FACHBEREICH INFORMATIK
LEHRGEBIET PROGRAMMIERSYSTEME
Prof. Dr. Friedrich Steimann

Abschlussarbeit im Studiengang
Bachelor of Computer Science

Integration von Tracing in ein Framework zur Verknüpfung von gescheiterten Unit-Tests mit Fehlerquellen

Vorgelegt von

Thomas Eichstädt-Engelen
Matrikelnummer 6236561
thomas.eichstaedt-engelen@fernuni-hagen.de

Krefeld, den 12. September 2008

Inhaltsverzeichnis

1 Einleitung....1
1.1 Aufgabenstellung....2
1.2 Aufbau der Arbeit....3
2 Ausgangslage....4
2.1 EzUnit1....4
2.2 Eclipse....4
2.3 Test & Performance Tools Platform-Project (TPTP)....5
2.4 Eclipse Modelling Framework (EMF)....5
2.5 JUnit....5
2.6 Statische Programmanalyse....6
3 Konzept des FaultLocators....8
4 Implementierung....9
4.1 Erweiterung durch Neuimplementierung....9
4.1.1 Dynamische Programmanalyse....9
4.1.2 ExtensionPoint "FaultLocator"....20
4.1.3 Einheitliche Filter für statische und dynamische Traces....23
4.2 Erweiterungen durch Integration....26
4.2.1 Integration einer View....26
4.2.2 Selektionen des Benutzers....28
4.2.3 Reaktion auf Änderungen des Quelltextes....30
5 Installation und Bedienung....32
6 Ergebnisse....34
6.1 Zeitverhalten bei der Trace Erstellung....34
6.2 Tracegröße....36
6.3 Speicherauslastung....37
6.4 Unvollständige Invalidierung der Trace-Caches....38
7 Diskussion und verwandte Arbeiten....39
7.1 Alternative Tracingansätze....39
7.2 Vergleich mit verwandten Arbeiten....40
8 Schlussbetrachtung....41
8.1 Ausblick....41
8.1.1 Integration des TPTP-Profilers....41
8.1.2 Darstellung geänderter Methoden....42
8.1.3 Callgraph statt Trace (Aufrufreihenfolge)....42
8.1.4 Automatischer Hinweis, wenn Filter „unglücklich" gesetzt sind....42
8.1.5 Abschaffung der MUT-Annotationen....42
8.1.6 Integration mit Continuous Integration Frameworks....43
8.1.7 Entwicklung eines Wahrscheinlichkeits-Frameworks....43
8.1.8 Beheben kleiner „Unsauberkeiten" bei der Programmanalyse....43
8.2 Fazit....44
9 Glossar....45
10 Verzeichnisse....46

1 Einleitung

Zu jedem Lebenszyklus einer Software gehört neben der Entstehungsphase auch die Produktions- und Wartungsphase in der der stabile Betrieb gegenüber der Integration neuer Funktionen Vorrang hat. Zur Stabilisierung der Software können strukturelle Änderungen unter Beibehaltung des gleichen Programmverhaltens (Refaktorierung) vorgenommen oder gemeldete Fehler behoben werden. In beiden Fällen wird punktuell in ein Softwaresystem eingegriffen und es ist sicherzustellen, dass der Eingriff schadlos für das restliche System war. Hierzu werden feingranulare Regressionstests auf der Ebene des Moduls (*engl.: Unit*) durchgeführt. Diese Tests sollten möglichst häufig laufen, idealerweise immer dann, wenn Dateien des Softwaresystems geändert wurden.

JUnit ist ein Framework, welches den Entwickler bei der Erstellung solcher Unit-Tests unterstützt [jUnitOrg]. Dabei werden die erstellten Tests werden mit Hilfe des so genannten *TestRunners* (einer kleinen Applikation) durchgeführt. Nachdem der *TestRunner* gelaufen ist, weiß der Entwickler, ob alle Tests erfolgreich waren. Sind Fehler aufgetreten, ist allerdings nur erkennbar, welche Testmethode fehlgeschlagen ist. Der Entwickler erhält keinen Hinweis darauf, in welcher Methode des getesteten Programms die Ursache zu suchen ist.

Eine erste Verbesserung dieses Problems brachte *EzUnit* [Meyer2007], ein Plug-In für die Entwicklungsumgebung *Eclipse*. *EzUnit* verknüpft Testmethoden mit den von ihr aufgerufenen Methoden, den so genannten *MethodUnderTest (MUT)* mit Hilfe von Annotationen. Das Plug-In unterstützt ebenfalls bei der Anlage und Pflege dieser Annotationen mit Hilfe der statischen Programmanalyse. Dieses Vorgehen ist allerdings fehleranfällig, weil die Vollständigkeit der Annotationen durch den Entwickler letztlich manuell sichergestellt werden muss. Darüber hinaus liefert die statische Programmanalyse mehr *MUT*-Kandidaten zurück, als tatsächlich aufgerufen werden.

Mit der dynamischen Programmanalyse, die detaillierte Informationen über den tatsächlichen Programmfluss gibt, könnte die Anzahl der *MUT*-Kandidaten reduziert werden.[1]

Im Rahmen der folgenden Abschlussarbeit wurde eine Form der dynamische Programmanalyse, das Tracing auf Basis eines Standard Frameworks, dem *Eclipse TPTP* integriert. Wir werden zeigen, dass die Traces in akzeptabler Zeit erzeugt werden können. Dies ist möglich, weil die besondere Eigenschaft von Unit-Tests ausgenutzt werden kann, dass ihre Traces deterministisch sind. Erst wenn sich eine Testmethode oder eine von ihr aufgerufene Methode ändert, ist der jeweilige Traces neu zu erzeugen. Wir werden außerdem zeigen, dass die dynamischen Traces im Mittel dreimal weniger *MUT*-Kandidaten enthalten als die statischen Traces. Überdies werden so genannte *FaultLocatoren* eingeführt, die über die Vollständigkeit der gesetzten Annotationen Auskunft geben. Alle diese Erweiterungen helfen dabei, die Anzahl der *MUT*-Kandidaten zu reduzieren und damit die Suche nach dem Verursacher eines fehlgeschlagen Testfalls zu erleichtern.

1 Das gerade das Nicht-Aufrufen einer Methode der Fehler sein könnte, wird hier außer Acht gelassen, weil die Analyse der Kontrolllogik nicht trivial und damit nicht in akzeptabler Zeit durchzuführen wäre.

1.1 Aufgabenstellung

Im Rahmen der folgenden Arbeit soll eine dynamische Programmanalyse implementiert und im *EzUnit*-Plug-In verankert werden. Möglichst langlebige Schnittstellen zur Akquisition der Tracedaten sind einer proprietären Umsetzung, wie beispielsweise mit Hilfe eines eigenen oder externen Tracers, vorzuziehen. Das Eclipse-Subprojekt *Test & Performance Tools Platform* (TPTP) kommt dafür in Frage.

Des weiteren soll eine View, die so genannte *MUTSelectionView*, entworfen werden, die alle *EzUnit* Funktionalitäten wie das Anzeigen des statischen/dynamischen Traces sowie das Setzen und Löschen von Annotationen nahtlos unterstützt. Die Auswahl einer Testmethode, zum Beispiel in der *Outline-View* oder einem geöffneten Editor, soll die Aktualisierung der *MUTSelectionView* nach sich ziehen. Auch sollen Änderungen an Methoden (Testmethoden oder *MUTs*) zur Anzeige führen, dass der zu Grunde liegende Trace möglicherweise nicht mehr stimmt. Im Falle des Scheiterns eines Testfalls sollen alle seit dem letzten erfolgreichen Testlauf geänderten Methoden markiert werden.

Zusammenfassend soll die *MUTSelectionView* für eine Testmethode die folgenden Informationen enthalten:

1. alle Methoden aus dem statischen Trace (mit Ausnahme der abstrakten),
2. alle Methoden aus dem dynamischen Trace
3. sowie zusätzlich die per Reflection aufgerufenen Methoden,
4. darunter markiert die Methoden, die im Falle des Scheiterns seit dem letzten erfolgreichen Testlauf der Testmethode geändert wurden,
5. sowie für die Testmethode selbst, ob sie bei der nächsten Ausführung wieder getracet werden muss (weil die Testmethode oder eine von ihr aufgerufene geändert wurde).

Der Ablauf soll sich in *EzUnit* (nach der Integration) wie folgt gestalten:

1. Der Benutzer wählt eine der Testmethoden einer JUnit-Testklasse in der *Outline-View* oder einem geöffneten Java-Editor aus.
2. Sollte im Cache für diese Testmethode ein Trace (statisch oder dynamisch) vorliegen, wird dieser verwendet.
3. Liegen keine Trace-Informationen vor, wird die Erzeugung im Hintergrund angestoßen.
4. Ein ProgressMonitor informiert den Benutzer darüber, dass Daten gesammelt werden
5. Das Klicken einer Checkbox in der *MUTSelectionView* führt zum Setzen/Entfernen einer Annotation für diese Methode.
6. Sollten alle/keine Methoden als Verursacher in Frage kommen, wird die „Alle auswählen/entfernen“ Aktion aufgerufen.
7. Durch Doppelklick auf eine Methode in der *MUTSelectionView* gelangt der Benutzer zur jeweiligen Methode (öffnen der Methode im Editor)
8. Neben den bereits bekannten T-Markern (möglicher Verursacher) sind weitere Marker unter Zuhilfenahme der dynamischen Analyse ergänzt worden[2].

2 Es sollen Marker für die Eigenschaft „Methode ist annotiert wurde aber nicht aufgerufen“ und „Methode wurde zwar aufgerufen, ist aber nicht annotiert“ implementiert werden.

Um EzUnit's Framework-Gedanken weiter zu leben, sollten weitere Fault Locatoren durch eine einheitliche Schnittstelle ergänzt werden können. Dazu könnten ExtensionPoints verwendet werden. Die in [Steimann2007] genannten Erweiterungen sollten implementiert sein.

Zur besseren Unterscheidung der verschiedenen Versionen des EzUnit Plug-Ins wird die ursprüngliche Version mit EzUnit1 referenziert. Die neue, im Rahmen dieser Arbeit entstandene, Version heißt EzUnit2.

1.2 Aufbau der Arbeit

Diese Arbeit ist wie folgt aufgebaut. Kapitel 2 beschreibt die Ausgangslage, sowie die wesentlichen Werkzeuge und Bibliotheken, die die Basis der Weiterentwicklung sind. Kapitel 3 führt danach das Konzept des *FaultLocators* ein. Es folgt die Beschreibung der Weiterentwicklungen, die im Rahmen dieser Arbeit durchgeführt wurden in Kapitel 4. Danach gibt Kapitel 5 einen knappen Überblick darüber, wie *EzUnit2* nebst weiterer Plug-Ins zu installieren ist und wie eine *ProfilingSession* manuell gestartet werden kann. Kapitel 6 stellt dann die beobachteten Ergebnisse vor. Kapitel 7 diskutiert anschließend alternative Tracing-Ansätze und grenzt diese Arbeit von verwandten Arbeiten ab. Schließlich gibt Kapitel 8 einige Hinweise auf Erweiterungen, die im Rahmen weiterer Arbeiten in Angriff genommen werden könnten.

2 Ausgangslage

In den folgenden Abschnitten sind die grundlegenden Werkzeuge vorgestellt, mit denen die Lösung erarbeitet wurde. Sie wurden nicht im Rahmen dieser Arbeit entwickelt.

2.1 EzUnit1

In der vorangegangenen Abschlussarbeit von Nils Meyer wurde das Eclipse Plug-In *EzUnit*[3], eine Erweiterung für *JUnit*, implementiert [Meyer2007][4]. Ziel von *EzUnit* ist es, den Komfort der Nutzung von *JUnit* durch die Verknüpfung von Programmcode zu Testfällen sowie umgekehrt von Testfällen zu den getesteten Programmeinheiten herzustellen. Diese Verknüpfung dient nicht nur der leichteren Navigation durch den Quelltext, sondern wird auch zur Bestimmung der Fehlerursache fehlgeschlagener Testfälle genutzt. Diese Verknüpfungen sind mit Hilfe von Annotationen[5] der Testmethoden realisiert.

Sollte ein Testfall bei einem Testlauf fehlschlagen, werden alle annotierten Methoden, die so genannten *MethodUnderTest* (MUT's), als mögliche Verursacher mit einem roten „T" markiert. Überdies werden sie in der Standard-Eclipse-Problemview unter der Kategorie *Other* aufgelistet.

Bei der Auswahl möglicher Verursachermethoden als Annotation erhält der Entwickler Unterstützung durch *EzUnit*. Es werden alle Methoden, die durch den Testfall aufgerufen werden könnten, in einem Dialog-Fenster aufgelistet und zur Auswahl angeboten. Der Benutzer entscheidet durch einfaches Auswählen, welche Methoden Verursacher für das Fehlschlagen dieses Testfalles sein könnten. Die dafür nötigen Daten werden aus der statischen Programmanalyse des jeweiligen Testfalles gewonnen. Aufgrund der technischen Restriktionen der statischen Analyse, wird die Menge möglicher Aufrufkandidaten sehr schnell sehr groß [Meyer2007]. Sie kann beispielsweise nicht entscheiden, welchen Weg die Ausführung eines Testfalls bei Verzweigungen und Wiederholungen nimmt, noch können dynamische Bindungen aufgelöst werden. So wird beispielsweise nicht erkannt, welche konkrete Unterklasse einer allgemein verwendeten Oberklasse in einem bestimmte Fall benutzt wird. Welche Aufrufe über Reflection vorgenommen werden, kann ebenfalls nicht ermittelt werden.

2.2 Eclipse

Als integrierte Entwicklungsumgebung (IDE) wurde die Eclipse-Plattform mit den für die Java-Entwicklung nötigen Plug-Ins des *Java Development Tools (JDT)* und *Plugin Development Environment (PDE)* Projektes eingesetzt.

Eclipse bezeichnet sich selbst als eine Entwicklergemeinschaft, deren Projekte darauf zielen, eine quelloffene Entwicklungsplattform mit erweiterbaren Frameworks und Werkzeugen zu entwickeln [eclipseOrg]. Zur Erreichung dieses Ziels hat die *Eclipse Foundation* ihre verschiedenen Entwicklungsstränge in zehn [Stand 13.01.2008] Toplevel- und viele darunter gruppierte Subprojekte organisiert. Bei der Erstellung von *EzUnit2* wurden neben den oben

3 Die :Webseite des Projektes ist unter www.fernuni-hagen.de/ps/prjs/EzUnit zu finden.
4 Diese Vorgängerversion wird im weiteren Verlauf dieser Arbeit *EzUnit1* genannt.
5 Annotationen sind Hilfsmittel zur Strukturierung des Quelltextes. In der Programmiersprache Java stehen sie seit der Version 5 zur Verfügung.

genannten JDT und PDE insbesondere auch einzelne Elemente der Projekte *Test & Performance Tools Platform Project (TPTP)* und *Eclipse Modelling Framework (EMF)* verwendet.

Für weitergehende Informationen zur Eclipse-Plattform und ihrer Architektur sind stellvertretend für viele die Werke von [BeckGamma2004] und [Daum2005] zu nennen.

2.3 Test & Performance Tools Platform-Project (TPTP)

Das Toplevel Projekt *Test & Performance Tools Platform Project (TPTP)* wurde 2002 unter dem Namen *Hyades* gegründet, später (2004) in *TPTP* umbenannt und zu diesem Zeitpunkt auch als Eclipse-Projekt aufgenommen. Es hat sich zum Ziel gesetzt, den kompletten Test- und Performance Lebenszyklus einer Software vom ersten Test bis in die Produktion zu begleiten.

TPTP ist in die vier Teilprojekte Plattform, Test, Trace und Profiling sowie Monitoring unterteilt. Zur Erstellung dynamischer Traces, die einen wesentlichen Anteil der erbrachten Lösung ausmachen, ist dabei in erster Linie das Teilprojekt *Trace und Profiling Tools* von Bedeutung. Es erweitert das Eclipse-Plattform Projekt um spezielle Datenkollektoren (Agenten) für Java und verteilte Anwendungen, die ihre Daten im so genannten *TraceModel* ablegen (siehe auch Abschnitt 2.4 „Eclipse Modelling Framework (EMF)“).

2.4 Eclipse Modelling Framework (EMF)

Das *Eclipse Modelling Framework (EMF)* [emfProject] ist ein Eclipse Subprojekt, welches das Design und die Erstellung strukturierter Datenmodelle erleichtert. Es handelt sich insofern um ein Infrastrukturprojekt, welches selten direkt verwendet wird. Viele andere Eclipse Projekte, wie auch das TPTP, setzen vielmehr auf diese Basisdienste auf.

Das EMF-Projekt stellt eine Reihe von Werkzeugen zur Verfügung [emfRedBook2004], wie ein Metamodell zur Beschreibung von neuen Modellen, eine Laufzeitumgebung zur Benachrichtigung interessierter Empfänger bei Änderungen des Modells, eine Persistenzunterstützung zur Serialisierung dieser Modelle in einem XML-Format und eine API zur generischen Erzeugung neuer und Manipulation bestehender Modelle.

Das vom EMF mitgelieferte Metamodell heißt *Ecore* und ist vom *Meta Object Facility (MOF*) Konzept der *Object Management Group (OMG)* abgeleitet. Das TPTP baut auf diesem Metamodell vier neue Modelle zur Speicherung der während eines *ProfilingSession* entstehenden Daten auf. Dies sind das Log-, Statistical-, Test- und das *TraceModel*, welches insbesondere für die Erzeugung dynamischer Traces von hoher Relevanz ist.

2.5 JUnit

JUnit ist ein leicht gewichtiges Framework, welches die Testautomatisierung auf Unit-Ebene (Methoden, Klassen) unterstützt und von Kent Beck und Erich Gamma entwickelt wurde [jUnitOrg].

Das Framework wurde in den gängigsten Entwicklungsumgebungen wie Eclipse, Netbeans und IntelliJ als Plug-In integriert. Viele Dialoge (Wizards) unterstützen den Benutzer beispielsweise bei der Erstellung von Testklassen. Die Dialoge untersuchen dabei in der Regel vorher die *KlasseUnterTest* und schlagen alle testbaren (also sichtbaren) Methoden vor, für die eine neue Testmethode angelegt werden soll.

Einmal erstellt, sollten die Tests eines Projektes möglichst automatisiert und so häufig wie möglich - idealerweise sogar bei jedem build-Vorgang - aufgerufen werden. Der Vorteil dieser Methode, auch *Continuous Integration* genannt, liegt darin, dass die Implementierung regelmäßig gegen die Spezifikation geprüft wird. Dabei gilt: je häufiger die Tests laufen, desto kleiner sind die entstehenden Deltas zwischen dem aktuellen und dem letzten Build. Je kleiner das Delta, desto schneller sind Fehler gefunden, die einen Testfall fehlschlagen lassen.

2.6 Statische Programmanalyse

EzUnit1 gewinnt alle Informationen darüber, welche Programmmethoden durch den Aufruf einer Testmethode aufgerufen werden und damit potentielle Verursacher eines Fehlers sind durch statische Programmanalyse (oder auch statische Traces). Die statische Programmanalyse analysiert - im Gegensatz zur dynamischen Programmanalyse - ein Programm durch „Durchlesen" des Quelltextes. Kontrollstrukturen wie Bedingungen und Schleifen bleiben dabei unbeachtet.

