HTW des Saarlandes

Sommersemester 2012

Ausarbeitung Softwareentwicklungsprozesse

Testgetriebene Wiederverwendung

Dozent: Prof. Dr. Helmut Folz

Gilles Baatz

Matr.Nr.: 3536491

und

Daniel Meiers

Matr.Nr.: 3538990

11. September 2012

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Software entwickelnde Firmen stehen vor der stetigen Herausforderung immer komplexere Software in besser Qualität, in immer kürzerer Zeit zu niedrigeren Kosten zu produzieren. Um diese Herausforderung meistern zu können, muss eine Vielzahl von Techniken und Methoden eingesetzt werden um den Entwicklungsprozess systematisch, planvoll, effizient und damit letztendlich erfolgreich zu gestalten.

Vor allem die Wiederverwendung bereits existierender Software ist ein wesentliches Mittel um dies zu erreichen. Obwohl die Vorteile von Wiederverwendung leicht einzusehen und nicht von der Hand zu weisen sind, stellt sich die konkrete Wiederverwendung von Software oft problematisch dar. Diese Hürden versucht die testgetriebene Wiederverwendung zu nehmen, indem Prinzipien aus der agilen Softwareentwicklung genutzt werden um die Wiederverwendung automatisiert in den Entwicklungsprozess einzubinden. Diese Idee unterscheidet sich erheblich von bereits bekannten Möglichkeiten der Wiederverwendung und ist ein aktuelles Forschungsthema.

Diese Ausarbeitung hat zum Ziel, die Prinzipien der testgetriebenen Wiederverwendung zu erläutern sowie den aktuellen Forschungsstand anhand bereits existierender Tools aufzuzeigen.

2 Wiederverwendung von Software

Bevor wir jedoch genauer auf die testgetriebene Wiederverwendung eingehen, möchten wir zunächst das Thema Wiederverwendung von Software allgemein betrachten.

2.1 Warum Wiederverwendung

Wie bereits eingangs erwähnt stehen Softwarefirmen vor einem immer stärkeren Konkurrenz-, Zeit- und Kostendruck. Die Anforderung bessere Software in kürzerer Zeit zu geringeren Kosten zu erstellen kann nur mithilfe verschiedenster Methoden u.a. der gezielten Wiederverwendung von Software realisiert werden.

Durch Wiederverwendung bereits entwickelter Software kann die Entwicklungszeit eines neuen System signifikant gemindert werden. Neben der reinen Entwicklungszeit wird somit auch die Zeit für Test und evtl. Validierung gespart. Die Wiederverwendung erhöht also die Produktivität. Die eingesparte Zeit wirkt sich natürlich auch positiv auf die Entwicklungskosten eines Projektes aus.

Des weiteren kann davon ausgegangen werden, das bereits exisitierende Softwarekomponenten eine höhere Zuverlässigkeit besitzen, da Fehler die evtl. im Entwurf oder der Implementierung gemacht wurden, durch den bereits stattgefundenen Einsatz aufgefallen und behoben sind. Dadurch wird implizit die Qualität der neu entwickelten Software erhöht.

Gerade die höhere Zuverlässigkeit und Qualität der Software, sowie die Einsparung von Zeit und Kosten führen zu einer besseren Planbarkeit für das gesamte Projekt.

Ein weiterer Vorteil ist, dass die Wartbarkeit von Software wesentlich erleichtert wird da evtl. auftretende Fehler in der wiederverwendeten Komponente nur einmal und nicht n-fach behoben werden müssen.

Die Vorteile der Wiederverwendung von Software sind also offensichtlich und bieten ein gute Möglichkeit die Softwareentwicklung produktiver, besser und billiger zu machen.

2.2 Was bedeutet Wiederverwendung?

Da der Begriff der Wiederverwendung bisher recht allgemein und ohne genaue Definition benutzt wurde, werden wir den Begriff zunächst anhand einschlägiger Literatur definieren.

2.2.1 Begriffsdefinition

In der Literatur finden sich vielzählige und recht unterschiedliche Definitionen für den Begriff der Software-Wiederverwendung.

