Ulrike Jaeger, Kurt Schneider (Hrsg.) (2009): Software Engineering im Unterricht der Hochschulen,
SEUH 11, Hannover, dpunkt.verlag, Heidelberg

Werkzeuge zur Qualitätssicherung in der Software-Engineering-Ausbildung

Stephan Kleuker

Fachbereich Design Informatik Medien
Fachhochschule Wiesbaden
kleuker@informatik.fh-wiesbaden.de

Zusammenfassung

Qualitätssicherung wird häufig als notwendiges Übel der vermeintlich kreativeren Softwareentwicklung angesehen. Mit den richtigen Werkzeugen und dem Wissen über passende Verfahren kann die Qualitätssicherung zur spannenden Aufgabe bei der Suche nach potenziellen Fehlern werden. Im hier vorgestellten Forschungsprojekt werden zur Zeit freie Werkzeuge zur Analyse von Java-Programmen für ihren Einsatz in der Software-Engineering-Lehre und der Praxis bewertet.

1 Einleitung

Die Qualitätssicherung (QS) als Teil der Softwareentwicklung spielt für den Projekterfolg eine immer größere Rolle, da unzuverlässige Softwaresysteme von Kunden immer weniger akzeptiert werden. Da allerdings die Mittel für die Qualitätssicherung nicht beliebig erhöht werden können, stellt sich die Frage, welche Ansätze der Qualitätssicherung unter welchen Randbedingungen die größten Erfolgswahrscheinlichkeiten bei der Suche nach gravierenden Fehlern haben.

QS ist damit ein wichtiges Thema in der Software-Engineering-Ausbildung im Studium. Leider verfolgt dieses Thema das Vorurteil der langweiligen, monotonen Arbeit, was durchaus berechtigt ist, wenn man an sich immer wiederholende Tests denkt, die nach jeder Softwareänderung erneut mit genau der gleichen Vorgehensweise manuell durchgeführt werden müssen. Interessant werden die Aufgaben mit neuen Werkzeugen, die das Aufspüren von Fehlern unterschiedlicher Art wesentlich erleichtern und die Testwiederholung automatisieren. Mit

der Idee, die Werkzeuge der »Computer Software Investigation«, kurz CSI, kennenzulernen, besteht die Möglichkeit, das Interesse auf das Thema QS zu leiten.

Allein im Bereich der Java-Programmierung wurde in den letzten Jahren eine große Anzahl von QS-Werkzeugen entwickelt, wobei es praktisch unmöglich ist, sich einen annähernd vollständigen Überblick zu verschaffen. Zentrale Anlaufpunkte sind die folgenden Seiten: [@oss], [@tut] und [@ecp]. Einen guten Überblick über kommerzielle Werkzeuge liefert [@imb]. Neben kurzen Werkzeug-Beschreibungen und Bewertungen einzelner Nutzer findet man wenig bis keine Informationen zu ihren Einsatzbereichen und der Integration mit anderen Werkzeugen. Oft sind die Werkzeuge schwer nutzbar, da die Nutzungsmöglichkeiten vielfältig sind und Einführungen bei Open-Source-Werkzeugen von Entwicklern geschrieben wurden, die nicht die Sicht von Außenstehenden einnehmen können. Es stellt sich damit die Frage, welche Werkzeuge sich für den Einsatz im Bereich QS der Software-Engineering-Ausbildung und der Praxis eignen. Diese Frage wird zunächst eingeschränkt auf ein enges Gebiet, pure Java-Programme und Open-Source-Werkzeuge, im Projekt »Überprüfung von freien QS-Werkzeugen in der Softwareentwicklung auf ihre Anwendbarkeit« (QSQS) untersucht. QSQS findet unter dem Projektdach mit dem Namen »Kombination von Qualitätssicherungsmaßnahmen« (KombiQu) [@kbq] statt. Die Fachhochschule Wiesbaden unterstützt die Forschungsbemühungen mit einer internen Finanzierung. In QSQS werden in Zusammenarbeit mit studentischen Hilfskräften QS-Werkzeuge für Standard-Java-Programme auf ihre Funktionalität, Nutzbarkeit und längerfristige Verwendbarkeit analysiert. Die bisher erreichten Ergebnisse werden kontinuierlich im Internet veröffentlicht.

Im folgenden Text werden zunächst einige Grundlagen der QS wiederholt, dann Bewertungskriterien für Werkzeuge vorgestellt, um darauf aufbauend Werkzeuge für unterschiedliche Aufgabenbereiche für ihre Nutzung in der Ausbildung zu analysieren. Diese Kenntnisse können leicht auf die Nutzbarkeit in Unternehmen übertragen werden, da sich Mitarbeiter häufig individuell im Selbststudium mit QS-Werkzeugen beschäftigen, um ihre Einsatzmöglichkeiten in der Praxis zu evaluieren. Abschließend wird ein erstes Fazit mit einem konkreten Vorschlag für eine Entwicklungsumgebung aus Sicht der QS gegeben.

