist typischerweise eine Datei [84]. Pessimistische Sperren auf Dateiebene sind zu restriktiv und verursachen zu oft Konfliktsituationen, bei der ein anderer Softwareentwickler eine bereits gesperrte Datei bearbeiten will.

4.1.2 Direkte und indirekte Konflikte

Es gibt zwei Arten von Konflikten: direkter und indirekter Konflikt [81] [85]. Ein direkter Konflikt wird durch die Änderung der gleichen Quellcode-Zeile zur gleichen Zeit durch zwei Softwareentwickler in deren persönlichen Arbeitskopie verursacht. Ein SCM System kann einen solchen Konflikt während der Abgabe des Quellcodes erkennen.

Ein indirekter Konflikt entsteht durch die Änderung von Quellcode in einer Datei, welche Auswirkungen haben auf konkurrierende Änderungen eines zweiten Softwareentwicklers in der gleichen oder einer anderen Datei. Dabei handelt es sich jedoch nicht um die gleichen Quellcode-Zeilen wie bei einem direkten Konflikt, sondern um Quellcode der abhängig voneinander ist. Wenn indirekte Konflikte nicht erkannt werden, dann können sie zu Syntax Fehlern im Quellcode, der im SCM Repository gespeichert ist, führen. Auch können neue Softwarefehler entstehen, die nur in der Kombination beider Quellcode Änderungen vorhanden sind. Jede Änderung für sich würde diesen Fehler nicht verursachen. Zum Beispiel, wenn zwei Entwickler die Ausführungszeiten unterschiedlicher Komponenten in einem multi-threading System verändern, dann würde jede Änderung für sich funktioniert, aber in Kombination kommt es zu Synchronisationsproblemen [81]. Continuous Integration (CI) wird dafür eingesetzt, um Syntax Fehler durch einen Build des Softwareprodukts zu finden (siehe Kapitel 2.7). Die neu entstandenen Softwarefehler können nur durch Testen gefunden werden zum Beispiel durch automatische Tests im Rahmen des CI [15]. Ein typisches Beispiel für einen Syntax Fehler durch einen indirekten Konflikt ist folgender (siehe Abbildung 47): Ein Softwareentwickler nennt eine Methode in einer Klasse im Rahmen einer Refaktorierung um (im Beispiel in der Abbildung von „getName“ zu „geLastName“) und ein anderer Softwareentwickler fügt einen neuen Aufruf dieser Methode in einer anderen Klasse mit dem ursprünglichen Methodennamen hinzu (in der Klasse „Validator“ in der Methode „validate“). Der Quellcode ist für jeden Entwickler in der persönlichen Arbeitskopie ohne Syntaxfehler kompilierbar. Wenn nun beide Entwickler ihre Änderungen zur gleichen Zeit in des SCM Repository abgeben, dann kommt es zu keinem direkten Konflikt, weil nicht die gleichen Quellcode-Zeilen geändert wurden. Daher wird der Entwickler, der als zweites seine Änderungen abgibt, zu keiner Aktualisierung gezwungen, was den Syntaxfehler in der persönlichen Arbeitskopie sichtbar machen würde. Aber im Quellcode, der im SCM Repository gespeichert ist, ist ein Syntaxfehler entstanden, weil in einer Klasse eine Methode aufgerufen wird, die mit diesem Namen nicht mehr existiert.

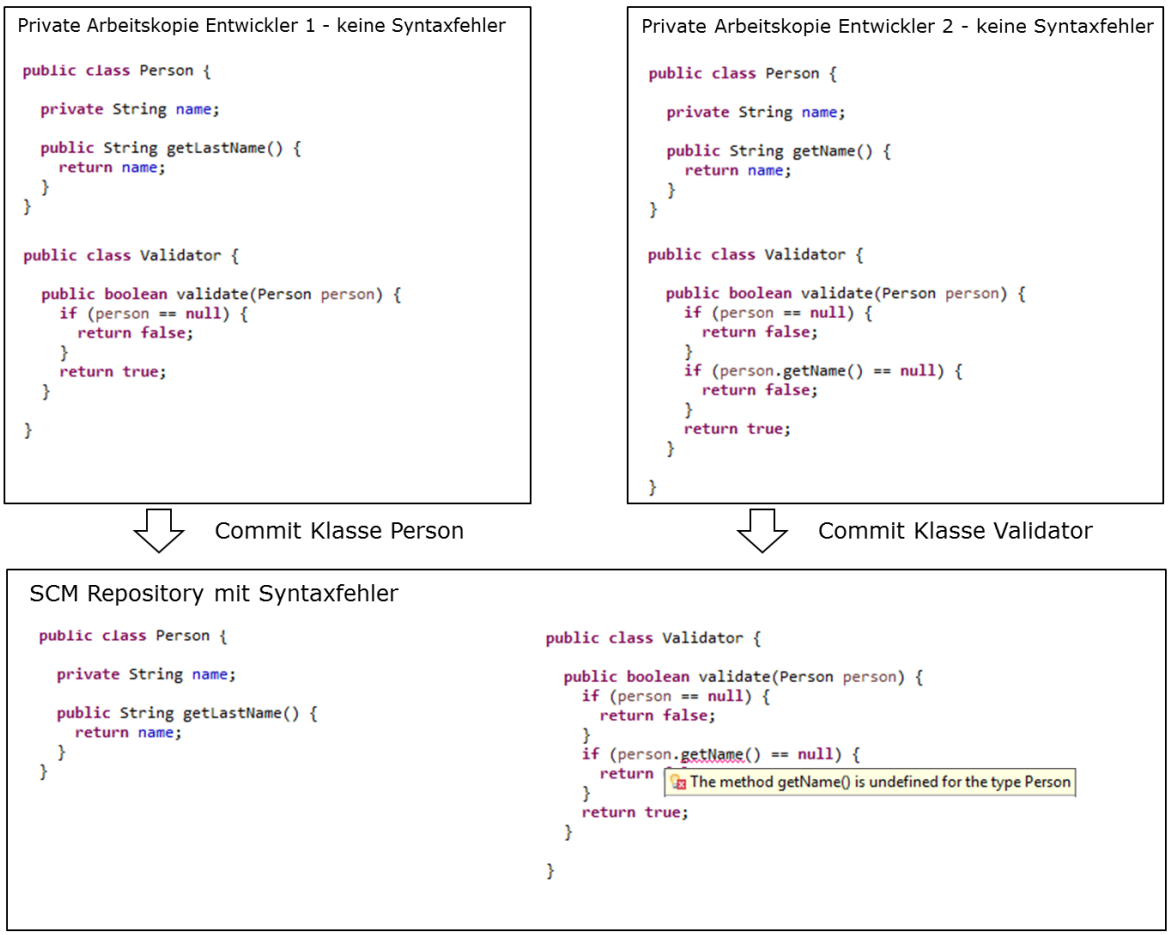


Abbildung 47: Syntaxfehler im SCM Repository durch einen indirekten Konflikt

4.1.3 Motivation und Ziele

Direkte Konflikte können teilweise nur sehr schwierig und aufwändig während des Verschmelzens gelöst werden [11]. Wenn mehrere Zeilen Quellcode betroffen sind zum Beispiel in einem komplizierteren Algorithmus, dann ist es in manchen Fällen nicht möglich das Verschmelzen durchzuführen ohne die Hilfe des anderen Entwicklers, der die zum Konflikt führenden Änderungen durchgeführt hat [14]. Manchmal ist es einfacher alle eigenen Änderungen rückgängig zu machen und die Änderungen im aktualisierten Quellcode nochmals durchzuführen. Diese Konflikte sind sehr störend für den Entwickler, weil es den Entwickler daran hindert seine Änderungen in das SCM Repository abzugeben, obwohl die Arbeit am Quellcode bereits abgeschlossen ist. Unter Umständen muss der Entwickler seine Änderungen nochmal grundsätzlich überarbeiten. Wenn zwei Entwickler am gleichen Quellcode arbeiten, dann muss der Entwickler, der seine Änderungen später abgibt, den Konflikt lösen. Das führt manchmal dazu, dass die Entwickler hastig ihre Arbeit abschließen um die Aufwände für das Verschmelzen zu vermeiden [86]. Außerdem kann die Sorge

Konflikte zu verursachen dazu führen, dass die Entwickler Quellcode nicht überarbeiten, um die Lesbarkeit und Verständlichkeit des Quellcodes zu erhöhen (Refaktorierung). Da eine Refaktorierung oft viele Stellen im Quellcode betrifft (zum Beispiel umbenennen von Klassen und Methoden), oft auch Quellcode, der nicht zur aktuellen Aufgabe des Entwicklers gehört, ist die Wahrscheinlichkeit höher mit einem anderen Entwickler einen Konflikt zu verursachen. Refaktorierung ist jedoch essentiell für die Verständlichkeit des Quellcodes [46].

Das Verschmelzen des Quellcode-Texts kann schon sehr schwierig sein, aber auf AST-Ebene wird das Verschmelzen noch komplexer. Bei der Verwendung eines Abstract Syntax Trees (AST) und Traceability Links (siehe Kapitel 3) wird das jedoch notwendig. Prinzipiell gibt es zwei Möglichkeiten das Verschmelzen durchzuführen:

- Die ASTs werden verschmolzen. Da die Entwickler jedoch gewohnt sind mit Quellcode-Text zu arbeiten und nicht mit ASTs, ist das für den Entwickler nur sehr schwierig zu bewerkstelligen. Dieser Vorgang müsste auf jeden Fall durch ein teilautomatisiertes Verschmelzen unterstützt werden. Konflikte, wo die gleichen AST Artefakte betroffen sind, kann man jedoch nicht automatisch lösen.
- Der Quellcode-Text wird wie bisher auch schon verschmolzen und dann werden daraus neue AST Artefakte generiert. In diesem Fall gehen jedoch alle Traceability Links und die Historie der AST Artefakte verloren. Deswegen muss für jedes AST Artefakt, das aus den verschmolzenen Quellcode-Text entstanden ist, entschieden werden, welches AST Artefakt aus dem aktuellen Modell der Vorgänger ist und welche Traceability Links für welche AST Artefakte erhalten bleiben sollen.

Egal welche Lösung gewählt wird, der Vorgang des Verschmelzens wird dadurch deutlich komplexer.

Indirekte Konflikte, die Syntaxfehler im SCM Repository verursachen, können andere Teammitglieder blockieren und an der Weiterarbeit hindern. Das kann auch Continuous Integration nicht verhindern, weil der Build einige Zeit dauern kann. Wenn die Entwickler ihren Quellcode mit den Änderungen aus dem SCM Repository und den damit eventuell einhergehenden Syntaxfehlern aktualisieren, dann müssen die Entwickler den Syntaxfehler erstmal beheben, bevor sie weiterarbeiten können. Möglicherweise beginnen zwei oder mehr Entwickler parallel den Syntaxfehler zu lösen. Auch der CI Lauf mit den automatischen Tests kann nicht ausgeführt werden. Solange der Syntaxfehler im SCM Repository existiert, werden

alle in dieser Zeit abgegebenen Quellcode-Änderungen nicht über automatische Tests abgesichert. Damit ist keine Aussage über die Qualität der Änderungen in dieser Zeit möglich.

Wenn pessimistische Sperren verwendet werden, dann ist es nicht notwendig den Quellcode zu verschmelzen und deswegen kann der Softwareentwickler seine Änderungen jeder Zeit abgeben. Wenn der Entwickler bereits gesperrten Quellcode ändern will, ist es notwendig mit den Kollegen zu kommunizieren, um die Arbeit aufeinander abzustimmen. Pessimistische Sperren können auch dazu verwendet werden, um indirekte Konflikte zu verhindern, die Syntaxfehler im SCM Repository verursachen. Jedoch sind pessimistische Sperren für die gesamte Datei nicht praktikabel. Pessimistische Sperren sind nur dann nützlich, wenn feingranulare Sperren unterstützt werden und parallele Änderungen einer Klasse oder einer Methode möglich sind. Daher werden in dieser Arbeit folgende Ziele verfolgt und umgesetzt [87] [88]:

- Keine Syntaxfehler im SCM Repository
- Pessimistische Sperren auf Abstract Syntax Tree (AST) Artefakten
- Ermöglichen des parallelen Arbeitens an Klassen und Methoden unter Verwendung einer möglichst kleinen Anzahl von feingranularen Locks
- Minimales, automatisches Sperren während des Editierens von Quellcode im Editor
- Behalten der Isolation gegenüber anderen Entwicklern während der Entwicklung und der Entwickler entscheidet, wann der Quellcode abgegeben werden soll

Beim optimistischen Sperren kann ein direkter Konflikt dadurch gelöst werden, dass die betroffene Quellcode-Datei zuerst aus dem SCM Repository aktualisiert wird und dann die Änderungen verschmolzen werden. Pessimistische Sperren erzwingen die sequentielle Änderung von Quellcode. Daher muss mit einer Sperre eines AST Artefakt das AST Artefakt mit dem Stand aus SCM Repository aktualisiert werden. Der Entwickler wird also gezwungen schon früher eine Aktualisierung des Quellcodes durchzuführen, als beim optimistischen Sperren. Jedoch kann der Entwickler immer selber entscheiden, wann er seine Änderungen für eine Anforderung oder ein Ticket abgeschlossen hat und wann er seine Änderungen allen anderen Entwickler zur Verfügung stellen will.

4.2 Stand der Technik

Es gibt viele Forschungsarbeiten, die sich mit dem Verschmelzen von Quellcode beschäftigen. In einem Ansatz werden zum Beispiel die Sequenzen von Änderungen in verschiedenen Entwicklungszweigen analysiert, um dann die Integration zu unterstützen in dem nicht nur der Quellcode-Text, sondern auch der AST betrachtet wird [89].

Optimistische Sperren werden hier an dieser Stelle jedoch nicht weiter vertieft. Die in diesem Kapitel vorgestellten Konzepte sind dem feingranularen Sperren auf AST Ebene viel näher. Konzepte, die eine Verbesserung im Umgang von Kollaborationskonflikten ermöglichen, können nach Intrusive- und Non-Intrusive-Strategien aufgeteilt werden [83].

4.2.1 Non-Intrusive-Strategien

Bei den Non-Intrusive-Strategien verbessern die Werkzeuge die Zusammenarbeit dadurch, dass Informationen über Quellcode-Änderungen anderer Teammitglieder im Team verteilt werden (im Gegensatz zum Verteilen von Quellcode bei den Intrusive-Strategien). Man spricht auch von „Awareness Enhancers“. Diese Tools reagieren auf potentielle auftretende Konflikte in dem sie die Teammitglieder auf diese Konflikte aufmerksam machen. Das kann zu einer frühen Erkennung von Konflikten und einer verbesserten Kommunikation zwischen den Teammitgliedern führen und damit eine Reduzierung der Aufwände für die Konfliktlösung. Im Folgenden werden drei Werkzeuge vorgestellt.

4.2.1.1 Syde

Syde [90] etabliert Teambewusstsein (Team Awareness) indem Informationen über Änderungen und Konflikte zwischen den lokalen Arbeitskopien ausgetauscht werden unter Verwendung eines änderungszentrierten Ansatzes. Die Hauptherausforderung ist zum einen die notwendigen Informationen dem Entwickler zur Verfügung zu stellen und zum anderen den Entwickler nicht mit irrelevanten Informationen zu überfrachten. Deswegen werden Änderungen nicht auf Dateiebene (Änderungen einzelner Zeilen) verfolgt, sondern auf Abstract Syntax Tree (AST) Ebene, also Änderungen als Baumoperationen im AST Baum. Wenn ein Entwickler innerhalb einer Klasse eine Änderung vornimmt, dann wird eine Baumoperation erfasst und zu einem Server übertragen. Anstatt Algorithmen zu verwenden, die auf Textebene versuchen die Änderungen später zu interpretieren, kann so präzisere Änderungsinformationen bereitgestellt werden und Konflikte durch Vergleich von Baumoperationen ermittelt werden. Syde stellt dabei die Information, wer ändert gerade welchen Teil des Systems in Echtzeit zur Verfügung. Konflikte werden sofort nach der

Entstehung entdeckt und die entsprechenden Entwickler informiert. Die Anwendung ist eine Client-Server-Anwendung. Ein Eclipse Plugin erfasst die Änderungen des Entwicklers im Quellcode-Editor und zeigt die Konfliktinformationen als visuelle Hinweise in der graphischen Benutzeroberfläche an. Alle Änderungen werden als Änderungsoperation in dem zentralisierten Server gespeichert. Diese Änderungen können zu einem späteren Zeitpunkt ausgewertet werden. Vom Server werden alle Änderungsinformationen an die Teammitglieder verteilt. Es ist die Änderungshistorie für jedes Package, Klasse, Methode oder Feld verfügbar. Die Unterschiede zwischen zwei Versionen des Programms können angezeigt werden und der Entwickler wird über Konflikte informiert. Es kann in Echtzeit angezeigt werden, wie sich das Programm entwickelt.

Syde verwendet einen eindeutigen Identifier für jede Programmentität und registriert Umbenennungen und Verschiebung. Für jeden Entwickler wird sein aktueller AST auf dem Server vorgehalten. Die Entwicklung des Programms wird über die Änderungsoperationen nachvollzogen, das heißt der Quellcode wird durch die sequentielle Ausführung der Änderungsoperationen von einer Version in die nächste Version überführt. Die Änderungen werden mit jedem Speichern des Quellcodes aufgezeichnet. Es gibt zwei Typen von Änderungsoperationen: Atomare Änderung und Refaktorierungen (siehe Abbildung 48).

Atomic Operations	
Insertion	Insert a node *n* as a child of parent *p*.
Deletion	Delete a node *n* from its parent *p*.
Property Change	Change the value *v* of property *r* of node *n*.
Property Insertion	Insert the value *v* of property *r* of node *n*.
Property Deletion	Delete the value *v* of property *r* of node *n*.

Refactorings	
Rename	Change the name of a node *n* and change all references of this node from the old name to the new one.
Move	Move a node *n* and all its decedents from old parent p_{old} to new parent p_{new}.

Abbildung 48: Änderungsoperation in Syde [90]

Syde erkennt strukturelle Konflikte in dem jede atomare Operation auf den AST des Entwicklers am Server angewendet wird und mit allen ASTs der anderen Entwickler verglichen wird. Der Konflikt wird in zwei Kategorien klassifiziert: Gelb bedeutet, dass es strukturelle Unterschiede zwischen zwei Versionen eines Knoten gibt, aber keine der

Änderungen bisher in das SCM Repository abgegeben wurden. Rot bedeutet, dass es einen Konflikt gibt und mindestens eine Änderung bereits in das SCM abgegeben wurde.

4.2.1.2 CollabVS

CollabVS [86] ist eine Erweiterung der Visual Studio Entwicklungsumgebung mit folgenden Zielen:

- Konflikte sollen sehr früh erkannt werden, also bereits mit dem Editieren einer Programmentität durch einen Entwickler.
- Das System soll Information über Abhängigkeiten unter Programmelementen nutzen um direkte und indirekte Konflikte zu erkennen.
- Die Entwickler sollen dabei unterstützt werden, gemeinsam potenzielle Konflikte zu beurteilen und unter Umständen gemeinsam sofort zu lösen.
- Die Bedienung des Systems soll einfach zu erlernen sein.

Für die Aufdeckung und der Umgang mit Konflikten wurde ein semi-synchrones Modell entwickelt, bei der das Editieren des Quellcodes immer asynchron erfolgt und die Konfliktlösung entweder synchron (das heißt mit zwei Entwicklern gleichzeitig) oder asynchron erfolgt (siehe Abbildung 49).

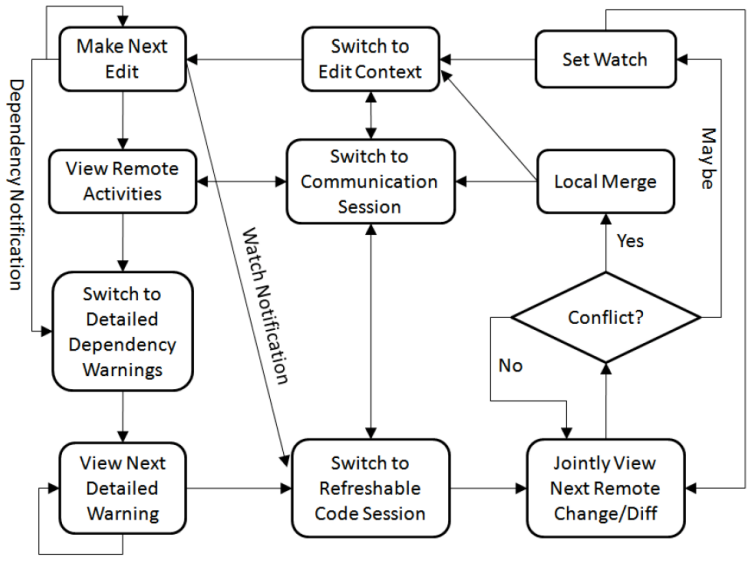


Abbildung 49: Semi-Synchrones Konfliktlösungsmodell von CollabVS [86]

Warnungen über potentielle Konflikte sollen den Entwickler nur einmal angezeigt werden und nicht wiederholt. Das erfolgt durch ein Popup-Fenster, dass nach einiger Zeit von alleine

wieder verschwindet. Der Entwickler kann in das Konfliktposteingangsfach wechseln und alle aktuelle Konflikte sichten (siehe Abbildung 50).

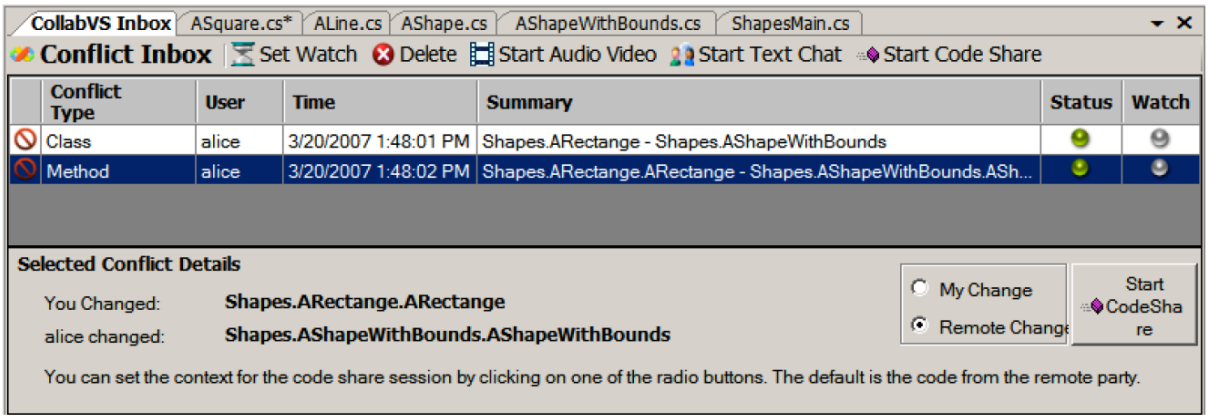


Abbildung 50: Konfliktposteingangsfach von CollabVS [86]

Danach kann der Entwickler mit dem Bearbeiten des Quellcodes fortfahren oder der Entwickler wechselt in eine Code-Session, die mit anderen Entwicklern geteilt werden kann. Hier kann der Quellcode des anderen Entwicklers eingesehen werden, um beurteilen zu können, ob tatsächlich ein Konflikt vorliegt. Es ist möglich eine Kommunikation-Session mit dem anderen Entwickler zu öffnen unter Verwendung von Textnachrichten, Audio- oder Videokommunikation. Wenn der Quellcode noch nicht vollständig ist und noch keine Konfliktlösung möglich ist, kann der Konflikt unter Beobachtung gestellt werden und der Entwickler wird über weitere Änderungen des anderen Entwicklers informiert, um nach Abschluss der Arbeiten den Konflikt nochmals neu zu bewerten und eventuell zu lösen. Wenn ein echter Konflikt festgestellt wurde, dann können die Änderungen des anderen Entwicklers direkt in den eigenen Quellcode eingearbeitet werden und damit der Konflikt bereits vor dem ersten Abgabe-Versuch gelöst werden.

Potenzielle Konflikte werden erkannt wenn ein Entwickler eine Programmentität ändert, die Abhängigkeiten zu anderen Programmelementen hat, die bereits von einem anderen Entwickler verändert und noch nicht abgegeben wurden. Es werden nach Abhängigkeiten unter den folgenden Elementen gesucht: Datei, Typ (Klasse oder Interface) und Methode. Jedes dieser Elemente ist von sich selbst abhängig. Ein Typ ist auch von Subtypen und Supertypen und eine Methode von dem Methodenaufrufer abhängig. Solche Abhängigkeiten können sich rekursive fortsetzen.

4.2.1.3 Palantír

Palantír [26] gibt den Entwickler Einblick in die privaten Arbeitsbereiche anderer Entwickler und baut auf existierenden Konfigurationsmanagement Anwendungen auf. Für das Erkennen von direkten Konflikten sind vier Schritte notwendig: Sammlung, Verteilung, Filterung und Visualisieren relevanter Informationen aus den einzelnen Arbeitsbereichen der Entwickler. Palantír wurde später erweitert um auch indirekte Konflikte erkennen zu können [81]. Dazu wurden die vier Schritte um zwei weitere ergänzt: Arbeitsbereich übergreifende Analyse der aktuellen Änderungen und Verteilung der Information über indirekte Konflikte an andere Arbeitsbereiche (siehe Abbildung 51).

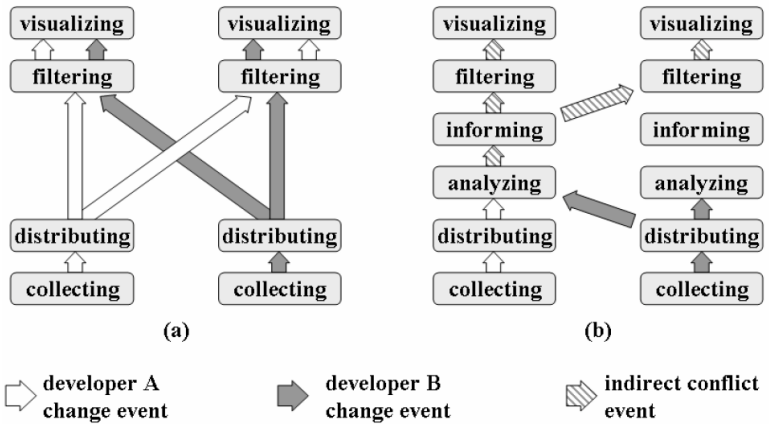


Abbildung 51: Gesamtprozess für direkte und indirekte Konflikte mit Palantír [81]

Die sechs Schritte beinhalten dabei folgende Aktivitäten für die indirekte Konflikterkennung:

- Collection: Es werden alle Änderungen an Zugriffs- oder Verwendungsmöglichkeiten einer Klasse erfasst zum Beispiel Umbenennung oder Änderung der Methodensignatur einer public Methode oder löschen von Interfaces und Klassen. Dabei wird das Editieren ständig beobachtet und registriert und damit ist ein immer aktuelles Bild von den aktuellen Änderungen vorhanden.
- Distribution: Diese erfassten Änderungen an einer Klasse werden nicht als zeilenbasierte textuelle Differenz übertragen, sondern in einem eigenen XML Format, in dem alle möglichen Änderungen an der Klasse eingefügt werden können. Das wird durch einen Push-basierten Eventservice realisiert.
- Analyzing: In diesem Schritt werden die Änderungen des lokalen Arbeitsbereichs mit den Änderungen der anderen Arbeitsbereiche zusammengebracht um vorhandene indirekte Konflikte entdecken zu können. Da nur inkrementelle Änderungen zur Verfügung stehen, muss aus Performancegründen ein Cache vorgehalten werden, der

den aktuellen Status des Arbeitsbereichs beinhaltet. Mit jeder Änderung wird der Cache aktualisiert. Eine Änderung eines Arbeitsbereichs wird mit dem Cache der anderen Arbeitsbereiche verglichen und die Abhängigkeiten in vorwärts und rückwärts Richtung untersucht.

- Informing: Nach dem die Analyse abgeschlossen ist und die indirekten Konflikte gefunden wurden, werden Informationen bezüglich dieser Konflikte an die Arbeitsbereiche versendet, in dem ein oder mehr involvierte Artefakte vorhanden sind. Die Information beinhaltet dabei sowohl das Artefakt, das den indirekten Konflikt verursacht hat als auch das Artefakt, das durch den Konflikt betroffen ist. Da durch die Rücknahme von Änderungen Konflikte wieder verschwinden können, müssen dafür entsprechende Information verschickt werden.
- Filtering: Der Entwickler hat einige Möglichkeiten Filterkriterien zu definieren, nach denen die Informationen gefiltert werden. Für indirekte Konflikte kann den Entwickler zum Beispiel eine minimale Anzahl von betroffenen Artefakten festlegen, die erreicht werden muss, bevor der Entwickler informiert wird.
- Visualizing: Die Konfliktinformationen werden dann dem Entwickler angezeigt. Die Anzeige erfolgt dabei in der Art und Weise, dass sie nicht zu sehr in den Vordergrund tritt, aber doch vom Entwickler bemerkt werden kann.

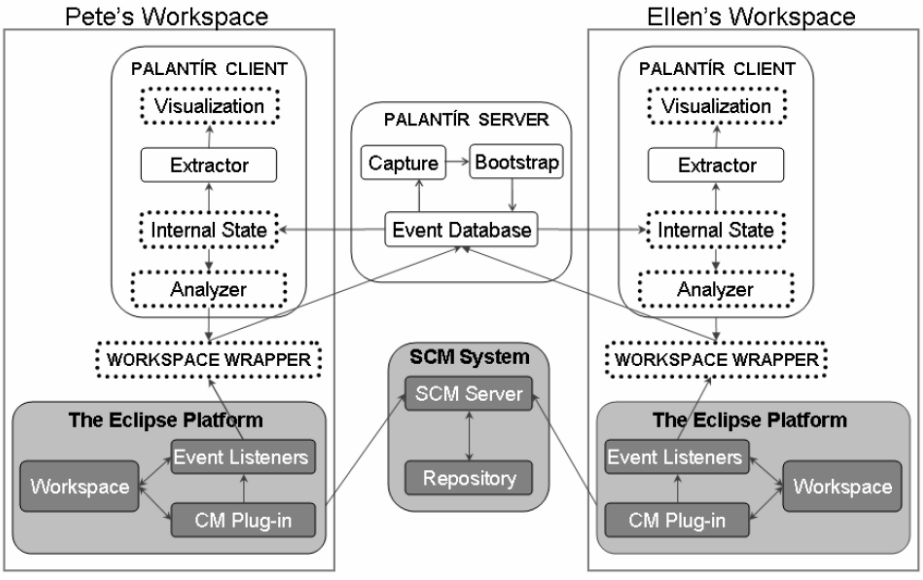


Abbildung 52: Die Architektur von Palantír [81]

In Abbildung 52 ist die Architektur von Palantír zu sehen. Die Eclipse Plattform und das allgemeine SCM System werden in den grauen Kästen dargestellt. Palantír verwendet diese

unverändert und erhält die notwendigen Informationen über den Workspace Wrapper, welcher die Events über die aktuellen Änderungen aufsammelt. Der Palantír Server ist verantwortlich die Events an die Arbeitsbereiche weiterzuleiten, die von dem Event betroffen sind. Der Server stellt historische Daten über alle Arbeitsbereiche bereit für die Arbeitsbereiche, die neu angelegt wurden und dann diese Daten benötigen (Bootstrap). Der Palantír Client besteht aus mehreren Unterkomponenten. Die Analysekomponente erkennt die indirekten Konflikte und die Visualisierungskomponente stellt die Konflikte dann in der graphischen Benutzeroberfläche dar.

4.2.1.4 Abgrenzung

Die in diesem Kapitel vorgestellten Werkzeuge zeigen den Entwicklern mögliche Konflikte in der graphischen Benutzeroberfläche an. Der Entwickler kann diese Information nutzen, er kann sie aber auch ignorieren. Grundsätzliche verhindern diese Ansätze die Konflikte nicht. Mit dem feingranularen pessimistischen Sperren werden proaktiv Konflikte von vornherein unterbunden und der Entwickler muss nicht auf Konflikte reagieren. Die dahinterliegende Annahme ist, dass der Entwickler lieber auf die Änderung von Quellcode wartet als mit dem Aufwand des Verschmelzens von Quellcode konfrontiert zu werden [27]. Das trifft im Besondern für das komplexe Verschmelzen von ASTs mit Historie und Traceability Links zu (siehe Kapitel 4.1.3). Wenn ein Konflikt zugelassen wird, dann muss er auch irgendwann gelöst werden mit den ganzen damit verbundenen Aufwänden und Schwierigkeiten. Die von diesen Werkzeugen für die Softwareentwickler bereitgestellten Informationen, welche Änderungen gerade in Arbeitsbereichen anderer Entwickler vorgenommen werden, stehen über die pessimistischen Sperren auch zur Verfügung. Der Softwareentwickler kann an Hand der pessimistischen Sperren erkennen, welcher Quellcode von welcher Person seit wann in Bearbeitung ist.