So beschreibt [6] den Begriff sehr allgemein wie folgt:

Software Reuse, [6] „ Software reuse is the process of creating software systems from existing software rather than building software systems from scratch.“

Ähnlich, allerdings aus eher technischer Sicht, beschreibt [11] den Begriff Software Reuse als:

Software Reuse, [11] „ Software reuse is the use of existing software components to construct a new system.“

Laut [3] findet sich in [2] eine noch umfassendere Definition des Begriffs. Hier wird die Wiederverwendung nicht rein auf die Wiederverwendung von Code und Codefragmenten reduziert:

Reusable Software Engineering, [3] „Reusable Software Engineering beschäftigt sich mit der Entwicklung von Software unter Wiederverwendung aller während der Software-Entwicklung generierten Informationen.“

2.2.2 Abstraktionsebenen

Wie obige Definitionen zeigen kann unter Wiederverwendung von Software weit mehr als die reine Wiederverwendung von Code verstanden werden. Dies liegt vor allem daran, dass Wiederverwendung auf verschiedenen Abstraktionsebenen durchgeführt werden kann, die wiederum neue Aspekte und Probleme mit sich bringen. Folgende Auflistung stellt eine Übersicht über die Verschiedenen Aspekte und Arten der Wiederverwendung von Software dar und ist angelehnt an [vgl. 11]. Eine genaue Diskussion der einzelnen Punkte würde den Rahmen dieser Ausarbeitung sprengen, sie gibt jedoch eine guten Einblick in die Vielfältigkeit und Komplexität der Wiederverwendung von Software.

Substanz, bzw. die Abstraktionsebene	Bereich	Modus	Absicht	Produkt
Ideen, Konzepte	Vertikal, d.h. gleicher Anwendungsbereich, System-Familien	geplant, systematisch	black-box, d.h. ohne Modifizierung	Source Code
Artefakte (z.B. Klassen), Komponenten	Horizontal, d.h. allgemein, über verschiedene Anwendungsbereiche hinweg	ad-hoc, opportunistisch	white-box, d.h. mit Modifizierung	Design
Prozeduren, Skills				Spezifikation Objekte Text Architekturen

Abbildung 2.1: Übersicht über Aspekte von Wiederverwendung

Die testgetriebene Wiederverwendung fokussiert die Wiederverwendung von Code-Elementen. Daher lehnen wir uns an die Definition von [11] und [6] an, wenn im Folgenden der Begriff Wiederverwendung von Software gebraucht wird.

2.3 Probleme beim Wiederverwenden

Die Wiederverwendung von Software ist seit Beginn der Softwareentwicklung ein zentrales Thema. Der Begriff wurde erstmals von McIlroy 1968 geprägt. Er beschreibt in [8] die damals katastrophale Lage der Softwareentwicklung und die Wiederverwendung als möglichen Ausweg aus der Softwarekrise.

Seitdem erhoffte man sich mit neuen Paradigmen der Softwareentwicklung stets auch die Wiederverwendung von Softwarekomponenten zu erleichtern und zu verbessern. Objektorientierte Programmierung und service-orientierte Architekturen sind nur wenige von vielen Beispielen dafür [vgl. 4]

Und auch heutzutage gestaltet sich die Wiederverwendung von Software recht problematisch und ist weit entfernt von einem praxistauglichen alltäglichem Einsatz. Doch warum lässt sich die recht einfache Idee der Wiederverwendung mit ihren vielen Vorteilen nur so schwer in die Praxis umsetzen?

Ein Grund dafür ist die rechtliche Sicht. Selbst wenn eine bereits existierende, funktional passende Softwarekomponente gefunden werden sollte, darf sie aufgrund lizentrechtlicher Vorgaben nicht immer verwendet werden. So erfordern manche Open Source Lizenzen, dass Produkte die mit Open Source Produkten entwickelt wurden bzw. diese enthalten, ebenfalls Open Source zur Verfügung gestellt werden müssen. Dies ist bei kommerziellen Produkten durchaus problematisch. Es müssen daher auch organisatorische Maßnahmen zur Wiederverwendung getroffen werden, denn bei der aktuellen Anzahl und Komplexität der Lizenzmodelle ist es nicht möglich, diese Entscheidung dem Entwickler zu überlassen, da eine Fehlentscheidung gravierende Folgen haben kann.

Das hauptsächliche und gravierendere Problem jedoch lässt sich recht einfach und anschaulich formulieren.

Software wiederverwenden heißt Software (wieder)finden.

Aufgrund der rasant gestiegenen Open Source Entwicklung ist die Anzahl an möglichen Wiederverwendungskandidaten enorm gestiegen. War zu den Anfängen der

Wiederverwendung von Software das Problem dass es zu wenige mögliche Kandidaten gab, so hat sich dies durch die Open Source Entwicklung ins Gegenteil gewendet. Jedoch ist dieser Umstand genauso problematisch. Die enorme Größe von Software-Repositories kann von einem Entwickler kaum noch Inhaltlich überblickt werden. Allein das aktuelle JDK umfasst ca. 3800 Basisklassen. Zieht man weitere bekannte Frameworks in die Rechnung mit ein kommt allein im Java-Umfeld auf 10.000 Klassen.Selbst Firmen-intern gestaltet sich das Wiederfinden von Software Komponenten oft schwierig. Oftmals geht entsprechendes Wissen über die Existenz entsprechender Komponenten mit dem Ausscheiden von Mitarbeiten verloren, oder wird nach Beendigung des Projekt schlicht vergessen. [vgl. 4]