2 Wesentliche Grundlagen der Qualitätssicherung

Innerhalb der Qualitätssicherung gibt es verschiedene Teilgebiete, die z.B. konstruktive Maßnahmen, wie Guidelines und die Auswahl passender Schulungen, und analytische Maßnahmen, wie das klassische Testen und den Einsatz von Metriken [Hen96], umfassen. Weitere Klassifizierungen der QS-Maßnahmen sind möglich, wie es z.B. in [Lig02] oder [Hof08] gezeigt wird. Im Mittelpunkt von QSQS stehen analytische Maßnahmen für einfache Java-Programme, die in der bisherigen Literatur nur punktuell bezogen auf einen Ansatz, ein Werkzeug

oder eine Java-Technologie beschrieben werden. Die genauer betrachteten analytischen Verfahren können generell wie folgt klassifiziert werden:

- Funktionale Tests:
  Ansätze zur Prüfung, ob ein in Entwicklung befindliches System die geforderte Funktionalität liefert.
  - Äquivalenzklassenanalyse:
    Ansatz zur systematischen Auswahl möglichst weniger Testfälle, die möglichst viele potenzielle Fehler aufdecken können.
  - Überdeckungstests:
    Ansätze zur Prüfung, ob bestimmte Anweisungen, Ablaufpfade oder Boolesche Bedingungen garantiert in Testfällen berücksichtigt werden.
- Prüfung nichtfunktionaler Anforderungen:
  - Performance-Analyse (Lasttest):
    Prüfung, ob ein in Entwicklung befindliches System auch unter bestimmten Lasten die erwartete Reaktionsgeschwindigkeit beibehält.
  - Speichernutzung:
    Prüfung, ob ein in Entwicklung befindliches System auch unter bestimmten Lasten Grenzen für die maximale Speichernutzung beibehält.
  - Codequalitätsanalyse durch Metriken:
    Berechnung von speziellen Kennzahlen einer Software, um Indikatoren für die Qualität des Programmcodes hinsichtlich Wartbarkeit und Änderbarkeit zu erhalten.
  - Usability-Prüfung:
    Analyse, ob ein in Entwicklung befindliches System benutzbar ist.
- Manuelle Prüfverfahren:
  Menschen prüfen (Teil-)Produkte eines in Entwicklung befindlichen Systems nach bestimmten Kriterien, die typischerweise nicht automatisch geprüft werden können, wie z.B. die Lesbarkeit von Dokumenten und die vollständige Umsetzung von Anforderungen.

Betrachtet man die aktuelle Literatur, z.B. [SW02], [HT05] und [Lin05], so konzentrieren sich die Möglichkeiten meist auf den Unit-Test-Ansatz von Kent Beck und Erich Gamma [Bec00] [Bec06], der es erlaubt, Tests in der Programmiersprache des Softwareprojekts zu formulieren. Ausgehend von diesem zentralen Ansatz sind viele weitere Werkzeuge entstanden, die die Ideen von Unit-Tests um weitere Varianten und auf spezielle Anwendungsbereiche erweitern. Die zugehörigen Ansätze und andere frei zugängliche Testwerkzeuge sind in Unternehmen zu wenig verbreitet [@ueb06], da sie oft nicht bekannt sind oder sich die Einarbeitung, die meist durch Mitarbeiter »nebenbei« passieren muss, als zu komplex

herausstellt. Kommerzielle Werkzeuge erschrecken üblicherweise mit ihren enormen Lizenz- und Folgekosten.

Die generellen Ausführungen zu den verschiedenen Testarten in den folgenden Kapiteln sind eng an [Kle08] angelehnt.

3 Bewertungskriterien

Der Ausgangspunkt für die Erstellung der Bewertungskriterien sind die Erfahrungen des Autors mit verschiedenen Vorlesungen zu den Themengebieten Software Engineering, Programmierung und Qualitätssicherung an den Fachhochschulen Wiesbaden und Nordakademie in Elmshorn. In den Praktika zu den Vorlesungen wurden verschiedene QS-Werkzeuge eingesetzt, zu denen im Vorlesungsteil nur die dahinter liegenden Ansätze vorgestellt wurden. Dies führt dazu, dass sich Studierende in den Übungen mit der neuen Methodik und der Nutzung des Werkzeugs im engen studentischen Zeitrahmen auseinander setzen müssen. Um hier erfolgreich mit dem Werkzeug arbeiten zu können, gibt es zwei zentrale Qualitätskriterien für die QS-Werkzeuge: Sie müssen intuitiv bedienbar und ihre Ergebnisse auf den ersten Blick interpretierbar sein. Eine intuitive Bedienbarkeit liegt z.B. vor, wenn man in einer Entwicklungsumgebung maximal eine neue Einstellung und dann in einem Kontextmenü zum Starten eine neue Auswahl vornehmen muss. Ergebnisse sind schnell interpretierbar, wenn sie visuell durch Farben oder erzeugte Tabellen angezeigt werden.

Markantes Beispiel für ein Werkzeug, das diese Anforderungen nicht erfüllt, ist Jester (JUnit Testtester, *http://jester.sourceforge.net*), ein Werkzeug, mit dem man zu testende Programme mutieren kann; genauer werden Bedingungen in if- und while-Befehlen durch false und true ersetzt sowie Zahlen verändert. Führt man dann seine Tests mit dem mutierten Programm durch, sollten die Tests bei genügender Präzision, z.B. detaillierter Äquivalenzklassenanalyse, diese Mutationen finden und Fehler ausgeben. Da das Werkzeug nicht ganz einfach installierbar ist, man außerdem aufpassen muss, dass nicht der ursprüngliche Programmquellcode verändert wird und als kritisch ausgegebene Mutationen von Hand analysiert werden müssen, kann es nicht »nebenbei« im Praktikum genutzt werden.