Um diese Analyse performant genug durchführen zu können, wird der Quelltext häufig in eine Zwischendarstellung, den so genannten Abstrakten Syntaxbaum *(engl.: Abstract Syntax Tree, AST)*, überführt. Ein abstrakter Syntaxbaum ist ein Objektnetzwerk, welches ein exaktes syntaktisches Abbild des zu Grunde liegenden Quellcodes darstellt. Im Gegensatz zur „normalen" Textdatei lässt sich der Syntaxbaum erheblich effizienter bearbeiten.

Auch die Eclipse-Plattform stellt eine Implementierung des *AST* für die Programmiersprache Java zur Verfügung. Wie Abbildung 1 zeigt, wird ein *AST* durch das Parsen eines Quellcode-Dokumentes erzeugt. Ein entsprechender Parser wird mit Hilfe der `createParser()` Fabrik-Methoden der Klasse *ASTParser* bereitgestellt.

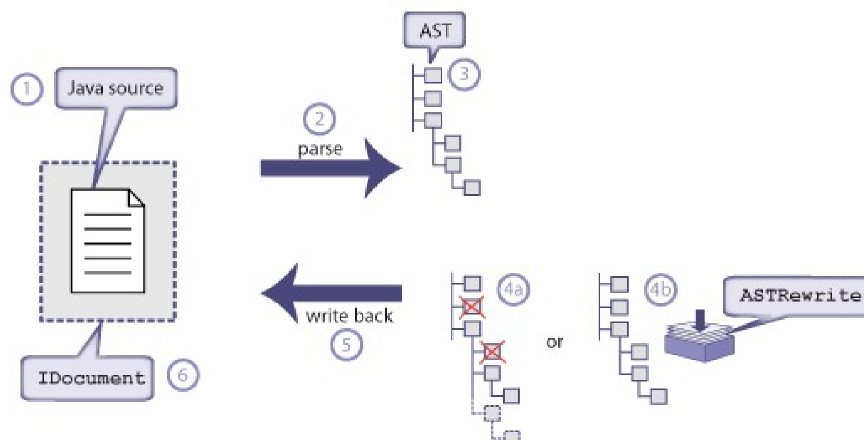


Abbildung 1 ~ Erzeugung eines AST (Abbildung nicht endgültig)

Jeder Knoten des *AST* wird durch ein *ASTNode* Objekt repräsentiert, wobei es für jeden Knoten des Java Modells (wie *ICompilationUnit*, *IMethod*) eine spezialisierende Klasse des *ASTNode* (*CompilationUnit*, *MethodDeclaration*) im *AST* gibt.

Zur Erzeugung eines statischen Traces wird der Syntaxbaum durchsucht. Es kommt dabei das sogenannte Besucher-Entwurfsmuster (*engl.: Visitor-Pattern)* zum Einsatz, welches von der abstrakten Klasse *ASTVisitor* implementiert wird. Bei der Erstellung der Traces wird dieser Besucher nun - beginnend mit den Wurzelknoten des jeweiligen (Teil)Baumes - jedem

weiteren Kindknoten des *AST* übergeben. Entsprechend der Implementierung des konkreten *ASTVisitors* können auf diese Weise für jeden besuchten Knoten Informationen gesammelt (wie z.B. der besuchte Methodenname) oder Manipulationen durchgeführt werden (wie z.B. das Löschen eines Knoten).

EzUnit1 implementiert die Klasse *StaticTraceASTVisitor*, die eine Subklasse der Klasse *ASTVisitor* ist. Zur Erzeugung des statischen Traces sind die beiden Methoden `visit(CallInstanceCreation)` und `visit(MethodInvocation)` überschrieben.

Für eine detaillierte Beschreibung der einzelnen Klassen und ihres Zusammenwirkens sowie eine Diskussion der bei dieser Art der Analyse auftretenden Besonderheiten (wie die Analyse von Vererbungsstrukturen und Reflection) sei auf [Meyer2007] verwiesen.

3 Konzept des FaultLocators

Zentrales Ziel von *EzUnit* ist, die Anzahl der Methoden zu reduzieren, die für das Fehlschlagen eines Testfalles verantwortlich sein könnten. EzUnit analysiert dazu Ausführungstraces von *JUnit*-Testfällen. Jede Art der Analyse der Testfälle und der von ihr aufgerufenen Methoden wird in diesem Kontext als *FaultLocalization* bezeichnet. Jede Klasse, die eine Lokalisierung durchführt, heißt *FaultLocator*.

Der Ablauf der Lokalisierung ist wie folgt:

1. Testlauf mit *EzUnit* Unterstützung (eingeschaltetem Tracing) starten
2. Während des Testlaufs Daten für Ausführungstraces sammeln
3. Nach Beendigung des Testlaufs *FaultLocatorManager* starten
4. Ausführungstraces für alle aufgerufenen Testmethoden erzeugen
5. Ersten *FaultLocator* mit der Liste aller Testmethoden starten
6. Ergebnismarker setzen
7. Solange es noch weitere Lokatoren gibt, zurück zu 5.)
8. Alle angeschlossenen Views aktualisieren
9. Lokalisierung beenden

Jeder Lokator implementiert die Schnittstelle *IFaultLocator* (siehe Listing 1), die zwei Parameter vom Typ *IProgressMonitor* und *List<ITestMethod>* aufnimmt. Die Liste *testMethods* enthält alle Testmethoden, die während des letzten Testlaufs aufgerufen wurden. Die für die Lokalisierung nötigen Ausführungstraces bezieht der Lokator am *TestMethodStore*.

```
public interface IFaultLocator {
	public abstract void process(IProgressMonitor monitor,
			List<ITestMethod> testMethods) throws JavaModelException;
}
```

Listing 1 ~ Das Interface *IFaultLocator*

Als Proof of concept stellt *EzUnit2* zwei Lokatoren zur Verfügung, die sich im Wesentlichen mit der Vollständigkeit der gesetzten *MUT*-Annotationen beschäftigen. Der *CalledButNotAnnotatedFaultLocator* weist den Benutzer darauf hin, dass der Ausführungstrace der jeweiligen Testmethode Methodenaufrufe enthält, die nicht in den Annotationen vermerkt sind. Das hätte zur Folge, dass *EzUnit* die jeweilige Methode nicht als möglichen Verursacher für das Fehlschlagen des Testes markieren kann. Umgekehrt dient der *AnnotatedButNeverCalledFaultLocator* dazu, Annotationen aufzudecken, die auf Methoden verweisen, die nicht im Ausführungstrace einer Testmethode enthalten sind. Solche Annotationen könnten dazu führen, dass der Benutzer auf die falsche Fährte geführt wird.

Die Ergebnisse der Lokatoren werden - wie auch schon in *EzUnit1* - mit Hilfe von Markern repräsentiert, die am linken Rand des Editors mit einem gelben (statt roten) T-Symbol visualisiert sind.

4 Implementierung

Dieses Kapitel beschreibt alle im Rahmen dieser Abschlussarbeit entwickelten Erweiterungen des *EzUnit2* Plug-In. Die Erweiterungen sind gegliedert in solche, die gänzlich neu zu implementieren waren (4.1 „Erweiterung durch Neuimplementierung“) und die, die auf Basis existierender Eclipse-*ExtensionPoints* (4.2 „Erweiterungen durch Integration“) entwickelt wurden.

4.1 Erweiterung durch Neuimplementierung

Die folgenden Abschnitte beschreiben den Teil der Erweiterungen, die auf Neuentwicklungen beruhen.

4.1.1 Dynamische Programmanalyse

In Erweiterung zur statischen Programmanalyse von *EzUnit1* implementiert *EzUnit2* eine einfache Form der dynamischen Programmanalyse, das Tracing.[6] Im Gegensatz zur statischen Programmanalyse werden Traces zur Laufzeit des Programms erstellt. Auf diese Weise kann ermittelt werden, welche Teile des Quelltextes tatsächlich ausgeführt werden. Neben Schleifen- und Kontrollstrukturen werden somit auch Methodenaufrufe erkannt, die erst durch dynamische Bindung (Reflection) zustande kommen. Die Verwendung des dynamischen Verfahrens, verspricht daher eine Reduktion der möglichen Verursachermethoden eines fehlgeschlagenen Testfalls.[7]

Als Basisframework zur Erzeugung dynamischer Traces wurde das TPTP Framework ausgewählt[8], welches in erster Linie auf die Analyse des Laufzeitverhaltens eines Programms (Profiling) spezialisiert ist [tptpProject]. In seiner ursprünglichen Implementierung ist auch keine direkte Programmierschnittstelle zum Auslesen eines Traces vorhanden. Daher wurden im Rahmen dieser Arbeit die zum Zwecke des Profilings gesammelten Daten neu interpretiert und daraus der dynamische Trace gewonnen.

Ein dynamischer Trace kann nur dann angezeigt werden, wenn dynamische Profilingdaten vorhanden sind. *EzUnit2* sammelt diese Profilingdaten während einer *ProfilingSession*, wenn die zu untersuchenden JUnit-Tests im *Profiling Modus* ausgeführt werden. Um die *MUTSelectionView* mit der Liste möglicher Verursacher zu füllen, muss dieser *Profiling Modus* im Hintergrund (seamless) automatisch gestartet werden.

In den folgenden Abschnitten wird die Integration des TPTP Frameworks näher beleuchtet. Abschnitt 4.1.1.1 führt in das programmgesteuerte Starten des Tracing Mechanismus (Profiling) ein. Dann erläutert Abschnitt 4.1.1.2 wie das TPTP Framework mit Hilfe von Datenkollektoren die Profilingdaten vom beobachteten Programm zum aufrufenden Programm (dem JUnit-Test) überträgt. Im weiteren Verlauf zeigt Abschnitt 4.1.1.3 wie der Zugriff auf die im Datenmodell (*TraceModel*) gesammelten Daten erfolgt. Anschließend zeigt Abschnitt 4.1.1.4

6 Weitere Formen der dynamischen Programmanalyse wären Ausführungszeitanalysen (Execution Time Analysis) oder Analysen des Speicherverhaltens (Memory Analysis).

7 Es hat sich gezeigt, das das Verhältnis der gefundenen *MUTs* zwischen statischem und dynamischem Trace etwa 3 zu 1 ist. Abschnitt 6.2 „Tracegröße“ gibt hierüber detaillierte Auskunft.

8 Weiterer Tracingansätze werden in Abschnit 7.1 „Alternative Tracingansätze“ beleuchtet.

aufgetretene Nebenläufigkeitsprobleme auf und schließlich gibt Abschnitt 4.1.1.5 Aufschluss darüber, wie einmal erzeugte Datenmodelle auch wieder aus dem Speicher der Entwicklungsumgebung entfernt werden können.

4.1.1.1 Programmgesteuertes Starten des TPTP Profilers

Zum Starten externer Programme bietet die Eclipse-Plattform das so genannte *LaunchFramework*, ein generisch erweiterbares Framework, an. Die Klasse *ILaunchConfigurationType* ist in diesem Kontext von zentraler Bedeutung. Sie repräsentiert eindeutig die Art und Eigenschaften des Programms, das über die Plattform gestartet werden soll. Eine Standard-Eclipse-IDE -Installation enthält beispielsweise die Typen:

- `org.eclipse.jdt.launching.localJavaApplication` zum Starten einer „normalen" Java Applikation
- `org.eclipse.jdt.junit.launchconfig` zum Ausführen einer JUnit Testsuite

Die Instanz eines *ILaunchConfigurationType* heißt *ILaunchConfiguration*. Sie hält eine Liste von Schlüssel-Wert Paaren, die eine konkrete Konfiguration eines LaunchTypen beschreiben. Eine Besonderheit ist, dass die Schnittstelle *ILaunchConfiguration* keine Methoden zur Änderung ihrer Konfiguration anbietet, da LaunchConfigurations als unveränderlich (engl.: immutable) definiert sind. Zur Anpassung einer LaunchConfiguration an die individuellen Bedürfnisse ist daher zunächst eine Arbeitskopie (engl.: WorkingCopy) zu erzeugen. Diese ist dann vom Typ *ILaunchConfigurationWorkingCopy* zu erzeugen und bietet die nötigen Methoden zu ihrer Manipulation an.

```
1: private ILaunchConfiguration createLaunchConfiguration() throws CoreException {
2:        ILaunchManager manager =
3:                DebugPlugin.getDefault().getLaunchManager();
4:        ILaunchConfigurationType type =
5:        manager.getLaunchConfigurationType("org.eclipse.jdt.junit.launchconfig");
6:        ILaunchConfigurationWorkingCopy wc =
7:                type.newInstance(null, LAUNCHCONFIG_NAME);
8:        [...]
9:        wc.setAttribute("org.eclipse.jdt.launching.MAIN_TYPE",
10:               context.getDeclaringType().getFullyQualifiedName());
11:       wc.setAttribute("org.eclipse.jdt.launching.PROJECT_ATTR",
12:               context.getJavaProject().getElementName());
13:       wc.setAttribute("org.eclipse.hyades.trace.ui.ATTR_AUTO_FILTER_CRITERIA",
14:               Boolean.FALSE);
15:       wc.setAttribute("org.eclipse.hyades.trace.ui.ATTR_FILTER_SET",
16:               TraceManager.FILTERSET_NAME);
17:       [...]
18:       return wc.doSave();
19: }
```

Listing 2 ~ Auszug, Erstellung *ILaunchConfiguration*

Das Wissen darüber, wie ein externes Programm vom Typ *ILaunchConfigurationType* mit der Konfiguration *ILaunchConfiguration* zu starten ist, enthält das *ILaunchConfigurationDelegate*. Es implementiert insbesondere die `launch()` Methode und sorgt darin unter anderem für die Erzeugung eines neuen Java-Prozesses. Die gestartete Session wird durch ein *ILaunch* Objekt repräsentiert.

Zum Starten einer *ProfilingSession* werden also folgende Schritte durchlaufen:

1. Beziehen einer Referenz auf den LaunchConfigurationType mit dem Schlüssel `org.eclipse.jdt.junit.launchconfig` (*Listing 2, Zeile 2-5*)
2. Erzeugen einer LaunchConfigurationWorkingCopy (*Listing 2 Zeile 6*)
3. Setzen der für diesen LaunchType notwendigen SchlüsselWert-Paare (*Listing 2, Zeilen 9-16*)
4. Starten des Programms durch Aufruf von `launch()` (*Listing 3, Zeile 2*)

```
1:        ILaunchConfiguration config = createLaunchConfiguration();
2:        ILaunch launch = config.launch(
3:                 ILaunchManager.PROFILE_MODE, new SubProgressMonitor(monitor, 1));
```

Listing 3 ~ Starten einer *LaunchConfiguration*

Fortan übernimmt das *LaunchConfigurationDelegate* dieses *ILaunchConfigurationType* die Kontrolle über das Starten des Programms. Rückmeldungen über den Fortschritt des Startvorgangs werden an den übergebenen *IProgressMonitor* weitergeleitet. Sobald das externe Programm[9] gestartet ist, also ein neuer Java-Prozess erzeugt wurde, kehrt die Methode `launch()` mit einer Referenz auf das gerade gestartete Programm zurück.

4.1.1.2 Datenkollektoren

Immer dann, wenn von einem Plug-In der Eclipse-Plattform Daten in strukturierter Form gesammelt und persistiert werden sollen, sollte auf die Infrastruktur des EMF zurückgegriffen werden [emfProject]. Auch das Trace- und Profiling Tools-Projekt speichert die während einer *ProfilingSession* anfallenden Daten mit Hilfe seines so genannten *TPTP Data Collections Framework* in einer Instanz des ECore-Modells, dem *TraceModel* ab.

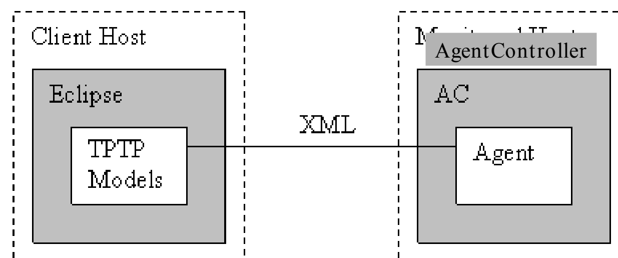


Abbildung 2 ~ vereinfachte Architektur des *DataCollection Framework*

Abbildung 2 (aus [howtoDatacollector]) zeigt eine stark vereinfachte Darstellung des *Data-Collection-Framework*. Gut erkennbar sind die drei Hauptkomponenten des Frameworks:

1. *Der AgentController* ~ ist ein Prozess, welcher auf dem zu beobachtenden Zielsystem läuft. Er kontrolliert den Lebenszyklus und die Kommunikation mit den ebenfalls dort installierten Agenten. Dabei vermittelt er Steuernachrichten von Client an Agent und umgekehrt, ohne Wissen über deren Syntax oder Semantik zu haben.
2. *Der Agent* ~ ist eine Applikation, die Clients ihre Dienste über einen *AgentController* anbietet. Typischerweise sammeln Agenten Daten vom/über das Zielsystem, auf dem sie laufen. Es können aber auch Services wie das Starten entfernter Applikationen oder das Kopieren von Dateien angeboten werden.

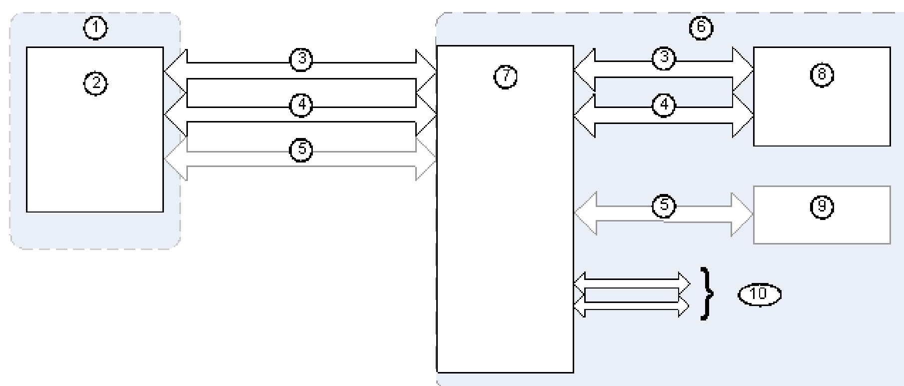
9 In unserem Fall wird eine JVM gestartet, die ihrerseits den *JUnitTestRunner* ansteuert.

3. *Der Client* ~ ist eine Applikation, die sich die gesammelten Daten zu Nutze macht. Ein Client kann hierzu mit mehreren *AgentControllern* verbunden sein und somit Dienste von vielen Agenten gleichzeitig in Anspruch nehmen.

Sichtbar werden die Agenten im Client, also der Workbench auf dem Reiter „Monitor“ der „Profiling LaunchConfiguration“ (siehe auch Kapitel 5 „Installation und Bedienung“).

In älteren Versionen des TPTP-Projektes war es notwendig, den *AgentController* nebst gewünschten Agenten auf dem Zielsystem unabhängig vom Clienten zu installieren (*Standalone AgentController*). Inzwischen (seit der Version 4.2) wird aber ein interner *AgentController* zur Vereinfachung der Handhabung angeboten. Damit sind nach der Installation aller benötigten Plug-Ins keine weiteren Schritte zum Starten einer *ProfilingSession* mehr erforderlich.