Oft sind die Konflikte bei diesen Werkzeugen nur potentielle Konflikte und müssen von dem Entwickler erst bewertet werden. Die Gefahr besteht, dass der Entwickler mit zu vielen Informationen über Änderungen anderer Entwickler belastet wird. Bei feingranularen pessimistischen Sperren wird der Entwickler mit dem Problem erst dann konfrontiert, wenn es tatsächlich relevant ist.

4.2.2 Intrusive-Strategien

Werkzeuge die zu der Kategorie der Intrusive-Strategie gehören, propagieren automatisch geänderten Quellcode von einem Teammitglied zum anderen. Die Idee ist proaktiv den

Quellcode der Teammitglieder zu synchronisieren und damit einen verschmolzenen und vereinheitlichten Quellcode-Stand zu haben. Dadurch, dass alle Teammitglieder immer an dem letzten und aktuellen Stand arbeiten, werden Konflikte von vornherein verhindert. Es gibt eine kleine Anzahl von Projekten, die diese Strategie umgesetzt haben: CloudStudio [91], Collabode [92] und Code Synchronizing Intelligence (CSI) [27]. Diese Projekte implementieren den Synchronized Software Development (SSD) Ansatz. Die CSI Anwendung arbeitet auch mit pessimistischen Sperren und wird daher im Detail vorgestellt.

4.2.2.1 Code Synchronizing Intelligence (CSI)

CSI [27] ist eine Prototypentwicklung um einen neuen Ansatz für ein Synchronized Software Development System zu demonstrieren. Dabei stehen zwei Hauptfeatures im Vordergrund:

1. Automatische Verteilung von geänderten Quellcode an alle anderen Entwicklungsumgebungen
2. Automatisches Setzen von feingranularen pessimistischer Sperren während des Editierens um Konflikte zu verhindern

Die Architektur ist eine zentralisierte Architektur und Quellcode und Informationen über Änderungen werden über einen Server zwischen allen Entwicklungsumgebungen verteilt.

CSI wurde für Eclipse und Java entwickelt unter Verwendung des Java Models von Eclipse. Mittels dieses Modells werden Änderungen am Quellcode auf semantischer Ebene (nicht auf textueller Ebene) verfolgt zum Beispiel das Anlegen einer neuen Methode. Diese Änderungen werden dann an andere Eclipse Entwicklungsumgebungen propagiert. Folgende Szenarien werden dabei unterstützt:

- Erstellung einer neuen Methode
- Ändern eines existierenden Methodennamens
- Ändern eines existierenden Methodenparameter
- Ändern eines existierenden Rückgabewert
- Ändern eines Methodeninhalts
- Löschen einer Methode
- Erstellung einer neuen Klassenvariablen
- Ändern einer existierenden Klassenvariablen
- Löschen einer Klassenvariablen

Durch die Verteilung der Information über gerade geänderten Quellcode an alle Entwicklungsumgebungen, können diese dann für andere Entwickler gesperrt werden. In dem Beispiel in der Abbildung 53 versucht Alice den Methodennamen zu verändern, der von Bob bereits verändert wurde. Alice erhält daher eine Fehlermeldung. Damit werden direkte Konflikte verhindert und es ist kein Verschmelzen notwendig.

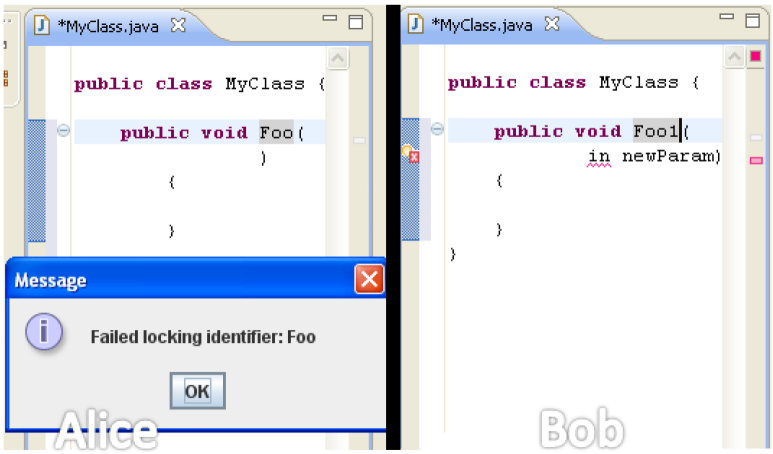


Abbildung 53: Fehlermeldung wegen eines bereits gesperrten Methodennamens in CSI [27]

Um auch indirekte Konflikte verhindern zu können müssen Elementabhängigkeiten mitberücksichtigt werden. Ein Codeelement wird als ein einzelnes syntaktisches Konstrukt definiert (zum Beispiel eine Anweisung) und eine Entität ist ein Identifier für eine spezifische Deklaration (zum Beispiel eine Variable oder Methode). Das Codeelement kann von einer oder mehreren Entitäten abhängig sein. Die Abhängigkeit besteht nur dann, wenn eine Änderung der Entität dazu führt, dass der Quellcode nicht mehr kompilierbar ist wegen Syntaxfehler im Codeelement. Um indirekte Konflikte zu verhindern dürfen Codeelement und Entität nicht gleichzeitig verändert werden. Das wird über pessimistische Sperren verhindert.

Der Fehlererkennungsmechanismus von Eclipse wird dazu genutzt, um das Auftreten und Verschwinden von Syntaxfehlern genau zu registrieren. Sobald es nach der Änderung von Quellcode keine Syntaxfehler mehr gibt, wird der Quellcode sofort an alle anderen Entwicklungsumgebungen verteilt. Damit arbeiten die Entwickler an einem gemeinsamen Quellcode-Stand, der sich nur für kurze Zeitspannen voneinander unterscheidet. Die Verteilung von nur syntaxfreiem Quellcode verhindert, dass die Entwickler mit nicht kompilierbarem Quellcode anderer Entwickler belastet werden. In manchen Fällen macht es Sinn diese automatische Verteilung ausschalten zu können, um zum Beispiel den Quellcode

mit Unit-Tests abzusichern. In dieser Zeit bleiben jedoch die pessimistischen Sperren erhalten und die Quellcode-Stände laufen immer mehr auseinander.

4.2.2.2 Abgrenzung

Die Intrusive-Strategie von CSI ist für den Ansatz in dieser Arbeit ungeeignet. Die Abgabe von Quellcode in CSI ist kein expliziter Vorgang, sondern erfolgt implizit sobald der Quellcode keine Fehler mehr beinhaltet. Da jedoch der geänderte Quellcode mit dem Änderungsgrund (Anforderung oder Ticket) verlinkt und mit einem Abgabe-Kommentar versehen werden soll, müsste der Entwickler diese Informationen ständig bereitstellen. Da jede Änderung sofort in einem Repository gespeichert werden müsste, wird auch die Historie der Artefakte viel größer. Die Historie enthält Quellcode-Umbauten, die sich als nicht machbar und fehlerhaft herausgestellt haben und bei einer expliziten Abgabe niemals im Repository gelandet wären. Das kann andere Entwickler beim Review der Historie sehr verwirren. Außerdem soll der Entwickler die Möglichkeit haben eine Anforderung oder ein Ticket zumindest in Teilen abschließen zu können, bevor er die Änderungen abgibt und diese für andere sichtbar werden.

Die in CSI vorgestellte Umsetzung ist nicht vollständig. Interfaces und überschriebene Methoden werden zum Beispiel nicht behandelt. Es gibt noch einige Sonderfälle, die nicht ausreichend berücksichtig wurden, zum Beispiel, wenn ein Entwickler einen Methodenaufruf im Quellcode hinzufügt zu einer Methode die gerade von einem anderen Entwickler verändert wird. Die Änderung eines bestehenden Methodenaufrufs würde über die pessimistischen Sperren verhindert werden, aber nicht das Hinzufügen eines neuen Methodenaufrufs. Ohne weitere Funktionalität für diesen Anwendungsfall könnte ein Entwickler eine Reihe von Methodenaufrufe in seinen Quellcode hinzufügen, die nach der automatischen Aktualisierung plötzlich nicht mehr vorhanden sind, weil sie über eine Refaktorierung in andere Klassen verschoben oder unbenannt wurden. Dann muss der Entwickler seinen Quellcode anpassen und der Konflikt wurde nicht verhindert. Ein weiteres Beispiel ist der Default-Konstruktor einer Java-Klasse. So lang kein anderer Konstruktor existiert wird ein Default-Konstruktor von Java bereitgestellt. Wenn nun ein Entwickler einen ersten Konstruktor mit Parametern hinzufügt, werden damit alle Default-Konstruktor-Aufrufe ungültig. Auch dieser Fall muss über pessimistische Sperren gesondert behandelt werden.

Die in CSI vorgestellten pessimistischen Locks könnten noch feingranularer sein und noch mehr paralleles Bearbeiten von Quellcode ermöglichen. Zum Beispiel wird der komplette

Methodeninhalt für das parallel bearbeiten gesperrt, obwohl auch innerhalb einer Methode parallele Änderungen möglich wären.

Alle diese und weitere unberücksichtigte Aspekte werden in dieser Arbeit adressiert und umgesetzt.

4.2.3 Alleinstellungsmerkmale

Damit lassen sich folgende Produktmerkmale bezüglich paralleles Editieren von Quellcode für die Darstellung der Alleinstellungsmerkmale definieren:

A) Awareness Enhancers

Dem Softwareentwickler werden Informationen über bereits von anderen Softwareentwicklern veränderte Quellcodetexte bereitgestellt.

B) SCM ohne Syntaxfehler

Es werden indirekte Konflikte beim parallelen Bearbeiten des Quellcodes verhindert, so dass es zu keinen Syntaxfehlern im SCM Repository kommen kann.

C) Feingranulare pessimistische Sperren

Es wird das pessimistische Sperren von AST Artefakten unterstützt, so dass es zu garantiert keinen Konflikten kommen kann und eine Abgabe des Quellcodes jederzeit möglich ist.

	Morpheus	Syde	CollabVS	Palantír	CSI
A) Awareness Enhancers	✓	✓	✓	✓	
B) SCM ohne Syntaxfehler	✓				
C) Feingranulare pessimistische Sperren	✓				✓[3]

Tabelle 2: Alleinstellungsmerkale bezüglich paralleles Editieren

4.3 Abstract Syntax Tree (AST)

Eine pessimistische Sperre auf einer Datei blockierte eine vollständige Klasse, obwohl parallele Änderungen in der Klasse in vielen Fällen kein Problem sind. Sperren auf Zeilen in der Quellcode-Datei sind nicht sinnvoll, weil Zeilen nur die Lesbarkeit des Quellcodes verbessern und keine syntaktische Bedeutung haben. Sie helfen daher nicht den Quellcode

[3] Teilweise: Es werden nicht alle Anwendungsfälle berücksichtigt.

syntaktisch korrekt zu halten. Der Abstract Syntax Tree (AST) ist feingranular genug um feingranulare pessimistische Sperren zu ermöglichen. Der AST ermöglicht die Auflösung von Abhängigkeiten im Quellcode. Die Abhängigkeiten sind relevant um indirekte Konflikte durch Sperren verhindern zu können.

4.3.1 Relationen und Abhängigkeiten im Abstract Syntax Tree

Der Java-AST von Eclipse kennt nur Kompositionsrelationen (siehe 2.5.2.2), das heißt die einzelnen AST Artefakte bilden einen Baum und es gibt keine anderen Relationen zwischen den AST Artefakten. Tatsächlich gibt es natürlich weitere Abhängigkeiten zwischen den AST Artefakten. Im Folgenden wird eine Definition dieser Abhängigkeiten angelehnt an Lewis Notation [27] eingeführt. Diese Abhängigkeitsbetrachtung trifft auf alle objektorientierte Sprachen zu, wird aber hier am Beispiel von Java gezeigt. Eine Abhängigkeit besteht zwischen einem Supplier-Artefakt und einem Client-Artefakt, wobei das Client-Artefakt vom Supplier-Artefakt abhängig ist. Client- und Supplier-Artefakte sind AST Artefakte. Das Supplier-Artefakt ist zum Beispiel eine Deklaration und das Client-Artefakt ist die Verwendung der Deklaration. Es besteht also eine Art von Verwendungsbeziehung zwischen den beiden Artefakten.

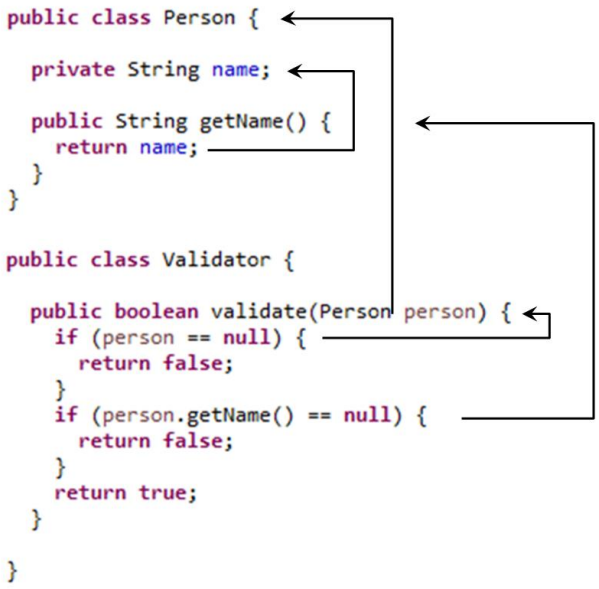


Abbildung 54: Quellcode-Beispiel mit mehreren Abhängigkeiten

In Abbildung 54 sind mehrere Client- und Supplier-Artefakte zu sehen. Zum Beispiel ist der Methodenaufruf „person.getName()“ ein Client-Artefakt vom Supplier-Artefakt, der Methodendeklaration „public String getName()“. Wenn die Methodendeklaration, also das

Supplier-Artefakt verändert wird (zum Beispiel Methodenname, Rückgabewerten oder Methodenparameter), dann muss der Client-Artefakt entsprechend dem Supplier-Artefakt angepasst werden, damit keine Syntaxfehler entstehen. Diese Anpassung ist nicht in jedem Fall notwendig. Zum Beispiel bei der Änderung des Rückgabewerts der Methode „getName“ von „String“ zu „Object“ muss in diesem Fall der Methodenaufruf nicht angepasst werden. Ein anderes Beispiel ist die Implementierung eines Interfaces in einer Klasse. Die Klasse (Client) ist vom Interface (Supplier) abhängig und bei Änderungen im Interface, zum Beispiel Änderungen des Interfacenamens oder der Interfacemethoden, muss die Klasse entsprechend angepasst werden. Die Methode in der Klasse, die die Interfacemethode implementiert, ist dabei ein Client-Artefakt. Das dazugehörige Supplier-Artefakt ist die Methode im Interface, die implementiert wird.

Diese Verwendungsbeziehung hat natürliche eine Richtung und man kann Client- und Supplier-Artefakte daher nicht einfach vertauschen. Im Beispiel vom Interface definiert das Interface alle Methodensignaturen und alle Implementierungen des Interfaces müssen diese Methodensignaturen implementieren. Eine Änderung im Supplier-Artefakt kann zu Syntaxfehler im Client-Artefakt führen, aber niemals umgekehrt. Eine Änderung im Client-Artefakt wird niemals zu Syntaxfehlern im Supplier-Artefakt führen.

Die Abhängigkeit zwischen Supplier- und Client-Artefakten kann damit folgendermaßen definiert werden [27]:

$$Abhängigkeit \subseteq \{Client - Artefakte\} \times \{Supplier - Artefakte\}$$

$$\text{für } Client \in \{Client - Artfakte\},\ Supplier \in \{Supplier - Artefakte\}:$$

$$(Client, Supplier) \in Abhängigkeit \Leftrightarrow$$

$$Änderung\ im\ Supplier\ verursacht\ Syntaxfehler\ in\ Client$$

Folgende Gründe sprechen für eine Abbildung dieser Abhängigkeiten im Metamodell über Assoziationen:

- Nachdem im Quellcode-Text kein direkter Link zwischen den abhängigen Quellcode-Stellen existiert, muss in der Regel über die Namen, die abhängige Quellcode-Stelle in anderen Projekten und Quellcode-Dateien gesucht werden. Wie bereits erwähnt, baut Eclipse in einem ersten Schritt seinen AST auf ohne diese Abhängigkeiten aufzulösen.

Das geht relativ schnell, weil nur der Quellcode-Text geparst werden muss. In einem zweiten Schritt (durch den Aufruf von zusätzlicher Funktionalität auf dem AST) kann dann diese Abhängigkeit aufgelöst und das abhängige AST Artefakt ermittelt werden. Dazu muss die entsprechende Quellcode-Datei gesucht und geparst werden, wenn sich das abhängige AST Artefakt nicht in derselben Datei befindet. Diese explizite Auflösung der Abhängigkeiten ist in einem Modell mit den entsprechenden Assoziationen nicht mehr notwendig. Daher können das Modell und damit der Quellcode sehr schnell durchlaufen und analysiert werden bezüglich der Abhängigkeiten. Verschiedene Auswertungen und Suchen sind im Modell schnell durchführbar zum Beispiel die Ermittlung der Aufrufhierarchie einer Methode.

- Für die Erkennung von indirekten Konflikten ist es wichtig die Abhängigkeit im Modell zu ermitteln ohne einen AST aufbauen und die Abhängigkeiten auf lösen zu müssen. Mit den Abhängigkeiten können die richtigen Artefakte gesperrt werden. Um den Softwareentwickler beim Editieren des Quellcodes nicht unnötig lang zu unterbrechen, wird hier eine schnelle Berechnung der Abhängigkeiten benötigt (siehe auch Kapitel 4.4.3).
- In den meisten Fällen kommt der Name des Supplier-Artefakts im Quellcode-Text auch im Client-Artefakt vor. Zum Beispiel steht der Name der Methode in der Methodendeklaration und im Methodenaufruf. Dadurch wird das Umbenennen einer Methode aufwendig, weil alle Methodenaufrufe in den entsprechenden Quellcode-Dateien korrigiert werden müssen. Die meisten Entwicklungsumgebungen stellen entsprechende Funktionalität für die Refaktorierung zur Verfügung, die diese Anpassungen im Text unterstützt. Im Modell wird der Name nur einmal im Supplier-Artefakt gespeichert. Das Client-Artefakt besitzt eine explizite Assoziation zum Supplier-Artefakt mit dem Namen. Damit muss bei einer Umbenennung nur noch der Name im Supplier-Artefakt verändert werden und es sind keine weiteren Anpassungen mehr notwendig. Dieser Vorteil relativiert sich jedoch, weil in den kompilierten Class-Dateien immer noch, wie ursprünglich im Quellcode-Text, der Name auch im Client-Artefakt steht. Daher müssen zwar nicht die verschiedenen Quellcode-Stellen angepasst werden, aber sie müssen neu kompiliert werden, damit der neue Name in die Class-Datei eingetragen wird.

Vor allem für die Erkennung des indirekten Konflikts werden die Assoziationen im Metamodell benötigt. Daher sind folgende Anpassungen und Erweiterungen notwendig.

4.3.2 Metamodell Anpassungen

Die zuvor erwähnte Definition zur Abhängigkeit hilft an dieser Stelle nicht weiter, um die entsprechende AST Metaklassen für die benötigten Assoziationen zu bestimmen. Die einfachste und häufigste Abhängigkeit ist jedoch im AST von Eclipse sehr leicht zu erkennen, weil das Supplier-Artefakt über einen Namen referenziert wird. Die AST Klassen in Eclipse heißen „SimpleName“ und „QualifiedName“. Im AST Metamodell werden an dieser Stelle Reference-Metaklassen eingeführt.

Im Folgenden werden beispielhaft einige Ausschnitte des Metamodells bezüglich solcher Referenzrelationen vorgestellt. In Abbildung 55 ist die Erzeugung einer neuen Instanz einer Klasse zu sehen. Es sind der Quellcode und ein Teil des für den Quellcode benötigten Metamodells abgebildet. In der „new“-Anweisung wird eine Referenz auf die zu instanziierende Klasse benötigt. Die Klasse „Person“ ist eine *TypeDeclaration* und wird über die Klasse *TypeReference* referenziert. *ClassInstanceCreation* ist die Client-Metaklasse und *TypeDeclaration* ist die Supplier-Metaklasse.

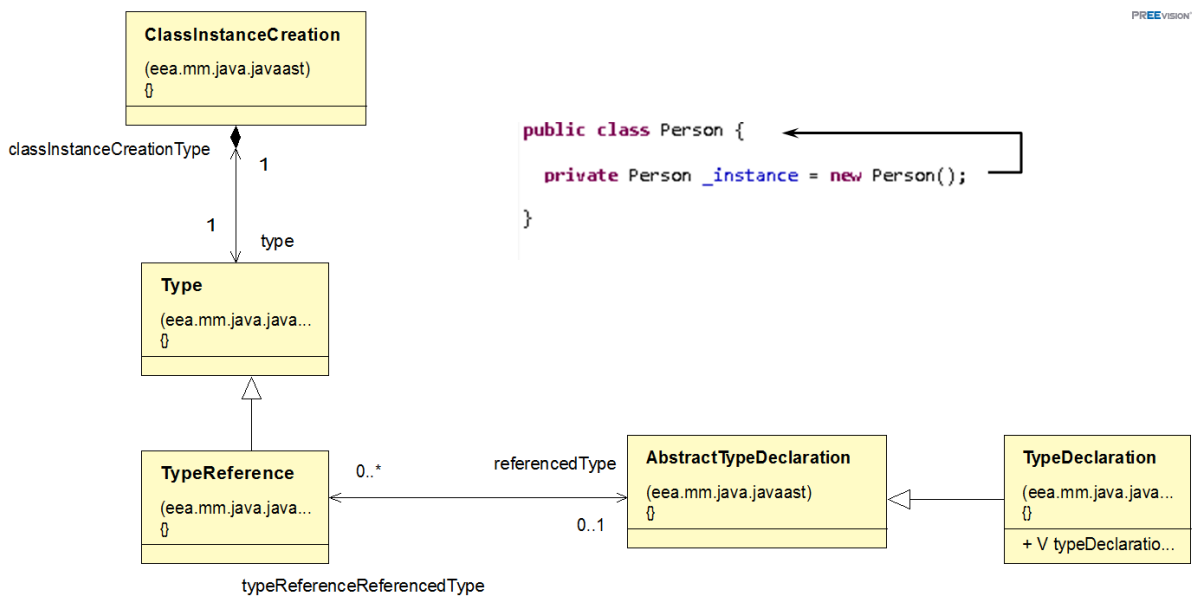


Abbildung 55: Erzeugung einer Klasseninstanz

Abbildung 56 zeigt den Aufruf einer Methode. Die Methode wird in der Metaklasse *MethodDeclaration* definiert und über die Metaklasse *MethodInvocation* aufgerufen. *MethodInvocation* benötigt dazu eine Referenz auf die aufzurufende Methode. *MethodInvocation* ist die Client-Metaklasse und *MethodDeclaration* ist die Supplier-Metaklasse.

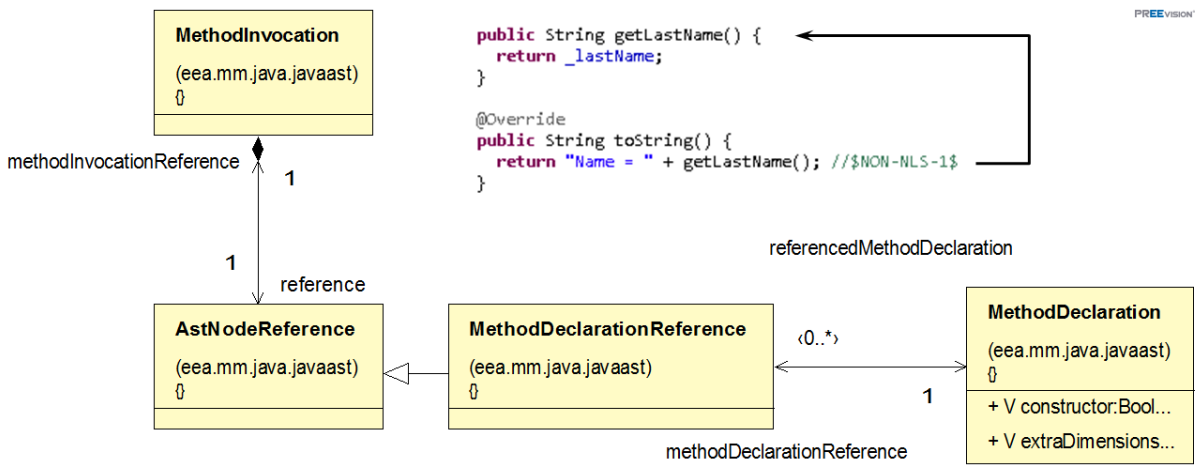


Abbildung 56: Aufruf einer Methode

In Abbildung 57 ist die Zuweisung eines Wertes zu einer Variablen zu sehen. Das *Assignment* bzw. die *Expression* braucht dazu die Referenz auf die *VariableDeclaration*, die über die Metaklasse *VariableDeclarationReference* bereitgestellt wird. Die *VariableDeclaration* befindet sich innerhalb eines *VariableDeclarationStatement*s. Die *Expression* ist dabei die Client-Metaklasse und die *VariableDeclaration* die Supplier-Metaklasse.

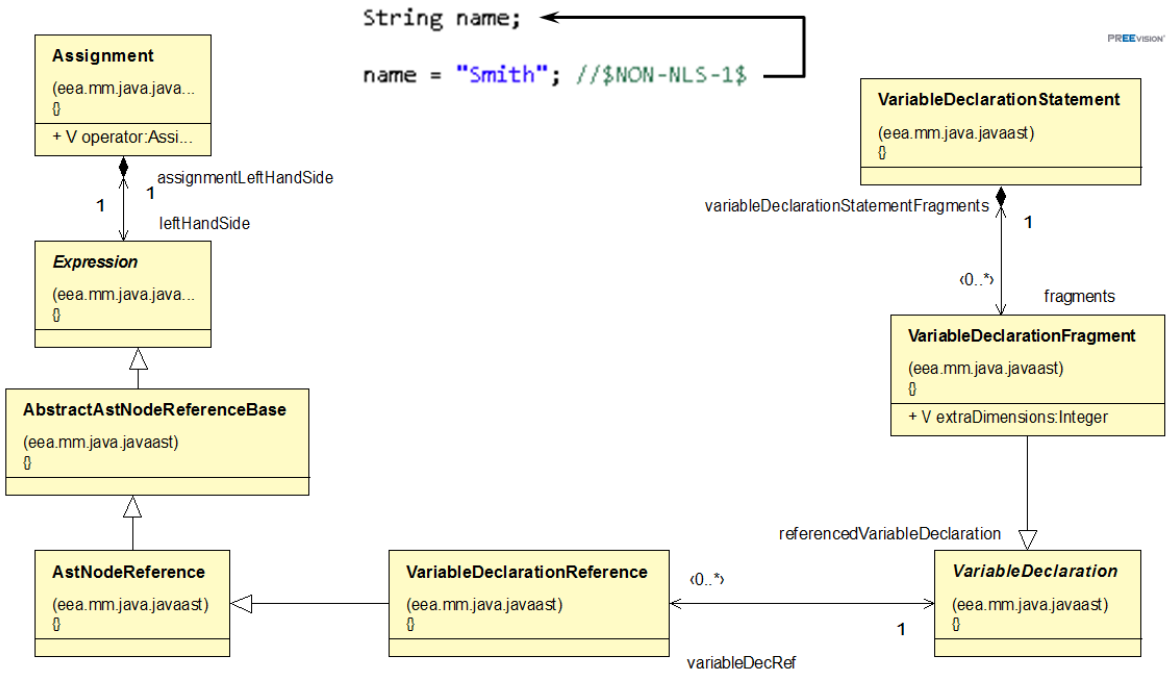


Abbildung 57: Initialisierung einer Variable

In Abbildung 58 geht es um die Verwendung eines Labels im Zusammenhang mit einer Break-Anweisung, um eine bestimmte Schleife abbrechen zu können. Das *BreakStatement*

benötigt dazu eine Referenz auf das *LabelStatement*, was über die *LableReference* ermöglicht wird. Das *BreakStatement* ist die Client-Metaklasse und das *LabeledStatement* die Supplier-Metaklasse.

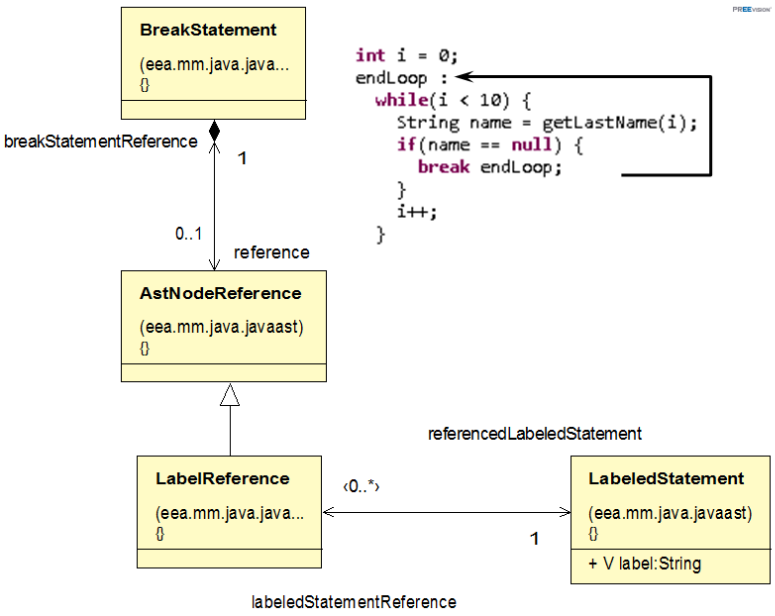


Abbildung 58: Verwendung eines Labels

In dem Quellcodebeispiel in Abbildung 59 wird eine Klasse über eine Import-Anweisung für die Verwendung bekannt gemacht. Die *ImportDeclaration* braucht eine Referenz auf die *TypeDeclaration* (die Java-Klasse). Die Metaklasse *TypeReferenceByName* stellt diese Referenz zur Verfügung. Die *ImportDeclaration* ist dabei die Client-Metaklasse und die *TypeDeclaration* die Supplier-Metaklasse. Die Import-Anweisung stellt insofern eine Besonderheit dar, weil sie eigentlich im Quellcode-Modell nicht mehr benötigt wird. Der Import legt den vollqualifizierten Namen der zu importierenden Klasse fest, die dann in der Klasse mit dem Import verwendet wird. An dieser Verwendungsstelle gibt es jedoch schon eine Referenz auf diese Klasse (siehe Abbildung 55) und damit ist die verwendet Klasse genau definiert und die Import-Anweisung erübrigt sich. Bei der Generierung des Quellcode-Texts aus dem Quellcode-Modell müssten die Import-Anweisungen automatisch hinzugefügt werden, damit der Java Compiler den Text fehlerfrei parsen kann. Der Softwareentwickler kann jedoch über die Import-Anweisungen Einfluss darauf nehmen, wie die Klasse im Text erscheint in dem er zum Beispiel nur ein Java Package importiert anstelle der Klasse. Dann muss die Klasse mit einem Teil des vollqualifizierten Namens im Quellcode-Text erscheinen. Deswegen wurde die Import Deklaration erstmal nicht aus dem Quellcode-Modell entfernt.

