Freie Universität Berlin
Fachbereich Mathematik und Informatik
Institut für Informatik

Studienarbeit

Thema:

Animation vergangener Codeänderungen von Java-Methoden

vorgelegt von Marco Kranz
mkranz@inf.fu-berlin.de
November 2006

betreut von Sebastian Jekutsch
Arbeitsgruppe Software Engineering

Gliederung

1. Einführung

1.1. Wissenschaftlicher Hintergrund der Arbeit

a) Qualitätssicherung in der Softwareentwicklung

Spricht man heutzutage von Qualitätssicherung im Bereich der Softwareentwicklung, so denkt man dabei häufig zunächst an testbasierte Möglichkeiten der Sicherstellung von Qualität. In diesem Zusammenhang geht es vor allem darum, durch Ausprobieren beziehungsweise Ausführen einzelner Softwarekomponenten oder kompletter Programme Fehlverhalten aufzudecken, welches auf Defekte innerhalb der Software hinweist.
Diesem weit verbreiteten Ansatz liegt der Gedanke zugrunde, dass das Auftreten von Defekten in Software gewissermaßen eine unvermeidliche Tatsache ist, was sicherlich mit darin begründet liegt, dass die Vorgänge, die überhaupt erst zum Auftreten solcher Fehlern führen, auf den ersten Blick weder klar erkennbar noch strukturierbar erscheinen.
Ein völlig gegensätzlicher Ansatz ist die Idee der Qualitätssicherung durch Fehlervermeidung. Dabei wird davon ausgegangen, dass Defekte in Software nicht nach einem zufälligen System auftreten, sondern viel mehr Ursache konkreter Ereignisse im Entwicklungsprozess sind, welche sich erkennen und beobachten lassen. Dementsprechend ist die Betrachtung des detaillierten Entwicklungsprozesses für den Erfolg dieser Methode von großer Bedeutung. Ziel der Forschung in diesem Bereich ist es unter anderem, durch eine automatisierte Analyse des Entwicklungsprozesses Handlungsabläufe bereits zum Zeitpunkt des Geschehens als potentiell fehlerverursachend zu erkennen, und den Handelnden (den Entwickler) direkt auf ein bestehendes Fehlerrisiko aufmerksam machen zu können.

b) Der Mikroprozess der Softwareentwicklung

Der Begriff des Mikroprozesses[1] beschreibt den eigentlichen Vorgang des Erzeugens von Code oder anderer entwicklungsspezifischer Dokumente im

Gegensatz zum allgemeinen Prozess, welcher nur die Stationen der Entwicklung, nicht jedoch deren Erreichen dokumentiert.
Der Mikroprozess des Programmierens besteht bei genauer Betrachtung aus atomaren Bestandteilen, wie zum Beispiel dem Verändern von Code, dem Suchen im Code, dem Öffnen oder Schließen von Dateien oder auch einfach nur Phasen der Untätigkeit. Diese atomaren Bestandteile wiederum lassen sich oftmals bestimmten Handlungsabläufen, so genannten Episoden[1] zuordnen. Solche Episoden beschreiben für die Softwareentwicklung typische Handlungsabläufe, wie etwa trial-and-error Phasen oder das Erstellen neuer Codeabschnitte durch das Kopieren und Verändern anderer Abschnitte.

c) Beobachtung des Mikroprozesses

Die Beobachtung des Entwicklungsprozesses erweist sich in der Praxis als schwierige Aufgabe. Um die Möglichkeit der Analyse des Mikroprozesses zu haben, muss dieser zunächst aufgezeichnet werden. Dies kann beispielsweise in Form von Videoaufzeichnungen oder 'screen capturing' (Aufzeichnung der Bildschirmwiedergabe, wie sie der Programmierer sieht) geschehen, oder aber durch direktes protokollieren der Veränderungen der zugrunde liegenden 'Arbeitsmaterialien' (Öffnen oder Schließen von Dateien, Bearbeiten von Dateien). Um das direkte Aufzeichnen von Veränderungen zu ermöglichen, sind so genannte Software-Sensoren erforderlich, welche, in die jeweilige Entwicklungsumgebung des Programmierers integriert, in der Lage sind, alle Aktionen automatisch zu protokollieren. Um die Analyse der auf diese Weise gewonnenen Daten zu unterstützen, werden Softwaretools verwendet, welche die Daten aufarbeiten und in übersichtlicher Form darstellen können.