Auch wenn der Entwickler bei der Suche nach Software, wie im nächsten Kapitel beschrieben, durchaus mit technischen Mitteln und Werkzeugen unterstützt werden kann, bleibt die Suche nach möglichen Wiederverwendungskandidaten oft umständlich und aufwendig. Dies liegt in den allermeisten Fällen daran, dass die Suche oftmals nicht in die gewohnte Entwicklungsumgebung integriert, geschweige denn automatisiert ist. Laut [4] erfordert dies zum „zeit- und konzentrationsraubenden Wechsel“ zwischen Entwicklungsumgebung und Suchwerkzeug. Zum anderen wird in [4] behauptet, dass „Entwickler sich häufig nicht bewusst sind, dass eine Suche nach wiederverwendbarem Material erfolgversprechend sein könnte“

3 Software suchen

Die für Menschen nicht mehr zu handelnde Größe von Softwarerepositories führt zu der Notwendigkeit das die Suche nach möglichen Kandidaten maschinell erfolgen muss.

Dies ist jedoch nicht als Nachteil zu sehen, da durch die maschinelle Unterstützung es für den Entwickler wesentlich leichter wird nach Komponenten beliebiger Funktionalität zu suchen.

Daher verwundert es nicht, dass die Idee einer maschinellen Suche nach Software nicht neu ist, sondern bereits seit über 25 Jahren exisitert. [vgl Quelle] Doch lange Zeit wurden keine nennenswerten Fortschritte in der Entwicklung von Suchmaschinen gemacht, da aufgrund der geringen verfügbaren Menge an Software die bis dato entwicklten Systeme vollkommen ausreichend waren. Erst mit der Open Source Bewegung wurde deutlich, dass weiterer Forschungsbedarf besteht. Die immer größere Menge an Software zeigte, dass bisherige Suchmaschinen zu schlechte Ergebnisse lieferten und für die Suche viel zu lange brauchten. [Quelle]. Auch heute noch besteht Forschungsbedarf um die Suchergebnisse qualitativ besser zu machen.

Die oftmals schlechte Qualität von Suchergebnissen liegt vor allem darin begründet, dass sich die Suche nach Software wesentlich von anderen Suchproblemen wie zb. der Internetsuche unterscheidet.

Es ist leicht einzusehen, dass eine rein textbasierte Suche wie sie in Internetsuchmaschinen zum Einsatz kommt bei der Suche nach Software nur mäßige Ergebnisse liefert. Dieses Phänomen wird in [4] wie folgt beschrieben: „So mag zwar der Suchbegriff “Matrix” durchaus Quelltexte liefern, die der Berechnung von Matrizen-Rechen-operationen dienen.Es ist allerdings beispielsweise bei Weitem

nicht offensichtlich, welche Funktionalität im Einzelnen unterstützt wird, oder ob nicht unter Umständen nur der Begriff “Matrix” zufällig im Quellcode oder der Beschreibung eines Suchergebnisses auftaucht.“

Wie obiges Beispiel zeigt ist die Suche nach Software oder genauer gesagt das Formulieren einer guten Suchanfrage eine nicht triviale und komplexe Angelegneheit. Hinzukommt, dass eine zu genaue Suchanfrage das Suchergebnis zu stark einschränkt und somit unter Umständen evtl. passende oder vielversprechende Kandidaten nicht weiter berücksichtigt werden.

3.1 Suchtechniken

Im Folgenden werden verschiedene Suchtechniken vorgestellt, die die Suchmöglichkeiten der Softwaresuche verbessern können. Eine sehr ausführliche Diskussion der verschiedenen Suchtechniken findet sich in [9].

3.1.1 Textbasierte Suche

Die textbasierte Suche entspricht der allseits bekannten Such im Internet. Sie ist natürlich auch auf Software anwendbar, da Software im allgemeinen aus einer Menge von Dokumenten, dem Quelltext, besteht. Die Suchanfrage besteht aus einem oder mehreren Begriffen die in den Quelltexten gesucht werden.

Wie bereits erwähnt hat diese Methode entscheidende Nachteile, was nicht zu letzt daran liegt, dass Software neben “normalen” Dokumenten (wie zb. Html) weitere Eigenschaften besitzt. Dazu gehören zb. Ausführbarkeit, prezise Semantik usw.

Deshalb ist der Einsatz weiterführender Technologien erforderlich.

3.1.2 Beschreibende Suchverfahren

Beschreibende Suchverfahren sind den textbasierten Suchverfahren sehr ähnlich. Der einzige Unterschied besteht in der internen Repräsentation der Kandidaten. Ihnen wird eine (evtl. auch strukturierte) Liste von Metadaten zugeordnet. Die Metadaten bestehen dabei in der Regel aus einer Menge von Keywords.