Zur weiteren Verdeutlichung der Rolle der einzelnen oben beschriebenen Komponenten des DataCollection Framework sei nachfolgend ihr Kommunikationsfluss untereinander verdeutlicht (siehe Abbildung 3):



#	Beschreibung	#	Beschreibung
1	Quellsystem	6	Zielsystem
2	Client (z.B. Eclipse Workbench)	7	AgentController
3	Steuerkanal	8	Agent
4	Datenkanal	9	zu beobachtende Applikation
5	Konsolenkanal	10	Einbindung weiterer Agenten

Abbildung 3 ~ Kommunikationsfluss des DataCollection Framework

Es wird deutlich, dass während einer *ProfilingSession* über den Datenkanal Tracedaten beim Client eintreffen und von ihm im *TraceModel* gespeichert werden. Im nächsten Schritt gilt es nun, die gesammelten Daten zu analysieren und aus ihnen einen Ausführungstrace zu erzeugen.

4.1.1.3 TraceModel und QueryEngine

Das *DataCollection* Framework verwendet die Infrastruktur des EMF-Projektes und sammelt die Ausführungsdaten im so genannten *TraceModel*, einer Instanz des Ecore Metamodells, baumartig. Zugriff auf das *TraceModel* erhält man beispielsweise über die Referenz eines *TRCAgent* Objektes. Sie ist der Startknoten in dem Suchbaum, der der QueryEngine zusammen mit dem Methoden- und dem Klassennamen der Testmethode, für die ein Trace erzeugt werden soll, übergibt.

Abbildung 4 zeigt die für die Erstellung dynamischer Traces (ExecutionTrace) relevanten Entitäten des *TraceModels*. Die Klasse *TRCMethodInvocation* repräsentiert in diesem Modell genau einen Aufruf der Methode einer Klasse. Jedes *TRCMethodInvocation* Objekt hält eine Referenz auf seinen Besitzer[10]. Die Klasse *TRCFullMethodInvocation* und die Klasse *TRCAggregatedMethodInvocation* enthalten jeweils ausführliche statistische Daten (Anzahl der Aufrufe, verbrauchte Prozessorzeit, uvm.) zu einzelnen Methodenaufrufen.

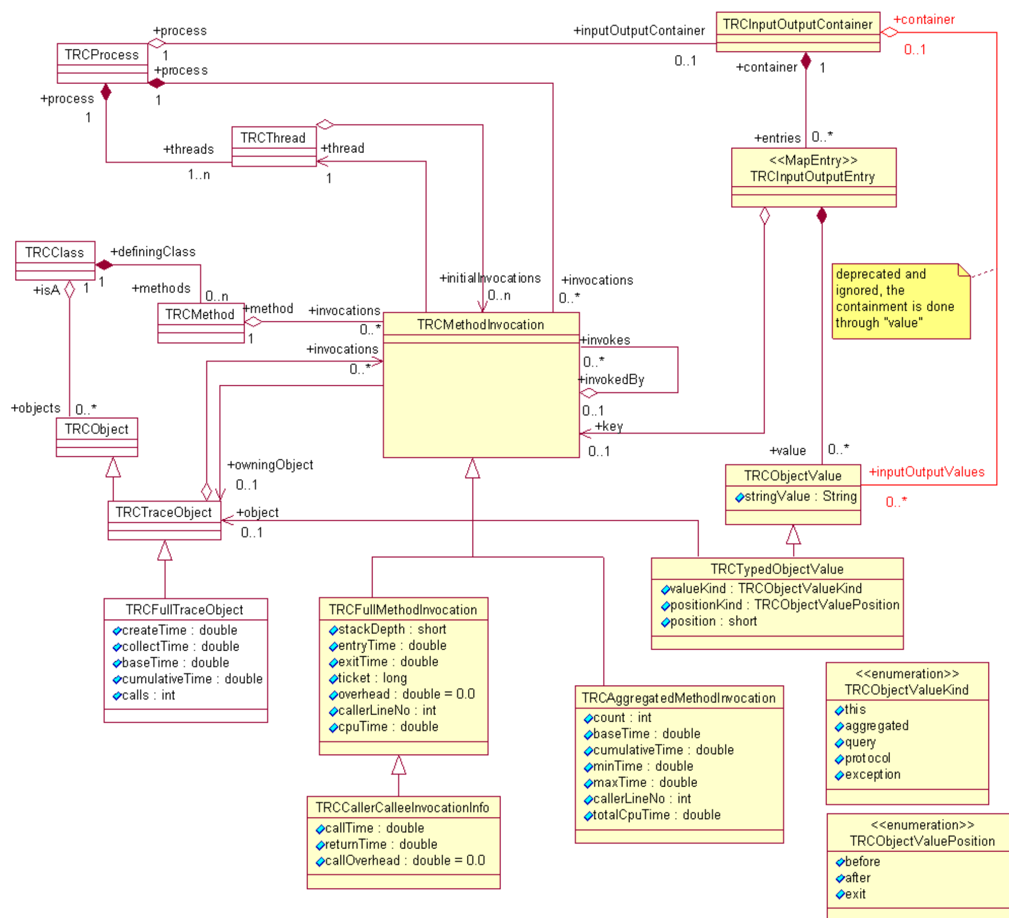


Abbildung 4 ~ Klassendiagramm des TraceModel (Auszug) [emfUmlModelle]

Zur Erstellung eines dynamischen Traces ist das oben dargestellte *TraceModel* (Abbildung 4) beginnend mit der Testmethode des JUnit-Tests rekursiv zu durchlaufen. Die von einer Methode wiederum aufgerufenen Methoden werden über den Aufruf der Methode `getInvokees()` der Klasse *TRCMethodInvocation* ermittelt und anschließend in den *MethodUnderTestStore* übertragen.

Um das Auffinden der Testmethode im *TraceModel* zu erleichtern und auch hinreichend effizient zu gestalten, liefert das EMF Projekt eine *QueryEngine* mit. Generell erfolgt der Zugriff auf das *TraceModel* über die Referenz eines Objektes der Klasse *TRCAgent*, das seinerseits eine Referenz auf die Klasse *TRCProcess* hält (siehe Zeile 4, Listing 4). Mit dieser Referenz wird dann ein Suchausdruck erzeugt. Als Suchkriterium (ähnlich einer SQL-WHERE-

10 Diese Referenz entspricht dem *this*-Objekt für diesen Methodenaufruf und ist in einem *TRCTraceObject* gekapselt.

Klausel) dient der Methoden- und Klassenname der übergebenen Testmethode. Mögliche Parameter werden bei der Suche nicht betrachtet[11].

```
private synchronized void traverseModel(TRCMethodInvocation invoker,
                ITestMethod testMethod) {
        EList<TRCMethodInvocation> invokees = invoker.getInvokes();
        [...]

        if (invokees != null && invokees.size() > 0
                        && stackdepth < STACKDEPTH_LIMIT) {
                for (TRCMethodInvocation invokee : invokees) {
                        IMethod invokeeAsIMethod = HelperMethods.getIMethod(invokee
                                        .getMethod());
                        if (invokeeAsIMethod != null) {
                                IMethodUnderTest mut = MethodUnderTestStore.getInstance()
                                                .findMut(invokeeAsIMethod);
                                mut.setTRCMethod(invokee.getMethod());
                                mut.addCallingTestMethod(testMethod);
                                mut.setPartOfDynmaicTrace();
                                this.addElement(mut);
                        }

                        // add more invokees recursively
                        traverseModel(invokee, testMethod);
                }
        }
}
```

Listing 4 ~ Suchmethode zum Auffinden einer IMethod im TRCModel

Sollte die Suche über Methoden- und Klassenname kein Ergebnis liefern, kann es sich um eine abgeleitete Testklasse handeln. Die Implementierung zu einer bestimmten Testmethode kann also auch in ihrer Superklasse zu finden sein. In diesem Fall baut EzUnit die Superklassen-Hierarchie (Listing 5) auf und sucht bis zum Erreichen der Superklasse *java.lang.Object* nach einer Implementierung der gesuchten Testmethode.

```
IType[] superTypes = declaringType.newTypeHierarchy(null).getAllSupertypes(declaringType);
for (int typeIndex = 0; typeIndex < superTypes.length; typeIndex++)
{
        resultEntry = createAndExecuteQuery(agent, testMethod,
                superTypes[typeIndex].getElementName(),
                superTypes[typeIndex].getPackageFragment().getElementName());
        // if we got a result, we stop walking through the
        // hierarchy
        if (resultEntry.size() > 0) {
                break;
        }
}
```

Listing 5 ~ Aufbau der Supertyp-Hierarchie

Wurde die Testmethode schließlich gefunden, startet von dort aus der rekursive Aufbau des dynamischen Traces entlang der Methoden, die `getInvokee()` zurückgibt (Listing 4).

Bei der Rückkonvertierung der *TRCMethod*- in *IMethod*-Objekte ist Vorsicht geboten. Die Notation der beiden Methodensignaturen unterscheidet sich und kann nicht ohne weitere Bearbeitung konvertiert werden. Bei der Konvertierung wird die Signatur des *TRCMethod*

11 Die Suchkriterien „Methoden- und Klassenname“ sind zum Finden der Testmethoden dennoch hinreichend genau, weil sie gemäß der Spezifikation des JUnit-Frameworks keine Parameter haben dürfen.

Objektes zunächst in seine Einzelteile, die so genannten Parameter, zerlegt. Dann werden die Parameter-Arrays der beiden Objekte verglichen, um schließlich mit der Hilfsmethode *Signature.toString()* konvertiert zu werden.

4.1.1.4 Nebenläufigkeiten und Asynchronität

Sowohl die Erstellung statischer als auch dynamischer Traces dauert eine gewisse Zeit. Statische Traces sind in durchschnittlich einer Sekunde erstellt[12]. Die Erstellung des dynamischen Traces dauert etwa zehn Sekunden[13]. Sollte der *Internal AgentController* (IAC) noch nicht gestartet sein, sind hierfür weitere vier Sekunden zu kalkulieren.

Während dieser Wartezeit ist es wichtig, dass die Benutzerschnittstelle weiterhin Aktionen entgegennehmen und andere Aufgaben, wie beispielsweise build, reconcilation, etc. durchführen kann, sie also in diesem Sinne „ansprechbar“ (engl.; responsive) bleibt [eclipse-von2004Ui]. Diese Form der Ansprechbarkeit (engl.: *Responsiveness*) kann dadurch erreicht werden, dass lang laufende Operationen in eigenen Programmfäden statt im Hauptprogrammfaden (*UIThread*/*MainThread*) gestartet werden. Mit der Version 3 der Eclipse-Plattform wurde daher die Architektur zur Verarbeitung von Nebenläufigkeiten, die *concurreny architecture*, überarbeitet.

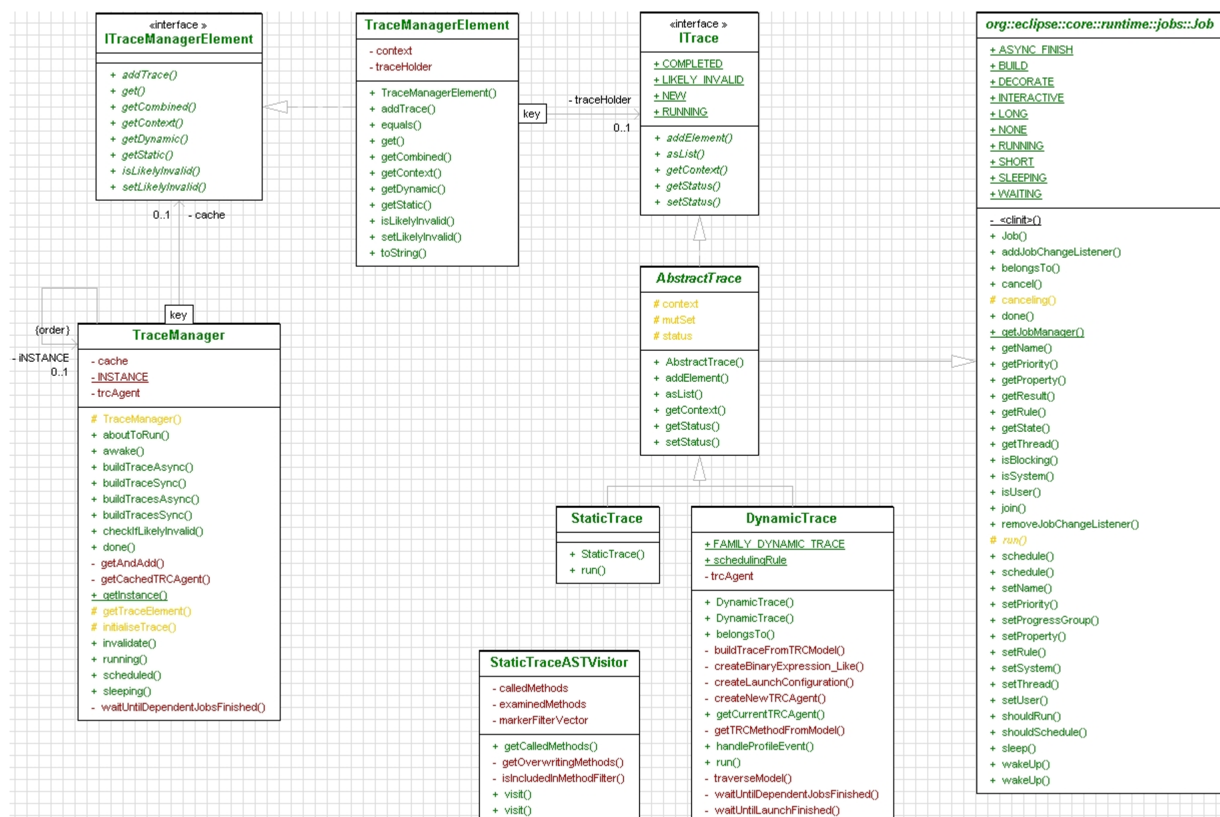


Abbildung 5 ~ Klassendiagramm des Paketes „trace“

Als zentrale Entität wurden die so genannten *Jobs* eingeführt. Ein *Job* repräsentiert eine Einheit asynchroner Operationen, die gleichzeitig mit anderen Jobs ausgeführt werden sollen [help-concurrency]. Die API wurde indes soweit vereinfacht, dass individuell erstellte Jobs

12 Das Kapitel Ergebnisse 6.1 „Zeitverhalten bei der Trace Erstellung“ zeigt eine detaillierte Übersicht über das Zeitverhalten bei der Tracerstellung.

13 inklusive einer vollständigen *ProfilingSession* der *JUnitTestSuite* der Klasse MoneyTest

(die lediglich eine `run()`-Methode implementieren müssen) durch den Befehl `schedule()` an den im Hintergrund laufenden *JobManager* übergeben werden. Dieser übernimmt fortan die Kontrolle über die Ausführung und die Einhaltung etwaiger Abhängigkeiten zu anderen Jobs. Auch EzUnit bedient sich dieser Infrastruktur zur Erstellung seiner Traces. Wie das Klassendiagramm (Abbildung 5) zeigt, sind die EzUnit Traces als Spezialisierungen der Klasse *Job* implementiert.

Die Klasse *StaticTrace* kapselt die Erzeugung statischer Traces, wobei die Implementierung der `run()`-Methode im Wesentlichen den Aufruf an den *StaticTraceASTVisitor* delegiert. Dieser extrahiert, beginnend mit der TestMethode `context()`, die *StaticTrace* (siehe Listing 6, Zeile 7) bei der Erzeugung übergeben wird, aus dem *AST* die aufgerufenen Methoden.

```
1: [...]
2: StaticTraceASTVisitor visitor = new StaticTraceASTVisitor();
3: MethodDeclaration methodDeclaration =
4:        ASTHelperMethods.getMethodDeclarationFromIMethod(context);
5:
6: if (methodDeclaration != null)
7:        methodDeclaration.accept(visitor);
8: [...]
```

Listing 6 ~ Delegation der Traceerzeugung an den StaticTraceASTVisitor

Die Erstellung des dynamischen Traces ist erheblich komplizierter. Die Komplexität ergibt sich in erster Linie aus der notwendigen Verwendung verschiedener Programmfäden, die zum Start eines neuen Java Prozesses erforderlich sind. Diese Programmfäden werden durch das TPTP Framework im Hintergrund erzeugt (siehe Abschnitt Datenkollektoren). Abbildung 6 listet die an der Erzeugung des Traces beteiligten Objekte auf und deutet deren Aufgaben während seiner Erstellung an.

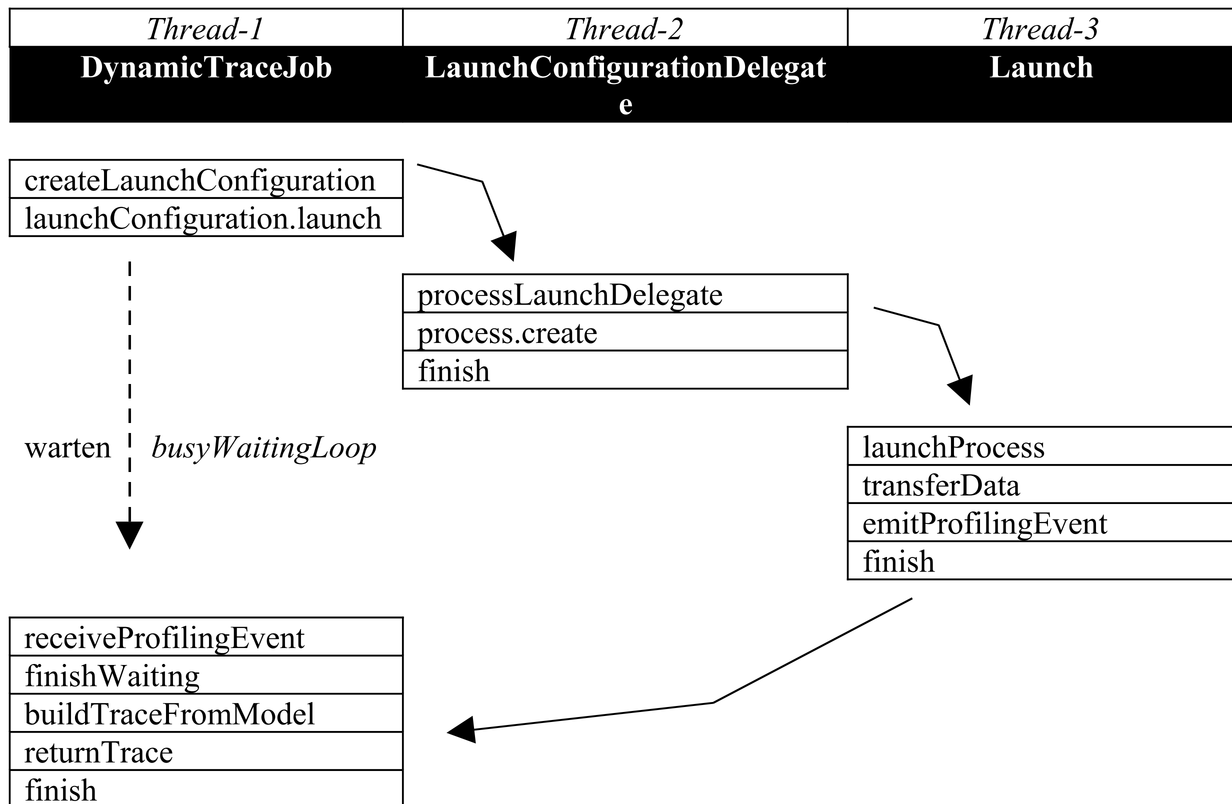


Abbildung 6 ~ Beteiligte Objekte an der Trace Erstellung

In vorigen Abschnitt wurde beschrieben, dass die Erstellung eines dynamischen Traces auf der Analyse des *TRCModels* basiert. Der Zugriff auf dieses Datenmodell kann beispielsweise über die gültige Referenz eines *TRCAgent* Objektes erfolgen. Sollte diese Referenz nicht vorliegen oder wurde die Erstellung eines neuen Traces erzwungen (*isLikelyInvalid*), startet der *DynamicTrace*-Job für den Benutzer unsichtbar eine neue *ProfilingSession*.

