Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Vereinfachte Wartung von Programmen durch gezielte Erhebung und Auswertung der Implementierungsgeschichte

Masterarbeit

im Studiengang Informatik

von

Philipp Kleybolte

Prüfer: Prof. Dr. Kurt Schneider
Zweitprüfer: Prof. Dr. Wolfgang Nejdl
Betreuer: M. Sc. Kai Stapel

Hannover, 15. Dezember 2010

Erklärung zu dieser Arbeit

Hiermit erkläre ich, dass ich die vorliegende Abschlussarbeit meiner Masterprüfung selbstständig und unter Verwendung der angegebenen Quellen und Hilfsmittel erstellt habe.

_____

Hannover, 15.12.2010 - Philipp Kleybolte

INHALTSVERZEICHNIS

1 Einführung

1.1 Motivation

Software-Systeme werden immer größer und komplexer. Immer weniger werden von Grund auf neu entwickelt, sondern bestehende Systeme werden verbessert und erweitert. Wartung ist ein bedeutender Bestandteil der Software-Entwicklung und wird immer wichtiger. Es gibt eine große Menge Legacy Systeme, die an aktuelle Gegebenheiten angepasst werden müssen. Immer wieder treten Bugs auf die nachträglich beseitigt werden müssen. Veränderungen an den Anforderungen führen zu kleinen und großen Änderungen an der Software.

Um ein Programm zu warten oder zu erweitern, muss man es verstanden haben. Man muss seine Struktur und sein Verhalten kennen. Man muss wissen, welchen Zweck es hat und auf welche Art es diesen erfüllt. Der wichtigste Punkt ist, dass man weiß an welchen Stellen im Code man ansetzen muss, um ein bestimmtes Verhalten der Software zu ändern, zu reparieren oder neue Funktionen hinzuzufügen. Das Lesen und Verstehen von Code kostet viel Zeit. Insbesondere wenn die Dokumentation der zu wartenden Software unvollständig oder schlecht ist, oder gar komplett fehlt.

Nach [1] macht die Wartung von Software zwischen 50 % und 80 % des gesamten Entwicklungsaufwands aus. Von der Wartung wiederum werden zwischen 47 % und 62 % der Zeit für das Einarbeiten in den alten Code gebraucht. Nach [2] braucht das Einlesen in Code sogar bis 60 % des gesamten Aufwands der Software-Entwicklung. Das Ziel dieser Arbeit ist, die Einarbeitungszeit zu verkürzen. Aus den genannten Zahlen ist abzulesen, dass damit eine deutliche Steigerung der Produktivität einherginge.

1.2 Aufgabe

Der Fokus liegt in dieser Arbeit auf der Unterstützung von Entwicklern, die sich in ihren eigenen Code einlesen. Das heißt, es geht um das Einarbeiten in Code, dessen Funktionsweise dem Entwickler früher einmal gut bekannt war. Nachdem der Entwickler längere Zeit nicht an dem Code gearbeitet hat, hat er jedoch viel vergessen und muss sich die Funktionsweise wieder in Erinnerung rufen.

Aus der kognitiven Psychologie ist bekannt, dass das Erinnerungsvermögen einer Person besser ist, wenn der Kontext der Person zum Zeitpunkt des Lernens mit dem Kontext zum Zeitpunkt der Erinnerung übereinstimmt. Dieser Kontext besteht aus der räumlichen Umgebung, anderen anwesenden Personen, Musik, dem emotionalen Zustand der Person und weiteren Faktoren. Faktoren, die intensiver und aufmerksamer aufgenommen werden, sind dabei relevanter als Faktoren, die nur beiläufig wahrgenommen werden. Beim Programmieren liegt der Aufmerksamkeitsfokus auf der Entwicklungsumgebung und dem enthaltenen Code. Das bedeutet, zu neu geschriebenem Code ist der umgebende Code der Kontext.

In dieser Arbeit soll untersucht werden, in wie weit es hilft die Implementierungsgeschichte gezielt aufzuzeichnen. Dann kann zu einem Code-Abschnitt der Kontext wiederhergestellt werden. Dadurch soll der Entwickler gedanklich zurückversetzt werden in den Zeitpunkt, zu dem er den Abschnitt geschrieben hat. Dadurch soll er sich leichter erinnern. Auf diese Weise soll die Einarbeitungszeit in alten Programmcode verkürzt werden. Der Entwickler kommt schneller in eine produktive Phase. Dadurch geht die Wartung von Software schneller und wird günstiger.

Es soll ein Plugin für eine Entwicklungsumgebung geschrieben werden, das Kontexte beim Programmieren speichert. Später soll es möglich sein, diese aufgezeichneten Kontexte wiederzugeben. Dafür ist zunächst zu prüfen, was relevante Aspekte des Kontextes sind. Eclipse ist Open Source und bietet Schnittstellen für Erweiterungen. Daher soll das in dieser Arbeit zu entwickelnde Plugin für Eclipse hergestellt werden. Das Eclipse-Plugin Mylyn speichert ebenfalls Kontexte. Es solle überprüft werden, ob Funktionen von Mylyn für diese Arbeit wiederverwendet werden können.

1.3 Gliederung

Das Ziel ist ein Werkzeug zu entwickeln, das bei der Einarbeitung in Code hilft. Im Kapitel 2 werden die Grundlagen gelegt, um zu verstehen, wie die Implementierungsgeschichte genutzt werden kann. Zunächst werden einige Erkenntnisse aus der kognitiven Psychologie erläutert. Es wird gezeigt, mit welchen Verfahren man Menschen beim Erinnern helfen kann und versucht diese Verfahren auf Software-Entwicklung zu übertragen.

Anschließend wird eine Umfrage vorgestellt. Die Umfrage wurde durchgeführt, um weitere Anforderungen an das zu entwickelnde Plugin zu finden. Wenn wir ein Plugin erstellen, das hilft sich zu erinnern, soll der Fokus dieser Hilfe auf Aspekten liegen, die häufig vergessen werden. Wir haben daher eine Umfrage unter Software-Entwicklern durchgeführt, um solche häufig vergessenen Aspekte zu finden.

Im Kapitel 3 wird auf Basis der in Kapitel 2 beschrieben Grundlagen ein theoretisches Konzept entwickelt wie ein Werkzeug beschaffen sein müsste, um Software-Entwickler beim Einarbeiten in ihren alten Code zu unterstützen. Im Rahmen dieser Arbeit wurde eine Implementierung des Konzepts erstellt. In Kapitel 4 wird erläutert wie diese Implementierung aufgebaut ist und in welchen Aspekten sie sich von dem theoretischen Konzept unterscheidet.

In Kapitel 5 werden die Ergebnisse dieser Arbeit diskutiert. Es werden einige ähnliche Arbeiten vorgestellt. Außerdem wird ein Ausblick gegeben, in welchen Richtungen man das Konzept noch erweitern könnte und wie die Implementierung noch verbessert werden kann.

2 Vorbetrachtungen

2.1 Grundlagen der kognitiven Psychologie

Die kognitive Psychologie versucht zu erklären, wie menschliche Intelligenz funktioniert. Sie beschäftigt sich damit, wie Menschen wahrnehmen, lernen, sich erinnern, Probleme lösen und wie Menschen Sprache verarbeiten. In dieser Arbeit sollen insbesondere die Erkenntnisse bezüglich des Lernens und Erinnerns genutzt werden, um Software-Entwicklern zu helfen, sich an eigenen Code zu erinnern.

2.1.1 Vergessen und Erinnern

Der Grund, der diese Arbeit nützlich macht, ist das „Vergessen“. In der Software-Entwicklung besteht ein großer Teil des Aufwands in der Einarbeitung in alten Code. Häufig müssen sich Software-Entwickler in ihren eigenen Code einarbeiten, weil sie vergessen haben, wie er funktioniert. Wenn man Entwicklern helfen will, sich wieder an ihren Code zu erinnern, muss man zunächst verstehen auf welche Weise das Erinnern und das Vergessen im menschlichen Gehirn geschieht.

Es gibt grundsätzlich zwei Arten, wie das Hirn Informationen vergisst.

1. Die Informationen sind eigentlich noch im Gedächtnis vorhanden, es gelingt aber nicht sie abzurufen.
2. Durch Verfall der Informationen selbst [3]

Der Vorgang des Vergessens lässt sich mit einer Festplatte vergleichen. Im ersten Fall ist die Dateizuordnungstabelle defekt. Die gesuchte Datei ist noch vorhanden, kann jedoch nicht gefunden werden. Im zweiten Fall ist die Datei selbst überschrieben. Für diese Arbeit ist der erste Fall der relevante. Mit Hilfe eines Werkzeugs wollen wir Software-Entwicklern helfen, sich die Informationen wieder zu erinnern, die noch im Gedächtnis des Entwicklers aber nicht von ihm abrufbar sind.

Verschiedene Experimente der Kognitiven Psychologie haben gezeigt, dass „vergessene“ Gedächtnisinhalte sehr häufig noch existieren, es jedoch nicht gelingt auf diese zuzugreifen. Der primäre Grund für Vergessen ist also der erste beschriebene Fall. Dies gilt zumindest,

wenn die Informationen ins Langzeitgedächtnis gelangt sind. Ob Informationen im Langzeit- oder im Kurzzeitgedächtnis eines Menschen sind, hängt hauptsächlich davon ab, wie lange er sich mit diesen Informationen beschäftigt hat. Entwickler beschäftigen sich für gewöhnlich längere Zeit mit dem Quellcode, den sie schreiben. Man kann also davon ausgehen das diesbezügliche Informationen das Langzeitgedächtnis erreichen.

Nachgewiesen wurde die Tatsache, dass vergessene Informationen oft noch vorhanden sind, unter anderem in einem Experiment, das von Nelson 1971 [4] durchgeführt wurde. Die Probanden mussten Zahl-Nomen-Paare lernen. Zwei Wochen später wurde getestet, an welche Nomen sie sich erinnern konnten, wenn die Zahl vorgegeben wurde. Die Paare sollten anschließend noch einmal gelernt werden. Paare, die nicht wiedergegeben werden konnten, wurden nun zur Hälfte verändert und zur Hälfte beibehalten. Wenn z.B. „43-Hund“ nicht erinnert wurde, dann wurde es in der Hälfte der Fälle zu „43-Haus“ verändert und in der Hälfte der Fälle als „43-Hund“ beibehalten. Wenn von den Informationen nichts mehr im Gedächtnis vorhanden wäre, müssten sich die veränderten Paare ebenso gut lernen lassen wie die unveränderten. In anschließenden Tests konnten die Probanden aber 78 % der unveränderten Paare wiedergeben und nur 43 % der veränderten. Das zeigt, dass von den Informationen noch Teile im Gedächtnis waren, selbst wenn sie nicht mehr wiedergegeben werden konnten.

Die Frage ist nun, wie man dem Hirn helfen kann, wieder einen Zugang zu Informationen zu finden, wenn die Informationen, wie oben beschrieben, noch vorhanden, aber nicht abrufbar sind. Die kognitive Psychologie gibt darauf eine Antwort. „Die Theorie der Kodierungsspezifität besagt, dass ein Hinweisreiz einen Informationsabruf begünstigt, wenn dieser Informationen enthält, die während des Lernens der Zielinformation verarbeitet wurden.“ Zitat [3] S.187

Das heißt Hinweisinformationen und -Reize, die man während des Lernens der Zielinformation aufgenommen hat, werden unterbewusst mit der Zielinformation verknüpft. Wenn die Hinweisinformationen oder -Reize reproduziert werden, hilft das sich an die Zielinformation zu erinnern. Das gilt auch, wenn die Zielinformation inhaltlich nicht mit den Hinweisinformationen oder -Reizen zusammenhängen.

Gezeigt wurde der hilfreiche Effekt von solchen Hinweisreizen in einem Versuch von Thomson und Tulving von 1970 [3]. Es mussten Testpersonen Wörter lernen. Es wurden Wortpaare vorgegeben, von denen jedoch immer nur ein Wort gelernt werden sollte. Zum Beispiel sollte das Wortpaar „Boden – Kalt“ gelernt werden. Davon war „Kalt“ das Zielwort, „Boden“ ein Hinweiswort. Durch zwei Ansätze wurde versucht, die Erinnerungsleistung zu steigern.

1. Durch vorgeben des Hinweiswortes
2. Durch vorgeben von Worten, die mit den Zielworten inhaltlich stark assoziiert waren. Z.B. „Heiß“ also Assoziationswort für „Kalt“

Wenn Hinweiswörter vorgeben wurden, konnten doppelt so viele Zielworte wiedergegeben werden, wie wenn starke Assoziationswörter vorgegeben wurden, die nicht während des Lernens verarbeitet wurden. Das zeigt, dass Hinweisreize einen positiven Effekt auf die Erinnerungsleistung haben und dass dieser Effekt sogar stärker ist, als die Hilfe mit Assoziationswörtern.

Wir versuchen diese Ergebnisse auf die Software-Entwicklung zu übertragen. Während der Entwickler programmiert, lernt er die Struktur und die Funktionsweise des Codes. Das ist die Zielinformation, an die er sich erinnern soll, wenn er sich einarbeitet. Alle Informationen die der Entwickler verarbeitet, während er programmiert, helfen ihm daher, sich später zu erinnern.

Zu klären ist noch, was zu den „verarbeiteten Informationen“ während des Programmierens zählt. In dem Experiment von Thomson und Tulving wurden die Hinweisreize „aufgebaut“, indem sie sie den Probanden zu Verarbeitung während des Lernens vorlegten. Solche klaren Hinweisreize gibt es beim Programmieren leider nicht. Solche Hinweisreize auch beim Programmieren aufzubauen erscheint jedoch nicht sinnvoll. Die Aufmerksamkeit des Entwicklers müsste zumindest teilweise auf den Hinweis gelenkt werden. Wir lenken den Entwickler daher ab, was sich negativ auf seine Produktivität beim Programmieren auswirken wird. Zum Zeitpunkt des Aufbaus des Hinweisreizes ist dabei noch nicht einmal klar, ob er jemals gebraucht wird. Wenn wir einen Hinweisreiz aufbauen, so senken wir also die Produktivität während des Programmierens. Im Fall, dass der Entwickler sich später nicht einarbeiten muss, haben wir diesen Aufwand verschwendet. Im Fall, dass der Entwickler sich

später einarbeiten muss, ist dennoch zweifelhaft, ob der Gewinn durch die verbesserte Erinnerungsfähigkeit den Verlust durch die Ablenkung aufwiegen kann. Es gibt aber noch andere Möglichkeiten den Entwickler zu unterstützen, wenn die Hinweisreize nicht extra aufgebaut werden.

In unterschiedlichen Studien hat sich gezeigt, dass das Erinnerungsvermögen auch steigt, wenn Hinweisreize geboten werden, die nur unbewusst verarbeitet wurden. Es gibt eine Reihe von Studien, in denen Testpersonen in einer Umgebung A gelernt haben und dann die Wiedergabeleistung in Umgebung A verglichen wurde mit der Wiedergabeleistung in einer anderen Umgebung B.

Ein solches Experiment führte S. M. Smith 1979 durch [3]. Alle Testpersonen lernten in einem Raum A eine Wortliste. Anschließend gingen sie in einen Raum B. Diesen sollten die Testpersonen zeichnen. Dadurch sollten sie sich mit dem Raum vertraut machen. Danach wurden sie in einen Warteraum C geschickt. Zum Schluss gingen sie in einen Testraum und sollten möglichst viele Wörter wiedergeben. Die Testräume waren Raum A und B. Die Wiedergabeleistung in Raum A war um 25% höher als in Raum B. Beide Gruppen mussten die gleiche Zeit warten und gleich oft den Raum wechseln. Außerdem waren beide Gruppen mit ihrem jeweiligen Testraum vertraut. Dennoch war die Wiedergabeleistung in dem Raum, in dem sie gelernt hatten, signifikant höher. Das zeigt, dass sogar ein so beiläufiger Aspekt wie der Raum, in dem man lernt einen Hinweisreiz darstellt.

Je deutlicher die Unterschiede zwischen Lern- und Wiedergabeumgebung und je intensiver sie wahrgenommen wird, desto stärker wirkt die Umgebung als Hinweisreiz. Ein extremes Experiment zu Lernen und Wiedergabe an unterschiedlichen Orten führten Godden und Baddeley durch. Sie ließen Taucher 40m unter Wasser oder an Land lernen. Der Effekt trat noch deutlich stärker hervor, als nur bei unterschiedlichen Räumen. [4]

Ähnliche Effekte stellen sich auch bei Übereinstimmen der Emotionen ein. Ein trauriger Mensch erinnert sich gut an Dinge, die er in traurigem Zustand gelernt hat. Hingegen kann ein Mensch in heiterem Zustand sich besser an Dinge erinnern, die er in heiterem Zustand gelernt hat. Da dies mit einer Entwicklungsumgebung im Computer aber weder messbar noch reproduzierbar ist, ist dieser Aspekt für das Konzept eines Werkzeugs in dieser Arbeit

vernachlässigbar. Er sollte aber bei empirischen Tests zu Effektivität des in dieser Arbeit entstandenen Programms bedacht werden.