1.2. Ziel der Arbeit – Hilfsmittel zur Analyse des Mikroprozesses

Ziel dieser Arbeit war die Erstellung eines Softwaretools zur Visualisierung von Veränderungen in Dokumenten innerhalb eines bestimmten Beobachtungszeitraums. Die Idee war eine Software, welche in der Lage sein

sollte, einen zuvor aufgezeichneten Programmiervorgang in einer Art 'Replay' wiederzugeben. Ein 'Replay' kann man sich in diesem Zusammenhang wie eine Art Film vorstellen, bei dem durch das Zusammenfügen der Einzelschritte eine anschauliche Wiederholung des Entstehungsvorgangs möglich ist.

Die Aufzeichnung wird mit Hilfe eines Sensors durchgeführt, welcher in die Entwicklungsumgebung integriert ist, und dort während des Programmiervorgangs die schrittweisen Veränderungen protokolliert. Die Daten entsprechen dabei einem beobachteten Codebereich, etwa dem Inhalt einer Datei oder bestimmten Teilen davon.

Das entwickelte Programm soll vor allem zur explorativen Beobachtung von Programmiertätigkeiten dienen, um auf diese Art Hinweise auf eventuelle Zusammenhänge zwischen bestimmten Vorgängen oder Vorgehensweisen und dabei auftretenden Fehlern zu erkennen.

2. Analyse und Entwurf

2.1. Anforderungsbeschreibung

Die nun folgenden Anforderungen sind im Verlauf der Entwicklung der Software entstanden. Als Ausgangsbasis diente eine Auflistung von Anforderungen, die bei der ursprünglichen Ausarbeitung der Aufgabenstellung erstellt wurde. Diese wurde dann im Verlauf der Entwicklung wiederholt besprochen und den sich verändernden Anforderungen angepasst. Einfluss auf die Gestaltung hatten dabei zum einen sich verändernde Vorstellungen über den späteren Einsatz der Software, zum anderen entwicklungstechnische Bedingungen oder Beschränkungen, welche sich im Verlauf der Entwicklung ergaben.

a) Funktionale Anforderungen

Allgemeines:
Die zu erstellende Software soll in der Lage sein, zuvor aufgezeichnete Daten aus einer Datei einzulesen. Bei den Daten handelt es sich um die schrittweise Aufzeichnung von Veränderungen in Java Code. Diese sind pro Klasse aufgeschlüsselt nach Methoden in einer Art Video animiert darzustellen, so dass man die Evolution einer Methode über die Zeit leicht erkennen kann.

Darstellung:

- → Die Darstellung des Codes soll in einem Fenster ähnlich dem eines Editors erfolgen.
- → Innerhalb der Codedarstellung soll für jeden Schritt erkennbar sein, was sich im Gegensatz zum vorherigen Schritt verändert hat. Dies kann zum Beispiel in Form von farblicher Markierung der entsprechenden Codeteile passieren.
- → Es soll - in Form eines Baums von animierbaren Methoden, gruppiert nach Klassen - die interessierende Methode ausgewählt werden können.

Steuerung:

- → Es sollte eine Handsteuerung (ähnlich wie bei Video- oder Audioplayern) geben.
- → Neben dem normalen Starten und Stoppen des Replay's soll die Möglichkeit geben, schnell große Bereiche zu überbrücken und zu beliebigen Ereignissen zu springen. Dies könnte beispielsweise mit Hilfe eines Sliders geschehen.
- → Es sollte eine zeitproportionale Abfolge (also proportional gleiche Geschwindigkeit wie im Original) oder eine einfache Aneinanderkettung der Änderungen zur Auswahl stehen.
- → Für die zeitproportionale Abfolge wäre ein einstellbarer Geschwindigkeitsfaktor im Verhältnis zum Original wünschenswert (1/10 bis 10), ebenso verschiedene Zeitabstände im Fall der Aneinanderkettung (0 bis 5 Sec.).
- → Bei zeitproportionaler Abfolge sollten Pausen ab einem bestimmten Schwellwert überbrückt werden.

