Leibniz Universität Hannover

Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Konzept und Implementierung einer aufgabenfokussierten Testumgebung für eine Crowdsourcing Plattform

Masterarbeit

im Studiengang Informatik

von

Florian Eiteljörge

Prüfer: Prof. Dr. Kurt Schneider
Zweitprüfer: Prof. Dr. Joel Greenyer
Betreuer: Dipl.-Math. Raphael Pham

Hannover, 30.04.2014

Zusammenfassung

Dass Unternehmen Teile ihres Softwareentwicklungsprozesses auslagern, ist heutzutage durchaus verbreitet. Insbesondere die Entwicklung von Softwaretests ist davon häufig betroffen - Stichwort Testing-as-a-Service. Bisher wird dabei vor allem an spezialisierte Unternehmen ausgelagert. Die Crowd wurde bisher höchstens zur massenhaften Ausführung von vordefinierten Tests genutzt, aber nicht um automatisierte Tests erstellen zu lassen. Dies liegt unter anderem daran, dass die Einstiegshürde für einen Tester aus der Crowd bisher sehr hoch ist.

In dieser Arbeit werden Konzepte vorgestellt, die es einem Tester erleichtern, automatisierte Tests für ein ihm unbekanntes Projekt zu entwickeln. Der Tester soll dabei aufgabenfokussiert arbeiten können. Tätigkeiten, die nicht unmittelbar zur Aufgabe gehören (z.B. Einrichten der Entwicklungsumgebung), sollen automatisiert werden. Außerdem soll der Tester bei der Einarbeitung in das Projekt durch die Entwicklungsumgebung unterstützt werden. Die präsentierten Konzepte wurden prototypisch in einem Eclipse-Plugin umgesetzt und anschließend evaluiert.

Abstract

The practice of companies to outsource parts of their software development process is quite common nowadays. Especially the development of software tests is often subject to outsourcing - see Testing-as-a-Service. So far these software tests are mostly contracted to specialized companies. The crowd was used at most to massively run predefined tests, but not to create automated software tests. One reason for this are the numerous difficulties a tester encounters when he is working on a project he is unfamiliar with.

In this thesis concepts will be introduced that simplify the work of a tester to create automated software tests for a project that was previously unknown to him. The tester should be able to work in a task-focussed way. Activities that are not directly associated with the task (e.g. the setup of the development environment) should be automated. Furthermore the tester should be supported by the development environment during the familarization process. The concepts presented here were implemented prototypical as an Eclipse plugin and evaluated.

Inhaltsverzeichnis

1. Einleitung **1**
1.1. Überblick . . . 1
1.2. Abgrenzung zu Crowdsourcing und Crowdtesting . . . 2
1.3. Vision . . . 3
1.4. Aufgabenstellung und Ziele . . . 5

2. Analyse des Testprozesses **7**
2.1. Unternehmensinternes Testen . . . 7
2.2. Ausgelagertes Testen . . . 8
2.3. Crowdsourcing von automatisierten Tests . . . 10

3. Analyse existierender Tools **12**
3.1. Eclipse IDE for Java EE Developers 4.3 (Kepler) . . . 12
3.2. NetBeans 7.4 . . . 15
3.3. IntelliJ IDEA 13.0 Ultimate . . . 17
3.4. Microsoft Visual Studio Ultimate 2013 . . . 18

4. Der Testauftrag **23**
4.1. Die zentralen Komponenten des Testauftrags . . . 24
4.2. Administrativ notwendige Komponenten . . . 25
4.3. Einschränkungen des Testauftrags . . . 26

5. Konzepte für die Entwicklungsumgebung - Aufteilung des Testprozesses **27**
5.1. Das Grundkonzept . . . 27
5.2. Phase 1: Explore-Perspective . . . 30
5.3. Phase 2: Recording-Perspective . . . 35
5.4. Phase 3: Coding-Perspective . . . 37
5.5. Weitere Konzepte . . . 39

6. Realisierung der Konzepte in Eclipse **42**
6.1. Übersicht über die Implementierung . . . 42
6.2. Der Testauftrag . . . 45
6.3. Recording-Perspektive . . . 46
6.4. Vermeidung von Konfiguration . . . 51
6.5. Einschränkungen des Prototyps . . . 51

7. Evaluation **53**
7.1. Aufbau und Ablauf . . . 53
7.2. Ergebnisse . . . 55

7.3. Erkenntnisse 59

8. Fazit **61**

A. Anhang **63**
A.1. Demographie-Fragebogen 64
A.2. Aufgabenstellung der Plugin-Gruppe 65
A.3. Aufgabenstellung und Hinweise der Kontrollgruppe 66
A.4. Abschlussinterviews für Probanden der Plugin-Gruppe 67

1. Einleitung

1.1. Überblick

„Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence“[1] schrieb Edsger Wybe Dijkstra schon 1972 [Dijkstra1972]. Trotzdem ist auch heute noch keine bessere Methode als das Testen bekannt, um Fehler eines Softwareprodukts aufzuspüren. Getestet wird entweder von Hand, indem die Software von einer Person ausgeführt und bedient wird, oder automatisiert durch eine Testsoftware. Voraussetzung für das Testen, egal ob manuell oder automatisiert durchgeführt, ist aber immer, dass das Sollverhalten der Software bekannt ist.

Setzt man dies voraus, ist es für das manuelle Testen darüber hinaus „nur“ erforderlich, dass die Software ausgeführt werden kann (dies kann je nach Art und Umgebung der Software natürlich auch schon einen erheblichen Aufwand darstellen). Für das automatisierte Testen müssen aber noch weitere Anstrengungen unternommen werden. Unter anderem müssen die Tests implementiert werden. Dies kann je nach Test mehr oder weniger aufwändig sein. Trotzdem lohnt sich in den meisten Fällen der zusätzliche Aufwand, der für die Erstellung der automatischen Tests nötig ist [Kasurinen2009]. Dieser wird nämlich dadurch ausgeglichen, dass das eigentliche Testen (also die Ausführung der Tests), schneller abläuft, als das Testen von Hand. Dadurch ist es möglich nach jeder Änderung der Software zu testen, ob bisherige Funktionen noch immer so funktionieren, wie sie sollten.

Trotz dieses Vorteils, wird in vielen Projekten nicht ausreichend oder nicht effizient getestet, was zu verminderter Softwarequalität führt. Die Gründe dafür sind vielfältig und umfassen z.B. Zeitdruck (es werden lieber weitere Features implementiert, als bestehende zu testen), fehlende Toolunterstützung, mangelnde Kenntnisse über das Testen, fehlendes Personal oder betriebswirtschaftliche Gründe [Kasurinen2009], [Rooksby2009].

Wenn die Entwickler selbst keine Tests erstellen können oder wollen, besteht die Möglichkeit die Entwicklung dieser auszulagern. In diesem Fall kann ein darauf spezialisiertes Testteam alle Testtätigkeiten übernehmen. Studien zeigen zudem, dass es durchaus vorteilhaft ist, wenn das Testen getrennt von der Entwicklung stattfindet [Kasurinen2009], [Dustin2002]. In vielen Unternehmen wird dies praktiziert [Tervonen2009]. Ein solches Testteam kann zum gleichen Unternehmen gehören. Das Testen kann aber auch bei einem anderen Unternehmen in Auftrag gegeben werden (Outsourcing).

[1] zu deutsch etwa: Softwaretests können sehr effektiv sein, um die Anwesenheit von Fehlern aufzuzeigen, aber sie können nicht deren Abwesenheit belegen.

Während diese Form des Outsourcings in der Literatur bereits relativ gut beschrieben ist, findet man zu einer anderen Form des Outsourcing, nämlich dem Crowdsourcing im Kontext von Softwaretests, relativ wenig Literatur ([Dolstra2013], [Pham2013]).

1.2. Abgrenzung zu Crowdsourcing und Crowdtesting

Crowdsourcing beschreibt nach [Howe2006] die Handlung eines Unternehmens eine Aufgabe, die bisher von unternehmenseigenen Mitarbeitern ausgeführt wurde, in ein unbekanntes (i. Allg.) großes Netzwerk von Menschen auszulagern. Das Auslagern erfolgt dabei in Form eines offenen Aufrufs. Die Aufgabe wird innerhalb des Netzwerks (der Crowd) entweder durch einzelne oder aber kollaborativ durch mehrere Individuen gelöst. Wichtig für das Crowdsourcing ist die Form des offenen Aufrufs an das Netzwerk und das große Netzwerk potenzieller Arbeiter selbst.

Erfolgreich eingesetzt wird das Crowdsourcing unter anderem für das sogenannte Crowdtesting [Wolter2013]. Beim Crowdtesting, wie z.B. auf der Plattform testhub.com eingesetzt, wird eine Software durch eine Crowd auf Fehler getestet. Das Vorgehen der Crowdtester ist dabei entweder explorativ, d.h. die Software wird während der Ausführung systematisch nach Fehlern abgesucht, oder die Tester halten sich an einen genau vorgegebenen Testablauf.

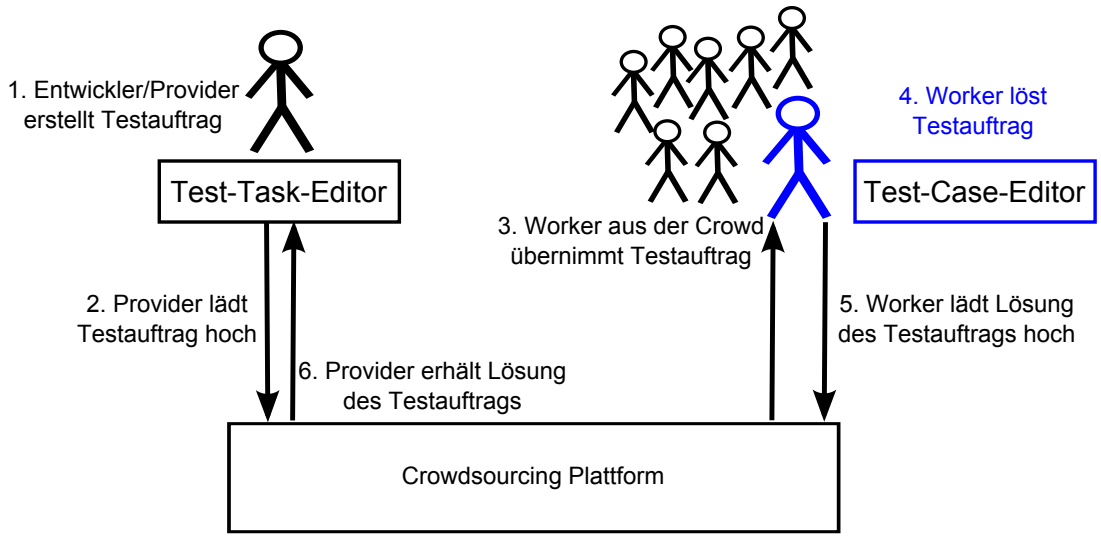

Abbildung 1.1.: Ablauf des Crowdsourcen von Tests. Der in dieser Arbeit betrachtete Teil ist blau markiert.

Diese Masterarbeit beschäftigt sich mit dem Auslagern der Erstellung von automatisierten Tests in die Crowd. In der Crowd sollen also nicht bereits vorhandene Tests ausgeführt werden, sondern es sollen automatisch ablaufende Tests von der Crowd erstellt werden. Der grundsätzliche Ablauf sieht dabei wie folgt aus (siehe auch Abbil-

dung 1.1):

1. Der Entwickler erstellt mithilfe des Test-Task-Editors einen Testauftrag, auch Test-Task genannt.
2. Der Entwickler lädt den Testauftrag auf eine Crowdsourcing Plattform hoch. Er wird damit zum Test-Task-Provider, im Folgenden auch kurz Provider genannt.
3. Ein angemeldeter Benutzer der Crowdsourcing Plattform entdeckt den Testauftrag und entscheidet sich dazu, diesen zu übernehmen. Er wird damit zu einem Worker innerhalb der Crowd.
4. Der Worker erstellt mithilfe des Test-Case-Editors eine Lösung für den Testauftrag in Form der geforderten Tests.
5. Der Worker lädt seine Lösung des Testauftrags auf die Crowdsourcing Plattform hoch.
6. Der Provider wird über die eingereichte Lösung des Testauftrags benachrichtigt. Sieht dieser die Lösung als valide an, bindet er den oder die enthaltenen Tests in sein Projekt ein.

Diese Arbeit beschäftigt sich ausschließlich mit dem vierten Schritt dieses Workflows.