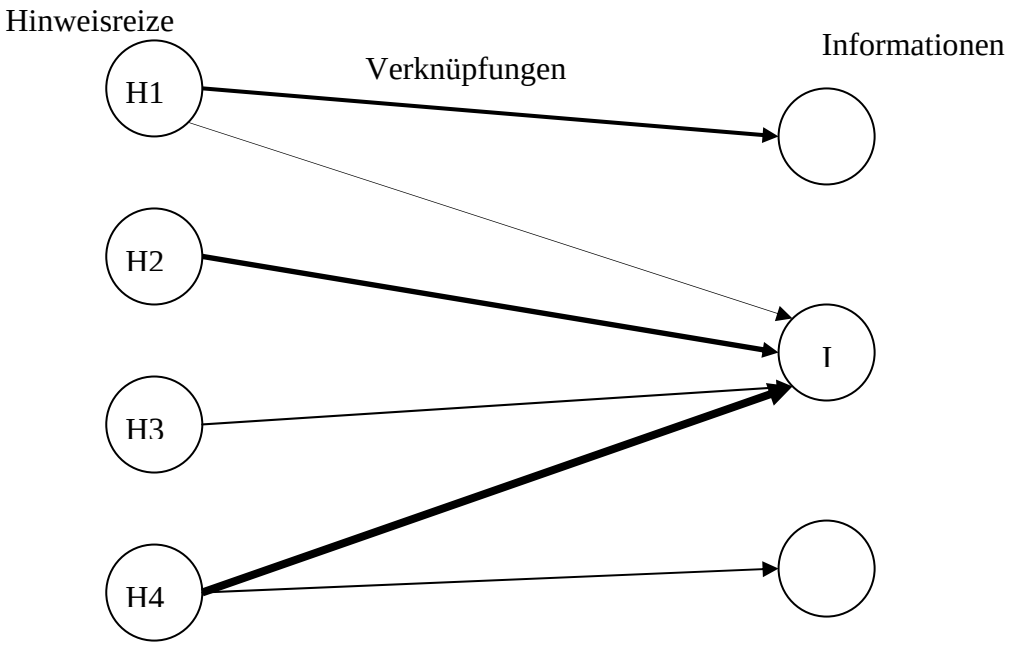
Für das Erinnerungsvermögen ist der gesamte Kontext einer Person zum Zeitpunkt des Lernens relevant. Der Kontext besteht hierbei aus dem Raum, in dem sie sich befindet, anderen anwesenden Personen, Hintergrundgeräuschen oder Musik, seinen Emotionen und der ausgeübten Beschäftigung. [3] Es hat sich herausgestellt, dass jene Aspekte des Kontextes, die besonders intensiv wahrgenommen werden, auch mehr Einfluss auf die Erinnerungsleistung haben als nur beiläufig aufgenommene Aspekte.

Die „verarbeiteten Informationen“ des Entwicklers bestehen also aus dem gesamten Kontext, dem er während des Programmierens ausgesetzt ist. Alle Bereiche des Kontextes können ihm helfen, sich zu erinnern, wenn sie reproduziert werden. Es gibt jedoch Unterschiede, in welchem Maß die verschiedenen Teile des Kontextes die Erinnerungsfähigkeit steigern können. Ein Entwickler nimmt während des Programmierens von seinem Kontext besonders stark die verwendete Entwicklungsumgebung und den gerade bearbeiteten Code wahr. Es ist daher davon auszugehen, dass das Erinnerungsvermögen eines Programmierers bezüglich eines entwickelten Programms gesteigert werden kann, wenn die Entwicklungsumgebung und der Code wiederhergestellt werden auf den Stand während der Programmierung. Ein Werkzeug könnte den Entwickler also unterstützen, wenn es alte Versionen des Codes und die dazugehörende Oberfläche der Entwicklungsumgebung reproduzieren kann.

2.1.2 Assoziationen

Ein Modell wie das menschliche Gedächtnis arbeitet, ist das Modell der Assoziationen. Das Hirn nimmt Reize wahr und assoziiert aufgrund dieser Reize andere Reize oder Informationen. Zum Beispiel kann das Hören eines Musikstücks dazu führen, dass man sich an eine bestimmt Situation erinnert, in der man das Stück zuvor gehört hat. Die Aufnahme des musikalischen Reizes hat die Erinnerung an eine andere Information verursacht.

Wie stark eine Assoziation zwischen zwei Informationen wirkt, hängt von verschiedenen Faktoren ab. Je öfter zwei Informationen zusammen auftreten, desto stärker werden sie miteinander assoziiert. Je intensiver und je länger man die Informationen aufnimmt, desto stärker ist die Bindung. Wenn die Verknüpfung länger zurück liegt, nimmt sie ab. [5]



1 Assoziationsmodell

Wenn man versucht sich an etwas zu erinnern, hilft es an möglichst viele Informationen gleichzeitig zu denken, die als Assoziation dienen könnten. Es ist zum Beispiel folgende Situation gegeben: Man sieht das Foto einer Person, kann sich aber an den Namen nicht mehr erinnern. Das Gesicht reicht in diesem Fall also als Assoziation nicht aus. Bekommt man jedoch zusätzlich die Informationen, wo man die Person getroffen hat, oder den Anfangsbuchstaben des Namens kann man sich eventuell doch erinnern. Je mehr Assoziationen vorliegen, die zu einer Information führen, desto besser kann man sich daher erinnern. In Abbildung 3 ist das abstrakt dargestellt. Je mehr von den Hinweisreizen H1 bis H4 vorliegen, desto höher ist die Wahrscheinlichkeit sich an Information I zu erinnern.

Anders herum wirken Assoziationen besser, wenn sie auf nur eine Information wirken. Der Blick auf ein Gesicht kann reichen, um sich an einen Namen erinnern, weil das Gesicht nur diesem Namen zugeordnet ist. Der Anfangsbuchstabe alleine führt zu keiner Assoziation mit einem Namen, weil er nicht eindeutig ist.

Übertragen auf den Programmierer bedeutet dies, dass die Entwicklungsumgebung alleine nicht viel hilft Informationen zum Code zu assoziieren, weil die Entwicklungsumgebung bei fast jedem entwickelten Programm gleich oder zumindest sehr ähnlich aussieht. Der Code

hingegen ist für jedes Programm einzigartig und eignet sich daher gut, sich Informationen ins Gedächtnis zu rufen. Wäre der Code nicht einzigartig sondern sehr ähnlich zu anderem, wäre seine Funktion und damit die Informationen, die man sich ins Gedächtnis rufen möchte, gleichermaßen ähnlich. Daher entstehen durch ähnlichen Code keine Nachteile für die Erinnerung seiner Funktion. An ähnlichen Code kann man sich sogar besser erinnern, da der Code und seine Bedeutung wiederholt gelernt wurden.

Die Daten, die ein Programm in der Kommandozeile ausgibt, oder Fehlermeldungen könnten ebenfalls als Erinnerungsstütze dienen. Diese Daten sind häufig der direkte Auslöser für den Entwickler, welche Änderungen am Code als nächstes nötig oder sinnvoll sind. Sie könnten daher insbesondere bei der Frage helfen, *warum* der Entwickler eine bestimmte Änderung vorgenommen hat.

2.1.3 Aufnahmefähigkeit von Entwicklern

In der kognitiven Psychologie hat man sich damit beschäftigt, wie viele Informationen ein Mensch kurzzeitig aufnehmen kann. Neben dem Langzeitgedächtnis, das Informationen über einen längeren Zeitraum speichert, besitzen Menschen ein Kurzzeitgedächtnis, auch Arbeitsgedächtnis genannt. In dieses Arbeitsgedächtnis passt nur eine sehr begrenzte Menge an Informationen. So können sich Menschen durchschnittlich sieben Zahlen merken oder 4,5 einsilbige Wörter. [4]

Ein Mensch kann also nur eine begrenzte Anzahl an Informationen kurzzeitig aufnehmen. Es ist aber unterschiedlich was als „eine Information“ gilt. Die meisten Menschen können sich problemlos vier einsilbige Wörter merken, wie z.B. „Laub, Spuk, Beil, Duft“. Diese vier Wörter bestehen aus 16 Buchstaben. Wenn man die Buchstaben mischt, ist es sehr unwahrscheinlich, sich diese merken zu können. „IUKLSEFUABPDULBT“ wird sich kaum ein Mensch kurzzeitig merken können.

Wenn das Hirn Informationen speichert, fasst es Informationen zu so genannten „Chunks“ zusammen. Was das Hirn als Chunks auffasst, ist davon abhängig, ob das Hirn oft mit diesen Informationsstrukturen umgehen muss. Menschen sind es gewöhnt mit Worten umzugehen. Deshalb können sie mehrere Buchstaben zu einem Chunk zusammenfassen, wenn sie ein Wort ergeben. Es gibt aber auch Informationsstrukturen, die nur wenige Menschen zu Chunks

zusammenfassen können. Gute Schachspieler begreifen die Stellungen mehrerer Figuren als Chunk, weil sie viele typische Stellungen kennen, normale Menschen müssen sich jede Figur einzeln merken. Die Schachspieler haben sich viel mit Schachstellungen beschäftigt und damit ihr Hirn automatisch darauf trainiert, diese Form von Informationen effizient zu erkennen und zu lernen. Die Fähigkeit eine bestimmte Sorte von Informationen besonders gut zu verarbeiten, wird Expertentum auf dem jeweiligen Gebiet genannt. [4]

1981 untersuchten McKeithen und Reitman wie sich Anfänger, Fortgeschrittene und Experten der Programmierung unterscheiden hinsichtlich ihrer Aufnahmefähigkeit von Quellcode [6]. Die betrachtete Programmiersprache war ALGOL W. Die Probanden sollten sich möglichst viele Zeilen eines 31 Zeilen langen Programms merken. Die Probanden wurden folgendermaßen eingeteilt:

- Anfänger fingen gerade einen Programmierkurs in ALGOL W an.
- Fortgeschrittene hatten einen solchen Kurs abgeschlossen.
- Experten lehrten ALGOL W und hatten über 400 Stunden Programmiererfahrung damit.

In dieser Arbeit besteht die Zielgruppe aus Berufs-Programmieren. Die Erfahrung solcher Entwickler geht über einen Einführungskurs deutlich hinaus. Daher sind für uns insbesondere die Ergebnisse interessant, die die Experten bei dem Versuch erzielen konnten.

Die Untersuchung ergab, dass die Experten nach einer kurzen Lernphase durchschnittlich fast sieben Zeilen Code auswendig wiedergeben konnten, doppelt so viele wie Anfänger. In einem weiteren Test wurden die Codezeilen zufällig gemischt. Das heißt, die Zeilen waren in sich noch vollständig, aber in keiner sinnvollen Reihenfolge. Bei den Experten nahm darauf hin die Merkfähigkeit um beinahe die Hälfte ab. Für Anfänger hingegen gab es nur einen sehr geringen Unterschied. Anscheinend fassen die Experten zusammenhängende Strukturen des Codes, wie z.B. eine for-Schleife, zu Chunks zusammen. Sie können sich daher Code gut merken, wenn er zusammenhängt. Wenn er aus dem Kontext gerissen ist, ist es deutlich schwieriger sich den Code zu merken. Es ist daher zu vermuten, dass die Fähigkeit sich Code zu merken ebenfalls abnimmt, wenn der zu merkende Code auf unterschiedliche Programmteile verteilt ist. Auf die Java-Entwicklung übertragen heißt das, dass sich Programmier wahrscheinlich gut sieben Zeilen innerhalb einer Klasse merken können, die

Erinnerungsfähigkeit aber schwächer ist, wenn sich der Programmierer Code aus unterschiedlichen Klassen merken soll.

Für ein Werkzeug das beim Erinnern unterstützen soll, ist daher die folgende Erkenntnis zu berücksichtigen: Entwickler können ca. sieben Zeilen schnell aufnehmen und gedanklich verarbeiten, wenn sie aus dem gleichen Programmteil stammen, sonst etwa die Hälfte.

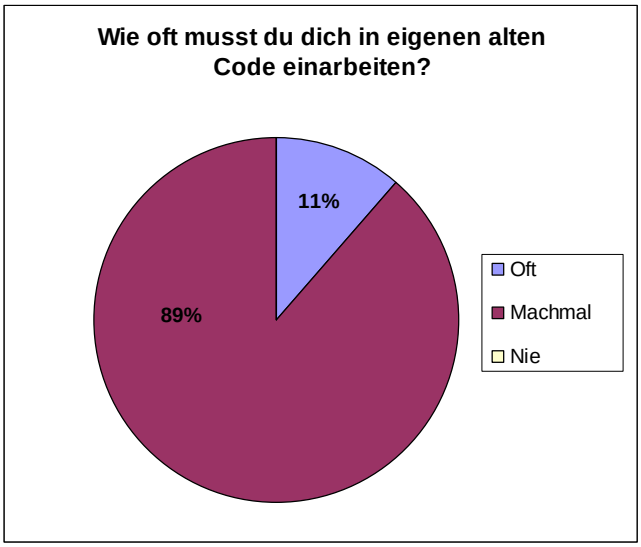
2.2 Empirische Problemuntersuchung

Im Rahmen dieser Arbeit wurde ein Fragebogen entwickelt. Mit diesem Fragebogen sollte herausgefunden werden, ob es sinnvoll ist Entwicklern bei der Einarbeitung in eigenen Code zu helfen. Außerdem sollte herausgefunden werden, welche Aspekte des Programmcodes Entwickler am häufigsten vergessen. Daraus sollte abgeleitet werden, wobei man den Programmierer am meisten unterstützen sollte.

Der Fragebogen wurde als Online-Fragebogen erstellt. [7] Er bestand aus zwei offenen Fragen und acht Multiple Choice Fragen. Außerdem wurde der Hintergrund der Antwortenden kurz abgefragt. Damit sollte überprüft werden, ob der Fragebogen nur Anfänger erreicht hat, oder ob die Befragten viel Programmiererfahrung haben. Der komplette Fragebogen ist im Anhang zu finden.

In der Grafik 2 sieht man, dass sich etwa ein Zehntel der Entwickler oft in ihren eigenen Code einarbeiten müssen. Alle anderen müssen dies manchmal tun. Niemand antwortete, er müsse sich nie in eigenen Code einarbeiten.

Dieses Ergebnis zeigt, dass es sinnvoll ist, ein Werkzeug zu entwickeln, dass Programmierern beim Einarbeiten hilft.



2 Einarbeitungshäufigkeit

2.2.1 Hintergrund der Befragten

Wie in Tabelle 1 zu sehen hat die Gruppe der Befragten sehr unterschiedliche Programmiererfahrung. Es sind Anfänger dabei, aber auch Leute mit mehreren Jahren Berufserfahrung. Es war nur eine Person, die sowohl privat, als auch in der Uni und beruflich seit weniger als einem Jahr programmiert.

Seit wann programmierst du schon?	<1Jahr	1-3 Jahre	>3 Jahre	unbeantwortet
Privat	2	3	29	1
Uni	3	8	23	1
Beruflich	9	8	9	9

Tabelle 1 Seit wann programmierst du schon?

Es sollte herausgefunden werden, ob die Befragten Erfahrungen mit größeren Projekten haben, oder ob sie bisher nur Projekte in der Größenordnung von Hausübungen durchgeführt haben. Es wurde daher nach der üblichen Projektdauer gefragt. Bei über 60 % der Befragten lag die Projektdauer bei mehr als einem Monat.

< 1 Woche	< 1 Woche	> 1 Monat	unbeantwortet
4	9	22	0

Tabelle 2 Wie lange laufen bei dir üblicherweise Projekte?

Insgesamt hatten die Befragten schon umfassende Programmiererfahrung, es gab aber auch einzelne die eher als Anfänger zu bezeichnen sind. Die Zusammensetzung der Befragten entspricht der Zielgruppe, der mit dieser Arbeit geholfen werden soll. Die weiteren Ergebnisse dieses Fragebogens können daher als Grundlage für die Entwicklung des Konzepts genutzt werden.

2.2.2 Offene Fragen

Schon bevor der Fragebogen entstand, gab es grobe Konzepte, wie man Entwickler unterstützen könnte. In Gesprächen mit Entwicklern über die Aufgabenstellung ergaben sich immer wieder neue Ideen. Mit den offenen Fragen sollten mehr solcher Ideen gesammelt werden und bisher nicht bedachte Szenarien gefunden werden. Es sollten Situationen aus der Software-Entwicklung und der Einarbeitung in Code gefunden werden, die in den Konzepten bisher nicht behandelt wurden. Gleichzeitig sollten möglichst wenig offene Fragen gestellt werden, um die Bearbeitungszeit des Fragebogens niedrig zu halten.

Es gab die folgenden zwei offenen Fragen:

1. „Wenn du dich in eigenen alten Quellcode einarbeiten musst, welche Aspekte hast du am ehesten vergessen?“

2. „Woran erinnerst du dich am besten, wenn du eigenen alten Code liest?“

Die Antworten auf die offenen Fragen ergaben keine der erhofften neuen Ideen. Sie legten aber den Schluss nahe, dass man die meisten Entwickler in zwei Klassen aufteilen kann. Die Antworten sind fast alle zwei Gruppen zuzuordnen. Die erste Gruppe kann sich gut an die Struktur eines Projekts erinnern, die zweite Gruppe erinnert sich eher, wie einzelne Funktionen implementiert sind, hat aber keine Übersicht mehr wozu diese notwendig waren.

Von 20 Leuten die auf die erste Frage geantwortet haben, schrieben 8, dass sie die Algorithmen und Details aus der Implementierung leicht vergessen. 7 Leute antworteten, dass sie Strukturen und Gesamtzusammenhänge am ehesten vergessen. Eine Person antwortete, bei kleinen Projekten vergäße sie eher Details der Implementierung, bei großen Projekten ist auch der Zusammenhang nicht mehr so gut im Gedächtnis. Siehe Tabelle 3.

Die zweite Frage beantworteten 18 Leute. 8 davon konnten sich am besten an Algorithmen und Optimierungstricks erinnern. Besonders bleiben Code-Abschnitte in Erinnerung, die besondere Probleme aufwarfen. 6 Leute antworteten, dass sie sich gut an die generelle Struktur und den groben Ablauf erinnern. Siehe Tabelle 4.