Laut [9] ist die Einfachheit der zugrundeliegenden Idee der Hauptgrund warum beschreibende Suchverfahren sehr oft in Softwaresuchmaschinen genutzt werden.

Ein Nachteil dieser Technik ist jedoch das die Metadaten nur schwer bis gar nicht automatisch zugeordnet werden können, was eine manuelle Erfassung der Metadaten erfordert. Diese manuelle Einteilung ist problematisch da sie vom Wissen und der Einschätzung der erfassenden Person abhängig ist. Diese muss jedoch nicht immer mit der Einteilung der Mehrheit überein stimmen, geschweige denn korrekt sein. Stimmen die Vorstellung der Metadaten der erfassenden Person und der suchenden Person nicht überein, wird die Suche nicht erfolgreich verlaufen. Dieses Problem sollte nicht unterschätzt werden.

Dass dieses Problem nicht zu unterschätzen ist, wird anhand einer aktuellen Werbung eines Sportsenders für ein Fußballpaket deutlich und humorvoll aufgegriffen: „Woran denkst du bei, Schwalben, Spielzügen und Rudelbildung [...] und woran denkst du bei Ball?“.[13]

Als mögliches Beispiel für beschreibende Suchverfahren nennt [4] die „[...]in der Forschung momentan mit enormen Aufwand vorangetriebene Verwendung von Ontologien im Bereich der *Semantic Web Services*, fällt beispielsweise in diese Kategorie.“

3.1.3 Bedeutungsbasierte Suchverfahren

Die bedeutungsbasierten Verfahren konzentrieren sich auf die Semantik der Software. Sie können unterschieden werden in operationale Verfahren und formale Verfahren.

Die operationalen Verfahren machen sich die besondere Eigenschaft von Software zu nutze, dass sie ausführbar ist. Die Idee geht auf [10] zurück. Sie fanden heraus, dass eine Softwarekomponente sich eindeutig durch ihr Verhalten auf einer zufälligen Menge von Eingabedaten beschreiben lässt. Sie ordnen der ausführbaren Software eine Menge von typisierten Eingabevariablen sowie einer Wahrscheinlichkeitsverteilung des jeweiligen Eingabebereichs zu. Ein passender Kandidat wird gefunden indem anhand der Warscheinlichkeitsverteilung Eingabeparamter gewählt werden, und mit diesen die Software ausgeführt wird. Stimmt das Ergebniss mit dem gewünschten Resultat überein, ist ein guter Kandidat gefunden. Durch Wiederholung des Experimentes kann die Wahrscheinlichkeit das falsche Ergebnisse zurückgeliefert werden, enorm gesenkt werden.

Eine genauere Betrachtung der formalen Methoden würde den Rahmen dieser Ausarbeitung sprengen, zusammenfassend kann jedoch gesagt werden, dass „die operationalen Verfahren, die Wiederverwendungskandidaten kurzerhand mit beispielhaften Eingabedaten (Samples)ausführen, den auf formalen Beschreibungstechniken basierenden (und entsprechend komplizierten) Ansätzen zumindest bisher überlegen zu sein.“[4] scheinen.

3.1.4 Strukturbasierte Suchverfahren

Strukturbasierte Methoden unterscheiden sich wesentlich von allen bisher vorgstellten Techniken. Sie abstrahieren von der funktionellen Eigenschaften, also dem was die Komponente leisten soll. Statt dessen untersuchen sie mögliche Kandidaten anhand ihrer Struktur.

Die zugrundeliegende Idee „[...]is the assumption that whenever two comonents have similiar functional properties, they also have similiar structures[...]“.[9] [page 52]

Dass eine solches Vorgehen notwendig sein kann wird in [9] anhand von zwei Beispielen verdeutlicht. Man stelle sich 2 verschiedene Implementierungen eines Sortieralgorithmus vor. Eine Implementierung benutzt QuickSort, die andere SelectionSort als zugrundeliegenden Sortiealgorithmus. Auch wenn beide Programme die gleiche Funktionalität bereitstellen so ist deren Struktur doch deutlich

unterschiedlich, da Quicksort ein rekursiver Algorithmus ist und grundlegend anders definiert.

Auch umgekehrt lässt sich ein Beispiel konstruieren. Man stelle sich ein Programm S sowie ein Programm M vor. S addiert die Elemente einer Liste, M multipliziert diese. Beide Programme haben eine unterschiedliche Funtkionalität, aufgrund ihrer ähnlichen Funktionsweise jedoch auch eine ähnliche Struktur. Bei den bisher vorgestellten Techniken würde nach einer Suche nach S, P nicht als Kandidat geliefert werden, aufgrund ihrer ungleichen Funktionalität. Jedoch kann P mit wenigen Änderungen auf die gewünschte Funtkionalität gebracht werden und ist deshalb ein interessanter Kandidat.