1.3. Vision

Um die in dieser Arbeit untersuchte crowdbasierte Testerstellung ein wenig anschaulicher zu machen und um die einzelnen Rollen näher zu beleuchten, möchte ich zunächst ein fiktives, aber durchaus realistisches, Praxisbeispiel geben. Im Anschluss daran werde ich erläutern, wo bisher Probleme existieren und wie sich die konkrete Aufgabenstellung dieser Arbeit daraus ableitet. Im Laufe dieser Arbeit werde ich das Beispiel wieder aufgreifen, um daran konkrete Probleme oder Sachverhalte zu erläutern.

> Peter ist ein erfahrener freiberuflicher Softwareentwickler und entwickelt für seinen neuesten Kunden einen genau auf dessen Bedürfnisse optimierten Webshop. Eine der ersten Komponenten, die er entwickelt, ist das Modul zur Registrierung neuer Benutzer. Peter ist es eigentlich gewohnt, nicht nur den Programmcode zu schreiben, sondern parallel dazu auch Tests zu entwickeln, mit denen er seine bisherige Arbeit prüfen kann. Dieses Mal möchte er aber nicht alle Tests selber entwickeln, sondern für einige einen neuen Ansatz ausprobieren: das Crowdsourcen von Tests.
>
> Peter öffnet in seiner Entwicklungsumgebung ein Plugin, mit dem er Tests zu seinem Code auf einer Crowdsourcing Plattform anfordern kann. Das Plugin hat Peters Arbeit im Hintergrund überwacht und schlägt direkt nach dem Öffnen vor, für das Registrierungsmodul Tests anzufordern. Nach Auswahl dieses Vorschlags sind die meisten notwendigen Felder im Anforderungsformular schon vorausgefüllt. Was noch fehlt, ist eine Beschreibung von dem, was die Tests prüfen sollen. Nach dem Peter dies beschrieben hat, sendet er den Testauftrag ab.

Kurze Zeit später entdeckt Willy Peters Testauftrag auf der Crowdsourcing Plattform. Willy ist ebenfalls Softwareentwickler und verfügt über langjährige Erfahrung in der Entwicklung von Web-Software. Er liest sich Peters Beschreibung durch und beschließt, sich das Projekt einmal anzusehen. Mit seinem Browser lädt er den Testauftrag, den Peters Plugin erstellt hat, von der Crowdsourcing Plattform herunter und öffnet sie mit seiner Entwicklungsumgebung. Diese wechselt nun in eine speziell für das crowdsourcingbasierte Testen optimierte Ansicht. Die Dateien, die nach Peters Ansicht für die Tests relevanten Quellcode enthalten, sind bereits in verschiedenen Tabs geöffnet. Im Projektexplorer sind diese Dateien auch speziell markiert. In einem Panel am Bildschirmrand liest Willy nun noch einmal Peters Beschreibung durch. Danach beginnt er sich die für ihn relevanten Teile des Projekts näher anzusehen.

Nach einer kurzen Einarbeitungsphase, beginnt Willy die von Peter geforderten Tests zu entwickeln. Hilfreich findet er einen am Bildschirmrand eingeblendeten Test, den Peter bereits entwickelt hat. Mit diesem Test im Blick, ist es für Willy nun keine große Herausforderung mehr, die geforderten Tests zu ergänzen. Kurze Zeit später sind diese geschrieben und Willy klickt in der Symbolleiste seiner IDE auf den Button zum Ausführen der neuen Tests. Gedanken über die Projektkonfiguration oder noch fehlende Bibliotheken, die für das Ausführen benötigt werden, muss sich Willy zu diesem Zeitpunkt nicht machen. Schließlich war in der Datei, die er heruntergeladen hat, alles Nötige enthalten. Einige Sekunden lang werden die neuen Tests nun ausgeführt, bevor das Fenster mit den Testergebnissen rot aufleuchtet. Einer der Tests ist fehlgeschlagen. Ein leeres Kennwortfeld wird von Peters Software noch nicht richtig behandelt.

Willy prüft noch einmal seine Tests, doch diese scheinen ihm korrekt zu arbeiten. Er schreibt noch einen kurzen Kommentar darüber, was er zum Projekt hinzugefügt hat und sendet dann, direkt aus seiner Entwicklungsumgebung heraus, die neuen Tests zurück an die Crowdsourcing Plattform.

Das vorangehende Beispiel zeigt, wie das crowdbasierte Erstellen von Tests aussehen kann und wie die einzelnen Akteure in einem solchen Szenario handeln. Da wäre zunächst einmal Peter, der im Beispiel die Rolle des Providers übernommen und Testaufträge erstellt hat. Er möchte andere Personen dazu bewegen, für sein Programm Tests zu erstellen. Eine solche Person war im Beispiel Willy, der als Worker agiert hat. Damit der Worker seine Aufgabe erfüllen kann, benötigt er eine Reihe von Informationen. Dazu zählt unter anderem z.B. welche Komponente der Software getestet werden soll. Im Beispiel war dies das Registrierungsformular. Diese Informationen können natürlich nur vom Provider zur Verfügung gestellt werden. Daraus resultiert ein gewisser Interessenskonflikt.

Auf der einen Seite möchte der Provider für das Erstellen eines Testauftrags so wenig Aufwand wie möglich betreiben. Wenn er erst eine Stunde lang eine Beschreibung erstellen muss, was genau der Worker testen soll und welche Schritte dafür nötig sind und dann zusätzlich noch eine Liste mit erforderlichen Bibliotheken erstellen muss, dann wird er sich diese Arbeit sparen. In der Zeit könnte er nämlich genauso gut selbst den entsprechenden Test entwickeln. Auch wenn, wie im Beispiel, nur zusätzliche Tests angefordert werden, sollte der Aufwand so niedrig wie möglich sein, da es sonst tendenziell vermieden wird, solche Aufträge zu erstellen.

Auf der anderen Seite benötigt der Worker gewisse Informationen. Er bekommt ein vollkommen unbekanntes Projekt und soll dafür einen Test entwickeln. Hat er nur wenige oder gar keine Informationen über das Projekt zur Verfügung, benötigt er unter Umständen sehr viel Zeit, um sich in das Projekt einzuarbeiten. Dies wäre aber gar nicht notwendig. Im Beispiel war Willy ein erfahrener Webentwickler. Er weiß, wie ein Registrierungsformular funktioniert und auf welche Feinheiten zu achten sind. Um dieses Wissen sinnvoll in das genannte Projekt einzubringen, muss er nicht wissen, über welches ORM-Framwork die Datenbank angesprochen wird oder welche Datenstrukturen für das Warenkorbmodul verwendet werden. Was er aber sehr wohl wissen muss, ist z.B. welche Methode die Registrierungsdaten entgegennimmt und wo dieser Code im Projekt zu finden ist und natürlich, was seine konkrete Aufgabe ist.

Wenn das Erstellen von Tests in der Crowd funktionieren soll, müssen also gewisse Grundbedingungen erfüllt sein:

- Für den Provider muss der Aufwand, einen Testauftrag zu erstellen, wesentlich geringer sein, als der Aufwand, den Test selbst zu erstellen.
- Für den Worker muss der weitaus größte Teil seiner Arbeit darin bestehen, einen oder mehrere konkrete Tests zu erstellen.

Im Beispiel waren beide Bedingungen erfüllt. Möglich wurde dies durch speziell auf das crowdbasierte Testen optimierte Entwicklungsumgebungen, sowohl auf Seite des Providers, als auch auf der Seite des Workers. Eine solche Entwicklungsumgebung existierte bisher aber nicht.

1.4. Aufgabenstellung und Ziele

Im Rahmen dieser Arbeit sollen Konzepte entwickelt werden, die es einem Worker in der Crowd erleichtern, automatisierte Tests für ein ihm unbekanntes Projekt zu entwickeln. Das Hauptaugenmerk liegt dabei auf der möglichst geringen Einstiegshürde für den Worker. Dieser soll in die Lage versetzt werden, sein Fachwissen zur Lösung eines Testauftrages einbringen zu können, ohne dass er sich mit nicht relevanten Projektkomponenten auseinandersetzen muss. Darüber hinaus sollen die erarbeiteten Konzepte prototypisch in der Entwicklungsumgebung Eclipse umgesetzt werden, sodass diese zum Bearbeiten von Testaufträgen geeignet ist. Die IDE soll so gestaltet werden, dass für die konkrete Aufgabe relevante Informationen hervorgehoben werden. Dazu zählt zum Beispiel, dass von dem Testauftrag betroffener Quelltext einfach zu erkennen ist oder dass auf bereits vorhandene Tests, die nützlich sein könnten, hingewiesen wird. Notwendige Testwerkzeuge werden ebenfalls eingeblendet, während Elemente, die für die Bearbeitung der Testaufgabe irrelevant sind, ausgeblendet werden. Auch die Installation oder Konfiguration von für die Ausführung notwendiger Komponenten, wie z.B. (Test-)Bibliotheken, soll vollkommen automatisiert ablaufen. Der Worker soll direkt mit der Bearbeitung des Testauftrags beginnen können.

Die Arbeit unterteilt sich in zwei Bereiche. Im ersten Teil wird es darum gehen, Konzepte zu entwickeln, die es einem Worker ermöglichen, aufgabenfokussiert zu arbeiten.

Ich werde dazu zunächst analysieren, wie der Testprozess an sich aussieht und durch wen die Aufgabe des Testens übernommen werden kann. Dabei werde ich darauf eingehen, wo die speziellen Herausforderungen liegen und welche besonderen Maßnahmen für das erfolgreiche Testen jeweils notwendig sind. Im Anschluss daran werde ich untersuchen, wie verschiedene Entwicklungsumgebungen den Tester bei seiner Arbeit unterstützen und welche Konzepte dabei jeweils verfolgt werden. Aus den bis dahin gewonnenen Informationen werde ich dann Konzepte ableiten, die das aufgabenfokussierte Arbeiten im Umfeld der crowdbasierten Erstellung von automatisierten Tests erleichtern. Im zweiten Teil der Arbeit werde ich erläutern, wie sich die entwickelten Konzepte in die Entwicklungsumgebung Eclipse integrieren lassen. Abschließend werde ich untersuchen, inwieweit die angepasste Eclipse-Umgebung zum Lösen von Testaufträgen durch einen Worker geeignet ist.

Um den zeitlichen Rahmen dieser Arbeit nicht zu überschreiten, werde ich mich ausschließlich auf das Crowdsourcen von GUI-Tests im Kontext von Webanwendungen beschränken. Konkret wird die Implementierung Play!-Anwendungen (ab Version 2.0)[2] und Selenium-Tests[3] unterstützen. Die Konzepte lassen sich grundsätzlich aber auch auf andere Frameworks und teilweise auch weitere Testarten übertragen.

Der Fokus dieser Arbeit wird vollkommen auf dem Worker liegen. Wie der Provider von Testaufträgen in seiner Arbeit unterstützt werden kann, wird in der parallel durchgeführten Masterarbeit „Konzept und Implementierung eines heuristischen Editors zum Erstellen von Testaufgaben auf einer Crowdsourcing Plattform“ [Riedel2014] von Benjamin Riedel untersucht. Das dort vorgestellte Eclipse-Plugin, der sogenannte Test-Task-Editor, liefert alle Daten, die die hier vorgestellte Eclipse-Version zur Einrichtung eines Projekts bzw. Testauftrags benötigt. Die Schnittstellen dazu wurden gemeinschaftlich erarbeitet und werden im Laufe der Arbeit noch ausführlich vorgestellt. Wie eine Crowdsourcing Plattform grundsätzlich aussehen muss, damit die hier beschriebenen Prozesse funktionieren, ist nicht Teil dieser Arbeit. Ebenso wenig werde ich darauf eingehen, unter welchen ökonomischen Randbedingungen das hier beschriebene Crowdsourcen sinnvoll ist.

[2] Play! ist ein auf Java und Scala basierendes Framework zur Erstellung von Webanwendungen. Siehe auch `http://www.playframework.com/`

[3] Selenium ist ein Framework für die Erstellung von GUI-Tests in Webanwendungen. Siehe auch `http://docs.seleniumhq.org/`

2. Analyse des Testprozesses

Im Unternehmensumfeld gibt es verschiedene Möglichkeiten, wie das Testen von Software organisiert werden kann. Während das unternehmensinterne Testen und das Outsourcen von Tests weit verbreitet sind, stellt das Crowdsourcen von automatisierten Tests ein neues Konzept dar. Die Organisationsformen unterscheiden sich dabei teilweise erheblich voneinander. Dies reicht von der Arbeitsweise der Entwickler über die Projektplanung bis hin zur Vertragsgestaltung. In diesem Kapitel möchte ich die Gemeinsamkeiten und Unterschiede aufzeigen und so die verschiedenen Organisationsformen voneinander abgrenzen.