Wenn du dich in eigenen alten Quellcode einarbeiten musst, welche Aspekte hast du am ehesten vergessen?	
Kategorie „Strukturen“	Kategorie „Details“
7 Antworten	8 Antworten
Bsp-Antwort: Welche Variablennamen stehen wofür? Wie heißen meine Funktionen? Wofür brauche ich welche Funktionsaufrufe?	Bsp-Antwort: „Wie ich Algorithmen aufgebaut habe und warum ich es so und nicht anders gemacht habe“

Tabelle 3

Woran erinnerst du dich am besten, wenn du eigenen alten Code liest?	
Kategorie „Strukturen“	Kategorie „Details“
6 Antworten	8 Antworten
Bsp-Antwort: „An die generelle Struktur (Packages, Klassen, Konzept...)“	Bsp-Antwort: „Warum ich den Code geschrieben habe. Eingesetzte Tricks in den Algorithmen/dem Stil.“

Tabelle 4

Es gab keine Personen, die antwortete, Details leicht zu vergessen und ebenfalls antwortet, sich an Details gut zu erinnern. Auch für Strukturen gab es keine solche Konstellation. Die Personen lassen sich daher alle eindeutig einer der Gruppen zuordnen.

In der Umfrage gibt exakt genau so viele Entwickler, die sich an Details gut erinnern, wie es Entwickler gibt, die Details leicht vergessen. Auch die Gruppen von Entwicklern, die Strukturen leicht vergessen und jene die sich gut erinnern, sind fast gleich groß. Aus den offenen Fragen lässt sich daher nicht ableiten, in welchen Bereichen Entwickler am meisten Unterstützung brauchen.

2.2.3 Multiple-Choice Fragen

In den Multiple-Choice Fragen sollten die Befragten zu unterschiedlichen Aspekten des Programmierens angeben, wie gut sie sich jeweils erinnern können. Mit diesen Fragen sollte herausgefunden werden, ob man Software-Entwickler in manchen Bereichen besonders stark unterstützen sollte und ob es Bereiche gibt, für die das unnötig wäre. Es wurde gefragt, wie gut sich Entwickler an Methoden, Klassen und Packages erinnern, jeweils in Bezug auf ihre Struktur und auf ihre Funktionsweise. Außerdem wurde gefragt, wie gut sich die Entwickler an Fehler und an deren Lösungen erinnern.

Aus den offenen Fragen ließ sich nicht erkennen, ob man eher das Erinnern an grobe Strukturen oder an Feinheiten unterstützen sollte. In den Multiple-Choice Fragen ist zu erkennen, dass Methoden schlechter erinnert werden als Klassen. Es wurde gefragt, wie gut man sich an die Funktion von Methoden/Klassen/Packages isoliert vom restlichen Code erinnert und wie gut man sich an den Zusammenhang von Methoden/Klassen/Packages

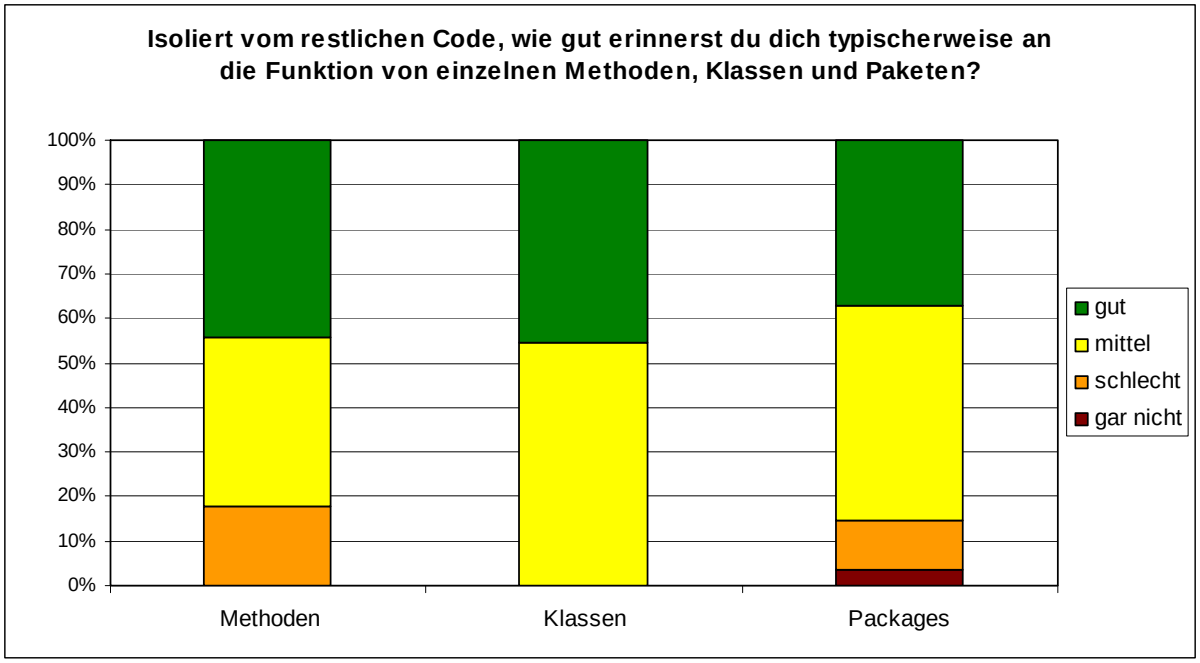
erinnert. Es gab jeweils die Abstufungen „gut“, „mittel“, „schlecht“ und „gar nicht“. Siehe Abb. 3 und 4.

Mit dem Programm „R“ wurde der Wilcoxon Rangsummen-Test (auch Mann-Whitney-Test) auf den Messreihen durchgeführt, um zu überprüfen, ob einzelne Aspekte statistisch signifikant besser erinnert werden als andere. Signifikant bedeutet in diesem Zusammenhang, dass es sehr unwahrscheinlich ist, dass die Unterschiede zwischen den Verteilungen zufällig entstanden sind. Mit der Umfrage ziehen wir Stichproben aus der Gesamtmenge der Software-Entwickler. Mit dem Wilcoxon-Test kann gezeigt werden, dass ein Unterschied in zwischen zwei Stichproben-Verteilungen mit hoher Wahrscheinlichkeit auf Unterschieden in der Gesamtmenge beruht. [8] [9] Wenn man davon ausgeht, dass die Wahrscheinlichkeit eine falsche Aussage zu treffen unter 5% liegen muss, ergaben die Tests folgende Ergebnisse.

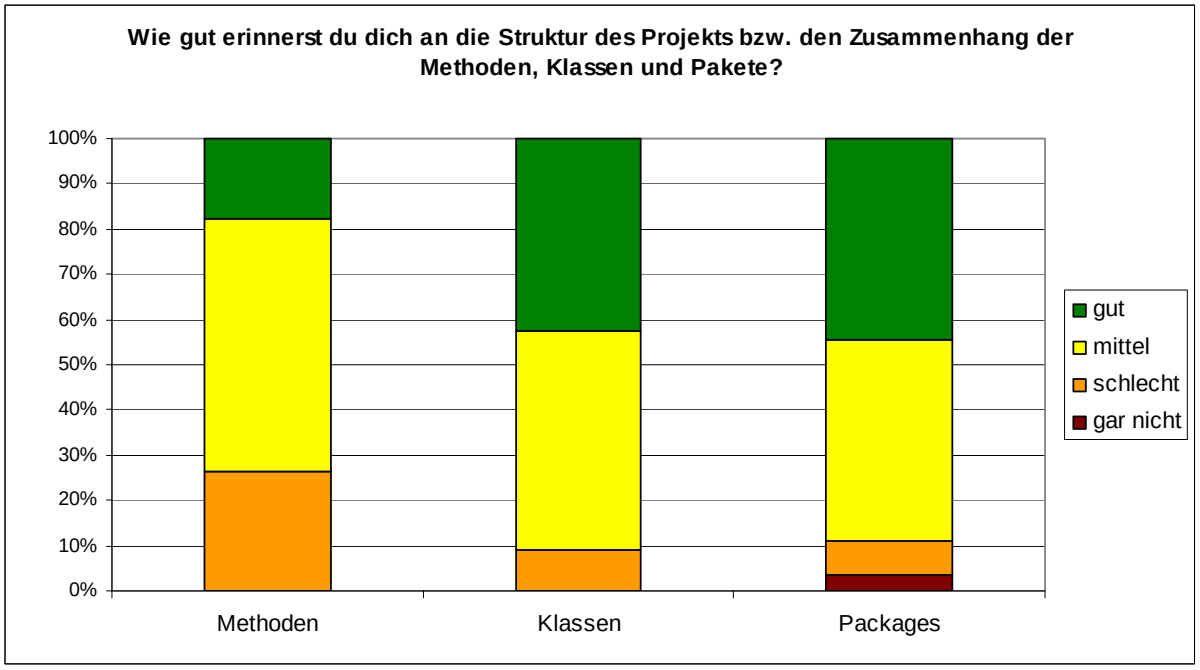
Entwickler erinnern sich an die Struktur von Klassen signifikant besser als an die von Methoden.

Entwickler erinnern sich an die Struktur von Packages signifikant besser als an die von Methoden.

Die übrigen Tests schlugen leider fehl. In den Graphen (Abb. 3 und 4) ist zu sehen, dass die Unterschiede in der Erinnerungsleistung nicht groß sind. Für erfolgreiche Signifikanztests reichen sie nicht aus. Die nicht durch Signifikanztests bestätigte Tendenz ist aber, dass Klassen sich sowohl bezüglich Struktur als auch bezüglich Funktion am besten erinnern lassen. Die Funktion von Packages lässt sich etwas schlechter erinnern als die von Methoden. Die Struktur von Packages lässt sich dafür signifikant besser erinnern als die von Methoden. Insgesamt spricht das dafür, Entwickler insbesondere beim Einarbeiten in Methoden, das heißt in Details der Implementierung, zu unterstützen.



3 Erinnerungsfähigkeit bzgl. Funktion von Code-Teilen



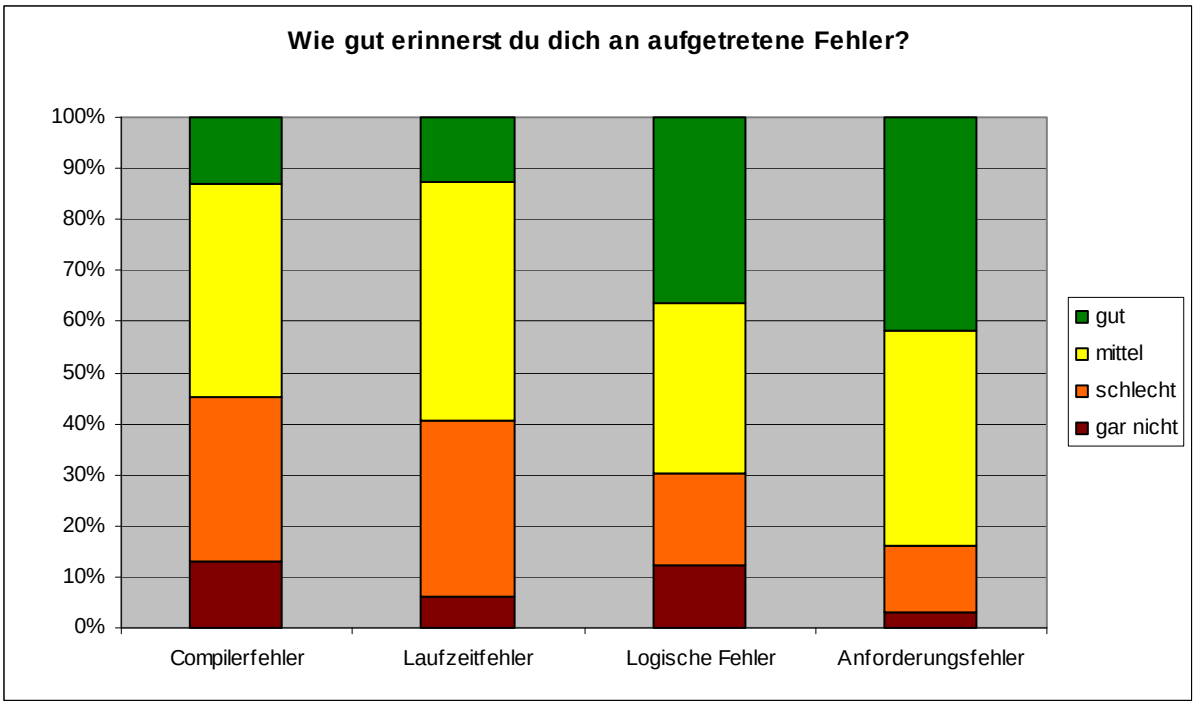
4 Erinnerungsfähigkeit bzgl. Struktur von Code-Teilen

Es wurde auch gefragt, wie gut die Entwickler sich an Fehler bzw. deren Lösungen erinnern können. Die Fehler und die entsprechenden Lösungen haben fast exakt die gleichen Verteilungen. Es ist zu erkennen, dass Entwickler sich an Fehler besser erinnern können, wenn sie abstrakter sind. An Compilerfehler kann man sich schlecht erinnern, an Anforderungsfehler hingegen gut.

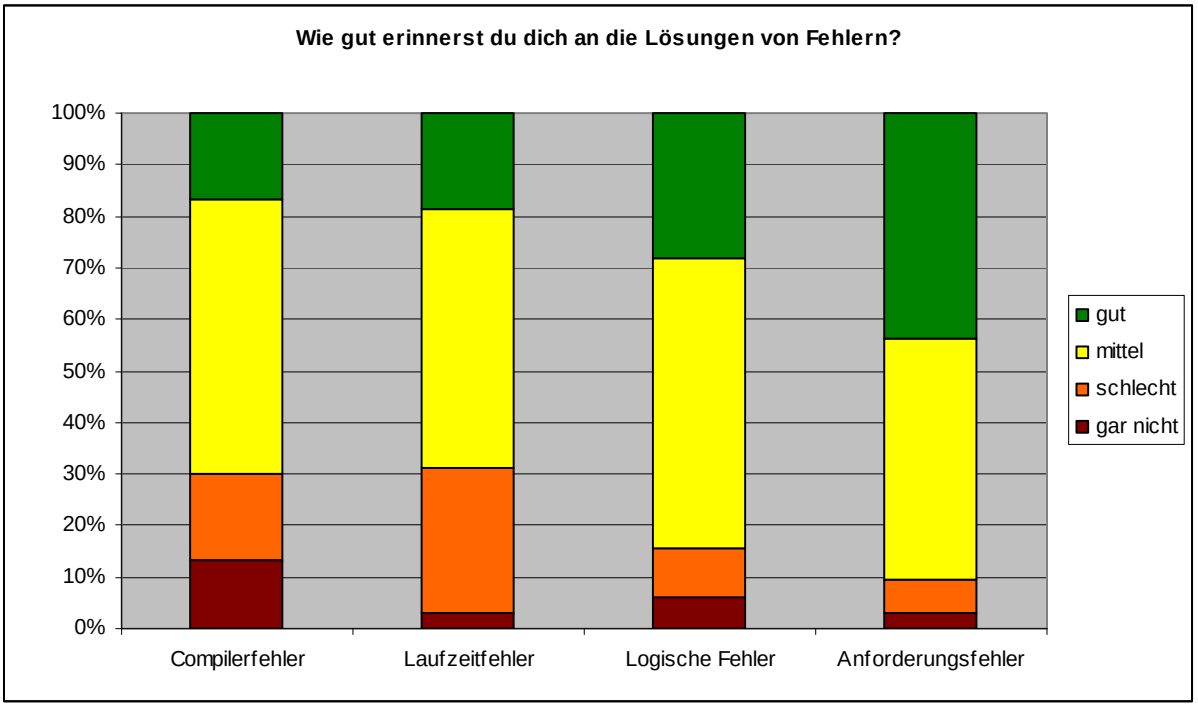
Signifikanztests mit der 5%-Grenze ergaben: Entwickler können sich an Anforderungs- und an logische Fehler signifikant besser erinnern als an Compilerfehler. Entwickler können sich signifikant besser an Anforderungsfehler erinnern als an Laufzeitfehler. Signifikanz lässt sich also nicht zwischen „benachbarten“ Abstraktionsstufen feststellen, aber immer wenn eine Stufe übersprungen wird.

Für die Lösungen von Fehlern war nur ein Signifikanztest positiv. Entwickler können sich an die Lösungen von Anforderungsfehlern signifikant besser erinnern als an die von Compiler- und Laufzeitfehlern.

Der Zusammenhang von Abstraktionsgrad zur Erinnerung des Fehlers hängt wahrscheinlich auch mit der Zeit zusammen, für die man sich mit dem Fehler beschäftigen muss. Compilerfehler sind meist klein und schnell behoben. Anforderungsfehler können hohen Aufwand verursachen. Menschen erinnern sich besser an Aktivitäten, mit denen sie lange beschäftigt waren als an solche, die nur kurz dauerten. Insofern ist es logisch, dass Entwickler sich besser an die Fehler mit hohem Abstraktionsgrad erinnern. Es erklärt aber nicht ganz, warum auch die Lösung besser im Gedächtnis bleibt. Der Entwickler hat sich zwar länger mit ihr beschäftigt, doch eine Lösung die viel Aufwand erforderte, ist auch komplexer und daher nicht so gut im Gedächtnis zu behalten. Allerdings muss hier angeführt werden, dass die Umfrage auf Selbsteinschätzung beruht und nicht in Experimenten überprüft wurde, ob die Testpersonen sich wirklich so gut erinnern, wie sie meinen.



5 Erinnerung aufgetretene Fehler



6 Erinnerung an Lösungen von Fehlern

3 Konzept zur Nutzung der Implementierungsgeschichte

Im letzten Kapitel wurden die grundlegenden Gedanken erläutert, wie man mit Hilfe der Implementierungsgeschichte Entwicklern helfen kann, sich in ihren Code einzuarbeiten. Es hat sich gezeigt, dass sich jeder Entwickler hin und wieder in seinen eigenen Code einarbeiten muss, weil er Teile davon vergessen hat. Es ist daher sinnvoll die Entwickler beim Erinnern an den alten Code zu unterstützen. Wir wollen die Entwickler mit einer Methode unterstützen, die auf der wichtigsten Erkenntnis von Kapitel 2.1 aufbaut. Menschen erinnern sich leichter an Informationen, wenn die äußeren Umstände zum Zeitpunkt des Erinnerns möglichst genau mit denen zum Zeitpunkt des Lernens übereinstimmen.