Für die Bereitstellung eines Traces ist der *TraceManager* (siehe Abbildung 5) zuständig. Diese als Singleton implementierte Klasse verwaltet aus Performancegründen einen Cache mit bereits erzeugten Traces und bietet darüber hinaus eine Reihe von Methoden zu ihrer Erzeugung an. Im Wesentlichen kann dabei zwischen zwei Arten, der synchronen und der asynchronen Erzeugung gewählt werden.

Die synchrone Erzeugung wird über die Methode `buildTraceSync()` angestoßen. Dieses Verhalten wird mit dem Aufruf der von *Job* geerbten `join()`-Methode (siehe Zeile 8, Listing 7) erreicht. Der aufrufende Programmfaden wird hier so lange blockiert, bis der *DynamicTraceJob* alle Befehle der Methode `run()` verarbeitet oder bis der Job unterbrochen und eine *InterruptedException* erzeugt wird[14].

14 Ein Abbruch eines Job's kann beispielsweise über den Aufruf der Methode `setCancelled()` der Klasse *Job* erreicht werden.

```
1:  [...]
2:  AbstractTrace trace =
3:     this.initialiseTrace(rebuild, context, traceType);
4:  trace.addJobChangeListener(this);
5:  trace.schedule();
6:
7:  try {
8:     trace.join();
9:  } catch (InterruptedException e) {
10:    e.printStackTrace();
11: }
```

Listing 7 ~ Auszug der Methode `buildTraceSync()`

Die synchrone Erzeugung wird insbesondere von den *FaultLocatoren* verwendet, die im Rahmen ihrer Untersuchung nacheinander die Traces aller Testmethoden aufrufen. Auf diese Weise werden Warteschlangen überflüssig, die ansonsten alle Trace-Anforderungen aufnehmen müssten, bis die jeweilige Verarbeitung beendet wäre.

Neben der synchronen Erzeugung ist auch die asynchrone Anforderung eines Traces implementiert. Diese Methode zielt in erster Linie auf die Versorgung der Views mit Tracedaten ab. Sobald erste Daten (in aller Regel die des statischen Traces) vorhanden sind, können diese auch schon angezeigt werden. Die Tabelle mit Tracedaten wird so immer weiter vervollständigt, und der Benutzer muss nicht unnötig lange auf Daten warten. Eine asynchrone Erzeugung wird dadurch erreicht, dass ein *DynamicTraceJob* erzeugt und dem Eclipse *JobManager* übergeben wird. Um weiterhin über den Status des Jobs auf dem Laufenden zu bleiben, wird ein *IJobChangeListener* beim Job registriert. Sobald die Verarbeitung beendet wird, wird die Methode `done()` des Listeners aufgerufen. Der *TraceManager* implementiert diese Schnittstelle und löst in diesem Fall die Aktualisierung aller registrierten Views aus.

Um Raceconditions und Blockaden zu vermeiden, muss sichergestellt werden, dass immer nur eine ProfilingSession gleichzeitig gestartet wird. Bei der Implementierung über das Job Framework sind dazu zwei Schritte erforderlich. Zunächst ist jeder Job der gleichen Job-Familie zuzuordnen. Dazu wird zunächst ein finales, statisches Objekt erzeugt, welches als Schlüssel für die jeweilige Familie dient (siehe Zeile 1, Listing 8). Dann ist die `belongsTo()`-Methode jedes dieser Familie zugehörigen Jobs so zu überschreiben, dass sie auf die Gleichheit mit dem oben genannten Schlüssel-Objekt prüft (siehe Zeile 4-6, Listing 8).

```
1: public static final Object FAMILY_DYNAMIC_TRACE = new Object();
2: [...]
3: @Override
4: public boolean belongsTo(Object family) {
5:     return family.equals(FAMILY_DYNAMIC_TRACE);
6: }
```

Listing 8 ~ Job Familie und Schlüssel-Objekt

Im zweiten Schritt ist darauf zu achten, dass der *TraceManager* vor der Erstellung jedes neuen dynamischen Traces sicherstellt, dass kein weiterer Job der Familie `FAMILY_DYNAMIC_TRACE` läuft, sondern auf dessen Beendigung gewartet wird. Erreicht wird dies durch die Synchronisierung mittels Aufruf der Methode auf `IJobManager.join()` auf die oben eingeführte Job-Familie (Listing 9).

```
1: [...]
2: final IJobManager jobManager = Job.getJobManager();
3: try {
4:     jobManager.join(DynamicTrace.FAMILY_DYNAMIC_TRACE,
5:                     new NullProgressMonitor());
6: } catch (InterruptedException e) {
7:     e.printStackTrace();
8: }
```

Listing 9 ~ Ausschnitt aus der Methode *waitUntilDependentJobsFinished()*

4.1.1.5 Automatisches Löschen/Entladen nicht mehr genutzter TraceModelle aus dem Arbeitsspeicher

Mit jeder *ProfilingSession* legt das TPTP Framework ein neues *TraceModel* im Arbeitsspeicher ab. Dieses Verhalten kann zwar für eine Analyse des Laufzeitverhaltens eines Programms sinnvoll sein, EzUnit2 benötigt diese Modelle, nachdem die dynamischen Traces erzeugt sind, allerdings nicht mehr[15].

Deshalb sollten alle nicht mehr benötigten *TraceModels* automatisch aus dem Arbeitsspeicher entfernt werden. Hierzu bietet die Programmierschnittstelle des TPTP Frameworks allerdings keine Methoden an. Problematisch ist überdies, den richtigen Zeitpunkt zum Löschen zu finden. Würde der falsche Zeitpunkt gewählt werden, würden mit dem Löschprozess Objekte gelöscht werden, auf deren Referenzen innerhalb des Trace Erstellungsprozesses noch zugegriffen wird. Auch der asynchrone Prozess *XMLDataProcessor* gibt keine Auskunft über seinen aktuellen Arbeitsstatus, so dass nicht zuverlässig ermittelt werden kann, wann kein Zugriff mehr auf das alte *TraceModel* erfolgt.

15 Zur Zeit implementiert EzUnit keinen Zugriff auf historische *TraceModels*, so dass sie ohne weiteres gelöscht werden können.

```
01: private void deletePreviousTrcModel(final TRCAgent trcAgent) {
02:     [...]
03:     // walk through the TRCxxx-Objekt Tree to find the list of
04:     // of TRCAgent's
05:     TreeIterator<Notifier> allContents =
06:             HierarchyResourceSetImpl.getInstance().getAllContents();
07:     while (allContents.hasNext()) {
08:             Notifier notifier = allContents.next();
09:             if (notifier instanceof TRCProcessProxy) {
10:                     TRCProcessProxy proxy = (TRCProcessProxy) notifier;
11:                     Iterator agents = proxy.getAgentProxies().iterator();
12:                     while (agents.hasNext()) {
13:                             TRCAgentProxy agent = (TRCAgentProxy) agents.next();
14:
15:                             long currentTime = System.currentTimeMillis();
16:                             long stopTime =
17:                                     Double.valueOf(agent.getStopTime()).longValue()*1000;
18:                             long delta = currentTime - stopTime;
19:
20:                             if (delta  > MIN_DELTA_BEFORE_DELETE) {
21:                                     if (!agent.isAttached() && !agent.isActive()) {
22:                                             deleteCandidates.add(agent.getProcessProxy());
23:                                     }
24:                             }
25:                             else {
26:                                     EzUnitPlugin.trace(this, "to young!
27:                                             you will be deleted next time! " + agent);
28:                             }
29:                     }
30:             }
31:     }
33:     [...]
34: }
```

Listing 10 ~ aus ProgrammaticLaunchHelper: Löschen/Entladen alter TraceModelle

In der Praxis hat sich herausgestellt, dass nach etwa zwei Sekunden nach Beendigung der *ProfilingSession* kein Zugriff mehr auf das *TraceModel* erfolgt. Dieses Alter wird beim Löschversuch mit betrachtet. Ist ein Modell zu „Jung“, wird es nicht gelöscht und vor der Erzeugung des nächsten Modells erneut überprüft. Listing 10 zeigt die Implementierung dieses Alterungsalgorithmus der Klasse *ProgrammaticLaunchHelper.*

Den Zugriff auf die im Speicher befindlichen Modelle (genauer gesagt, auf alle ModellObjekte) erhält man über den Singleton [GoF1996] *HierarchyResourceSetImpl.getInstance()*.

4.1.2 ExtensionPoint “FaultLocator”

Bei der Entwicklung der Eclipse-Plattform wurde großes Augenmerk auf ihre Erweiterbarkeit gelegt. Das Fundament zur Sicherstellung der Erweiterbarkeit bildet die so genannte *Platform Runtime*, die eine Implementierung des *OSGi Services Model* ist. Sie ist für das Starten der Basisdienste sowie den Aufbau und die Pflege der Plug-In Registratur zuständig. Jedes weitere Subsystem (in Abbildung 7 bezeichnet als „New Tool“) der Eclipse-Plattform ist in Form von Plug-Ins strukturiert, die ihren Anteil bestehend aus Quelltext, Dokumentation oder beidem zur Gesamtfunktionalität der Plattform beitragen.

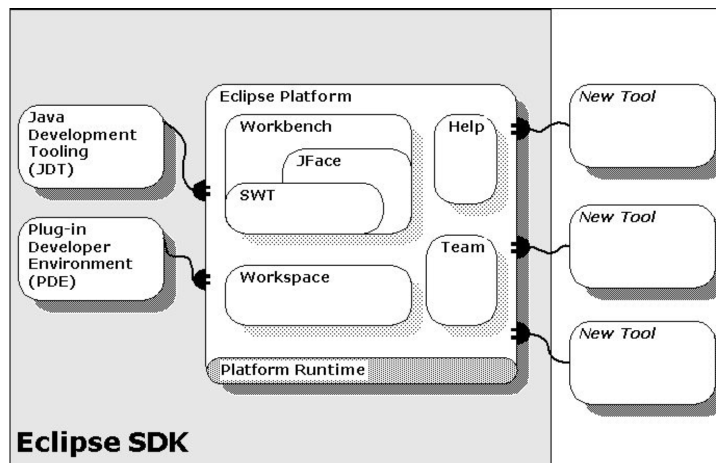


Abbildung 7 ~ Schema der Eclipse Plattform Architektur

Das Kernkonzept für die oben beschriebene Erweiterbarkeit ist das so genannte *Eclipse Extension System*, in dem jedes Plug-In zwei Rollen einnehmen kann. Durch Beschreibung seiner öffentlichen Schnittstelle, den so genannten *ExtensionPoint* in Form einer XML-Schema-Datei, nimmt es die Rolle eines Dienstleisters ein. Die Laufzeitumgebung der Eclipse-Plattform propagiert diese Schnittstelle schließlich in einer entsprechenden Registratur und gewährleistet damit die Auffindbarkeit. In seiner zweiten Rolle nutzt das Plug-In bestehende *ExtensionPoints* durch Registrierung so genannter *Extensions*. Übertragen auf die Elektrik stellen *Extensions* die Stecker dar, die in die *ExtensionPoints* (Steckdosen) gesteckt werden können.

Auch EzUnit nutzt diesen Plug-In Mechanismus, indem es einige Extensions (View, Preferences, Listener) deklariert. Darüber hinaus wird auch ein eigener ExtensionPoint mit dem Identifiklator `org.projectory.ezunit.faultLocator` angeboten. Mit dem Interface *IFaultLocator* wird das in EzUnit1 umgesetzte Konzept der Fehlerlokalisierung auf Basis von Traces verfeinert und „externen" Nutzern zugänglich gemacht.

Beispielhaft wurden die beiden Lokatoren *CalledButNotAnnotatedFaultLocator* und *AnnotatedButNeverCalledFaultLocator* über den *ExtensionPoint* angebunden. Listing 11 zeigt die entsprechenden Konfigurationseinträge in der Datei *plugin.xml*.

```
<extension id="faultLocator" name="FaultLocator"
point="org.projectory.ezunit.faultLocator">
  <faultLocator faultLocatorClass="org.projectory.ezunit.
     internal.faultlocator.annotation.AnnotatedButNeverCalledFaultLocator"/>
  <faultLocator faultLocatorClass="org.projectory.ezunit
    .internal.faultlocator.annotation.CalledButNotAnnotatedFaultLocator" />
</extension>
```

Listing 11 ~ Deklaration der Beispiellokatoren

Der Zugriff auf das *ExtensionSystem* erfolgt über das Fassadenobjekt *Platform* (siehe Listing 12), welches die Aufrufe an die entsprechenden internen Klassen weiterleitet und ein Objekt mit der Schnittstelle *IExtension* zurückgibt. Als Suchschlüssel wird dabei der Wert verwendet, der in der Datei *plugin.xml* als `extensionid` vergeben wurde. Aus Vereinfachungsgründen wurde in beiden Fällen der Schlüssel und der Name des *ExtensionPoints* gleichgesetzt.

```
IExtension extension =
        Platform.getExtensionRegistry().getExtension("org.projectory.trace.faultLocator");

// transfer all 'evaluator' elements to a special list to get the total amount of
// ticks of this job
List<IConfigurationElement> locatorList = new ArrayList<IConfigurationElement>();
for (IConfigurationElement element : extension.getConfigurationElements()) {
        if ("faultLocator".equals(element.getName())) {
                locatorList.add(element);
                locatorList.process(subProgress, testMethods);
        }
}
```

Listing 12 ~ Zugriff auf die ExtensionRegistry

Das returnierte *IExtension*-Objekt enthält alle unter dem gleichen Namen[16] registrierten Extensions aller Plug-Ins der Eclipse-Plattform. Konkret werden also die beiden Lokatoren aus Listing 11 aufgelistet, auf denen schließlich die in der *IFaultLocator* Schnittstelle vereinbarte Methode `process()` aufgerufen wird (siehe Kapitel 3 „Konzept des FaultLocators“).

Fraglich bleibt allerdings, wie EzUnit2 über die Beendigung eines Testlaufs informiert wird. Das JUnit-Plug-In enthält zu diesem Zweck eine Schnittstelle, an der Beobachter, die von der Klasse *TestRunListener* abgeleitet sind, registriert werden können[17]. Die Klasse *TestRunListener* bietet für alle im Lebenszyklus eines Testlaufs relevanten Phasen Callback-Methoden an, die implementiert werden können.

```
public class FaultLocationTrigger extends TestRunListener {
        @Override
        public void sessionStarted(ITestRunSession session) {
                // remove all existing markers
        }

        @Override
        public void sessionFinished(ITestRunSession session) {
                super.sessionFinished(session);
                EzUnitPlugin.trace("TestRun ended");
                if (TestMethodStore.getInstance().getAllTests().size() == 0) {
                        EzUnitPlugin.trace("no tests added to TestMethodStore -> return!");
                        return;
                }

                if (!session.getTestRunName().equals(DynamicTrace.LAUNCHCONFIG_NAME)) {
                        FaultLocationManager.getInstance().process();
                }
        }

        @Override
        public void testCaseFinished(ITestCaseElement testCaseElement) {
                super.testCaseFinished(testCaseElement);
                TestMethodStore.getInstance().addTest(testCaseElement);
        }
}
```

Listing 13 ~ Implementierung der Schnittstelle *TestRunListener*

16 in diesem Fall „faultLocator“
17 Die `start()`- und `stopp()`-Methode der Klasse *EzUnitPlugin* werden dazu genutzt, die Klasse *FaultLocationTrigger* Beobachter am *JUnitCore* zu registrieren.

Listing 13 zeigt einen Ausschnitt der Klasse *FaultLocationTrigger* und die für die *FaultLocation* relevanten Zustände. Die Zustände sind im Einzelnen der Beginn `sessionStarted()` und das Ende `sessionFinished()` eines kompletten Testlaufs, sowie die Beendigung einer speziellen TestMethode `testCaseFinished()`. Das Erreichen des Startzustands wird dazu verwendet, alle bereits existierenden Tracemarker zu löschen. Mit jedem Erreichen des Status `testCaseFinished()` wird der durchgeführte Testlauf im *TestMethodStore* zur weiteren Auswertung (beispielsweise der Testergebnisse) gespeichert. Durch das Erreichen des Endzustands wird die *FaultLocalization* gestartet.

4.1.3 Einheitliche Filter für statische und dynamische Traces

EzUnit verfolgt das Ziel, dem Benutzer möglichst genaue Auskunft über die möglichen Verursacher-Methoden eines fehlgeschlagenen Testfalles zu geben und ihn nicht mit einer unnötig großen Anzahl vorgeschlagener Methoden auf die falsche Fährte zu bringen. Erreicht wird dies zum einen durch die *MUT*-Annotationen und zum anderen durch die Reduktion der überhaupt für einen Trace in Betracht kommenden Methoden. Diese Präzisierung wird durch Filter erreicht, die auf die anzuzeigenden *MUT*'s angewendet werden.

In EzUnit1 wurden bereits Filter eingesetzt[18], die im Bereich *EzUnit Preferences* verwaltet werden konnten. Für jedes Filterkriterium wurde eine neue Zeile mit dem Wert „Filter" und dem Typ *Include* oder *Exclude* erstellt. Der Filterwert wird als regulärer Ausdruck notiert. Ein typisches Filterkriterium zeigt Abbildung 8. Bei der Speicherung der Kriterien wurde die Liste der Kriterien schließlich zeilenweise konkateniert und unter einem bestimmten Schlüssel im *PreferenceStore* der Eclipse-Plattform abgespeichert.