Erweiterbarkeit:

Beim Entwurf wird darauf Wert gelegt, dass er eventuelle spätere Erweiterungen der Software nach Möglichkeit unterstützt. Ebenso sollte die Dokumentation des Codes und des Entwurfs diese erleichtern. Geplante zukünftige Erweiterungen sind:

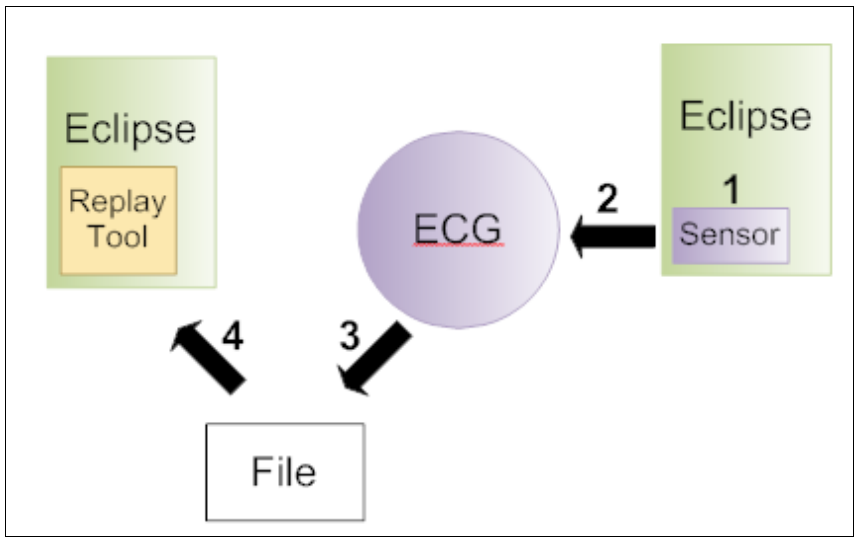
- → Auf Wunsch sollte am Farb-/Grauton einer Codezeile erkennbar sein, wie "alt" sie ist, also wie lange sie nicht mehr geändert wurde.
- → Verallgemeinerung von Methoden auf ganze Klassen oder auf Teile von Methoden.
- → Verallgemeinerung von Java auf andere Programmiersprachen.

b) Technische und datenspezifische Anforderungen

Als Grundlage für die Form der Daten dienen Ereignisse vom Typ „CodeChange“ aus dem Projekt ElectroCodeoGram(ECG)[a]. Diese liegen in einem XML-Format vor. Die Ereignisse des Typs „CodeChange“ beschreiben Veränderungen auf Dateiebene, im Rahmen dieser Arbeit sollen jedoch Veränderungen innerhalb von Methoden(in Javacode) beobachtbar gemacht werden. Dies kann durch Auswertung aufgezeichneter „CodeChange“-Ereignisse geschehen, es ist jedoch auch eine Erweiterung des ECG Eclipse Sensors möglich, um geeignete Ereignisse direkt zu protokollieren. Ich entschied mich letztendlich dafür, einen neuen Ereignistyp zu definieren(siehe Kap. 2.2.a).
Als Darstellungsrahmen soll das Eclipse Framework[b] verwendet werden, das heißt, die zu erstellende Software soll als Eclipse Plugin entwickelt werden.

c) Zusammenhang der einzelnen Komponenten

Die folgende Abbildung soll einen kurzen Überblick über den Zusammenhang der beteiligten Komponenten geben.



1. Protokollieren der Ereignisse in der Eclipse Entwicklungsumgebung.
2. Senden der Ereignisse an das Electrocodeogram.
3. Speichern der empfangenen Ereignisse im Dateisystem.
4. Einlesen der gespeicherten Ereignisse ins Replay Tool.