Die strukturbasierte Suche findet jedoch kaum Anwendung in der Praxis. Falls doch werden die besten Ergebnisse erzielt falls lediglich die Operationssignaturen verwendet werden. Die Verwendung der internen Code Struktur ist problematisch, da sich nur schwer genaue Suchanfragen formulieren lassen. [vgl. 4, 9]

3.2 Beispiele

Die folgende Tabelle gibt eine Übersicht über aktuelle Softwaresuchmaschinen und gibt Aufschluss über die jeweils unterstützen Sprachen, sowie die verwendetet Suchtechniken und ist entommen aus [4].

Besonderes Augenmerk ist auf die Merobase Suchmaschine zu legen, da sie mit dem später vorgestellten Tool CodeConjurer zusammenarbeitet.

Name	Ursprung	Jahr	Größe	Sprachen	Suchverfahren	Bemerkungen
UDDI Business Registry	kommerziell	2000	< 500 Services	WSDL	textuell auf Metadaten	2006 mangels Einträgen abgeschaltet
SPARS-J	wissenschaftlich	2004	~ 150.000 Artefakte	Java	textuell	adaptierte erstmals Googles Page-Rank-Algorithmus für Software
koders.com	kommerziell	2004	~ 3 GLOC	> 30	textuell, auch auf Klassen- & Methodennamen	erste kommerzielle Software-Suchmaschine
Google Codesearch	kommerziell	2006	mehrere Millionen	> 50	textuell, auch mit regulären Ausdrücken	
Krugle Opensearch	kommerziell	2006	~ 800.000 Artefakte	> 20	textuell, auch auf Klassen- & Methodennamen	
merobase.com	wissenschaftlich	2006	~ 10 M Artefakte	Java, C++, C#, WSDL	textuell, auch auf Klassen- & Methoden-namen und Metadaten, auf Klassen-Schnitt-stellen & testgetrieben	unterstützt durch das Eclipse-Plug-In Code Conjurer
Sourcerer	wissenschaftlich	2007	~ 3 M Artefakte	Java	namensbasiert, strukturbasiert	unterstützt durch das Eclipse-Plug-In Code Genie

Abbildung 3.1: Übersicht der wichtigsten Software-Suchmaschinen der letzten Jahre [4]

4 Testgetriebene Wiederverwendung

4.1 Idee der testgetriebenen Wiederverwendung

Um Komponenten wiederzuverwerten, muss der Benutzer diese über Suchmaschinen suchen, wobei die gefundenen Komponenten manuell überprüft und getestet werden müssen, ehe diese in einem Projekt verwendet werden können. Die Suchmaschinen liefern allerdings viele Komponenten, die den Anforderungen nicht entsprechen, sogenannte "False Positives". [vgl. 4, S. 10] Diese False Positives kommen unter anderem zustande, da die Menge an Open-Source Code beachtlich ist und dieser Code durchaus wiederverwendet werden kann. Allerdings arbeiten die Suchmaschinen nicht gut genug um sämtliche falschen Ergebnisse herauszufiltern. Dies macht das manuelle heraus filtern sehr zeitaufwendig, so dass der Zeitgewinn bei der Wiederverwendung gering oder sogar negativ ausfällt und es schneller wäre die entsprechende Komponente neu zu schreiben. Diese Ungewissheit ob der Zeit gewonnen wird oder nicht, ist ein Argument gegen die Wiederverwendung, da durchaus viel Zeit verschwendet werden kann.

Hier kommt nun die testgetriebene Wiederverwendung ins Spiel. Diese versucht die Anzahl der False Positives zu verringern, indem die gefundenen Komponenten automatisch geprüft und getestet werden. Dem Benutzer werden nur die vielversprechenden Komponenten angezeigt, also jene die den Test überstanden haben. Somit wird dem Benutzer Arbeit abgenommen und das Wiederverwenden, aus Sicht des Benutzers, ist nicht mehr so zeitaufwändig. Die Idee dass die Komponenten automatisch getestet werden, ist dank Unit-Tests und den agilen Entwicklungsmethoden nicht so abwegig. Im konkreten Fall bedeutet dies, dass der

Benutzer einen Unit-Test für eine Komponente schreibt, die noch nicht entwickelt wurde. Dies ist eine Herangehensweise, die besonders bei agilen Entwicklungsmethoden häufig anzutreffen ist. Da der Test den Aufbau der gesuchten Komponente beschreibt, kann eine Suchmaschine diesen als Suchanfrage verwenden, um nach ähnlichen Komponenten zu suchen. Danach können die gefundenen Komponenten mit jenem Test getestet werden und der Benutzer erhält als Suchergebnis nur noch die Komponenten, die den Test erfolgreich bestanden haben. Die gefundenen Ergebnisse passen per Definition ins Projekt des Benutzers, da sie den von ihm erstellten Test bestanden haben.