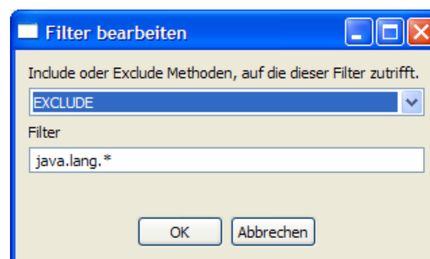


Abbildung 8 ~ Dialog „Filterkriterium bearbeiten" EzUnit

Auch das TPTP sieht eine Filterung der beim Profiling zu betrachtenden Klassen und Methoden mit Hilfe der so genannten *FilterSets* vor. *FilterSets* enthalten eine Menge von *FilterTableElement*-Objekten, die jeweils eine Zeile des *FilterSets* repräsentieren. Die Zeile enthält die Filterkriterien, die aus einem Tripel aus Klassen-, Methodennamen sowie dem Typen des Filters, *Include* oder *Exclude*, bestehen. Mit einem Filter des Types *Include* spezifiziert man genau die Elemente, die im Trace enthalten sein sollen. Das Gegenteil wird mit Filtern des Typs *Exclude* erreicht. Neben der expliziten Angabe von Klassen- und Methodennamen sind auch so genannte Wildcards „*" am Anfang oder Ende eines Wertes möglich. Im Gegensatz zu den EzUnit1-Filtern sind aber keine vollständigen regulären Ausdrücke erlaubt. Die Verarbeitung der Einträge erfolgt von oben nach unten. Sobald ein Filter „zuschlägt", wird die Auswertung der weiteren Filter abgebrochen. Würde man beispielsweise mit dem Eintrag „`*;*;EXCLUDE`"[19] starten, enthielte der resultierende Trace keine Daten. Ein Eintrag der Art „`org.projectory.*;test*;INCLUDE`" würde bedeuten, dass alle Methoden,

18 Die Filter wurden dort *MarkerFilter* genannt.
19 Dieser Filtereintrag bedeutet: Die Methoden aller Klassen, die gefunden werden, sollen ausgefiltert werden.

deren Namen mit `test` beginnen und die zu Klassen gehören, die im Package `org.projectory` liegen, in den dynamischen Trace aufgenommen werden.

Aufgrund der großen Ähnlichkeit der beiden Filterkonzepte lag es nahe, diese zu vereinheitlichen, so dass sowohl statische als auch dynamische Traces die gleichen Filterkriterien, also Klassen und Methoden enthalten. Da es sich bei den *FilterTableElement* Objekten um die (geringfügig) komplexeren Objekte handelte, wurde die bestehende EzUnit1 Lösung angepasst. Dann wurden die ohnehin schon in der Klasse *MarkerFilter* enthaltenen Konvertierungsmethoden `markerFilterToFilterTableElement()` und `filterSetToMarkerFilterVector()` so refaktorisiert, dass sie den TPTP Ansatz adaptierten (siehe Listing 14).

```
public static Vector<MarkerFilter> filterSetToMarkerFilterVector() {
        Vector<MarkerFilter> markerFilterVector = new Vector<MarkerFilter>();
        FilterSetElement filterSet = TraceManager.getInstance().getFilterSet();
        for (int i = 0; i < filterSet.getChildren().size(); i++) {
                FilterTableElement filter = (FilterTableElement)
                        filterSet.getChildren().get(i);
                        MarkerFilter markerFilter = new MarkerFilter();
                        markerFilter.setType(filter.getVisibility()
                                .equals(TraceMessages.EXCLUDE) ? EXCLUDE : INCLUDE);
                        markerFilter.setFilter(filter.getText());
                        markerFilterVector.add(markerFilter);
        }
        return markerFilterVector;
}

public static ArrayList<FilterTableElement> markerFilterToFilterTableElement(
        Vector<MarkerFilter> markerFilters) {
        ArrayList<FilterTableElement> filterTableElements =
                new ArrayList<FilterTableElement>();
        for (MarkerFilter markerFilter : markerFilters) {
                String filterType = markerFilter.getType() == INCLUDE ?
                        TraceMessages.INCLUDE : TraceMessages.EXCLUDE;
                filterTableElements.add(new FilterTableElement(
                        markerFilter.getFilter(), "*", filterType));
        }
        return filterTableElements;
}
```

Listing 14 ~ Konvertierungsmethoden in die verschiedenen FilterSets

Von Haus aus enthält das TPTP bereits eine Liste typischer FilterSets (Abbildung 9). Diese grenzen - im Falle des „Default“ Sets - alle Klassen der Pakete *java.**, *javax.** und *sun.** aus. Die beiden anderen FilterSets sind auf das Profiling von Applikation mit dem Webshere-Applikationsserver (einem Serverprodukt von IBM) spezialisiert. Diesem Konzept folgend wurde in *EzUnit2* ein FilterSet namens „EzUnitFilterSet“ implementiert, vorausgewählt und im *PreferenceStore* der Eclipse-Plattform abgespeichert.

Dieses FilterSet ist zurzeit auf die Benutzung des Testprojektes „MoneyTest“ zugeschnitten. Eine denkbare Erweiterung wäre die Berechnung eines „sinnvollen“, d.h. auf das aktive Testprojekt angepasste FilterSet automatisch, oder durch den Benutzer aktiviert, berechnen und abspeichern zu lassen. Diese Funktionalität ist zwar grundsätzlich schon vorhanden (siehe „Automatically determine filtering criteria“), müsste aber für diesen Fall so erweitert werden, dass dafür ein eigenes FilterSet angelegt und abgespeichert wird, damit auch im Kontext der statischen Traces auf diese Daten zugegriffen werden kann. Eine weitere Möglichkeit bestünde darin, die Implementierung zur Berechnung des automatischen Filters zu extrahieren und bei jeder Anforderung erneut auszuführen. Hierbei sollte allerdings besonderes Augenmerk auf die Performance gelegt werden.

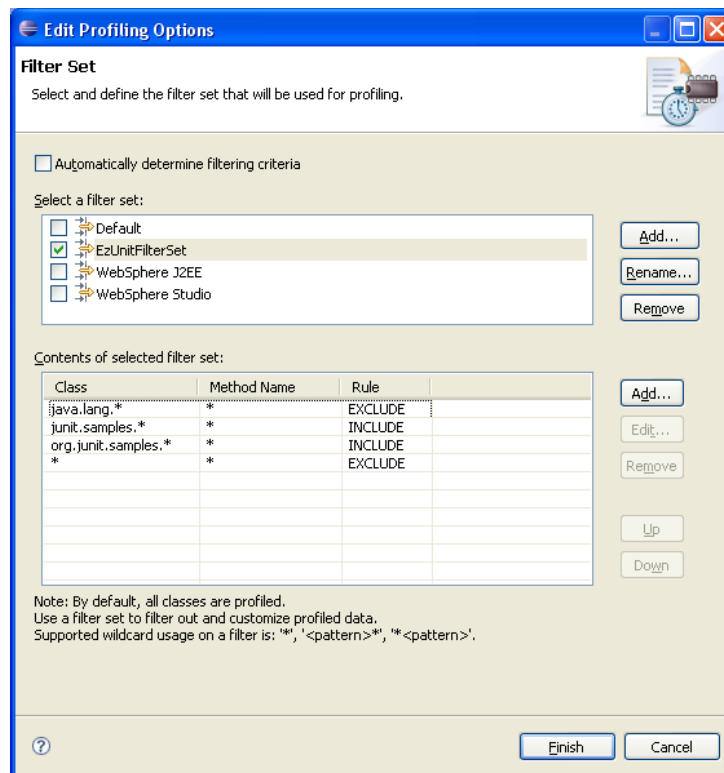


Abbildung 9 ~ Dialog zum Editieren von FilterSets

Zur Speicherung wurde auf die existierende (interne) Klasse *TraceFilterManager* zurückgegriffen, die im Umfeld des TPTP Projektes als zentrale Fassade [GoF1996] zur Verwaltung von *FilterSets* implementiert wurde. Sie stellt neben vielen Zugriffsmethoden insbesondere auch Methoden zur einheitlichen Serialisierung der *FilterSets* in Form von XML-Dokumenten zur Verfügung. Listing 15 zeigt das „EzUnitFilterSet“ in seiner XML-Repräsentation, wie es im *PreferenceStore* abgespeichert wird.

```
[...]
<filter id = 'EzUnitFilterSet' key = '' name = 'EzUnitFilterSet'>
  <contents>
    <content text = 'java.lang.*' method = '*' visibility = 'EXCLUDE'/>
    <content text = 'junit.samples.*' method = '*' visibility = 'INCLUDE'/>
    <content text = 'org.junit.samples.*' method = '*' visibility = 'INCLUDE'/>
    <content text = '*' method = '*' visibility = 'EXCLUDE'/>
  </contents>
</filter>
[...]
```

Listing 15 ~ XML Repräsentation eines *FilterSets* im *PreferenceStore*

Mit der oben beschriebenen Lösung ist es möglich, das *FilterSet* an mehreren Stellen zu pflegen, aber trotzdem auf eine gemeinsame Datenbasis zuzugreifen.

4.2 Erweiterungen durch Integration

Die folgenden Abschnitte beschreiben die Erweiterungen von *EzUnit2*, die auf existierenden Programmierschnittstellen (Abschnitt 4.2.1 und Abschnitt 4.2.2) und Klassenbibliotheken beruhen (Abschnitt 4.2.3).

4.2.1 Integration einer View

Die Eclipse-Plattform stellt verschiedene „Gestaltungselemente“ (siehe Abbildung 10) zur Verfügung, mit der ein Benutzer seine Arbeitsoberfläche individualisieren kann. Diese Gestaltungselemente sind hierarchisch gruppiert. So enthält die *Workbench* eine Menge von *Workbench Windows*. Jedes *Workbench Window* enthält seinerseits eine oder mehrere *Pages*, die wiederum eine Menge von *Editoren* und *Views* enthalten. Während *Editoren* typischerweise dazu eingesetzt werden, Dokumente oder Objekte zu ändern, dienen *Views* in der Regel dazu, bei der Navigation durch Information zu unterstützen, Editoren zu öffnen oder Informationen darzustellen.

```
<extension point="org.eclipse.ui.views">
  <category id="org.projectory.ezunit" name="%categoryEzUnit" />
  <view allowMultiple="false"
     category="org.projectory.ezunit"
     class="org.projectory.ezunit.internal.views.MUTSelectionView"
     icon="icons/ezUnit_warn.gif"
     id="org.projectory.ezunit.internal.views.MUTSelectionView"
     name="%mutSelectionView.name">
  </view>
</extension>
```

Listing 16 ~ ExtensionPoint ‚views’ aus plugin.xml

Zur Erstellung einer neuen *View* sind zwei Schritte durchzuführen. Zunächst ist eine Viewklasse zu erstellen, die von der Klasse *ViewPart* abgeleitet sein sollte, damit auf das dort bereits implementierte Basisverhalten von Views zurückgegriffen werden kann. Danach ist für die erstellte Viewklasse eine Extension in der Datei *plugin.xml* für den ExtensionPoint `org.eclipse.ui.views` zu registrieren (siehe Listing 16).

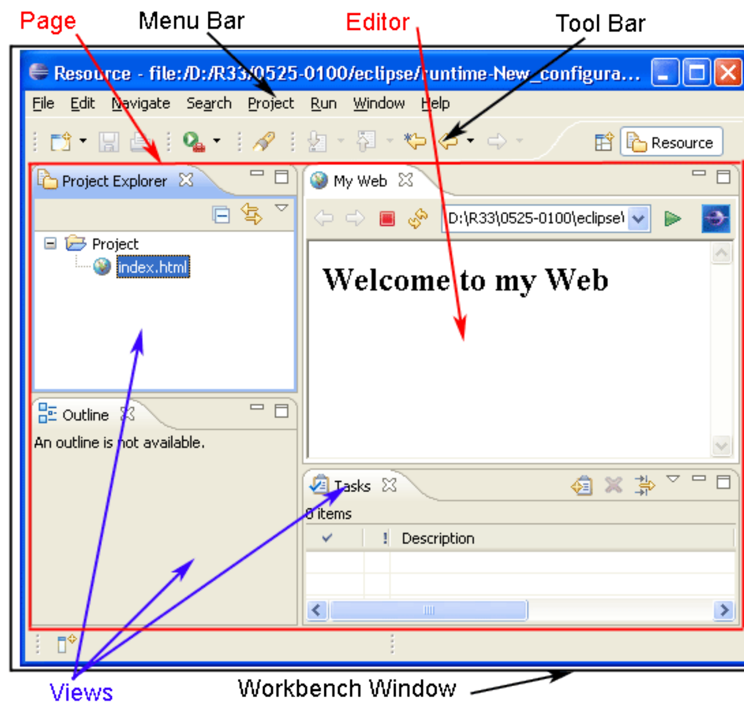


Abbildung 10 ~ Übersicht Gestaltungselemente aus [help-workbench]

Jede *View* durchläuft einen definierten Lebenszyklus, in dessen Verlauf eine Reihe von Methoden in definierter Reihenfolge aufgerufen werden. Zuerst wird eine Instanz der konfigurierten Viewklasse erzeugt. Dabei überprüft die Eclipse-Plattform, ob sie das Interface *IViewPart* implementiert[20]. Danach werden die Methoden `init()`, `createPartControl()` und nach dem Schließen der View die Methode `dispose()` aufgerufen.

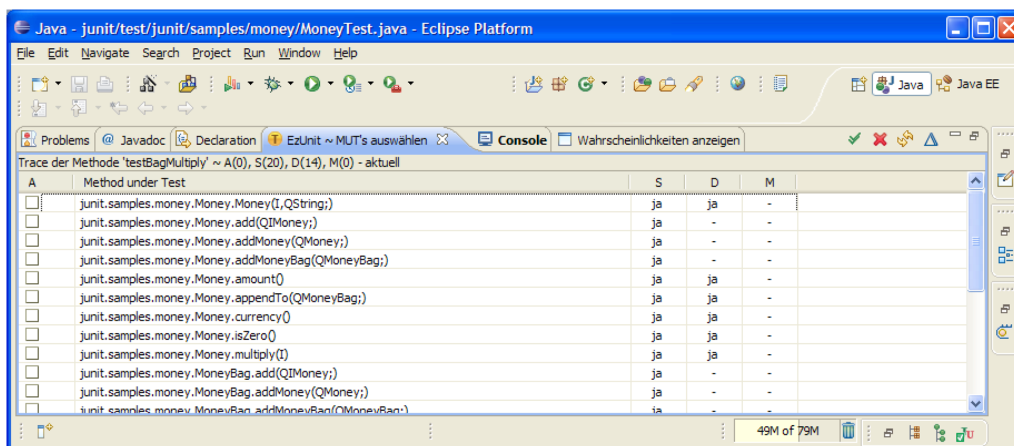


Abbildung 11 ~ *MUTSelectionView*

EzUnit2 ersetzt mit der *MUTSelectionView*, den in EzUnit1 verwendet eigenständigen Dialog[21] (siehe Abbildung 11). Die beschriebenen Lifecycle-Methoden implementiert *EzUnit2* ebenfalls. Die Methode `createPartControl()` erzeugt - wie empfohlen - die anzuzeigenden Interaktions-Elemente (Schaltflächen) und registriert die angeboten *Actions*.

20 Sollte dies nicht der Fall sein, wird eine *PartInitException* erzeugt.
21 Der Dialog war erreichbar über einen Rechtsklick auf die Testmethode, dann MUT's auswählen

Die Implementierung der beiden Methoden `init()` und `dispose()` nimmt die An- und Abmeldung der View am *SelectionService* vor.

4.2.2 Selektionen des Benutzers

Zur Synchronisierung der Daten in den vielen verschiedenen, voneinander völlig entkoppelten Views bietet die Eclipse-Plattform das *SelectionFramework* an. Bei diesem Framework handelt es sich um eine erweiterte Implementierung des Beobachter-Entwurfsmusters, welches in [GoF1996] vorgestellt wird. Abbildung 12 vermittelt einen schematischen Überblick und zeigt die beteiligten Komponenten auf.

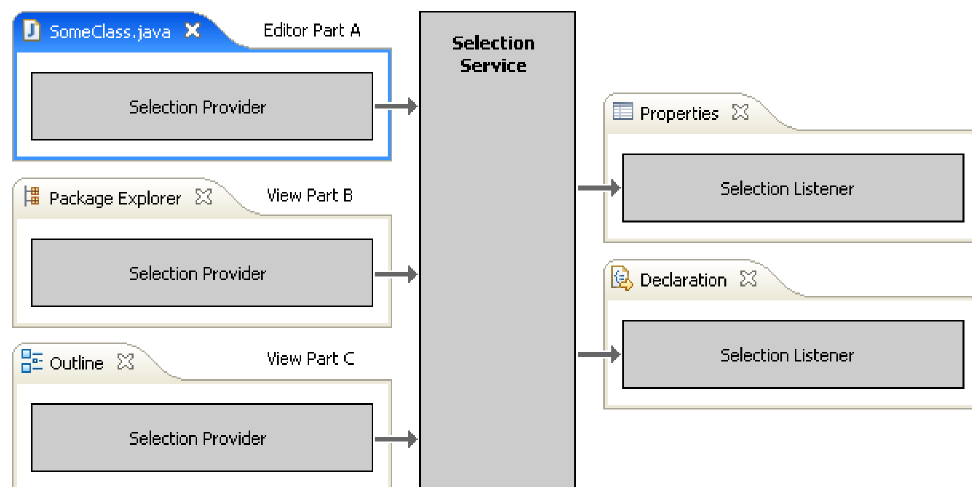


Abbildung 12 ~ SelectionService der Eclipse-Plattform

Die beteiligten Subjekte heißen *SelectionProvider*. Sie lösen immer dann neue Ereignisse aus, wenn sich die Selektion im aktuellen Bereich der Benutzerschnittstelle ändert oder ein anderer Bereich der Workbench aktiviert wird. Jedes Ereignis enthält neben dem auslösenden Quellobjekt ein *ISelection* Objekt, welches eine Datenstruktur mit den selektierten Objekten enthält. Alle Beobachter des *SelectionFramework* implementieren die *ISelectionListener*-Schnittstelle.

In Erweiterung zu dem ursprünglich formulierten Entwurfsmuster besteht aber keine direkte Verbindung der Beobachter- zu den Subjektelementen. Vielmehr wurde ein Vermittler, der *SelectionService*, zwischengeschaltet, der die eingehenden Ereignisse aller *SelectionProvider* an alle registrierten *ISelectionListener* weiterleitet. Auf diese Weise wurde die Kardinalität der Beziehung Subjekt/Beobachter von 1:n auf m:n erweitert und somit eine hohe Entkopplung der beteiligten Komponenten erreicht [selectionFramework].

Um die *MUTSelectionView* an den Ereignissen der *SelectionProvider* partizipieren zu lassen, muss sie also zunächst das Interface *ISelectionListener*, welches die Methode `selectionChanged()` anbietet (siehe Listing 17), implementieren.

```
1: public void selectionChanged(IWorkbenchPart sourcepart, ISelection selection) {
2:     // we ignore our own selections
3:     if (sourcepart != MUTSelectionView.this) {
4:             IMethod selectedMethod =
5:                     extractIMethodFromSelection(sourcepart, selection);
6:
7:             if (selectedMethod != null) {
8:                     // do only show MUT's if this iMethod is annotated as a testMethod
9:                     if (HelperMethods.isTestMethod(selectedMethod)) {
10:                            setSelectedMethod(selectedMethod);
11:                            showMUTTable(selectedMethod);
12:                    } else {
13:                            // in all other cases we do not show anything
14:                            // (beside this little message)
15:                            showAlternativeText(Messages
16:                                    .MUTSelectionView_nothingToDisplay);
17:                    }
18:            }
19:    }
```

Listing 17 ~ Implementierung *ISelectionListener* in *MUTSelectionView*

Interessant ist in diesem Zusammenhang die Methode `extractIMethodFromSelection()` (Zeile 5, Listing 17). Sie kapselt insbesondere das Wissen über die verschiedenen Objekte der *ISelection* Schnittstelle und wie im jeweiligen Fall auf die selektierte Methode geschlossen werden kann. Schließlich gibt sie ein *IMethod* Objekt zurück.

```
1:  if (selection instanceof ITextSelection) {
2:      structuredSelection = SelectionConverter.getStructuredSelection(sourcepart);
3:  } else if (selection instanceof IStructuredSelection
4:      && ((IStructuredSelection) selection).size() == 1) {
5:      structuredSelection = (IStructuredSelection) selection;
6:  }

[...]

7:  if (structuredSelection != null
8:      && structuredSelection.getFirstElement() instanceof IMethod) {
9:      return (IMethod) structuredSelection.getFirstElement();
10: }
```

Listing 18 ~ Extraktion eines *IMethod* Objektes aus *ISelection*

Darüber hinaus ist die *MUTSelectionView* am *SelectionService* über die Methode `getSite().getWorkbenchWindow().getSelectionService().addPostSelectionListener(listener)` zu registrieren[22].