2.2. Wichtige Designentscheidungen

a) Auswahl des Ereignistyps

Der als Ausgangsbasis vorgegebene „CodeChange“-Ereignistyp enthält jeweils den kompletten Inhalt einer Datei. Im Falle des ECG Eclipse Sensors ist dies eine Javaklasse. Die Aufzeichnung eines Ereignisses im Eclipse Sensor wird durch Pausen in der Programmiertätigkeit ausgelöst, das heißt, nach jeder Veränderung im Code, auf die eine Pause(von 1-2 Sekunden) folgt, wird der aktuelle Inhalt der Datei aufgezeichnet. Eine beobachtete Veränderung wird also jeweils aus dem Unterschied zweier aufeinander folgender Ereignisse bestimmt. Dieser Unterschied('difference' oder kurz „Diff“) muss durch Vergleichen der beiden Dateien bestimmt werden, er ist nicht explizit im Ereignis enthalten.
Um die vorhandenen „CodeChange“-Ereignisse zu nutzen, wäre es nötig gewesen, jedes dieser Ereignisse zu parsen, um so die enthaltenen Methoden zu extrahieren. Des weiteren hätte für jedes einzelne Ereignis festgestellt werden müssen, ob es sich dabei um eine Veränderung innerhalb oder außerhalb einer Methode handelt.
Das Parsen von Code ist nicht nur eine komplexe Angelegenheit, sondern in diesem Fall auch ein unnötiger Aufwand, da der Code auf Seiten von Eclipse zum Zeitpunkt der Aufzeichnung bereits in geparster Form vorliegt. Dies war ein entscheidendes Argument für die Erweiterung des bestehenden Eclipse Sensors um einen neuen Ereignistyp. Das Auftreten redundanter Aufzeichnung von Ereignissen in mehreren Ereignistypen wurde dabei von mir als akzeptabel eingeschätzt.
Die hierfür nötige Einarbeitung in das Eclipse-interne 'Java Model' erwies sich als nicht ganz einfach, da die Dokumentation in diesem Bereich des Eclipse SDK unzureichend ist. Die wichtigste Erkenntnis war, dass sich innerhalb des 'Java Model' Veränderungen auf Methodenebene beobachten lassen, und somit das Parsen von Code entfällt. Die direkte Änderung am Code wird jedoch leider nicht unmittelbar ersichtlich. Um die tatsächliche Veränderung festzustellen, muss man den alten und neuen Codeabschnitt miteinander vergleichen. Dies erscheint vielleicht auf den ersten Blick als nicht weiter von Bedeutung, bei näherer Betrachtung wird man jedoch feststellen, das sich auf diese Weise nicht jede

Veränderung innerhalb des Codes eindeutig nachvollziehen lässt(siehe dazu Kap. 2.2.d).

b) Definition eines neuen Ereignistyps

Die Entscheidung, den vorhandenen Eclipse Sensor zu erweitern, bedeutete gleichzeitig auch die Definition eines neuen Ereignistyps(„ExactCodeChange“). Das Format musste dem der ECG Ereignisse angepasst sein, da die Ereignisse einerseits vom ECG Sensor versendet und andererseits vom ECG selbst aufgezeichnet werden sollten.
Weiterhin musste darauf geachtet werden, dass es sich zur Beschreibung beliebiger Code- bzw. Dokumentänderungen eignet, und nicht auf die Beschreibung von Java Codeänderungen beschränkt ist.

Folgende ereignisspezifische Daten werden aufgezeichnet:

- → Pfad: Der Pfad gibt den Ursprungspunkt des Ereignisses relativ zur Gesamtstruktur wieder. Im Falle des neuen „ExactCodeChange“-Ereignisses ist dies der Klassenname. Es wäre beispielsweise auch möglich, dem noch den Paketnamen voranzustellen. Dies ermöglicht die spätere Darstellung der Ereignisse in Form eines Verzeichnisbaumes.
- → Art der Änderung: Ein Schlüsselwort, welches Hinweise auf die Ursache des Ereignisses liefert. Es gibt vier Basisursachen für ein Ereignis („added“, „changed“, „deleted“, „identifier_changed“), sowie eine Reihe von Schlüsselworten, die detailliertere Auskunft über die Art der Änderung geben. Eine komplette Liste ist in der Schemadatei zu finden. „identifier_changed“ bezeichnet dabei eine mögliche Änderung des Ereignisidentifiers auf Seiten von Eclipse (dieses Feld ist nicht Ereignis- sondern Sensorspezifisch, d.h. Für Sensoren, welche Ereignisse mit konstanten „Identifiern“ aufzeichnen, ist diese Feld irrelevant).
- → Der Name des beschriebenen Elementes, in unserem Fall der Name der Methode.

- → Der „Identifier“: Ein eindeutiger „Identifier“, welcher die Zuordnung eines einzelnen Ereignisses zu einem beobachteten Element ermöglicht (der Name eines Elementes ist als „Identifier“ nicht geeignet, da er beispielsweise im Falle von Javamethoden, weder eindeutig noch konstant ist).
- → Das Element selbst: Der komplette Code bzw. Textabschnitt, den das beobachtete Element zum Zeitpunkt der Aufzeichnung umfasst.
- → Im Falle einer Ereignisidentifieränderung der neue „Identifier“.