Basierend auf diesen Lehrerfahrungen stellt sich die Frage nach einer Werkzeugsammlung, die Studierende in Praktika einfach nutzen können und mit denen wesentliche Aspekte der QS behandelt werden. Zur Beantwortung müssen aktuelle Werkzeuge analysiert werden. Da die derzeitige Prüfungsordnung keine Integration dieser Aufgabenstellung in eine Projektform im Studienplan ermöglichte, findet die Arbeit in einem kleinen Forschungsprojekt statt. Die Untersuchungen werden dabei von Bachelor-Studierenden durchgeführt, die als generelle Aufgabe ein Werkzeug nach den genannten Kriterien untersuchen. Konkretes Ziel ist es dabei, möglichst schnell ein eigenes Beispiel zu konstruieren, mit dem man die wesentliche Funktionalität des untersuchten Werkzeugs vorstellen kann.

Parallel zu den Lehrerfahrungen wird auch der Bedarf in Unternehmen mehr und mehr erkannt, QS-Werkzeuge in die eigenen Entwicklungsprozesse zu integrieren. Dies wird durch verschiedene Abschlussarbeiten deutlich, in denen sich Studierende mit QS-Werkzeugen für konkrete Teilaufgaben beschäftigen. Aus diesem Grund wurde die Werkzeugbewertung um Kriterien erweitert, die für Unternehmen, aber auch für Studierende in studentischen Projektgruppen, in denen das Thema QS intensiver betrachtet wird, relevant sind.

Folgende Dokumentationsschablone wurde ab dem Punkt »Einsatzgebiete« zur Bewertung der Werkzeuge genutzt. Dabei wurde neben der Funktionalität besonders auf die Installierbarkeit, Nutzbarkeit und die Support-Seiten geachtet. Die Kriterien können für bestimmte Anwendungsszenarien gewichtet werden. Die letzte Spalte gibt mit einem Kreuz an, ob das Kriterium für die Grundlagenausbildung besonders relevant ist.

Kriterium	Beschreibung	LV
Name	Name des Werkzeugs	
Homepage	Ausgangsseite zur Homepage des Werkzeugs	
Lizenz	Unter welcher Lizenz ist das Werkzeug nutzbar?	
Untersuchte Version	Genaue Angabe der Versionsnummer und der genutzten Dateien	
Zeitpunkt	Wann wurde die Analyse zuletzt aktualisiert?	
Kurzbeschreibung	Beschreibung in ein bis vier Sätzen, welche Hauptaufgabe(n) das Werkzeug lösen soll	
Fazit	Ist das Werkzeug für Einsteiger in die Qualitätssicherung nutzbar, lohnt sich der Einstieg in die Einarbeitung für erfahrene Nutzer, gibt es besondere Highlights/Lowlights bei der Nutzung, Dokumentation oder Installation? Wie gut ist das Werkzeug im Vergleich zu Werkzeugen mit vergleichbarer Funktionalität?	
Einsatzgebiete	Wo kann das Werkzeug eingesetzt werden? Gibt es Bereiche, in denen das Werkzeug nicht nutzbar ist?	X
Einsatzumgebungen	Kann das Werkzeug ohne andere Werkzeuge benutzt, kann es (zusätzlich) als Plug-in in Eclipse oder Netbeans installiert werden?	X
Installation	Wie wird installiert bzw. wo (Link) steht eine Installationsanleitung? Kurze Beurteilung, ob Installation einfach für Programmierer durchführbar.	X
Dokumentation	Macht die Dokumentation für einen Entwickler mit Programmier-, aber wenig bis keiner QS-Erfahrung einen lesbaren Eindruck, gibt es einen Schnelleinstieg, gibt es die Dokumentation in deutsch/englisch, ist sie leicht herunterladbar (z. B. PDF oder/und Teil des Downloads), gibt es besondere Highlights (z. B. Videos)?	X
Wartung der Projektseite	Machen die Seiten des Werkzeugs den Eindruck, dass sie häufiger gepflegt werden, gibt es eine kontinuierliche Versionsgeschichte, erkennt man, dass neue Features (z.B. Generics und Annotationen in Java) die Entwicklung beeinflussen?	→

Kriterium	Beschreibung	LV
Nutzergruppen und Support	Gibt es eine detaillierte Fragen- und Antwortseite, gibt es Foren und Nutzergruppen zur Diskussion über das Werkzeug? Gibt es eine E-Mail-Adresse für Anfragen und Kommentare?	
Intuitive Nutzbarkeit	Kann sich ein erfahrener Entwickler anhand der Dokumentation leicht einarbeiten, kann er schnell zu eigenen Beispielen kommen? Werden besondere Fähigkeiten bzw. Kenntnisse gefordert? Gab es besondere Beobachtungen bei der Einarbeitung?	X
Automatisierung	Kann das Werkzeug ohne Nutzer nach dem Start laufen, werden die gewonnenen Ergebnisse lesbar aufbereitet (z. B. HTML-Export), kann der Nutzungsprozess einfach automatisiert werden (z. B. mit Shell-Skripten oder als Ant-Task)?	

Tab. 1 *Dokumentationsschablone*

4 Funktionstest

Die Funktionalität eines Systems lässt sich sehr gut mit Unit-Tests prüfen. Dabei kann man den Code sehr genau analysieren (White-Box-Tests) oder Zusammenhänge zwischen Komponenten (Gray-Box-Test) oder ganze Systeme (Black-Box-Test). Als Standardwerkzeug kann hier JUnit (*http://www.junit.org/*) angesehen werden. Mit JUnit ist es möglich, Tests direkt in Java zu schreiben, Studierende lernen dabei, warum es wichtig ist, für jeden Test die gleiche Umgebung zu schaffen und dass Tests automatisierbar sein können.