In diesem Kapitel wird ein Konzept entwickelt, wie ein Werkzeug aussehen könnte, dass die Überlegungen aus Kapitel 2 in die Praxis umsetzt. Das Werkzeug ist konzipiert als ein Plugin für eine bestehende integrierte Entwicklungsumgebung (IDE). In diesem Konzept trägt es den Namen Code Historian.

Das Werkzeug wurde im Rahmen dieser Arbeit implementiert. Die Implementierung weicht aber in einigen Punkten vom Konzept ab. Siehe dazu Kapitel 4.

3.1 Ziel von Code Historian

Code Historian soll dem Entwickler helfen, sich in seinen eigenen Code wieder einzuarbeiten. Aus der kognitiven Psychologie ist bekannt, dass Menschen sich leichter an Informationen erinnern, wenn die äußeren Umstände zum Zeitpunkt des Informationsabrufs mit denen zum Zeitpunkt des Lernens übereinstimmen. Dem Entwickler soll geholfen werden, indem ihm die Implementierungsgeschichte dargestellt wird. Die äußeren Umstände zum Zeitpunkt der Code-Entwicklung sollen so wiederhergestellt werden. Dies geschieht dadurch, dass nur der Code angezeigt wird, der schon existierte, als der zu erinnernde Code produziert wurde. Auf diese Weise kann der Entwickler zu jedem Code-Abschnitt leichter herausfinden, wann er ihn produziert hat und warum er für das Endprodukt nötig war.

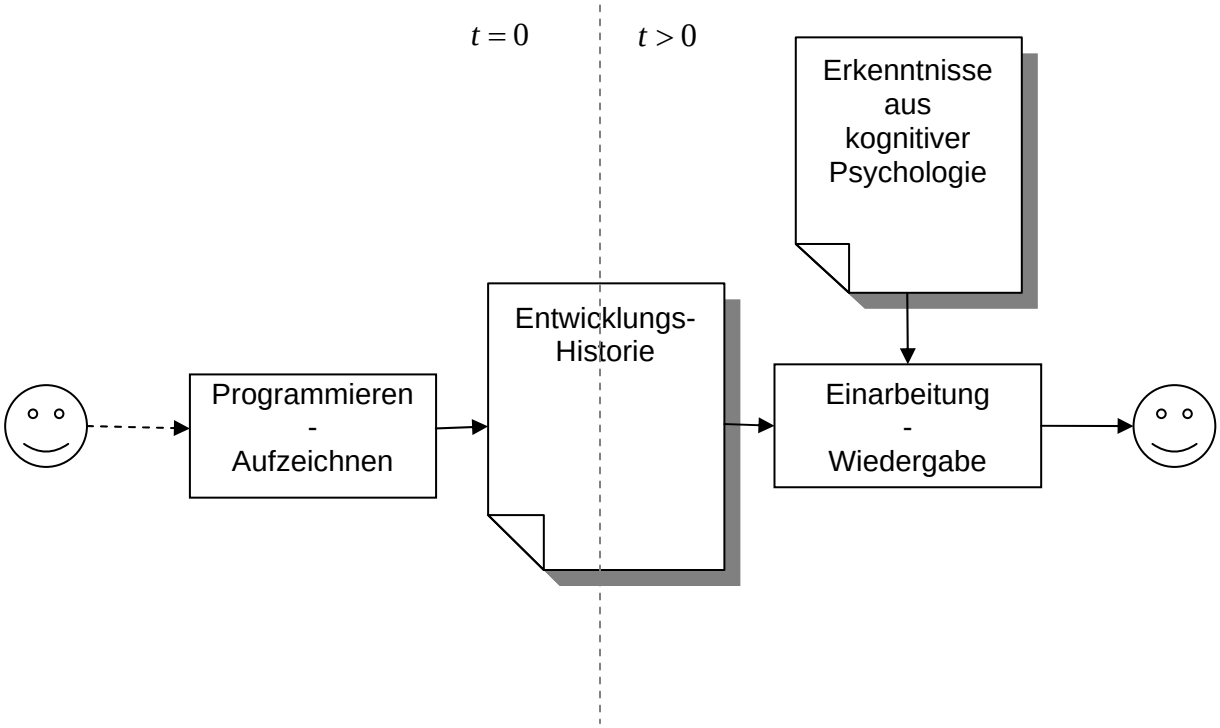
Der Entwickler soll dabei die IDE möglichst exakt so vor sich sehen, wie sie aussah, als der Code entstand. Auf diese Weise sollen die äußeren Umstände zum Zeitpunkt der Entwicklung nachgestellt werden.

Aus dem Fragebogen hat sich ergeben, dass es in Bezug auf das Erinnern eigenen Quellcodes hauptsächlich zwei Sorten von Entwicklern gibt:

1. Entwickler, die sich gut Zusammenhänge merken und Details leichter vergessen
2. Entwickler, die sich Details gut merken und Zusammenhänge leichter vergessen

Diese beiden Gruppen können auf eine ähnliche Weise unterstützt werden. Die Anforderungen sind leicht unterschiedlich, aber nicht widersprüchlich. In den Multiple Choice Fragen hat sich ergeben, dass die Gruppe, die Details leichter vergisst, etwas größer ist. Daher liegt der Fokus des zu entwickelnden Plugins darauf, Details ins Gedächtnis zu rufen.

Code Historian besteht aus zwei Teilen, wie in Abbildung 7 zu sehen ist. Die Implementierungsgeschichte muss zunächst aufgezeichnet werden, während der Entwickler programmiert. Wenn der Entwickler sich nach einer längeren Pause in den Code wieder einarbeiten möchte, muss Code Historian die Implementierungsgeschichte wiedergeben. In die Wiedergabe sollen die Erkenntnisse aus der kognitiven Psychologie einfließen. Die Wiedergabe soll in einer Weise geschehen, die die Erinnerung des Entwicklers bestmöglich stimuliert. Das heißt, die Unterschiede zwischen den angezeigten Versionen sollen genau so groß sein, dass die Code-Änderungen nicht aus dem Zusammenhang gerissen sind und der Entwickler sie leicht aufnehmen kann. Außerdem muss der Entwickler gut erkennen können, was sich gegenüber der letzten Zwischenversion geändert hat.



7 Aufbau von Code Historian

3.2 Implementierungsgeschichte

Es muss geklärt werden, was im Folgenden als Implementierungsgeschichte verstanden wird. Zur Implementierungsgeschichte gehören unterschiedliche Aspekte. Der wichtigste Teil sind die Zwischenstände, die der Code im Lauf der Entwicklung hatte. Damit sind nicht nur die Zwischenstände gemeint, die der Entwickler explizit gespeichert oder per Version Control System festgehalten hat. Die wichtigen Zeitpunkte sind immer dann, wenn für den Entwickler ein Gedankengang abgeschlossen ist. Zu diesen Zeitpunkten muss eine Zwischenversion gespeichert werden. Das Ziel ist, dass der Entwickler beim Einarbeiten sich an die zu diesen Zwischenversionen gehörenden Gedankengänge erinnert.

Ebenfalls Teil der Implementierungsgeschichte sind alle Aktionen des Entwicklers, die zur Entstehung des Codes beigetragen haben. Damit hängt eng zusammen die Oberfläche der Entwicklungsumgebung zum Zeitpunkt dieser Aktionen. Dazu gehören die Dateien, die zu diesem Zeitpunkt geöffnet waren und die aktiven Sichten auf den Code.

Es werden ein einzelner Entwickler und dessen Erinnerungsvermögen betrachtet. Aus diesem Grund gehören Änderungen am Code, die durch fremde Entwickler vorgenommen werden, erst ab dem Moment zur Implementierungsgeschichte, in dem diese Änderungen explizit importiert werden, z.B. durch ein Version Control System. Ein kurzes Beispiel zur Veranschaulichung ist dargestellt in Tabelle 5.

Zeitlicher Ablauf	**In Implementierungsgeschichte dargestellt aus Sicht von Entwickler A**
Entwickler A macht Änderung 1 Entwickler B macht Änderung 2 Entwickler A macht Änderung 3 Entwickler A importiert Änderung 2 von B	Änderung 1 Änderung 3 Änderung 2

Tabelle 5 Veranschaulichung der Strukturierung der Implementierungsgeschichte bei mehr als einem Entwickler

Die **Implementierungsgeschichte** ist die Sequenz von Zwischenständen, die der Code eines Programms während seiner Entwicklung aus Sicht eines Entwicklers durchlaufen hat. Besonders relevant für den Entwickler sind die Änderungen des Codes, die von einem Zwischenstand zum nächsten gemacht wurden.

3.3 Wiedergabe der Aufzeichnungen

Wie in Abbildung 3 gezeigt, soll ein Modul die Implementierungsgeschichte wiedergeben. Der Entwickler wählt einen Zeitpunkt aus. Code Historian stellt dann den Code auf dem Stand des gewählten Zeitpunktes dar. Es wird automatisch die zu dem Zeitpunkt geänderte Datei geöffnet. Der Entwickler soll erkennen, wie der Code entstanden ist. Die wichtigste Information für den Entwickler ist daher, was sich am Code seit der letzten Version geändert hat.

Der Entwickler soll die Änderungen, die zwischen zwei Versionen am Code gemacht wurden, schnell und übersichtlich finden. Die Änderungen sollen daher besonders markiert werden. Damit das Syntax-Highlighting im Editor wie vom Entwickler gewohnt geschehen kann, werden die Änderungen farbig hinterlegt. Es ist darauf zu achten, dass die Farben, mit denen

Änderungen markiert werden, in gutem Kontrast zu den Farben stehen, die für das Syntax-Highlighting verwendet werden. Die Änderungen schnell zu finden, hilft nicht, wenn der Code schlecht zu lesen ist, oder der Entwickler durch die Anstrengung beim Lesen schnell ermüdet.

Um dem Entwickler Sucharbeit abzunehmen, wird automatisch auf die Änderungen fokussiert. Das heißt, es wird die Datei geöffnet, die geändert wurde und an die richtige Stelle gescrollt. Gibt es mehrere Änderungen, soll versucht werden, auf die wichtigste oder größte Änderung zu fokussieren. Beispielsweise soll nicht auf veränderte Imports fokussiert werden, sondern auf die Änderungen, die die Imports erforderten.

Es gibt unterschiedliche Arten Programmcode zu ändern. Man kann manuell bestehenden Code verändern, löschen und neuen schreiben. Außerdem bieten IDEs viele Möglichkeiten Code zu generieren wie z.B. Getter und Setter für Klassen-Variablen. Manche IDEs können auch automatisches Refactoring durchführen. Beispielsweise kann Eclipse Namen einer Variablen oder Methode automatisch im ganzen Projekt durch einen anderen ersetzen.

Diese verschiedenen Arten Code zu ändern, sind von unterschiedlicher Bedeutung für den Entwickler und sein Erinnerungsvermögen. Generierter Code und refactored Code sind für den Entwickler von niedrigerem Wiedererkennungswert. Solcher Code repräsentiert keine komplexen Gedanken, an die er erinnert werden muss, sondern entspricht einfacher Schreibarbeit, die dem Entwickler durch die IDE abgenommen wurde. Manuell geschriebener Code ist für den Entwickler deutlich wichtiger. In diesem Code spiegelt sich die geistige Arbeit des Entwicklers wieder. Falls mehrere Entwickler an einem Projekt arbeiten, gibt es auch noch Änderungen, die andere Entwickler vorgenommen haben. Dieser Code ist zwar wichtiger als generierter Code, aber er hat für die Erinnerungsfähigkeit nicht, die gleiche Bedeutung wie selbst hergestellter Code. Der Entwickler soll die für seine Erinnerung wichtigeren, manuellen Code-Änderungen schnell finden und von den weniger wichtigen, automatischen Änderungen unterscheiden können. Daher sollen die unterschiedlichen Arten von Änderungen mit unterschiedlichen Farben markiert werden.

Es sollen folgende Typen von Code-Änderungen unterschieden werden

- Manuell geänderter Code
- Manuell hinzugefügter Code

- Generierter Code / Automatic Refactored Code
- Code anderer Entwickler

In Abbildung 8 ist beispielhaft dargestellt, wie Code mit markierten Änderungen aussehen könnte. Wenn man die Bedeutung der Farben kennt, kann man alle Änderungen zu letzten Version sofort erfassen.

- Zeile 6 und 7 sind neu hinzugekommen
- In Zeile 8 wurde der Startwert der Schleifenvariable auf 2 verändert
- Die Methode getFibonacci() wurde generiert

```
1
2  public class Fibonacci {
3      private int[] fibonacci=new int[20];
4
5      public void calculateFibonacci(){
6          fibonacci[0]=0;
7          fibonacci[1]=1;
8          for(int i=2;i<fibonacci.length;i++){
9              fibonacci[i]=fibonacci[i-1]+fibonacci[i-2];
10         }
11     }
12
13     public int[] getFibonacci() {
14         return fibonacci;
15     }
16 }
17
18
19
20
```

8 Beispiel Code mit Markierung der letzten Änderung

Oben wurde festgelegt, dass beim Öffnen der Dateien automatisch auf die wichtigste Änderung gescrollt werden soll, damit der Entwickler nicht in der Datei danach suchen muss. Auch dafür ist die Unterscheidung in die Änderungstypen nützlich. Auf generierten Code soll nicht fokussiert werden, falls es manuell geänderten Code gibt.

Einarbeitung in Code kann auf unterschiedlichen Ebenen geschehen. Entweder der Entwickler arbeitet sich in die Details der Implementierung ein, oder er beschäftigt sich nur mit der

Architektur des Programms. Um die Details im Code zu lernen, muss man den Code lesen. Das geschieht normalerweise im Editor. Um Architektur und Struktur von Code zu erkennen benutzen Entwickler andere Sichten. IDEs stellen für gewöhnlich die Struktur eines Projekts in einem Explorer-Fenster als Baumdiagramm dar. Dieses Baumdiagramm zeigt, welche Packages welche Klassen beinhalten und welche Methoden diese Klassen haben. Ebenfalls mögliche Sichten um sich in Code-Strukturen einzuarbeiten, wären Klassendiagramme nach UML, oder ähnliche Diagramme.

Da Code Historian eher die Entwickler unterstützen soll, die Details leichter vergessen, soll Code Historian primär Änderungen im Editor wiedergeben. Die Wiedergabe im Explorer Fenster soll erst hinzugefügt werden, wenn die Wiedergabe im Editor vollständig funktioniert. Die Wiedergabe als Klassendiagramm o. ä. wäre ein ganz neuer Ansatz und wird von Code Historian nicht verfolgt.

Die Grundidee dieser Arbeit ist, den Kontext für den Entwickler so weit wie möglich wiederherzustellen. Das bedeutet konkret, dass bei der Wiedergabe der Implementierungsgeschichte die IDE möglichst exakt so aussehen soll wie zum Zeitpunkt des Programmierens. Dafür müssen sämtliche Views und Editor-Fenster wiederhergestellt werden. In Kapitel 2.1 wurde gezeigt, dass Hinweisreize besonders hilfreich sind, wenn sie eindeutig sind, das heißt nur Hinweisreiz für eine Erinnerung sind. Die Oberfläche der Entwicklungsumgebung ist aber bei vielen Projekten gleich oder zumindest sehr ähnlich. Wir vermuten daher, dass die vollständige Wiederherstellung der Oberfläche gegenüber der reinen Wiederherstellung des Codes das Erinnerungsvermögen nur in geringem Maß verbessert. Es ist aber immenser Aufwand nötig, um die Wiederherstellung zu implementieren. Die Oberfläche lässt sich nicht exakt wieder herstellen, da für die Steuerung der Wiedergabe Platz auf dem Bildschirm benötigt wird, der während des Programmierens anderweitig belegt war. Die ganze Oberfläche ließe sich nur wiederherstellen, wenn die Steuerung der Wiedergabe auf einen Extra-Bildschirm verlegt wird. Insbesondere wegen des schlechten Kosten/Nutzen-Verhältnisses sollte die Wiederherstellung der IDE-Oberfläche nachrangig verfolgt werden.

Bei der Nutzung von Code Historian im Prototypenstadium ergab sich hin und wieder das Bedürfnis eine Datei in einer Vergleichsansicht zu öffnen, in der man den gewählten Zwischenstand direkt mit der Vorgängerversion vergleichen kann. Wenn nur eine Version im Standardeditor für die Datei geöffnet ist, ähnelt dies mehr dem Aussehen zum Zeitpunkt der

Coder-Erzeugung. Außerdem ist die verfügbare Bildschirmfläche für die Datei größer und in den meisten Fällen reicht es aus, die Änderungen zu markieren, um sie nachvollziehen zu können. Daher sollte Code Historian die Dateien per Voreinstellung im Standardeditor darstellen. In Einzelfällen ist eine Gegenüberstellung von zwei Versionen aber hilfreicher. Es sollte daher die Option geben zwei Versionen in einer solchen Gegenüberstellung zu öffnen.

3.4 Granularität der Wiedergabe

Das Programm zeichnet die Entwicklungshistorie von Programmen auf. Es muss geklärt werden, wie oft Zwischenstände abgespeichert werden sollten, um dem Entwickler bestmöglich zu helfen. Es ist nicht nötig jeden Anschlag oder jedes Wort aufzuzeichnen. Um den Entwickler zu unterstützen, muss aber fein genug aufgezeichnet werden, dass er die Änderung gegenüber der letzten Version einfach und schnell erfassen kann. Zu häufiges Zwischenspeichern der Änderungen führt dazu, dass Änderungen aus dem Zusammenhang gerissen werden. Es gibt mehr Suchaufwand, wenn einzelne Änderungen gefunden werden sollen und es müssen mehr Daten gespeichert werden.