Für die *MUTSelectionView* sind *SelectionChangedEvents* erst dann interessant, wenn die View auch tatsächlich geöffnet ist. Aus diesem Grund wurden die beiden Methoden `init()` und `dispose()` aus dem Lebenszyklus der View genutzt, um sie als Beobachter am *SelectionFramework* zu registrieren (sieh auch Abschnitt 4.2.1 „Integration einer View“).

22 Aus Kompatibilitätsgründen wird auch noch die Methode `getSite().getPage().addPostSelectionListener()` angeboten. Die Verwendung dieser Methode ist jedoch nicht zu empfehlen, da bei einem Doppelklick kein `selectionChanged()`-Event ausgelöst wird.

4.2.3 Reaktion auf Änderungen des Quelltextes

Gemäß den Anforderungen sollen strukturelle Änderungen an Testmethoden oder Methoden, die Teil eines Traces (*MUT*) sind, zur Markierung und Invalidierung der jeweiligen Traces führen (siehe Abschnitt 1.1 „Aufgabenstellung“).

Die Eclipse-Plattform bietet zur Verfolgung struktureller Quelltextänderungen den so genannten *Delta Mechanismus* an. Der *Delta Mechanismus* implementiert das Beobachter-Entwurfsmuster. Die registrierten Beobachter werden durch den Versand von *ElementChangedEvent*s über Änderungen informiert.

Jedes *ElementChangedEvent* enthält ein *IJavaElementDelta* Objekt, welches das geänderte Objekt selbst[23] und einen Identifikator über die Art der Änderung (ADD, CHANGE oder REMOVE) enthält. Der Detailierungsgrad der erkannten Deltas reicht in der Eclipse Standardimplementierung bis zur Ebene der *Member* hinab. Als *Member* werden alle Elemente bezeichnet, die die Struktur einer Übersetzungseinheit (*ICompilationUnit*) bestimmen. Konkret sind dies die Klassen Type (*IType*), Methoden (*IMethode*), Attribute (*IField*) und Initialisierungen (*Initializer*).

Deltas, die beispielsweise durch Hinzufügen einer neuen Methode entstehen, werden in Form eines Baumes repräsentiert. Das hinzugefügte Element ist dabei ein Blatt des Delta-Baumes. Durch Auslesen der Methode `getKind()` erlangt der Entwickler Einblick darüber, welcher Art die Änderung ist (add, change oder remove).

Deltas die „unterhalb“ der *Member*-Ebene liegen, werden zwar ebenfalls vom Delta Mechanismus erkannt und gemeldet, es fehlt jedoch jede detaillierte Information darüber, was der Inhalt der Änderung ist. Das liegt daran, dass solche Änderungen nicht die Struktur der *CompilationUnit* verändern und somit nicht als Delta im Sinne des ursprünglichen Eclipse Delta Mechanismus erkannt werden.

Diese Unzulänglichkeit wurde durch die Einbindung des stark erweiterten Delta Mechanismus von [sschmidt2007] behoben. Schmidt hat gezeigt, dass zur Berechnung eines Deltas, welches hinreichend detaillierte Informationen über die Veränderungen von Methoden oder lokalen Variablen enthält, neben der aktuellen *CompilationUnit* auch ihre jeweilige Vorgängerversion mit einzubeziehen ist. Die Klasse *DeltaProcessor* hält diesen Cache und informiert alle registrierten Beobachter über strukturelle Änderungen von Methoden.

23 Das geänderte Objekt ist vom Typ *IJavaElement*.

```
public void methodEvent(ElementEvent<IMethod> event) {
        // if the changing method is a testMethod return!
        if (HelperMethods.isTestMethod(event.getElement()))
                return;

        IMethodUnderTest mut = this.findMut(event.getElement());
        if (mut != null) {
                mut.setModified(true);
                EzUnitPlugin.trace(this, "method '" + mut.getDisplayName()
                                + "' which is contained in mutMethodStore changed");

                // invalidate each traceManagerElement this method is contained in
                TraceManager.getInstance().invalidate(mut.getCallingTestMethods());

                // hence this method changed we cannot assure that the old list
                // of callingTestMedthods is still correct! Since we invalidated
                // all corresponding traces they will be rebuilded if requested
                // -> so remove the existing list of callingTestMethods!
                mut.resetCallingTestMethods();
        }
}
```

Listing 19 ~ Implementierung von *IMethodEventListener* in *MethodUnderTestStore*

Jeder an Änderungen interessierte Beobachter muss die Schnittstelle *IMethodEventListener* implementieren[24]. Zur Zeit implementieren die Klassen *MethodUnderTestStore* und *TestMethodStore* diese Schnittstelle. Sie verwenden die Änderungsinformation, um ihren Methodencache aktuell zu halten. Diese Caches werden invalidiert, sobald eines der folgenden Ereignisse eintritt:

1. Eine Testmethode hat sich geändert.
2. Der Methodenrumpf einer der *MUTs* einer Testmethode hat sich geändert
3. Eine *MUT* dieser Testmethode wurde gelöscht.
4. Die Signatur einer *MUT* hat sich geändert[25]

24 Listing 19 zeigt die Implementierung dieser Schnittstelle der Klasse *MethodUnderTestStore*.

25 Der neue Delta Mechanismus gibt bisher keinen Aufschluss darüber, ob eine *MUT* von einer neuen Methode überschrieben wurde oder eine Überladung dieser Methode hinzugefügt wurde. Das kann zur Folge haben, dass die Liste der potentiellen Verursacher in diesen Fällen unvollständig ist.

5 Installation und Bedienung

Die Benutzung von EzUnit2 Funktionalität setzt eine Eclipse-Umgebung nebst vollständig installierten TPTP-Plug-Ins voraus. Zur Vereinfachung der Installation bietet das TPTP–Projekt eine fertige Distribution, dass „*TPTP all-in-one package*" an, welche alle Abhängigkeiten erfüllt. Voraussetzung zur Installation ist ein installiertes JDK 1.5 [reqTptp].

Nachdem diese Distribution in einem beliebigen Verzeichnis extrahiert wurde, ist im zweiten Schritt das EzUnit2 Plug-In, welches als jar-Archiv ausgeliefert wird, im vorhandenen Unterverzeichnis „plugins" abzulegen.

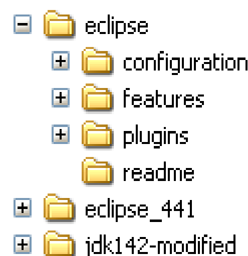


Abbildung 13 ~ Verzeichnisstruktur einer Eclipse Installation

Nach der Installation des Packages ist eine zusätzliche *LaunchConfiguration* (neuer Button neben „Run") verfügbar. Hierüber ist das Starten einer Applikation im „Profile-Mode" möglich. Zur „manuellen" Erstellung eines Traces sind nun folgende Konfigurationen durchzuführen. Zunächst ist eine *LaunchConfiguration* zu erzeugen (der entsprechende Dialog öffnet sich durch Betätigen der rechten Maustaste auf den Ordner und auswählen von JUnit Tests -> Profile As -> JUnit Test). Im Dialog ist auf dem Reiter „Monitor" der Kollektor „Execution Time Analysis" auszuwählen.

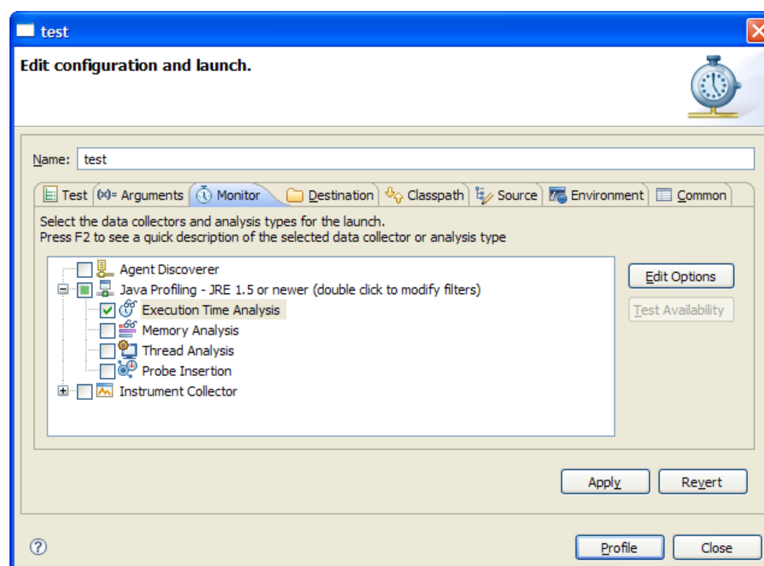


Abbildung 14 ~ Konfiguration „Monitor" - Launchconfiguration "Profile Mode"

Zur Erzeugung des dynamischen Traces werden die Daten dieses Kollektors mit der Option „Show execution flow graphical details“ benötigt. Die Option „Show execution statistics (compressed)“ reduziert den Datenverkehr vom DataCollectionFramework in das *TraceModel* zwar erheblich, stellt allerdings keine *TRCFullMethodeInvocation* Objekte zur Verfügung, die für die Erzeugung des dynamischen Traces benötigt werden.

Ferner öffnet sich durch Doppelklick auf die Gruppe „Java Profiling“ der *FilterSet* Konfigurationsdialog. Auch die *FilterSets* haben die Aufgabe, die Menge der in das *TraceModel* zu übertragenen Ereignisse zu reduzieren[26].

Mit dem Button „Profile“ kann der *JUnitTestRunner* gestartet werden und arbeitet alle ausgewählten JUnit Tests ab. Währenddessen übertragen die Datenkollektoren die *ProfileEvents* ins *TraceModel*.

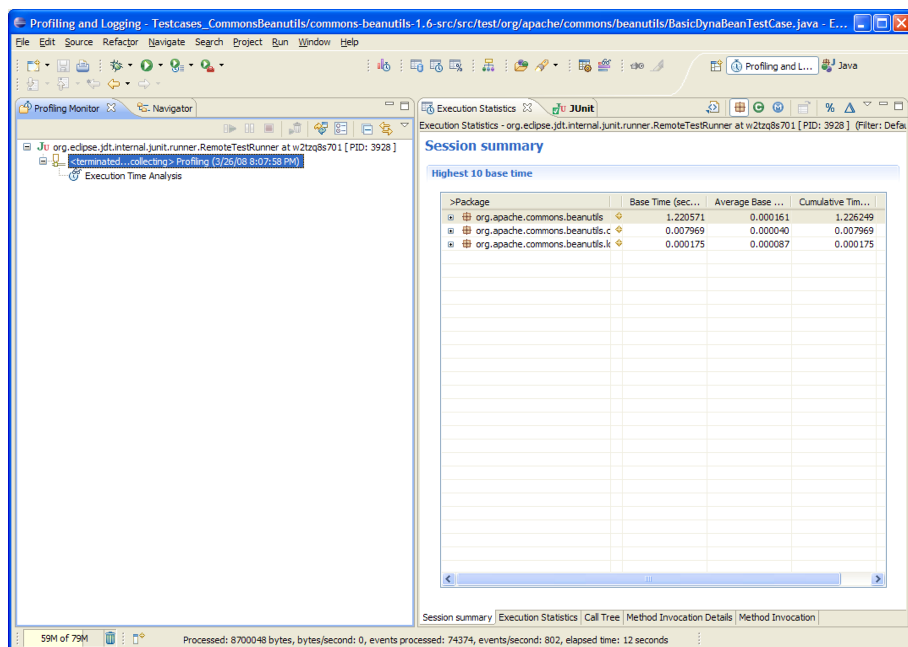


Abbildung 15 ~ Perspektive *Profiling* mit dem Ergebnis eines Traces

Die ebenfalls mit dem TPTP installierte „Profiling and Logging“ Perspektive gibt Auskunft über den aktuellen Status der *ProfilingSession* (genauer des Daten Kollektors) und ermöglicht erste Einblicke in die gesammelten Daten.

26 Eine detaillierte Beschreibung der FilterSets ist in Abschnitt 4.1.3 „Einheitliche Filter für statische und dynamische Traces“ zu finden.

6 Ergebnisse

Alle im Abschnitt „1.1 Aufgabenstellung“ formulierten funktionalen Anforderungen an *EzUnit2* wurden erfüllt. Einzig die Invalidierung der Trace-Caches wird derzeit nicht vollständig adressiert. Eine detaillierte Diskussion hinsichtlich der fehlenden Invalidierungen erfolgt in Abschnitt 6.4.

Die Integration als View (Abschnitt „Integration einer View“), der Anschluss an das *SelectionFramework* (Abschnitt „Selektionen des Benutzers“), sowie die Einbindung eines erweiterten Delta-Mechanismus (Abschnitt „Reaktion auf Änderungen des Quelltextes“), konnten erfolgreich umgesetzt werden. Auch die Implementierung des Tracers auf TPTP-Basis (Abschnitt „Zeitverhalten bei der Trace Erstellung“) und die Bereitstellung des Extension-Point (Abschnitt „“) sind erfolgt.

Die folgenden Abschnitte zeigen die Ergebnisse aus nicht-funktionaler Sicht auf. Abschnitt 6.1 betrachtet das Zeitverhalten bei der Erzeugung dynamischer Traces, während Abschnitt 6.2 die Größe dynamischer Traces ins Verhältnis zu statisch erzeugten Traces setzt. Abschnitt 6.3 geht schließlich auf die Speicherauslastung ein.

6.1 Zeitverhalten bei der Trace Erstellung

Ein wichtiges, nicht-funktionales Kriterium zur Beurteilung der Qualität einer Software ist ihr Zeitverhalten, welches im folgenden Abschnitt näher erläutert wird.

EzUnit2 bietet zwei Trace-Erstellungs-Modi an, die sich im Umfang der zu erstellenden Traces, also in der Anzahl der zu tracenden Testmethoden unterscheiden. Der Umfang wird dadurch reguliert, dass entweder die komplette Testsuite getracet wird, oder nur Teile daraus. Ein vollständiger Trace wird zu Beginn einer Arbeitssitzung erzeugt. Alle folgenden Traces umfassen immer nur die Testmethoden, deren Traces aufgrund bestimmter Kriterien ungültig geworden sind (Kriterien, siehe Abschnitt 4.2.3). Diese Teiltraces werden *inkrementelle Traces* genannt.

	1	2	3	4	5	6	7	8
Projekt	*Anzahl Tests*	*Kein Trace [sec]*	*Dauer Dyna–mischer Trace [sec]*	*Dauer Stati–scher Trace [sec]*	*Mittlere Anzahl Tests / Mut*	*Mittlere Anzahl betrof-fener Klassen*	*Dauer inkrementeller dynamischer Trace [sec] - Mittelwert*	*Dauer inkremen-teller statischer Trace [sec]*
junit.samples.money	22	0,0940	4	13	12	1	4	7
junit.tests	308	1,0160	35	541	21	7	26	37
org.apache.commons.beantutils	336	1,4780	11	1363	52	3	5	211
org.apache.commons.codec	191	0,7180	107	127	10	2	34	7
org.apache.commons.mail	75	0,0120	11	58	6	2	14	5
org.apache.commons.math	1022	35,3230	5604	12018	24	7	485	282

Tabelle 1 ~ Performance der Erstellung statischer, sowie dynamischer Traces[27]

27 Dauer in Sekunden – gemessen auf einem Notebook PC mit Intel Centrino Duo T5600 Prozessor 1,83 GHz mit 2GB Arbeitsspeicher

Maßgeblich für die Dauer der Erstellung inkrementeller Traces ist die Anzahl der Testmethoden, die neu getracet werden müssen. Ein weiteres Kriterium ist, in welcher Granularität neue Traces technisch überhaupt erzeugt werden können. Statische Traces können Testmethodengenau erzeugt werden. Die dynamischen Traces können jedoch aufgrund einer Restriktion im *JUnitTestRunner*[28] nur Testklassengenau erzeugt werden. Abbildung 16 zeigt, dass die Startkonfiguration des *JUnitTestRunners* keine Auswahl einer konkreten Testmethode der Testklasse zulässt.[29] Das kann bedeuten, dass eine komplette Testklasse neu getracet wird, obwohl dies aufgrund der Gültigkeit der Traces gar nicht notwendig wäre.[30]

Zur Berechnung der Dauer der Traces sind aus diesem Grund die beiden Tracetypen (dynamisch und statisch) zu unterscheiden.

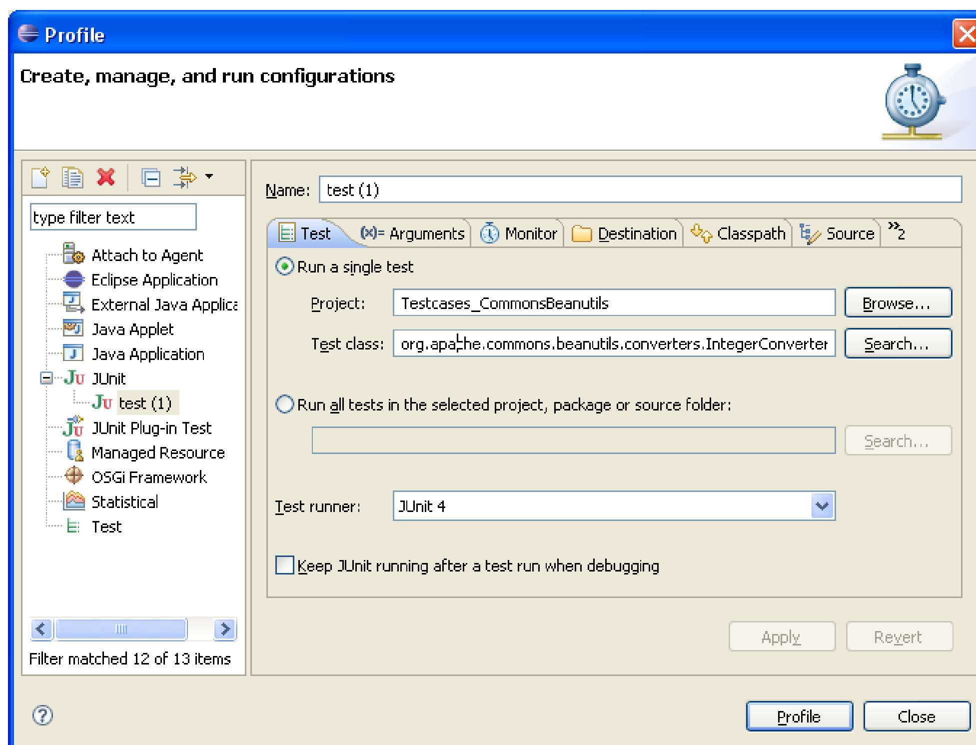


Abbildung 16 ~ Konfigurationsmöglichkeiten des JUnit TestRunners

Bei der Betrachtung inkrementeller statischer Traces ist festzuhalten: Ändert sich die Testmethode selbst, ist nur ihr eigener Trace zu erneuern. Sollte der Trace aufgrund der Änderung einer *MUT* ungültig geworden sein, können weitere Testmethoden betroffen sein, in deren Trace die *MUT* ebenfalls enthalten ist. Spalte fünf (aus Tabelle 1) zeigt, in wie vielen Tests eine *MUT* im Mittel enthalten ist. Am Beispiel des Projektes `junit.samples.Money` sind im Mittel zwölf Traces zu erneuern, wenn sich eine *MUT* geändert hat.