Nach einer durchaus aufwendigeren Einarbeitung, in der das Grundverständnis für die Vorgehensweise dem Entwickler verständlich werden muss, ist es sehr leicht, systematisch Tests zu schreiben. JUnit ist in alle wichtigen Entwicklungsumgebungen integriert, kann aber auch sehr gut ohne Umgebung und integriert in eine Automatisierungsumgebung ablaufen. Die systematische Testerstellung mit JUnit ist damit immer der Einstiegspunkt in das systematische Testen.

JUnit existiert zurzeit in den Versionen 3.8 und 4.x, wobei 3.8 »klassisches« Java ohne Annotationen nutzt. JUnit 4.x ist etwas flexibler durch die Verwendung von Annotationen, was die Nutzbarkeit aber vergleichbar lässt. Welche Variante man nutzt, ist fast egal, hängt davon ab, was man noch weiter im Testumfeld machen will. Der Übergang von JUnit 3.8 zu Unit-Tests anderer Programmiersprachen wie C++ gestaltet sich etwas einfacher, weiterhin gibt es weitere QS-Werkzeuge, die JUnit-Tests zur Testwiederholung generieren, wobei diese Tests teilweise nur unter JUnit 3.8 laufen.

Eine Alternative zu JUnit stellt TestNG (*http://www.testng.org/*) dar, das genau den gleichen Prinzipien von JUnit folgt. JUnit-Tests können leicht in TestNG-Tests verwandelt werden. TestNG bietet etwas mehr Unterstützung bei der systematischen Gruppierung von Tests in Teilgruppen und bei der Ausführung gleichartiger Tests mit unterschiedlichen Parametern, die auch aus Dateien

eingelesen werden können. Die Dokumentation von TestNG ist noch verbesserungswürdig, gerade in den Bereichen, in denen mehr Funktionalität als JUnit geboten wird.

Einen vollständig anderen Ansatz verfolgt das Werkzeug FIT (*http://fit.c2.com/*), dass davon ausgeht, dass Tests von Leuten geschrieben werden, die diese nicht ausprogrammieren wollen oder können. FIT geht einen eher unkonventionellen Weg, indem die Tests nicht in der jeweiligen Programmiersprache geschrieben werden, sondern von Entwicklern Methoden zur Verfügung gestellt werden, die FIT aufruft. Die Aufrufparameter stehen dann in einer vom Tester geschriebenen HTML-Tabelle. Das Ergebnis wird ebenfalls in eine HTML-Datei geschrieben. Nach kurzer Einarbeitung ist auch dieser Ansatz sehr intuitiv nutzbar. Wichtig ist die Gestaltung der Testschnittstelle, da sie wesentlich beeinflusst, was überhaupt testbar wird.

Als Fazit für eine minimale Umgebung wird JUnit 3.8 vorgeschlagen, wobei alle anderen genannten Werkzeuge für eine vertiefte Betrachtung des Themengebiets sehr interessant sind.

5 Überdeckungstest

Zu jedem Programm kann man einen gerichteten Kontrollflussgraphen angeben, der die möglichen Programmausführungen beschreibt. Dabei sind die Knoten des Graphen ausgeführte Befehle, die durch gerichtete Kanten verbunden werden. If-Befehle, Switch-Befehle und Schleifen führen dabei zu Verzweigungen im Graphen, der Graph kann durch Normalisierung unabhängig von der Programmformatierung gemacht werden. Abbildung 1 zeigt eine Methode und den zugeordneten Graphen.

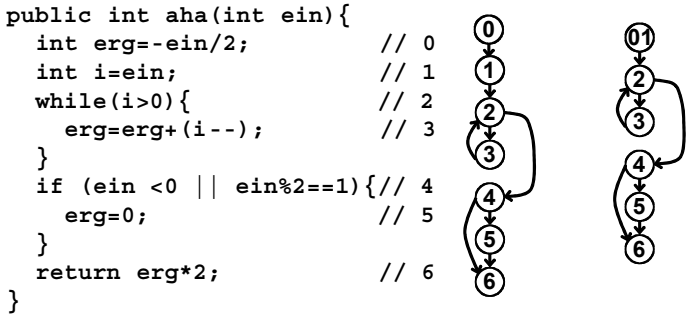

Abb. 1 *Methode mit Kontrollflussgraph und normalisierter Variante [Kle08]*

Der Ansatz bei den vorgestellten Überdeckungsmaßen ist, dass man fordert, dass mit einer Menge von Testfällen das jeweilige Maß möglichst groß, genauer möglichst der Wert Eins, ist. Für die sogenannte C0-Überdeckung wird gezählt, wie

viele Knoten von den Tests im Verhältnis zur Anzahl aller Knoten besucht wurden. Bei der C1-Überdeckung, auch Zweig- oder Kantenüberdeckung genannt, wird gefordert, dass alle Kanten überdeckt werden. In verwandten Maßen werden nur die Boole'schen Bedingungen der Programme betrachtet und gefordert, dass alle Bedingungen oder gar alle Teilbedingungen mit Tests einmal nach wahr und einmal nach falsch ausgewertet werden.

Generell kann die Aufforderung an Studierende, ihre Programme zu testen, dadurch konkretisiert werden, dass die Erreichung bestimmter Überdeckungsmaße gefordert wird. Im Rahmen von QSQS wurden folgende Werkzeuge untersucht.

Coverlipse (*http://coverlipse.sourceforge.net/index.php*) ist einfach als Plug-in in Eclipse installierbar und erlaubt die Prüfung auf die Anweisungsüberdeckung. Das Werkzeug prüft die Überdeckung bezüglich vorher zu entwickelnder JUnit-Tests. Die Nutzung ist recht einfach, statt die Tests als JUnit-Tests zu starten, existiert eine Startmöglichkeit mit dem Menü-Punkt »JUnit w/Coverlipse«. Nach der Programmausführung werden die ausgeführten Zeilen markiert und das Überdeckungsmaß angezeigt. Sehr interessant ist eine zweite Testart, mit der geprüft wird, ob einzelne Zuweisungen Auswirkungen haben, also der Wert der zugewiesenen Variable später noch mal genutzt wird. Aktuell scheint es keine Weiterentwicklung zu geben.