Man kann zum Zeitpunkt der Wiedergabe noch Zwischenstände herausfiltern, die nicht hilfreich erscheinen. Man kann aber keine nicht gespeicherten Zwischenstände mehr wiederherstellen. Es sollte daher eher zu fein aufgezeichnet werden, als zu grob.

Im Abschnitt über kognitive Psychologie haben wir gesehen, dass erfahrene Entwickler Programmierstrukturen als Chunk (Erinnerungseinheit) auffassen. Entwickler können daher sieben Zeilen Code problemlos auf einmal erfassen, wenn diese in einer Klasse stehen. Sind die Zeilen auf unterschiedliche Klassen verteilt, lassen sich drei bis vier Zeilen schnell merken. Eine feinere Aufzeichnung als alle sieben geänderten Zeilen scheint daher unnötig. Da es für die Zielsetzung von Code Historian nicht nötig ist, den Code auswendig zu lernen, sondern der Programmierer nur den Sinn erkennen muss, ist vermutlich auch eine Aufzeichnung alle 10 bis 15 Zeilen völlig ausreichend.

Bei der Wiedergabe sollte ein Versionssprung möglichst einen Gedankengang des Programmierers umfassen. Auf diese Weise soll er an eben diesen Gedankengang erinnert werden, der ihn dazu veranlasste, sein Programm um diese Änderungen zu erweitern. Das

Ziel sollte es also sein, zu erkennen, wann ein Programmierer einen Gedankengang vollendet hat.

Wenn für den Entwickler ein Gedankengang zu Ende ist und er diese Gedanken fertig programmiert hat, wird er vermutlich den Code speichern. Es ist ebenso zu vermuten, dass der Entwickler seine letzten Änderungen nicht speichert, solange er mit seinen letzten Änderungen noch unzufrieden ist. Wir gehen daher davon aus, dass das Ende eines Gedankengangs äquivalent zu einem Speichervorgang durch den Entwickler ist. Dies ist natürlich nur eine Heuristik, die nicht auf alle Entwickler zutrifft.

Es gibt einige Entwickler, die ständig einen Speicher-Shortcut benutzen. In diesem Fall sollten Versionen nur aufgenommen werden, wenn sich mindestens sieben Zeilen geändert haben *und* der Entwickler speichert. Wenn ein Entwickler ständig speichert, ist es sehr schwer das Ende eines Gedankengangs zu erfassen. Ein deutliches Zeichen für einen abgeschlossenen Zwischenstand ist es, wenn er das Programm laufen lässt, oder einen Test durchführt. Das wiederum kann aber unter Umständen sehr selten erfolgen. In diesem Fall wäre es hilfreich, wenn der Entwickler bei der Wiedergabe selbst einstellen könnte, wie groß Änderungsschritte sein sollen.

Andere Entwickler speichern ihre Arbeit sehr selten ab. Die Gedankengänge sind dann umfangreicher, ein Speichervorgang entspricht aber dennoch dem Ende eines Gedankengangs. Zumindest um das Programm laufen zu lassen, wird der Entwickler speichern müssen. Dies kann im Extremfall aber sehr weit auseinander liegen. Auch in diesem Fall wäre es für den Entwickler vermutlich angenehm, wenn er bei der Wiedergabe steuern kann, wie groß Zwischenstände sein sollen.

Ein wichtiger Zwischenschritt ist abgeschlossen, wenn ein Commit in ein Version Control System (VCS) erfolgt. Wir gehen davon aus, dass ein Entwickler seine Änderungen nur hochläd, wenn er sie als abgeschlossen betrachtet. „Abgeschlossen“ muss dabei nicht zwangsläufig heißen, dass das Programm lauffähig oder fehlerfrei ist. „Abgeschlossen“ bedeutet aber, dass für den Entwickler ein Gedankengang beendet ist. Eine Ausnahme könnte auftreten, wenn ein anderer Entwickler schnell die neuste Version benötigt. Aber selbst dann wird vermutlich kaum ein Entwickler eine Version commiten, die z.B. Compilerfehler enthält.

Wir gehen davon aus, dass für den Entwickler ein Gedankengang zu Ende ist, wenn er seine Änderungen in die Versionsverwaltung hochlädt. Er wird aber nicht nach jedem Gedankengang ein Commit durchführen. Daher reicht ein Version Control System als einzige Informationsquelle für die Implementierungsgeschichte nicht aus.

Der Entwickler schreibt in seiner Commit-Nachricht, was er erreicht hat. Er gibt damit Gründe an, die ihn dazu bewegt haben, den Code in diesem Arbeitsschritt gerade so zu verändern, wie er es getan hat. Die Commit-Nachrichten können daher einen großen Beitrag leisten, den Entwickler an seine Gedanken beim Programmieren zu erinnern. Sie sollten bei der Wiedergabe angezeigt werden, oder zumindest optional einsehbar sein. Versionen aus einem Version Control System sind durch das Commit explizit vom Entwickler als Zwischenstände gekennzeichnet. Bei der Wiedergabe sollten sie daher von den nur mit einer Heuristik automatische aufgenommen Zwischenständen unterscheidbar sein.

Die Zwischenstände bei der Wiedergabe entsprechen dem manuellen Speichern durch den Entwickler. Commits durch den Entwickler werden als besondere Zwischenstände markiert und die dazu gehörenden Commit-Nachrichten angezeigt.

3.5 Steuerung der Wiedergabe

Es wird immer nur die Historie von einem Projekt gezeigt. Der Nutzer wählt zunächst aus, zu welchem Projekt er die Implementierungsgeschichte sehen möchte. Danach startet der eigentliche Wiedergabemodus.

Code Historian soll mit Zeitleisten gesteuert werden. Auf einer Zeitleiste kann ein Zeitpunkt gewählt werden und Code Historian öffnet den dazu aufgenommenen Zwischenstand.

Viele Software-Projekte sind so groß, dass es mit *einer* Zeitleiste schwierig ist, einen bestimmten Zeitpunkt zu wählen. Daher soll Code Historian zwei Zeitleisten haben. Davon wird eine die Commits repräsentieren (VCS-Zeitleiste). Wurde damit die Revision n ausgewählt, soll die zweite Zeitleiste von Revision n-2 bis Revision n+2 reichen und alle

dazwischen automatisch aufgezeichneten Zwischenstände beinhalten (Detail- Zeitleiste). Auf diese Weise kann mit der VCS-Zeitleiste schnell der richtige Zeitraum gewählt werden und anschließend mit der Detail-Zeitleiste ein konkreter Zwischenstand gewählt werden. Per Voreinstellung soll die VCS- Zeitleiste sich über sämtliche Revisionen erstrecken. Es gibt jedoch Projekte, die sehr lange laufen und für die es daher eine große Menge Revisionen gibt. Daher kann man die Start- und die End-Revision für die VCS-Zeitleiste per Tastatureingabe angeben.

Wenn kein VCS verwendet wird, soll die VCS-Zeitleiste über alle automatisch aufgenommenen Zwischenstände reichen. Auch ohne VCS ist es so einfach mit der ersten Zeitleiste grob den Zeitraum zu wählen und mit der zweiten den konkreten Zeitpunkt. Wird kein VCS verwendet und das Projekt ist klein, so kann die VCS-Zeitleiste ganz wegfallen. 80 Zwischenstände lassen sich noch gut auf einer Zeitleiste darstellen und weniger Steuerelemente machen die Benutzeroberfläche übersichtlicher.

Für jede Zeitleiste gibt es einen „weiter“- und einen „zurück“-Button. Außerdem kann der Zwischenstand per Eingabe in ein Textfeld gewählt werden.

Wir gehen davon aus, dass Programmierer häufig nur einen bestimmten Abschnitt lernen wollen. Dafür braucht Code Historian eine Suchfunktion, die zu einem gewählten Stück Code den Entstehungszeitpunkt findet. Auf diese Weise kann der Entwickler erkennen auf welchem Stand das Projekt war, als er den Code geschrieben hat und erinnert sich warum er das getan hat. Ohne die Suchfunktion müsste er einen Großteil der Implementierungsgeschichte durchlaufen, um den Bereich zu finden, der für ihn relevant ist.

3.6 Nicht-Funktionale Anforderungen

Code Historian soll den Entwickler unterstützen und seine Produktivität steigern, indem es die Einarbeitungszeit in Code verkürzt. Wenn Code Historian den Entwickler beeinträchtigt, während er programmiert, senkt das seine Produktivität. Die IDE darf daher durch die Aufzeichnung der Entwicklungshistorie nicht merklich langsamer werden. Es darf keine häufigen Rückfragen durch Code Historian geben.

Wenn Code Historian kleine interne Bugs hat, ist das zu verkraften. Es darf aber keinesfalls die IDE negativ beeinflussen oder zu Abstürzen führen. All dies ließe die Produktivität des Entwicklers sinken. Das Ziel wäre deutlich verfehlt.

Das Einarbeiten in Code soll mit Code Historian schneller gehen als ohne. Alle durch Code Historian verursachten Verzögerungen wirken sich daher negativ auf den Projekterfolg aus. Deshalb muss die Bedienung so intuitiv sein, dass keine lange Einarbeitungszeit nötig ist. Die Steuerung von Code Historian muss schnell reagieren. Wenn der Entwickler die früheren Versionen des Codes durchsucht, müssen die geänderten Dateien schnell geöffnet werden, damit der Entwickler keine Wartezeiten hat. Der Entwickler muss gut erkennen können, welche Änderungen gegenüber der letzten Version gemacht wurden.

3.7 Zusätzliche Features

Es gibt noch ein paar Funktionen für Code Historian, die zwar nützlich erscheinen, jedoch für den Projekterfolg eher unbedeutend sind. Sie sollen hier vorgestellt werden.

Häufig sind Fehler der Grund für Änderungen am Programm. Ein gefundener Bug führt zu Bugfixes. Die Konsolen- und Fehlerausgaben können so helfen sich zu erinnern. Ähnlich wie Commit-Nachrichten geben sie an, warum Änderungen gemacht wurden. Dadurch kann man die Änderungen schneller nachvollziehen. Commit-Nachrichten sind dabei deutlich aussagekräftiger, doch einen geringen Erinnerungswert können auch Konsolen- und Fehlerausgaben haben. Es sollen daher auch Konsolen- und Fehlerausgaben zu Zwischenständen des Codes durch Code Historian protokolliert werden. Da der Erinnerungswert aber nicht sehr hoch ist, ist dies bei der Implementierung von Code Historian nachrangig zu verfolgen.

Es wäre nützlich, wenn man die Archive, in denen die Zwischenstände gespeichert sind einfach importieren, exportieren und zusammenfügen könnte. Man sollte das Archiv sowohl als Datei exportieren können als auch direkt in ein Version Control System. Insbesondere wenn ein Entwickler an mehreren Computern abwechselnd an einem Projekt arbeitet, z.B. im Büro und zu hause, wäre es eine Verbesserung, wenn das Archiv per Version Control System übertragen werden kann.

Die Granularität der Aufzeichnung und Wiedergabe beruht auf Annahmen. Es kann sich herausstellen, dass eine andere Granularität für die Entwickler angenehmer ist. Dies ließe sich einfach realisieren, wenn man die Granularität konfigurieren könnte, ohne den Quellcode von Code Historian zu verändern. Dann könnte jeder Entwickler Code Historian so anpassen, wie es ihm am hilfreichsten erscheint. Ebenso sollte es möglich sein, Projekte von der Protokollierung der Implementierungsgeschichte auszunehmen.

Eine Funktion, die die Implementierungsgeschichte abspielt wäre schön. Die Geschwindigkeit zum Fortschreiten muss dabei vom Umfang der Änderungen abhängen. Die Geschwindigkeit muss außerdem einstellbar sein. Das wäre eine schöne Funktion zu Demonstrationszwecken. Um sich einzuarbeiten ist aber sinnvoller, bei jedem Zwischenstand genau so lange zu bleiben, wie man braucht ihn aufzunehmen. Das kann sowohl deutlich länger als auch kürzer sein als bei einer automatischen Wiedergabe.

4 Implementierung eines Prototypen

Im Rahmen dieser Arbeit wurde ein Werkzeug erstellt, das dem Konzept zur Nutzung der Implementierungsgeschichte entspricht, soweit es in der gegebenen Zeit möglich war. In diesem Kapitel ist unter Code Historian die konkrete Implementierung des unter Kapitel 3 erstellten Konzepts zu verstehen. Das Werkzeug ist wie im Konzept aufgeteilt in die Aufzeichnung, genannt Code Historian Monitor und die Wiedergabe, genannt Code Historian Replay.

4.1 Rahmenbedingungen und Umfeld

Es erschien sinnvoll, das Werkzeug an eine bestehende IDE anzubinden. Das Ziel war möglichst viele bestehende Funktionen der IDE zu nutzen. Eclipse ist Open Source, bietet Schnittstellen um Erweiterungen einzubinden und wird im Fachgebiet, an dem diese Arbeit entstand, häufig genutzt. Daher wurde Code Historian als Plugin für Eclipse hergestellt.

Die im Rahmen dieser Arbeit erstellte Implementierung ist ein Prototyp. Sie dient dazu, die Hypothese zu unterstützen, dass sich Entwickler schneller in ihren eigenen Code einarbeiten können, wenn ihnen der Kontext zum Entstehungszeitpunkt des Codes präsentiert wird. Code Historian dient nicht dazu, in einer industriellen Umgebung eingesetzt zu werden.

Das Plugin soll von Software-Entwicklern beim Programmieren genutzt werden. Da an der Universität Hannover für die Lehre meist Java genutzt wird, ist die primäre Ausrichtung die Unterstützung von Java-Code. Das soll helfen das Plugin mit Studenten zu evaluieren. Es wäre gut, wenn es auch für andere Sprachen nutzbar wäre, doch das ist ein nachrangiges Ziel.

4.2 Code Historian Monitor

Das Ziel des Monitors ist die Aktionen des Entwicklers während des Programmierens aufzuzeichnen. Aus dem Konzept ist bekannt wie fein mindestens aufgezeichnet werden soll.

- Commits durch den Entwickler werden als besondere Zwischenstände angezeigt und die Commit-Nachrichten angezeigt.
- Die Zwischenstände bei der Wiedergabe entsprechen dem manuellen Speichern durch den Entwickler.

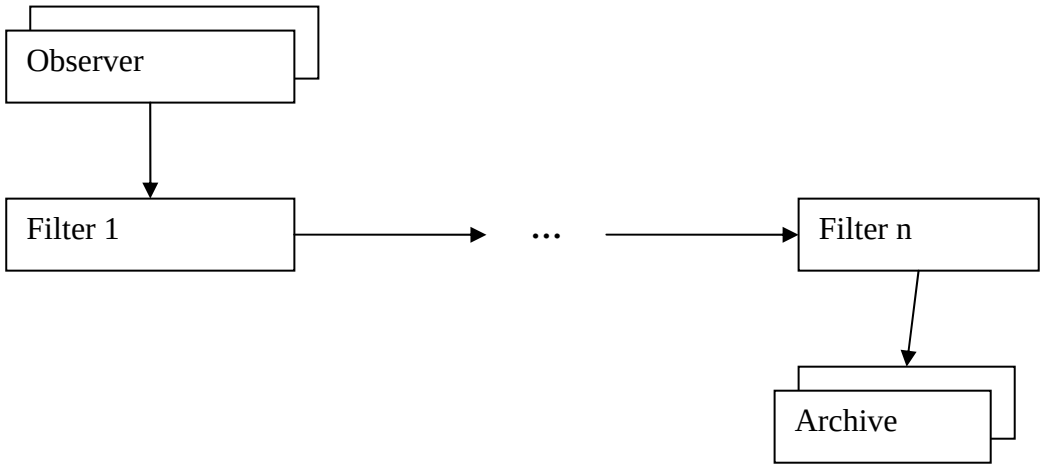
Es muss also mindestens nach jedem Speichervorgang durch den Entwickler eine Version gespeichert werden. Die Zwischenstände durch Commits lassen sich aus dem entsprechenden Repository beziehen. Laut Konzept soll Code Historian die Änderungstypen unterscheiden können. Dazu ist eine noch genauere Überwachung der Aktionen des Entwicklers nötig.

Die Architektur ist so konzipiert, dass der Monitor leicht erweitert werden kann. Der Monitor besteht aus folgenden drei Grundkonzepten.

Observer erkennen Aktionen des Entwicklers wie Speichervorgänge, Commits und Refactorings. Sie geben die geänderten Ressourcen weiter an Filter. Es können mehrere Observer gleichzeitig genutzt werden.

Filter dienen dem Ausfiltern von zu archivierenden Dateien oder Versionen. So können zum Beispiel .class Dateien aussortiert werden oder es kann verhindert werden, dass zwischen zwei gespeicherten Versionen weniger als fünf Sekunden liegen. Filter werden nacheinander durchlaufen.

Archive dienen dem abspeichern der Versionen. Es ist prinzipiell möglich mehrere Archive zu nutzen, doch normalerweise sollte eins reichen.



9 Struktur von Code Historian Monitor

Da Code Historian Monitor sehr viele Daten archiviert, war es ein Ziel den Speicher effizient zu nutzen. Es sollte aber auch nicht viel Zeit dafür verloren gehen effiziente Speichermethoden zu entwickeln. Daher verwendet der Monitor ein SVN-Repository.