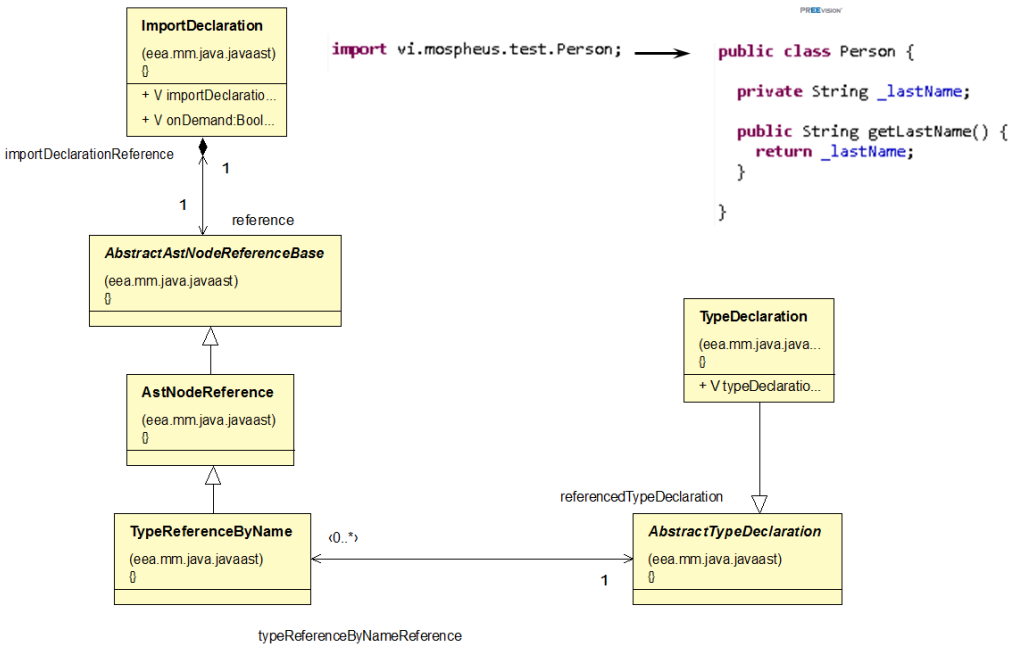


Abbildung 59: Verwendung einer Import-Anweisung

Die oben aufgeführten Metamodelle sind nicht vollständig. So gibt es zum Beispiel noch Referenzen auf Annotations und Enumerations und auch aus Javadoc heraus können Klassen oder Methoden referenziert werden. Javadoc ermöglicht die Dokumentation von Quellcode basierend auf HTML und ist Teil des Java Development Kits (JDK). In Abbildung 60 ist ein Beispiel dargestellt mit einem Link auf die Klasse „Person“.

```
/**
 * Tests the class {@link vi.mospheus.test.Person}
 */
public class UnitTest extends TestCase {

  public void testPerson() {
    Person p = new Person();
```

Abbildung 60: Javadoc-Beispiel mit einem Link auf die Klasse „Person“

Diese Links können sehr schnell veralten und auf eine nicht mehr existente Quellcode-Stelle zeigen, zum Beispiel nach einer Umbenennung oder einer Verschiebung des referenzierten Quellcodes. Da ungültig gewordene Referenzen im Javadoc nicht vom Java Compiler über Syntaxfehler angezeigt werden (im Gegensatz zu fehlenden Referenzen im Quellcode), kommen diese „kaputten“ Referenzen in der Praxis sehr häufig im Quellcode vor. Im Quellcode-Modell werden jedoch an diesen Stellen tatsächlich Referenzen auf die AST Artefakte abgespeichert und so führt eine Refaktorierung zu keinen fehlerhaften Links. Damit wird mit Hilfe des Metamodells für den Quellcode die Qualität der Dokumentation verbessert.

Es gibt noch eine weitere Besonderheit. Eine Java Klasse, die keinen Konstruktor besitzt, erhält automatisch einen Default-Konstruktor ohne Parameter. Nachdem der Default-Konstruktor nicht im Quellcode-Text steht und damit das AST Artefakt auch nicht existiert, kann es in der Anweisung mit dem Neuanlegen einer Klasse nicht referenziert werden. Dieser Punkt wird bei der Frage des indirekten Konflikts nochmals untersucht (siehe Kapitel 4.4.3.3).

Eine weitere Abhängigkeit, die so nicht im AST von Eclipse erkennbar ist, ist das Überschreiben einer Methode aus einer Basisklasse oder einem Interface. Hier sind Änderungen in einer Methode für die Methoden, die diese Methode überschreiben, relevant. Daher gibt es im Metamodell eine Assoziation zwischen einer *MethodDeclaration* und einer *MethodDeclaration*. Die *MethodDeclaration* ist dabei Supplier- und Client-Metaklasse zugleich. In dem Beispiel in Abbildung 61 wird die „toString“ Methode der Basisklasse überschrieben. Mit der Annotation „@Override“ wird die Methode, die eine andere Methode überschreibt, gekennzeichnet.

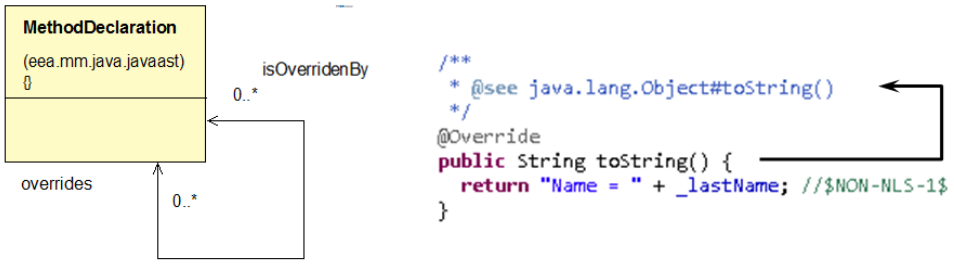


Abbildung 61: Überschreiben der Methode „toString“ der Klasse „Object“

4.3.3 Import und Speichern des ASTs

Beim erstmaligen Import eines Plugins oder beim Speichern des Quellcodes müssen diese neuen Assoziationen gesetzt oder aktualisiert werden. Beim Aufbau des AST Modells werden alle offenen Assoziationen zwischengespeichert. Am Ende des Imports oder Speichervorgangs werden die Abhängigkeiten mittels Eclipse Funktionalität aufgelöst und für jedes Client-Artefakt das entsprechende Supplier-Artefakt an Hand des Plugins und des vollqualifizierten Namens gesucht. Für die Methode müssen neben dem Namen die Parameter bei der Suche mitberücksichtig werden, da es mehrere Methoden mit gleichen Namen aber unterschiedlichen Parametern geben kann.

Im Gesamtmodell können jedoch durchaus mehrere Workspaces verwaltet werden, das heißt das Supplier-Artefakt kann es mehr als nur einmal im Gesamtmodell geben. Daher muss ein Suchbereich festgelegt werden. Der gesamte Workspace mit allen Plugins wird in einer

SourceCodePackage-Hierarchie verwaltet (siehe Kapitel 3.7.1). Die Wurzel der *SourceCodePackage*-Hierarchie wird über das Attribut „isScopeRoot“ gekennzeichnet und damit wird der Suchbereich definiert und das Supplier-Artefakt wird nur unter diesem *SourceCodePackage* gesucht.

Im Databackbone wird man nur den Quellcode ablegen, der von den Softwareentwicklern erstellt und weiterentwickelt wird und damit für die Historie, die Dokumentation und die Tests wichtig sind. Das trifft auf JARs, die von Java bereitgestellt werden (die Java Runtime Class Libraries) oder andere externe JARs nicht zu. Aus dem AST Modell im Databackbone werden jedoch Klassen, Methoden oder andere Quellcode-Teile aus diesen JARs verwendet (zum Beispiel „ArrayList“, „OutputStream“, „Exception“, „Math“ und noch viel mehr aus der Java Runtime Class Library). Dieser verwendete Quellcode stellt die Programmierschnittstelle (API) des JARs dar. Jedes der Client-Artefakte im Modell braucht laut Metamodell ein Supplier-Artefakt, die jedoch im Modell für eine Assoziation nicht vorhanden sind. Prinzipiell gibt es zwei Lösungen dafür:

- Es wird im Metamodell die Möglichkeit vorgesehen, dass ein Referenz-Artefakt (Client-Artefakt) ohne ein tatsächlich referenziertes Supplier-Artefakt auskommt. Stattdessen wird nur der Name des Supplier-Artefakts im Referenz-Artefakt abgespeichert. Dieser wird für die Generierung des Quellcode-Texts aus dem AST Modell benötigt.
- Nur die API des verwendeten Quellcodes wird im Modell abgelegt und kann von den Client-Artefakten referenziert werden. Das heißt es werden zum Beispiel alle benötigten öffentlichen Interfaces und Klassen mit ihren öffentlichen Methoden und Attributen im Modell abgespeichert. Die Methode hat dabei keinen Inhalt. Es werden nur die Artefakte im Modell erzeugt, die tatsächlich referenziert werden. Wenn bei der Auflösung der Abhängigkeiten und bei der entsprechenden Suche das Supplier-Artefakt nicht im Modell gefunden wird, wird es mit seinem Kontext (JAR, Java Package, Klasse) automatisch angelegt und für die Referenz verwendet. So wächst die API im Modell mit der Zeit und mit jeder neuen Verwendung einer API Methode oder Klasse.

Für die indirekte Konfliktbehandlung wären beide Lösungen denkbar, weil das parallele Editieren der API nicht verhindert werden muss. Die API dieser JARs wird nicht während der Softwareentwicklung bearbeitet, sondern bleibt während der Entwicklung unverändert. Für

diese Arbeit wurde die zweite Lösung gewählt. Damit ist es möglich die verwendete API im Modell zu sehen und auch zu analysieren (siehe auch Abbildung 62).

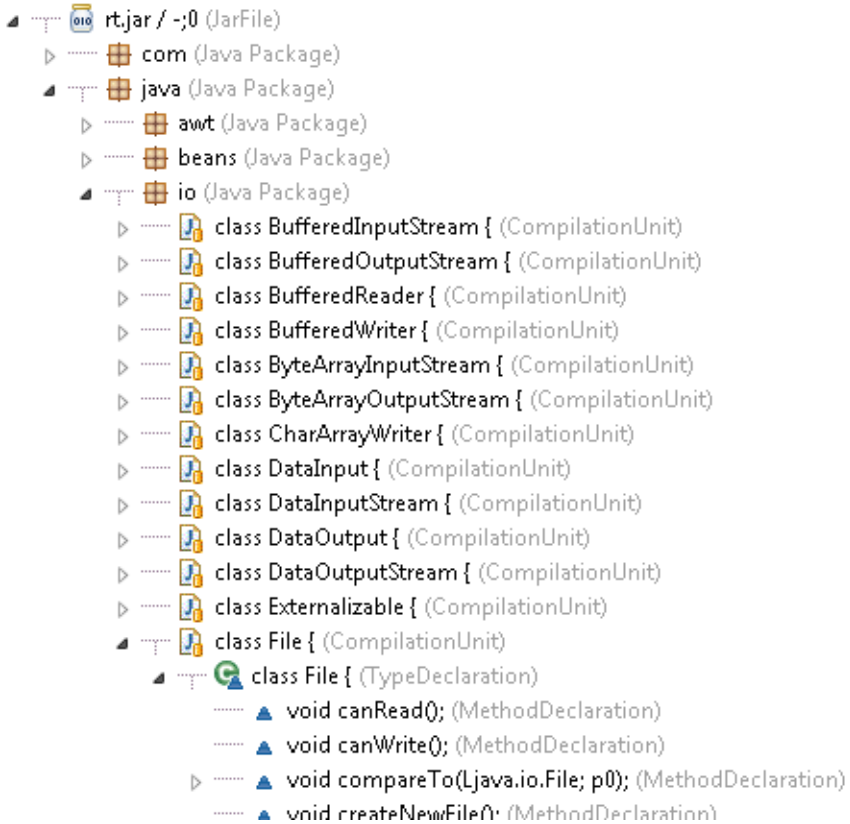


Abbildung 62: API im Modell für das JAR "rt.jar"

4.3.4 Metamodell und Syntax Fehler

Ein Ziel, das in Kapitel 4.1.3 definiert wurde, ist das Verhindern von Syntaxfehlern im Software Configuration Management Repository. Quellcode, der nach der Abgabe von Änderungen, Syntaxfehler beinhaltet, kann nicht gebaut und getestet werden. Eine Textdatei kann beliebigen Text beinhalten, der nur wenig oder überhaupt nicht mit der Syntax von Java übereinstimmen muss. Die herkömmlichen dateibasierten SCM Repositories werden die Abgabe solcher Dateien nicht verhindern. Ein Metamodell definiert eine Syntax für das Modell beziehungsweise in diesem Fall für den Quellcode (siehe Kapitel 2.3). Damit kann nicht beliebiger Text im Modell gespeichert werden.

Im Quellcode-Text wird die Verbindung zwischen Methodendeklaration und Methodenaufruf über einen vollqualifizierten Namen realisiert: Paketname, Klassenname und Methodenname. Durch die Veränderung des Namens in der Methodendeklaration aber nicht im Methodenaufruf kann diese Assoziation brechen, weil die Namen nicht länger übereinstimmen und es entsteht ein Syntaxfehler. Im Metamodell ist für die Assoziation

zwischen *MethodDeclarationReference* und *MethodDeclaration* eine Kardinalität von „1..1“ vorgesehen: die *MethodInvocation* muss genau eine *MethodDeclaration* haben (siehe Abbildung 56). Daher stellt das Metamodell sicher, dass es den Syntaxfehler „Methodenaufruf ohne die passende Methodendeklaration“ nicht geben kann. Nur durch die Verwendung eines Metamodells für den Quellcode sind einige Syntaxfehler nicht mehr möglich. Aber das Metamodell verhindert nicht alle Syntaxfehler. Zum Beispiel, wenn die Anzahl der Parameter und die Datentypen der Parameter im Methodenaufruf nicht mit der Methodendeklaration übereinstimmen, dann ist es immer noch möglich einen gültigen AST aufzubauen, der auch im Repository gespeichert werden kann. Der Java Compiler entdeckt solche Syntaxfehler und es macht keinen Sinn, diese Funktionalität in das Metamodell zu integrieren. Um das Ziel „keine Syntaxfehler im SCM Repository“ zu erreichen, müssen daher folgende drei Punkte beachtet werden:

- Der Softwareentwickler sollte nicht in der Lage sein, Quellcode mit Syntaxfehler abzugeben. Wenn Eclipse irgendwelche Syntaxfehler im aktuellen Quellcode anzeigt, wird die Abgabe abgelehnt. Über das Eclipse Feature „Continuous Build“ wird sichergestellt, dass der Quellcode mit jeder Änderung immer sofort gebaut wird und Syntaxfehler sofort sichtbar werden[4].
- Bei einer Teil-Abgabe will der Softwareentwickler nicht alle seine Änderungen aus dem gesamten Workspace abgeben, sondern nur eine Teilmenge davon. In diesem Fall müssen alle zusammenhängenden Änderungen abgegeben werden, ansonsten können Syntaxfehler im SCM Repository entstehen. Das heißt wenn es Änderungen im Client-Artefakt und im Supplier-Artefakt gibt, dann müssen beide Änderungen zusammen abgegeben werden. Da PREEvision derzeit noch keine Teil-Abgabe unterstützt, stellt das in dieser Arbeit noch kein Problem dar.
- Syntaxfehler auf Grund von indirekten Konflikten müssen durch das feingranulare Sperren verhindert werden.

4.4 Feingranulare AST Sperren

4.4.1 Einleitung

In Kapitel 2.6.6 wurde bereits die Vorgehensweise für das Bearbeiten des Modells in PREEvision mit dem pessimistischen Sperrkonzept vorgestellt. Dieses Konzept wird nun auf das Modell mit dem AST angepasst und angewendet. Im Wesentlichen gibt es geteilte

[4] Wenn die Option „Build Automatically“ in der Eclipse Entwicklungsumgebung aktiv ist.

Sperren und exklusive Sperren. Diese Sperren können jetzt für die AST Artefakte verwendet werden, um direkte und indirekte Konflikte zu verhindern.

Ein Ziel ist es so viel paralleles Bearbeiten des Quellcodes zu erlauben wie möglich (siehe Kapitel 4.1.3). Um das zu erreichen werden die geteilten Sperren benötigt (siehe Kapitel 2.6.6). Das soll am folgenden Beispiel verdeutlicht werden: Wenn ein Softwareentwickler einen neuen Methodenaufruf in seinem Quellcode hinzufügen will, dann wird ein AST Artefakt von der Klasse *MethodInvocation* erzeugt mit einer Assoziation zu der entsprechenden *MethodDeclaration*. Um sicherzustellen, dass kein anderer Softwareentwickler zu selben Zeit die Methodensignatur der aufgerufenen Methode verändert oder die Methode vielleicht sogar löscht, wird eine Sperre auf der *MethodDeclaration* benötigt. Eine exklusive Sperre würde andere Softwareentwickler daran hindern ebenfalls einen neuen Methodenaufruf zu dieser Methode in ihrem Quellcode hinzuzufügen, weil nur ein Softwareentwickler eine exklusive Sperre auf der *MethodDeclaration* besitzen kann. Diese Änderungen würden aber niemals einen direkten oder indirekten Konflikt verursachen. Mehrere Entwickler können eine geteilte Sperre für ein AST Artefakt besitzen. Eine exklusive Sperre ist für ein Artefakt mit geteilter Sperre nicht mehr möglich. Das heißt in diesem Fall wird eine geteilte Sperre für die *MethodDeclaration* verwendet. Damit kann die Methode mehrfach verwendet werden, aber eine Veränderung der Methode ist nicht mehr möglich, bis die neuen Methodenaufrufe abgegeben worden sind.

Damit der Softwareentwickler den Quellcode ohne Konflikte ändern kann, müssen gewisse AST Artefakte mit exklusiven und geteilten Sperren versehen werden. Dazu werden in weiterer Folge Sperrregeln definiert. In den Sperrregeln wird zwischen Kompositions- und Assoziationsrelationen unterschieden (siehe Abbildung 63).

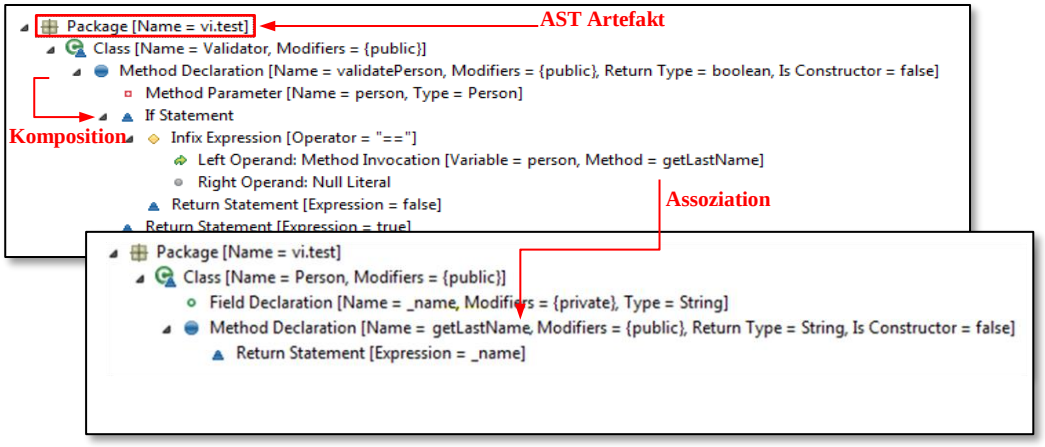


Abbildung 63: Kompositions- und Assoziationsbeziehungen

Assoziationsrelationen mit den Client- und Supplier-Artefakten wurden schon ausführlich behandelt. Die AST Artefakte bilden einen hierarchischen Baum, die über Kompositionsrelationen miteinander verbunden sind. Das Komposit-Artefakt besitzt ein oder mehrere Komponenten-Artefakte (siehe Metamodell in Abbildung 64). Zum Beispiel ist die If-Anweisung unter einer Methodendeklaration angeordnet. Die Methodendeklaration ist dabei das Komposit-Artefakt und die If-Anweisung das Komponenten-Artefakt.

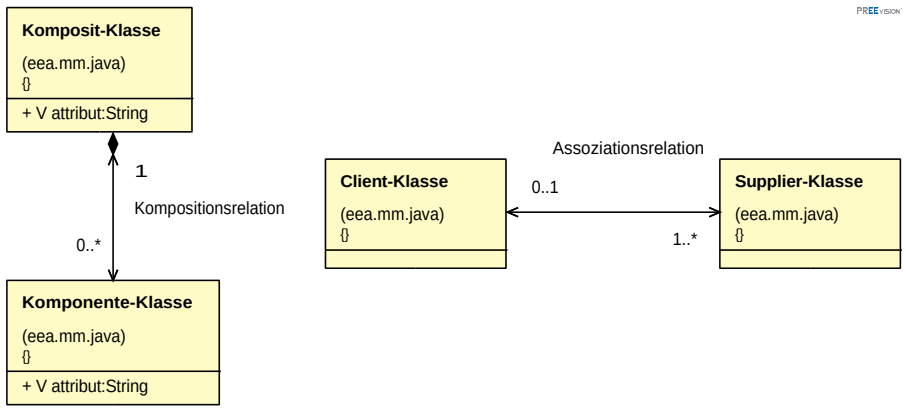


Abbildung 64: Kompositions- und Assoziationsbeziehungen im Metamodell

Die folgenden drei Konfliktarten werden für die Sperrregeln berücksichtigt: direkter Konflikt, indirekter Konflikt und Namenskonflikt.

4.4.2 Sperrregeln für direkte Konflikte

4.4.2.1 Konfliktsituationen beim Verschmelzen der Modelle

Nach dem der Softwareentwickler sein lokales Modell mit den AST Artefakten in seinem Workspace verändert hat, muss beim Abgeben das Modell mit dem Modell vom Server verschmolzen werden (siehe auch Kapitel 2.6.3). Das Verschmelzen ist deswegen notwendig,

weil das Modell vom Server eventuell von anderen Softwareentwicklern verändert worden ist und daher nicht einfach überschrieben werden kann. Es gibt dabei drei relevante Modellstände:

- Modellstand 1: Der letzte Modellstand, den der Softwareentwickler vom Server geladen hat. Auf Basis dieses Modellstands führt er seine lokalen Änderungen durch.
- Modellstand 2: Der Modellstand, der durch die lokalen Änderungen des Softwareentwicklers entstanden ist.
- Modellstand 3: Der Modellstand, der durch die Abgabe anderer Softwareentwickler entstanden ist und aktuell auf dem Server vorhanden ist.

Durch das Verschmelzen müssen die Änderungen zwischen Modellstand 1 und 2 mit den Änderungen zwischen Modellstand 1 und 3 zusammengeführt werden. Ein direkter Konflikt liegt dann vor, wenn während das Verschmelzen eine Entscheidung notwendig wird, ob die Änderung zwischen Modellstand 1 und 2 oder die Änderungen zwischen Modelstand 1 und 3 übernommen werden soll. Für neue Artefakte kann kein direkter Konflikt entstehen, weil das Artefakt nur im Modellstand 2 existiert. Die Auswahl zwischen zwei sich widersprechenden Änderungen kann die Komponenten für das Verschmelzen nicht automatisch durchführen und die Entscheidung müsste von einem Menschen getroffen werden. Diese Konfliktsituation gilt es zu verhindern. In weiterer Folge werden die möglichen Änderungen am Modell, die zu einem Konflikt führen können, im Einzelnen analysiert: das Ändern eines Attributwertes (1), das Löschen eines Artefakts (2) und das Löschen oder Anlegen einer Kompositionsrelation (3) (siehe Abbildung 65). Das Löschen oder Anlegen von Assoziationsrelationen wird erst für die indirekten Konflikte relevant.

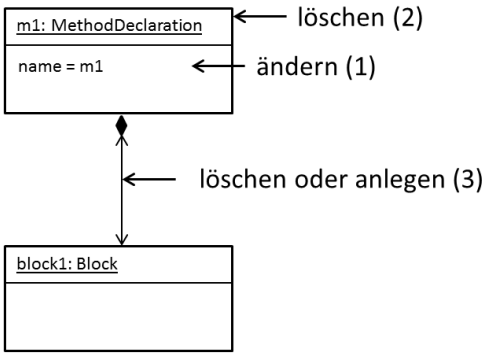


Abbildung 65: Mögliche Änderungen im Modell

4.4.2.2 Ändern eines Attributes

Wenn zwei Entwickler zur gleichen Zeit das Attribut „Name“ einer *MethodDeclaration* ändern, dann ist ein Verschmelzen nicht möglich, weil die Komponente für das Verschmelzen nicht entscheiden kann, welcher Attributwert der richtig ist. Daraus ergibt sich die erste Sperrregel, die eine exklusive Sperre benötigt (siehe Kapitel 2.6.6):

Regel 1:

Änderung eines Attributes: Das AST Artefakt mit dem Attribut benötigt eine exklusive Sperre.

4.4.2.3 Löschen eines Artefakts

Beim Löschen eines AST Artefakts werden alle Komponenten-Artefakte unterhalb des zu löschenden Artefakts ebenfalls gelöscht. Zum Beispiel, wenn eine Methode gelöscht wird, wird der vollständige Inhalt der Methode gelöscht. Das Löschen und Ändern eines AST Artefakts zur gleichen Zeit führt ebenfalls zu einem direkten Konflikt.

Regel 2:

Löschung eines AST Artefakts: Das AST Artefakt und alle Komponenten-Artefakte unterhalb des AST Artefakts benötigen eine exklusive Sperre.

4.4.2.4 Löschen oder Anlegen einer Kompositionsrelation

Als nächstes soll es um das Anlegen eines neuen AST Artefakts mit der dazugehörigen Kompositionsrelation gehen, zum Beispiel eine Java Klasse. Die Klasse wird unterhalb eines Pakets angelegt. Um sicherzustellen, dass das Paket nicht von einem anderen Softwareentwickler gelöscht wird, wird eine Sperre für das Paket benötigt. Eine exklusive Sperre würde andere Entwickler daran hintern parallel dazu eine Klasse im selben Paket anzulegen, was aber kein Konflikt darstellt. Deswegen wird an dieser Stelle eine geteilte Sperre verwendet (siehe Kapitel 2.6.6).

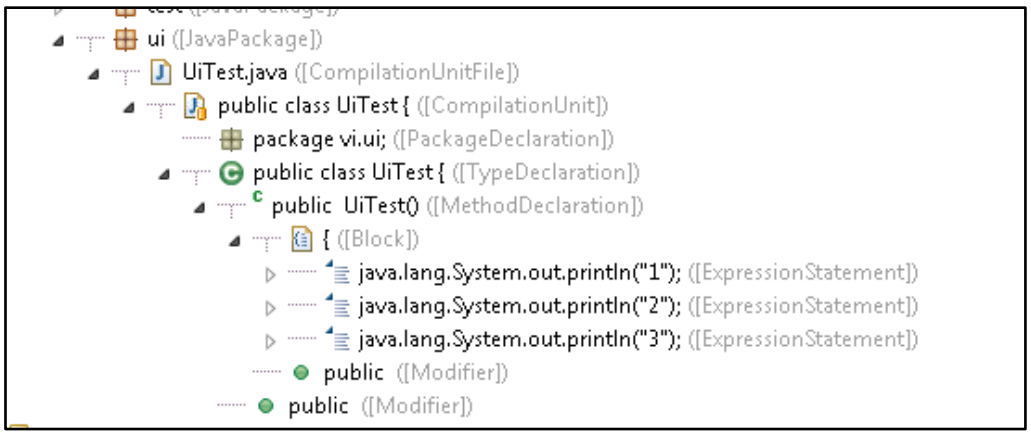


Abbildung 66: Abstract Syntax Tree mit mehreren *ExpressionStatements*

Im AST Beispiel in Abbildung 66 sind drei *ExpressionStatement*s unter einem *Block* zu sehen. Der *Block* entspricht der geschweiften Klammer im Quellcode und kann eine Liste von Anweisungen aufnehmen. Die Reihenfolge der Anweisung ist relevant. In diesem Beispiel soll die Ausgabe auf der Konsole „1“, „2“ und „3“ sein. Wenn ein Entwickler eine neues *ExpressionStatement* hinzufügt, um „4“ nach der „3“ auszugeben und es gibt nur eine geteilte Sperre auf dem *Block*, dann könnte ein anderer Entwickler zu selben Zeit ein neues *ExpressionStatement* hinzufügen, um „fertig“ nach der „3“ auszugeben. In diesem Fall entsteht ein direkter Konflikt, weil die Komponente für das Verschmelzen nicht entscheiden kann, ob „4“ oder „fertig“ nach der „3“ kommen soll. Wenn die Reihenfolge der AST Artefakte in einer Relation relevant ist, wird eine exklusive Sperre benötigt. Die Reihenfolge der Klassen im Paket ist nicht relevant. Auch eine geänderte Reihenfolge der Methoden in einer Klasse kann keine Syntaxfehler verursachen oder das Verhalten des Programms beeinflussen.

Man kann natürlich jetzt argumentieren, dass der Entwickler sehr wohl die Methoden in einer gewissen und nicht in einer beliebigen Reihenfolge in der Klasse sehen will. Jedoch kann die gewünschte Reihenfolge sich durchaus von einem Anwendungsfall zum anderen unterscheiden. Wenn der Entwickler zum Beispiel eine Methode mit einem bestimmten Namen sucht, dann kann eine alphabetische Reihenfolge Sinn machen. Um sich einen Überblick über die implementierte Funktionalität der Klasse zu verschaffen, sind die „public“ Methoden am Anfang der Liste der Methoden sinnvoll. In der Regel wird der Quellcode aber in sehr festen Strukturen dem Softwareentwickler präsentiert, die sich stark am Quellcode-Text orientieren. Der Quellcode könnte aber durchaus in unterschiedlichen Ansichten dargestellt werden, die auch über mehrere Klassen hinweggehen könnten. Zum Beispiel

könnten Methoden unterschiedlicher Klassen, die für einen bestimmten Aspekt der Software relevant sind, in einer Ansicht zusammengefasst werden [63]. Das AST Artefakt *MethodDeclaration* könnte im Metamodell um zusätzliche Attribute (zum Beispiel Kategorie) erweitert werden, die dann die Reihenfolge oder die Auswahl der Methoden für unterschiedliche Sichten bestimmt.

Regel 3:

Erzeugung oder Löschung von Kompositionsrelationen mit der Kardinalität „0..n“: Wenn die Reihenfolge der Komponenten-Artefakte relevant ist, dann benötigt das Komposite-Artefakt eine exklusive Sperre. Wenn die Reihenfolge irrelevant ist, dann eine geteilte Sperre.

Geordnete Relationen sind zum Beispiel Anweisungen in einer Methode oder Parameter in einer Methodendeklaration.

Im nächsten Schritt soll eine Java Klasse von einer anderen Klasse abgeleitet werden. Dafür wird eine *TypeReference* unterhalb der *TypeDeclaration* (Java Klasse) angelegt. Natürlich kann die Klasse nur von einer Basisklasse in Java abgeleitet werden. Wenn ein Entwickler eine Klasse von der Basisklasse B1 ableitet und ein anderer Entwickler parallel zur gleichen Zeit von der Basisklasse B2, dann entsteht ein direkter Konflikt. Die Komponente für das Verschmelzen kann nicht entscheiden, welcher der beiden Basisklassen B1 oder B2 die Richtige ist. Deswegen wird in diesem Fall eine exklusive Sperre für diese Klasse benötigt.

Regel 4:

Erzeugung oder Löschung von Kompositionsrelationen mit der Kardinalität „0..1“: Das Komposite-Artefakt benötigt eine exklusive Sperre.

Für die Ableitung einer Kasse von einem Interface gilt die Regel 3: Die Reihenfolge der Interfaces ist nicht relevant und daher wird nur eine geteilte Sperre beim Hinzufügen oder Entfernen eines Interfaces für die Klasse benötigt.

Als nächstes soll eine Methode verändert werden. Es wurde bereits festgestellt, dass die Anweisungen in einer Methode geordnet sind. Für das Einfügen neuer Anweisungen, das Löschen bestehender Anweisungen oder eine Änderung der Reihenfolge der Anweisungen wird eine exklusive Sperre auf dem Komposit-Artefakt (dem *Block*) benötigt (siehe auch Abbildung 67).

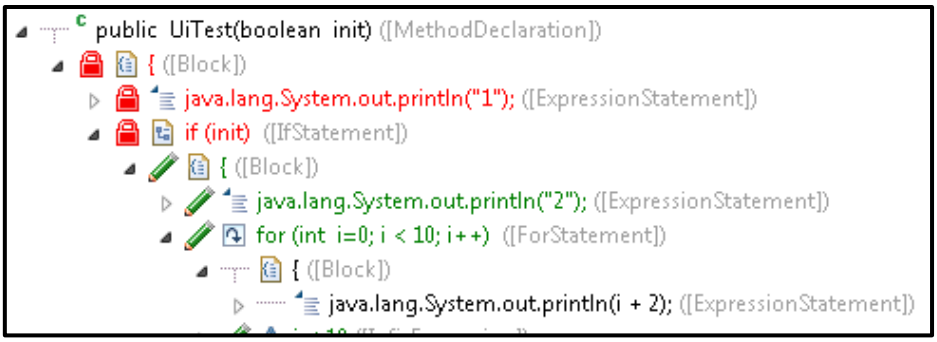


Abbildung 67: AST Artefakte in einer *MethodDeclaration*

Änderungen an einer bestehenden Anweisung könnte auch ohne eine exklusive Sperre auf dem *Block* durchgeführt werden. Jedoch um die Logik zu vereinfachen wird auch in diesem Fall eine exklusive Sperre für den *Block* verwendet. Wenn der Entwickler den Quellcode verändert, wird der Entwickler in den meisten Fällen neue Anweisungen erzeugen, bestehende Anweisungen löschen oder die Reihenfolge ändern und damit wird auch eine exklusive Sperre notwendig. Wenn der *Block* exklusiv gesperrt ist, dann können alle Anweisungen unter dem *Block* ebenfalls exklusiv gesperrt werden, weil kein anderer Entwickler eine Anweisung unter dem *Block* verändern kann, weil er dazu eine exklusive Sperre auf dem *Block* benötigen würde. Wenn es eine Anweisung unter dem exklusiv gesperrten Block gibt, die auch einen *Block* besitzt (zum Beispiel ein For-Anweisung oder eine If-Anweisung), dann ist es nicht notwendig auch diesen *Block* zu sperren. Mit einem neuen *Block* beginnt eine neue geordnete Liste von Anweisungen und deswegen sind in diesem *Block* Sperren anderer Entwickler wieder möglich. In Abbildung 67 wurde von einem Entwickler die Anweisungen unter der *MethodDeclaration* gesperrt (in roter Farbe) und von einem anderen Entwickler die Anweisungen unter dem *IfStatement* (in grüner Farbe). Der *Block* unter dem *ForStatement* könnte wiederum von einem weiteren Entwickler gesperrt werden.