2.1. Unternehmensinternes Testen

Software unternehmensintern zu testen, stellt den Regelfall dar. Dabei kümmern sich Mitarbeiter des eigenen Unternehmens um den gesamten Testprozess. Wie dieser organisiert wird, ist sehr variabel und unterscheidet sich von Unternehmen zu Unternehmen. Wie Zuständigkeiten für das Testen in Unternehmen festgelegt sind, lässt sich laut [Deak2013] grundsätzlich in vier verschiedene Kategorien unterteilen.

- Die Entwickler sind selbst für Tests verantwortlich.
- Innerhalb des Entwicklungsteams gibt es spezielle Tester, die nur für das Testen verantwortlich sind.
- Es gibt eine spezielle Abteilung, die sich ausschließlich um Testaufgaben kümmert.
- Eine Mischung aus den beiden vorangehenden Möglichkeiten.

Die gewählte Organisationsform hat großen Einfluss auf die Testtätigkeiten. So wird in [Dustin2002] darauf hingewiesen, dass die Testumgebung unbedingt von der Entwicklung getrennt sein sollte und dies zahlreiche Vorteile hat. Auch die Autoren von [Kasurinen2009] weisen darauf hin. Vorteilhaft ist u.a., dass die Entwickler bei getrennten Umgebungen während eines Testlaufs keine Veränderungen am Code durchführen können, der sich auf die Testausführung auswirkt. Dies könnte sonst zu verfälschten Testergebnissen führen und kann auch erhöhten Aufwand bedeuten. Entwickler würden unter Umständen an Tester zurückmelden, dass sie bestimmte Fehler nicht reproduzieren konnten und in der Folge weiterer Aufwand betrieben wird, um diese nachzustellen, während diese aber in der Zwischenzeit unbemerkt gefixt wurden.

Nachteilig ist es, wenn Entwickler ihren eigenen Code testen müssen. Das Ziel eines Entwicklers ist es, funktionierenden Code zu entwickeln. Daher wird er wird dann

kaum Freude dabei haben, in diesem teilweise langwierig erarbeiteten Ergebnis Fehler aufzuspüren. Außerdem wird er wahrscheinlich keine Aspekte testen, an die er beim Entwickeln des Codes überhaupt nicht gedacht hat.

Ein weiterer Vorteil der Trennung von Entwicklung und Test liegt im Fachwissen, das spezielle Tester besitzen. Ein Entwickler, dessen Haupttätigkeit das Testen darstellt, wird über die Zeit einen teilweise erheblichen Erfahrungsvorsprung gegenüber „normalen“ Entwicklern aufbauen. Dies führt dazu, dass ein spezieller Tester schneller bessere Tests erstellen kann. Er wird effizienter eine notwendige Testumgebung einrichten können und wird detaillierteres Fachwissen über Testframeworks besitzen. Dies ist offenbar besonders wertvoll, da eine Befragung von 65 australischen Softwareunternehmen ergeben hat, dass die größte Schwierigkeit beim Einführen neuer Methoden und Techniken fehlendes Fachwissen ist [Ng2004]. Die Umfrage hat zudem ergeben, dass das Training von Personal als besonders teuer und zeitintensiv angesehen wird. Dies kann als ein weiterer Grund angesehen werden, warum es effizienter ist, dediziertes Testpersonal einzusetzen. So müsste sich ein Entwickler beim Erwerb von neuen Kenntnissen bezüglich des Testens relativ lange mit einer Tätigkeit auseinandersetzen, die eigentlich nicht zu seiner Kernaufgabe gehört. Zudem haben dedizierte Testentwickler den Vorteil, dass sie sich projektübergreifend einsetzen lassen.

Trotz dieser Vorteile führten laut [Haberl2012] 1997 nur in etwa einem Viertel der Unternehmen speziell ausgebildete Tester die vorgesehenen Tests durch. 2011 war dieser Wert auf ca. 77% gestiegen, was auch durch [Garousi] bestätigt wird, wo im Jahre 2009 dies bei rund zwei Drittel der Unternehmen festgestellt werden konnte. In der Umfrage [Haberl2012] gaben jedoch rund 86% der Befragten an, dass in ihrem Unternehmen auch Entwickler mit Testen beschäftigt sind.

2.2. Ausgelagertes Testen

Unternehmen tendieren in der heutigen Zeit dazu, Aufgaben, die nicht unmittelbar zum Kernkompetenzbereich gehören, auszulagern. Dies ist im Bereich der Softwareentwicklung nicht anders. Auch hier werden verschiedene Aufgaben an andere Unternehmen ausgelagert. Dazu gehören neben der Entwicklung von einzelnen Softwarebestandteilen, wie z.B. Treibern, auch einzelne Aktivitäten im Entwicklungsprozess, die von Unternehmen nicht mehr immer selbstständig durchgeführt werden. Eine Aktivität, die von Unternehmen bevorzugt ausgelagert wird, ist das Testen [Tervonen2009]. Man spricht dabei auch von Outsourcing. Griffiths definiert dieses in [Griffiths2001] als „the strategic use of outside resources to perform activities traditionally handled by internal staff and resources“.[1] Die Gründe dafür sind vielfältig. Er nennt hier u.a. die Reduktion von Kosten, das Einsparen von internen Ressourcen und das Aufteilen von Risiko.

Nicht nur aus diesen unternehmerischen Gründen macht das Outsourcen von Testaktivitäten Sinn. Wie im vorherigen Abschnitt bereits erläutert wurde, ist es effizienter die

[1] zu deutsch etwa: die strategische Verwendung von externen Ressourcen um Aktivitäten auszuführen, die traditionell durch interne Mitarbeiter und Ressourcen ausgeführt werden

Entwicklung einer Software und deren Testen voneinander zu trennen, als beides von den gleichen Entwicklern ausführen zu lassen. Beim Outsourcen wird diese Trennung nun noch weiter geführt. Entwickler und Tester gehören hier nicht mehr zum gleichen Unternehmen, sondern sind bei verschiedenen angestellt. Diese noch striktere Trennung wirkt sich natürlich auch auf die Kompetenzen der Tester aus. In einem Unternehmen, das ausschließlich auf das Testen von Software spezialisiert ist, wird man in der Regel natürlich mehr Expertise bezüglich Softwaretests vorfinden, als bei einem Unternehmen, dessen Hauptaufgabe das Entwickeln von Software ist und wo das Testen eine Aktivität neben vielen anderen ist. In dieser Hinsicht ist das Outsourcen von Softwaretests natürlich der Qualität zuträglich.

Allerdings ergeben sich durch das Outsourcen selbstverständlich auch einige Nachteile. Um überhaupt testen zu können, benötigt man eine Referenz (Sollwerte) wogegen getestet werden soll. Dies sollte in erster Linie die Dokumentation sein, und falls diese nicht ausreichend genau oder detailliert genug ist, stellt der Kunde die letzte Instanz dar. Der Kunde eines Unternehmens, das auf Softwaretests spezialisiert ist, ist jedoch nicht der eigentliche Auftraggeber der zu entwickelnden Software, sondern das Unternehmen, das die Software entwickelt. Also ist es die Aufgabe dieses Unternehmens, die nötige Referenz zur Verfügung zu stellen und ggf. vorhandene Fragen zu beantworten. Die Autoren von [Shah2011] berichten jedoch, dass fehlende Informationen kaum geliefert werden. So seien In-house-Entwickler (Entwickler des Unternehmens das outsourced) teilweise nicht bereit, überhaupt Hilfestellung zu bieten. Außerdem bestehe oftmals das Problem, überhaupt die für einen Sachverhalt zuständige Person zu ermitteln. Auch Prioritäten würden nur unzureichend übermittelt, obwohl Zeitdruck ein beständiger Faktor in Projekten sei.

Weitere Probleme kommen hinzu, wenn das Testen in andere Länder, wie z.B. Indien, verlagert wird; was gängige Praxis ist. Man spricht dann von Offshoring. In diesem Fall unterscheiden sich die kulturellen Hintergründe des Entwicklungs- und des Testteams. So wird in [Shah2011] von einem Fall berichtet, bei dem ein Unternehmen 100 Tests für eine Software angefordert hat. Ein zuständiger Offshore-Entwickler merkte gegenüber den Autoren der Studie an, dass es wesentlich sinnvoller gewesen wäre, lediglich fünf bis sechs Tests zu entwickeln, statt der 100. Er merkte weiterhin an, dass die Entwickler in dem Unternehmen aber nicht die Autorität hätten, Anforderungen des Auftraggebers infrage zu stellen. Eine direkte Kommunikation wird durch Offshoring ebenfalls erschwert, da sich die potenziellen Gesprächspartner in unterschiedlichen Zeitzonen befinden und verschiedene Sprachen sprechen.

Hinzu kommen noch unternehmerische Probleme. Änderungen an Verträgen, die etwa durch sich ändernde Anforderungen oder fehlende Abschnitte im Vertrag nötig werden, ziehen meist Kosten nach sich, die ohne Outsourcing so nicht auftreten würden [Tervonen2009]. Das bedeutet, dass ein gewisser Grad an Flexibilität, was den Inhalt des Projekts betrifft, dadurch verloren geht oder erst wieder monetär erkauft werden muss. Ein weiteres Problem ist Vertraulichkeit. Lagert man das Testen aus und macht dem Vertragspartner den eigenen Quellcode zugänglich, was z.B. für White-Box-Tests auch zwingend notwendig ist, so besteht potenziell die Gefahr, dass das eigene Know-How in die Hände von Konkurrenzfirmen gelangt. Zwar versuchen sich die Unternehmen

mit Verschwiegenheitserklärungen davor zu schützen, jedoch existieren in manchen Ländern keine ausreichend starken Gesetze. Auch geopolitische Risiken spielen eine Rolle [Tervonen2009]. Politische Instabilitäten, die z.B. zu Streiks und damit zu Verzögerungen führen, finden ebenso Berücksichtigung, wie die lokale Infrastruktur (z.B. kommt es regelmäßig zu Stromausfällen).

2.3. Crowdsourcing von automatisierten Tests

Im Gegensatz zu den bisher genannten Organisationsformen des Testens, ist das Crowdsourcen von automatisierten Testfällen bisher nicht besonders umfangreich untersucht worden. In [Dolstra2013] wird beschrieben, inwieweit GUI-Tests von der Crowd ausgeführt werden können. Allerdings beschränken sich die Autoren darauf, zu untersuchen, wie man GUI-Tests in der Crowd von Workern per Hand ausführen lassen und die Ergebnisse für Continous Integration nutzen kann. Einzig in [Pham2013] wird die Idee beschrieben, automatisierte Tests von der Crowd erstellen zu lassen. Aber auch in dieser Arbeit findet keine Abgrenzung zu den oben genannten Testformen statt.

Der größte Unterschied zum unternehmensinternen und zum ausgelagerten Testen besteht wohl darin, dass die Testfallentwickler beim Crowdsourcing keine längerfristige Bindung an das Projekt besitzen. Daraus entsteht eine ganze Reihe von technischen und organisatorischen Vor- und Nachteilen gegenüber den anderen Testformen. Zunächst einmal hat ein Worker in der Crowd keinerlei Wissen über den inneren Aufbau der Software. Dies hat wiederum sowohl positive, als auch negative Folgen. So wird ein Worker versuchen, ein gegebenes Problem so einfach wie möglich zu lösen. Da er nicht durch Projektinterna „vorbelastet“ ist, entwickelt er möglicherweise eine einfachere und bessere Teststruktur als projektgebundene Entwickler dies tun würden.

Damit dies aber gelingen kann, müssen zunächst einige Hürden überwunden werden. In einem ersten Schritt muss ein Worker überhaupt in die Lage versetzt werden, etwas für das Projekt entwickeln zu können. Dafür benötigt er eine Entwicklungsumgebung. In den anderen Szenarien würde sich jeder Entwickler selbst seine Umgebung einrichten. Bei Projekten, in denen die Infrastruktur für das Testen umfangreicher ausfällt, z.B. wenn die Software mit Fremdsystemen zusammenarbeiten muss (was im Unternehmensumfeld eher die Regel als die Ausnahme ist), würden sich mehrere Mitglieder des Testteams um die Einrichtung der nötigen Infrastruktur kümmern. Im Crowdsourcingszenario kann dies aber kaum vom Worker verlangt werden, da die Einrichtungszeit unter Umständen wesentlich länger wäre, als die Zeit für das Erstellen des eigentlichen Tests. Dies könnte einer der Hauptgründe sein, warum bisher automatisierte Tests nicht in die Crowd ausgelagert wurden.

Ein wesentlicher limitierender Faktor für die crowdbasierte Testentwicklung ist dabei nicht nur die Einrichtungszeit, sondern auch die dem Worker zur Verfügung stehenden Ressourcen. So ließe sich z.B. ein Lasttest kaum in die Crowd verlagern, da man nicht davon ausgehen sollte, dass ein einzelner Worker über ausreichende Kapazitäten verfügt, diesen auszuführen. Teilweise führt dies selbst in kleineren Unternehmen zu Problemen [Rooksby2009].