Die Struktur des neuen Ereignistyps ist in der Schemadatei msdt.exactcodechange.xsd definiert.

c) Vorgehensweise bei der Erstellung eines Replays

Eine grundsätzliche Frage war die nach der Vorgehensweise bei der Erstellung eines Replays. Wie lässt sich der Unterschied zwischen zwei Elementen möglichst einfach bestimmen und anzeigen? Was bedeutet das für die Aufbereitung der zugrunde liegenden Daten?
Es gab dazu von Anfang an zwei denkbare Vorgehensweisen. Die erste Idee war, das Replay als Summe der beobachteten Veränderungen zu betrachten. Dies bedeutet nichts weiter, als das man jeden Zustand des beobachteten Elements einfach durch das 'Wiederholen' der bis dahin geschehenen Veränderungen erzeugen kann. Diese Herangehensweise war vor allem gegen Anfang der Arbeit im Gespräch, als noch nicht entschieden war, ob der neue Ereignistyp nur die Veränderungen oder das komplette Element zum Zeitpunkt der Aufzeichnung enthalten soll. Diese Lösung birgt jedoch gewisse Probleme in sich. Zunächst ist es offensichtlich, dass das ständige neu Zusammensetzen eines Zustandes einen nicht unwesentlichen Aufwand darstellt, der sich unter Umständen sogar merklich auf die Leistungsfähigkeit der Software auswirken könnte. Des Weiteren ist das Zusammensetzen selbst ein nicht-trivialer (also fehleranfälliger) Vorgang, dessen Funktionsweise und Komplexität schwer abzuschätzen war. Zum einen würde sich jeder eventuell auftretende Fehler beim Wiederherstellen eines Zustands auf alle Folgezustände auswirken. Außerdem war unklar, ob einzelne

Zustandsänderungen nicht eventuell aus mehreren Teiländerungen(„Substrings“) bestehen könnten, bei denen dann ebenfalls wieder eine Abhängigkeit voneinander bestanden hätte (Abhängigkeiten kommen dadurch zustande, dass jede Veränderung eine Positionsangabe beinhaltet, welche jedoch relativ zum Gesamtzustand ist).
Die zweite Möglichkeit bestand darin, für jeden Zustand den kompletten Inhalt des Elements sowie die Differenz zum vorangegangenen Zustand zu erfassen. Die Anzeige eines Zustands wäre damit nur noch von jeweils zwei aufeinander folgenden Zuständen abhängig. Dies bedeutete zwar einen erhöhten Speicherbedarf(sowohl in Dateiform als auch zur Laufzeit) durch das zusätzliche Vorhalten der kompletten Inhalte, dies erschien jedoch nach grober Abschätzung als nicht problematisch (bei einer durchschnittlichen Zeichenanzahl von beispielsweise 1000 pro Ereignis und verfügbarem Speicher von 512MB wären rund 2.600.000 Ereignisse speicherbar,[Java „char“ size: 16 Bit]). Die Wahl fiel letztlich auf die zweite Methode, da die technische Umsetzung einfacher erschien, und sie von mir insgesamt als wesentlich weniger Fehleranfällig eingeschätzt wurde.

d) Bestimmung der „Differenz“ zwischen einzelnen Zuständen

Bei der Frage nach der Bestimmung der „Differenz“ zwischen zwei Zuständen ging es vor allem um die Wahl eines geeigneten Algorithmus. Eine weitere zu treffende Entscheidung betraf den Zeitpunkt, an dem der „Diff“ berechnet werden sollte. Man konnte ihn bereits zum Zeitpunkt der Aufzeichnung berechnen (was bedeuten würde, dass der „Diff“ im gesendeten Ereignis enthalten sein müsste), oder ihn auf Seiten des „Replaytools“ bestimmen. Ich entschied mich letztendlich dafür, ihn im „Replaytool“ erzeugen zu lassen, und das aus zweierlei Gründen. Zum einen befürchtete ich, dass das Berechnen des „Diffs“ ein leistungskritischer Vorgang sein könnte, was eventuell zur Behinderung des Programmierers während der Aufzeichnung geführt hätte (zu diesem Zeitpunkt ging ich von einem Aufwand von $O(n^2)$ für die berechnung des „Diffs“ aus). Zum anderen machte die Verschiebung des Algorithmus ins „Replaytool“ Sinn, da zu vermuten war, dass bei der Entwicklung zukünftiger Sensoren diese Frage erneut auftreten würde

(und dem entsprechenden Entwickler somit eine mögliche Lösung zur Verfügung gestellt wird).
Zur Beantwortung der Frage nach einem geeigneten Algorithmus recherchierte ich zunächst Lösungen für das allgemeine Problem der Bestimmung von Unterschieden zwischen zwei Textabschnitten.