Regel 5:

Einfügen, löschen oder ändern von Anweisungen innerhalb einer Methoden Deklaration: Beginnend mit dem Komposit-Artefakt Block benötigen alle Komponenten-Artefakte und der Block selber eine exklusive Sperre bis zum nächsten Block.

4.4.3 Sperrregeln für indirekte Konflikte

Ein indirekter Konflikt entsteht, wenn nach dem Verschmelzen der Abstract Syntax Trees ein Syntaxfehler entstanden ist, der vor dem Verschmelzen noch nicht im Quellcode vorhanden

war. Das ist der Fall, wenn das Supplier-Artefakt verändert wurde ohne die Client-Artefakte anzupassen.

Die meisten möglichen Konflikte können über eine allgemeine Sperrregel verhindert werden unter Berücksichtigung der Assoziation zwischen Client-Artefakt und Supplier-Artefakt. Es gibt jedoch einige Spezialfälle, die gesonderte gelöst werden müssen. In diesen Fällen ist die Assoziation zwischen Client-Artefakt und Supplier-Artefakt nicht so offensichtlich oder nur indirekt vorhanden.

4.4.3.1 Allgemeine Regel

Lewin [27] verhindert indirekte Konflikte dadurch, dass das Supplier- und Client-Artefakt nicht konkurrierend durch zwei verschiedene Entwickler verändert werden dürfen. Daher werden, sobald eine Änderung am Supplier-Artefakt vorgenommen wird, alle Client-Artefakte sofort gesperrt. Die Client-Artefakte können dann nicht mehr durch einen anderen Entwickler verändert werden. Für diese Arbeit wurde eine andere Vorgehensweise gewählt, die weniger restriktiv ist. Das Ändern eines Supplier-Artefakts und damit auch aller abhängigen Client-Artefakte kann in vielen Fällen sehr gut automatisiert werden. Die meisten Entwicklungsumgebungen (auch Eclipse) stellen entsprechende Funktionalitäten für die Refaktorierung bereit wie zum Beispiel für das umbenennen einer Methode oder die Änderung der Methodensignatur einer Interface-Methode. Die Entwicklungsumgebung passt dann alle Client-Artefakt entsprechend der Änderung automatisch an. Das heißt die Änderung am Supplier-Artefakt und an den Client-Artefakten passieren fast gleichzeitig und für jede diese Änderungen sind entsprechend der bereits definierten Sperrregeln Sperren notwendig. Die Sperren der Client- und Supplier-Artefakte werden zur gleichen Zeit vom Server angefordert. Aber selbst wenn der Softwareentwickler die Änderungen manuell durchführt und über einen längeren Zeitraum, so muss er doch bei der Änderung eines Supplier-Artefakts alle Client-Artefakte anpassen, bevor er seine Änderungen abgeben kann. Um Syntaxfehler im Repository zu verhindern, wurde bereits in Kapitel 4.3.4 festgelegt, dass Syntaxfehler nicht abgegeben werden dürfen. Bei Anpassungen über einen längeren Zeitraum gibt es im Wesentlichen zwei Fälle: 1. Der Entwickler ändert das Supplier-Artefakt und passt die Client-Artefakte nach und nach an. Entsprechende Sperren werden mit den Änderungen vom Server geholt. 2. Der Entwickler ändert das Supplier-Artefakt und kann jedoch die Client-Artefakte nicht anpassen, weil sie durch Sperren anderer Entwickler gesperrt sind. Dann muss der Entwickler warten, bis die Änderungen der anderen Entwickler abgeschlossen und die

Sperren wieder freigegeben worden sind. In keinen der beiden Fälle kommt es zu Konfliktsituationen, die im Repository Syntaxfehler verursachen oder ein Verschmelzen notwendig machen würden. Es besteht auch nicht die Notwendigkeit mit dem Supplier-Artefakt auch gleichzeitig die Client-Artefakte zu sperren. Für bestehende AST Artefakte sind daher für diese Anwendungsfälle keine zusätzlichen Sperrregeln notwendig.

Für das Erzeugen neuer Assoziationen zwischen Client-Artefakten und Supplier-Artefakten werden jedoch zusätzliche Sperrregeln benötigt. Über diese Sperre wird sichergestellt, dass das Supplier-Artefakt nicht geändert oder gelöscht wird bis das Client-Artefakt mit der neuen Assoziation in die Datenbank abgespeichert wurde. Der Entwickler kann sich zum Beispiel darauf verlassen, dass der neu erstellte Methodenaufruf wirklich so korrekt ist, weil über die Sperre jede Veränderung an der Methodendeklaration verhindert wird. In diesem Fall wäre der Entwickler, der das Supplier-Artefakt verändert will, überhaupt nicht in der Lage, das entsprechende Client-Artefakt anzupassen, weil dieses Artefakt neu ist und der Entwickler noch nichts von diesem Client-Artefakt weiß.

In diesen Fällen sind nur geteilte Sperren notwendig und es ergibt sich daraus die nächste Sperrregel:

Regel 6:

Erstellung neuer Assoziationen: Das Supplier-Artefakt benötigt eine geteilte Sperre.

Es gibt jetzt eine Reihe von speziellen Modellkonstellationen, die genauer untersucht und unter Umständen gesondert behandelt werden müssen.

4.4.3.2 Spezialfall „Überschreiben von Methoden“

Wie bereits im Kapitel 4.3.2 erwähnt, wurde für das Überschreiben einer Methode eine Assoziation im Modell hinzugefügt. Wenn der Entwickler eine neue Methode erstellt um eine bestehende Methode der Basisklasse zu überschreiben, wird eine neue Assoziation zwischen den Methoden erstellt und die Regel 6 kommt zur Anwendung. Es wird sichergestellt, dass die überschriebene Methode nicht verändert oder gelöst wird. Weitere Regeln sind nicht notwendig.

4.4.3.3 Spezialfall „Default-Konstruktor“

In einer Java Klasse ohne einen Konstruktor stellt Java automatisch eine Default-Konstruktor ohne Parameter zur Verfügung, der für das Erzeugen einer Instanz verwendet werden kann.

Ab dem Anlegen eines ersten Konstruktors in der Klasse steht der Default-Konstruktor nicht länger zur Verfügung. Jedoch benötigt das AST Artefakt der Metaklasse *InstanceCreation* immer einen Konstruktor (ein Artefakt der Metaklasse *MethodDeclaration*), um die Assoziation „constructor invocation“ anlegen zu können. Außerdem muss dieser Default-Konstruktor gesperrt werden können, um das Löschen des Default-Konstruktors durch das Anlegen eines ersten Konstruktors in der Klasse zu verhindern. Deswegen wird beim Anlegen einer neuen Klasse ohne Konstruktor im Modell auch immer automatisch ein Default-Konstruktor ohne Parameter erzeugt, der für die Assoziation und für die Sperre verwendet werden kann. Zusätzlich Sperrregeln sind nicht notwendig.

4.4.3.4 Spezialfall „Return-Anweisung“

In der Return Anweisung gibt es eine Abhängigkeit zwischen zwei AST Artefakten, die nicht über einen expliziten Namen definiert ist: der Methodenrückgabetyp und der Wert in der Return Anweisung. Der Datentyp an beiden Stellen muss übereinstimmen. Wenn der Datentyp in der *MethodDeclaration* für den Rückgabewert verändert wird, müssen alle *ReturnStatement*s angepasst werden. Zum Beispiel, wenn der Return-Datentyp in der Methodendeklaration von „void“ auf „int“ geändert wird, müssen alle Return-Anweisungen von „return“ zu „return <*integer*>“ geändert werden. Für diese Abhängigkeit wurde keine neue Assoziation im Metamodell eingeführt und deswegen ist eine zusätzliche Sperrregel notwendig:

Regel 7:

Hinzufügen eines neuen ReturnStatements: Die Komposit-Methodendeklaration benötigt eine geteilte Sperre.

4.4.3.5 Interfaces und abstrakte Methoden

Beim Hinzufügen, Löschen oder Ändern einer Methode in einem Interface ist mit den bisher definierten Sperrregeln nur eine geteilte Sperre auf dem Interface notwendig. Das heißt ein anderer Entwickler kann zur gleichen Zeit eine neue Klasse anlegen, die von diesem Interface erbt und benötigt dazu auch nur eine geteilte Sperre auf dem Interface. Damit kann es jedoch zu einem Syntaxfehler im Repository kommen, weil die neue Klasse eventuell Änderungen im Interface nicht berücksichtigt hat. Jede Methode im Interface muss implementiert werden. Selbiges gilt auch für abstrakte Methoden in einer Klasse. Deswegen müssen an dieser Stelle die Sperrregeln etwas verschärft werden:

Regel 8:

Änderungen an einem Interface oder an abstrakten Methoden einer Klasse: Das Komposit-Interface bzw. die Komposit-Klasse benötigt eine exklusive Sperre.

Damit können neue Klassen von dem Interface erst ableiten, wenn die Änderungen am Interface abgeschlossen sind.

4.4.4 Namenskonflikt

Es gibt einen weiteren potentiellen Konflikt, der bisher noch nicht berücksichtigt wurde. In Java stellt ein Paket, eine Klasse, eine Methode oder ein Block (geschweifte Klammern) einen Namensraum bereit. Innerhalb des Namensraums müssen Namen eindeutig sein. Zum Beispiel ist es nicht möglich zwei Methoden mit demselben Namen innerhalb einer Klasse zu haben. Diese Methoden könnten nicht mehr voneinander unterschieden werden, weil sie über den Namen referenziert werden. Die geteilte Sperre auf der Klasse ermöglicht es, dass zwei Entwickler jeweils eine Methode mit dem gleichen Namen anlegen. Um das Problem zu lösen, gibt es zwei Möglichkeiten:

1. Für das Anlegen von Methoden innerhalb einer Klasse muss eine exklusive Sperre auf der Klasse verwendet werden. Dann kann nur ein Entwickler eine neue Methode innerhalb der Klasse anlegen. Damit wären die Möglichkeiten des parallelen Arbeitens eingeschränkter.
2. Es wird die Möglichkeit geschaffen, Namen innerhalb eines Namensraums zu reservieren. Diese Reservierung kann ähnlich gehandhabt werden wie die Sperren. Anstatt eines Artefakts wird ein vollqualifizierter Name in der Datenbank gesperrt. Bei einer Abgabe werden die Sperren automatisch wieder aufgehoben.

Namenskonflikte werden seltener auftreten als indirekte oder direkte Konflikte (zwei Entwickler müssten zufälligerweise den gleichen Namen in einem gemeinsamen Namensraum wählen). Außerdem kann der Konflikt sehr leicht gelöst werden, indem ein Artefakt umbenannt wird. Das könnte sogar automatisch durch eine Regel durchgeführt werden. Deswegen wurde keine der oben genannten Lösungen umgesetzt und dieser Konflikt wird nicht verhindert. Jedoch muss der Konflikt erkannt werden, damit keine Syntaxfehler im Modell auf dem Server entstehen. Deswegen wird eine Aktualisierung auf der Klasse unmittelbar vor dem Abgeben durch eine exklusive Sperre erzwungen. Diese Aktualisierung wird irgendwelche existierenden Namenskonflikte aufdecken, weil im Falle von Änderungen

innerhalb der Klasse durch die Aktualisierung die Klasse neu gebaut wird und Syntaxfehler den Konflikt anzeigen.

4.5 Integration in den Java AST Editor

Der Entwickler ist mit dem Sperren der AST Artefakte entsprechend der Sperrregeln überfordert. Deswegen ist eine Integration in den Java AST Editor notwendig. Dabei werden zwei Ziele verfolgt:

1. Anzeigen von Sperren anderer Benutzer im Java AST Editor (Awareness Enhancer): Der Entwickler kann erkennen welche Teile des Quellcodes durch andere Entwickler gerade verändert werden.
2. Automatische Sperren beim Editieren von Quellcode: Abhängig vom geänderten Quellcode sollen entsprechend den Sperrregeln automatisch die Sperren vom Server angefordert werden ohne Beteiligung des Softwareentwicklers.

4.5.1 Automatische Sperren

Der Java AST Editor kennt für jede Stelle im Quellcode die entsprechenden AST Artefakte (siehe auch Kapitel 3.5). Wenn der Softwareentwickler anfängt den Quellcode im Java AST Editor zu ändern, werden alle betroffenen AST Artefakte ermittelt und mit Hilfe der passenden Sperrregeln die notwendigen Sperren vom Server angefordert. Das wird sofort mit der ersten Tastatureingabe durchgeführt, weil der Softwareentwickler eventuell die Sperren nicht bekommt. Ein anderer Softwareentwickler könnte bereits denselben Quellcode geändert und entsprechende Sperren vom Server bezogen haben, von denen der Client noch nichts weiß. In diesem Fall sollte der Entwickler nicht in der Lage sein, seine Änderungen durchzuführen, ansonsten müsste der Quellcode verschmolzen werden.

Im Editor werden die Textbereiche, die nicht geändert werden können wegen fremden Sperren mit dunkelgrauem Hintergrund dargestellt. In diesen Zeilen wird keine Tastatureingabe des Entwicklers akzeptiert. Der Entwickler kann jedoch auf diesen Zeilen ein Popup-Fenster öffnen, in dem die folgenden Informationen angezeigt werden: Art der Sperre (exklusive oder geteilt), Besitzer der Sperre und seit wann die Sperre existiert (siehe auch Abbildung 68).

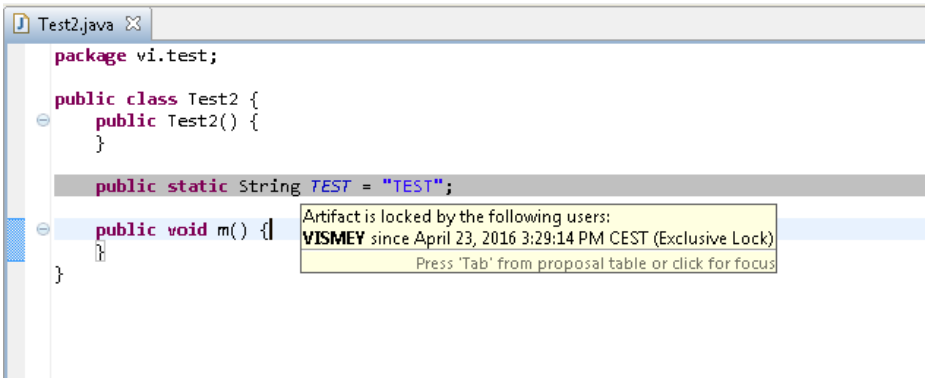


Abbildung 68: Java AST Editor mit Informationen über die fremde Sperre

4.5.2 Sperren für indirekte Konflikte

Das automatische Sperren für die indirekten Konflikte stellt sich etwas komplexer dar. Auch hier kann die Sperre nicht zu einem späteren Zeitpunkt zum Beispiel mit dem Speichern im Editor angefordert werden, sondern muss sofort, wenn sie relevant wird, akquiriert werden. Sobald der Entwickler ein neues Client-Artefakt im Quellcode hinzugefügt hat, muss das entsprechende Supplier-Artefakt gesperrt werden. Wenn diese Sperre nicht angefordert werden kann, muss der Entwickler sofort informiert werden, bevor er fortfährt Änderungen im Quellcode durchzuführen.

Die notwendige Funktionalität soll an Hand eines Beispiels erklärt werden. Der Softwareentwickler möchte einen neuen Methodenaufruf in seinem Quellcode hinzufügen. Er beginnt mit dem Editieren und Eclipse wird sofort mit der ersten Tastatureingabe einen Syntaxfehler anzeigen, weil Eclipse mit jeder Änderung im Quellcode versucht den Text zu kompilieren. Damit bekommt der Entwickler sehr schnell Rückmeldung über eventuell vorhandene Syntaxfehler für jede Zeile im Quellcode. Dabei wird auch immer ein AST von Eclipse erstellt, auch wenn der Quellcode noch Syntaxfehler beinhaltet. Mit der Eingabe der ersten Zeichen ist noch kein Quellcode ohne Syntaxfehler vorhanden. Wenn der Entwickler den Methodenaufruf in der Quellcode-Zeile vollständig und syntaxfehlerfrei eingegeben hat, verschwindet die Fehleranzeige für diese Zeile. Das wiederum triggert eine Überprüfung ob zusätzliche geteilte Sperren notwendig sind. Es wird überprüft ob Client-Artefakte in der Zeile vorhanden sind und wenn ja, welche Supplier-Artefakte betroffen sind. Die Frage ist nun, ob das neue Assoziationen sind oder ob es die bereits zuvor gegeben hat und daher keine neuen Sperren notwendig sind. Das wird an Hand des aktuellen Modells im Client überprüft, das ja noch nicht die Änderungen aus dem Quellcode Editor beinhaltet. Diese werden erst mit dem Speichern des Editorinhalts in das Modell verschmolzen. Wenn von den eigenen

gesperrten und nicht neuen AST Artefakten im Modell bereits eine Assoziation zu dem Supplier-Artefakt gibt, ist keine Sperre notwendig. Wenn das nicht der Fall ist, wird eine geteilte Sperre vom Server angefragt.

4.5.3 Updates im Editor

Mit jeder angeforderten Sperre kommt auch ein Update mit, um sicherzustellen, dass die Änderungen auf dem neuesten Modellstand durchgeführt werden (siehe auch Kapitel 2.6.6). Wenn der Entwickler nach der Änderung von Quellcode im Editor eine neue Sperre anfordert (zum Beispiel, weil er eine zusätzliche Methode in der Klasse anfängt zu editieren), dann kann ein Update des Quellcodes im Editor notwendig werden. Ein anderer Entwickler könnte bereits Änderungen in der Klasse an einer anderen Stelle, die nicht vom Entwickler gesperrt war, vorgenommen und abgegeben haben. In diesem Fall muss der Java AST Editor den geänderten Quellcode des Entwicklers mit dem Quellcode vom Server verschmelzen und den neuen Quellcode-Text im Editor aktualisieren. Auch die Zuordnung von AST Artefakt zur Position und Länge im Quellcode-Text muss verschmolzen werden. Dieses Verschmelzen wird ohne Konflikte möglich sein, weil über die Sperren ein konfliktfreies Verschmelzen sichergestellt ist.

Diese Funktionalität im Editor ist für den Softwareentwickler ungewohnt. Im schlimmsten Fall fängt der Entwickler an eine Methode zu editieren, die plötzlich verschwindet, weil ein anderer Entwickler sie inzwischen gelöscht hat.

5 Continuous Integration

5.1 Einführung

Im nächsten Schritt soll nun der vom Softwareentwickler geänderte Quellcode verifiziert und validiert werden (siehe auch Kapitel 1.2.3) [15] [93]. Für die Validierung werden in regelmäßigen Abständen sehr stabile Produktversionen benötigt mit den Änderungen der Entwickler. Die Verifizierung erfolgt durch automatische Tests. Da der Aufwand für die Behebung des Softwarefehlers zunimmt je später der Softwarefehler gefunden und behoben wird, soll der Softwareentwickler unmittelbar über das Ergebnis der automatischen Tests informiert werden. Nach dem der Softwareentwickler seinen Quellcode in das Software Configuration Management Repository (SCM) abgegeben hat, soll der Entwickler innerhalb kurzer Zeit automatisch mittels eines Email Reports über die Auswirkungen seiner Änderungen informiert werden.

Continuous Integration (CI) ist ein Schritt Richtung Lösung der obengenannten Probleme (siehe auch Kapitel 2.7). Abhängig von der Größe des Softwareprodukts und der Anzahl der auszuführenden Tests, kann ein kompletter CI Lauf einige Zeit dauern. Ohne irgendwelche Optimierungen dauert für große Softwareprodukte das auschecken des gesamten Quellcodes aus dem SCM Repository und das Übersetzen des Quellcodes mehrere Stunden [94] und die Ausführung aller automatischen Tests mehrere Tage oder Wochen [95]. Die Zeiten lassen sich insbesondere bei der Ausführung der Tests durch entsprechend viele CI Server reduzieren, aber um die Ausführungszeiten von Wochen auf Minuten zu reduzieren, wäre dann der nötige Hardware Einsatz und die damit verbundenen Kosten doch zu groß. Daher soll über entsprechende Optimierungen die Übersetzungszeiten des Quellcodes und Ausführungszeiten der Tests minimiert werden.

5.2 Stand der Technik

Im Folgenden werden die bereits existierenden Lösungsansätze dargestellt.

5.2.1 Unterschiedliche Konfigurationen des CI

In großen Softwareprojekten wäre es unmöglich mit jeder Abgabe von Quellcode den kompletten CI Lauf mit allen automatischen Tests durchzuführen. Die Anzahl der benötigten

CI Server wäre viel zu groß und die Wartezeiten viel zu lange. Daher gibt es folgende Möglichkeiten:

1. Es werden alle automatischen Tests über Nacht ausgeführt mit den Änderungen aller Softwareentwickler eines ganzen Tages. Dazu sind unter Umständen auch mehrere CI Server notwendig. Der Nachteil dieser Lösung ist folgender: Es ist nicht mehr feststellbar, welche Quellcode Änderung welchen fehlgeschlagenen Test verursacht hat, da die Tests mit den Änderungen aller Entwickler ausgeführt wurden. Die Verantwortlichkeiten sind auch unklar: Wer untersucht welchen Test und behebt das jeweilige Problem? Der Softwareentwickler muss proaktiv am nächsten Tag die Ergebnisse der Tests sichten ohne genau zu wissen, welche Tests relevant für ihn sind. Außerdem muss der Entwickler einen ganzen Tag warten bevor er irgendwelche Ergebnisse bekommt.
2. Die zweite Möglichkeit ist die Verwendung einer Build Pipeline [18]. Der Abgabe-Build wird ausgeführt, wenn jemand Quellcode in das SCM Repository abgibt. Während dieses Builds werden nur ein paar wenige Regressionstests oder nur schnell laufende Unittests mit kurzen Ausführungszeiten gestartet. Der Sekundär -Build wird nicht mit jeder Abgabe vom Quellcode getriggert, sondern in gewissen Zeitintervallen und es werden mehr Tests mit längeren Ausführungszeiten gestartet. Der Nachteil des Abgabe-Builds ist, dass nicht alle relevanten Tests ausgeführt werden, im Besonderen nicht länger laufende Systemtests. Der Sekundär-Build ist ähnlich zu der Lösung mit dem CI Build über Nacht und hat die gleichen Nachtteile.

5.2.2 Beschleunigter Build

In diesem Kapitel werden die bisherigen Ansätze, um die Build-Zeiten zu reduzieren, erörtert.

5.2.2.1 Beschreibung

Paralleler Build

In der von E. H. Baalbergen [96] vorgeschlagenen Lösung wird mit Hilfe des Build-Management-Tools „make“ das Kompilieren der C-Dateien in einer Multiprozessor Umgebung auf mehrere Prozessoren verteilt. Der Link Phase kann in weiter Folge nicht weiter parallelisiert werden.

Verteilter Build

Mit Hilfe des verteilten Compilers „distcc“ [97] ist es möglich das Kompilieren von C und C++ Quellcode auf unterschiedliche Rechner in einem gemeinsamen Netzwerk zu verteilen. Dazu wird ein Serverprogram auf die Rechner installiert, dass von einem Client Programm die Kompilieraufträge entgegennimmt und nach Beendigung das Ergebnis an den Client zurückgibt.

Inkrementeller Build

Die Idee des inkrementellen Builds ist, dass nur die tatsächlich geänderten Komponenten und deren abhängigen Komponenten neu gebaut werden. Damit kann die Build Zeiten deutlich reduziert werden. Maven [98] ist ein Build-Management-Tool, das einen solchen inkrementellen Build unterstützt und wird besonders im Zusammenhang mit Java Programmen verwendet. Maven erlaubt die Definition eines Projekt Objekt Models mit den Abhängigkeiten zwischen den Projekten. Wenn der Quellcode eines Projekts verändert wird, werden nur das Projekt selber und alle direkt und indirekt abhängigen Projekte neugebaut.

Van der Storm [99] stellt ein Konzept vor für die Unterstützung des inkrementellen Build innerhalb von CI. Zu einem bestimmten Zeitpunkt wird ein Abzug des Quellcode Repository erstellt und daraus ein Source Tree mit den aktuellen Abhängigkeiten ermittelt. Zu Beginn werden alle Komponenten erstmalig gebaut. Bei der Änderung einer Komponente werden mit Hilfe des Source Trees die Abhängigkeiten untersucht und die Komponenten, die nicht neu gebaut werden müssen, werden aus dem letzten Build wiederverwendet. Wenn eine Komponente wegen eines Kompilierfehlers nicht neugebaut werden kann, wird bei weiteren Änderungen versucht mit den bereits gebauten Komponenten trotzdem ein Produkt zu erstellen unter der Annahme ein Build ist besser als gar kein Build.

Caching Build

Bei diesem Ansatz werden die übersetzten Source Code Dateien (Binaries) in einem Cache gespeichert und für den Produkt-Build wiederverwendet. Der „CompilerCache“ für C/C++ Compiler verwendet dieses Konzept [100]. Compilercache ist ein Skript, das nach dem Übersetzen das Ergebnis in einen lokalen Cache ablegt. Wenn die Datei nochmal übersetzt wird, wird das Ergebnis aus dem Cache geholt. Der Vorteil gegenüber dem Build-Management-Tool „make“ ist, dass eine Makedatei mit den Definitionen der Abhängigkeiten nicht gepflegt werden muss. Außerdem können Compiler Einstellung hin und her gewechselt werden, ohne dass der Quellcode neu übersetzt werden muss.

5.2.2.2 Abgrenzung

Trotz der Build Optimierungen müssen alle geänderten Komponenten und direkt und indirekt abhängigen Komponenten neu gebaut werden. Im schlimmsten Fall müssen alle Komponenten neugebaut werden, wenn Projekte mit vielen abhängigen Projekten geändert wurden. Deswegen kann der Build immer noch erhebliche Zeit in Anspruch nehmen während eines CI Laufs und möglicherweise haben sich nur ein paar wenige Binaries im Produkt verändert. Ziel ist es die Zeit bis zum Feedback so kurz wie möglich zu halten und so viel Zeit wie möglich für die Ausführung der Tests zu haben.

Der Caching Build ist nur für C/C++ geeignet und beachtet nicht die Besonderheiten von Java. Außerdem ist keine Integration in Eclipse vorhanden. Der Caching Build wäre auch nur brauchbar, wenn die gecachten Binaries dem CI Build zur Verfügung stehen würden und auf einem Server abgelegt wären.

5.2.3 Continuous Testing

Continuous Testing (CT) [101] gibt schnelles Feedback über die Qualität des Quellcodes indem automatisch Regressionstests im Hintergrund ausgeführt werden, während der Softwareentwickler den Quellcode abändert. Es gibt einige kommerzielle Produkte am Markt. Stellvertretend wird NCrunch [102] in diesem Kapitel vorgestellt.

5.2.3.1 Beschreibung

NCrunch speichert während der Ausführung der automatischen Tests die Code Abdeckung und weiß daher welche Codestellen der Test durchläuft. Daher können beim Ändern des Quellcodes die passenden Tests automatisch im Hintergrund ausgeführt werden. Das Ergebnis der Tests wird unmittelbar im Editor für jede Codezeile über verschiedene Farben angezeigt: Weiß bedeutet, dass es überhaupt keine Tests für diese Zeile gibt. Rot zeigt einen fehlgeschlagenen Test an und Grün, dass alle Tests fehlerfrei durchgelaufen sind (siehe Abbildung 69). NCrunch unterstützt auch einen stark optimierten Build für den geänderten Quellcode, um eine schnelle Testausführung zu ermöglichen. Der Anwender kann entscheiden wieviel Prozessoren vom lokalen Rechner NCrunch nutzen darf. Alternativ um den lokalen Rechner nicht zu sehr mit der Testausführung zu belasten, kann NCrunch auch die Testausführung auf andere Server auslagern. Außerdem kann der Anwender festlegen welche Tests nur manuell ausgeführt werden sollen, weil zum Beispiel die Ausführung zu viel Zeit in Anspruch nehmen würde.

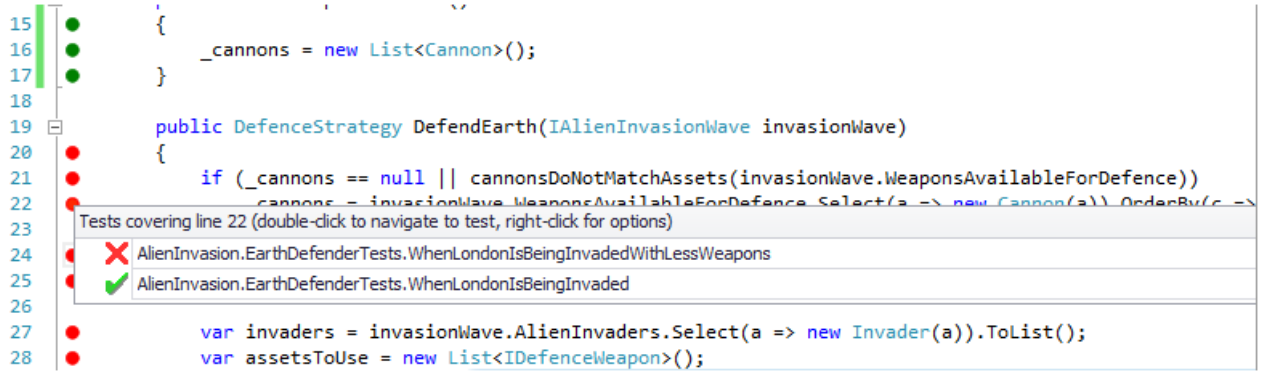


Abbildung 69: Anzeige der Testergebnisse im Editor über NCrunch [102]

5.2.3.2 Abgrenzung

Die in dieser Arbeit vorgestellte Lösung ist kein Ersatz für Continuous Testing (CT) sondern CT ist eine sinnvolle Ergänzung. CT ist besonders gut für Unittests und Komponententests geeignet zum Beispiel für das Testen einer Methode oder einer Klasse. Der Entwickler kann lokale Probleme entdecken zum Beispiel, das eine Methode nicht mehr so funktioniert wie zuvor und kann dann die Methode oder den Test korrigieren. Obwohl NCrunch auch die Ausführung von Integrations- und Systemtests unterstützt, scheint das weniger sinnvoll, weil die Wahrscheinlichkeit, dass diese Tests fehlschlagen, sehr hoch ist solange der Entwickler seine gesamten Änderungen noch nicht abgeschlossen hat. CT liefert Testresultate für die gerade geänderten Codezeilen und nicht unbedingt über alle Änderungen hinweg. Außerdem werden Quellcodeänderungen anderer Entwickler, die bereits im SCM Repository abgegeben sind, nicht berücksichtig. Die Tests werden auch nicht mit einer definierten Produktversion in einer Umgebung, die der Produktivumgebung möglichst nahekommt, ausgeführt. Für die Ausführung von Systemtests kann es notwendig sein, dass eine Datenbank und ein Server aufgesetzt werden müssen. Das wird alles innerhalb des CI Builds mit dem aktuellen Datenbankschema und dem aktuellen im Repository enthaltenen Middle-Tier und Client Quellcode durchgeführt. Durch die im CI Build ausgeführten Tests kann zu einer ganz bestimmten Produktversion eine Aussage bezüglich der Qualität gemacht werden.

5.2.4 Test Case Priorisierung

Test Case Priorisierung (TCP) hat bereits sehr viel Aufmerksamkeit in der Forschungsgemeinschaft [103], [95], [104], [105], [106] bekommen, weil es die Kosten für Regressionstest deutlich senken kann. Die meisten der vorgeschlagenen Techniken basieren auf Code Abdeckung von automatischen Tests. PORT [103] umfasst noch weitere Aspekte und wird in diesem Kapitel stellvertretend vorgestellt.

5.2.4.1 Beschreibung

Über die PORT Methodik werden die Prioritäten für die auszuführenden Testfälle berechnet unter Berücksichtigung der vom Kunden vergebenen Prioritäten der Anforderungen, der Volatilität der Anforderungen (Wie oft haben sich die Anforderungen geändert?) und die Fehler Anfälligkeit der Anforderungen (Wie oft wurden Fehler zu den Anforderungen vom Kunden zurückgemeldet?) und die Komplexität der Implementierung. Die notwendigen Informationen werden folgendermaßen erfasst: Der Kunde definiert und ändert die Anforderungen und vergibt Prioritäten. Außerdem berichtet der Kunde Fehler aus dem Betrieb. Der Requirement Analyst dokumentiert die Anforderungen und die Änderungen der Anforderungen. Der Maintenance Engineer behebt die Fehler und verlinkt die Fehler mit der dazugehörigen Anforderung. Der Developer liefert Informationen über die Komplexität der Implementierung einer Anforderung. Der Tester schreibt Testfälle, verlinkt die Testfälle mit den Anforderungen und führt die Testfälle aus (siehe Abbildung 70).