4.2 Vergleich und Stand von Tools

Die in [4, S. 11] erwähnten Werkzeuge, die eine testgetriebene Wiederverwendung ermöglichen, sind:

- Code Genie
- S6
- Code Conjurer

Diese 3 Werkzeuge werden im Folgenden, vorgestellt wobei auf Code Genie und S6 nur kurz eingegangen wird, und Code Conjurer ausführlicher betrachtet wird.

4.2.1 Code Genie

Die in [7, S. 2] angegebene URL zu Code Genie existiert nicht mehr und es wurde auch sonst keine Möglichkeit gefunden um Code Genie herunterzuladen und zu testen. Laut [4, S. 11] ist Code Genie ein Eclipse-Plug-In das die Suchmaschine Sourcerer benutzt. Allerdings verlangtes ein “manuelles Integrieren und/oder Testen der Kandidaten in einer Kopie des eigenen Eclipse-Projektes.“

4.2.2 S6

S6 ist ein Web-Interface das über [12] erreichbar ist. Das Interface hat zwei große Nachteile:

- Es können nur einfache Testfälle angegeben werden, die die Maske des Interfaces zulässt. Kompliziertere Unit-Tests sind somit nicht realisierbar, da das Hochladen von Dateien nicht möglich ist. Siehe in der Hilfe von: [12]
- Weiterhin fehlt eine Anbindung an eine Entwicklungsumgebung, wie etwa Eclipse, so dass das Arbeiten mit S6 recht umständlich ist. [12, S. 10]

4.3 Code Conjurer

Code Conjurer ist ein Eclipse Plug-In das testgetriebene Wiederverwendung für Java ermöglicht. Beim Ausprobieren von Code Conjurer ist uns aufgefallen, dass das Werkzeug das wir benutzten stark von dem Abweicht was im Artikel beschrieben wird. Dies liegt daran dass im Artikel Code Conjurer 1 beschrieben wird. Allerdings wurde Code Conjurer zwischenzeitlich von Grund auf neu geschrieben [1], so dass nur noch Code Conjurer 2 angeboten wird und die alte Version nicht mehr verfügbar ist. Das Problem ist allerdings, dass in der neuen Version noch einige Features fehlen. Im folgenden wird Code Conjurer 1 beschrieben und danach folgen unsere Erfahrungen mit Code Conjurer 2, sowie eine Auflistung mit den Fähigkeiten, die wir vermissen.

4.3.1 Beschreibung von Code Conjurer 1

Dieser Abschnitt beschreibt Code Conjurer 1, da diese Version des Werkzeugs, nicht mehr verfügbar war, wurde die Beschreibung aus [5, S. 48-51]übernommen.

Prinzipielle Funktionsweise von Code Conjurer 1

- In einem ersten Schritt schreibt der Benutzer einen JUnit-Test für die ungeschriebene Komponente, die er benötigt.
- In einem Status Feld, sieht der Benutzer eine Übersicht über die gefundenen Ergebnisse. In einem weiteren Fenster kann sich der Benutzer die Details zu der Suche explizit anzeigen lassen.
- Code Conjurer wird automatisch aktiv wenn der Benutzer den Test ausführt.
- Das Werkzeug sendet eine testgetriebene Suchanfrage an die Suchmaschine Merobase, welche die eigentlich Arbeit übernimmt.
  - Merobase erstellt ein Interface aus dem Testfall und sucht nach Komponenten, die auf diese zutreffen.
  - Merobase teilt Code Conjurer die Anzahl der gefundenen Komponenten mit, welches es im Status Feld anzeigt.
  - Danach testet Merobase die Kandidaten mit Hilfe des Testfalls, in einer abgesicherten virtuellen Maschine, und filtert jene Kandidaten aus, die den Test nicht bestanden haben.
- Code Conjurer zeigt die Anzahl der erfolgreich getesteten Kandidaten in dem Status Feld an.
- In dem Fenster mit den Details zur Suche kann der Benutzer sich den Aufbau der gefundenen Komponenten anschauen, sowie den Quellcode überprüfen.
- Hat sich der Benutzer für eine Komponenten entschieden, kann er diese,per Drag-And-Drop zu seinem Projekt hinzufügen, wobei Code Conjurer die Abhängigkeiten automatisch auflöst.

Weitere Funktionen von Code Conjurer 1

Weitere Funktionen von Code Conjurer 1 sind die Automated Adaptation und Proactive Reuse Recommendations,die im Folgenden vorgestellt werden.