EclEmma (*http://www.eclemma.org/*) ist einfach als Plug-in in Eclipse installierbar und erlaubt die Prüfung auf eine Anweisungsüberdeckung. Die Nutzung ist optimal vereinfacht, statt das Programm über den Eclipse-Startknopf laufen zu lassen, existiert ein neuer Knopf, um das Programm mit Coverlipse zu starten. Natürlich sind auch JUnit-Tests startbar. Nach der Programmausführung werden die ausgeführten Zeilen und das Überdeckungsmaß angezeigt. Leider kann man selten eine 100 %-Überdeckung erreichen, da das Werkzeug Schwierigkeiten mit Zeilen hat, die z.B. Klassen-Deklarationen enthalten. Eine Testautomatisierung außerhalb von Eclipse ist nur über das verwandte Werkzeug Emma möglich.

CodeCover (*http://www.codecover.org/*) wurde an der Uni Stuttgart entwickelt und kann allein oder als Eclipse-Plug-in genutzt werden. Nachdem man die etwas aufwendige Projekteinrichtung verstanden hat, CodeCover muss aktiviert und dann für die zu analysierenden Klassen eingeschaltet werden, ist es leicht nutzbar. CodeCover ist sehr interessant, da es neben der reinen Anweisungsüberdeckung die Möglichkeit bietet, für Bedingungen zu analysieren, ob sie nach wahr und falsch ausgewertet werden (verwandt zur Zweigüberdeckung, die allerdings nicht berechnet wird). Weiterhin besteht die Möglichkeit, für Boole'sche Ausdrücke genau die Werte der einzelnen Ausdrücke zu analysieren, wobei eine Form der Bedingungsüberdeckung, die Modifizierte Bedingungs-/Entscheidungsüberdeckung (MC/DC) geprüft wird. Unschönes Detail ist, dass return-Anweisungen nicht als eigene Programmzeilen aufgefasst werden. Durch die Auswertungsvarianten dauert es, bis Studierende die Möglichkeiten des Werkzeugs

begriffen haben, danach bleibt es ein wichtiger Baustein auch bei weiteren Softwareprojekten.

Als Fazit lässt sich feststellen, dass CodeCover sehr gut für die Lehre geeignet ist, da es nach kleiner Einarbeitungshürde intuitiv nutzbar ist und verschiedene Maße liefert, evtl. findet in weiteren Releases eine Orientierung an klassischen C0-, C1-, C2-Maßen statt. Für vertiefende Betrachtungen ist auch Coverlipse interessant, da es zusätzlich die Möglichkeit bietet, die Abhängigkeit von Programmfragmenten zu nutzen.

6 Oberflächengesteuerte Tests

Generell sind Oberflächen normale Softwarekomponenten und können deshalb in der Entwicklung und Nutzung genau wie andere Software auch getestet werden. Dabei wird das Design getrennt betrachtet. Insbesondere die Äquivalenzklassenmethode eignet sich zur Erstellung von Testklassen.

Ein eher technisches Problem ist, dass die Ausführung der Tests meist nicht mit den Standardtestwerkzeugen ausgeführt werden kann. Aus diesem Grund werden hierzu häufig sogenannte Capture-and-Replay-Werkzeuge genutzt. Die zentrale Idee ist dabei, dass die Nutzung der Oberfläche, also z. B. das Eintragen von Werten und das Ziehen sowie Klicken der Maus, aufgezeichnet wird. In das Protokoll dieser Aufzeichnung kann man die geforderten Eigenschaften, z.B. Ausgaben, angezeigte Texte und Anzeigefarben, eintragen. Teilweise besteht auch die Möglichkeit, dass Teile des Bildschirms als Bild mit vorher gemachten Bildern, also Screenshots, in Teilen oder als Ganzes verglichen werden.

Die so aufgezeichneten Testfälle können dann immer wieder, bei jedem neuen Software-Release, möglichst automatisch durchgeführt werden. Testwerkzeuge mit Replay-Möglichkeit führen dann die aufgezeichneten Mausbewegungen aus. Weiterhin ist eine spätere Anpassung der Testfälle möglich, so dass z.B. geprüft werden kann, ob in einem Ausgabefeld ein erwarteter Wert steht.

Abbot/Costello (*http://abbot.sourceforge.net/*) erlaubt die Aufzeichnung von Mausbewegungen und Klicks auf Standard-Java-Oberflächen, die AWT und Swing nutzen. Die Aufzeichnungen können erneut abgespielt werden, weiterhin kann man die Prüfung von Eigenschaften der GUI-Elemente einbauen. Gerade beim ersten Kontakt ist die Handhabung recht trickreich, und Studierende können so leicht den Spaß am Experimentieren verlieren. Ist die erste Aufzeichnung gelungen, lehrt das Werkzeug, dass man Oberflächen sauber strukturieren und allen Elementen lesbare Namen zuordnen muss, damit man in einer hierarchisch strukturierten Oberfläche das Element identifizieren kann, dessen Eigenschaft man prüfen möchte. Die Handhabung von Testskripten ist nicht immer intuitiv, kleine Fehler werden nicht verziehen, so dass dieses generell interessante Werkzeug für den Einstieg nicht geeignet ist.