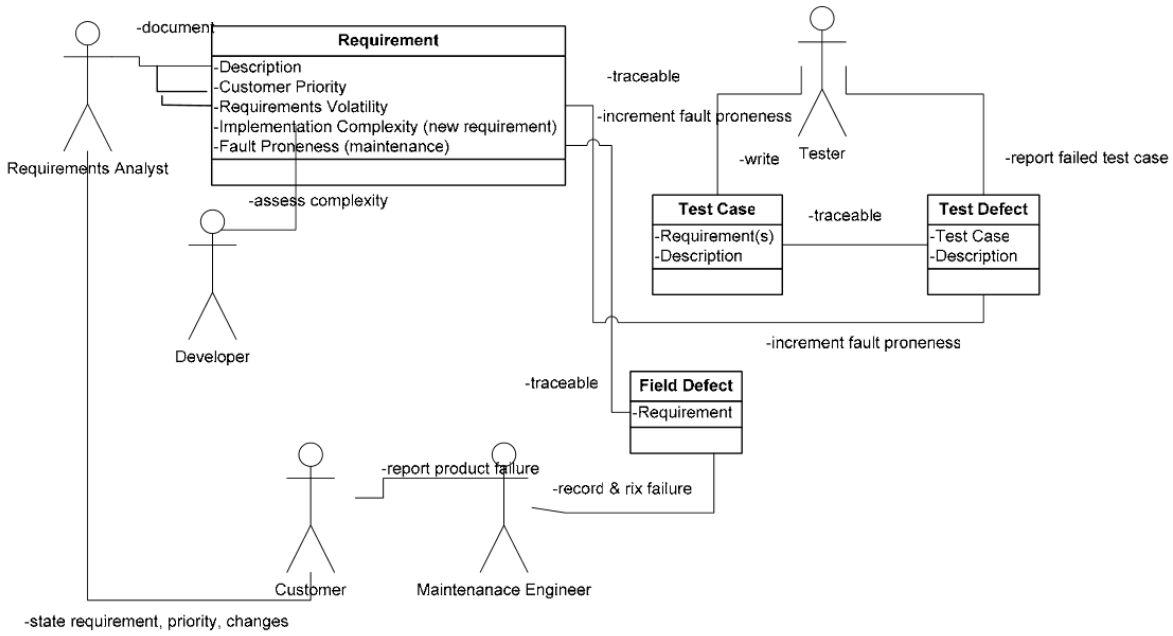


Abbildung 70: PORT Erfassungsprozess der Faktoren für die Berechnung der Prioritäten [103]

Mit diesen Informationen werden über den PORT Algorithmus die Prioritäten der Testfälle berechnet.

5.2.4.2 Abgrenzung

Für PORT gibt es keine Integration in den CI Prozess. Dazu fehlt auch die Möglichkeit aus den aktuellen Quellcodeänderungen die Prioritäten der Testfälle zu berechnen. Um das tun zu können, müsste es verlässliche Links in den Quellcode hinein geben. Der PORT Algorithmus

könnte aber durchaus in Morpheus zur Anwendung kommen für eine verbesserte Priorisierung der Testfälle. Die meisten der benötigten Daten müssten noch nicht einmal manuell erfasst werden, sondern könnten direkt aus dem Modell ermittelt werden.

5.3 Konzeptidee und Alleinstellungsmerkmale

Folgende Konzeptideen wurden für Morpheus umgesetzt:

1. Reduzierung der Feedback Zeit für den Softwareentwickler
    a. Minimieren der Build Zeit.
    b. Verkürzen der Testausführungszeit durch eine passende Auswahl der Testfälle. Zur Bestimmungen der Testfälle sollen die verfügbaren Informationen im Modell genutzt werden. Dazu werden die Traceability Links verwendet, um verschiedene Informationen zu verknüpfen.
2. Verbessern der Qualität des Feedbacks mit den Testergebnissen
    a. Ausführen der automatischen Tests mit einer definierten Produktversion in einer Umgebung, die ähnlich zur der Umgebung des Kunden ist.
    b. Personalisierte Bereitstellung von Testergebnissen nur für den Quellcode, den der Entwickler geändert hat.
    c. Automatisches Feedback mit der Information ob ein Test fehlgeschlagen ist wegen der letzten Änderung des Entwicklers oder wegen bereits zuvor getätigten Änderungen.

Die Lösung wurde auf Basis von Continuous Integration mit der Ausführung von automatischen Tests entwickelt. Die Konfiguration des CI sieht dabei folgendermaßen aus: Mit jeder Abgabe von Quellcode in das SCM Repository wird ein eigener CI Lauf gestartet. Das stellt sicher, dass alle Testergebnisse nur zu den Änderungen eines Softwareentwicklers gehören. Die automatischen Tests werden mit einem Produkt-Build ausgeführt in einer Umgebung, die möglichst ähnlich zu der Umgebung bei den Kunden ist. Die Ausführung der Tests auf dem Rechner des Softwareentwicklers in der Softwareentwicklungsumgebung ist nicht verlässlich genug um eine Aussage über die Qualität der Software treffen zu können.

Der beschleunigte Produkt-Build wird durch die Speicherung der Binary-Dateien im SCM Repository erreicht und die Reduzierung der Testausführungszeit erfolgt durch eine Selektion von automatischen Tests, die relevant sind für den geänderten Quellcode. Nach dem CI Lauf bekommt der Entwickler eine Benachrichtigung mit einem Report über die Testläufe und

deren Ergebnisse. Der Entwickler kann dann die Softwarefehler beheben, bevor mehr Probleme verursacht werden und er sich neuen Aufgaben zuwendet. Die implementierte Lösung wird in Teilen[5] derzeit produktiv von einem Team von 30 Entwicklern für die Entwicklung einer kommerziellen 3-Schichten Anwendung basierend auf Eclipse mit mehr als 1200 Plugins und etwa 4 Millionen Quellcode-Zeilen eingesetzt.

Damit lassen sich folgende Produktmerkmale bezüglich eines verbesserten Continuous Integration für die Darstellung der Alleinstellungsmerkmale definieren:

A) Schneller Build mit Binary Cache

Es werden binäre Dateien in einem Cache abgelegt, um den lokalen Build zu beschleunigen.

B) Schneller Build mit Java Class Dateien

Es wird ein schneller Produktbuild ermöglicht unter Verwendung von Class-Dateien, die mit dem Quellcode in einem Repository abgespeichert werden. Diese Class-Dateien stehen auch dem CI Build zur Verfügung.

C) Automatischer Test

Es werden Tests automatisch für den geänderten Quellcode ausgeführt. Es erfolgt eine automatische Testauswahl und eine schnelle Rückmeldung an den Entwickler.

D) Test Case Priorisierung

Es erfolgt eine Testauswahl um die Anzahl der Tests und damit die Testausführungszeit zu verkürzen.

E) Test Case Priorisierung mit Traceability Links

Die Tests für eine automatische Testausführung werden auf Grund von Traceability Informationen zwischen geänderten Quellcode, Anforderungen, Testspezifikationen, Testimplementierungen und Test-Quellcode ausgewählt.

[5] Der komplette Abstract Syntax Tree wird noch nicht produktiv im PREEvision Repository abgespeichert (siehe auch Kapitel 6).

	Morpheus	CompilerCache für C/C++	PORT	NCrunch
A) Schneller Build mit Binary Cache	✓	✓		
B) Schneller Build mit Class-Dateien	✓			
C) Automatischer Test	✓			✓
D) Test Case Priorisierung	✓		✓	
E) Test Case Priorisierung mit Traceability Links	✓			

Tabelle 3: Alleinstellungsmerkmale bezüglich Continuous Integration

5.4 Beschleunigter Build

Eclipse unterstützt das Feature „Continuous Build“. Das bedeutet, dass Eclipse bei jedem Speichern einer Quellcode Datei einen Build antriggert[6]. Während des Builds übersetzt ein Java Compiler die Java Source Datei in eine Class Datei (Binär Datei). Damit ist der Softwareentwickler immer in der Lage die gerade entwickelte Anwendung zu starten, ohne vorher einen Build anstoßen zu müssen (zum Beispiel zum Debuggen der Anwendung). Wenn der Entwickler die lokalen Quellcode Dateien aus dem SCM Repository aktualisiert um Änderungen anderer Entwickler zu erhalten, werden alle veränderten Quellcode Dateien sofort neu gebaut.

Nachdem der Softwareentwickler den geänderten Quellcode in das SCM Repository abgegeben hat, wird der Quellcode während CI wieder übersetzt und nochmals auf jeden Rechner jedes Softwareentwicklers, wenn sie ihren Quellcode aktualisieren. Es gibt eigentlich keinen Grund den Quellcode immer wieder von neuem zu kompilieren. Stattdessen könnten die Class Dateien mit dem Quellcode in das SCM Repository abgegeben werden. Dann können die Class Dateien aus dem Repository dazu verwendet werden, um den Produkt-Build vor der Ausführung der automatischen Tests zu aktualisieren. Alle anderen Entwickler können auch mit der Aktualisierung ihres Quellcodes die Class Dateien übernehmen und

[6] Wenn die Option „Build Automatically“ in der Eclipse Entwicklungsumgebung aktiv ist.

damit den lokalen Build verhindern. Das Abgeben und Aktualisieren der Class Dateien verursacht keine zusätzlichen Aufwände für den Entwickler.

Die folgenden Erweiterungen sind notwendig um von der Speicherung der Class Dateien im SCM Repository zu profitieren für einen beschleunigten Produkt-Build und für die Vermeidung der lokalen Builds in der Softwareentwicklungsumgebung.

5.4.1 Eclipse Integration

Der Softwareentwickler kann sehr komfortable innerhalb der Eclipse Integrated Development Environment (IDE) das Abgeben und das Aktualisieren von Quellcode anstoßen. Dabei müssen jedoch ein paar Besonderheiten von Java beachtet werden und Morpheus muss entsprechend erweitert werden. Außerdem muss an einigen Stellen die Funktionalität innerhalb der Eclipse IDE angepasst werden.

5.4.1.1 Abgabe von Class Dateien und abhängigen Class Dateien

Morpheus muss während der Abgabe von Quellcode in das SCM Repository sicherstellen, dass alle dazugehörigen Class Dateien mit abgegeben werden. Der Entwickler sollte nicht verantwortlich sein, die richtigen Class Dateien für die Abgabe auswählen zu müssen. Wenn nicht alle oder falsche Class Dateien abgegeben werden, dann ist der Quellcode mit den Class Dateien nicht mehr konsistent innerhalb des SCM Repository.

In einigen Situationen erzeugt der Compiler eine geänderte Class Datei obwohl die dazugehörige Quellcode Datei überhaupt nicht verändert worden ist. Zum Beispiel wird in der Quellcode Datei „Const.java“ folgender konstanter Wert definiert:

```
public static final int LOCKED = 122;
```

Dieser Wert wird in einer zweiten Quellcode Datei „Class.java“ verwendet:

```
int errorCode = Const.LOCKED;
```

Nehmen wir an der Wert wird vom Entwickler von `122` nach `123` geändert, dann ändert der Compiler beide Class Dateien ab, obwohl nur eine Quellcode Datei „Const.java“ geändert wurde. Der Grund ist, dass der Wert `123` direkt in die Class Datei „Class.class“ inkludiert wird (siehe Abbildung 71). Über den Abstract Syntax Tree (AST) sind die Abhängigkeiten zwischen den Quellcode Dateien bekannt. Wenn der Softwareentwickler eine Quellcode Datei für die Abgabe auswählt, dann sucht Morpheus nach allen abhängigen und geänderten Class Dateien und fügt diese Dateien automatisch der Abgabe-Menge hinzu. Ein anderes Beispiel

ist die Änderung des Datentyps für den Parameter einer Methode. Wenn der Methodenparameter von „int“ nach „double“ geändert wird, muss die Quellcode Datei mit dem Aufruf der Methode nicht angepasst werden, aber die dazugehörende Class Datei würde sich ändern und muss bei der Abgabe mitberücksichtigt werden.

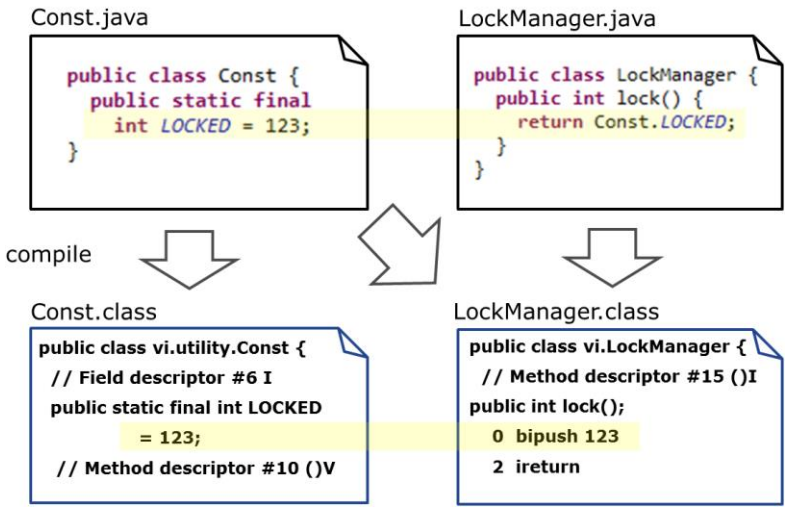


Abbildung 71: Quellcode mit den abhängigen Class Dateien

Die Situation verkompliziert sich, wenn die Quellcode Dateien der automatisch hinzugefügten Class Dateien sich auch geändert haben, aber vom Entwickler nicht für das Abgeben ausgewählt wurden. Manchmal arbeiten die Entwickler parallel an zwei oder mehr Aufgaben zum Beispiel an zwei unterschiedlichen Softwarefehlern. Der Entwickler möchte aber nur die Änderungen für die Behebung eines Fehlers abgeben, weil die andere Änderung noch nicht abgeschlossen ist. So kann eine Class Datei die Änderungen für zwei unterschiedliche Softwarefehler enthalten, einmal über eine direkte Änderung der Quellcode Datei und einmal über eine indirekte Änderung wie oben beschrieben. In diesem Fall wird der Entwickler auf die Konfliktsituation hingewiesen und der Entwickler muss die Abgabe-Menge erweitern um die Abgabe durchführen zu können. In dem obigen Beispiel kann sowohl die Quellcode Datei „Const.java“ nicht ohne die Quellcode Datei „LockManager.java“ abgegeben werden als auch „Const.java“ nicht ohne „LockManager.java“. In beiden Fällen wäre sonst der Quellcode nicht mehr mit den Class Dateien konsistent.

Eine einfache Abhängigkeitsbetrachtung (die Datei „LockManager.java“ ist von der Klasse „Const.java“ abhängig) reicht jedoch nicht aus für eine Entscheidung, ob ein Konflikt vorliegt oder nicht. So kann es zum Beispiel sein, dass der Entwickler überhaupt nicht den konstanten Wert „Locked“ in der Quellcode Datei „Const.java“ verändert hat, sondern irgendeine andere

Änderung durchgeführt hat. Wenn der konstante Wert „Locked“ die einzige Abhängigkeit zwischen den beiden Dateien ist, dann ist in diesem Fall sehr wohl eine getrennte Abgabe der beiden Änderungen möglich. Nur wenn die Änderungen in der einen Quellcode Datei wirklich eine Änderung in der anderen Class Datei verursacht, ist eine gemeinsame Abgabe notwendig. Deswegen werden von Morpheus die AST Artefakte, die eine Abhängigkeit zwischen den Quellcode Dateien verursachen, dahingehend untersucht, ob sie verändert worden sind oder nicht.

5.4.1.2 Verhindern eines Build nach dem Update

Quellcode und Class Dateien werden bei der Aktualisierung aus dem SCM Repository geladen und so sind für jede Quellcode Datei die entsprechenden Class Dateien sofort verfügbar. Es gibt daher keinen Grund einen Build zu starten und Morpheus unterdrückt den Build von Eclipse mit einer Ausnahme: Wenn eine Quellcode Datei bereits vom Entwickler lokal geändert wurde und über die Aktualisierung Änderungen in dieselbe Quellcode Datei integriert werden, dann treten in den Class Dateien Konflikte auf. Die Class Datei wurde lokal schon geändert und es kommt eine neue geänderte Version der Datei vom SCM Repository. Beide Änderungen in der Class Datei zu verschmelzen wäre sehr kompliziert und aufwändig. Deswegen löst Morpheus den Konflikt in dem zuerst die Class Datei aus dem SCM Repository übernommen wird und dann das Plugin, zu dem die Datei gehört in einer Liste gespeichert wird. Am Ende der Aktualisierung werden für alle Plugins, die in der Liste gespeichert sind, der Build nicht unterdrückt und der Compiler erzeugt neue Class Dateien von der Quellcode Datei mit den verschmolzenen Änderungen.

Wenn der Entwickler eine Aktualisierung seines Quellcodes durchführen will, müssen alle Plugins aktualisiert werden, auf Grund der oben erwähnten potenziellen Abhängigkeiten zwischen den Class Dateien. So wird sichergestellt, dass alle Quellcode Dateien und Class Dateien konsistent sind. Das ist auch anders nicht möglich, weil das Quellcode Modell mit dem Abstract Syntax Tree (AST) nicht in Teilen aktualisiert werden kann.

5.4.1.3 Verhindern eines Updates oder Abgabe während des Builds

Ein gerade ausgeführter Build bedeutet, dass noch nicht alle Quellcode Dateien kompiliert worden sind und einige Class Dateien noch nicht mit dem Quellcode konsistent sind. Deswegen muss Morpheus während eines laufenden Builds die Abgabe von Quellcode unterbinden, weil sonst veraltete Class Dateien abgegeben werden würden. Morpheus stellt

auch sicher das die Option „Build Automatically“ (Continuous Build) in der Eclipse IDE vor der Abgabe aktiv ist.

Wenn der Entwickler versucht während eines Builds eine Aktualisierung des Quellcodes durchzuführen, wird der Entwickler von Morpheus über einen Dialog darauf hingewiesen und der Entwickler kann entscheiden, ob er die Aktualisierung trotzdem fortführen will oder auf das Ende des Builds warten möchte. Im Falle der Fortsetzung der Aktualisierung, wird der Build nach der Aktualisierung von Morpheus nicht unterdrückt und die Class Dateien werden über den Build neu generiert. Wenn der Entwickler auf das Ende des Builds wartet, muss nach der anschließenden Aktualisierung kein weiterer Build ausgeführt werden. So werden mögliche Konflikte, die durch die gleichzeitige Aktualisierung der Class Dateien aus dem SCM Repository und der Neuerstellung durch den Compiler verhindert.

5.4.2 Erzeugen eines Produkt Builds

Für ein schnelles Feedback während CI muss in sehr kurzer Zeit ein Produkt-Build erstellt werden. Das wird durch die Verwendung der Class Dateien aus SCM Repository erreicht. Die Class Dateien können nur verwendet werden, um einen bestehenden Produkt-Build zu aktualisieren. So wird zu Anfang über den normalen automatisierten Build ein Produkt-Build erstellt.

Wenn der Entwickler Quellcode und Class Dateien in das SCM Repository abgibt, dann wird Morpheus vom SCM getriggert einen Produkt-Build zu erstellen (siehe Abbildung 72). Dazu lädt Morpheus vom SCM Repository ein Change Set, das Informationen über alle abgegebenen Dateien seit der letzten Aktualisierung des Produkt-Builds enthält und analysiert das Change Set. Es identifiziert die von Änderungen betroffenen Plugins und sucht die Plugins im Produkt-Build. Danach werden die geänderten Class Dateien vom SCM Repository geladen und Morpheus ersetzt die Class Dateien im Produkt Build mit den Class Dateien vom SCM Repository. Wenn das Plugin ein Java Archiv (Jar) ist, muss das Plugin zuerst ausgepackt werden bevor die Class Dateien ersetzt werden können. Danach kann der aktualisierte Produkt-Build für die Ausführung der automatischen Tests verwendet werden.

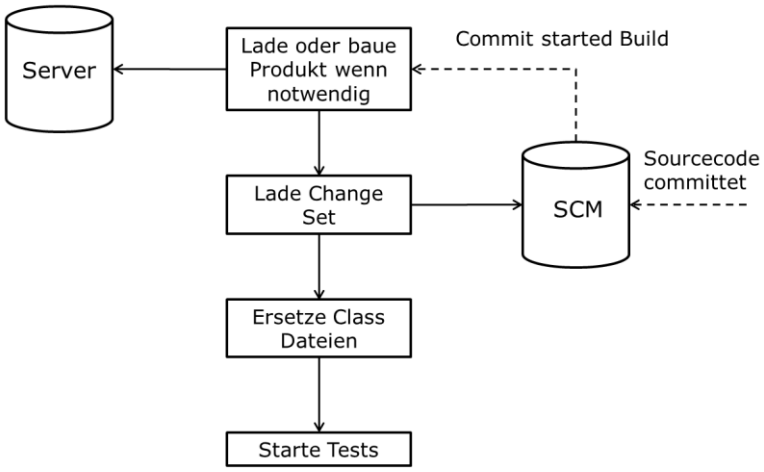


Abbildung 72: Erstellung des Produkt-Builds

In einigen Fällen wird Morpheus feststellen, dass eine Aktualisierung des Produkt-Builds nicht möglich ist, zum Beispiel, wenn neue Plugins angelegt wurden oder ein bestehendes Plugin gelöscht wurde. Dann muss der Produkt-Build von Grund auf neu erstellt werden und die Ausführung der Tests verzögert sich. Aber in den meisten Fällen enthält das Change Set nur geänderten Quellcode und geänderte Class Dateien und der beschleunigte Build kann verwendet werden.

Ein wichtiger Grund für CI, neben der Ausführung von automatischen Tests, ist das Finden von Syntaxfehlern, die durch das gleichzeitige Abgeben von Quellcode mehrerer Softwareentwickler entstehen können (indirekter Konflikt). Mit dem beschleunigten Build wird jedoch der Compiler zum Übersetzen des Quellcodes überhaupt nicht mehr verwendet und es können daher auch keine Syntaxfehler mehr gefunden werden. Durch die Verwendung des pessimistischen Lockens auf AST Ebene werden Syntaxfehler im SCM Repository jedoch von vornherein ausgeschlossen (siehe auch Kapitel 4). Daher stellt das kein Problem mehr da. Da es keine Syntaxfehler mehr geben kann, ist auch sichergestellt, dass die automatischen Tests innerhalb des CI auch mit jeder Abgabe von Quellcode ausgeführt werden können.

5.4.3 Ergebnisse

Der oben beschriebene Lösungsansatz wurde für die Entwicklung einer Anwendung basierend auf Eclipse mit über 1000 Plugins und über 4 Millionen Quellcode-Zeilen eingesetzt. Ein typischer Entwicklerrechner hat einen 4 x 3.6 GHz Prozessor und 64 GB Arbeitsspeicher.

Ohne irgendwelche Optimierungen variiert die Aktualisierung des Quellcodes innerhalb der Eclipse IDE sehr stark, abhängig davon wie viele Plugins und welche Plugins geändert

wurden. Wenn von der Aktualisierung einige Plugins mit sehr vielen abhängigen Plugins betroffen sind, dann kann der lokale Build bis zu ½ Stunde[7] dauern, weil Eclipse sehr viele abhängige Plugins neu bauen muss. Der Eclipse IDE Build ist ein inkrementeller Build mit dem Management von Abhängigkeiten und so sind die Build-Zeiten vergleichbar mit dem eines inkrementellen Produkt-Builds. Mit Morpheus ist überhaupt kein Build notwendig. Natürlich sind die Zeiten für die Aktualisierung des lokalen Arbeitsbereichs etwas höher im Vergleich zu der Aktualisierung ohne Class-Dateien, aber üblicherweise liegen die Aktualisierungszeiten innerhalb weniger Minuten.

Die Build-Zeiten für einen kompletten Produkt-Build innerhalb CI liegen bei etwa 1 ½ Stunden ohne Morpheus und unter Verwendung eines beschleunigten Produkt-Builds mit Morpheus innerhalb von wenigen Minuten natürlich abhängig von der Anzahl der Änderungen. Der verwendete Rechner für den CI-Build hatte einen 4 x 3.6 GHz Prozessor und 32 GB Arbeitsspeicher.

5.5 Auswahl der automatischen Tests

Nach dem die Build-Zeiten verkürzt wurden, geht es als nächstes darum die Testausführungszeiten zu senken. Eine deutliche Verbesserung könnte durch die Reduktion der Anzahl der auszuführenden Tests erreicht werden. Nur Tests, die den geänderten Quellcode testen, sollten ausgewählt werden. Damit bekommt der Softwareentwickler nur Testergebnisse, die direkt mit seinem geänderten Quellcode in Verbindung stehen. Vier verschiedene Strategien wurden implementiert um passende Tests zu finden.

5.5.1 Anforderungsorientierte Testauswahlstrategie

Es gibt immer einen konkreten Grund, warum der Quellcode geändert wird, zum Beispiel eine neue Anforderung soll umgesetzt werden oder ein Softwarefehler muss behoben werden. Um sicherzustellen, dass die implementierte Funktionalität für eine Anforderung im nächsten Release auch noch fehlerfrei funktioniert oder der Softwarefehler nicht wiederauftaucht, werden automatische Tests erstellt um die Anforderungen oder die Lösung des Fehlers auch in zukünftigen Versionen wieder testen zu können. All diese Informationen können miteinander verlinkt werden: der geänderte Quellcode wird mit dem Änderungsgrund (Anforderung oder Testreport) verlinkt [107]. Die Anforderung oder der Testreport wird wiederum mit dem Testfall verlinkt [103] (zum Beispiel die Anforderung A wird durch den

[7] Abhängig von der Hardware des Entwicklerrechners

Testfall X getestet). Der Testfall wird mit dem Quellcode des automatischen Tests verknüpft. In Abbildung 73 und Abbildung 74 werden die Zusammenhänge vereinfacht dargestellt. In grüner Farbe ist der Produktiv-Quellcode, in blauer Farbe der Test-Quellcode und in brauner Farbe sind die Application Lifecylce Managment (ALM) Artefakte dargestellt. Das dafür notwendige Metamodell wurde bereits im Kapitel 3.4.3 definiert. Das heißt die Artefakte und die Verlinkung der Artefakte sind bereits im PREEvision Repository persistiert und können jetzt für die Auswahl der Testfälle genutzt werden.

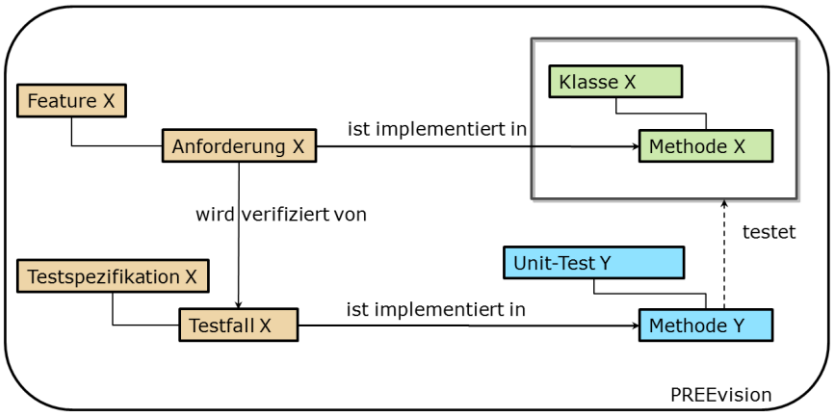


Abbildung 73: Auswahl des Tests über die Anforderung

Wie sieht die Bestimmung der richtigen Tests im Detail aus? Bevor der Softwareentwickler seine Änderungen abgibt, legt er einen Testfall an und verlinkt ihn mit dem Test-Quellcode (zum Beispiel mit einer Methode eines Unittests). Alternativ kann natürlich auch ein bereits vorhandener und geeigneter Testfall ausgewählt werden. Der Testfall wird dann vom Entwickler mit der Anforderung oder dem Testreport verlinkt. Bei der Abgabe des Quellcodes wählt der Entwickler dann die dazugehörende Anforderung oder den Testreport aus. Morpheus muss dann nur noch die Links durchlaufen vom abgegebenen AST Artefakt über die Anforderung oder den Testreport zum Testfall und der Testmethode im Test-Quellcode. Die gefundenen Testmethoden werden dann von Morpheus ausgeführt.

Mit der Hilfe der Historie der abgegebenen AST Artefakte ist es möglich sicherzustellen, dass bereits existierende Funktionalität durch die neuen Änderungen nicht wieder kaputtgegangen ist. Die AST Artefakte können nämlich über vorangegangen Abgaben bereits Links zu anderen Anforderungen und Testreports haben. Es macht auch Sinn nicht nur die abgegebenen AST Artefakte, sondern auch weitere AST Artefakte, die sich zum Beispiel in der gleichen Methode oder Klasse befinden, zu berücksichtigen und deren Links zu

Anforderungen und Testreports auszuwerten. Über die gefundenen Anforderungen und Testreports werden dann weitere Testmethoden bestimmt, die dann ausgeführt werden.

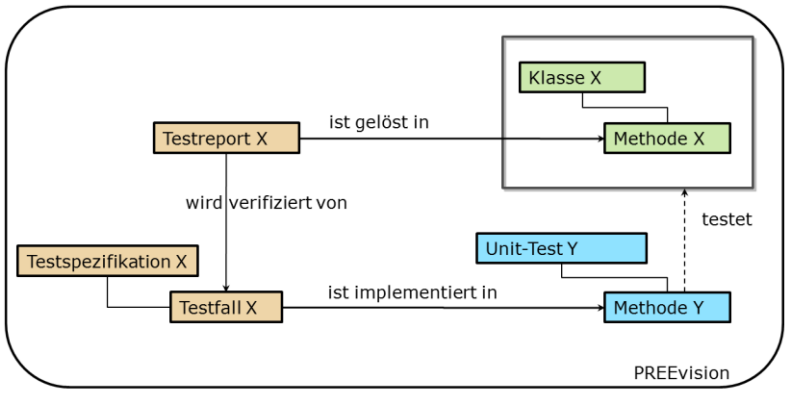


Abbildung 74: Auswahl des Tests über den Testreport

PREEvision unterstützt für die ALM Artefakte auch Lifecycles mit unterschiedlichen Zuständen wie zum Beispiel „neu“, „akzeptiert“ oder „abgeschlossen“. Während der Lifecylce-Übergänge können Überprüfungen durchgeführt werden, um sicherzustellen das gewisse Vorbedingungen für den Übergang erfüllt sind. Für diesen Anwendungsfall wird überprüft, dass die Anforderung oder der Testreport tatsächlich einen Testfall besitzen, wenn sie in den Lifecycle-Zustand „abgeschlossen“ wechseln. Damit wird auch eine gewisse Prozesssicherheit gewährleistet und der Anwender wird bei der Bearbeitung der Artefakte entsprechend geführt.

5.5.2 Software-Architekturorientierte Testauswahlstrategie

In Eclipse gibt es die allgemeine Praxis den Quellcode für die automatischen Tests vom Produktiv-Quellcode in jeweils eigene Plugins zu trennen. Für jedes Produktiv-Plugin wird ein Test-Plugin angelegt mit automatischen Tests, die den Quellcode des Produktiv-Plugin verifizieren (siehe Abbildung 75). Da nicht nur der komplette AST im PREEvision Repository abgelegt wird, sondern auch alle Packages und Plugins, wurde auch eine Relation im Metamodell zwischen Plugins definiert, die diese Beziehung repräsentiert („wird getestet von“). Über diese Relation kann zu einem Produktiv-Plugin das dazugehörige Test-Plugin bestimmt werden. Wenn der Entwickler den Quellcode in einem Produktiv-Plugin ändert, werden von Morpheus alle Testmethoden des dazugehörigen Testplugin berücksichtigt.

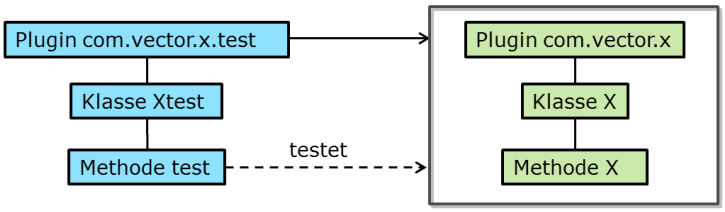


Abbildung 75: Auswahl des Tests über die Software-Architekturorientierte Testauswahlstrategie

5.5.3 Seiteneffektorientierte Testauswahlstrategie

Manchmal wird Code für eine bestimmte Funktionalität ausgeführt, die für den Entwickler nicht offensichtlich oder auch überraschend ist. Zum Beispiel ändert der Entwickler eine Klasse für eine bestimmte Anforderung und plötzlich funktioniert ein komplett anderes Feature nicht mehr, weil über mehrere Umwege dieser Code zur Ausführung kommt. Das Messen der Code-Abdeckung kann dazu verwendet werden, Tests für solche Art von Softwarefehlern zu finden [104]. Es gibt einige Werkzeuge am Markt, die eine Bewertung der Code-Abdeckung nach der Ausführung von Tests ermöglichen. Für das Morpheus Projekt wurde für diesen Zweck JaCoCo [108] verwendet. Während der Ausführung von Tests speichert JaCoCo alle besuchten Klassen für eine bestimmte Testklasse in einer Ausführungsdatei. Morpheus verwendet diese Datei um für eine geänderte Klasse die dazugehörenden Testklassen zu finden und fügt diese Testklassen zu der Liste der auszuführenden Tests hinzu.