Longest Common Subsequence:
Beim „longest common subsequence problem“ geht es darum, für eine Menge von Sequenzen(zum Beispiel Zeichenketten) die längste mögliche Teilsequenz zu ermitteln, die allen gemeinsam ist. Zu beachten ist hierbei, dass eine Teilsequenz nicht gleich einem „substring“ ist. Während ein „substring“ einen zusammenhängenden Bereich von Zeichen beschreibt, ist eine „subsequence“ lediglich eine beliebige Teilmenge von Zeichen, welche zwar in dieser Reihenfolge, jedoch nicht zusammenhängend in der betrachteten Sequenz vorkommen müssen. Die „longest common subsequence“ zweier Zeichenketten ist also nichts anderes als eine Sequenz bestehend aus allen Zeichen, die diese beiden gemeinsam haben.
Es handelt sich hierbei um ein klassisches Problem der Informatik, zu dessen Lösung bereits eine Vielzahl von Algorithmen entwickelt wurden. Die meisten von ihnen basieren auf der Vorgehensweise des „dynamischen Programmierens“ (siehe [X]). Die Laufzeit solcher Algorithmen liegt bei $O(n^2)$.
Eine der bekanntesten Anwendungen eines solchen Algorithmus ist das Unix-Programm „diff“, welches zeilenweise die Unterschiede zwischen zwei Textdateien bestimmt.
Bei der Bestimmung der Differenz zwischen zwei Zuständen ihm Rahmen der Arbeit war es jedoch nötig, alle Zeichen zu bestimmen, die *nicht* in beiden Elementen vorkommen, also sozusagen eine „longest uncommon subsequence“.
Dazu wurde der verwendete Ausgangsalgorithmus dahingehend verändert, dass aus dem zweiten der beiden Eingangszustände alle Zeichen bestimmt werden, die nicht Teil der Subsequenz sind.
Dabei stellte sich dann jedoch heraus, dass dieser Algorithmus in vielen Fällen nicht das gewünschte Ergebnis liefert. Das Problem war, dass zwar alle Zeichen gefunden wurden, welche den Unterschied zwischen zwei Zuständen ausmachten,

dadurch jedoch nicht zwangsläufig auch die Positionen, an denen die Veränderung tatsächlich stattgefunden hat. Da dies auf den ersten Blick vielleicht nicht ganz klar wird, hier ein kurzes praktisches Beispiel:

String 1: **T h e M a t r i x**

String 2: **T h e T h e M e M a t r i x**

Änderung: **T h e T h e M e M a t r i x**

lcs: **T h e T h e M e M a t r i x**

Ergebnis: **T h e T h e M e M a t r i x**

String 1 stellt in diesem Beispiel den alten Zustand dar, *String 2* den neuen. *Änderung* zeigt die tatsächlich vorgenommene Änderung. *lcs* zeigt ein mögliches Ergebnis einer „longest common subsequence“. *Ergebnis* schließlich zeigt die durch den Algorithmus bestimmte Veränderung. Wie unschwer zu erkennen ist, weicht das berechnete Ergebnis nicht nur bei der Position der einzelnen Zeichen, sondern sogar bei deren Reihenfolge, von der Erwartung(der tatsächlichen Veränderung) ab.
Da solche abweichenden Ergebnisse häufiger auftreten, und diese die Darstellung der Veränderungen nachhaltig beeinflussen, erwies sich dieser Algorithmus als zur Lösung dieser speziellen Problemstellung unbrauchbar.

Alternativer Algorithmus:
Bei den Tests des lcs-Algorithmus hatte sich unter anderem gezeigt, dass die Veränderung zwischen zwei Zuständen in den meisten Fällen auf einen zusammenhängenden Bereich beschränkt ist. Bei näherer Betrachtung ist dies auch leicht zu erklären, da es sich bei den beobachteten Veränderungen ja im weitesten Sinne um die Arbeit einer Person an einer Textdatei handelt, und es

somit gewissermaßen in der Natur der Sache liegt, dass eine einzelne Änderung am Dokument immer in genau einem Bereich stattfindet(da man normalerweise nicht an zwei Stellen gleichzeitig Arbeiten kann). Eine Ausnahme stellen dabei Hilfsmittel der Arbeitsumgebung, wie beispielsweise *Code Completion* in Eclipse, dar.

Somit ließ sich die Problemstellung auf die Erkennung eines einzelnen veränderten Bereichs zwischen zwei Zuständen eingrenzen. Dieses Problem war nun vergleichsweise einfach zu lösen, und ich wählte dafür folgenden naheliegenden Algorithmus.