UISpec4J (*http://www.uispec4j.org/*) basiert auf JUnit und hat als Konzept, dass die GUI-Klassen aus Java durch eigene GUI-Klassen ersetzt werden, die die gleiche Funktionalität nach außen haben, nach innen aber mehr Testmöglichkeiten bieten. Dieser interessante Ansatz hat als Nachteil, dass es lange dauert, bis Java-Aktualisierungen in UISpec4J-Klassen übernommen werden. Weiterhin ist der Ansatz für ineinandergeschachtelte GUIs sehr aufwendig, da man immer im Detail alle GUI-Komponenten bei der Testerstellung kennen muss, wodurch leider der Vorteil der schnellen Testerstellung wieder verloren geht. Da gewisse Funktionalität unter Linux nicht lauffähig ist und man in der Wahl der Java-Version eingeschränkt wird, ist das Werkzeug zum Einsatz in der Lehre nicht zu empfehlen.

FEST (*http://fest.easytesting.org/wiki/pmwiki.php*) legt sogenannte Fixtures an, die eine Schnittstelle zu den nativen Java-Swing-Komponenten zur Verfügung stellen, um diese entsprechend zu testen. Als Grundlage werden JUnit oder TestNG genutzt. FEST ist modular aufgebaut, kann einfach als Jar-Datei in einem Projekt genutzt werden und unterstützt fast alle GUI-Komponenten beim Capture-and-Replay. Mit Hilfe der Dokumentation ist die hier nicht sehr aufwendige Einarbeitung möglich, wieder sollte das zu untersuchende GUI gut strukturiert und dokumentiert sein. Mit etwas Einarbeitungszeit ist das Werkzeug durch Studierende gut nutzbar.

Jacareto (*http://jacareto.sourceforge.net/*) ist ebenfalls ein Werkzeug, mit dem Klicks- und Mausbewegungen aufgezeichnet und abgespielt werden können. Da Jacareto GUI-Komponenten erkennt, können auch auf der GUI ausgegebene Ergebnisse mit einem erwarteten Ergebnis verglichen werden. Es fehlt allerdings die Möglichkeit, komplexere Prüfungen einfach zusammenzufassen. Weitere Tests haben die Nutzbarkeit für kleine Beispiele gezeigt, allerdings traten bei etwas komplexeren Anwendungen einige Fehler auf, weiterhin ist die Nutzung wenig intuitiv. Die Integrationsmöglichkeit in komplexere Testumgebungen ist praktisch nicht gegeben, die Nutzung unter Linux machte Probleme. Die Versionsnummer 0.7.12 trägt die betrachtete Version als Vorstufe zu einem einsetzbaren Werkzeug zu Recht.

Bei *Marathon* (*http://marathontesting.com/Home.html*) muss der Tester ebenfalls keinen Programmcode schreiben, Testfälle werden wieder aufgezeichnet und können mit Prüfungen auch während der Ausführungen annotiert werden. Das Werkzeug basiert auf JUnit 3.8. Die Entwicklungsumgebung von Marathon ist sehr strukturiert. Kennt man sich mit Eclipse bereits aus, findet man sich auch in dieser Umgebung schnell zurecht. Marathon kann unabhängig von Entwicklungsumgebungen genutzt werden und ist auch für unerfahrene Nutzer nach kurzem Studium der guten Dokumentation einsetzbar.

Die untersuchten Werkzeuge benötigen eine recht große Einarbeitungszeit oder müssen von Experten in Übungen sehr detailliert erklärt werden, da es hier leicht zu Misserfolgen kommen kann, wenn z.B. eine bei der ersten Nutzung

nicht intuitive und deshalb vergessene Einstellung die Werkzeugnutzbarkeit beeinflusst. Letztendlich können FEST, Marathon und auch Abbot für eine genauere Betrachtung von GUI-Testern genutzt werden.

In studentischen Softwareprojekten wurde deutlich, dass GUI-Tester nur sehr vereinzelt genutzt werden können, da sich die grafische Oberfläche nicht oder nur marginal ändern darf, damit vorher aufgezeichnete Tests wiederverwendet werden können. Dies ist bei kurzen studentischen Projekten schwer erreichbar.

Betrachtet man »außer Konkurrenz« GUI-Tester, die nicht für Standard-Java-Oberflächen geschrieben wurden, so sticht für Webapplikationen das Werkzeug *Selenium* (*http://selenium.openqa.org*) heraus, mit dem man nach relativ kurzer Einarbeitungszeit sehr intuitive Tests für Weboberflächen erstellen und wiederholen kann [KG08].

7 Mocking

Häufig tritt bei der parallelen Entwicklung durch mehrere Personen der Fall auf, dass man zum Test seiner eigenen Klassen die Klassen anderer Leute benötigt, die diese aber noch nicht implementiert haben. Statt zu warten, gibt es dann den Ansatz, sich diese Klassen in minimaler Form selbst zu schreiben. Dieser Ansatz wird in der Literatur die Erstellung sogenannter Mocks (teilweise auch Stubs) genannt. Man sucht dabei nach einer möglichst einfachen Implementierung, so dass man seine eigenen Tests sinnvoll ausführen kann.

Die einfachst mögliche Implementierung besteht darin, immer einen Default-Wert zurückzugeben. Für den Rückgabetypen void kann man eine vollständig leere Implementierung nutzen. Soll ein echtes Objekt zurückgegeben werden, reicht es für eine lauffähige Implementierung aus, dass null zurückgegeben wird. Bei Rückgabetypen wie boolean oder int muss man sich auf Standardwerte wie false und 0 einigen.