Automated Adaptation In manchen Fällen werden nur wenige oder keine Komponenten gefunden, die mit dem Testfall übereinstimmen, aus diesem Grund bietet Code Conjurer die sogenannte Automated Adaptation. Dabei handelt es sich um Heuristiken, die z.B. Objekt- und Methodennamen ignoriert, so dass die Anzahl der gefundenen Komponenten erhöht wird. Außerdem, werden die Methoden so vertauscht, wenn dies möglich ist, dass mehr Testfälle durchlaufen werden können. Kurzum mit der Hilfe der Automated Adapation kann Code Conjurer Komponenten vorschlagen, die dem von Merobase erstellten Interface nicht entsprechen und trotzdem die benötigte Funktionalität liefern. Die Unterschiede zwischen der “normalen“ Suche und Automated Adaptation sind der Tabelle in der Abbildung 4.1 zu entnehmen.

Es fällt auf, dass teilweise deutlich mehr Kandidaten gefunden wurden und auch erfolgreich getestet werden konnten, allerdings hat sich die Ausführungszeit teilweise deutlich erhöht. Dies sieht man besonders beim Calculator Beispiel bei der die Dauer von 19 Sekunden auf 20 Stunden und 24 Minuten gestiegen ist. Trotzdem darf man diese Möglichkeit nicht außer acht lassen, da etwa bei Spreadsheet Beispiel mit der Automated Search 4 funktionierende Komponenten gefunden werden konnten, während die “normale“ Suche zu keinem Ergebnis kam.

	Interface-based matching			Automated adaptation engine		
Desired component	**Candidates**	**Matches**	**Time**	**Candidates**	**Matches**	**Time**
Calculator(sub(int,int):int add(int,int):int mult(int,int):int div(int,int):int)	4	1	19 sec.	23,759	22	20 hrs., 24 min.
Stack(push(Object):void pop():Object)	692	150	26 min.	35,634	611	18 hrs., 23 min.
Matrix (Matrix(int, int) get(int,int):double set(int,int, double):void multiply(Matrix): Matrix)	10	2	23 sec.	137	26	5 min., 25 sec.
ShoppingCart(getItemCount():int getBalance():double addItem(Product):void empty():void removeItem(Product):void)	4	4	26 sec.	12	4	47 sec.
Spreadsheet (put(String,String):void get(String):String)	0	0	3 sec.	22,705	4	15 hrs., 13 min.
ComplexNumber (ComplexNumber(double,double) add(ComplexNumber): ComplexNumber getRealPart():double getImaginaryPart():double)	1	0	3 sec.	89	32	1 min., 19 sec.
MortgageCalculator(setRate(double):void setPrincipal(double):void setYears(int):void getMonthlyPayment():double)	0	0	4 sec.	4,265	15	3 hrs., 19 min.

Abbildung 4.1: Vergleich der “normalen“ Suche gegenüber von Automated Adapation

Proactive Reuse Recommendations Code Conjurer kann den Entwickler proaktiv beim Designen seines Projektes unterstützen, in dem es ihm Vorschläge zu den noch unfertigen Klassen gibt. Damit der Benutzer nicht von seiner eigentlichen Arbeit, dem Designen von KLassen abgelenkt wird, schlägt Code Conjurer die Suche jedes mal automatisch an, sobald eine Methode zu der Klasse hinzugefügt, entfernt oder verändert wird. Die Vorschläge werden wie gewohnt angezeigt und können ins Projekt integriert werden.

Auch wenn der Benutzer die vorgeschlagenen Komponenten nicht benutzt, so unterstützt ihn Code Conjurer trotzdem, da er sieht welcher Aufbau ähnliche Klassen häufig besitzen. Dies Vorgehensweise kann somit das Design des Projektes verbessern, da keine geläufigen Methoden vergessen werden. Als Beispiel, der Benutzer will eine Stack Klasse erstellen und schreibt eine Klasse mit folgendem Aufbau:

```
Stack{
    void push(Object o){}
    Object pop(){}
}
```

Daraufhin schlägt Code Conjurer folgende Klasse vor, von der er die zusätzlichen Methoden übernehmen kann.

```
Stack{
    boolean isEmpty(){}
    Object pop(){}
    void push(Object arg1){}
    Object top(){}
}
```

Zusammenfassende Featureliste von Code Conjurer 1

Aus den vorherigen Abschnitten kann eine Featureliste von Code Conjurer 1 erstellt werden, die dann später verwendet werden kann, um aufzuzeigen, welche Features bei Code Conjurer 2 fehlen.

- Suche nach Klassen mit Hilfe des Klassenaufbaus
- Testgetriebene Suche
- Anzeigen einer Übersicht zu den Suchergebnisse in einem Status Feld
- Anzeigen der Suchergebnisse mit Quellcode

- Hinzufügen einer Komponente per Drag-And-Drop zum eigenen Projekt
- Automatisches Auflösen der Abhängigkeiten von einem hinzugefügten Projekt
- Automated Adaptation
- Proactive Reuse Recommendations

4.3.2 Erfahrungen mit Code Conjurer 2

Um Code Conjurer 2 zu testen wurden verschiedene Beispiele aus Abbildung 4.1 verwendet.