Es wird, wie beschrieben, zunächst vorausgesetzt, dass der Unterschied zwischen zwei Zuständen in genau einem geschlossenen Bereich liegt. Ist dies der Fall, so kann man weiterhin sagen, dass dieser neue Bereich entweder den Inhalt des alten Zustands in der Mitte teilt, oder am Anfang bzw. am Ende des Inhalts des alten Bereichs liegt(teilweises oder komplettes Ersetzen des alten Inhalts ist dabei möglich). Um den veränderten Bereich zu bestimmen, reicht es also, den Inhalt der beiden Zustände zu erst von rechts und danach von links beginnend, Zeichen für Zeichen miteinander zu vergleichen. Man hat ein Ende des neuen Bereichs gefunden, sobald man auf zwei ungleiche Zeichen(oder das Ende des Strings) stößt. Zur Verdeutlichung der Vorgehensweise auch hier wieder ein kurzes Beispiel:

String 1: **T h e M a t r i x**

String 2: **T h e T h e M e M a t r i x**

Änderung: **T h e T h e M e M a t r i x**

T h e **M a t r i x**

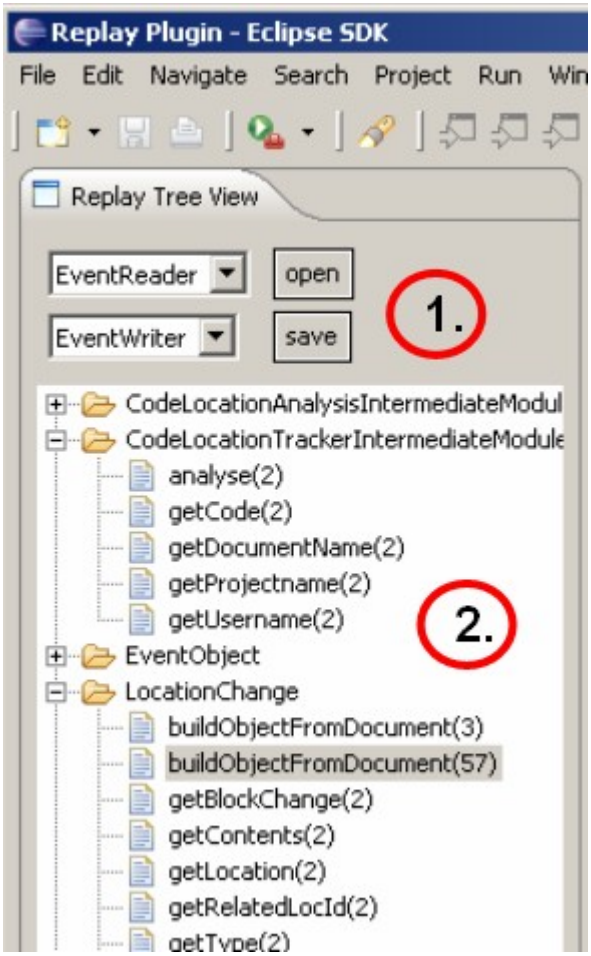
| | | **| | | | |**

Ergebnis: **T h e T h e M e M a t r i x**

String 1 und *String 2* stellen wieder den alten bzw. neuen Zustand dar, *Änderung* das, was tatsächlich verändert wurde. Das *Ergebnis* entspricht, wie man sieht, der tatsächlichen Änderung, die roten Linien sollen hier die Vorgehensweise des Algorithmus verdeutlichen.