5.5.4 Geänderter Test-Quellcode Testauswahlstrategie

Wenn während der Produktentwicklung neue automatische Tests erstellt oder bestehende Tests verändert wurden, dann berücksichtigt Morpheus diese Tests ebenfalls bei der Testausführung innerhalb des CI Laufs.

5.5.5 Priorisierung der gefundenen Tests

Man könnte jetzt argumentieren, dass die seiteneffektorientierte Testauswahlstrategie die einzige Strategie oder zumindest die Strategie mit der höchsten Priorität sein sollte. Aber die Kombination aller Strategien ist sinnvoll aus folgenden Gründen:

- Bevor der automatische Test erstmalige ausgeführt wurde, gibt es überhaupt keine Informationen welchen Code er durchlaufen wird und daher kann in diesem Fall kein einziger Test über diese Strategie bestimmt werden.
- Die Liste der besuchten Klassen für eine Testklasse ist nur eine Momentaufnahme und kann sich mit jeder Änderung des Quellcodes signifikant ändern.

- Für sehr grundlegende Funktionalitäten (zum Beispiel für die Datenzugriffsschicht) wird die Strategie eine sehr große Zahl von Tests finden, weil viele Tests diesen Code durchlaufen. Aber möglicherweise gibt es nur ein paar wenige Testklassen, die die Funktionalität sehr gründlich in einer sehr kurzen Zeit testen.

Die Testauswahlstrategien können eine große oder kleine Anzahl von automatischen Tests finden abhängig davon wie viele Tests verfügbar sind. Wenn die Anzahl der gefundenen Tests zu einer Ausführungszeit von mehreren Stunden führt, dann kann der Entwickler nicht mehr auf das Ergebnis warten. In diesem Fall muss die Anzahl der Tests reduziert werden indem eine Priorisierung der Tests erfolgt, sodass die Testausführungszeit ein definiertes Maximum nicht übersteigt. Das kann man zum Beispiel dadurch erreichen, dass eine Testauswahlstrategie als wichtiger bewertet wird als eine andere oder es werden erst Tests mit kürzen Testausführungszeiten von einigen wenigen Sekunden ausgeführt.

Die Anzahl der ausgeführten Tests ist ein Kompromiss zwischen ausführlichem Testen und einem schnellen Feedback für den Softwareentwickler, das ihm erlaubt zu einem sehr frühen Zeitpunkt Softwarefehler in seinem geänderten Quellcode zu finden und zu beheben. Die dann wieder erfolgreich durchlaufenden Tests können anderen Softwareentwicklern wichtige Hinweise über potentielle Fehler liefern. Ein über längerer Zeit fehlschlagender Test kann keine neuen Fehler aufdecken.

Die Testauswahlstrategien sind nur eine Annäherung um die am besten geeigneten Tests zu finden. Es ist nicht garantiert, dass die ausgewählten Tests tatsächlich die Softwarefehler finden, die über einen oder mehrere automatischen Tests gefunden werden könnten. In Zukunft könnten die Testauswahlstrategien durch Feedback von Softwareentwicklern und verschiedene Analysen und Auswertungen über das Modell noch weiter optimiert werden. Da nicht alle Tests ausgeführt werden, kann der CI Lauf mit der Ausführung aller Tests nicht weggelassen werden und wird weiterhin benötigt (zum Beispiel ein CI Lauf über Nacht).

5.6 Feedback für den Entwickler

Ein CI Lauf wird durch das Abgeben von geänderten Quellcode und den dazugehörigen Class Dateien gestartet. Der Softwareentwickler stellt den Änderungsgrund durch die Auswahl der entsprechenden Anforderung oder des Testreports bereit. Dann wird parallel in zwei Threads ein Produkt-Build erstellt und die automatischen Tests unter Verwendung der Testauswahlstrategien werden ermittelt. In unserem Fall (die Entwicklung einer 3-Schichten

Anwendung) wird die Datenbank aufgesetzt und die Middle-Tier und der Client werden gebaut und gestartet. Die ausgewählten automatischen Tests werden ausgeführt und die Ergebnisse werden in einem Report zusammengefasst. Dann verschickt Morpheus den Report zurück an den Softwareentwickler über eine E-Mail. Der Report enthält folgende Informationen:

- Es werden Informationen über den Change Set der Abgabe, die den Testlauf an getriggert hat, bereitgestellt (Änderungskommentar, Zeit der Abgabe, die abgegebenen Dateien)
- Der Report beinhaltet einen Überblick über alle gefundenen und ausgeführten Tests und deren Ergebnisse (erfolgreich oder fehlgeschlagen). Im Falle eines fehlgeschlagenen Tests wird der Softwareentwickler auch darüber informiert, wie oft der Test bereits fehlgeschlagen ist. Der Test kann wegen anderer Änderungen anderer Entwickler in der Vergangenheit schon fehlgeschlagen sein oder der Test ist fehlgeschlagen wegen der letzten Änderung.
- Der Report beinhaltet auch URLs zu Webseiten mit detaillierten Informationen über die fehlgeschlagenen Tests (welche Exception wurde geworfen und wie sieht der Stack Trace aus)
- Ein Testabdeckungsreport für alle ausgeführten Tests wird im Report zur Verfügung gestellt. Dieser Report erlaubt es dem Entwickler festzustellen, ob der Test wirklich den geänderten Quellcode ausgeführt hat und ob die Code-Abdeckung der geschriebenen Tests gut genug ist.

5.7 Ergebnisse

In dem Entwicklungsprojekt „PREEvision“ kam folgendes Vorgehen zur Anwendung: Der Softwareentwickler gab seine Quellcode Änderungen in das SCM Repository ab und es wurde von ihm erwartet nach der Ausführung der automatischen Tests die Ergebnisse im Laufe des nächsten Tages zu überprüfen. Der Entwickler musste raten welche Testergebnisse für ihn relevant waren und ob der Test wegen seinen Änderungen oder wegen den Änderungen eines anderen fehlgeschlagen ist. Häufige wurden fehlgeschlagene Tests überhaupt nicht untersucht, weil sich niemand dafür zuständig fühlte. Mit der Einführung von Morpheus bekommt der Entwickler automatisch Feedback via E-Mail über seinen geänderten Quellcode innerhalb einer Zeitspanne von 10 Minuten bis zu 3 Stunden abhängig von der Anzahl der ausgeführten Tests. Die Testergebnisse stehen direkt im Zusammenhang mit seinen Quellcode Änderungen.

Ein Ziel war die Reduzierung der Feedback Zeit. Die Build Zeiten sind jetzt so kurz, dass die Feedback Zeit im Wesentlichen von der Anzahl der ausgeführten Tests und der Dauer der Ausführung abhängig ist. Für die Systemtests müssen die Ausführungszeiten weiter optimiert werden. Die Startup- und Tear-Down Zeiten für die Tests sind mit ungefähr 5 Minuten noch zu hoch (starten der Anwendung, anlegen von Testdaten und löschen der Testdaten nach dem Test).

Morpheus wurde während der Entwicklung eins Service Packs einer bereits veröffentlichten Version des Produkts eingeführt und mitten in der aktuellen Produktentwicklung für das nächste Release. Für die Service Pack Entwicklung bekamen die Entwickle sehr gutes Feedback von Morpheus, weil fast alle automatischen Tests fehlerlos durchliefen und neu entstandene Softwarefehler sehr gut durch ein oder mehrere fehlschlagende Tests sichtbar wurden. Für die aktuelle Produktentwicklung gab es sehr viel fehlschlagende Tests wegen größerer Umbauten und Änderungen im Quellcode. Wenn ein automatischer Test bereits vor der Abgabe von Quellcode fehlschlägt, können neu entstandene Fehler nicht erkannt werden und der Test ist in diesem Fall wertlos. Deswegen sollte die Gesamtanzahl der fehlschlagenden Tests immer sehr niedrig sein. Bei einem konsequenten Einsatz und Nutzung von Morpheus kann das gelingen, weil dann die Fehler früh erkannt und behoben werden können.

Ein anderes Problem ist, dass Morpheus manchmal überhaupt keine automatischen Tests für die Quellcode Änderungen finden kann, weil es noch alten Quellcode gibt ohne irgendwelche automatischen Tests. Um die Situation zu verbessern ist jeder Softwareentwickler angehalten auch für Softwarefehler automatische Tests zu schreiben. Über die bereits beschriebenen Mechanismen des Lifecycles (Softwarefehler kann erst abgeschlossen werden, wenn ein Test vorhanden ist) kann das prozesstechnisch abgesichert werden.

6 Performance und Speicherbedarf

6.1 Einführung

Ein kritischer Faktor, der in dieser Arbeit beschriebenen Lösung, ist die Größe des Modells und der damit verbundene Speicherbedarf und die sich daraus ergebende Performance. Im Folgenden sollen an Hand von zwei konkreten Quellcode Projekten untersucht werden, wo die größten Herausforderungen liegen und es sollen auch schon erste konkrete Optimierungen umgesetzt werden. Das erste Projekt ist das Literaturverwaltungsprogram JabRef [109]. Der Quellcode ist frei verfügbar. Das zweite Projekt ist PREEvision selbst. Für beide Projekte soll der Quellcode nach PREEvision importiert werden. Neben Speicher- und Performanceuntersuchungen wird damit auch verifiziert, ob Morpheus wirklich in der Lage ist den Quellcode zu importieren und zu persistieren. In kleinen Quellcode-Beispielen fehlt die breite Nutzung der vielen verschiedenen Java Sprachkonzepte, was in einem großen Projekt besser abgedeckt ist. Es wurde in einem ersten Schritt eine Analyse an Hand des Quellcodes von JabRef durchgeführt und darauf aufbauend zwei Optimierungen bezüglich der Größe des Modells umgesetzt [110]. In einem zweiten Schritt wurde dann der Quellcode von PREEvision importiert.

Das Modell von Morpheus beinhaltet zum einen prozessrelevante Artefakte fürs Application Lifecycle Management und zum anderen den Quellcode (siehe auch Kapitel 3.1). PREEvision wird bereits für die Entwicklung von PREEvision als Application Lifecycle Management (ALM) Werkzeug eingesetzt. Es werden dabei alle in der Entwicklung anfallenden Daten in PREEvision in einem Modell erfasst und persistiert: Anforderungen, Features, Tickets, Testprojekte, Projektpläne, im Projekt tätige Personen und Dokumentation. Der Quellcode wurde bisher noch nicht in PREEvision abgelegt, sondern in Subversion. In den letzten 5 Jahren sind dabei etwa 5 Millionen ALM Artefakte entstanden. Mit diesen Modellgrößen kann PREEvision schon sehr gut umgehen und hier hat sich PREEvision in der Praxis bewährt. Die für den Quellcode zu erwartenden Modellgrößen sind jedoch deutlich höher und stellen eine neue Herausforderung dar. Traditionell werden Programmgrößen in „Lines of Code“ LOC (physikalische Zeilen) und SLOC (Source LOC) gemessen [111]. SLOC entspricht LOC aber ohne leere Zeilen und Kommentare. Da die Kommentare auch Teil des Abstract Syntax Trees sind, wird in dieser Arbeit LOC verwendet. Scheidgen [111] untersuchte für verschiedene Softwareprojekte die zu erwartende Modellgrößen an Hand von

tatsächlich gemessenen LOC (siehe auch Abbildung 76).

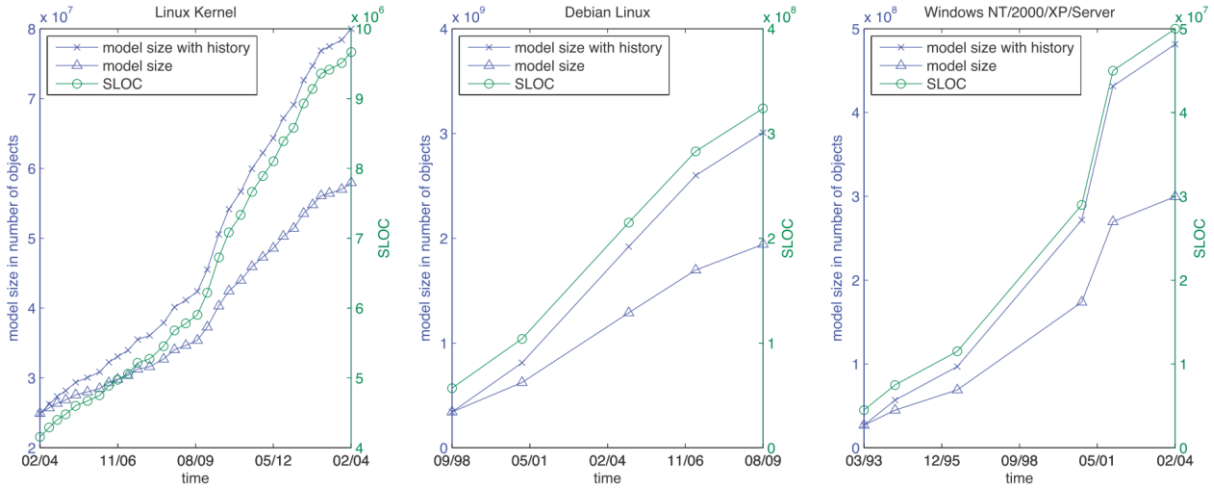


Abbildung 76: Grobe Abschätzung der Modellgrößen für unterschiedliche Softwareprojekte [111]

Die Anzahl der AST-Artefakte im Modell liegt in einer Größenordnung von bis zu 10^9. Die Berechnungen basieren auf der Annahme, dass 5,99 Artefakte pro Quellcode-Zeile vorhanden sind. Diese Zahl ergibt sich durch die gesamte Anzahl der Artefakte dividiert durch die Anzahl der Quellcode-Zeilen. Diese Zahl wurde für den Linux Kernel ermittelt. Die Anzahl der Artefakte pro LOC variiert abhängig von der Programmiersprache und vom Metamodell des ASTs. Für die ca. 6 Millionen LOC in PREEvision wird sich eine Modellgröße im Bereich von ungefähr 10^7 ergeben. Damit werden die Größe und das Wachstum des Gesamtmodells stark vom Quellcode-Anteil bestimmt und daher muss auch dieser genauer analysiert werden.

6.2 Analyse

6.2.1 Messpunkte

PREEvision ist eine 3-Schichten Anwendung (siehe auch 2.6.3). Der Speicherbedarf und die Performance müssen daher an verschiedenen Stellen gemessen werden. Dabei sind die Dauer und der maximale Speicherbedarf für die Durchführung einer bestimmten Operation (zum Beispiel die Abgabe des Modells in die Datenbank) und der permanente Speicherbedarf am Client und auf der Middle-Tier interessant. Beim Speicherbedarf ist zwischen Speicher auf der Festplatte und dem Arbeitsspeicher des Computers zu unterscheiden. Im Zuge dieser Analyse wird der Festplattenspeicher nicht weiter berücksichtig, weil der Festplattenspeicher für diese Anwendungsfälle weniger kritisch ist, als der Arbeitsspeicher. In Abbildung 77 sind die verschiedenen zu untersuchenden Operationen dargestellt.

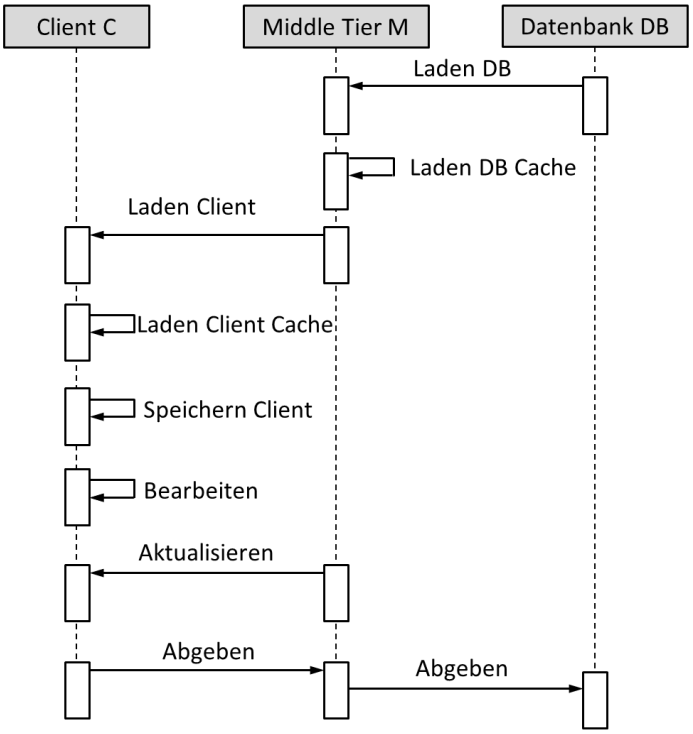


Abbildung 77: Kollaborative Operationen in PREEvision

Operation Laden DB: Das Laden des Modells aus der Datenbank findet während des ersten Hochfahrens des Servers statt (siehe Kapitel 2.6.3). Hier ist vor allem die benötigte Zeit für die Ausführung der Aktion interessant. Für das Laden wird nicht mehr Speicher notwendig sein, als permanent am Server benötigt wird. Die Zeitdauer ist nicht sehr kritisch, weil der Server in der Regel nur selten vollständig neu gestartet wird (ohne Cache-Datei).

Operation Laden DB Cache: Beim zweiten Hochfahren des Servers wird eine Cache-Datei verwendet. Damit wird der Start sehr viel schneller als beim Laden aus der Datenbank.

Operation Laden Client: Das Laden des Modells von der Middle-Tier ist für die erstmalige Anmeldung eines Anwenders wichtig. Hier ist die Dauer direkt für den Anwender erlebbar.

Operation Laden Client Cache: Das Laden des zwischengespeicherten Modells von der Festplatte wird bei jedem Start von PREEvision durchgeführt (mit der Ausnahme des erstmaligen Anmeldens) und ist deutlich wichtiger als die Ladezeit „Operation Laden Client“.

Operation Speichern: Das Speichern des Modells auf die lokale Festplatte wird beim Beenden von PREEvision durchgeführt und ist ähnlich kritisch wie die Zeit der Operation „Laden Client Cache“.

Operation Bearbeiten: Das Editieren von Quellcode Dateien oder auch das Importieren von ganzen Projekten ist bezüglich der Dauer und des maximalen Speicherbedarfs wichtig. Der maximale Speicherbedarf wird hier höher sein, weil das Meta Data Framework (MDF) von PREEvision eine Rückgängig-Funktion (Undo) bereitstellt (siehe auch Kapitel 2.6.2). Dafür werden zusätzliche Daten im Speicher gehalten. Quellcode-Dateien sind nicht beliebig groß und sollten aus Übersichtlichkeitsgründen auch nicht größer als einige 1000 Quellcode-Zeilen werden. Das Öffnen und Speichern einer Quellcode-Datei muss jedoch relativ schnell gehen, weil sie direkt vom Anwender erlebbar ist. Außerdem wird bei jedem Zugriff auf die Datei aus den AST Artefakten Text erzeugt, was sich auf die Kompilierzeiten auswirken kann. Der Import großer Quellcode-Projekte ist hier die größte Herausforderung. Da große Imports jedoch eher die Ausnahme darstellen, ist dieser Punkt nicht so kritisch.

Operation Aktualisieren: Das Aktualisieren ist wichtig bezüglich der Dauer. Das Aktualisieren wird entweder vom Anwender explizit angestoßen oder implizit über das Sperren von Artefakten. Die Größe der Aktualisierungsmenge wird von der Häufigkeit der Aktualisierung und der abgegebenen Datenmenge anderer Entwickler abhängen. Auch hier gilt, dass große Aktualisierungen durch große Imports die größte Herausforderung darstellen. Daher ist der Punkt ähnlich kritisch wie die Operation Bearbeiten.

Operation Abgeben: Beim Abgeben wird nicht unterschieden zwischen dem Abgeben zwischen Client und Middle-Tier und Middle-Tier und Datenbank, weil die gesamte Abgabe erst abgeschlossen ist, wenn beide Abgaben durchlaufen sind. Hier ist die Gesamtdauer wichtig und der maximale Speicherbedarf auf der Middle-Tier. Für das Durchlaufen der Änderungsmenge und das Aufbereiten der SQL Befehle für die Datenbank wird auf der Middle-Tier einige MByte mehr Speicher benötigt. Auch hier gilt das gleiche wie bei den Operationen Bearbeiten und Aktualisieren bezüglich großen Imports.

Speicher Middle-Tier: Der Speicherbedarf auf der Middle-Tier hängt direkt mit der Modellgröße zusammen. Der Wert ist nicht ganz so kritisch, weil ein Server in der Regel bezüglich der Hardware gut ausgestattet ist oder ausgestattet werden kann.

Speicher Client: Der Speicherbedarf am Client ist kritisch. Heutige Rechner haben in der Regel nicht mehr als 64 GB Arbeitsspeicher. Außerdem wird der Arbeitsspeicher auch noch für andere Programme und Daten benötigt.

Die Messpunkte sind von ihrer Bedeutung unterschiedlich wichtig. Daher wurden sie auch bei der Analyse unterschiedlich genau behandelt. Wenn das komplette Modell am Client geladen wird, dann ist der Speicherbedarf am Client und auf der Middle-Tier ungefähr gleich. Daher wird der Punkt Speicher Middle-Tier nicht weiter behandelt. Die Operation Aktualisieren wurde ebenfalls nicht weiter untersucht.

6.2.2 Messinstrumente

Für die Messung des Speicherbedarfs werden entsprechende Werkzeuge benötigt. Der Taskmanager von Windows ist zu ungenau, weil er nur den gesamten verwendeten Speicher anzeigt und nicht den tatsächlich für das Modell verwendete Heap-Speicher. Der Heap-Speicher ist der Speicherbereich in Java Programmen, wo die Modellartefakte abgelegt werden. Java stellt über das Java Development Kit (JDK) entsprechende Programme zur Messung des Heap-Speichers zur Verfügung.

6.2.2.1 JConsole

Mit JDK kommen die Programme VisualVM und JConsole. JConsole ist besser für solche Messungen geeignet und unterstützt auch den Export von Messwerten in CSV[8]-Dateien. Das Programm zeichnet verschiedene Werte (zum Beispiel Heap-Speicher oder die Nutzung der CPU[9]) über die Zeit auf (siehe auch Abbildung 78).

[8] Comma Separated Values
[9] Central Processing Unit

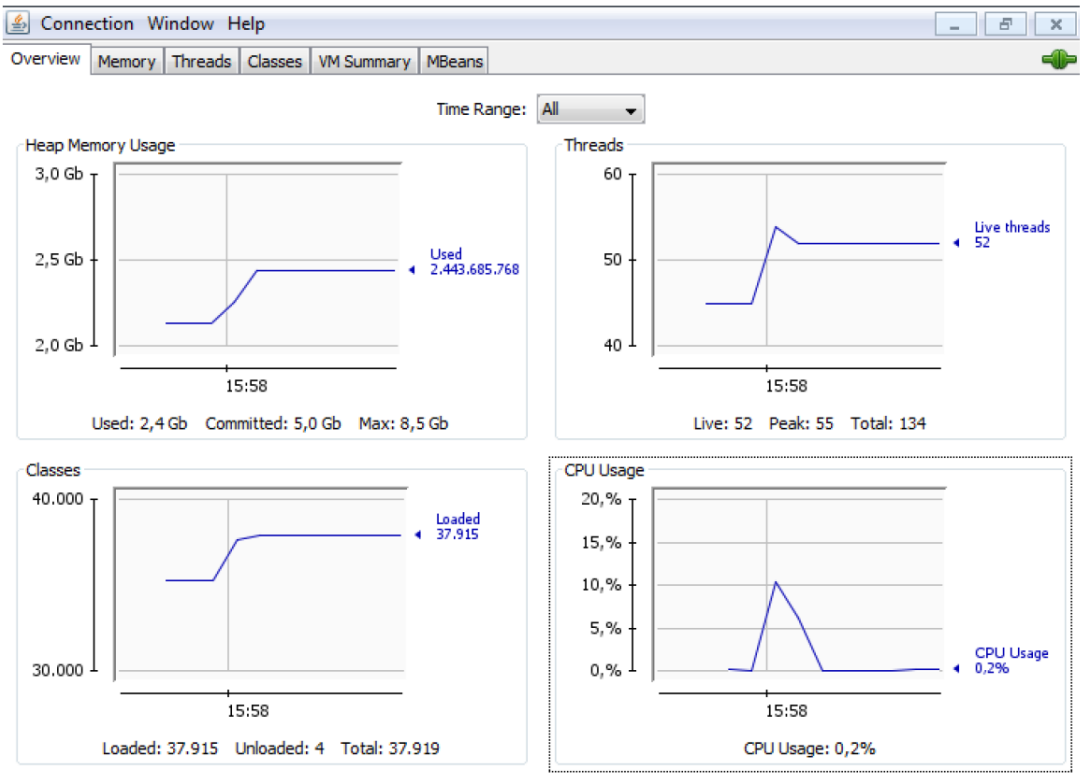
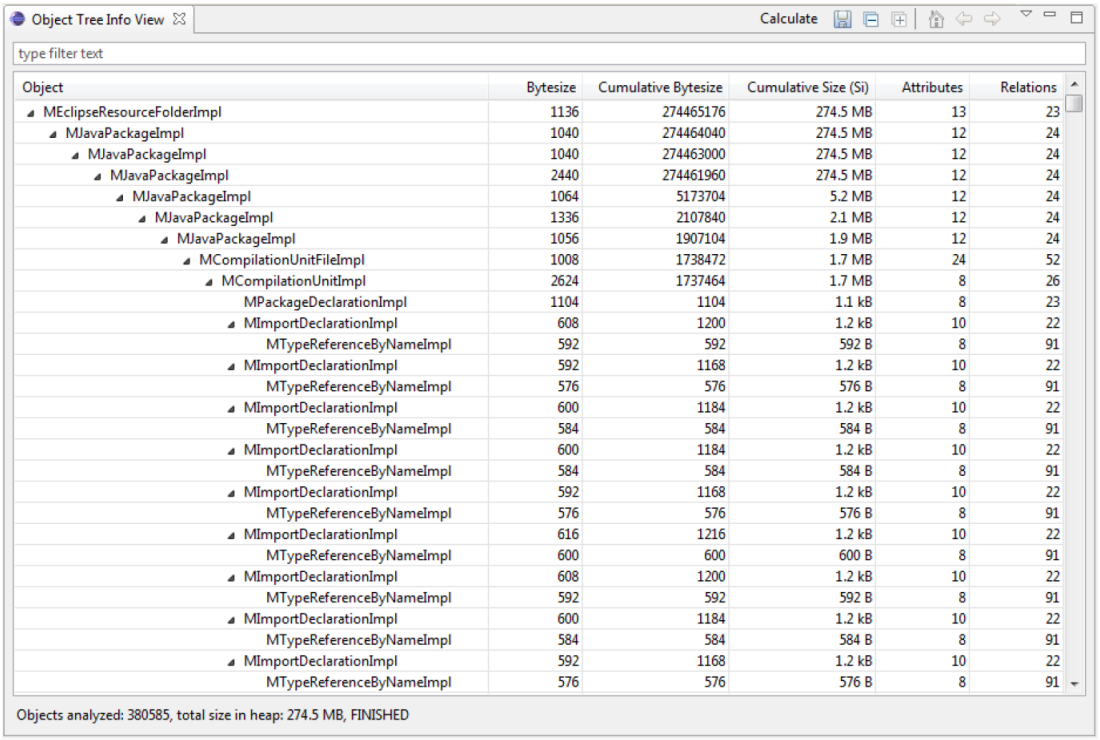


Abbildung 78: Benutzeroberfläche der JConsole

6.2.2.2 Object Tree Info View

Die Heap-Größe alleine gibt jedoch wenig Auskunft aus über einzelne Modellartefakte und deren Speicherverbrauch. Prinzipiell ist auch eine Heapdump-Analyse möglich, die ist jedoch sehr aufwändig. Um mögliche Optimierungen ableiten zu können ist eine genauere Untersuchung des Modells notwendig. Deswegen wurde eine neue Ansicht (View) entwickelt, die auf einem Modellartefakt im Baum geöffnet werden kann. Die Object Tree Info View zeigt für jedes Artefakt die Größe in Bytes und die kumulierte Größe unter Einbeziehung aller darunterliegenden Kind-Artefakte an. In Abbildung 79 ist ein Beispiel für ein Java-Projekt dargestellt. Außerdem wird entsprechend der Metaklasse des Artefakts auch die Anzahl der Attribute und Relationen angezeigt.



Object	Bytesize	Cumulative Bytesize	Cumulative Size (Si)	Attributes	Relations
MEclipseResourceFolderImpl	1136	274465176	274.5 MB	13	23
MJavaPackageImpl	1040	274464040	274.5 MB	12	24
MJavaPackageImpl	1040	274463000	274.5 MB	12	24
MJavaPackageImpl	2440	274461960	274.5 MB	12	24
MJavaPackageImpl	1064	5173704	5.2 MB	12	24
MJavaPackageImpl	1336	2107840	2.1 MB	12	24
MJavaPackageImpl	1056	1907104	1.9 MB	12	24
MCompilationUnitFileImpl	1008	1738472	1.7 MB	24	52
MCompilationUnitImpl	2624	1737464	1.7 MB	8	26
MPackageDeclarationImpl	1104	1104	1.1 kB	8	23
MImportDeclarationImpl	608	1200	1.2 kB	10	22
MTypeReferenceByNameImpl	592	592	592 B	8	91
MImportDeclarationImpl	592	1168	1.2 kB	10	22
MTypeReferenceByNameImpl	576	576	576 B	8	91
MImportDeclarationImpl	600	1184	1.2 kB	10	22
MTypeReferenceByNameImpl	584	584	584 B	8	91
MImportDeclarationImpl	600	1184	1.2 kB	10	22
MTypeReferenceByNameImpl	584	584	584 B	8	91
MImportDeclarationImpl	592	1168	1.2 kB	10	22
MTypeReferenceByNameImpl	576	576	576 B	8	91
MImportDeclarationImpl	616	1216	1.2 kB	10	22
MTypeReferenceByNameImpl	600	600	600 B	8	91
MImportDeclarationImpl	608	1200	1.2 kB	10	22
MTypeReferenceByNameImpl	592	592	592 B	8	91
MImportDeclarationImpl	600	1184	1.2 kB	10	22
MTypeReferenceByNameImpl	584	584	584 B	8	91
MImportDeclarationImpl	592	1168	1.2 kB	10	22
MTypeReferenceByNameImpl	576	576	576 B	8	91

Abbildung 79: Object Tree Info View zur Berechnung des Speicherverbrauchs

Die Daten können auch exportiert werden und dann für zusätzliche Auswertungen herangezogen werden zum Beispiel welche AST Klassen sind am häufigsten im Modell und wie groß ist der Speicherverbrauch der dazugehörigen Artefakte?

6.2.3 Messung JabRef

Im Folgenden sind die Ergebnisse der Messungen für den Quellcode des Projekts JabRef aufgeführt. Die Messungen wurden mit PREEvision 7.0 durchgeführt. Am Ende jedes Vorgangs wurde die Freigabe von nicht mehr genutzten Speichers (Garbage Collection) angetriggert. Es wurde jeweils ein Rechner für den Client und ein Rechner für den Server (Middle-Tier, die Datenbank und den Subversion-Server) verwendet (siehe auch Tabelle 4).

	Client	**Server**
Prozessor	8 x 3,6 GHz	8 x 3,6 GHz
Arbeitsspeicher	32 GB	128 GB
Festspeicher	250 GB SSD	250 GB SSD
Netzwerkanbindung	1 GBit/s	1 GBit/s
Systemtyp	64 Bit	64 Bit

Tabelle 4: Rechnerausstattung für den Import des Projekts JabRef

6.2.3.1 Operation Bearbeiten

An dieser Stelle wurde der extreme Fall getestet: der Import des kompletten JabRef-Projekts. In der Praxis werden die Änderungen eines Entwicklers deutlich kleiner sein.