Shopping Card

Bei diesem Beispiel soll eine Komponente für einen Einkaufswagen gefunden werden, dazu wurde eine Java Klasse angelegt, die den gleichen Aufbau hat, wie dies oben in der Abbildung gezeigt wird. In einem ersten Schritt soll Code Conjurer die normale Codesuche verwenden, diese findet allerdings keine Ergebnisse. Gibt man die Klasse bei Merobase direkt ein, so erhält man zwar Suchergebnisse, allerdings kann Merobase den Quellcode dieser Komponenten nicht mehr finden. Ein weiteres Testen mit diesem Beispiel ist demnach sinnlos.

Complex Number

Bei der Suche nach Komponenten für komplexe Zahlen wurden zumindest Suchergebnisse gefunden, die auf den ersten Blick vielversprechend aussahen. Diese Ergebnisse sollten mit Hilfe der testgetriebene Suche gefiltert werden. Allerdings konnte keine der Komponente den Test bestehen. Dies kann einerseits daran liegen, dass die Suche auf 30 Komponenten beschränkt ist. Diese Beschränkung ist von Merobase so vorgegeben, um die Server nicht zu sehr zu belasten. Andererseits kann es auch auf den Code zurückzuführen sein, den Merobase zur Verfügung

steht. Stichprobenweise stellte sich heraus, dass die Methoden nicht implementiert waren, Methoden fehlten oder dass die Komponenten nichts mit komplexen Zahlen gemein hatten.

Fehlende Features in Code Conjurer 2

5 Diskussion

5.1 Probleme

5.2 Fazit

Literatur

[1] *Code Conjurer - SourceForge.net*. URL: http://sourceforge.net/projects/codeconjurer/files/.

[2] P. Freeman, Hrsg. *Tutorial, software reusability*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1987. ISBN: 0-818-60750-5.

[3] Harald Gall. *Kapitel 12 - Software Wiederverwendung*. URL: https://files.ifi.uzh.ch/rerg/amadeus/teaching/courses/kvse_ss05/kap12-reuse.pdf (besucht am 07. 09. 2012).

[4] O. Hummel und W. Janjic. „Schwerpunkt-Probieren geht üer studieren: Testgetriebene Wiederverwendung von Softwareobjekten“. In: *Objekt Spektrum* 5 (2011), S. 8.

[5] O. Hummel, W. Janjic und C. Atkinson. „Code conjurer: Pulling reusable software out of thin air“. In: *Software, IEEE* 25.5 (2008), S. 45–52.

[6] Charles W. Krueger. „Software reuse“. In: *ACM Comput. Surv.* 24.2 (Juni 1992), S. 131–183. ISSN: 0360-0300. DOI: 10.1145/130844.130856. URL: http://doi.acm.org/10.1145/130844.130856.

[7] Otávio Augusto Lazzarini Lemos u. a. „CodeGenie: using test-cases to search and reuse source code“. In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ASE '07. Atlanta, Georgia, USA: ACM, 2007, S. 525–526. ISBN: 978-1-59593-882-4. URL: http://dx.doi.org/10.1145/1321631.1321726.

[8] Doug McIlroy. „Mass-Produced Software Components“. In: *Proceedings of NATO Software Engineering Conference*. Hrsg. von P. Naur und B. Randell. Garmisch, Germany, 1968, S. 138–155.

[9] A. Mili, R. Mili und R. T. Mittermeir. „A survey of software reuse libraries“. In: *Ann. Softw. Eng.* 5 (Jan. 1998), S. 349–414. ISSN: 1022-7091. URL: http://dl.acm.org/citation.cfm?id=590631.590637.

[10] Andy Podgurski und Lynn Pierce. „Retrieving reusable software by sampling behavior“. In: *ACM Trans. Softw. Eng. Methodol.* 2.3 (Juli 1993), S. 286–303. ISSN: 1049-331X. DOI: 10.1145/152388.152392. URL: http://doi.acm.org/10.1145/152388.152392.

[11] Rubén Prieto-Díaz. „Status Report: Software Reusability“. In: *IEEE Softw.* 10.3 (Mai 1993), S. 61–66. ISSN: 0740-7459. DOI: 10.1109/52.210605. URL: http://dx.doi.org/10.1109/52.210605.

[12] *S6 Search Page*. URL: http://conifer.cs.brown.edu:8180/S6Search/s6search.html.

[13] *Sky Werbefilm:Hör auf den Fan in Dir*. URL: http://www.youtube.com/watch?v=JDfmiaHD6ZU (besucht am 07. 09. 2012).