Ist die Entwicklungsumgebung des Workers eingerichtet, so fehlt diesem aber immer noch das Wissen über die interne Struktur der Software. Ähnlich wie bei der Entwicklungsumgebung, kann ihm auch hier nicht zugemutet werden, sich alles Nötige selbst zu erarbeiten, wie es etwa im Unternehmensumfeld üblich wäre. Da bisher keine Strukturen existieren, einen Worker bei dieser Arbeit zu unterstützen, kann dies als weiterer wesentlicher Grund angesehen werden, warum das Crowdsourcen von Tests bisher nicht praktiziert wird.

Neben diesen eher technischen, gibt es noch eine Reihe weiterer organisatorischer Aspekte, die beim Crowdsourcing eine Rolle spielen. Im unternehmerischen Bereich spielt die Zeitplanung eines Projekts eine zentrale Rolle. Neben der Verfügbarkeit von Ressourcen (Personal, Ausrüstung), müssen oftmals auch bestimmte Termine eingehalten werden. Aus diesem Grund werden die verschiedenen Aktivitäten zu Beginn eines Projekts geplant. Dies umfasst bei einer sorgfältigen Planung natürlich auch die Testaktivitäten. Verlagert man aber nun die Erstellung von Tests in die Crowd, so hat man keinerlei Garantien, in welchem Zeitraum ein Testtask bearbeitet wird oder ob dieser überhaupt gelöst wird. Selbst wenn Lösungen eingeschickt werden, müssen diese noch von eigenen Mitarbeitern auf ihre Richtigkeit überprüft werden. Andererseits fällt natürlich der Aufwand der eigentlichen Testentwicklung weg, und es kann dadurch unter Umständen erhebliche Zeit eingespart werden. Dies trifft insbesondere dann zu, wenn die Entwickler oder Tester nur geringe Kenntnisse bezüglich der verwendeten Testtechnik haben.

Die Crowd bietet allerdings auch eine gewisse Flexibilität. In Phasen, in denen mehr Tests erstellt werden müssen, können relativ leicht mehr Testtasks vergeben werden. Müsste diese Arbeit vollständig von eigenem Personal erledigt werden, müssten entsprechend mehr Mitarbeiter beschäftigt werden, die vielleicht aber nicht die gesamte Zeit über benötigt würden. In der Praxis greifen Unternehmen in Zeiten mit hoher Auslastung auf externe Mitarbeiter zurück [Deak2013]. Allerdings erfordert dies auch eine gewisse Vorausplanung und nicht immer kann der Testaufwand im Vorfeld richtig abgeschätzt werden.

Ein weiterer Aspekt, der im Crowdsourcingumfeld auf jeden Fall beachtet werden muss, ist die Offenlegung von Know-How. Wenn man sich nicht vollständig auf Black-Box-Tests beschränkt, ist es notwendig, zumindest Teile des Programmcodes dem Testentwickler zugänglich zu machen. Ist dieser in der Crowd beheimatet, so hat prinzipiell jeder Zugriff auf den dort hinterlegten Code. Zudem unterliegen die Entwickler, die wirklich einen oder mehrere Tests beisteuern, sich also mit der zu testenden Software auseinandergesetzt haben, keiner Verschwiegenheitsverpflichtung. Dies wäre bei unternehmenseigenen Entwicklern oder Entwicklern eines mit dem Testen beauftragten Unternehmens hingegen meist der Fall. Die fehlende vertragliche Bindung hat aber auch einen Vorteil. Es müssen vor Beginn des Projekts keine vertraglichen Regelungen getroffen werden, die im Falle von Änderungen unter Umständen Kosten nach sich zögen, wie es etwa beim Outsourcing der Fall ist.

3. Analyse existierender Tools

In diesem Kapitel möchte ich darauf eingehen, wie verschiedene Entwicklungsumgebungen (Integrated Development Environment, kurz IDE) versuchen, die Entwickler bei ihren Testtätigkeiten zu unterstützen. Dazu werde ich zunächst einen Überblick darüber geben, welche konkreten Funktionen die einzelnen IDEs jeweils besitzen und worin sie sich voneinander unterscheiden. Konkret habe ich mich für Eclipse, Netbeans und IntelliJ als die wohl bekanntesten und meist verwendeten IDEs im Java-Umfeld, sowie für Visual Studio entschieden. Die IDEs sind für einen großen Teil der am meisten verwendeten Programmiersprachen geeignet (vgl. [tiobe2013] und [redmonk2013]) und sind relativ bekannt. Dabei werde ich nur auf direkt in die IDE integrierte Funktionen eingehen. Die ausgewählten Versionen der IDEs sind jeweils die umfangreichsten, die direkt vom Hersteller bzw. Entwickler angeboten werden. Alle hier genannten IDEs unterstützen grundsätzlich Plugins, über die zahlreiche weitere Funktionen nachgerüstet werden können. Eine vollständige Betrachtung ist aufgrund der großen Anzahl nicht möglich. Erwähnt sei an dieser Stelle noch einmal, dass Eclipse die IDE ist, die im praktischen Teil der Arbeit auf die Bedürfnisse des Crowdsourcings angepasst wurde.

3.1. Eclipse IDE for Java EE Developers 4.3 (Kepler)

„Eclipse IDE for Java Developers“ bietet nur wenig Unterstützung für den Testprozess selbst. Speziell dafür enthält es standardmäßig lediglich das Plugin „JUnit“ zur Erzeugung und Ausführung von Unit Tests. Über einen Assistenten können neue Tests für Version 3 und 4 von JUnit angelegt werden. Neben den technisch notwendigen Abfragen für einen Testnamen und den Speicherort, kann noch ausgewählt werden, ob Methodenrümpfe (Stubs) für den Testaufbau und Testabbau (setUp, tearDown; jeweils auf Test und Klassenebene) generiert werden sollen. Weiterhin kann die „Class under Test“, also die Klasse, die getestet werden soll, ausgewählt werden. Wird diese angegeben, ist es möglich, einzelne Tests für die Methoden der Klasse generieren zu lassen. Dabei werden lediglich die Methodenrümpfe generiert. Darüber hinaus können auch sogenannte Test-Suites für JUnit generiert werden. Diese fassen die Ausführung mehrerer Testklassen zusammen. Welche Klassen eine Suite umfassen soll, kann beim Erstellen über ein Menü festgelegt werden. Ist die JUnit-Bibliothek bisher nicht im Projekt eingebunden, so fragt Eclipse beim Erstellen eines entsprechenden Tests nach, ob diese eingebunden werden soll.

Nach dem Erstellen der Testklassen, können die Tests allerdings noch nicht ausgeführt werden. Dafür muss zunächst noch eine sogenannte „Run Configuration“ erstellt werden. Dabei hat der Nutzer die Wahl, alle Tests im Projekt auszuführen, Tests in bestimmten Verzeichnissen oder bestimmte Methoden in einer einzigen Klasse aus-

zuführen.

Weiterhin steht noch eine View (ein frei in der IDE verschiebbares Fenster) zur Verfügung, die über Testergebnisse Auskunft gibt. Diese wird eingeblendet, sobald einer der erstellten JUnit-Tests in der IDE ausgeführt wird. In der View werden alle ausgeführten Testklassen und die dazugehörigen Tests angezeigt, sowie jeweils das Ergebnis und die Dauer der Ausführung. Ist ein Test fehlgeschlagen, so kann zusätzlich für jeden Test der Stacktrace angezeigt werden. Außerdem bietet das JUnit-Plugin eine Anbindung an das Plugin „Mylyn“.

Mylyn

Das Mylyn-Plugin hat nichts unmittelbar mit dem Testen an sich zu tun und kann auch unabhängig von Testtätigkeiten verwendet werden. Da Mylyn aber eine aufgabenfokussierte Benutzeroberfläche zum Ziel hat und dies sehr gut zum Thema dieser Arbeit passt, möchte ich an dieser Stelle detailliert auf das Plugin eingehen.

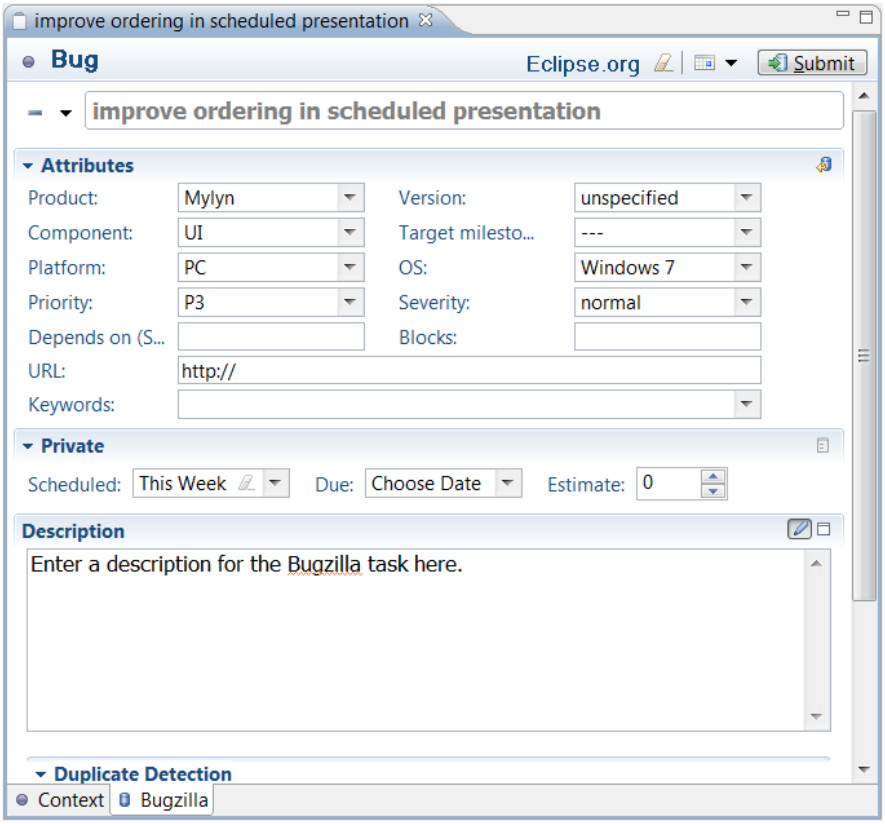

Abbildung 3.1.: Der Mylyn-Task-Editor mit einem offenen Task für ein Bugzilla-Repository. Quelle: `http://wiki.eclipse.org/index.php/Mylyn_User_Guide`, 17.12.2013

Mylyn bietet ein „task-focused interface“ [1]. Zu diesem Zweck lassen sich in Mylyn sogenannte Tasks anlegen. Tasks stellen zu erledigende Aufgaben dar. Zu einem Task gehören ein Name, eine Beschreibung und optional ein Zeitraum, in welchem der Task erledigt werden sollte. Ein Task wird innerhalb eines sog. „Task Repository“ gespeichert. Ein solches Repository kann privat und lokal auf dem Rechner sein, es kann aber auch in einer Kollaborationssoftware angesiedelt sein. Dazu existieren zahlreiche Konnektoren, die unterschiedliche Kollaborationssoftware unterstützen, z.B. Bugzilla, Edgewall Trac, zahlreiche IBM-Programme oder Microsofts TFS. Je nach ausgewähltem Repository, werden einem Task zusätzliche Attribute hinzugefügt. So enthält ein Task für ein Bugzilla Repository, in dem in der Regel Bugs gemeldet werden, beispielsweise Auswahlfelder für das betreffende Produkt, die dazugehörige Komponente, eine Priorität oder den Schweregrad eines Fehlers (vgl. Abbildung 3.1). Zusätzlich dazu kann der Task dann auch einer Person zur Bearbeitung zugewiesen werden und es können Kontaktdaten hinterlegt werden.

Einem Task können nun noch Dateien aus dem Projekt hinzugefügt werden. Standardmäßig werden die Dateien hinzugefügt, die geöffnet sind, während der Task angelegt wird. Es lassen sich aber auch über den Project Explorer, den Package Explorer und den Navigator weitere Dateien hinzufügen. Ist ein Task aktiviert, werden im Package und Project Explorer und im Navigator alle Dateien und Verzeichnisse ausgeblendet, die nicht zu diesem Task gehören. Dadurch soll ein aufgabenfokussiertes Arbeiten erreicht werden. Diese Filterung kann in jeder betroffenen View unabhängig voneinander ein- und ausgeschaltet werden.

Darüber hinaus wird auch die Outline-View unterstützt, die eine Übersicht über die in der aktuell geöffneten Klasse vorhandenen Variablen und Methoden gibt. Auch diese können jeweils einzeln als zum Task gehörig markiert werden.