28 Die Klasse *JUnitTestRunner* ist Teil des JUnit-Plug-Ins und ist für die Ausführung der Unit-Tests verantwortlich.

29 Abhilfe könnte eine Erweiterung des *JUnitTestRunners* schaffen, so dass auch das Tracen einer einzelnen Testmethode möglich wird. Hierzu wäre ein neuer *TestRunner* zu implementieren, der in Erweiterung zum *JUnitTestRunner* die Auswahl einer Testmethode zulässt.

30 Bei den verwendeten Testprojekten viel allerdings auf, dass die Änderung einer MUT häufig zur Invalidierung mehrerer Traces einer Testklasse führte. Je nach Verhältnis der gültigen zu ungültigen Traces einer Testklasse, reduzierte das den zeitlichen Nachteil, der durch die Restriktion existiert.

Im Falle der inkrementellen dynamischen Traces wurde aufgrund der oben genannten Restriktion ermittelt, in wie vielen Testklassen eine *MUT* im Mittel enthalten ist (Tabelle 1, Spalte 6). Da das Projekt `junit.samples.Money` nur aus einer Testklasse besteht, sind die Zeiten für den vollständigen Trace (Spalte 3) und für den inkrementellen Trace (Spalte 7) identisch.

Tabelle 1 zeigt, dass die Erstellung vollständiger dynamischer Traces im Mittel 218-mal (Spalte 3) und die Erstellung vollständiger statischer Traces im Mittel 1163-Mal (Spalte 4) länger dauern als die Abarbeitung der gleichen Testsuite ohne aktiviertes Tracing (Spalte 2). Dieser Faktor ergibt sich aus dem zusätzlichen Rechenaufwand, der für die Traversion der *AbstractSyntaxTrees* (statisch), bzw. für die Erzeugung und Verarbeitung der *ProfilingEvents*.

Über alle gemessenen Projekte ist die Erstellung des vollständigen dynamischen Traces schneller als die Erstellung des vollständigen statischen Traces. Abweichend davon, dauerte die Erstellung inkrementeller dynamischer Traces in den Projekten `commons.codec`, `commons.mail` und `commons.math` länger als die statischen, da diese Projekte viele Testklassen beinhalten, die rechenintensive Testmethoden mit Schleifen und Kontrollstrukturen enthalten. Diese Konstrukte werden von den statischen Traces aber gar nicht ausgewertet, wodurch sich der Geschwindigkeitsvorteil ergibt.

Die Messpunkte sind in der Klasse `AbstractTrace` in den Methoden `beforeCreation()` und `afterCreation()` zu finden. Im Fall des dynamischen Traces wird die Dauer der *ProfilingSession* hinzu addiert.[31]

6.2 Tracegröße

Der Vergleich der Größen statischer und dynamischer Traces bestätigt die Vermutung, dass die dynamischen Traces in der Regel kleiner sind als ihr statisches Pendant. Im Mittel ergibt sich bei den getesteten Projekten der Faktor drei.

Projekt	*Mittlere Tracegröße (statisch)*	*Mittlere Tracegröße (dynamisch)*	*Verhältnis Statisch zu Dynamisch*
junit.samples.money	16	13	1
junit.tests	38	9	4
org.apache.commons.beantutils	62	9	7
org.apache.commons.codec	11	6	2
org.apache.commons.mail	12	10	1
org.apache.commons.math	39	10	4
Mittelwert	*30*	*10*	*3*

Tabelle 2 ~ Vergleich Anzahl der Methoden statischer vs. dynamischer Traces

Der Unterschied wird deutlicher, je höher die Vererbungshierarchie der betrachteten Klassen ist. Das liegt daran, dass gleichzeitig mit der Höhe der Hierarchie auch die Anzahl der Überschreibungen steigt, die vom statischen Trace als Methode im Aufrufgraph erkannt wird.

31 Die Dauer der ProfilingSession kann einem Logeintrag des TPTP-Frameworks entnommen werden. Der Logeintrag kann über die View „ErrorLog“ eingesehen werden.

6.3 Speicherauslastung

Ein weiteres nicht funktionales Kriterium ist die Speichereffizienz. Sie beschreibt wie „sorgsam“ eine (Software)Komponente mit der Ressource „Arbeitsspeicher“ umgeht. *EzUnit2* legt die gewonnenen Tracedaten, zur weiteren Verarbeitung im Arbeitsspeicher ab (in-memory).[32] Tabelle 3 zeigt den durchschnittlichen Speicherverbrauch der Eclipse-Plattform nachdem die vollständige Testsuite des jeweiligen Projektes getracet wurde.

Projekt	***Anzahl Tests***	***Speicherauslastung***
junit.samples.money	22	34MB
junit.tests	308	51MB
org.apache.commons.beantutils	336	39MB
org.apache.commons.codec	191	168MB
org.apache.commons.mail	75	57MB
org.apache.commons.math	1022	1398MB

Tabelle 3 ~ Speicherauslastung nach Trace

Es hat sich gezeigt, dass der zur Speicherung des Trace-Modells benötigte Arbeitsspeicher während des Tracings trotz sorgsam gesetzter Filter stetig ansteigt. Insbesondere im Falle des Projektes `org.apache.commons.math` wird der vollständige unter Windows zuweisbare Speicher von 1,4GB verbraucht. In einigen Fällen brach die Verarbeitung der Testsuite mit einer `OutOfMemoryException` ab.

Der extrem hohe Speicherverbrauch für das Math-Projekt liegt zum Einen darin begründet, dass es sich um sehr großes Projekt mit vielen Tests handelt; Zum anderen sind viele Tests mit vielen Schleifendurchläufen enthalten, die wiederum zu vielen einzelnen *ProfilingEvents* führen.

Generell lässt sich der hohe Speicherverbrauch mit der für die Speicherung gewählten Datenstruktur, dem *TraceModel*, erklären. Das *TraceModel* ist generisch und soll möglichst viele unterschiedliche Anforderungen erfüllen. Es soll alle gesammelten Daten speichereffizient (normalisiert) ablegen und gleichzeitig schnellen Suchzugriff auf die Daten ermöglichen (denormalisiert). Das *Tracemodel* wurde für den schnellen Suchzugriff optimiert und ist daher relativ denormalisiert modelliert. Um das Ziel der Speichereffizienz nicht aufzugeben, wurden zwei Datenstrukturen modelliert, die sich in ihrem Datenumfang unterscheiden. Der Benutzer kann zu Beginn einer ProfilingSitzung entscheiden, welchen Detaillevel diese haben soll. Darauf aufbauend wird entweder die schmalere Klasse *TRCMethodInvocation* oder die Klasse *TRCFullMethodInvovation* zur Datenspeicherung verwendet.

Leider wird die für die Trace Erzeugung nötigen Aufrufer-Aufrufende-Beziehung (*Caller - Callee*) in der umfangreichen Klasse *TRCFullMethodInvocation* modelliert. Damit werden neben diesen wichtigen Informationen auch viele für unseren Zweck irrelevanten Daten abgelegt. Während einer Profiling Sitzung werden leicht einige hunderttausend Objekte dieser Klasse erzeugt, wobei 80.000 Instanzen etwa 80MB Arbeitsspeicher verbrauchen. Dieses Problem verschärft sich dadurch, dass die derzeitige Implementierung erst nach Beendigung der Profiling Sitzung gestartet werden kann[33].

32 Eine alternative Speicherung der Trace-Modelle in einer Datei (on-disc) wird in Abschnitt 8.1.1 „Integration des TPTP-Profilers“ beschrieben.

33 Grund ist der große Anteil asynchroner Verarbeitungsschritte und die fehlende Rückmeldung des TPTP Frameworks über den Status dieser Schritte.

Des weiteren existieren auch nach dem Löschen der Datencontainer des *TraceModels* noch Referenzen auf *TRCFullMethodInvocation* Objekte. Damit kann der Garbage Collector diese nicht mehr benötigten Objekt nicht entfernen. Der verbrauchte Arbeitsspeicher wird somit nicht wieder freigegeben.

Ein möglicher Ansatz zur Lösung des Speicherproblems wäre die Umstellung der bisher sequentiellen Verarbeitung (erst vollständiges Profiling, dann Erzeugung der Traces) hin zu einer parallelen Verarbeitung. So sollte direkt mit der Erzeugung eines Traces begonnen werden, wenn alle nötigen Modelldaten vorhanden sind. Nachdem ein Trace erzeugt ist, sollten die korrespondierenden Modellelemente gelöscht werden. Hierzu sind allerdings weitreichende Eingriffe in das TPTP Framework erforderlich, um die nötigen Rückmeldungen über den Status einzelner Verarbeitungsschritte des Frameworks zu erhalten.

6.4 Unvollständige Invalidierung der Trace-Caches

Wie in Abschnitt 4.2.3 „Reaktion auf Änderungen des Quelltextes“ beschrieben, implementiert *EzUnit2* einen erweiterten Delta-Mechanismus, der in erster Linie für die Invalidierung der Trace-Caches zuständig ist. Ziel ist es, aufgrund der Laufzeit der Trace-Erstellung (siehe oben), jeden Trace solange weiter zu verwenden, bis sein Inhalt ungültig geworden ist.

Die Ungültigkeit eines Traces wird durch eine Reihe von Kriterien bestimmt (siehe Abschnitt 4.2.3). Über diese vier Kriterien hinaus, können sich Traces aber auch dann verändern, wenn eine *MUT* von einer neuen Methode überschrieben wurde oder eine Überladung dieser Methode hinzugefügt wurde. In beiden Fällen können sich die Traces verlängern, was im Umkehrschluss bedeutet, das die Liste der potentiellen Verursacher kürzer ausfällt.

Dies liegt darin begründet, dass die derzeitige Implementierung des *DeltaProcessors* nur die Änderungen auf Datei-Ebene[34] zur Berechnung des Deltas heranzieht. Das Überschreiben einer Methode (*MUT*) in einer abgeleiteten Klasse führt aber nicht zur strukturellen Änderung seiner Superklasse. Insofern bleibt diese Änderung vom *DeltaProcessor* unerkannt.

Eine mögliche Lösung dieses Problems wäre, dass zusätzlich zum schon berechneten Delta alle *CompilationUnits* der Superklassen vom *DeltaProcessor* unter Beobachtung genommen werden. Dabei müsste die Superklassen-Hierarchie bei jeder Erkennung eines Deltas neu aufgebaut werden. Fraglich ist, ob die Erstellung der Typ-Hierarchie ausreichend schnell durchgeführt werden kann, um den Arbeitsablauf nicht zu stören.

34 Genau genommen werden Deltas auf der Ebene der *CompilationUnit* (siehe Abschnitt 4.2.3) berechnet.

7 Diskussion und verwandte Arbeiten

In diesem Kapitel werden weitere Tracingansätze beleuchtet ein Überblick über die in diesem Kontext aktuellen und relevanten Arbeiten gegeben.

7.1 Alternative Tracingansätze

Im ersten Versuch wurde der Ansatz verfolgt, Daten des Debuggers als Informationsquelle zu nutzen. Die JavaVirtualMachine (JVM) der Firma Sun bietet zu Zwecken der Fehlersuche eine Reihe von Schnittstellen an, die unter der so genannten *Java Platform Debugger Architecture (JPDA)* zusammengefasst sind [jpda]. Es wurden Haltepunkte (*engl.: Breakpoints*) der Typen *MethodEntry* und *MethodExit* gesetzt. Sobald ein Haltepunkt erreicht wurde, wurden die gewonnen Daten in einer speziellen Datenstruktur protokolliert. Die Menge aller *MethodEntry*-Events enthielt die nötigen Daten für den dynamischen Trace. Durch Auswerten beider Eventtypen konnten auch Verschachtelungen (Baumstruktur) erkannt werden. Der dynamische Aufrufgraph konnte damit erzeugt werden.

Der Prototyp zeigt allerdings, dass die Aufrufe der Programmierschnittstellen der JPDA (genauer des *JavaDebuggingInterface JDI*) sehr tief in die interne Implementierung der Eclipse-Plattform eingewoben sind und nicht zur Verwendung Dritter gedacht sind.[35] Insbesondere war kein wahlfreier Zugriff auf die zu tracenden Methoden möglich, um gezielt Haltepunkte zu setzen. Um das Problem zu umgehen, erhielten alle Methoden, die aufgerufen wurden (auch die der JRE), Haltepunkte. Das führte dazu, dass die Menge der zu verarbeitenden Events sehr hoch war. Die JPDA sieht zur Zeit aber keine Filterung der Events auf Client-Seite, also in der das JDI implementierenden Software, vor. Auch auf Serverseite war keine Einschränkung möglich. Somit mussten alle Events zunächst über das JDWP zum Client transportiert werden, um dann die clientseitige Filterung vorzunehmen.[36]

Beim zweiten Ansatz wurden so genannte *Probes* als Datenlieferanten verwendet. Probes sind Java Quelltext Fragmente, die mit Hilfe des TPTP Probekit Editors erstellt und kompiliert werden [help-probekit]. Probes können aus einer Reihe *FragmentTypes* bestehen. Diese definieren, wann das jeweilige Codefragment ausgeführt werden soll. Damit Probes nicht auf alle Methoden angewendet werden, können *Targets* definiert werden. Die Target-Definitionen sind mit den Filtern des Profiling Frameworks zu vergleichen (siehe auch 4.1.3 „Einheitliche Filter für statische und dynamische Traces“).

Dank guter Dokumenation waren die Probes schnell erstellt. Sie ließen sich programmatisch über die *ProbeRegistry* zu Beginn einer Profiling Sitzung deployen. Auch das Sammeln der Tracedaten und das Erstellen des dynamischen Traces verlief problemlos. Nachteilig war allerdings, das das Probekit Framework keinen direkten Zugriff auf die erzeugten Objekte erlaubt. Das liegt daran, dass diese Objekte in einer anderen JVM-Instanz und damit einem anderen Speicherbereich als der eigenen Eclipse-JVM-Instanz erzeugt werden. Um trotzdem an die Objekte zu gelangen, sind diese zunächst auf der Festplatte zwischen zu speichern, um Sie dann in der eigenen Eclipse-JVM wieder einzulesen.

35 Sie werden beispielsweise nicht als ExtensionPoint angeboten.

36 Ein Ansatz könnte sein, mit der Aktivierung (`setEnabled()`) von Events zu arbeiten. Es könnte dazu ein eigener Filter implementiert werden, der zu gegebener Zeit die benötigten Events de/aktiviert (siehe [jdi]).

Darüber hinaus konnten Probes nur im Kontext des älteren Java Virtual Machine Profiler Interface (JVMPI) [jvmpi] verwendet werden . Eine Implementierung des Probekits auf Basis des neueren Java Virtual Machine Tooling Interface (JVMTI) [jvmti] lag zum Zeitpunkt der Implementierung des Prototypen (April 2008) nicht vor.

7.2 Vergleich mit verwandten Arbeiten

Es gibt einige Arbeiten, die sich mit der Analyse der Ursache fehlgeschlagener Testfälle beschäftigen. Häufig beruhen diese auf einer Form der dynamische Programmanalyse. Dieser Abschnitt beleuchtet diese Arbeiten und gibt Hinweise, wie sich diese Arbeiten von *EzUnit* unterscheiden.

Prof. Zeller und sein Team haben ein generelles Vorgehen für die Fehlersuche namens *Delta-Debugging* entwickelt [Zeller1999]. Es basiert auf der Idee, die jeweilige Eingabemenge (gleich ob Quelltext, Programmeingaben oder Programmstatus) solange zu halbieren, bis der Fehler isoliert ist. In einer Erweiterung des Vorgehens wird Bezug auf die Änderungen genommen, die zwischen der letzten funktionierenden Version und der nun fehlerhaften Version des Programms liegen. Diese Erweiterung wird *DDChange* genannt [DDChange]. Im Vergleich zu EzUnit macht DeltaDebugging keine Annahmen darüber (weder durch statische noch durch dynamische Programmanalyse), welche Änderungen des Quelltextes überhaupt für das Fehlschlagen eines Testfalles in Fragen kämen.

In eine ähnliche Richtung zielen Ansätze, die unter dem Begriff *Change impact Analyse (CIA)* zusammengefasst werden. Die CIA analysiert aufeinander folgende Versionen eines Programms miteinander und versucht, deren semantische Änderungen aus den Code-Änderungen abzuleiten. *Chianti* ist ein solches Analysewerkzeug [Chianti], welches atomare Fragmente des Quelltextes identifiziert, die für das Fehlschlagen eines Testfalles verantwortlich sein könnten. Basierend auf Chianti wurden Erweiterungen implementiert, die die identifizierten Änderungen klassifizieren [JUnitCIA], und die es dem Entwickler ermöglichen, eine bestimmte Auswahl an Änderungen auf den Quelltext anzuwenden [Crisp].

Chianti greift bei der Impact Analyse auf statische und dynamische Programmanalysen zurück. Im Gegensatz zu EzUnit wird allerdings kein öffentlich verfügbarer online Tracer verwendet. Chianti setzt vielmehr auf eine proprietäre offline Implementierung, die die Tracedaten durch Bytecode Instrumentierung gewinnt (ähnlich unseres Probe-Ansatzes 7.1 „Alternative Tracingansätze“).

8 Schlussbetrachtung

8.1 Ausblick

EzUnit konnte im Rahmen dieser Arbeit um einige Funktionen bereichert werden. Dennoch bietet sich immer Raum für Erweiterungen. Das folgende Kapitel listet diese möglichen Erweiterungen auf.

8.1.1 Integration des TPTP-Profilers

In Abschnitten 6.1 und 6.2 wurden schon das Laufzeitverhalten und der sehr hohe Speicherverbrauch von *EzUnit* angesprochen. Eine Verbesserung könnte erreicht werden, indem die erzeugten Tracedaten in eine Datei umgeleitet werden, wofür das TPTP-Framework bereits die nötige Funktionalität anbietet.