Mit der einfachst möglichen Implementierung kann man dann feststellen, ob die eigene Klasse implementierbar ist. Dies reicht aber häufig nicht aus, um die eigene Klasse vollständig zu testen, wenn man z.B. daran interessiert ist, ein bestimmtes Überdeckungsmaß zu erfüllen.

Generell ist es eine sehr gute Übungsaufgabe, Mocks zur Erreichung eines bestimmten Überdeckungsgrades von Hand zu schreiben, da dies eine intensive Auseinandersetzung mit dem Programmcode fördert. Danach stellt sich aber die Frage, welche Automatisierungsmöglichkeiten es gibt. Insgesamt wurden hierzu die folgenden Werkzeuge betrachtet.

JMock (*http://www.jmock.org/*) bietet eine einfache Möglichkeit, nicht vorhandene Klassen durch sogenannte Mock-Objekte zu testen. Es ist eine Klassenbibliothek, die zusammen mit JUnit nutzbar ist. Über Reflection ist es möglich, für eine fehlende Klasse, deren Interface bekannt sein muss, passende Objekte anzulegen. Neben der Objekterzeugung kann das weitere Verhalten eines Objek-

tes nach außen festgelegt werden, so dass echte Überdeckungstests der zu prüfenden Klasse möglich sind. Mit etwas Einarbeitungszeit ist die Nutzung auch für Anfänger intuitiv möglich.

EasyMock (*http://www.easymock.org/*) ist von seiner Arbeitsweise fast identisch wie JMock, stellt aber eine eigene Entwicklung dar. Wieder können Erwartungen an das Verhalten des entstehenden Mock-Objekts recht einfach festgelegt werden.

MockLib (*http://mocklib.sourceforge.net/mocklib3/index.html*) funktioniert im Prinzip wie die beiden anderen Mock-Werkzeuge, ist dabei aber komplexer zu bedienen. Zahlreiche kleine und nur unzureichend dokumentierte Fallstricke machen den Einstieg zäher und langwieriger, als er sein müsste. Die gebotene Funktionalität ist dagegen gut, trotzdem sind die anderen Werkzeuge MockLib vorzuziehen.

Zusammenfassend kann festgestellt werden, dass JMock und EasyMock generell gut nutzbar sind, man aber auf die Nutzung expliziter Mock-Werkzeuge in der Grundausbildung verzichten kann, da sie eine nicht triviale Einarbeitung benötigen und der Erkenntnisgewinn nicht sehr hoch ist, wenn Studierende gelernt haben, Mock-Objekte selbst zu schreiben. In vertiefenden Veranstaltungen z.B. zu verteilten Systemen, sollte die Mocking-Idee wieder aufgegriffen werden, da man dann ganze Komponenten, wie Server oder Datenbankverbindungen, durch Mocks ersetzen kann.

8 Weitere QS-Werkzeuge

Neben den testorientierten QS-Werkzeugen gibt es noch andere Ansätze, die die Qualität erhöhen können. Ein Beispiel sind Metriken, die auf dem Programmcode basierend berechnet werden und so gewisse Indizien für die Codequalität liefern. Typischer Vertreter einer Softwaremetrik ist die McCabe-Zahl, mit der die Anzahl der Verzweigungen einer Methode gemessen wird. Dieses Maß ist ein Indikator für die Komplexität und den Testaufwand, wobei allerdings implizite Verzweigungen, die z.B. durch Polymorphie möglich sind, nicht betrachtet werden. Ein Indikator für gute Objektorientierung heißt »Lack of Cohesion in Methods« (LCOM*) und prüft, ob Exemplarvariablen in möglichst vielen verschiedenen Methoden genutzt werden. Recht viele Maße sind intuitiv, und die Aussagekraft sowie Probleme von Metriken generell können so sehr gut mit Studierenden erarbeitet werden. Als Werkzeug kann hier *Metrics* (*http://metrics.sourceforge.net/*) als Plug-in für Eclipse empfohlen werden, da es einige Maße anbietet und sehr einfach bedienbar ist. Das Werkzeug wurde zwar seit Eclipse 3.1 nicht weiterentwickelt, ist aber auch in Eclipse 3.4 nutzbar. Als hilfreich stellen sich auch Werkzeuge zum Prüfen der Codeformatierung wie *CheckStyle* (*http://eclipse-cs.sourceforge.net/*) und zur statischen Analyse auf potenzielle Fehler wie *Findbugs* (*http://findbugs.sourceforge.net/*) heraus.

Das Projekt QSQS konzentriert sich zunächst auf einfache Standard-Java-Programme, wobei in den nächsten Schritten Werkzeuge für den Last- und Performance-Test betrachtet werden, auch wenn es in den betrachteten Themengebieten weitere interessante Kandidaten gibt. Für fortgeschrittene Java-Technologien gibt es eine Vielzahl weiterer Testwerkzeuge, die teilweise auf den hier genannten Werkzeugen basieren. Die Analyse der hier vorgestellten Werkzeuge und der neuen Werkzeuge ist ein offenes Forschungsfeld. Dabei müssen die Werkzeuge nicht nur einzeln betrachtet werden; es stellt sich die Frage, mit welcher Kombination von Werkzeugen man die größte Chance hat, möglichst viele kritische Fehler zu finden.