Eine Übersicht über alle Tasks bietet die Task List (vgl. Abbildung 3.2). Hier können die Tasks organisiert werden. Dazu stehen frei wählbare Kategorien und Subtasks zur Verfügung, d.h. für jeden Task können Unteraufgaben definiert werden. Außerdem können Tasks nach Kategorien oder nach ihrer Fälligkeit sortiert werden. Bereits erledigte Tasks lassen sich ausblenden. Des Weiteren steht eine Suche zur Verfügung, mit der sich die eigenen Tasks durchsuchen lassen. Dabei kann auch gezielt innerhalb von Feldern gesucht werden. Zusätzlich dazu lassen sich sogenannte „Queries“ anlegen. Mit einem Query kann ein Repository detailliert nach bereits existierenden Tasks durchsucht werden. Queries werden dabei als spezielle Kategorien angezeigt. Das jeweils zugehörige Repository wird regelmäßig nach passenden Tasks durchsucht und der Benutzer wird ggf. über neue Tasks informiert.

Neue Tasks lassen sich ebenfalls über die Task List erstellen. Zusätzlich dazu ist Mylyn noch in weitere Views integriert. Dazu zählt unter anderem die Problems-View. Hier werden automatisch erkannte Probleme (z.B. Syntaxfehler) aufgelistet. Über einen Unterpunkt im Kontextmenü des jeweiligen Problems lassen sich Mylyn-Tasks anlegen, wobei bei dieser Vorgehensweise die Felder Titel und Beschreibung mit den vorhandenen Informationen vorausgefüllt werden. Eine solche Anbindung besitzt, wie bereits

[1] `http://www.eclipse.org/mylyn/index.php`, 17.12.2013

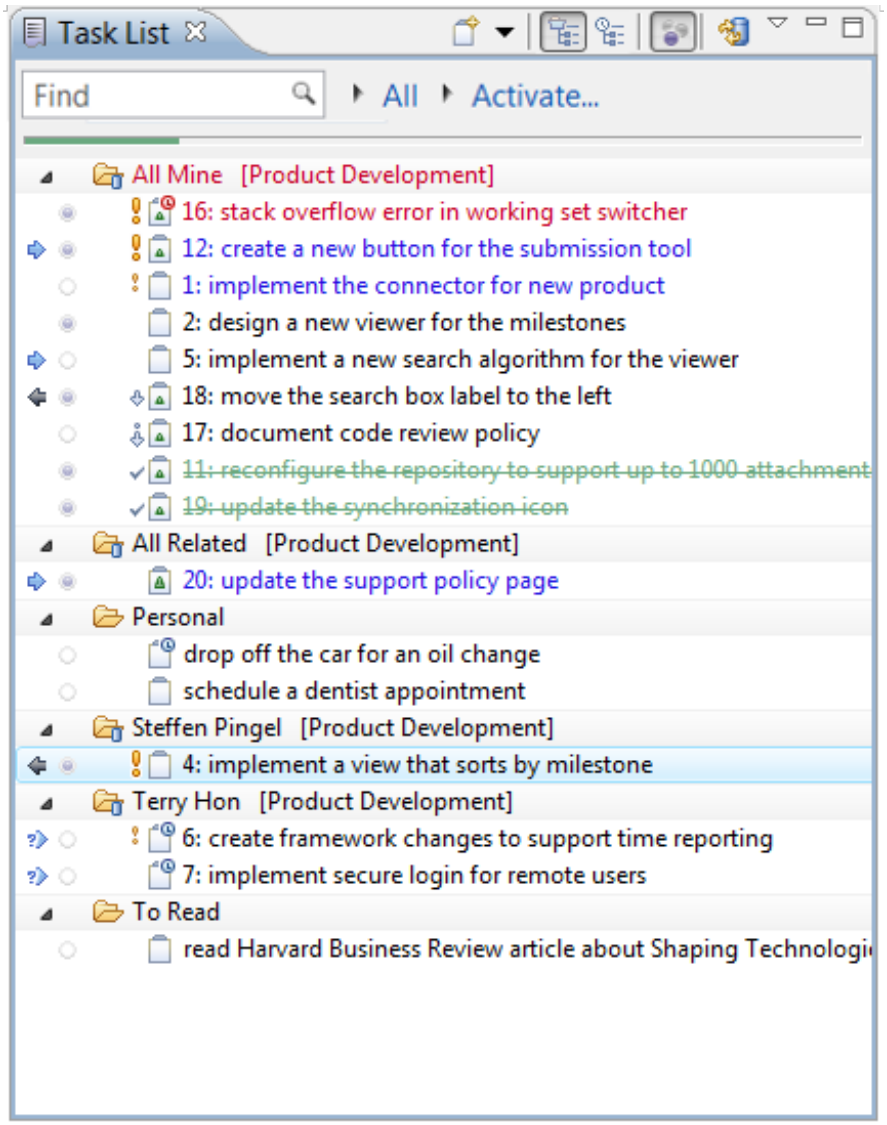

Abbildung 3.2.: Die Mylyn-Task-List mit einer Übersicht über die verschiedenen verfügbaren Task. Quelle: `http://wiki.eclipse.org/index.php/Mylyn_User_Guide`, 17.12.2013

erwähnt, auch JUnit. So lässt sich aus fehlgeschlagenen Testläufen direkt ein Task erstellen, wobei der jeweilige Stack Trace als Beschreibung eingefügt wird.

3.2. NetBeans 7.4

Auch in NetBeans ist standardmäßig das Testframework JUnit eingebunden. Ähnlich wie in Eclipse lassen sich hier auch JUnit-Tests erstellen. Der Assistent zum Erstellen

neuer Tests hat fast den gleichen Umfang wie in Eclipse. Allerdings lassen sich bei NetBeans auch Methodenkörper für die Tests generieren. Dadurch werden schon rudimentäre Grundlagen für den konkreten Test erstellt. So wird jeweils ein Objekt der Testklasse deklariert und zunächst mit null initialisiert. Darüber hinaus wird für diese Instanz die aktuell zu testende Methode aufgerufen. Besitzt diese Methode Parameter, werden für diese ebenfalls Objekte deklariert und mit null initialisiert. Hat die zu testende Methode einen Rückgabewert, so wird zusätzlich noch ein Objekt für das erwartete Ergebnis angelegt und ebenfalls mit null initialisiert. Außerdem wird in dem Fall noch ein assertEquals() angelegt, welches mit dem Rückgabewert der Methode und dem Ergebnisobjekt aufgerufen wird. Zu Beginn jeder Testmethode wird über einen System.out.println()-Befehl ausgegeben, welche Methode getestet wird. Am Ende der Testmethode wird fail() aufgerufen, um zu verhindern, dass ein Test erfolgreich abgeschlossen wird, falls er noch nicht mit echtem Verhalten gefüllt wurde.

Tests können in NetBeans grundsätzlich nur dateiweise ausgeführt werden. Dabei werden immer die Tests der Klasse ausgeführt, die gerade den Fokus im Editor besitzt. Ist dies keine Klasse, die Tests enthält, ist eine Testausführung nicht möglich. Mehrere Tests auf einmal können nur mit einer TestSuite ausgeführt werden.

Darüber hinaus findet sich in NetBeans eine View, die die Testergebnisse, eventuelle Programmausgaben und bei Fehlschlag eines Tests auch den entsprechenden Stack Trace anzeigt. Vom Funktionsumfang und vom Aufbau her gleicht diese View der in Eclipse weitestgehend. Die View wird allerdings noch für ein weiteres Testframework zur Ergebnisanzeige verwendet: TestNG.

TestNG ist laut offizieller Website[2] ein Testframework, das durch JUnit und NUnit[3] inspiriert wurde, aber über deren Funktionsumfang hinausgeht. So ist TestNG speziell darauf ausgelegt Multithreading-Anwendungen zu testen. Außerdem lassen sich Tests über Annotations und über XML konfigurieren. Datengetriebenes Testen wird ebenfalls untersützt. Die Unterstützung in NetBeans gleicht der von JUnit. Jedoch kann beim Erstellen eines neuen Tests diesem keine Klasse zugewiesen werden und somit werden auch klassenspezifische Methoden generiert.

Darüber hinaus liefert NetBeans das Jemmy-Framework mit. Dieses Framework ist speziell für GUI-Tests der NetBeans Platform entwickelt worden. Die NetBeans Platform wiederum ist ein Framework, mit dem auf Swing-Basis Desktop-Applikationen erstellt werden können. Es enthält u.a. Komponenten zum Aufbau einer GUI, Konfigurations- und Speichermanagement oder etwa ein Plugin-System. Es stellt die Grundlage für die NetBeans IDE dar, kann aber auch für andere Projekte verwendet werden. Außer der bereits erwähnten Ergebnis-View für die Anzeige von Testergebnissen, ist aber keine weitere nennenswerte Unterstützung in NetBeans für Jemmy enthalten.

[2] `http://testng.org/doc/index.html`, 17.12.2013

[3] Ein Testframework, welches JUnit im Funktionsumfang weitestgehend gleicht, aber für Microsofts .NET-Umgebung geeignet ist

3.3. IntelliJ IDEA 13.0 Ultimate

Die Unterstützung des Nutzers bei Testtätigkeiten in IntelliJ entspricht vom Umfang her etwa der von Eclipse. Das Anlegen eines Tests unterscheidet sich jedoch von allen anderen hier beschriebenen IDEs. Eine Testklasse muss einer Klasse zugeordnet werden, die man testen möchte. Dementsprechend lässt sich ein neuer Test nur dann anlegen, wenn aktuell eine Klasse geöffnet ist. Dann kann der Nutzer grundsätzlich zwischen fünf nativ unterstützten Frameworks für seinen Test wählen: Groovy JUnit, Spock, JUnit 3 und 4 und TestNG. Wird ein Framework ausgewählt, das bisher nicht in das Projekt eingebunden war, weist die IDE darauf hin und bietet an, dieses einzubinden. Wie auch in Eclipse kann der Nutzer hier Methoden zum Testauf- und Abbau generieren lassen. Außerdem kann angegeben werden, für welche Methoden der zu testenden Klasse einzelne Tests angelegt werden sollen. Sind alle Angaben gemacht, wird eine entsprechende Testklasse angelegt und geöffnet. Von hier aus kann der Nutzer nun jederzeit durch eine entsprechende Tastenkombination oder über einen bestimmten Menüpunkt zu der dem Test zugeordneten Klasse springen.

Ausführbar ist der Test zu diesem Zeitpunkt nicht. Dazu muss vom Benutzer zunächst noch eine „Run Configuration“ angelegt werden. Der Benutzer hat beim Anlegen einer solchen Konfiguration die Wahl, welche Tests ausgeführt werden sollen. Es kann dabei zwischen Packages, Verzeichnissen, Klassen, Methoden und einer beliebigen Zahl von einzeln anzugebenden Klassen gewählt werden. Danach können die Tests ausgeführt werden. Dabei wird am unteren Bildschirmrand ein Menü eingeblendet (falls es nicht vorher schon vorhanden war), das fehlerhafte Tests anzeigt. Bestandene Tests können optional eingeblendet werden. Wie in Eclipse und Netbeans werden auch hier jeweils die einzelnen Tests und der dazugehörige Stackt Trace angezeigt. Einzelne Tests können von hier aus neu ausgeführt werden.

IntelliJ bietet eine Funktionalität namens „Tasks & Contexts“. Diese ähnelt dem Mylyn-Plugin in Eclipse, ist aber bei weitem nicht so fortgeschritten. Auch in IntelliJ können Tasks angelegt werden. Einem Task kann außer einem Namen aber lediglich ein sogenannter Kontext zugeordnet werden. Ein Kontext ist eine Sammlung von offenen Dateien, die im Editor angezeigt werden. Zwischen Tasks kann über ein Menü hin und her gewechselt werden. Andere Dateien werden aber in der Projekt- bzw. Package-Übersicht nicht ausgeblendet, wie dies in Mylyn der Fall ist. Zu einem Task wird außerdem angezeigt, wie lange dieser bereits existiert.

Darüber hinaus können Tasks verschiedener Groupware-Server[4] eingebunden werden (u.a. YouTrack, Redmine, Trac und GitHub). Ein oder mehrere entsprechende Server müssen dem Projekt zugeordnet werden. Anschließend stehen die dort hinterlegten Aufgaben bereit. Diese müssen allerdings erst explizit geöffnet werden, damit sie in der Übersicht der aktiven Tasks angezeigt werden. Für Tasks, die über einen Server eingebunden wurden, werden auch über den Namen hinausgehende Beschreibungen angezeigt, die sich je nach Groupware unterscheiden. Die Beschreibung kann wahlweise direkt in der IDE angesehen werden oder die zugeordnete Internetseite kann

[4] Groupware ist Software, die einer Gruppe von Personen dabei hilft, gemeinsam an einem Projekt zu arbeiten und dabei möglichst räumliche und zeitliche Distanz überbrückt.