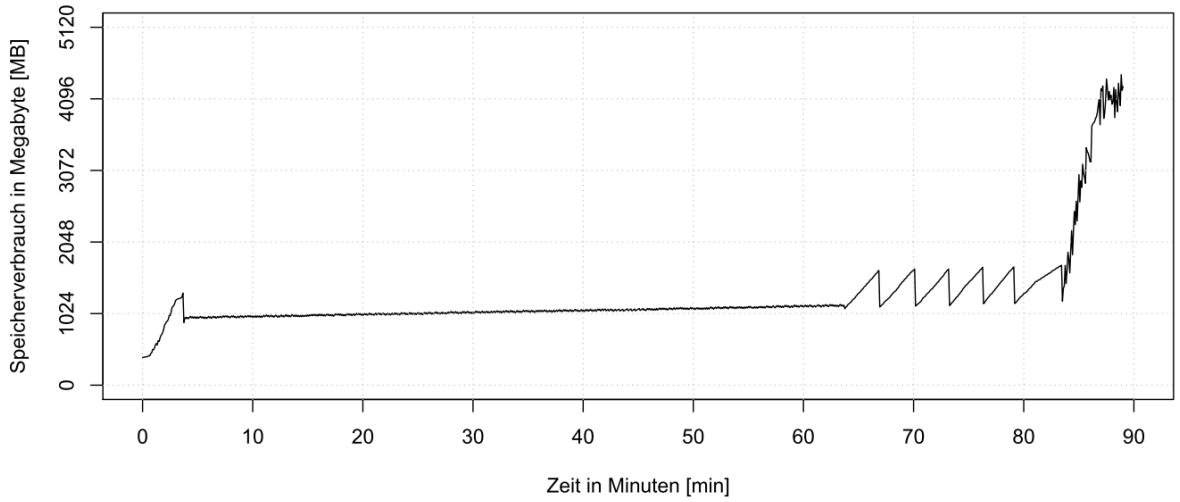


Abbildung 80: Speicherbedarf des Clients beim Importieren des Java-Projektes JabRef

Aus der Abbildung 80 kann man die Dauer und den Speicherbedarf für den Import entnehmen. Beim Import werden zuerst alle Java Dateien geparst und der AST für jede Datei in einem eigenen Import-Modell erstellt. Dann werden die Abhängigkeiten zwischen den AST Artefakten aufgelöst (siehe auch Kapitel 4.3.1). Die Auflösung der Abhängigkeiten nimmt die meiste Zeit in Anspruch. Zum Schluss wird das Import-Modell in das Hauptmodell verschmolzen. Hier kommt es zu einem starken Anstieg des Speicherbedarfs, weil alle Änderungen für die Undo-Funktion und die Änderungsmenge aufgezeichnet werden. Die Änderungsmenge wird bei der Abgabe dann an die Middle-Tier übertragen. Der relativ hohe Speicherbedarf verringert die Anzahl von Artefakten die für eine Abgabe erstellt werden können. Bei großen Projekten mit mehr als 100.000 Quellcode-Zeilen wird man den Import auf mehrere Importe aufteilen.

6.2.3.2 Operation Abgeben

Beim Abgeben des Modells sind der Client und der Middle-Tier relevant.

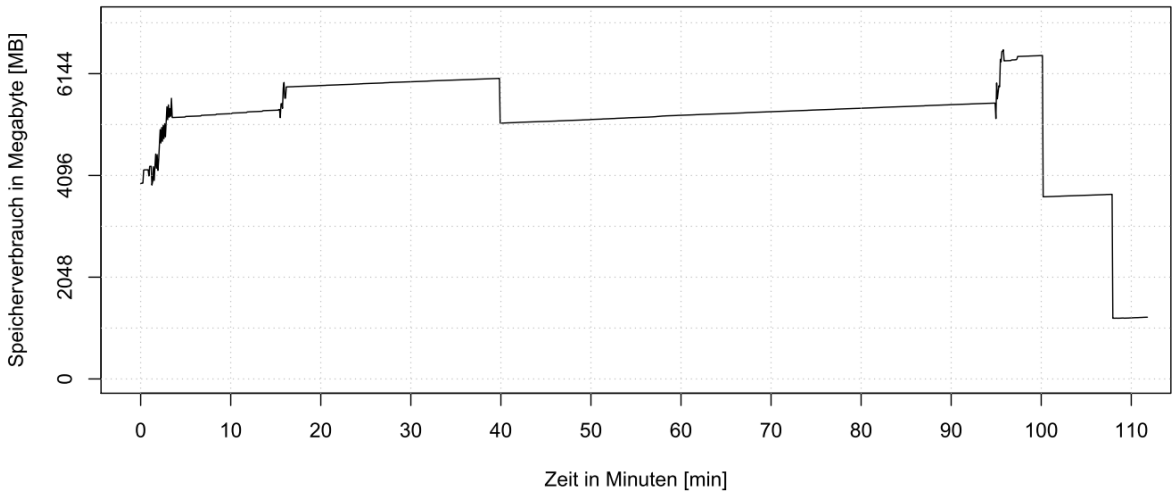


Abbildung 81: Speicherbedarf des Clients beim Commit des Java-Projektes JabRef

Am Client geht wie erwartet der Speicherbedarf nach der Abgabe und der Freigabe des Speichers wieder deutlich zurück (siehe Abbildung 81).

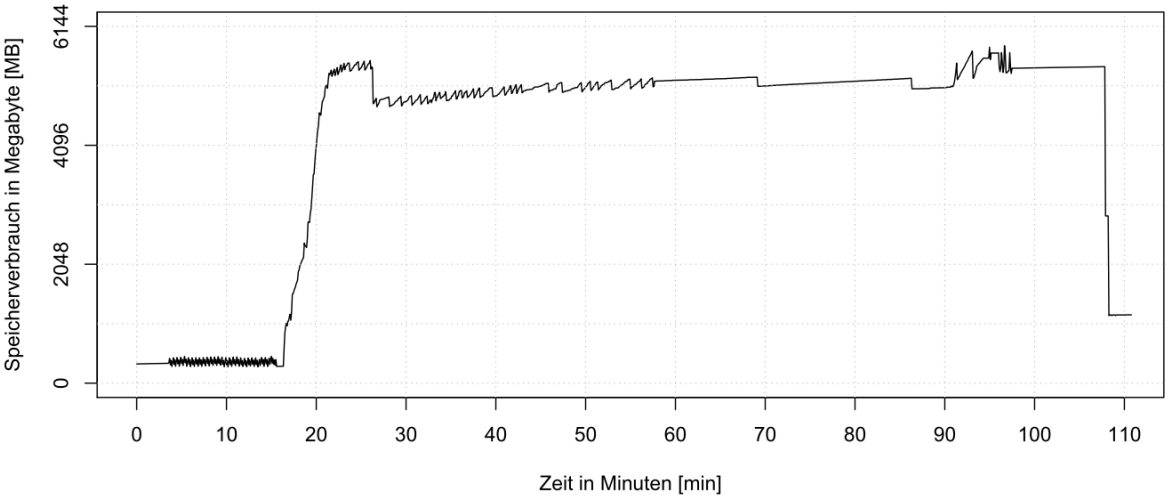


Abbildung 82: Speicherbedarf der Middle-Tier beim Commit des Java-Projektes JabRef

Auf der Middle-Tier steigt der Speicherbedarf während der Verarbeitung der Änderungsmenge an (siehe Abbildung 82). Nach dem die Änderungen in die Datenbank abgegeben sind, kann der Speicher wieder freigegeben werden. Als Differenz zum Speicherbedarf vor der Abgabe, bleibt der tatsächlich für das Modell benötigte Speicherbedarf.

6.2.3.3 Speicherbedarf Client

Für das Projekt JabRef wurden knapp 350.000 Modellartefakte angelegt mit einem Speicherbedarf von 408 Megabyte. Die LOC ohne Leerzeilen beträgt etwas mehr als 100.000 und damit ergeben sich 3,47 Modellartefakte pro LOC (siehe Tabelle 5 für Details).

	JabRef
Dateien (*.java)	659
Lines of Code (LOC) ohne Leerzeichen	100.409
AST-Modellartefakte	348.752
Speicherverbrauch in MB	408
Lines of Code (LOC) je Datei	152
Modellartefakte je Datei	529
Modellartefakte je LOC	3,47

Tabelle 5: Messung Speicherverbrauch und Modellgröße für das Projekt JabRef

6.2.4 Messung PREEvision

Die Messungen wurden mit PREEvision 7.5 und dem Quellcode von PREEvision 7.5 durchgeführt. In diesem Fall wurde nur ein Rechner für Client und Server verwendet (siehe Tabelle 6 für Details). Bei diesem Import wurde bereits die Optimierung des Metamodells verwendet (siehe Kapitel 6.3.1.1).

	Client und Server
Prozessor	8 x 3,5 GHz
Arbeitsspeicher	128 GB
Festspeicher	500 GB SSD
Systemtyp	64 Bit

Tabelle 6: Rechnerausstattung für den Import des PREEvision Quellcodes

6.2.4.1 Speicherbedarf Client

Für den PREEvision Quellcode wurden ungefähr 23,3 Millionen Artefakte angelegt mit einem Speicherbedarf von 30,8 GB. Die LOC ohne Leerzeilen beträgt etwas mehr als 6,5 Millionen und damit ergeben sich 3,57 Modellartefakte pro LOC (siehe Tabelle 7).

	PREEvision	Vergleich JabRef
Dateien (*.java)	41.541	
Lines of Code (LOC) ohne Leerzeichen	6.537.255	
AST-Modellartefakte	23.297.674	
Speicherverbrauch in GB	30.2	
Lines of Code (LOC) je Datei	157	152
Modellartefakte je Datei	561	529
Modellartefakte je LOC	3.57	3,47

Tabelle 7: Messung Speicherverbrauch und Modellgröße für das Projekt PREEvision

Wie auch aus der Tabelle 7 zu entnehmen ist, sind die Werte für LOC je Datei, Modellartefakte je Datei und Modellartefakte je LOC im Vergleich zu dem Projekt JabRef sehr ähnlich, obwohl es sich bei den zwei Projekten um sehr unterschiedliche Projekte handelt.

Im Modell werden neben den AST Artefakten auch alle Artefakte des Workspaces abgelegt also zum Beispiel das Plugin mit allen Dateiartefakten (siehe auch Kapitel 3.7.1). Für den PREEvision Quellcode sind das ungefähr 178.000 zusätzliche Artefakte. Im Vergleich zu den AST Artefakten spielt die Anzahl dieser Artefakte keine entscheidende Rolle (weniger als 1%). Auch die AST Artefakte, die nur als Supplier-Artefakte dienen und deren vollständiger Quellcode nicht im Modell gespeichert wird, zum Beispiel die verwendeten APIs aus der Java Runtime Class Libraries (siehe auch Kapitel 4.3.3), tragen mit ungefähr 83.000 zusätzlichen Artefakten nicht entscheidenden zur Größe des Modells bei. In Abbildung 83 sind die häufigsten vorkommenden Metaklassen aus dem importierten PREEvision Modell zu sehen.

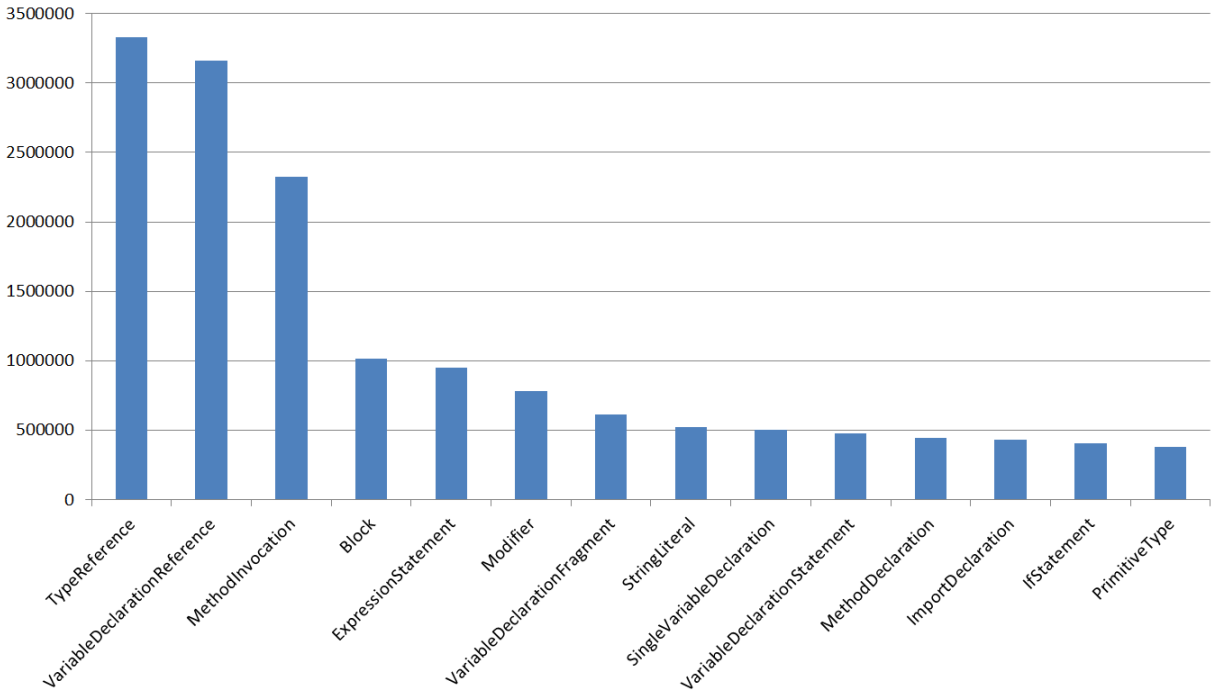


Abbildung 83: Die häufigsten Metaklassen mit ihrer Anzahl aus dem Projekt PREEvision

6.2.4.2 Operationen Laden DB, DB Cache, Client, Client Cache und Speichern Client

In Tabelle 8 sind die gemessenen Zeiten für das Laden des Modells auf der Middle-Tier und am Client und die Zeit für das Speichern des Modells am Client aufgelistet.

	Gesamtdauer	**Reine Lade- bzw. Speicherzeit**
Laden DB	5 Stunden	
Laden DB Cache	5 min 45 sec	
Laden Client	4 min 50 sec	3 min 50 sec
Laden Client Cache	5 min 04 sec	4 min 04 sec
Speichern	3 min 5 sec	2 min 20 sec

Tabelle 8: Ladenzeiten für das Projekt PREEvision

Am Client wurden die gesamte Startup-Zeit (vom Login-Dialog bis die Anwendung bedienbar ist) und die reine Ladezeit gemessen. Beim Speichern wurde ebenfalls die Gesamtdauer für das Beenden der Anwendung und die reine Speicherzeit gemessen. Beim Laden auf dem Server wurde nur die gesamte Startup-Zeit des Servers gemessen.

Zwei Dinge fallen sofort auf: die extrem lange Zeit für das Laden der Daten aus der Datenbank (Laden DB) und das Laden vom Server (Laden Client) geht schneller als von der lokalen Cache-Datei (Laden Client Cache). Die langen Ladezeiten aus der Datenbank sind

sicher ein Problem und bedürfen weiterer Optimierungen. Sie sind aber nicht ganz so kritisch, weil über das Laden aus der Cache-Datei (Laden DB Cache) der Server in angemessener Zeit hochgefahren werden kann. Da Client und Server sich auf dem gleichen Rechner befinden, fällt die Übertragungszeit des Modells zwischen Client und Server nicht wirklich ins Gewicht. Trotzdem ist es verwunderlich, warum das Laden von der Cache-Datei länger braucht. Hier gibt es offensichtlich noch Optimierungspotential und wahrscheinlich kommt man für diesen Fall auch unter die 5 Minuten. Bei einer weniger guten Verbindung zwischen Client und Server wird die Ladezeit vom Server natürlich entsprechend zunehmen. Aber diese Ladezeit ist nur für die erstmalige Anmeldung relevant.

6.2.4.3 Operation Bearbeiten

Eine weitere untersuchte Operation ist das Lesen des Java Quellcodes aus dem Modell. Bei dieser Aktion wird aus den AST Artefakten Quellcode-Text erzeugt. Die Dauer der Operation ist unabhängig von der Modellgröße, aber hängt von der Größe der Datei ab. In Tabelle 9 sind einige Messpunkte dargestellt.

Größe der Datei [LOC]	**Anzahl Artefakte**	**Dauer [ms]**
6	14	15
600	3100	300
1200	6800	600

Tabelle 9: Lesen einer Quellcode Datei

Die Dauer der Operation ist deswegen so wichtig, weil Eclipse relativ oft auf den Dateiinhalt zugreift. So wird schon beim Öffnen der Datei, der Dateiinhalt zweimal angefordert. Bei einem Neu-Bauen des kompletten Quellcodes werden alle Dateien mindestens zweimal gelesen. Das direkte Lesen der Datei von der Festplatte braucht weniger als 1 ms.

Die Konvertierung von AST Artefakten zu Text braucht pro LOC ungefähr 0.5 ms und eine Datei hat durchschnittlich 150 LOC. Das sind ungefähr 150 ms, die man zusätzlich für das Kompilieren einer Datei braucht, wenn die Datei zweimal von Eclipse gelesen wird. Bei 40,000 Dateien sind das dann 100 Minuten. Zwar ist die Konvertierung noch nicht performanceoptimiert, aber es wird wahrscheinlich nicht möglich sein, die Zeiten so maßgeblich zu verringern, dass sie keine Rolle mehr spielen.

6.2.5 Schlussfolgerungen

Die Messergebnisse bestätigen die Erwartungen. Der kritischste Punkt ist der Speicherbedarf am Client und die sich daraus ergebenden Performanceprobleme beim Laden und Speichern des Modells. Die Reduktion des Speichers am Client ist daher essentiell.

Die Startup-Zeit des Clients mit etwa 5 Minuten ist zwar etwas hoch, aber erstmal noch in einem vernünftigen Bereich.

Die 100 Minuten zusätzlich für das Kompilieren des Quellcodes sind dagegen nicht akzeptabel und damit ebenfalls ein kritischer Punkt.

6.3 Optimierungen

Das erste Ziel ist es den Speicherbedarf am Client zu reduzieren. Grundsätzlich gibt es zwei mögliche Wege der Optimierung:

1. Die Größe der AST Artefakte wird verkleinert, um so den Speicherbedarf insgesamt zu reduzieren.
2. Es werden nicht alle Modellartefakte dauerhaft im Speicher gehalten.

Das zweite Ziel ist die Lesezeiten für die Dateien besonders für den Kompiliervorgang zu verringern.

Die folgenden Erwartungen müssen dabei die Optimierungen erfüllen:

- Durch die Umsetzung darf die Laufzeit des Systems nicht unverhältnismäßig verschlechtert werden.
- Die Verfügbarkeit des Systems muss nach der Umsetzung unverändert hoch bleiben.
- Durch die Umsetzung darf unter keinen Umständen die Konsistenz von Modellen verletzt werden.
- Durch die Umsetzung dürfen sich im Umgang mit dem System keine Änderungen in der Bedienung ergeben.

6.3.1 Speicherreduktion der AST Artefakte

Für eine Verringerung des Speicherbedarfs der AST Artefakte kann zum einen das Metamodell und der aus dem Metamodell generierte Quellcode optimiert werden.

6.3.1.1 Optimierung des Metamodells

Das Metamodell bestimmt wie viele Instanzen mit welchen Attributen und Relationen für den AST im Modell angelegt werden und damit auch die Größe des Modells. Das Metamodell des ASTs orientiert sich am Eclipse AST, um möglichst viel Features von Eclipse wiederverwenden zu können. Daraus ergibt sich jedoch aus speichertechnischer Sicht kein optimales Metamodell. Durch Zusammenfassen von Klassen könnte zum Beispiel die Anzahl der Artefakte reduziert werden. Ein Umbau des Metamodells würde jedoch ein größerer Umbau von Morpheus nach sich ziehen und unter Umständen auch die Neuimplementierung von Eclipse Funktionaltäten.

Eine weitere weniger aufwändigere Möglichkeit über das Metamodell Einfluss auf die Größe des Modells zu nehmen, sind die Basisklassen der AST Klassen zu überprüfen. Über Basisklassen werden generische Konzepte zur Verfügung gestellt (siehe auch 2.6.1). Für die AST Artefakte muss überprüft werden, welche dieser Konzepte wichtig sind. Die Klassen zwischen *ASTNode* (Basisklasse aller AST Metaklassen) und *CoreArtefakt* (Basisklasse aller Metaklassen in PREEvision) sind daher genauer zu betrachten. Folgende Basisklassen kommen für eine Entfernung in Frage:

- *AbstractCreationAndModificationArtefact* : Diese Basisklasse stellt ein Erstellungs- und Änderungsdatum bereit.
- *AbstractNameAttributeArtefact:* Diese Basisklasse stellt ein Attribut für den Namen bereit.

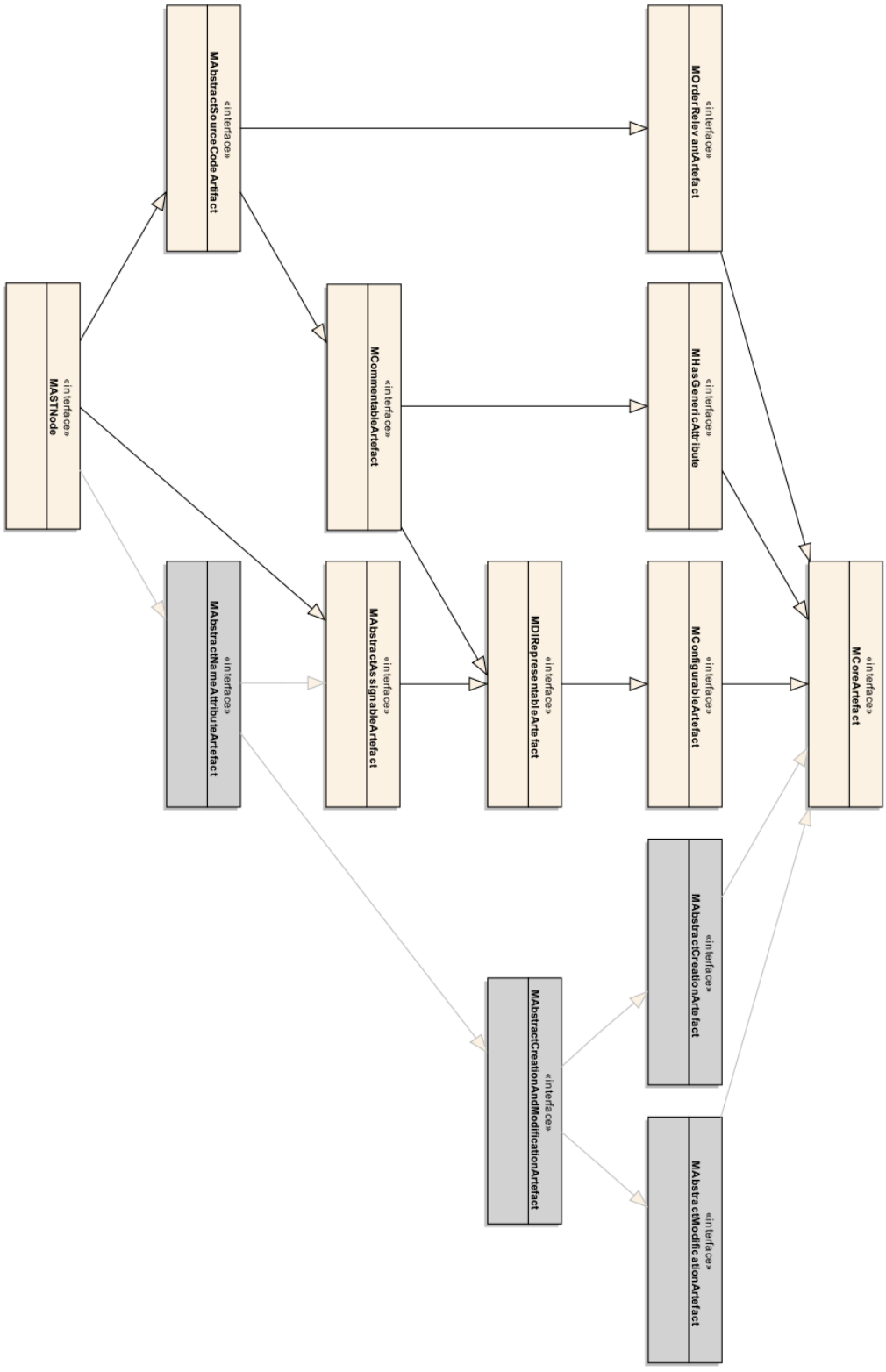


Abbildung 84: Auszug aus den Basisklassen des Metamodells

Nicht jede AST Klasse braucht einen Namen und ein Erstellungs- und Änderungsdatum. Daher wurden diese Basisklassen von der *ASTNode* Klasse entfernt. In Abbildung 84 sind die entfernten Klassen mit grauem Hintergrund dargestellt. Dafür musste für die Klassen, wo ein

Namensattribut notwendig ist (zum Beispiel *TypeDeclaration* oder *MethodDeclaration*), das Attribut explizit hinzugefügt werden.

6.3.1.2 Optimierung des generierten Quellcodes

Eine weitere Möglichkeit der Optimierung besteht in der Anpassung des aus dem Metamodell generierten Java Quellcodes (siehe auch Kapitel 2.6.2). Die Modellartefakte besitzen eine Reihe von Attributen (zum Beispiel Name) und Relationen, die nicht immer alle einen konkreten Inhalt haben. Dafür wird Speicherplatz belegt, der aber tatsächlich nicht genutzt wird (Null-Referenzen). Unter 64-Bit-Systemen fallen für eine Referenz beispielsweise 8 Byte an. Bei Millionen von Artefakten summiert sich der Speicherbedarf auf.

Für das Projekt JabRef besteht ein Speicherbedarf von 408.526.180 Bytes für 348.742 Artefakte. Durchschnittlich besitzt ein AST Artefakt somit 1171 Bytes. 350 Bytes werden als Basisgröße in jedem Fall benötigt (zum Beispiel für UUID und XmiID). Die restlichen Attribute und Relation könnten in einer HashMap abgelegt werden [112]. In der HashMap werden nicht belegte Attribute und Relationen keinen Speicher belegen. Der Zugriff auf die Attribute und Relationen ist dafür etwas langsamer, weil der Wert in der HashMap erst gesucht werden muss. Anstelle der HashMap kommen noch weitere alternative Collections (Sammlungen) in Betracht. Damit lässt sich der Speicherbedarf noch weiter reduzieren, wobei hier eine Abwägung zwischen Ersparnis und Performance zu treffen ist.

Der Quellcode-Generator von MMEdit wurde für die Verwendung der HashMap oder alternativ einer MutableMap der GS Collection [113] entsprechend umgebaut. Man kann dabei zwischen der Generierung des Quellcodes auf dem herkömmlichen Weg oder mit einer HashMap bzw. einer GS Collection wählen. Nach dem Umbau des Quellcode-Generators wurde der Java Quellcode neu generiert und für das Laden des Modells verwendet.

6.3.1.3 Evaluierung der Speicheroptimierungen

Um die Auswirkungen der Änderungen beurteilen zu können, müssen entsprechende Messungen durchgeführt werden. Dazu wurden die Messungen für das Projekt JabRef wiederholt. Die Ergebnisse sind in der Tabelle 10 dargestellt.

	Speicherbedarf insgesamt JabRef [Byte]	Speicherbedarf je Artefakt [Byte]
Vor der Optimierung	408.526.160	1.172
Nach Anpassung Metamodell	318.145.880	912
Nach Anpassung Metamodell und Generator (HashMap)	222.126.320	640
Verwendung der GS Collection	*186.212.856*	*534*

Tabelle 10: Ergebnisse der Optimierung an Hand des Projekts JabRef

Mit den Optimierungen konnte der Speicherbedarf von 408 Megabyte auf 220 Megabyte unter der Verwendung der HashMap reduziert werden. Hier sind keine signifikanten Verschlechterungen bei der Performance zu beobachten. Bei der Verwendung der GS Collection ist eine weitere Reduktion des Speicherbedarfs auf 186 Megabyte möglich. Jedoch sind mit der GS Collection deutliche Einbußen bei der Performance spürbar. Die Zugriffszeiten sind etwa um den Faktor vier höher. Daher kommt die GS Collection nicht in Frage.

Die Abbildung 85 gibt nochmal einen Überblick über den Speicherbedarf des Projekts JabRef mit der Ausgangssituation und den umgesetzten Anpassungen. In Abbildung 86 ist der Speicherbedarf während des Imports des Projekts JabRef vor und nach der Optimierung dargestellt. Die Importzeiten haben sich dabei nicht wesentlich verändert.

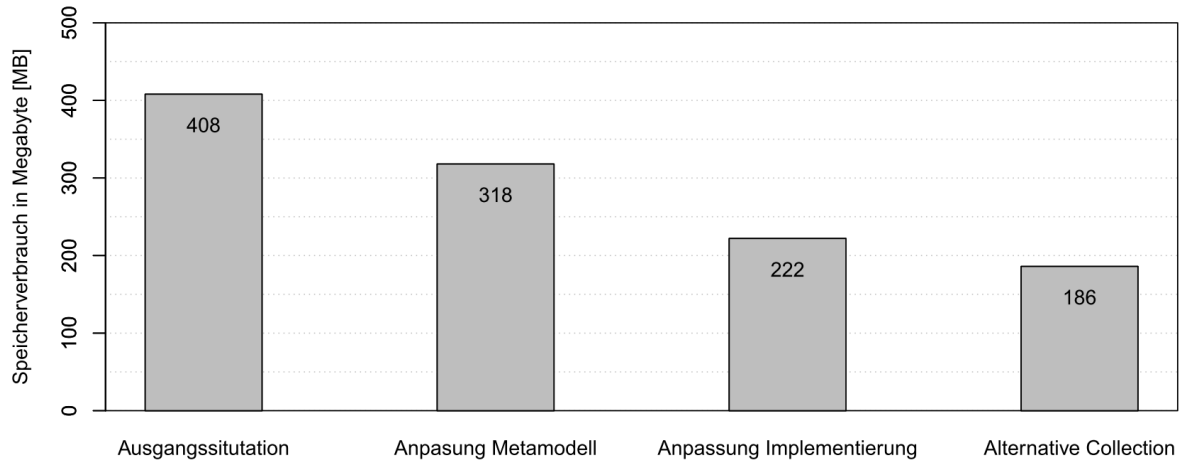


Abbildung 85: Vergleich der Ergebnisse des Speicherverbrauchs mit und ohne Optimierung

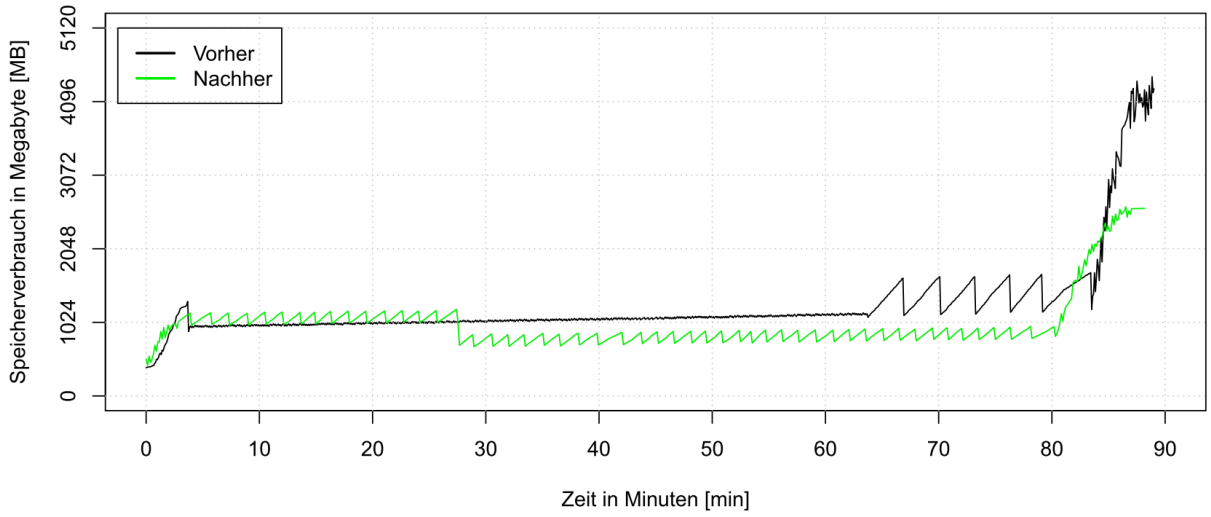


Abbildung 86: Speicherbedarf des Clients beim Import des Projekts JabRef vor und nach der Optimierung

Für den PREEvision Quellcode ergibt sich daher eine Verbesserung von ungefähr 30% von ca. 30 GB auf 20 GB.

6.3.2 Nachladen und Teilmodellverarbeitung

Die zweite Möglichkeit den Speicherbedarf für das Modell zu reduzieren ist nur ein Teil des Modells im Arbeitsspeicher zu halten. PREEvision bietet dazu zwei Möglichkeiten an:

1. **Nachladen**: Zu Beginn wird nur der die Wurzel und die erste Ebene des Modells geladen. Je nach der durchgeführten Aktion des Anwenders werden sukzessiv benötigte Artefakte vom Server nachgeladen. Dabei kann eine Nachladestrategie zum Einsatz kommen, um mit jeder Nachladeanfrage an den Server nicht nur ein Artefakt, sondern mehrere Artefakte nach zu laden, die wahrscheinlich in weiterer Folge benötigt werden. Ansonsten kann sich die Performance massiv verschlechtern, weil jedes Nachladen einen zeitlichen Overhead mit sich bringt.
2. **Teilmodellverarbeitung**: Der Anwender legt den Teil des Modells (Scope) fest mit dem er arbeiten will. Nur dieser Teil wird dann vom Server geladen. Auf Artefakte außerhalb des Scopes kann der Anwender nicht zugreifen, außer er fügt diese explizit dem Scope hinzu.