9 Fazit und Ausblick

Eine funktionierende Qualitätssicherung benötigt Werkzeuge, mit denen man in möglichst geringer Zeit möglichst viele Fehler finden kann. Qualitätssicherung muss damit im Rahmen der Grundausbildung im Software Engineering ein wichtiger Baustein sein. Für einen kompakten Bachelor-Studiengang ergeben sich damit folgende notwendigen Inhalte:

- Einführung von Testverfahren: Äquivalenzklassentests und Überdeckungen
- Programme testbar machen; Erstellung von Mocks
- Konzepte der White-, Gray- und Black-Box-Tests
- Integration der Testansätze in den gesamten Entwicklungsprozess, Regressionstests, Änderungsmanagement [Wal01]

Um diese sehr trockenen Ideen mit Leben zu füllen, muss die Theorie in praktischen Übungen an konkreten Beispielen vertieft werden. Mittlerweile gibt es im Java-Umfeld eine unübersichtliche Anzahl von Werkzeugen, die für Tests frei zur Verfügung stehen. Wesentliches Teilergebnis des Projekts QSQS ist ein Vorschlag für eine minimale CSI-Ausstattung einer Software-Entwicklungsumgebung mit Testwerkzeugen. Dieser Vorschlag umfasst folgende Werkzeuge:

- JUnit 3.8, nach kurzer Einarbeitung eine sehr intuitive Möglichkeit, Tests in Java für Java-Programme zu erstellen
- CodeCover, zur Überprüfung der Anweisungsüberdeckung und zur Analyse Boole'scher Verzweigungsmöglichkeiten
- Metrics, zur statischen Analyse des Programmcodes, u. a. mit zyklomatischer Zahl nach McCabe und LCOM*

Möchte man das Thema Qualitätssicherung intensiver betrachten, gibt es vielfältige Möglichkeiten zur Vertiefung. Ein sehr interessantes Gebiet ist dabei der Test ausgehend von Oberflächen. Die dazu untersuchten Werkzeuge können als »nutzbar, aber verbunden mit recht hoher Einarbeitungszeit« klassifiziert werden.

Insgesamt erlaubt das Thema Qualitätssicherung mit seiner Integration in den Entwicklungsprozess hervorragend den Aufbau eigener Vertiefungsveranstaltungen. Ausgehend von der Betrachtung von Standard-Java-Programmen ergeben sich für die mannigfaltigen Java-Technologien viele weitere Analysemöglichkeiten für vorhandene Werkzeuge. Bei der Entwicklung einer Vertiefung kann die Möglichkeit zur Integration des ersten Zertifikats »Foundation Level« zum »Certified Tester« [@gtb] [@ist], wie z.B. in der GI-Fachgruppe »Test, Analyse und Verifikation von Software« [@stt] diskutiert, geschaffen werden. Durch die enorme Tool-Vielfalt gerade auch für neuere Java-Technologien eröffnet sich ein wunderbares Feld für unterschiedliche Arten von Experimenten zur Einrichtung eines spezialisierten CSI-Labors. Die aktuellen Analyseergebnisse stehen unter [@kbq] der interessierten Öffentlichkeit zur Verfügung.

Danksagung

Zu guter Letzt sei Andreas Himmighofen und Daniel Knott für die sehr gute Zusammenarbeit gedankt.

Literatur und Webadressen

Links zuletzt am 1.12.2008 geprüft.

[@ecp] Eclipse Plug-in Central, *http://www.eclipseplugincentral.com/*

[@gtb] German Testing Board, *http://german-testing-board.info/*

[@imb] imbus AG – Testlabor, *http://imbus.de/tool-list.shtml*

[@ist] ISTQB:Home, *http://www.istqb.org/*

[@kbq] Kombination von Qualitätssicherungsmaßnahmen, *http://www.informatik.fh-wiesbaden.de/~kleuker/KombiQu/kombiqu.html*

[@oss] open source software testing tools, news and discussion, *http://www.opensourcetesting.org/functional.php*

[@stt] GI Softwaretechnik.Trends, *http://pi.informatik.uni-siegen.de/stt*

[@tut] Eclipse-Plug-ins, die's bringen, *http://www.tutego.com/java/eclipse/plugin/eclipse-plugins.html#Testen*

[@ueb06] S. Ueberhorst, Wer zu spät testet, verschleudert Geld, *http://www.tecchannel.de/test_technik/software/448205, 22.09.2006*

[Bec00] K. Beck, Extreme Programming, Addison-Wesley, München, 2000

[Bec06] K. Beck, JUnit die Chance geben, ein wenig länger zu leben, Javamagazin 08/06, Seiten 23-24

[Hen96] B. Henderson-Sellers, Object-Oriented Metrics, Measures of Complexity, Prentice Hall, USA, 1996

[Hof08] D. W. Hoffmann, Software-Qualität, Springer-Verlag, Berlin Heidelberg, 2008

[HT05] A. Hunt, D.Thomas, Pragmatisch Programmieren: Unit-Tests mit JUnit, Hanser, München, Wien, 2005

[KG08] T. Kamann, M. Groh, Webtests mit Selenium, Groovy, TestNG und Maven, Javamagazin 08/08, Seiten 65-69

[Kle09] S. Kleuker. Grundkurs Software-Engineering mit UML, Vieweg+Teubner, Wiesbaden, 2009

[Lig02] P. Liggesmeyer, Software-Qualität. Testen, Analysieren und Verifizieren von Software, Spektrum Akademischer Verlag, Heidelberg Berlin Oxford, 2002

[Lin05] J. Link, Softwaretests mit JUnit, 2. Auflage, dpunkt.verlag, Heidelberg, 2005

[SW02] H. M. Sneed, M. Winter, Testen objektorientierter Software, Hanser, München, Wien, 2002

[Wal01] E. Wallmüller, Software-Qualitätsmanagement in der Praxis, 2. Auflage, Hanser, München Wien, 2001