Wird das Umleiten in eine Datei aktiviert, baut das TPTP-Framework das *TraceModel* nicht im Speicher auf, sondern erzeugt eine Datei mit der Endung „*.trcxml“. Diese Datei ist ein mit zip/jar komprimiertes Archiv, das ein XML-Dokument mit den Tracedaten enthält (Listing 20 zeigt Beispieldaten). [37]

```
<?xml version="1.0"?>
<TRACE>
  <node nodeId="" hostname="localhost" ipaddress="127.0.0.1" timezone="-60"
time="1217880950.036750000"/>
  <agentCreate agentId="131adb92-2e37-4aa2-a035-a66870ec1c9b" version="2.000"
     processIdRef="ae2a72f4-5ada-45a6-b91d-eb2ccdc08e96" agentName="
     org.eclipse.tptp.jvmti" agentType="Profiler" agentParameters="server=controlled"
     time="1217880950.380000000"/>
  <traceStart traceId="4888c7eb-b493-4e2d-b7cd-869debfa48f6" agentIdRef=
      "131adb92-2e37-4aa2-a035-a66870ec1c9b" time="1217880953.670832494"/>
  <classDef name="org/apache/commons/beanutils/converters/IntegerConverterTestCase"
     sourceName="IntegerConverterTestCase.java" classId="390" time=
     "1217880960.980400888"/>
  <methodDef name="suite" signature="()Ljunit/framework/TestSuite;" startLineNumber="93"
     endLineNumber="93" methodId="70694" classIdRef="390"/>
  <methodEntry threadIdRef="3" time="1217880960.980503415" methodIdRef="70694"
     classIdRef="390" ticket="0" stackDepth="1"/>
  [...]
</TRACE>
```

Listing 20 ~ Beispieldaten eines umgeleiteten Traces

Dieses Vorgehen hat einige Vorteile: Einmal erzeugte Traces könnten als Referenztraces gespeichert werden. Insbesondere wenn mit neuen *FaultLocatoren* experimentiert wird, kann die Zeit für das Erstellen der Traces eingespart werden. Darüber könnte sichergestellt werden, dass immer die gleiche Datenbasis verwendet wird, und nicht unterschiedlich gesetzte Filtereinstellungen zu schwer erklärbaren Ergebnissen führen. Überdies könnten auch Fehlersituationen durch das „Zurechtbiegen“ eines Traces provoziert werden.

37 Durch einfaches Umbenennen in *.zip kann die Datei mit Hilfe von Standard Archivierungsprogrammen entpackt werden.

Bisher nicht abzuschätzen ist der zeitliche Aufwand des Einlesens eines *TraceModels*. Hier sind gesonderte Untersuchungen zu führen. Außerdem ist darauf zu achten, dass bereits während des Einlesens des *TraceModel* mit der Erzeugung der Traces begonnen wird, und abgearbeitete Elemente wieder aus dem Speicher entfernt werden, da sonst ähnliche Speicherprobleme wie bei der „online“ Variante zu erwarten sind.

8.1.2 Darstellung geänderter Methoden

EzUnit markiert geänderte Methoden derzeit mit einem „*“ in der *MUTSelectionView*. Darüber hinaus wäre es wünschenswert, auch direkt das Delta zur Vorgängerversion der Datei angezeigt zu bekommen. Hierzu bietet sich die bereits vorhandene *JavaStructuralCompareView*, ein Side-By-Side Editor an, die ein sehr übersichtliches Diff zweier Versionen einer Java Datei darstellen kann.

Die *MUTSelectionView* könnte nun so erweitert werden, dass sie sich wie folgt verhält:

1. Fall '*MUT* ist unverändert: Doppelklick in *MUT*-View öffnet die *MUT* im *JavaEditor*
2. Fall ‚*MUT* wurde verändert': Doppelklick in *MUT*-View öffnet die *MUT* im *DiffView*

8.1.3 Callgraph statt Trace (Aufrufreihenfolge)

Die derzeitig implementierte Datenstruktur der Traces (sowohl statisch als auch dynamisch) enthält keine Information über die Aufrufreihenfolge der *MUTs*. Dennoch könnte gerade in der falschen Aufrufreihenfolge der Fehler liegen.

Fraglich ist allerdings, wie die richtige Aufrufreihenfolge der *MUTs* ermittelt bzw. mit vertretbarem Aufwand spezifiziert werden kann. Eine Möglichkeit bestünde darin, die Reihenfolge der annotierten *MUTs* einer Testmethode zu analysieren. Diese Reihenfolge könnte als Soll-Reihenfolge für die jeweilige Testmethode interpretiert werden.

8.1.4 Automatischer Hinweis, wenn Filter „unglücklich“ gesetzt sind

Zur Zeit prüft *EzUnit* nicht die Sinnhaftigkeit des vom Benutzer erzeugten *FilterSets*. Gerade im Umgang mit mehreren geöffneten Projekten kann leicht vergessen werden, die Filter auf das neue Projekt anzupassen.

Es wäre daher sinnvoll, die gesetzten Filter gegen das gerade zu tracende Projekt zu prüfen. In dem Fall, in dem die Filter zu knapp gesetzt sind, und der Trace aus diesem Grund keine Daten enthält, sollte eine entsprechende Warnmeldung ausgegeben werden. Es wäre auch denkbar, die gefundenen Methoden in der *MUTSelectionView* in einer anderen Farbe darzustellen.

8.1.5 Abschaffung der MUT-Annotationen

Kapitel 6.2 hat gezeigt, dass die dynamischen Traces im Mittel dreimal kleiner sind, als die korrespondierenden statischen Traces. Um den gleichen Faktor sinkt auch die Anzahl der potentiellen Verursacher eines Fehlers. Die manuelle Annotation der Testmethoden könnte daher für diesen Zweck (Auffinden von *MUTs*) entfallen, um den Arbeitsfluss des Entwicklers weniger zu beeinträchtigten.

Die Annotationen könnten allerdings bei der Sicherstellung der Testabdeckung unterstützen. Hierzu würden Benutzer nur bestimmte, kritische *MUTs* annotieren, deren Aufruf sichergestellt werden soll. Weitere Tools (wie Clover [clover]), die auf das Thema „Testabdeckung“ spezialisiert sind, könnten hier ebenfalls Hilfestellung geben.

8.1.6 Integration mit Continuous Integration Frameworks

Die Erzeugung dynamischer Traces ist zur Zeit an die Eclipse-Plattform und seine Frameworks (wie *JUnitTestRunListener*, TPTP, etc.) gebunden. Es sollte zukünftig möglich sein, von den Vorteilen der dynamischen Traces auch im Rahmen von *Continuous-Integration Systemen* (wie beispielsweise *CruiseControl*) zu partizipieren. Hierzu müssten die dynamischen Traces allerdings standalone, also außerhalb der Eclipse Entwicklungsumgebung erzeugt werden können

Die dazu nötigen *AgentController* des TPTP können bereits jetzt standalone betrieben werden. Darüber hinaus müssten die aufgerufenen *JUnitTestCases* abgefangen werden (weil der *TestRunListener* im Ant-Task wahrscheinlich nicht funktioniert) und ein standalone Launcher implementiert werden. Dieser müsste das TPTP-Framework mit den nötigen Startparametern versorgen und schließlich die *ProfilingSession* starten.

8.1.7 Entwicklung eines Wahrscheinlichkeits-Frameworks

Neben der reinen Auflistung der potentiellen Verursacher eines Fehlers wäre es wünschenswert, *EzUnit* um ein Framework zu erweitern, welches diese Liste gemäß der Verursacher-Wahrscheinlichkeit sortiert.

Eine weitere Bachelorarbeit am Lehrstuhl Programmiersysteme beschäftigt sich derzeit mit Erweiterung von *EzUnit* um ein solches Wahrscheinlichkeits-Framework, die so genannten *LikelihoodFaultLocatoren*. Solch ein Wahrscheinlichkeits-Lokator bestimmt einen Wert für das Maß an Sicherheit, zu dem an einer bestimmten Stelle im Programm ein Fehler vorliegt.

8.1.8 Beheben kleiner „Unsauberkeiten“ bei der Programmanalyse

Die Erzeugung eines Traces beginnt bei beiden Tracetpyen (statisch wie dynamisch) mit der Testmethode selbst (Beispiel `testCurrency()`). Zur Aufrufsequenz eines Testfalles gehört aber ebenfalls die Methode `setUp()` sowie die Methode `tearDown()`. Sie sind dazu gedacht, die nötigen Vorbedingungen zur Ausführung des Testfalles (setUp), sowie alle Aufräumarbeiten nach dessen Durchführung zu erledigen (tearDown). Gerade in die Vorbereitung eines Testfalles können sich ebenfalls Fehler eingeschlichen haben, die mit der derzeitigen Implementierung nicht gefunden würden. Der Trace der Methode `setUp()` sollte daher zu dem der jeweiligen Testmethode hinzuaddiert werden.

Im statischen Trace sind nur die Methoden enthalten, bzw. können nur die Methoden inspiziert werden, deren Quelltext vorliegt. Die Methode `assertEquals()` des JUnit Frameworks liegt in der Regel nicht im Quellcode vor und kann daher nicht weiter untersucht werden. Dennoch wird durch `assertEquals()` implizit die Methode `equals()` des übergebenen Objektes aufgerufen, ist aber folglich nicht im statischen Trace enthalten. Um sich diesem Problem zu nähern, könnte man allerdings das Wissen darüber, dass `assertEquals()` die `equals()` Methode implizit aufruft hart codieren und damit den statischen Trace korrigieren.

8.2 Fazit

Die Integration der dynamischen Programmanalyse ist gelungen. Auch die erzielte Reduktion der potentiellen Verursachermethoden um den Faktor drei ist eine hilfreiche Erweiterung des EzUnit Plug-In.

Für einen praktischen Einsatz von EzUnit im täglichen Arbeitsablauf eines Entwicklers dürften die lange Laufzeit und der immense Speicherverbrauch allerdings ein Hemmnis sein.

9 Glossar

Begriff / Abkürzung	Erläuterung
MUT	**M**ethod **U**nder **T**est Hergeleitet vom JUnit-Begriff ClassUnderTest. Bezeichnet die Methoden, die innerhalb eines Testfalles aufgerufen werden.
AST	**A**bstract **S**yntax **T**ree
EzUnit	Bezieht sich auf das Produkt/Konzept an sich ohne eine spezielle Version zu referenzieren.
EzUnit1	Die Vorgängerversion von EzUnit, die im Rahmen der Arbeit von [Meyer2007] erstellt wurde.
EzUnit2	Die Version von EzUnit, die das Ergebnis dieser Arbeit reflektiert.
EzUnit3	Repräsentiert eine zukünftige Version von EzUnit.
TPTP	**T**est & **P**erformance **T**ools **P**latform Project ist ein Teilprojekt der Eclipse-Plattform. Es wurde 2002 unter dem Namen *Hyades* gegründet, später (2004) in TPTP umbenannt und zu dieser Zeit auch als Eclipse-Projekt aufgenommen

10 Verzeichnisse

Abbildung 1 ~ Erzeugung eines AST (Abbildung nicht endgültig)....................................6
Abbildung 2 ~ vereinfachte Architektur des DataCollection Framework....................................11
Abbildung 3 ~ Kommunikationsfluss des DataCollection Framework....................................12
Abbildung 4 ~ Klassendiagramm des TraceModel (Auszug) [emfUmlModelle]....................13
Abbildung 5 ~ Klassendiagramm des Paketes „trace“....................................15
Abbildung 6 ~ Beteiligte Objekte an der Trace Erstellung....................................17
Abbildung 7 ~ Schema der Eclipse Plattform Architektur....................................21
Abbildung 8 ~ Dialog „Filterkriterium bearbeiten“ EzUnit....................................23
Abbildung 9 ~ Dialog zum Editieren von FilterSets....................................25
Abbildung 10 ~ Übersicht Gestaltungselemente aus [help-workbench]....................................27
Abbildung 11 ~ MUTSelectionView....................................27
Abbildung 12 ~ SelectionService der Eclipse-Plattform....................................28
Abbildung 13 ~ Verzeichnisstruktur einer Eclipse Installation....................................32
Abbildung 14 ~ Konfiguration „Monitor“ - Launchconfiguration "Profile Mode"....................32
Abbildung 15 ~ Perspektive Profiling mit dem Ergebnis eines Traces....................................33
Abbildung 16 ~ Konfigurationsmöglichkeiten des JUnit TestRunners....................................35

Listing 1 ~ Das Interface IFaultLocator....................................8
Listing 2 ~ Auszug, Erstellung ILaunchConfiguration....................................10
Listing 3 ~ Starten einer LaunchConfiguration....................................11
Listing 4 ~ Suchmethode zum Auffinden einer IMethod im TRCModel....................................14
Listing 5 ~ Aufbau der Supertyp-Hierarchie....................................14
Listing 6 ~ Delegation der Traceerzeugung an den StaticTraceASTVisitor....................................16
Listing 7 ~ Auszug der Methode buildTraceSync()....................................18
Listing 8 ~ Job Familie und Schlüssel-Objekt....................................18
Listing 9 ~ Ausschnitt aus der Methode waitUntilDependentJobsFinished()....................................19
Listing 10 ~ aus ProgrammaticLaunchHelper: Löschen/Entladen alter TraceModelle............20
Listing 11 ~ Deklaration der Beispiellokatoren....................................21
Listing 12 ~ Zugriff auf die ExtensionRegistry....................................22
Listing 13 ~ Implementierung der Schnittstelle TestRunListener....................................22
Listing 14 ~ Konvertierungsmethoden in die verschiedenen FilterSets....................................24
Listing 15 ~ XML Repräsentation eines FilterSets im PreferenceStore....................................25
Listing 16 ~ ExtensionPoint ‚views’ aus plugin.xml....................................26
Listing 17 ~ Implementierung ISelectionListener in MUTSelectionView....................................29
Listing 18 ~ Extraktion eines IMethod Objektes aus ISelection....................................29
Listing 19 ~ Implementierung von IMethodEventListener in MethodUnderTestStore............31
Listing 20 ~ Beispieldaten eines umgeleiteten Traces....................................41

[BeckGamma2004] Kent Beck, Erich Gamma (2004): Contributing to Eclipse, Addison-Wesley, Boston

[Chianti] Ren, X., Ryder, B. G., Stoerzer, M., and Tip, F. 2005. Chianti: a change impact analysis tool for java programs. In *Proceedings of the 27th international Conference on Software Engineering* (St. Louis, MO, USA, May 15 - 21, 2005). ICSE '05. ACM, New York, NY, 664-665. DOI= http://doi.acm.org/10.1145/1062455.1062598

[clover] Webseite des Herstellers Atlassian, http://clover.atlassian.com/

[Crisp] Chesley, O. C., Ren, X., Ryder, B. G., and Tip, F. 2007. Crisp--A Fault Localization Tool for Java Programs. In *Proceedings of the 29th international Conference on Software Engineering* (May 20 - 26, 2007). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 775-779. DOI= http://dx.doi.org/10.1109/ICSE.2007.29

[Daum2005] Dr. Berthold Daum (2005): Java-Entwicklung mit Eclipse 3.1, dpunkt.verlag, Heidelberg

[DDChange] Webseite des Projektes DDChange, http://ddchange.martin-burger.de/

[eclipsevon2004Ui] Arthorne Lemieux, Writing responsive UIs using the Eclipse 3.0 concurrency architecture, http://www.eclipsecon.org/2004/EclipseCon_2004_Technical TrackPresentations/39_Arthorne-Lemieux.pdf

[eclipseOrg] About Us der Eclipse Foundation, http://www.eclipse.org/org/

[emfProject] Webseite des Eclipse EMF-Projektes, http://www.eclipse.org/modeling/emf/

[emfRedBook2004] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, Philippe Vanderheyden (2004), Eclipse Development - using the Graphical Editing Framework and the Eclipse Modeling Framework, ibm.com/redbooks

[emfUmlModelle] UML Darstellung der vier vom TPTP Projekt verwendeten EMF-Modelle, http://www.eclipse.org/tptp/platform/documents/resources/models/index.html

[GoF1996] Erich Gamma (1996): Elemente wieder verwendbarer objektorientierter Software, Addison-Wesley-Longman

[help-concurrency] Beschreibung der Eclipse Concurrency Infrastruktur (Auszug aus dem Eclipse Hilfesystem), http://help.eclipse.org/help33/topic/org.eclipse.platform.doc.isv/guide/ runtime_jobs.htm

[help-probekit] Beschreibung des Eclipse Probebit (aus dem Eclipse Hilfesystem), http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.hyades.probekit.doc.user/topics/ c_toc_probekit.htm

[help-workbench] Übersicht über die Workbench aus dem Eclipse Hilfesystem „Workbench under the covers“, http://help.eclipse.org/help33/topic/org.eclipse.platform.doc.isv/guide/workbench_structure.htm

[howtoDatacollector] How-To zur Erstellung eines eigenen Datenkollektors (gute Einführung in das DataCollection Framwork), http://www.eclipse.org/tptp/platform/documents/drafts/HowtowriteaTPTPDataCollectionAgent.htm

[jdi] Javadoc mit der Erläuterung der setEnabled()-Methode, http://java.sun.com/javase/6/docs/jdk/api/jpda/jdi/index.html?com/sun/jdi/request/EventRequest.html

[JUnitCIA] Stoerzer, M., Ryder, B. G., Ren, X., and Tip, F. 2006. Finding failure-inducing changes in java programs using change classification. In *Proceedings of the 14th ACM SIGSOFT international Symposium on Foundations of Software Engineering* (Portland, Oregon, USA, November 05 - 11, 2006). SIGSOFT '06/FSE-14. ACM, New York, NY, 57-68. DOI= http://doi.acm.org/10.1145/1181775.1181783

[jUnitOrg] Webseite des JUnit Frameworks, http://junit.org/home

[jpda] Dokumentation der *Java Platform Debugger API,* http://java.sun.com/javase/technologies/core/toolsapis/jpda/

[jvmpi] Dokumentation des Java Virtual Maschine Profiler Interface (JVMPI), http://java.sun.com/j2se/1.5.0/docs/guide/jvmpi/jvmpi.html

[jvmti] Dokumentation des Java Virtual Maschine Tooling Interface (JVMTI), http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html

[launching] Artikel aus dem Eclipse-Corner mit dem Titel „We Have Lift-off: The Launching Framework in Eclipse“, http://www.eclipse.org/articles/Article-Launch-Framework/launch.html

[Meyer2007] Nils Meyer (2007): Ein Eclipse-Framework zur Markierung von logischen Fehlern im Quellcode

[reqTptp] Systemanforderung für das TPTP Framwork, http://www.eclipse.org/tptp/

[selectionFramework] Artikel aus dem Eclipse-Corner zur Handhabung des Eclipse Selection Framework, http://www.eclipse.org/articles/Article-WorkbenchSelections/article.html

[sschmidt2007] Konzeption und Entwicklung eines PlugIn-basierten Diagnosesystems für semantische Qualitätsdefekte, Diplomarbeit am Fraunhofer Institut für Experimentelles Software Engineering

[Steimann2007]: Steimann, et al. 2007, “EzUnit: A Framework for Associating Failed Unit Tests with Potential Programming Errors”

[tptpProject] Webseite des Eclipse TPTP Plattform Projektes, http://www.eclipse.org/tptp/platform/documents/design/arch_tptp_platform.html

[Zeller1999] A Zeller “Yesterday, my program worked. Today, it does not. Why?” in: *ESEC / SIGSOFT FSE* (1999) 253–267.

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Thomas Eichstädt-Engelen

Krefeld, den 12. September 2008