über ein Menü aufgerufen werden. Eine Suchfunktion, die auch den Inhalt der Tasks durchsucht, steht allerdings nicht zur Verfügung. Ebenso wenig können solche Tasks in der IDE bearbeitet oder erstellt werden.

3.4. Microsoft Visual Studio Ultimate 2013

Das Erste was auffällt, wenn man versucht mit Visual Studio Tests zu erstellen, ist, dass sich innerhalb eines „gewöhnlichen“ Projekts (z.B. Konsolen- oder Desktopanwendung) keine Tests erstellen lassen. Möchte man Tests erstellen, muss dafür ein eigenes spezielles Testprojekt angelegt werden. Dies ist bei den drei wichtigsten, am besten von der IDE unterstützten Sprachen Visual Basic, Visual C# und Visual C++[5] der Fall. In Visual Basic und Visual C# gibt es jeweils drei verschiedene Testprojektarten: jeweils eine für UI-Tests, Unit-Tests und Lasttests. Für C++ werden ausschließlich Unit-Tests unterstützt. Dabei muss man sich jedoch entscheiden, ob nativer Code oder Bytecode (von Microsoft als „Managed Code“ bezeichnet) getestet werden soll. Die folgenden Ausführungen gelten sowohl für Visual Basic, als auch für C#. Für C++ gelten nur die Ausführungen zu den Unit-Tests.

Legt man ein neues Testprojekt an, so erstellt Visual Studio direkt einen leeren Test, wobei sowohl die Klasse als auch die angelegte Testmethode keinerlei Bezug zur zu testenden Anwendung haben (die Klasse heißt z.B. „UnitTest1“ und die darin enthaltene Methode „TestMethod1“). Legt man einen neuen Test an, so ist es nicht möglich eine Klasse auszuwählen, auf die sich der Test beziehen soll. Dementsprechend werden auch keinerlei Testmethoden (außer der schon bereits erwähnten) generiert. Auch Methoden zum Testauf- und Abbau lassen sich nicht automatisiert generieren, wie dies bei den IDEs für die Java-Entwicklung der Fall war. Möchte man nun eine vorher erstellte Applikation testen, muss zunächst im Testprojekt noch ein Verweis auf das entsprechende Projekt erstellt werden, da ansonsten Visual Studio den entsprechenden Namespace nicht kennt. Dies ist auch dann der Fall, wenn die Applikation bereits in Visual Studio geöffnet ist und sich in der gleichen Projektmappe[6] befindet. Für das Erstellen von Testfällen macht die vorher ausgewählte Projektart (UI-Test-, Unit-Test- oder Lasttest-Projekt) bei Visual Basic und C# keinerlei Unterschied. In jeder Projektart können alle verfügbaren Testarten erstellt werden. Lediglich die von Visual Studio initial erstellte Testklasse variiert zwischen den Projektarten. Die Tests können direkt nach dem Anlegen ausgeführt werden, es muss zuvor keine Run-Configuration, wie bei Eclipse oder IntelliJ, angelegt werden. Bei der Ausführung der Tests steht dem Benutzer vollkommen frei, welche Tests er ausführt. Er ist in keiner Weise an Verzeichnisse oder Klassen gebunden. Jede Methode, die als Test deklariert ist, kann einzeln in beliebiger Kombination mit anderen ausgeführt werden.

Führt man vorhandene Tests aus, wird auch in Visual Studio eine Übersicht über die

[5] Microsoft verwendet für seine Implementationen verschiedener Programmiersprachen jeweils das Präfix „Visual“.

[6] Eine Projektmappe in Visual Studio entspricht in etwa einem Workspace in Eclipse. Sie stellt eine Sammlung von Projekten dar.

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace UnitTestProject
{
    [TestClass]
    0 Verweise
    public class UnitTests
    {
        [TestMethod]
        | 0 Verweise
        public void SuccessTest()
        {
            Assert.AreEqual(1, 1);
        }