Der erste Ansatz (Nachladen) scheint besser für den Anwendungsfall dieser Arbeit geeignet zu sein. Es ergibt sich jedoch in beiden Fällen folgendes Problem: Wenn Quellcode verändert wird, muss der Quellcode und abhängiger Quellcode neu gebaut werden. Dazu muss auf den Quellcode bzw. das Modell zugegriffen werden. Das wird jedoch dazu führen, dass große Anteile des Modells und im Falle eines vollständigen Neu-Bauens der Anwendung das

komplette Modell im Speicher benötigt wird. Eine mögliche Lösung wäre den Quellcode dann doch in Dateien auf der Festplatte vorzuhalten. Es wäre also ein Mischbetrieb notwendig, wo mit Dateien und Modell parallel gearbeitet und der Dateiinhalt immer mit dem Modell synchron gehalten wird.

6.3.3 Lesezeiten des Quellcodes reduzieren

Die in Kapitel Nachladen und Teilmodellverarbeitung 6.3.2 vorgeschlagene Lösung würde auch hier das Problem lösen. Daher wurde die *CompilationUnit* Klasse um ein zusätzliches Attribut erweitert, das die Checksumme für den Dateiinhalt abspeichert. Wenn die Checksumme im Attribut nicht mit der Checksumme der aktuell vorhandenen Datei übereinstimmt, dann wird der Text neu generiert und auf die Platte gespeichert. Damit wird verhindert, dass auf einen veralteten Dateiinhalt zugegriffen wird. Wenn der Quellcode verändert wird, wird für den Quellcode-Text die neue Checksumme berechnet und im Modell aktualisiert.

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

Die Entwicklung vieler Features für ein Softwareprodukt in nur kurzen Release-Zyklen stellen besondere Herausforderungen an die Arbeitsteilung, Dokumentation und Kommunikation, Qualität und Prozesssicherheit dar. Mehrere Softwareentwickler müssen parallel am gleichen Quellcode arbeiten, den Quellcode verstehen und Änderungen am Quellcode mit hoher Qualität durchführen. Diese Herausforderungen kann man mit verbesserten und optimierten Prozessen und sehr guter Toolunterstützung begegnen. Oft gehen die zwei Themen Hand in Hand, weil die Werkzeuge die Prozesse unterstützen und Prozesssicherheit gewährleisten können. Es gibt mittlerweile eine Vielzahl von verschiedenen Werkzeugen für unterschiedliche Aufgaben in der Softwareentwicklung. Die Daten der Werkzeuge werden dabei in unterschiedlichen Repositories abgespeichert und können nur sehr umständlich über Traceability Links miteinander verknüpft und ausgewertet werden. Für eine verbesserte Toolunterstützung ist daher die Zusammenführung aller Daten in einem gemeinsamen Repository ein erster wichtiger Schritt. Die Daten müssen dabei feingranular abgespeichert werden. Das trifft besonders auf den Quellcode zu, der in der Regel nur als Datei mit Text abgelegt wird, ohne Wissen um den Inhalt der Datei. Erst feingranulares Speichern der Daten erlaubt es die Daten inklusive des Quellcodes miteinander zu verlinken und ermöglicht das parallele Bearbeiten der Daten. Dabei wird die Funktionalität eines Konfigurationsmanagementsystems benötigt um die Änderung der Daten mit deren Traceability Links historisch nachvollziehen zu können. Mit der Zusammenführung der Daten in ein Repository wird auch eine Benutzeroberfläche für die Sichtung und Bearbeitung der Daten benötigt. Damit wird ein durchgängiges Bedienkonzept möglich und die Einarbeitung in verschiedene Werkzeuge (zum Beispiel Entwicklungsumgebung, Bugtracker, Testdatenverwaltung, Dokumentenverwaltung) erübrigt sich. Die Informationen aus verschiedenen Domänen können kombiniert dargestellt werden und sind immer unmittelbar im Zugriff. Zum Beispiel kann im Quellcode-Editor Informationen über das umgesetzte Feature angezeigt werden. Diese verlinkten Daten mit ihrer Historie können dann analysiert und ausgewertet werden, um das Softwareprodukt zu verbessern (zum Beispiel verbessertes Testen) und den Softwareentwicklungsprozess zu optimieren. Der Quellcode und die

Traceability Links in den Quellcode können dabei in die Analyse miteinbezogen werden ohne den Quellcode vorher parsen zu müssen zum Beispiel für verschiedene Software-Metriken.

Für diese Ziele wurde in dieser Arbeit Morpheus entwickelt, das ein Repository mit einer Benutzeroberfläche zur Verfügung stellt, um alle Daten einschließlich des Quellcodes bearbeiten, verlinken, browsen (entlang der Links und über die Historie), analysieren, durchsuchen und verifizieren zu können.

Traceability

Um Traceability Links in den Quellcode zu ermöglichen, wird für den Quellcode ein Metamodell definiert (Abstract Syntax Tree - AST). In diesem Metamodell werden dann die Traceability Links zu anderen Metaklassen wie zum Beispiel Anforderungen, Tickets, Testimplementierungen und Dokumente definiert. Das Modell umfasst dann alle für die Softwareentwicklung relevanten Daten und wird in einem Repository abgespeichert. Für die Bearbeitung des Quellcodes wurde der Eclipse Java Editor so erweitert, dass die Links oder die Historie einzelner AST Artefakte beim Bearbeiten des Quellcodes nicht verloren gehen. Die Traceability Links können in diesem Editor sehr einfach über Drag & Drop und während der Abgabe des Quellcodes erstellt werden. Außerdem werden die Zielartefakte der Traceability Links auch im Editor visualisiert. Die Historie einzelner AST Artefakte kann einschließlich der Traceability Links auch im Editor zur Anzeige gebracht werden.

Paralleles Editieren von Quellcode

Durch das Vorliegen des Quellcodes als AST wird auch ein feingranulares pessimistisches Sperren einzelner AST Artefakte ermöglicht. Damit ist das parallele Bearbeiten einer Klasse oder einer Methode durch mehrere Entwickler möglich, ohne dass der Quellcode verschmolzen werden muss. Es können auch indirekte Konflikte und damit Syntaxfehler im Software Configuration Management (SCM) Repository verhindert werden. Es wurde eine Reihe von Sperrregeln definiert unter der Verwendung von exklusiven und geteilten Sperren, die sicherstellen, dass es zu keinen Konflikten kommen kann. Das Sperren erfolgt automatisch während des Editierens des Quellcodes im Eclipse Java Editor.

Continuous Integration

Nach der Abgabe des Quellcodes in das Repository soll möglichst schnell Feedback über die Qualität der Änderungen an den Entwickler zurückgemeldet werden. Deswegen wurde der Continuous Integration dahingehend erweitert, dass durch Abspeichern von Class-Dateien im

Repository ein schneller Build ermöglich wird und durch verschiedene Testauswahlstrategien für den geänderten Quellcode relevante Tests ausgeführt werden. Eine Testauswahlstrategie verwendet dabei die Traceability Informationen zwischen geändertem Quellcode, Anforderung, Testspezifikation und dem Test-Quellcode.

Performance und Speicherbedarf

Bei größeren Projekten entstehen sehr große Modelle, wenn der Quellcode als AST im Modell abgelegt wird. Diese Modelle sind eine Herausforderung bezüglich des Speicherbedarfs und der Performance. Zur Analyse wurden zwei Quellcode-Projekte importiert und der Speicherbedarf untersucht. Das größere der beiden Projekte hatte über 6,5 Millionen Quellcode-Zeilen. Der Speicherbedarf des Modells konnte durch Änderungen am Metamodell und am Quellcode-Generator verringert werden. Außerdem wurden die Möglichkeiten nur Teile des Modells im Speicher zu halten diskutiert.

7.2 Alleinstellungsmerkmale

In diesem Kapitel werden noch einmal die wichtigsten Alleinstellungsmerkmale dieser Arbeit zusammengefasst dargestellt. Dazu wurden bereits folgende Produktmerkmale definiert:

A) Feingranulare SCM

Der Quellcode wird nicht mehr als Text, sondern feingranular in einem Repository abgespeichert. Die Historie der Artefakte ist verfügbar, jedoch wird nicht unbedingt der vollständige Abstract Syntax Tree verwendet. Teile des Quellcodes können weiter als Text abgelegt werden.

B) SCM mit AST

Der Quellcode wird vollständig als Abstract Syntax Tree im Software Configuration Management (SCM) Repository abgespeichert. Jedes AST Artefakt hat seine eigene Historie, hat eine eindeutige ID und kann referenziert werden.

C) Stabile Traceability Links in den Code

Es werden Traceability Links in den Quellcode unterstützt. Beim Bearbeiten des Quellcodes gehen diese Links nicht verloren und es werden auch Refaktorierungen zum Beispiel umbenennen und verschieben unterstützt.

D) SCM ohne Syntaxfehler

Es werden indirekte Konflikte beim parallelen Bearbeiten des Quellcodes verhindert, so dass es zu keinen Syntaxfehlern im SCM Repository kommen kann.

E) Ein Repository für alle Daten

Es werden alle Daten, die im Zuge der Entwicklung einer Software entstehen, in einem Repository abgelegt (zum Beispiel Anforderungsmanagement, Änderungsmanagement und Testdatenmanagement). Es gibt auch eine Benutzeroberfläche um diese Daten zu bearbeiten. Traceability Links zwischen den Artefakten wird unterstützt.

F) Feingranulare pessimistische Sperren

Es wird das pessimistische Sperren von AST Artefakten unterstützt, so dass es zu keinen Konflikten kommen kann und eine Abgabe des Quellcodes jederzeit möglich ist.

G) Editieren AST

Es wird das Editieren des ASTs ermöglicht, wobei die Objektidentität durch Verschieben oder Umbenennen nicht verloren geht.

H) Editieren AST und Traceability Links

Es wird das Editieren des ASTs unterstützt und das Erstellen von Traceability Links auf einzelne AST Artefakte. Bereits bestehende Traceability Links werden im Editor direkt im Quellcode angezeigt.

I) Schneller Build mit Java Class Dateien

Es wird ein schneller Produktbuild ermöglicht unter Verwendung von Class-Dateien, die mit dem Quellcode in einem Repository abgespeichert werden.

J) Automatischer Test

Es werden Tests automatisch für den geänderten Quellcode ausgeführt. Es erfolgt eine automatische Testauswahl und eine schnelle Rückmeldung an den Entwickler.

K) Test Case Priorisierung mit Traceability Links

Die Tests für eine automatische Testausführung werden auf Grund von Traceability Informationen zwischen geänderten Quellcode, Anforderungen, Testspezifikationen, Testimplementierungen und Test-Quellcode ausgewählt.

	Morpheus	MolhadoRef	Stellation	UNICASE Trace Client	Capra	Microsoft Visual Studio	MPS	CSI	NCrunch
A) Feingranulare SCM	✓	✓	✓						
B) SCM mit AST	✓								
C) Traceability Links in den Code	✓			✓[10]	✓[11]				
D) SCM ohne Syntaxfehler	✓								
E) Ein Repository für alle Daten	✓			✓		✓			
F) Feingranulare pessimistische Sperren	✓							✓[12]	
G) Editieren AST	✓						✓		
H) Editieren AST und Traceability Links	✓								
I) Schneller Build mit Class-Dateien	✓								
J) Automatischer Test	✓					✓			✓
K) Test Case Priorisierung Traceability	✓								

Tabelle 11: Alleinstellungsmerkmale

7.3 Ausblick

In PREEvision fehlen noch zwei wichtige Features, um das Speichern von Quellcode als Abstract Syntax Tree optimal zu unterstützten:

- PREEvision erlaubt es nur alle Änderungen am Modell auf einmal abzugeben. Eine Teil-Abgabe mit einem Teil der Änderungen ist nicht möglich. Der Softwareentwickler arbeitet aber manchmal an mehr als nur einem Ticket oder Feature. Nach der Fertigstellung will der Entwickler dann nur die Änderungen für ein Ticket oder Feature abgeben und die anderen Änderungen noch lokal behalten. Solange die Änderungen unabhängig voneinander sind und keine gemeinsamen überlappende Änderungen beinhalten, sollte das auch kein Problem sein.

[10] Teilweise: Traceability Links nur bis zur Klasse
[11] Teilweise: Traceability Links nur bis zur Methode
[12] Teilweise: Es werden nicht alle Anwendungsfälle berücksichtigt.

- Beim Anlegen eines Entwicklungszweigs für zum Beispiel die Erstellung eines Servicepacks werden in PREEvision alle Artefakte für den Entwicklungszweig kopiert. Die Anzahl der Artefakte im Modell würde sich damit auf einen Schlag verdoppeln und das Modell extrem schnell wachsen. Der Entwicklungszweig dürfte daher nur die tatsächlichen Änderungen im Vergleich zum ursprünglichen Modell beinhalten.

Nach dem der Quellcode als AST vorliegt, eröffnen sich dadurch eine ganze Reihe von zusätzlichen Möglichkeiten:

- PREEvision besitzt eine mächtige Modellsuche. Damit können Suchanfragen sehr einfach graphisch definiert werden. Diese Modellsuche kann vom Entwickler genutzt werden, um auf einfache Weise sehr spezifische Suchen im Quellcode durchführen zu können, die über die Möglichkeiten von Eclipse weit hinausgehen.
- PREEvision stellt auch eine Modell-zu-Modell Transformation (M2M) [79] zur verfügen, die ebenfalls graphisch definiert werden kann. Diese M2M kann für komplizierte Refaktorierungen genutzt werden. Dabei bleiben die Historie einzelner AST Artefakte und die Traceability Links erhalten.
- Das Metrik-Framework von PREEvision kann zur Berechnung verschiedenster Metriken verwendet werden. Dabei können alle Daten des Modells und eben auch der Quellcode und dessen Verlinkungen ausgewertet und entsprechende Kennzahlen berechnet werden.
- PREEvision stellt auch eine Reihe von verschiedenen Diagrammtypen zur Verfügung, unter anderem auch ein Klassendiagramm für die Definition von Metamodellen. Diese Diagramme können zur Dokumentation des Quellcodes verwendet werden. Durch die enge Kopplung zwischen Diagrammelementen und AST Artefakten bleiben diese Diagramme immer aktuelle und veralten nicht. Auch könnten diese Diagramme direkt aus dem Quellcode Editor zur einer Klasse oder Methode geöffnet werden, ohne das Tool wechseln zu müssen.

8 Literaturverzeichnis

[1] S. Roock and H. Wolf, *Scrum - verstehen und erfolgreich einsetzen,* Heidelberg: dpunkt.verlag GmbH, 2016.

[2] L. Lindstrom and R. Jeffries, *Extreme programming and agile software development methodologies,* Information systems management 21.3, 2004, pp. 41-52.

[3] J. Humble and D. Farley, *Continuous Delivery,* Addison-Wesley, 2011.

[4] D. E. Perry, H. P. Siy and L. G. Votta, *Parallel changes in large-scale software development: an observational case study,* ACM Transactions on Software Engineering and Methodology (TOSEM) 10.3, 2001, pp. 308-337.

[5] A. Bacchelli, M. D'Ambros, M. Lanza and R. Robbes, *Benchmarking lightweight techniques to link e-mails and source code,* 16th Working Conference on Reverse Engineering IEEE, 2009, pp. 205-214.

[6] E. Hajiyev, M. Verbaere and O. De Moor, *Codequest: Scalable source code queries with datalog,* European Conference on Object-oriented Programming. Heidelberg, Springer Berlin, 2006.

[7] A. J. D. R. ,. a. V. G. Ko, *Information needs in collocated software development teams,* Proceedings of the 29th international conference on Software Engineering. IEEE Computer Society, 2007.

[8] J. Sillito, G. C. Murphy and K. D. Volder, *Asking and answering questions during a programming change task,* IEEE Transactions on Software Engineering 34.4, 2008, pp. 434-451.

[9] A. Delater, *Tracing requirements and source code during software development,* Dissertation, 2013.

[10] J. Adersberger, *Modellbasierte Extraktion, Repräsentation und Analyse von*

Traceability-Informationen, Herbert Utz Verlag, 2013.

[11] J. Estublier and S. Garcia, *Process model and awareness in SCM,* Proceedings of the 12th international workshop on Software configuration management. ACM, 2005.

[12] B. Khan and A. Sarma, *Cassandra: Proactive conflict minimization through optimized task scheduling,* Software Engineering (ICSE), 2013 35th International Conference on. IEEE, 2013.

[13] R. Conradi and B. Westfechtel, *Version models for software configuration management,* ACM Computing Surveys (CSUR) 30.2, 1998, pp. 232-282.

[14] R. E. Grinter, *Using a configuration management tool to coordinate software development,* Proceedings of conference on Organizational computing systems. ACM, 1995.

[16] *Building a better bug-trap,* Economist Magazine, June 2003.

[17] G. Mogyorodi, *Requirements-based testing: an overview,* Technology of Object-Oriented Languages, International Conference on. IEEE Computer Society, 2001, p. 286.

[18] M. Fowler and M. Foemmel, *Continuous integration,* Thought-Works) http://www. thoughtworks. com/Continuous Integration.pdf, 2006.

[19] Eclipse Foundation, *Eclipse Neon,* Retrieved 2017-07-30 from http://eclipse.org, 2016.

[20] Oracle, *Java SE Specifications,* Docs.oracle.com. Retrieved 2017-07-30 from https://docs.oracle.com/javase/specs/.

[21] Vector Informatik GmbH, *PREEvision – Development Tool for model-based E/E Engineering,* Retrieved 2016-07-30 from https://vector.com/vi_preevision_en.html., 2016.

[22] A. Leon, *Software configuration management handbook,* Artech House, 2015.

[23] C. M. Pilato, B. Collins-Sussman and B. W. Fitzpatrick, *Version control with*

subversion, 2008: O'Reilly Media, Inc..

[24] A. Koc and A. U. Tansel, *A survey of version control systems,* ICEME 2011, 2011.

[25] *Git,* Retrieved 2017-05-28 from https://git-scm.com/.

[26] A. Sarma, Z. Noroozi and A. Van Der Hoek, *Palantír: raising awareness among configuration management workspaces,* Software Engineering, 2003. Proceedings. 25th In-ternational Conference on. IEEE, 2003.

[27] S. Levin, *Synchronized software development,* Diss. Tel-Aviv University, 2013.

[28] J. Cleland-Huang, O. Gotel and A. Zisman, *Software and systems traceability. Vol. 2. No. 3.,* Springer, 2012.

[29] O. C. Gotel and A. C. Finkelstein, *An analysis of the requirements tracea-bility problem,* Requirements Engineering, Proceedings of the First International Conference on. IEEE, 1994.

[30] F. A. Pinheiro, *Requirements traceability,* Kluwer International Series in Engineering and Computer Science, 2004, pp. 91-114.

[31] C. S. Corley and e. al, *Recovering traceability links between source code and fixed bugs via patch analysis,* Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering. ACM, 2011.

[32] A. Egyed and P. Grunbacher, *Automating requirements traceability: Beyond the record & replay paradigm,* Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on. IEEE, 2002.

[33] A. Delater and B. Paech, *Tracing requirements and source code during software development: An empirical study,* Empirical Software Engineering and Measu-rement, 2013 ACM/IEEE International Symposium on. IEEE, 2013.

[34] K. E. Emam, W. Melo and J.-N. Drouin, *SPICE: The theory and practice of software process improvement and capability determination,* IEEE Computer Society Press,

1997.

[35] A.-P. Bröhl and W. Dröschel, *Das V-Modell*, München, Wien: Oldenburg-Verlag, 1995.

[36] S. Winkler and J. Pilgrim, *A survey of traceability in requirements engineering and model-driven development,* Software and Systems Modeling (SoSyM) 9.4, 2010, pp. 529-565.

[37] A. Krishnan, *Analysis, Design and Traceability of Model Transformations. Shaker*, Shaker, 2013.

[38] O. M. Group, Retrieved 2016-07-30 from http://www.omg.org.

[39] Object Managment Group (OMG), *OMG's MetaObject Facility (MOF) Home Page,* Omg.org. Retrieved 2016-07-30 from http://www.omg.org/mof/.

[40] Object Managment Group (OMG), *Unified modeling language,* Retrieved 2016-07-30 from http://www.uml.org/.

[41] MathWorks, *Simulink,* Retrieved 2018-01-30 from http://www.mathworks.de/products/simulink/..

[42] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers, Principles, Technique,* Boston: Addison wesley, 1986.

[43] B. Daum, *Java-Entwicklung mit Eclipse 3.3,* dpunkt.verlag, 2008.

[44] OSGi Alliance, *OSGI*, Retrieved 2017-07-30 from http://www.osgi.org.

[45] Oracle, *Eclipse Java development tools (JDT),* Retrieved 2017-05-28 from https://www.eclipse.org/jdt/.

[46] M. Fowler and K. Beck, Refactoring: improving the design of existing code, 1999: Addison-Wesley Professional.

[47] M. Kersten, *Eclipse Mylyn Open Source Project,* Eclipse.org. Retrieved 2016-07-30 from http://www.eclipse.org/mylyn/.

[48] J. Matheis, *Abstraktionsebenenübergreifende Darstellung von Elektrik/Elektronik-Architekturen in Kraftfahrzeugen zur Ableitung von Sicherheitszielen nach ISO 26262*, Shaker, 2010.

[49] *AUTOSAR (automotive open system architecture)*, Retrieved 2018-01-30 from http://www.autosar.org.

[50] Oracle, *Oracle Database*, Retrieved 2017-07-30 from https://www.oracle.com/de/database/index.html, 2017.

[51] K. Beck, *Embracing change with extreme programming*, Computer 32.10, 1999, pp. 70-77.

[52] P. M. Duvall, S. Matyas and A. Glover, *Continuous integration: improving software quality and reducing risk*, Pearson Education, 2007.

[54] A. Marcus and J. Maletic, *Recovering documentation-to-source-code trace-ability links using latent semantic indexing*, Software Engineering, 2003. Proceedings. 25th International Conference on. IEEE, 2003.

[55] S. Maro and J.-P. Steghöfer, *Capra: A Configurable and Extendable Traceability Management Tool*, Requirements Engineering Conference (RE), 2016 IEEE 24th International. IEEE, 2016.

[56] ISO, ISO, 26262–1: 2011 road vehicles functional safety, Geneva, Switzerland: International Organization for Standardization, 2011.

[57] Siemens Inc., *Application Lifecycle Management (ALM), Requirements Management, QA Management, Polarion Software*, Polarion.com. Retrieved 2016-07-30 from http://www.polarion.com/.

[58] IBM, *IBM - Rational Team Concert*, Www-03.ibm.com. Retrieved 2017-07-30 from http://www-03.ibm.com/software/products/de/rtc.

[59] Microsoft, *Microsoft Application Lifecycle Management*, Retrieved 2017-05-28 from https://blogs.msdn.microsoft.com/visualstudioalm/.

[60] U. Asklund, *Configuration Management for Distributed Development in an Integrated Environment*, Univ., 2002.

[61] B. Bruegge, A. H. Dutoit and T. Wolf, *Sysiphus: Enabling informal collabora-tion in global software development,* Global Software Engineering, 2006. ICGSE'06. International Conference on. IEEE, 2006.

[62] D. Dig and et al., *Molhadoref: a refactoring-aware software configuration management tool,* Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications. ACM, 2006.

[63] M. C. Chu-Carroll, J. Wright and D. Shields, *Supporting aggregation in fine grained software configuration management,* Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering. ACM, 2002.

[64] M. C. Chu-Carroll and S. Sprenkle, *Coven: Brewing better collaboration through software configuration management,* Vol. 25. No. 6. ACM, 2000.

[65] G. Antoniol and et a, *Recovering traceability links between code and documentation,* IEEE transactions on software engineering 28.10, 2002, pp. 970-983.

[66] N. Ali, Y.-G. Guéhéneuc and G. Antoniol, *Trustrace: Mining software repositories to improve the accuracy of requirement traceability links,* IEEE Transactions on Software Engineering 39.5, 2013, pp. 725-741.

[67] A. Egyed and P. Grünbacher, *Supporting software understanding with automated requirements traceability,* ternational Journal of Software Engineering and Knowledge Engineering 15.05, 2005, pp. 783-810.

[68] A. D. Eisenberg and K. D. Volder, *Dynamic feature traces: Finding features in unfamiliar code,* 21st IEEE International Conference on Software Maintenance (ICSM'05). IEEE, 2005.

[69] M. Langhammer, Automated Coevolution of Source Code and Software Architecture Models, 2017.

[70] Microsoft, *Visual Studio,* Retrieved 2016-07-30 from https://www.visualstudio.com/de/.

[71] CMMIT Produkt Team, *CMMI for Development, version 1.2,* 2006.

[72] JetBrains, *MPS overview,* Retrieved 2017-05-28 from https://www.jetbrains.com/mps.

[73] R. Robbes and M. Lanza, *Spyware: A change-aware development toolset,* Proceedings of the 30th international conference on Software engineering. ACM, 2008.

[74] Y. Sharon, M. Lanza and R. Robbes, *Eclipseye-spying on eclipse,* University of Lugano, Bachelor's thesis, 2007.

[76] D. D. McCracken and E. D. Reilly, Backus-naur form (bnf), Encyclopedia of Computer Science, 2003, pp. 129-131.

[77] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1994.

[78] S. Gao and et al., W3C XML schema definition language (XSD) 1.1 part 1: Structures, W3C Candidate Recommendation 30.7.2, 2009.

[79] C. Reichmann, *Grafisch notierte Modell-zu-Modell-Transformationen für den Entwurf eingebetteter elektronischer Systeme,* Shaker, 2005.

[81] A. Sarma, G. Bortis and A. Van Der Hoek, *Towards supporting awareness of indirect conflicts across software configuration management workspaces,* Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. ACM, 2007.

[82] P. Dewan, *Dimensions of tools for detecting software conflicts,* Proceedings of the 2008 international workshop on Recommendation systems for software engineering. ACM, 2008.

[83] S. Levin and A. Yehudai, *Alleviating Merge Conflicts with Fine-grained Visual Awareness,* arXiv preprint arXiv:1508.01872, 2015.

[84] P. Brosch, *Improving conflict resolution in model versioning systems,* Software

Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on. IEEE, 2009.

[85] Y. Brun and et al, *Proactive detection of collaboration conflicts,* Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. ACM, 2011.

[86] P. Dewan and R. Hegde, *Semi-synchronous conflict detection and resolution in asynchronous software development,* ECSCW 2007. Springer London, 2007, pp. 159-178.

[89] V. U. Gómez, S. Ducasse and A. Kellens, *Supporting streams of changes during branch integration,* In Science of Computer Programming, 2014, pp. 84-106.

[90] L. Hattori and M. Lanza, *Syde: a tool for collaborative software development,* Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2. ACM, 2010.

[91] M. Nordio, H. Estler, C. A. Furia and B. Meyer, *Collaborative software development on the web,* arXiv preprint arXiv:1105.0768, 2011.

[92] M. Goldman, G. Little and R. C. Miller, *Collabode: collaborative coding in the browser,* Proceedings of the 4th international workshop on Cooperative and human aspects of software engineering. ACM, 2011, pp. 65-68.

[94] S. McConnell, *Daily build and smoke test,* IEEE software 13.4, 1996, pp. 144-144.

[95] S. Elbaum, A. G. Malishevsky and G. Rothermel, *Prioritizing test cases for regression testing,* Vol. 25. No. 5. ACM, 2000.

[96] E. H. Baalbergen, *Design and implementation of parallel make,* Computing Systems, 1(2), 1988, p. 135–158.

[97] M. Pool, *DistCC, a fast free distributed compiler,* 2004: In Proceedings of linux.conf.au.

[98] Maven, Retrieved 2018-01-30 from https://maven.apache.org/index.html.

[99] T. Van Der Storm, *Backtracking incremental continuous integration*, Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on. IEEE, 2008.

[100] E. Thiele, *CompilerCache*, Retrieved 2016-07-30 from http://www.erikyyy.de/compilercache/.

[101] D. Saff and M. D. Ernst, *Continuous testing in Eclipse*, Proceedings of the 27th international conference on Software engineering. ACM, 2005.

[102] *NCrunch*, Retrieved 2017-05-28 from http://www.ncrunch.net/.

[103] H. Srikanth, L. Williams and J. Osborne, *System test case prioritization of new and regression test cases*, Empirical Software Engineering, 2005. 2005 International Symposium on. IEEE, 2005.

[104] R. Beena and S. Sarala, *Code Coverage Based Test Case Selection and Prioritization*, arXiv preprint arXiv:1312.2083, 2013.

[105] M. Yoon, *A Test Case Prioritization through Correlation of Requirement and Risk*, Journal of Software Engineering and Applications 5.10, 2012, p. 823.

[106] G. Rothermel, *Test case prioritization: An empirical study*, Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on. IEEE, 1999.

[107] U. Asklund, L. Bendix and T. Ekman, *Software configuration management practices for eXtreme programming teams*, Nordic Workshop on Programming and Software Development Tools and Techniques NWPER, 2004.

[108] *JaCoCo*, Retrieved 2017-05-28 from http://www.eclemma.org/jacoco/.

[109] O. Kopp, *JabRef Hilfe*, Retrieved 2016-07-30 from http://help.jabref.org/de/, 2016.

[111] M. Scheidgen, *How big are models–an estimation*, 2012: Tech. rep.

[112] Oracle, *Java Platform API Specification: Class HashMap,* Retrieved 2017-05-28 from http://docs.oracle.com/javase/7/docs/api/help-doc.html, 2016.

[113] Goldman Sachs, *GS Collections: User Reference Guide,* Version 5.0. New York, 2014.

Eigene Veröffentlichungen

[15] M. Eyl, C. Reichmann und K. Müller-Glaser, *Fast Feedback from Automated Tests Executed with the Product Build,* International Conference on Software Quality, Springer International Publishing, 2016.

[75] M. Eyl, C. Reichmann und K. Müller-Glaser, *Traceability in a Fine Grained Software Configuration Management System,* International Conference on Software Quality. Springer, Cham, 2017.

[87] M. Eyl, C. Reichmann und K. Müller-Glaser, *Prevent Collaboration Conflicts with Fine Grained Pessimistic Locking,* MODELSWARD 17, 2017.

Betreute Bachelor und Master Thesis

[53] H. Zhu, *Verknüpfung von Entwicklungselementen mit prozessrelevanten Artefakten in einer verteilten Umgebung,* Master Thesis, Hochschule Reutlingen, 2015.

[80] A. Maksimow, *Editieren eines in einer Datenbank persistierten Java Abstrakten Syntaxbaumes,* Bachelor Thesis, Hochschule Karlsruhe, 2013.

[88] S. Basavarajaiah, *Extension of Eclipse Java Editor to Java Abstract Syntax Tree existing in a Model-Based Database*, Master Thesis in Software Technology, Hochschule für Technik Stuttgart, 2014.

[93] T. Eger, *Entwicklung einer Build-Umgebung zur zeitnahen Integration von Änderungen und automatischen Ausführung änderungsspezifischer Tests,* Bachelor Thesis, Hochschule Karlsruhe, 2013.

[110] A. Laich, *Untersuchung und Verbesserung des Speicherverbrauchs einer modellbasierten Versionsverwaltung in einem System zur rechnergestützten Entwicklung von E/E-Architekturen,* Master Thesis, Hochschule Reutlingen, 2016.

Abkürzungsverzeichnis

ALM	Application Lifecycle Management
AST	Abstract Syntax Tree
API	Application Programming Interface
BNF	Backus Naur Form
CASE	Computer-aided software engineering
CI	Continuous Integration
CT	Continuous Testing
CM	Configuration Management
CMMI	Capability Maturity Model Integration
CPU	Central Processing Unit
CSI	Code Synchronizing Intelligence
CSV	Comma Separated Values
DSL	Domain-Specific Language
EMF	Eclipse Modelling Framework
EXE	Executable
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
JAR	Java Archiv
ID	Identität
JDK	Java Development Kit
JDT	Java Development Tools
LIS	Linking Integration Strategy
LOC	Lines of Code
MDF	Meta Data Framework
MOF	Meta Object Facility
MPS	JetBrains Meta Programming System
M2M	Model-to-Model
M2T	Model-to-Text
OMG	Object Management Group
OSGI	Open Services Gateway Initiative
PCM	Palladio Component Model

PDE	Plugin Development Environment
RCP	Rich Client Platform
RIS	Reconstructive Integration Strategy
SCM	Software Configuration Management
SLOC	Source Lines of Code
SPICE	Software Process Improvement and Capability Determination
SSD	Synchronized Software Development
SWT	Standard Widget Toolkit
TCP	Test Case Priorisierung
TFVC	Microsofts Team Foundation Version Control
UI	User Interface
UML	Unified Modeling Language
UUID	Universally Unique Identifier
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VM	Virtual Machine
XSD	XML Schema Definition
XMI	XML Metadata Interchange
XmiId	XMI Identifier
XML	Extensible Markup Language
XP	Extreme Programming