Code Historian Monitor nutzt das SVNKit [10]. SVNKit ist eine Java-Bibliothek, die die Nutzung von SVN-Repositories unterstützt. Der Monitor nutzt die Funktion von SVNKit ein Repository auf der lokalen Festplatte zu simulieren. Es wurde versucht die Daten in ein Repository zu speichern, das nicht lokal auf der Festplatte lag, sondern per Internet angesprochen werden musste. Dies erwies sich aber als zu langsam. Da sehr regelmäßig Zwischenstände gespeichert werden, führt das zu häufigen Commits. Die häufigen Commits bremsten Eclipse merklich aus. Das war nach dem Konzept für Code Historian verboten. Die Speicherung in ein Repository auf der lokalen Festplatte schien so die beste Alternative.

Code Historian Monitor archiviert Versionen, wann immer eine Datei gespeichert wird. Das entspricht der Vorgabe aus dem Konzept, dass die Granularität der Wiedergabe den Speichervorgängen entspricht. Dies war dank einer Schnittstelle in Eclipse namens *IResourceChangeListener* einfach zu realisieren.

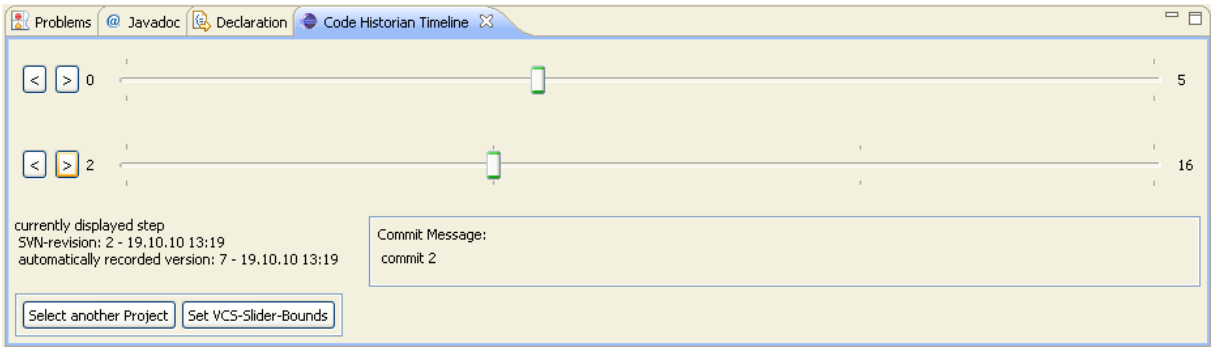
In Code Historian Monitor ist derzeit keine feinere Aufzeichnung implementiert als Zwischenstände bei jedem Speichervorgang aufzunehmen. Es ist Code Historian daher nicht möglich manuelle Änderungen, Änderungen durch andere Programmierer und automatische Änderungen zu unterscheiden. Dazu wäre eine feinere Überwachung der Aktionen des Entwicklers nötig. Aus Zeitmangel konnte diese Überwachung nicht mehr realisiert werden. Mit Hilfe von zusätzlichen Observern lässt sich die feinere Überwachung nachträglich einbauen. Es gibt dafür jedoch keine einfach anzusprechende Schnittstelle in Eclipse. Es müssten eine Vielzahl von Listenern an unterschiedlichen Stellen eingebaut werden.

Es gibt Plugin für Eclipse namens Mylyn. Mylyn speichert ebenfalls Kontextinformationen und Benutzerinteraktionen mit Eclipse. Den Monitor von Mylyn für Code Historian zu übernehmen ging leider nicht. Mylyn speichert keine zeitliche Abfolge, sondern generiert einen Degree-of-Interest-Tree. Außerdem werden keine Versionen des Quellcodes gespeichert. Mehr zu dem Zweck von Mylyn ist im Kapitel über ähnliche Arbeiten zu finden.

Ein Teil von Mylyn könnte für Code Historian benutzt werden. Mylyn fasst mit dem Interface *IInteractionEventListener* viele Listener zusammen und bietet so eine sehr feine Überwachung der Nutzeraktionen. Bisher nutzt Code Historian dieses Interface aus zwei Gründen nicht. Der erste Grund ist, dass die Nutzung hätte erheblichen Mehraufwand für die Implementierung verursacht. Es erfordert Aufwand aus den abgefangenen Events von Eclipse für Code Historian verwertbare Daten zu extrahieren und es gibt sehr viele Events, sodass zusätzliche Filter nötig gewesen wären. Aus Zeitmangel konnte das nicht mehr realisiert werden. Der zweite Grund ist, dass die Schnittstelle nicht Teil der offiziellen API ist und daher mit jeder Aktualisierung von Mylyn hinfällig werden könnte.

4.3 Code Historian Replay

Code Historian Replay dient dazu, die aufgezeichnete Implementierungsgeschichte wiederzugeben auf eine Weise, die dem Entwickler beim Erinnern helfen soll. Replay lässt sich über eine View steuern. (Abb. 10)



10 View für die Steuerung von Code Historian

Wenn der Entwickler mit den Zeitleisten eine Version gewählt hat, öffnet Code Historian Replay alle Dateien, die sich in dieser Version gegenüber der letzten geändert haben. In der Regel ist das nur eine. Die Datei wird mit dem Editor geöffnet, der standardmäßig von Eclipse für den Dateityp verwendet wird. Dadurch wird am ehesten die Umgebung zum Zeitpunkt der Programmierung wieder hergestellt. Ist der geöffnete Editor ein Texteditor, werden die Änderungen in der Datei gegenüber der letzten Version farbig markiert.

Da vom Monitor nur die Dateiversionen gespeichert werden und nicht auf welche Art sie entstanden sind, kann Replay nur neue und geänderte Code-Teile unterscheiden. Generierter

oder refactored Code wird dementsprechend nicht extra markiert, sondern erscheint ebenfalls als neuer oder geänderter Code.

Um die geänderten Code-Teile zu erkennen, wird der von Eclipse bereitgestellte Compare-Algorithmus der Klasse *RangeDifferencer* verwendet. Dieser benötigt einen Tokenizer, der den Text in vergleichbare Tokens zerlegt. Handelt es sich um Java-Code verwendet Code Historian einen speziellen Java-Tokenizer [11]. Ansonsten wird ein eigner Tokenizer für allgemeinen Text benutzt.

Dem Konzept entsprechend wird Code Historian mit zwei Zeitleisten gesteuert. Falls für das ausgewählte Projekt ein SVN-Repository verwendet wurde, enthält die obere Zeitleiste sämtliche Revisionen. Wurde kein SVN-Repository verwendet enthält die obere Zeitleiste alle automatisch aufgenommenen Zwischenstände. Die Grenzen der unteren Zeitleiste hängen von der mit der oben Zeitleiste ausgewählten Version ab. Über die untere Zeitleiste lassen sich die automatisch erzeugte Zwischenstände aus dem begrenzten Zeitraum wählen.

4.4 Inkrementelle Arbeit

Es war abzusehen, dass sich noch während der Entwicklung neue Anforderungen ergeben würden. Für den Autor dieser Arbeit war die Entwicklung eines Eclipse-Plugins neu. Spikes hatten gezeigt, dass die Einarbeitung in die unterschiedlichen Schnittstellen für Plugins viel Zeit brauchen. Daher war schon früh zu erkennen, dass die Implementierung am Ende des Bearbeitungszeitraums nicht alle Anforderungen erfüllen kann, die sich aus dem Konzept ergaben.

Es sollte inkrementell entwickelt werden, um Entwicklung anhand des jeweils aktuellen Inkrements die Funktion des Plugins zu prüfen, neue Anforderungen zu finden und diese nachträglich einzubauen.

Nach dem letzten Inkrement hat die Implementierung folgende Funktionen:

- Die Implementierungsgeschichte wird aufgezeichnet, d.h. Änderungen am Code
- Die Implementierungsgeschichte wird im Editor wiedergegeben
- Änderungen am Code werden farbig markiert. Es wird zwischen neuem und geändertem Code unterschieden

- Es wird im Editor automatisch auf Änderungen fokussiert
- Vom Entwickler genutztes SVN wird eingebunden und die entsprechenden Commit-Nachrichten angezeigt
- Die Steuerung funktioniert wie im Konzept über zwei Zeitleisten. Deren Optik könnte aber noch verbessert werden

Folgende Funktionen fehlen noch:

- Aktionen des Nutzers werden nicht aufgezeichnet

  → Automatisierte, fremde und manuelle Code-Änderungen werden nicht unterschieden
- Andere Views wie z.B. der Package Explorer werden nicht durch Code Historian beeinflusst. Klassen und Packages die zum gewählten Zeitpunkt nicht existierten werden im Package Explorer nicht ausgeblendet
- Suchfunktion, die zu Code-Abschnitten den Implementierungszeitpunkt findet, fehlt

4.5 Evaluation

Um eine ausführliche Evaluation durchzuführen, fehlte leider die Zeit. Wie eine solche Evaluation aussehen könnte, wird in Kapitel 5.2.3 erläutert. Es wurde jedoch eine kleine Evaluation durchgeführt werden, um zumindest eine Tendenz festzustellen, ob Code Historian seine Aufgabe erfüllt oder nicht.

Einigen Entwicklern wurde eine Programmieraufgabe gestellt. Sie sollten einen Algorithmus entwickeln, der *n* Schachdamen auf einem Schachfeld von *n* mal *n* Feldern platziert, ohne dass diese sich gegenseitig schlagen können. Etwas mehr als zwei Wochen nachdem sie diese Aufgabe erfüllt hatte, wurden die Entwickler gebeten, sich mit Code Historian wieder in den Code einzuarbeiten. Anschließend sollten sie in einem Fragebogen ausfüllen. Die Entwickler sollten einschätzen in wie weit Code Historian ihnen geholfen hat. Diesen Fragebogen haben vier Personen ausgefüllt. Die Aufgabenstellung und der Fragebogen sind im Anhang zu finden.

Es wurde gefragt: „Wieviel vom Code hast du vergessen?“ Es wurde erwartet, dass nach nur zwei Wochen wenig bis gar nichts vergessen wurde. Wie in Tabelle 6 zu sehen, waren die Antworten aber weit gestreut. Das ist eine hilfreiche Information für spätere Evaluationen.

Wenn wir davon ausgehen, dass schon nach zwei Wochen bei den meisten Entwicklern das Vergessen einsetzt, so reicht ein Vergessenszeitraum von wenigen Wochen aus. Eine Evaluation könnte so unter Umständen innerhalb nur eines Semesters im Vorlesungszeitraum durchgeführt werden.

Viel	1
Mittel	2
Wenig	0
Gar nichts	1

Tabelle 6 Wieviel hast du vergessen?

Die wichtigste Frage war, ob Code Historian beim Einarbeiten geholfen hat. Die Probanden wussten zwar, wozu Code Historian gedacht war, doch sie kannten Benutzeroberfläche nicht. Obwohl sie erst die Benutzung des Werkzeugs erlernen mussten, antworteten zwei Personen dass sie sich schneller einarbeiten konnten. Nur eine Person fühlte sich verlangsamt.

stark beschleunigt	0
beschleunigt	2
nicht verändert	1
verlangsamt	1
stark verlangsamt	0

Tabelle 7 Hast du das Gefühl, dass dir Code Historian die Einarbeitung beschleunigt hat?

Die Aufgabe, die die Probanden lösen mussten, war relativ kurz. Der Code, in den sie sich einarbeiten mussten, war daher nicht sehr umfangreich. Code Historian ist eigenlich konzipiert, um bei größeren Projekten eingesetzt zu werden. Die Probanden sollten daher abschätzen, ob Code Historian ihnen helfen würde, wenn das durchgeführte Projekt umfangreicher wäre und der Vergessenzeitraum größer. Eine Person antwortete darauf nicht. Alle anderen vermuteten, dass Code Historian die Einarbeitung beschleunigen würde.

Die Probanden sollten noch zu den folgenden Aussagen angeben, ob sie ihnen zustimmen.

Ich finde es im Großen und Ganzen schon ganz gut	1
Die Implementierungsgeschichte wiederzugeben bringt grundsätzlich nichts	0
Steuerung der Wiedergabe / GUI	3
Versionsschritte müssen größer	0
Versionsschritte müssen kleiner	1
Ich hätte mir noch ein bestimmtes Feature gewünscht	4

Auffällig ist, dass sich alle noch zusätzliche Features wünschen. Und drei der vier Probanden Veränderungen an der GUI möchten. Bei den Fragen, ob die Versionsschritte größer oder

kleiner sein müssten, antwortete eine Person, die Schritte sollten kleiner sein. Wegen der geringen Anzahl an Probanden lässt sich aber keine Aussage treffen, ob das ein Einzelfall ist, oder ob die eine Person eine relevante Quote von Entwickler repräsentiert, für die die Schrittgröße angepasst werden sollte.

Insgesamt kann man aus den Ergebnissen des Fragebogens ableiten, dass Code Historian bei der Einarbeitung hilft. Es wäre jedoch noch hilfreicher, wenn die eine oder andere Funktion zusätzlich implementiert wäre und die Steuerung intuitiver wäre.

4.5.1 Verbesserungsvorschläge

In der letzten Frage des Evaluationsbogens wurden die Probanden gebeten konkrete Verbesserungsvorschläge zu machen. Die Vorschläge waren recht unterschiedlich. Einige Vorschläge, die gemacht wurden, sind im Konzept enthalten, wurden jedoch noch nicht implementiert, andere sind ganz neu.

Im Konzept schon bereits beschrieben und in den Fragebögen ebenfalls vorgeschlagen wurden:

- Eine Play-Funktion
    - Galt im Konzept als Zusatzfeature
- Erkennung der Nutzeraktionen wurde von zwei Personen vorgeschlagen
    - Beispiel eines Probanden war die automatische Codeformatierung. Code Historian zeigt nach der Formatierung eine große Menge Änderungen, aber sie sind alle unbedeutend
    - Der andere Proband wollte erkennen können, wozu die angezeigte Zwischenversion verwendet wurde. Ob die Zwischenversion ein Commit war, oder das Programm in dieser Version gestartet wurde, oder Eclipse gerade gestartet wurde oder anderes
- Direkter Vergleich von Versionen im Merge Viewer von Eclipse
- Darstellung der Änderungen im Package Explorer
- Anzeigen von Fehlerausgabe oder JUnit-Ausgabe
    - Galt im Konzept als Zusatzfeature

Eine Person fand die Granularität der Wiedergabe zu grob. Sie empfand die Versionsschritte nicht als Gedankenschritte. Für diese Person war die Heuristik, dass Speichern dem Abschluss eines Gedankengangs entspricht, offensichtlich nicht gültig.

Eine andere Person möchte die Zwischenversionen gerne nachträglich gruppieren und kommentieren können. Für das Konzept wurde überlegt das Kommentieren von Versionen aufzunehmen. Es wurde aber verworfen, weil vermutet wurde, dass das Kommentieren mehr Aufwand verursacht, als es Nutzen für die Einarbeitung bringt.

Alle Probanden wollten die GUI verändern. Die Zeitleisten erweisen sich nicht als intuitiv bedienbar. Die Zeitleisten wurden mit den Slidern realisiert, die in der Grafikbibliothek vorhanden waren. Eine speziell angepasste Version, wäre vermutlich besser gewesen. Zwei Personen schlagen vor die Versionen zusätzlich als Liste anzuzeigen. Eine solche Liste wird auch im Werkzeug Replay verwendet, das im Kapitel über ähnliche Arbeiten vorgestellt wird. Es erscheint sinnvoll eine zweite View für Code Historian herzustellen, die eine solche Liste anzeigt. Die beiden Views könnten dann alternativ zueinander oder auch zusammen verwendet werden.

5 Diskussion

5.1 Ähnliche Arbeiten

Es gibt bereist unterschiedliche Ansätze, die Entstehungsgeschichte von Code darzustellen und auch zu nutzen. Teilweise dienen sie zur Visualisierung, teilweise sollen sie dem Entwickler helfen.

5.1.1 Gource

Gource ist ein Open Source Projekt. Es visualisiert die Entwicklung eines Programms anhand eines Version Control Systems. Unterstützt werden derzeit Git, Bazaar und Mercurial. Mit Third Party Programmen funktioniert es auch für SVN- und CVS-Repositories. Dargestellt wird ein Baum mit dem root-Verzeichnis als Wurzel. Äste entsprechen Verzeichnissen, Blätter entsprechen Dateien. Außerdem gibt es Programmierer, die durch den Raum schweben und die Äste sowie die Blätter erzeugen. Gource animiert, wie der Baum wächst. Mit Gource lässt sich der Quellcode nicht betrachten. Es visualisiert aber anschaulich, in welcher Reihenfolge verschiedene Bereiche des Programms entstanden sind. Es zeigt die Dynamik, mit der sich Bereiche entwickelt haben und wie viele Entwickler daran beteiligt waren. [12]

11 Gource-Darstellung des Blender-Projekts: http://code.google.com/p/gource/wiki/Screenshots

5.1.2 SVN Time-Lapse View

SVN Time-Lapse View zeigt die Entstehung einer Datei anhand ihrer Historie in einem SVN-Repository. Mit einer Scrollbar kann man durch die unterschiedlichen Revisionen navigieren. Dargestellt wird der Code in zwei Fenstern ähnlich wie bei einem Merge-Editor. Ein Fenster zeigt die alte und ein Fenster die neue Version an. Änderungen werden farbig markiert. Der auffälligste Unterschied zwischen SVN Time-Lapse View und dem in dieser Arbeit erstellten Werkzeug (Code Historian) ist, dass SVN Time-Lapse View die Historie einer Datei darstellt und Code Historian die Historie eines ganzen Projekts.

SVN Time-Lapse View ist an keine Entwicklungsumgebung angebunden. Da Code Historian ein Eclipse-Plugin ist, lassen sich viele Funktionen von Eclipse während der Verwendung von Code Historian nutzen, die SVN Time-Lapse View fehlen. Dazu gehören z.B. Syntax-Highlighting, ein Package Explorer oder die Möglichkeit automatisch vom Aufruf zur Deklaration einer Methode zu springen.