        [TestMethod]
        | 0 Verweise
        public void FailTest()
        {
            Assert.Fail();
        }
    }
}
```

Abbildung 3.3.: C#-Testklasse in Visual Studio 2013: Der obere Test ist beim letzten Durchlauf erfolgreich verlaufen, während der untere fehlgeschlagen ist. Erkennbar ist dies am kleinen grünen Haken bzw. roten Kreuz unmittelbar über den Methoden.

Tests und das Ausführungsergebnis angezeigt. Dieser „Test-Explorer“ ist mit den Übersichten in den Java-Umgebungen vergleichbar und zeigt im Wesentlichen die gleichen Informationen an und bietet auch ähnliche Funktionalität. Darüber hinaus wird in Visual Studio direkt im Code angezeigt, ob ein Test bei der letzten Ausführung erfolgreich war oder fehlgeschlagen ist (vgl. Abbildung 3.3).

Ein großer Unterschied zwischen Visual Studio und allen beschriebenen Java-IDEs ist der Umgang mit Oberflächentests. Diese können in den Java-IDEs nur mithilfe externer Frameworks eingebunden werden. Unterstützung auf IDE-Seite ist dabei nicht von vornherein vorgesehen. In Visual Studio ist dies anders. Dieses unterstützt den Nutzer beim Erstellen von GUI-Tests, indem es Möglichkeiten zur Aufzeichnung von GUI-Interaktionen bietet. Erstellt man eine UI-Testklasse, wird man aufgefordert, zwischen einer bereits existierenden Aufzeichnung und einer neu zu erstellenden Aufzeichnung als Basis der neuen Testklasse zu wählen. Wählt man Erstere, kann eine Aufzeichnung

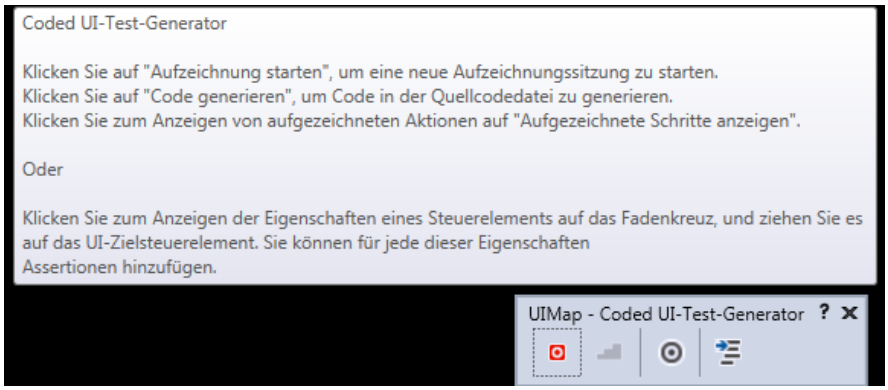

Abbildung 3.4.: Rekorder zum Aufzeichnen von GUI-Interaktionen und zum Generieren von entsprechendem Testcode

vom angebunden Team Foundation Server gewählt werden.[7] Wählt man hingegen eine neue Aufzeichnung, so minimiert sich Visual Studio und am unteren rechten Bildschirmrand werden einige Schaltflächen eingeblendet, die zum sogenannten Coded UI-Test-Generator gehören. Zusätzlich wird darüber ein Tooltip angezeigt, der erklärt, wie die Aufzeichnung von GUI-Interaktionen funktioniert (vgl. Abbildung 3.4). Dieser verschwindet, sobald man mit einem Fenster interagiert. Die Schaltflächen hingegen liegen in einer Ebene über allen anderen Fenstern des (Windows-)Betriebssystems.

Nach einem Klick auf die Aufnahmeschaltfläche startet die Aufnahme und es werden alle Interaktionen mit fast beliebigen Fenster aufgenommen. In selbst durchgeführten Versuchen funktionierte dies je nach Programm unterschiedlich gut. Am besten funktionierte die Aufnahmefunktion in .NET-basierten Programmen, also Programmen, die mit Visual Studio selbst erstellt wurden. Die Aufnahme kann jederzeit pausiert werden. Alle bisher aufgenommenen Aktionen werden dem Benutzer in einer Liste angezeigt. Daraus lässt sich per Knopfdruck Code in der jeweiligen Projektsprache (also Visual Basic oder C#) generieren. Dieser wird dort eingefügt, wo sich der Cursor beim Starten des Coded UI-Test-Generator befand.

Alternativ können auch Oberflächen auf Eigenschaften geprüft werden. Dazu kann mit gedrückter Maustaste ein Fadenkreuz auf ein beliebiges Element gezogen werden. Ein blauer Rahmen zeigt, welches Element gerade erkannt wurde. Lässt man die Maustaste los, so wird das Element ausgewählt und in einem Fenster werden alle erkannten Eigenschaften des Elements aufgelistet (vgl. Abbildung 3.5). Dazu zählt z.B. der Name des Elements, ob eine Tastenkombination zugeordnet ist, ob es existiert, ausgewählt ist oder auch welche Position auf dem Bildschirm das Element einnimmt. Dazu kommen noch elementspezifische Eigenschaften, wie zum Beispiel bei einem Textfeld der enthaltene Text, die maximale Länge oder ob Inhalt ausgewählt wurde. Eine Besonderheit stellen Elemente von Webseiten dar. Wählt man diese aus, so werden auch HTML-spezifische Eigenschaften erkannt. Dies funktioniert allerdings nur mit

[7] Der Team Foundation Server stellt das von Visual Studio bevorzugte Repository dar. Dies ein gesondertes Produkt von Microsoft und wird in der IDE an vielen Stellen unterstützt.

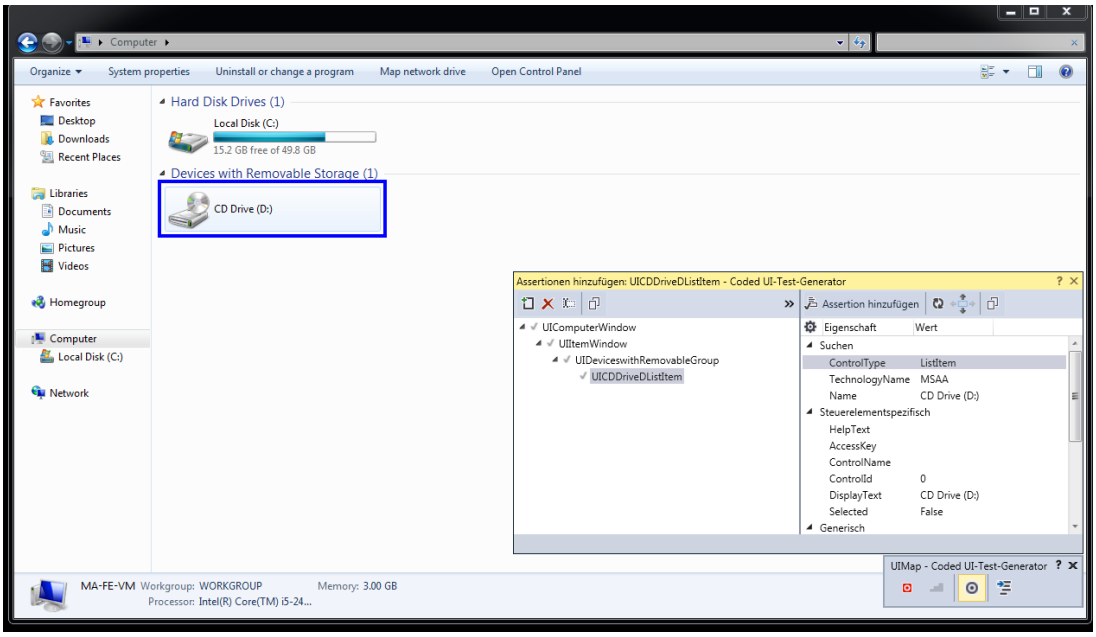

Abbildung 3.5.: Visual Studio UI-Test-Generator: Ausgewählt ist ein Element des Windows Explorer. Dieses wird mit einem blauen Rahmen markiert. Dazu werden alle erkannten Eigenschaften angezeigt.

dem Microsoft Internet Explorer. Bei anderen Browsers wird nicht erkannt, dass es sich um HTML-Elemente handelt. Die Elemente werden in dem Fall mit den Eigenschaften erkannt, die in einer beliebigen Desktopanwendung auch erkannt würden.

Über diesen Mechanismus lassen sich nach der Auswahl von UI-Elementen auch Tests, sogenannte Assertions, zu den Eigenschaften der Elemente hinzufügen. So können für jede erkannte Eigenschaft beliebig viele Assertions erstellt werden. Dazu kann bei der Assertion aus einer Liste ein Vergleichsoperator ausgewählt werden (z.B. AreEqual, AreNotEqual, IsNull, StartsWith etc.). Anschließend wird der Sollwert (im Programm Vergleichswert genannt) der entsprechenden Eigenschaft eingegeben. Zusätzlich lässt sich noch eine Meldung angeben, die ausgegeben wird, sollte die Assertion fehlschlagen, d.h. wenn der Sollwert und der tatsächliche Wert der Eigenschaft nicht übereinstimmen. Auch daraus lässt sich dann Code generieren. Dieser wird allerdings nicht direkt in die eigentliche Testklasse integriert, sondern in drei spezielle Dateien, die zusammen eine sogenannte UIMap bilden. Darin befindet sich sämtlicher Code, der sich ausschließlich auf die GUI bezieht. Die aufgenommenen Aktionen bzw. Assertions sind über öffentliche Methoden abspielbar bzw. aufrufbar.

Eine weitere von Visual Studio unterstützte Testart sind sogenannte Webleistungstests. Diese dienen dazu, die Leistungsfähigkeiten einer Webseite zu testen. Dazu können, wie bei GUI-Tests, Schritte im Internet Explorer aufgezeichnet werden, um später im Test Verwendung zu finden. Auch alle bisher beschriebenen Testarten können in einen Lasttest integriert werden. Wird der Test schließlich ausgeführt, so zeigt Visual Studio im Anschluss detaillierte Statistiken an. Neben einer Zusammenfassung

stehen tabellarische Ansichten, sowie beliebig konfigurierbare Diagramme der erfassten Parameter zur Verfügung.

Abschließend lässt sich sagen, dass Visual Studio mit Abstand die umfangreichste Unterstützung bezüglich der Erstellung von Tests bietet. Insbesondere die integrierte Unterstützung zur Aufnahme von GUI-Tests, ist ein sehr mächtiges und hilfreiches Feature, dass in den anderen beschriebenen IDEs vollkommen fehlt.

4. Der Testauftrag

In diesem und im folgenden Kapitel möchte ich darauf eingehen, wie das Crowdsourcen von automatisierten Tests auf der Seite des Workers konzeptionell gestaltet werden kann. Dabei möchte ich zunächst auf den Testauftrag eingehen. Dieser stellt das zentrale Kommunikationselement zwischen dem Provider und dem Worker dar. Daher ist die Struktur des Testauftrags enorm wichtig für das Gelingen des Crowdsourcings in diesem Kontext. Allein durch die Angaben im Testauftrag, muss der Worker in der Lage sein, zu erschließen, was seine konkrete Aufgabe ist. Außerdem muss der Testauftrag alle für die Erledigung der Aufgabe notwendigen Informationen bzw. Dateien enthalten.

Der Testauftrag befindet sich dabei in einem Spannungsfeld. Auf der einen Seite ist der Provider, der den Testauftrag erstellen muss. Dieser möchte so wenig Aufwand wie möglich für die Erstellung aufwenden. Denn je größer der Aufwand für das Auslagern der Testerstellung ist, desto kleiner wird der Nutzen, der daraus gezogen werden kann. Wäre der Aufwand für die Erstellung eines Testauftrags ähnlich hoch oder gar höher, als der Aufwand den Test selbst zu entwickeln, würde das Auslagern der Testerstellung keinen Sinn mehr ergeben. Auf der anderen Seite des Spannungsfeldes befindet sich der Worker. Dieser soll einen Test für ein ihm unbekanntes Projekt erstellen und benötigt dementsprechend viel Hilfe, um in das Projekt hineinzufinden und im Anschluss die ihm gestellte Aufgabe zu lösen.

Deshalb müssen die Struktur und der Umfang des Testauftrags einen Kompromiss zwischen beiden Seiten darstellen. Denn auf der einen Seite können nicht alle Hilfen, die für den Worker vielleicht wünschenswert wären, realisiert werden, während auf der anderen Seite der Provider gewisse Angaben, zu dem was getestet werden soll, einfach machen muss. Unter diesen Gesichtspunkten habe ich mit Benjamin Riedel [Riedel2014] die Struktur des Testauftrags gemeinsam festgelegt. Dabei hat er die Seite des Providers vertreten, während ich die Rolle des Workers übernommen habe.

Bevor ich zu den einzelnen Komponenten komme, möchte ich noch kurz erläutern, welchen Umfang wir für das Ergebnis eines Testauftrags vorgesehen haben. Ein Testauftrag zielt auf die Erstellung eines oder mehrerer zueinander verwandten Tests ab. Als Test definieren wir hier eine konkrete Interaktion mit der Application Under Test (AUT) und dem daraus resultierenden gewünschten Verhalten, dessen Eintreten automatisiert überprüft werden soll. Die Tests eines Testauftrags sollen dabei stets auf eine bestimmte Komponente der AUT abzielen und eine einheitliche Testvorbereitung (Setup-Phase) benötigen. Wie das Ergebnis eines Testauftrags letztendlich aussieht, hängt aber von der konkreten Aufgabenstellung und deren Komplexität ab.

4.1. Die zentralen Komponenten des Testauftrags

Ich werde im Folgenden eine vollständige Übersicht über die Bestandteile des Testauftrags geben, wie sie zwischen Benjamin Riedel und mir vereinbart wurden. Nicht alle dieser Komponenten werden zur Unterstützung des Workers benötigt. Einige davon sind aus rein administrativen Gründen im Testauftrag enthalten (siehe dazu Abschnitt 4.2).

- **Name**: Der Name des Projektes bzw. Testauftrags.
- **Software**: Die Software, für die ein automatisierter Test erstellt werden soll (AUT), ist in kompilierter Form Bestandteil des Testauftrags. Bibliotheken, die für die Ausführung der Software notwendig sind, sind ebenfalls enthalten.
- **Art des Tests**: Was für eine Art von Test soll erstellt werden? Der Fokus dieser Arbeit liegt auf der Erstellung von GUI-Tests. Jedoch sind prinzipiell auch andere Testarten, wie Unit-, Integrations- oder Systemtests denkbar. Im Test-Task-Editor [Riedel2014] werden neben GUI- auch bereits Unit-Tests unterstützt.
- **zu testende Datei**: Die Datei, die die Klasse oder GUI enthält, die getestet werden soll.
- **zu testende Methode**: Die Methode, die getestet werden soll. Diese Angabe ist bei GUI-Tests nicht vorhanden und für andere Testarten gedacht.
- **relevante Dateien**: Dies ist eine Liste von Quellcodedateien, die vom Provider als relevant für den Worker eingestuft wurden. Sie sollen dem Worker einen Einblick in die Funktionsweise der von ihm zu testenden Softwarekomponente ermöglichen und es ihm damit vereinfachen, einen Test zu entwickeln.
- **im Projekt enthaltene Tests**: Dies sind Tests aus dem Projekt des Providers. Sie können für den Worker als Grundlage für den zu erstellenden Test dienen.
- **Links zu Tutorials**: Eine Liste von Links, die zur Online-Dokumentation der verwendeten Test-Technologie führen (also z.B. JUnit oder Selenium).
- **Beispieltest**: Ein zur verwendeten Test-Technologie passendes, projektunabhängiges Codebeispiel, das die technischen Grundlagen zeigt und durch den Test-Task-Editor vorgegeben wird (siehe auch [Riedel2014] „Felder eines Testauftrags“).
- **Dokumentation/unterstützende Dokumente**: Hier kann der Provider Dokumente angeben, die dem Worker beim Verständnis des Projekts helfen. Denkbar wären hier u.a. Spezifikationsdokumente oder (UML-)Diagramme.
- **Voraussetzungen für den Test**: Die Bedingungen dafür, dass der Test ausgeführt werden kann, also z.B. ein bestimmter Datenbankzustand. Der Provider formuliert die Bedingungen grundsätzlich als Freitext, ist aber angehalten, diese sinnvoll in einzelne Punkte zu untergliedern. So können die notwendigen Vorbereitungen für den Testablauf dem Worker in Form einer Liste angezeigt werden, sodass dieser Bedingungen dann Schritt für Schritt abarbeiten kann.

- **Hauptszenario/Normalverhalten**: Das Verhalten, dass von einer Komponente (z.B. Methode oder GUI) im Allgemeinen erwartet wird.
- **zu testendes Verhalten**: Hier wird dem Worker mitgeteilt, wofür er ganz konkret Tests entwickeln soll. Dies kann entweder das Normalverhalten sein, dass getestet werden soll, oder Abweichungen bzw. Erweiterungen des Hauptszenarios (z.B. wie verhält sich eine Komponente bei fehlerhaften Eingabedaten). Dieser Teil des Testauftrags liegt in Form einer Liste von Wenn-Dann-Sätzen (bzw. englisch if-then) vor. Der Wenn-Teil beschreibt eine Interaktion mit der AUT, während der Dann-Teil die gewünschte Reaktion der AUT beschreibt, also das Sollverhalten. Durch diese Form soll es dem Worker möglichst leicht gemacht werden, zu verstehen, welche Aktionen im Test codiert werden müssen und welches Verhalten der AUT der Test letztendlich überprüfen soll. Für jeden einzelnen Wenn-Dann-Satz der Liste muss vom Worker ein automatisierter Test erstellt werden.