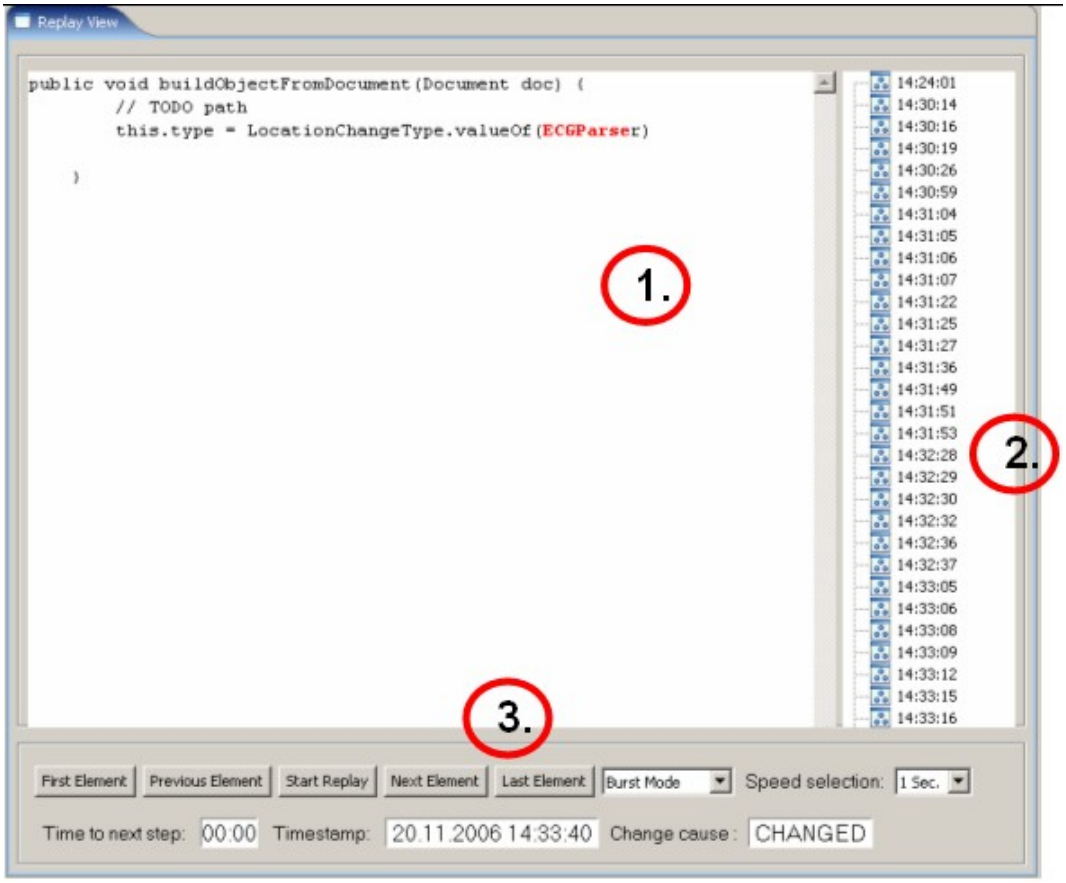
Die Tatsache, dass der Algorithmus in Situationen, in denen mehrere veränderte Bereiche auftreten, lediglich in der Lage ist, den kleinsten alle Änderungen umfassenden Bereich als Ergebnis zu liefern, ist für die Gesamtdarstellung der Veränderungen eine vertretbare Einschränkung. Dies sowie die mit O(n) günstige Laufzeit des Algorithmus gaben den Ausschlag für seine Verwendung.

3. Erläuterung der GUI Funktionalität



1. Möglichkeit zum Laden bzw. Speichern von Replays. Die Auswahlboxen links daneben(EventReader, EventWriter) geben die Möglichkeit, je nach Dateiformat einen passenden Reader / Writer auszuwählen.

2. Zeigt eine Übersicht über alle vorhandenen Replays und dient außerdem zur Auswahl eines Replays. In diesem Beispiel sind Methoden als einzelne Replays, sortiert nach Klassen, zu sehen.



1. Das (read only) Editorfenster. Hier wird der jeweilige Zustand angezeigt, Text in roter Schrift zeigt dabei eine Veränderung im Vergleich zum vorausgegangenen Element an.

2. Eine Liste aller Einzelzustände des ausgewählten Replays. Über diese Liste ist ein schnelles springen zu einem beliebigen Zustand möglich.

3. Die Bedienelemente. Buttons zum starten/stoppen sowie zum schrittweisen durchlaufen des Replays. Weiterhin besteht die Möglichkeit, einen von zwei r*eplay modes* auszuwählen(*realtime* oder *burst mode*). Im *realtime mode* wird das Replay in Echtzeit wiedergegeben, im *burst mode* in einer fest einstellbaren Zeit pro Schritt(siehe *speed selection*). Es gibt außerdem noch Anzeigen für die Zeit bis zum nächsten Schritt(*realtime mode*) sowie für den *timestamp* und die Ursache der Veränderung im aktuellen Zustand(*change cause*).

4. Literatur

1. Sebastian Jekutsch: Utilizing the Micro-processes of Software Development for Defect Prevention; http://projects.mi.fu-berlin.de/w/pub/Main/SebastianJekutsch/phdworkshop04.pdf 01.12.2006
2. Corman, Leiserson, Rivest: Introduction to Algorithms, McGraw-Hill 1992

Links

a) www.electrocodeogram.org
b) www.eclipse.org