5.1.3 Mylyn

Mylyn ist ein Plugin für Eclipse. Es wurde unter dem Namen Mylar von Mik Kersten für seine Doktorarbeit entwickelt. Mylyn soll ebenfalls dem Entwickler helfen sich zu erinnern. Es verfolgt dabei aber einen anderen Ansatz als Code Historian. Der Fokus von Mylyn liegt auf kurzen Unterbrechungen. Entwickler werden häufig gestört. Sie müssen kurzfristig ihre derzeitige Aufgabe unterbrechen, um eine andere Aufgabe höherer Priorität zu erledigen. Anschließen müssen sie an ihre ursprüngliche Aufgabe wieder anknüpfen. Mylyn versucht dies zu unterstützen, in dem für jede Aufgabe eine eigene Sicht auf Eclipse erzeugt wird. Mylyn verwendet ein Modell namens Degree of Interest (DOI).

Der Ablauf ist folgender. Der Entwickler legt in Mylyn einen Task an und aktiviert ihn. Ab jetzt berechnet Mylyn für jede Datei anhand der Verwendungshäufigkeit dieser Datei dem DOI. Dem Entwickler angezeigt, werden nur solche Dateien, die einen gewissen Schwellenwert überschreiten. Für jeden angelegten Task werden die Degress of Interest neu berechnet. Auf diese Weise kann der Entwickler zwischen Tasks hin und her springen und bekommt jeweils nur die Dateien angezeigt, die für diesen Task relevant sind. Beim Verlassen eines Tasks speichert Mylyn außerdem welche Dateien, Views und Editoren gerade geöffnet waren. Dadurch kann der Entwickler beim wieder aufnehmen des Tasks exakt an der Stelle

weitermachen, an der er aufgehört hat. Ohne Mylyn müsste er erst wieder alle Dateien suchen und öffnen, die er braucht.

Mylyn gibt für jeden Task aber nur den letzten Stand wieder. Das hilft für kurzzeitige Unterbrechungen gut, wie auch die Evaluation von Mylyn gezeigt hat. Der Fokus von Code Historian liegt auf langfristigen Unterbrechungen. Dann hat das Vergessen schon stärker eingesetzt und es hilft deutlich weniger, den letzten Stand zu sehen. Code Historian zeigt dem Entwickler nicht nur an welche Daten er bearbeitet hat, sondern auch *was* er geändert hat. Insbesondere die Oberfläche von abgeschlossenen Tasks kann nur schwer wiedergeben, was während der Bearbeitung des Tasks passiert ist. Dies ist aber von Bedeutung, wenn nach längerer Zeit auffällt, dass doch noch etwas geändert werden muss. [13]

5.1.4 Replay

Replay ist ein Werkzeug, das ähnlich wie Code Historian vergangene Änderungen wiedergeben kann. Es ist ebenfalls als Eclipse Plugin realisiert. Die Zielrichtung von Replay ist jedoch nicht wie bei Code Historian das Erinnern an eigenen Code sondern das Verstehen von Änderungen, die andere Entwickler gemacht haben. Insbesondere soll Replay helfen Fehler zu finden, die durch erfolgte Änderungen neu hinzugekommen sind. Replay speichert eine Version zwischen, wenn kompiliert wird. Da in Eclipse standardmäßig jeder Speichervorgang auch Kompilieren auslöst, werden die Änderungen werden also mit der gleichen Granularität erfasst in der Implementierung von Code Historian.

Anders als Code Historian zeigt Replay die Menge der gespeicherten Versionen nicht mit einer Zeitleiste, sondern in einer Liste an. Die Liste lässt sich nach mehreren Kriterien filtern. Die Versionen sind nach Erstellungsdatum sortiert unabhängig vom Entwickler. Code Historian sortiert die Version ebenfalls nach der Zeit, doch Änderungen durch andere Entwickler werden erst für den Zeitpunkt einsortiert, zu dem diese vom dem betrachtete Entwickler importiert werden.

Die Darstellung des Codes erfolgt im Merge-Editor. Code Historian hingegen verwendet den Standardeditor der jeweiligen Datei und modifiziert diesen nur leicht durch Markieren der Unterschiede gegenüber der letzten Dateiversion. [14][15]

5.1.5 Degree-of-Interest oder Zeitleiste

Entwickler werden häufig während ihrer Tätigkeit gestört. Diese Störungen haben einen negativen Einfluss auf die Effektivität. Chris Parnin und Robert DeLine haben in [16] zwei Strategien untersucht, wie man Entwickler am besten helfen, kann ihre unterbrochene Tätigkeit leichter wieder aufzunehmen.

Die erste Strategie ähnelt Mylyn. Es wird ein degree-of-interest (DOI) Baum angezeigt. In diesem Baum werden die Programmteile (Projekte, Dateien, usw.) je nach Wichtigkeit ausgegraut, normal angezeigt oder fett geschrieben.

Die zweite Strategie ist dem Entwickler eine chronologisch sortierte Liste seiner letzten Aktivitäten zu zeigen. Diese Liste nimmt die Aktivitäten feingranularer auf, als Code Historian. Es werden nicht nur Speichervorgänge, sondern auch jede Code-Auswahl und Änderung aufgezeichnet.

Es zeigte sich in einer Untersuchung, dass beide Strategien den Entwicklern in gleichem Maß helfen. In einer anschließenden Befragung gaben die Teilnehmer aber an, dass die chronologisch sortierte Liste subjektiv als angenehmer empfunden wurde.

5.2 Ausblick

5.2.1 Erweiterung der Implementierung

Die Implementierung von Code Historian lässt sich noch um einiges erweitern. Wie schon in Kapitel 4.4 beschrieben fehlen einige Funktionen, die das Konzept beschreibt.

- Aktionen des Nutzers aufzeichnen und automatisierte und manuelle Code-Änderungen unterscheiden
- Package Explorer auf früheren Stand wiederherstellen
- Suchfunktion

Auch die in Kapitel 4.5 behandelte Evaluation der Implementierung ergab viele Anregungen für Veränderungen. Insbesondere die Benutzeroberfläche benötigt eine Überarbeitung, damit Code Historian einfacher zu bedienen ist.

Bisher kann man als Versionsverwaltung nur SVN-Repositories anbinden. Das ist ausreichend um zu überprüfen, ob das Verfahren grundsätzlich funktioniert. Wenn man Code Historian für die tägliche Arbeit einsetzen möchte, wäre aber die Anbindung weiterer Versionsverwaltungssystem wie CVS, Git oder Mercurial sinnvoll. Bisher liegt der Fokus von Code Historian klar auf Java. Um einen erweiterten Nutzerkreis zu erreichen, sollte man auch spezielle Module für die Unterstützung von C++ hinzufügen.

5.2.2 Wiedergabe in UML-Diagrammen

Bisher wurde die Implementierungsgeschichte nur in Form von Quellcode wiedergegeben. Insbesondere wenn man sich in die Struktur eines Projekts einarbeiten möchte, könnten aber auch andere Sichten auf den Quellcode hilfreich sein. Man könnte beispielsweise UML-Klassendiagramme aus dem Code generieren. Wenn man die Implementierungsgeschichte in dieser Form abspielt, könnte der Entwickler sehr schnell erkennen, wie Package- und Klassen-Strukturen entstanden sind.

Die Darstellung in Klassendiagrammen o. ä. entspricht nicht dem Ansatz, die IDE möglichst so wieder herzustellen, wie der Programmierer sie zum Zeitpunkt der Codeentstehung vor sich hatte. Der Kontext, der dem Entwickler beim Erinnern helfen soll, wäre nur in abstrakter Form gegeben. Gerade diese abstrakte Form ist zum Verstehen der Struktur aber möglicherweise hilfreicher als der Quellcode.

5.2.3 Evaluation

Es wurde bisher nur eine kleine Evaluation durchgeführt. Es haben nur wenige Entwickler teilgenommen. Das durchgeführte Projekt war in ca. einer Stunde zu bewältigen. Die Entwickler konnten nur angeben, ob die Einarbeitung mit Code Historian subjektiv „gefühlt“ schneller geht. Ein Vergleich, in dem die Dauer mit und ohne Werkzeugunterstützung gemessen wird, wurde nicht gemacht. Um zu überprüfen, ob Code Historian das Einarbeiten in Code beschleunigt, müsste ein umfangreicheres Software-Experiment durchgeführt werden. So ein Experiment war im Bearbeitungszeitraum der Aufgabe nicht realisierbar. Es soll aber skizziert werden, wie ein solches Experiment durchzuführen wäre.

Code Historian soll bei folgendem Szenario helfen: Ein Entwickler schreibt ein Programm. Längere Zeit später muss er sich erneut einarbeiten, um eine Erweiterung zu produzieren.

Um zu testen, ob Code Historian dabei hilft, müssten einige Entwickler diese Aufgabe einmal mit und einmal ohne Code Historian ausführen. Dann könnte man sehen, ob es mit Code Historian schneller ging.

Offensichtlich kann aber nicht der gleiche Entwickler zweimal die gleiche Erweiterung schreiben. Er wäre beim zweiten Mal deutlich schneller unabhängig von Code Historian. Er kann aber auch nicht zwei unterschiedliche Erweiterungen schreiben, da der Umfang der Erweiterungen vermutlich mehr Einfluss auf die Dauer hat als die Nutzung von Code Historian. Es gibt zwei Alternativen, um diesem Problem zu begegnen.

Es werden mehrere Probanden benötigt. Jedem Proband wird die gleiche Aufgabe für ein Software-Projekt gestellt. Nach der Durchführung der Aufgabe haben die Versuchspersonen Zeit zu vergessen, wie sie die Aufgabe gelöst haben. Anschließend müssen sie eine Erweiterung für das Projekt herstellen. Die Hälfte der Probanden kann dabei Code Historian nutzen, die andere nicht. Wenn Code Historian einen positiven Einfluss auf die Einarbeitungszeit hat, sollte die Hälfte mit Unterstützung durch Code Historian schneller die Erweiterung fertig stellen als die Hälfte ohne.

Damit aus dem Experiment ein valides Ergebnis resultiert, müssen ein paar Faktoren erfüllt sein. Es müssen genug Probanden teilnehmen, sodass anschließend statistisch signifikante Aussagen getroffen werden können. Der produzierte Code muss umfangreich und komplex genug sein, sodass es merklicher Aufwand ist, sich erneut einzuarbeiten. Die Probanden müssen genug Zeit haben, zu vergessen. Während dieser Zeit dürfen die Probanden den Code nicht bearbeiten oder nutzen. Nach Möglichkeit bearbeiten sie auch keine Aufgaben, die der sehr Testaufgabe ähnlich sind. In der Kurzevaluation gaben die Probanden an, dass sie schon nach zwei Wochen einiges vergessen hatte. Eine Vergessenszeit von vier bis sechs Wochen sollte daher für ein valides Ergebnis ausreichend sein.

Leider wird sehr viel Arbeitskraft verschwendet, wenn alle Entwickler die gleiche Aufgabe ausführen. Das macht das Experiment sehr teuer im Verhältnis zum Umfang der dabei entstandenen Software. Eine Alternative wäre, unterschiedliche Projekte zu betrachten, für die

Erweiterungen erstellt werden. Die Hälfte darf wiederum Code Historian nutzen, die andere nicht. Man kann messen, wie die Erweiterungen im Zeitplan liegen. Es wird überprüft, ob die Erweiterungen schneller, genau nach Plan, langsamer, oder gar nicht, oder abgeschlossen werden. Wenn Code Historian den Entwickler hilft, sollte das in einer verhältnismäßig kürzeren Projektlaufzeit resultieren. Wird das Experiment auf diese Weise durchgeführt, können sich die Entwickler, ohne Zeit zu verschwenden, mit ihren ohnehin bearbeiteten Projekten beschäftigen. Für ein solches Vorgehen wird aber eine große Menge von Projekten benötigt, um ein valides Ergebnis zu erhalten. Es muss sehr genau darauf geachtet werden, ob es Störfaktoren gibt, die das Ergebnis beeinflussen. Beispielsweise könnten einzelne Entwickler(-Teams), die besonderes gut oder schlecht arbeiten, das Ergebnis unter Umständen stärker beeinflussen als Code Historian.

6 Zusammenfassung

Wartungsarbeiten machen einen großen Anteil von Softwareprojekten aus. Die Einarbeitung in den vorhandenen Quellcode nimmt dabei viel Zeit in Anspruch. Entwickler müssen sich dazu nicht nur in fremden Code einarbeiten, sondern häufig auch in eigenen alten Code, weil sie z.B. seine Funktion vergessen haben. In dieser Arbeit wurde ein Konzept entwickelt und prototypisiert, wie durch die Aufnahme und Auswertung der Implementierungsgeschichte die Einarbeitung in eigenen alten Code verbessert werden kann. Das Ergebnis der ersten Evaluation hat gezeigt, dass das gelungen ist.

Es sollte den Entwicklern beim Erinnern geholfen werden. Um Vergessen und Erinnern zu verstehen, haben wir uns mit kognitiver Psychologie beschäftigt. Aus der kognitiven Psychologie haben wir die Erkenntnis übernommen, dass Menschen sich leichter an eine Information erinnern, wenn der Kontext, in dem sie sich beim Erinnern befinden, mit dem Kontext übereinstimmt, in dem sie die Information gelernt haben. Wir haben diese Erkenntnis auf die Einarbeitung in Code übertragen. Für den Software-Entwickler ist der wichtigste Aspekt seines Kontexts der gerade bearbeitete Code und die Entwicklungsumgebung, die er benutzt. Wir haben daher angenommen, dass wenn man den Code und die Entwicklungsumgebung soweit wie möglich auf einen früheren Zeitpunkt wiederherstellt, der Entwickler sich leichter erinnert, wie und warum er zu diesem Zeitpunkt den Code verändert hat.

Es wurde ein Konzept entwickelt, wie ein Werkzeug aussehen kann, dass den Entwickler auf Basis dieser Annahme unterstützt. Dem Werkzeug gaben wir den Namen Code Historian. Während der Entwickler an einem Projekt arbeitet, protokolliert Code Historian mit, wann der Entwickler welche Änderungen am Code vornimmt. Wenn der Entwickler sich später in das Projekt wieder einarbeiten möchte, kann Code Historian ihm vorspielen, wie der Code sich verändert hat und welche Zwischenstände er auf dem Weg zur Fertigstellung angenommen hat. Für Code Historian wurde eine Prototypimplementierung hergestellt. Diese Prototypimplementierung ist als Plugin für die Entwicklungsumgebung Eclipse realisiert.

Mit einer kleinen Anzahl von Probanden wurde überprüft, ob Code Historian tatsächlich seinem Anspruch gerecht wird, den Entwickler beim Einarbeiten in seinen Quellcode zu unterstützen. Die Probanden mussten dafür eine kleine Programmieraufgabe lösen. Zwei

Wochen nach der Fertigstellung sollten sie sich in ihren Code einarbeiten. Die Probanden wurden befragt, ob sie Code Historian dabei als hilfreich empfanden. Das Ergebnis war, dass Code Historian tatsächlich bei der Einarbeitung hilft. In der Form wie es in der Prototypimplementierung realisiert war, fanden es die Probanden aber umständlich zu bedienen. Mit der Prototypimplementierung sollte jedoch nur festgestellt werden, ob der entwickelte Ansatz tatsächlich funktioniert. Eine gebrauchsfertige Software war nicht das Ziel. Die durchgeführte Evaluation basierte nur auf subjektiven Einschätzungen der Probanden. Um mit mehr Aussagekraft beurteilen zu können, ob Code Historian seinen Zweck erfüllt, müsste eine größer angelegte Studie mit mehr Probanden durchgeführt werden. Ein Konzept für eine solche Studie wurde entwickelt.

Abschließend lässt sich festhalten, dass die Nutzung von Erkenntnissen aus der kognitiven Psychologie helfen kann, die Wartung von Software zu verbessern. Die Einarbeitung in eigenen alten Code kann beschleunigt werden, wenn die Implementierungsgeschichte aufgezeichnet wird und zur späteren Einarbeitung in geeigneter Form präsentiert wird.

7 Anhang

7.1 Kurzanleitung für Code Historian

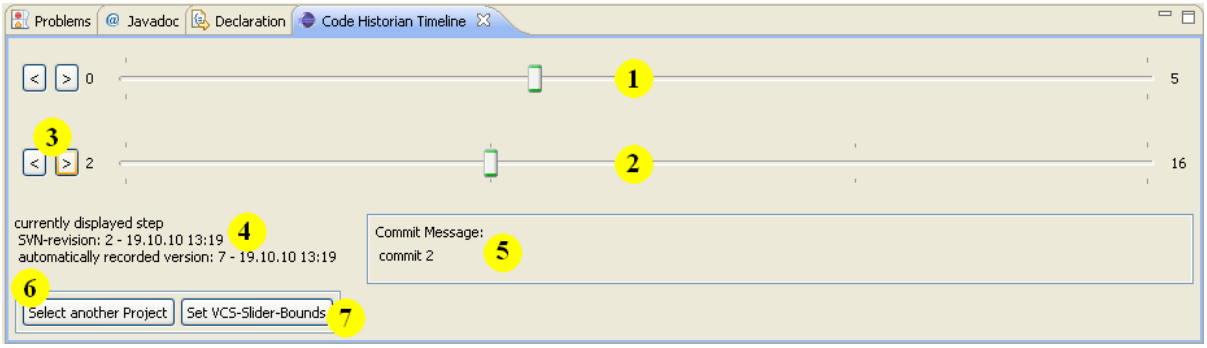
Code Historian wird gesteuert mit der View „Code Historian Timeline“. Sie lässt sich im Menü über Window/Show View/Other…/Code Historian aktivieren.

Zunächst muss aus einer Liste ein Projekt ausgewählt werden, dessen Implementierungshistorie man betrachten möchte. (Abbildung 12)

12 Projektauswahl

Wenn für das Projekt ein SVN-Repository benutzt wurde, fragt Code Historian nach den Zugangsdaten. Diese werden nicht gespeichert. Man kann Code Historian aber auch ohne die Unterstützung des SVN benutzen.

Anschließend kann Code Historian über folgende Schaltflächen gesteuert werden.



13 Benutzeroberfläche mit Zeitleisten

1. **SVN-Zeitleiste**
   Wenn der Entwickler ein SVN-Repository benutzt hat, können hiermit Versionen aus diesem Repository angewählt werden.

Wurde kein SVN benutzt, enthält diese Leiste alle automatisch aufgenommenen Zwischenstände

2. **Detail-Zeitleiste**
   Die Grenzen dieser Leiste passen sich an die mit der SVN-Zeitleiste ausgewählte Version an. Die Detail-Leiste enthält nur die automatisch aufgezeichneten Versionen in der Nähe, der mit der SVN-Leiste ausgewählten Version. Einzelne Versionen können so präziser angewählt werden.
3. **Vor- und Zurückbuttons**
   Es gibt vor und Zurück-Buttons für beide Leisten
4. In einem Felder werden **Revisionsnummer und Zeit** der ausgewählten Version angezeigt
5. Die **Commitnachricht** zur ausgewählten SVN-Revision, falls ein Repository genutzt wurde
6. Button, führt zurück zur Projektauswahl
7. Hier können die Grenzen der SVN-Zeitleiste festgelegt werden. Das ist gedacht für nur sehr große Projekte, wenn die Leiste zu viele Revisionen enthält und man diese nicht mehr präzise genug wählen kann.

7.2 Empirische Problemuntersuchung

7.2.1 Fragebogen

Dies ist der komplette Fragebogen. Er war als Webformular produziert und wurde für diesen Anhang in das Word-Format übersetzt. Inhaltlich wurde nichts verändert, aber das Layout sah im Original daher anders aus.

1. Seit wann programmierst du schon?

	< 1 Jahr	1-3 Jahre	> 3 Jahre	nicht zutreffend
Privat				
Universität/Ausbildung				
Beruflich				

2. Wie lange laufen bei dir üblicherweise Projekte?

☐ < 1 Woche ☐ < 1 Monat ☐ > 1 Monat

3. Wie lange programmierst du pro Woche?

☐ 0-5 Stunden ☐ 5-10 Stunden ☐ > 10 Stunden

4. Wie oft musst du dich in eigenen alten Code einarbeiten?

☐ Oft ☐ Manchmal ☐ Nie

5. Wie ausführlich kommentierst du üblicherweise deinen Code?

☐ Gar nicht

☐ Manchmal an wichtigen Stellen

☐ Ich schreibe JavaDoc für alle wichtigen Methoden

☐ Ausführliches JavaDoc und zusätzliche Kommentare an kritischen Stellen in den Methoden

6. Wenn du dich in eigenen alten Quellcode einarbeiten musst, welche Aspekte hast du am ehesten vergessen?

7. Woran erinnerst du dich am besten, wenn du eigenen alten Code liest?

8. Isoliert vom restlichen Code, wie gut erinnerst du dich typischerweise an die Funktion von einzelnen Methoden, Klassen und Paketen? Bsp: Was macht diese Methode und wie sieht die Rückgabe aus?

	gut	mittel	schlecht	gar nicht	nicht zutreffend
Methoden					
Klassen					
Packages					

9. Wie gut erinnerst du dich an die Struktur des Projekts bzw. den Zusammenhang der Methoden, Klassen und Pakete? Bsp: Warum brauche ich diese Methode und von welchen anderen wird sie aufgerufen?

	gut	mittel	schlecht	gar nicht	nicht zutreffend
Methoden					
Klassen					
Packages					

10. Kannst du dich erinnern

	gut	mittel	schlecht	gar nicht	nicht zutreffend
welche Patterns du benutzt hast					
und warum du sie benutzt hast?					

11. Wie gut erinnerst du dich an verwendete Sprachkonstrukte?

	gut	mittel	schlecht	gar nicht	nicht zutreffend
Was macht dieses IF?					
Warum ist dieser Catch-Block leer?					
Warum läuft die Schleife bis n-1 statt bis n?					

12. Wie gut erinnerst du dich an aufgetretene Fehler?

	gut	mittel	schlecht	gar nicht	nicht zutreffend

Compilierfehler					
Laufzeitfehler					
Logische Fehler					
Anforderungs- und Entwurfsfehler					

13. Wie gut erinnerst du dich an die Lösungen von Fehlern?

	gut	mittel	schlecht	gar nicht	nicht zutreffend
Compilierfehler					
Laufzeitfehler					
Logische Fehler					
Anforderungs- und Entwurfsfehler					

14. Hier ist noch Platz falls du Kommentare oder Anmerkungen hast, z.B. zu Fragen bei denen du „nicht zutreffend“ gewählt hast.

7.2.2 Antworten auf offene Fragen

Nachfolgend sind sämtliche Antworten auf die offenen Fragen.

Wenn du dich in eigenen alten Quellcode einarbeiten musst, welche Aspekte hast du am ehesten vergessen?

Die Verwendung und Abhängigkeiten von Programmteilen Die erneut in ein Projekt eingebunden werden sollen.
Welche Nebenbedingungen Die Verwendung einer Klasse besitzt.
Die exakte Funktionsweise des ehemals programmierten Algorithmus.

---

die Abhängigkeiten zu anderen Stellen in der Anwendung
Vergessen habe ich manchmal die Klassen Struktur. Im Zweifel muss man sie nochmal als UML nachbilden.

Strukturen

Die Gesamtzusammenhänge

In welcher Datei stand was, wo liegt die Datei

Welche Funktionalitäten in welchen Objekten steckt. Bei Threading ist es besonders schwierig.

Welche Variablennamen stehen wofür?

Wie heißen meine Funktionen?

Wofür brauche ich welche Funktionsaufrufe?

Welche Methoden es gibt und was sie für Parameter haben.

Wie ich Algorithmen aufgebaut habe und warum ich es so und nicht anders gemacht habe.

Die genaue Logik.

Ausnahmen, zu ausgetüftelte Konstrukte, dynamische Aspekte

Wie die Algorhitmen in den Methoden genau funnktionieren

---

welche funktionen es eigentlich gibt. was irgenwelche "hacks" bewirken und wieso sie nötig waren

---

Aufrufkonventionen der Unterroutinen.

Die Details der Implementierungen.

-Sinn und Zweck von manchen Funktion in Klassen (meist bei Funktionen, die nicht gerade ins Design der Klasse fallen aber für ein ganz spezielle Sache nötig wurden).
-Verwendung von nicht ausreichend dokumentierten Funktionsparametern.

Architektur des Low-Level-Zeugs (Anbindung/Layout Datenbanken usw)

Wie "small numbers" zu Stande kamen; warum man bei geometrischen Operationen nun 2 abziehen musste, damit die Logik stimmt etc.

Bei kleineren Projekten (bezüglich Zeit und/oder Codelänge) sind es wohl eher Details, die ich schnell wieder vergesse. Dazu gehören z.B. mathematische Formeln, die man sich entweder vorher mühsam abgeleitet hat, oder die man erst recherchieren musste, sich dann aber vielleicht auch nicht die Mühe gemacht hat sie vollständig verstehen zu wollen (-> hauptsache, sie tun es).
Bei größeren Projekten kann es dann auch der Zusammenhang oder die Zusammenarbeit zwischen einzelnen Modulen/Objekten sein. Ist nur im Code dokumentiert und sind Variablen und Funktionen nicht optimal benannt, kann es dann eher Probleme geben. Wieviel Zeit ich für die Namensgebung verwende (die ich für sehr wichtig halte) hängt davon ab, wie wahrscheinlich es für mich ist, dass ich den Code einmal warten muss oder wiederverwenden werde.

Woran erinnerst du dich am besten, wenn du eigenen alten Code liest?

Gut erinnere ich mich meistens, wenn der Code meine eigenen Standards (Dinge, die ich immer so mache, oder Variablen, die ich immer nach dem gleichen Schema benenne geben mir beim Überfliegen des Programms sofort einen Hinweis, was ich da gemacht habe, so dass ich mich auch schnell an den Rest erinnere) oder allgemeine Standards (Coding-Conventions, z.B. wieder allgemeine oder spezielle Regeln der Namensgebung) einhält.
Klingt bescheuert: Am besten erinnere ich mich, wenn ich Programmteile mit anderen Dingen assoziere, besonders wenn dabei Emotionen im Spiel sind. Beispiel: Lustig gemeinte Kommentare oder Variablennamen (sozusagen als Eselsbrücke, oder zur besseren Vernetzung im Gehirn. Ohne jede Freude bei der Sache erinnere ich mich später sehr viel schwerer). Das geht natürlich nur in privaten Projekten, da andere Programmierer sich durch so etwas wohl ziemlich genervt fühlen würden. Bei der Teamentwicklung klappt das jedoch trotzdem noch ein bisschen: Z.B. wurden hier in einem alten Projekt, mit dem ich hier in der Arbeit beschäftigt bin, die Namen vor vielen Jahren nach ungarischer Notation (teilweise weil die Strings nicht zu lang werden durften), es gibt sicher tausende von Klassen, das Programm ist unüberschaubar groß, und in keinster Weise ausreichend dokumentiert (genauer: es ist NUR im Code dokumentiert, und das sehr spärlich...wenn ich etwas nicht weiß, kann ich nichts nachlesen, sondern muss jedesmal irgendwen fragen :/). Einer meiner Kollegen erinnert sich nun beispielsweise daran, welche Masken an welchen Aufgaben beteiligt sind, in dem er die Namen der Dateien und Klassen verknüpft: - z.B. nach dem Klang (die Namen für diesen Dialog klingen alle russisch, das hilft ihm wohl sich zu erinnern; bei der Datei "SplitVert.cpp" denkt er an das "Splitpferd",)
Total sinnlos also, aber der emotionale Bezug zu den Dingen hilft sehr dabei, sich später zu erinnern...

An die direkte Funktion.

Idee hinter dem Code?
Was sollte der Code theoretisch machen?

Ablauf einzelner Funktionen

Warum ich den Code geschrieben habe.
Eingesetzte Tricks in den Algorithmen/dem Stil.

Gute Ideen

Optimierungstricks bleiben mir im Gedächtnis.

"Schwierige" Stellen, die bei der ursprünglichen Programmierung viele Probleme bereitet haben.
Meistens an etwaige Probleme die damals aufgetreten sind - diese Stellen beleiben mir besonders (gut) im Gedächtnis.
An die gewünschte Funktionalität
An wiederkehrenden Strukturen, von denen ich weiß, dass ich sie öfter einsetze und dass hoffentlich auf eine immer ähnliche Art und Weise. An der Wahl der Bezeichner und der Strukturierung von Programmen durch z.B. Leerzeilen, da ich mir meist was dab
An die generelle Struktur (Packages, Klassen, Konzept...)
Der grobe Ablauf
grober funktionsablauf von methoden wenn diese anständig benannt sind
Sinn der Funktionen
Wenn ich ihn lesen, dann an alles, solange er nicht zu verschachtelt ist. Natürlich nur mit 5-10 Minuten einlesen

7.3 Evaluation

7.3.1 Aufgabe

Das Code Historian Plugin wird installiert indem die .jar Datei in den Plugin-Ordner von Eclipse kopiert wird. Nach einem Neustart von Eclipse ist das Plugin geladen. Überprüfen sie das bitte kurz. Es sollte im Menü unter Window/Show View/Other den Ordner „Code Historian“ mit der View „Code Historian Timeline“ geben.
Bei Problemen mit der Installation schreiben sie diese bitte an philipp.kleybolte@gmx.de.

Während die Aufgabe angefertigt wird, wird Code Historian im Hintergrund aufzeichnen, was in welcher Reihenfolge programmiert wird. Code Historian tut das für jedes Projekt. Wenn sie das nicht wollen, sollten sie nach Durchführung der Aufgabe die .jar Datei wieder entfernen.

Nach ca. zwei Wochen soll Code Historian benutzt werden, um sich erneut in den Code einzulesen. Dazu wird es noch einen gesonderten kleinen Fragebogen geben.
Dazu wäre es schön, wenn sie sich nach der Bearbeitung bei Philipp Kleybolte melden würden.

Aufgaben:

Die Aufgabe ist ein Standard-Problem für das sich Lösungen googlen lassen. Das ist nicht Sinn der Aufgabe. Es soll versucht werden selbst einen Algorithmus zu finden.
Es müssen nicht zwingend alle drei Aufgaben erfüllt werden, wenn ihre Zeit es nicht zulässt. Damit es ein funktionierendes Programm gibt, in das man sich einarbeiten kann, sollte aber zumindest die erste Aufgabe erfüllt werden.

1) Lösen sie mit einem Java-Programm das n-Damen-Problem. Es müssen auf einem Schachfeld mit n x n Feldern n Damen platziert werden, ohne dass diese sich gegenseitig schlagen können. Gemeint sind Damen wie beim normalen Schachspiel. Das heißt, sie laufen vertikal, horizontal und diagonal beliebig weit.

2) Entwickeln sie eine Variante, die eine Lösung findet und eine Variante die alle Lösungen findet. Verwenden sie zum Testen nur kleine n. Die Laufzeit wird für große n sehr schnell ansteigen.

3) Erzeugen sie eine GUI, die eine Lösung darstellt.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14
eindeutig	1	0	0	1	2	1	6	12	46	92	341	1.787	9.233	45.752
insgesamt	1	0	0	2	10	4	40	92	352	724	2.680	14.200	73.712	365.596

Anzahl an Lösungen:

7.3.2 Fragebogen

Benutze bitte Code Historian, um dich in deinen Code zum n-Damen-Problem wieder einzuarbeiten.
Ich habe bei den Fragen die Option „Weiß nicht“ weg gelassen. Falls du eine Frage nicht beantworten kannst oder willst, lass sie bitte aus.

1)
Wie viel Zeit ist ungefähr vergangen zwischen dem Anfertigen der Programmieraufgabe und dem erneuten Einarbeiten?

2)
Wieviel vom Code hast du vergessen?

viel	mittel	wenig	gar nichts

3)
Hat Code Historian dir geholfen, dich zu erinnern?

ja	nein

4)
Gibt es etwas bestimmtes, woran du dich durch Code Historian erinnert hast?

5)
Hast du das Gefühl, dass dir Code Historian die Einarbeitung beschleunigt hat?

Code Historian hat meine Einarbeitung…

stark beschleunigt	beschleunigt	nicht verändert	verlangsamt	stark verlangsamt

6)
Denkst du, dass Code Historian dir hätte geholfen, dich zu erinnern, wenn das Projekt in das du dich eingearbeitet hast größer gewesen wäre und länger zurück läge?

ja	nein

7)
Denkst du, dass Code Historian die Einarbeitung beschleunigt hätte, wenn das Projekt in das du dich eingearbeitet hast größer gewesen wäre und länger zurück läge.
Code Historian hätte meine Einarbeitung…

stark beschleunigt	beschleunigt	nicht verändert	verlangsamt	stark verlangsamt

8)
Was denkst du muss an Code Historian verändert werden, damit Entwickler (mehr) bei der Einarbeitung hilft?

Ich finde es im großen und ganzen schon ganz gut	
Die Implementierungsgeschichte wiederzugeben bringt grundsätzlich nichts	
Steuerung der Wiedergabe / GUI	
Versionsschritte müssen größer	
Versionsschritte müssen kleiner	
Ich hätte mir noch ein bestimmtes Feature gewünscht	

9)
Versuch bitte möglichst konkrete Verbesserungsvorschläge für Code Historian zu machen. Z.B. Wie würdest du die GUI verändern, oder welche Funktion sollte zusätzlich eingebaut werden.

Literaturverzeichnis

[1] Michael L. Nelson - *A Survey of Reverse Engineering and Program Comprehension*, April 19, 1996 released as web-only

[2] Cornelissen - *Evaluating Dynamic Analysis Techniques for Program Comprehension*, Dissertation an der Delft University of Technology, 2009

[3] Michael G. Wessel - *Kognitive Psychologie*, 1994, UTB für Wissenschaft

[4] John R. Anderson - *Kognitive Psychologie*, 2001, Spektrum Verlag

[5] Gerhard Strube – *Assoziation der Prozess des Erinnerns und die Struktur des Gedächtnisses*, Springer Verlag, 1. Auflage

[6] Mc Keithen, Reitman - *Knowledge Organization and Skill Differences in Computer Programmers*, Cognitive Psychology, 307-325 (1981)

[7] https://www.soscisurvey.de/ (24.11.10) Entstanden aus einem Forschungsprojekt an der Universität München

[8] C. Wohlin et. al. - *Experimentation in Software Engineering*, Kluwer Academic Publishers, 1. Auflage, 2000

[9] http://www.r-project.org/ (17.11.2010)

[10] http://svnkit.com/ (17.11.2010)

[11] Torben Wichers, ParserSuite, 2010, http://www.psue.uni-hannover.de/ParserSuite/

[12] http://code.google.com/p/gource/ (17.11.2010)

[13] Mik Kersten - *Focusing knowledge work with task context*, Thesis for Doctor of Philosophy at the University of British Columbia 1999

[14] L. Hattori, M. Lungu, M. Lanza – *Replaying Past Changes in Multi-developer Projects*, IWPSE-EVOL ’10, September 20-21, 2010 Antwerp, Belgium

[15] L. Hattori - *Enhancing Collaboration of Multi-developer Projects with Synchronous Changes*, ICSE ’10, May 2-8 2010, Cape Town, South Africa

[16] C. Parnin, R. DeLine – *Evaluating Cues for Resuming Interrupted Programming Tasks*, CHI 2010, April 1015, 2010, Atlanta, Georgia, USA