- **Anmerkungen**: Anmerkungen und Hinweise zum Testauftrag, die an anderer Stelle fehl am Platz wären.
- **Screencast**: Ein Video des Bildschirminhalts des Providers, welches mit Audiokommentaren hinterlegt ist. Mit Hilfe dieses Screencasts kann der Provider, zusätzlich zur textuellen Beschreibung, Sachverhalte innerhalb kurzer Zeit tiefergehend erläutern. Die Screencast-Funktionalität fehlt aus Zeitgründen im Test-Task-Editor [Riedel2014], ist bei mir konzeptionell aber vorgesehen.

4.2. Administrativ notwendige Komponenten

- **Codezugriff**: Gibt an, ob der Testauftrag den vollständigen Quellcode des Projekts enthält oder ob lediglich ausgewählte Teile für den Worker zur Verfügung stehen (siehe dazu auch Abschnitt 5.5).
- **Aufwand**: Eine automatisch erfolgte Abschätzung darüber, wie aufwändig die Bearbeitung dieses Testauftrags ist. Dieses Feld ist für die Crowdsourcing Plattform relevant und soll einem potenziell interessierten Worker die Möglichkeit geben, den Aufwand, den er zur Lösung des Auftrags investieren muss, abzuschätzen. Mögliche Angaben sind: niedrig, mittel, hoch.
- **Entlohnung**: Die Höhe der Bezahlung, die der Worker bei erfolgreicher Lösung des Testauftrags erhält. Es besteht bei diesem Feld sicherlich ein Zusammenhang zum Aufwand. Ein aufwändigerer Auftrag wird in der Regel eine höhere Entlohnung aufweisen müssen, als ein nicht so aufwändiger, um von der Crowd bearbeitet zu werden. Allerdings kann ein Provider über die Entlohnung unter Umständen die Geschwindigkeit beeinflussen, mit der sein Testauftrag gelöst wird. So wird ein wenig aufwändiger, aber hoch entlohnter Auftrag sicherlich schnell Bearbeiter finden.
- **AUT-Version**: Für welche Version der AUT wurde der Testauftrag erstellt? Zwischen dem Erstellen des Testauftrags und der Integration der Lösung kann sich

die AUT verändert haben, so dass der entwickelte Test möglicherweise nicht mehr funktioniert. Daher muss der Entwickler beim Integrieren der Lösungen eines Testauftrags wissen, für welche Softwareversion der Test entwickelt wurde.

- **E-Mail-Adresse des Providers**: Die E-Mail-Adresse des Providers des Testauftrags. Diese wird für das Stellen von Rückfragen verwendet. Dieses Feld entfällt unter Umständen, abhängig von der Gestaltung der Crowdsourcing Plattform.

4.3. Einschränkungen des Testauftrags

Was genau ein Provider mit einem Testauftrag auslagert, hängt weitestgehend von ihm selbst ab. Allerdings sind nicht alle Applikationen bzw. Tests für die Auslagerung in die Crowd geeignet. Wird für einen Test eine komplexe Umgebung benötigt, in der die AUT z.B. mit anderen Systemen zusammenarbeitet, ist dieser nicht für die Crowd geeignet, da es einem Worker nicht ohne Weiteres möglich ist (vor allem nicht in kurzer Zeit), eine komplexe Testumgebung mit einer Vielzahl von Systemen aufzubauen. Dies gilt auch, wenn ein tiefes Verständnis der AUT, welches nicht innerhalb kurzer Zeit erworben werden kann, benötigt wird.

Einschränkungen erfolgen aber auch durch die Struktur des Testauftrags. Grundsätzlich kann mit einem Testauftrag nicht nur ein einzelner Test angefordert werden. Wichtig bei der Anforderung von mehreren Tests ist aber, dass die Tests den gleichen Bereich der AUT betreffen. Für GUI-Tests heißt dies konkret etwa, dass die Tests nur eine bestimmte Oberfläche betreffen können. Dabei können mit einem Testauftrag durchaus Tests für mehrere Abläufe gefordert werden. Beispielsweise könnten Tests für einen Use Case mit all seinen Erweiterungen und Sonderfällen in einem Testauftrag zusammengefasst werden. Das Gleiche gilt auch für Unit-Tests.

Ein exploratives Vorgehen ist ebenfalls denkbar. So könnte ein Provider im Testauftrag einen Use Case oder eine Komponente beschreiben und die bisher vorhandenen Tests dazu dem Auftrag hinzufügen. Die Aufgabe der Crowd wäre es nun, Tests beizusteuern, die Aspekte abdecken, die von anderen Tests bisher nicht erfasst wurden und an die der Provider bisher nicht gedacht hat. Als Beispiel können hier ungewöhnliche Benutzereingaben dienen, die so in erster Linie nicht zu erwarten sind, aber vielleicht trotzdem die AUT zum Absturz bringen können oder sicherheitskritisch sind.

5. Konzepte für die Entwicklungsumgebung - Aufteilung des Testprozesses

In diesem Kapitel möchte ich darauf eingehen, wie der Worker bei seiner Arbeit mit der Entwicklungs- und Testumgebung unterstützt werden kann. Dazu werde ich zunächst das Grundkonzept vorstellen, auf dem alle weiteren Konzepte basieren. Danach werde ich einzelne Komponenten innerhalb der IDE vorstellen, durch die das Gesamtkonzept realisiert wird, und die den Worker bei seiner jeweiligen Tätigkeit unterstützen sollen.

5.1. Das Grundkonzept

Ein Worker soll beim Lösen eines Testauftrags von der Entwicklungsumgebung unterstützt werden. Die Tätigkeiten, die er dabei ausübt, ändern sich während des zeitlichen Voranschreitens des Lösungsprozesses. Seine Kenntnisse bezüglich der AUT erweitern sich dabei ständig (siehe Abbildung 5.1). Entsprechend halte ich es für sinnvoll, dass die Entwicklungsumgebung an die jeweilige Tätigkeit angepasst ist und jeweils spezifische Hilfestellungen bietet. Daher möchte ich an dieser Stelle den Lösungsprozess eines Testauftrages in drei Phasen aufteilen.

Phase 1: Entdecken des Projekts

Nachdem sich ein Worker dafür entschieden hat, einen Testauftrag anzunehmen, muss er zunächst einen Einstiegspunkt in das Projekt finden. Dies ist für ihn nicht einfach zu bewerkstelligen. Der Worker hat zu Beginn keinerlei Kenntnisse über das Projekt, dessen Ziele oder Strukturierung. Die Situation weist Parallelen zu einem Tester beim ausgelagerten Testen auf. Auch dieser hat zu Beginn eines Projekts kein detailliertes Wissen über das Projekt und muss sich einarbeiten. Das kann, je nach Projektgröße, unter Umständen erhebliche Zeit in Anspruch nehmen. Bei einem solchen Tester ist dies aber akzeptabel, da von ihm erwartet wird, über einen längeren Zeitraum Tests zu entwickeln. Bei einem Worker in der Crowd ist eine solch längere Einarbeitungszeit inakzeptabel. Dieser soll lediglich einen bzw. eine sehr kleine Zahl an Tests entwickeln, d.h. er hat lediglich eine kleine, sehr spezifische Aufgabe zu lösen. Daher macht es keinen Sinn, dass dieser sich mit allen Details des Projekts auseinandersetzt. Das bedeutet, dass er gerade so viel Wissen über das Projekt erwerben muss, um in der Lage zu sein, diese Aufgabe zu lösen. Alles, was darüber hinausginge, wäre überflüssig.

Ziel dieser ersten Phase ist es jedoch nicht, ihm dieses Wissen bereits vollständig zu vermitteln. Vielmehr soll der Worker zunächst lediglich in die Lage versetzt werden, mit

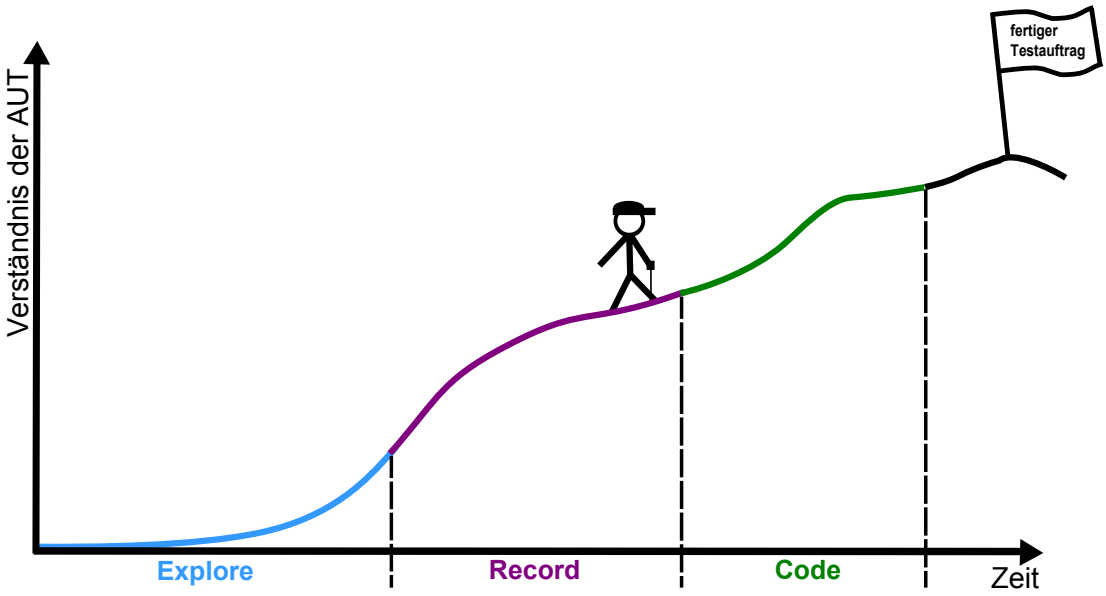

Abbildung 5.1.: Übersicht über die drei zentralen Phasen des Testprozesses. Mit zunehmender Zeit wächst das Verständnis des Workers für die AUT. Daran richten sich die Phasen aus.

der Entwicklung des im Testauftrag beschriebenen Tests zu beginnen. Das bedeutet, er weiß, welche Komponente der Software vom zu entwickelnden Test betroffen ist, welcher Aspekt getestet werden soll und er hat sich, falls der Testauftrag dies erfordert, einen ersten Überblick über die entsprechenden Codeabschnitte verschafft. Sämtliche Details muss er zu diesem Zeitpunkt noch nicht kennen.

Was bedeutet dies nun für die Entwicklungsumgebung? Diese sollte die Aufmerksamkeit des Workers genau auf die für ihn relevanten Aspekte des Projekts lenken. Alles, was nicht zur Lösung des konkreten Testauftrags beiträgt, sollte idealerweise auch nicht angezeigt werden. Doch auch wenn die Entwicklungsumgebung zu einer Fokussierung des Workers auf die wesentlichen Komponenten beiträgt, so besteht trotzdem noch das Problem, dass auch diese wenigen Komponenten dem Worker bisher gänzlich unbekannt sind. Das bedeutet, dass die Entwicklungsumgebung den Worker bei der Einarbeitung in die relevanten Projektteile unterstützen sollte. Für ihn ist es dabei wichtig, sich leicht einen Überblick verschaffen zu können.

Phase 2: Strukturierung des Testfalls/GUI-Interaktion

Die zweite Phase sieht Interaktionen mit der GUI der AUT vor. Dies halte ich insbesondere bei der Erstellung von GUI-Tests für sinnvoll. Inwieweit eine solche oder ähnliche Phase auch bei anderen Testarten (z.B. Unit-, Integrations- oder Systemtests) Sinn macht, müsste in zukünftigen Arbeiten untersucht werden.

Der Code zur Erzeugung einer Benutzeroberfläche erreicht bei nicht ganz einfachen

Oberflächen relativ schnell einen erheblichen Umfang. Dies gilt sowohl bei der Verwendung einer Programmiersprache (z.B. Java mit SWT oder Swing), als auch bei Nutzung einer Beschreibungssprache (z.B. XML oder HTML mit CSS). Prinzipiell wäre es natürlich möglich, einen GUI-Test nur auf Basis der Codeansicht zu erstellen. Einem Worker, der sich bisher nur wenig mit dem Projekt beschäftigt und die Benutzeroberfläche bisher nicht als solche gesehen hat, wird es jedoch schwer fallen, sich diese nur anhand von Code vorzustellen. Entsprechend wird er dann auch Schwierigkeiten haben, schnell die für ihn relevanten Elemente zu ermitteln und die für den Test notwendigen Interaktionen zu erstellen.

Einfacher könnte dies erledigt werden, wenn der Worker direkt mit der GUI interagieren kann. Eine Benutzeroberfläche ist idealerweise so auf den Nutzer zugeschnitten, dass dieser möglichst effizient mit ihr arbeiten kann. Selbst wenn der Worker vor Annahme des Testauftrags nicht in der Lage gewesen sein sollte, die Software zu bedienen, so sollte er spätestens nach Abschluss der ersten Testphase zumindest den für ihn relevanten Teil bedienen können. Ansonsten würde es ihm auch schwerfallen, einen entsprechenden GUI-Test zu codieren. Man kann also an dieser Stelle davon ausgehen, dass der Worker weiß, wie mit der Oberfläche zu interagieren ist. Dieses Wissen kann genutzt werden.

Ermöglicht man es dem Worker, GUI-Interaktionen aufzuzeichnen, so muss dieser sich zunächst einmal nicht mit dem Quellcode der Benutzeroberfläche auseinandersetzen. Alle Aktionen kann er auf der GUI ausführen und so eine erste grobe Struktur für den Testablauf festlegen, die später zur Erzeugung von Testcode genutzt werden kann. Dadurch wird unter Umständen auf zweifache Weise Zeit eingespart. Der Worker muss sich an dieser Stelle des Testprozesses nicht die Details des Quellcodes der Oberfläche einarbeiten. Eventuell muss er dies zu einem späteren Zeitpunkt tun. Es gibt aber Fälle, in denen die Details auch später keine Rolle spielen werden und in denen so Zeit gespart werden konnte. Der zweite Aspekt, der eine Zeitersparnis darstellt, ist der Umstand, dass dem Worker durch die Aufzeichnung der GUI-Interaktionen das Programmieren des Ablaufs auf Quellcodeebene abgenommen wird. Neben GUI-Interaktionen können so auch (einfache) Assertions erstellt werden, die z.B. das Vorhandensein von bestimmten Elementen oder bestimmte Attribute in der GUI sicherstellen. Dieses Konzept ist bereits sehr umfangreich in Visual Studio von Microsoft kommerziell umgesetzt worden (siehe dazu auch Abschnitt 3.4). Mit leichten Modifikationen versehen, schätze ich dieses Konzept als sehr gut geeignet für die Verwendung innerhalb des Crowdsourcing-Kontextes ein.

Phase 3: Codierung des Testfalls

Nachdem der Worker den groben Ablauf des Tests in der zweiten Phase festgelegt hat, wird daraus zu Beginn der dritten Phase Quellcode generiert. Nun hat der Worker die Möglichkeit, die Details zu programmieren. Dazu gehört etwa das Setup des Tests, die Anbindung von Datenquellen (für Eingabedaten), oder auch das Mocken anderer Softwarekomponenten. Im Wesentlichen werden die Teile des Tests erstellt, die sich nicht durch Interaktion mit der AUT erstellen lassen. Dazu können auch Assertions ge-

hören, die komplexe Sachverhalte abdecken sollen. Diese lassen sich einfacher über die Eingabe von Quellcode erzeugen, als über komplexe Menüs.

Einzelkonzepte für die Eclipse-IDE

Die Dreiteilung des Testprozesses sollte sich in der Testumgebung wiederfinden. Da diese Arbeit eine prototypische Umsetzung einiger Konzepte in Eclipse beinhaltet, werde ich die einzelnen Elemente, durch die sich das Grundkonzept meiner Ansicht nach umsetzen ließe, anhand von Eclipse beschreiben. Dieses unterstützt Perspektiven, die es ermöglichen, Elemente (Views) auf der Arbeitsfläche der IDE gezielt zu platzieren. Einige Elemente tauchen in mehreren Perspektiven auf. Ich werde diese jedoch nur einmalig beschreiben, da ihr Inhalt und ihre Funktion jeweils identisch sind.

5.2. Phase 1: Explore-Perspective

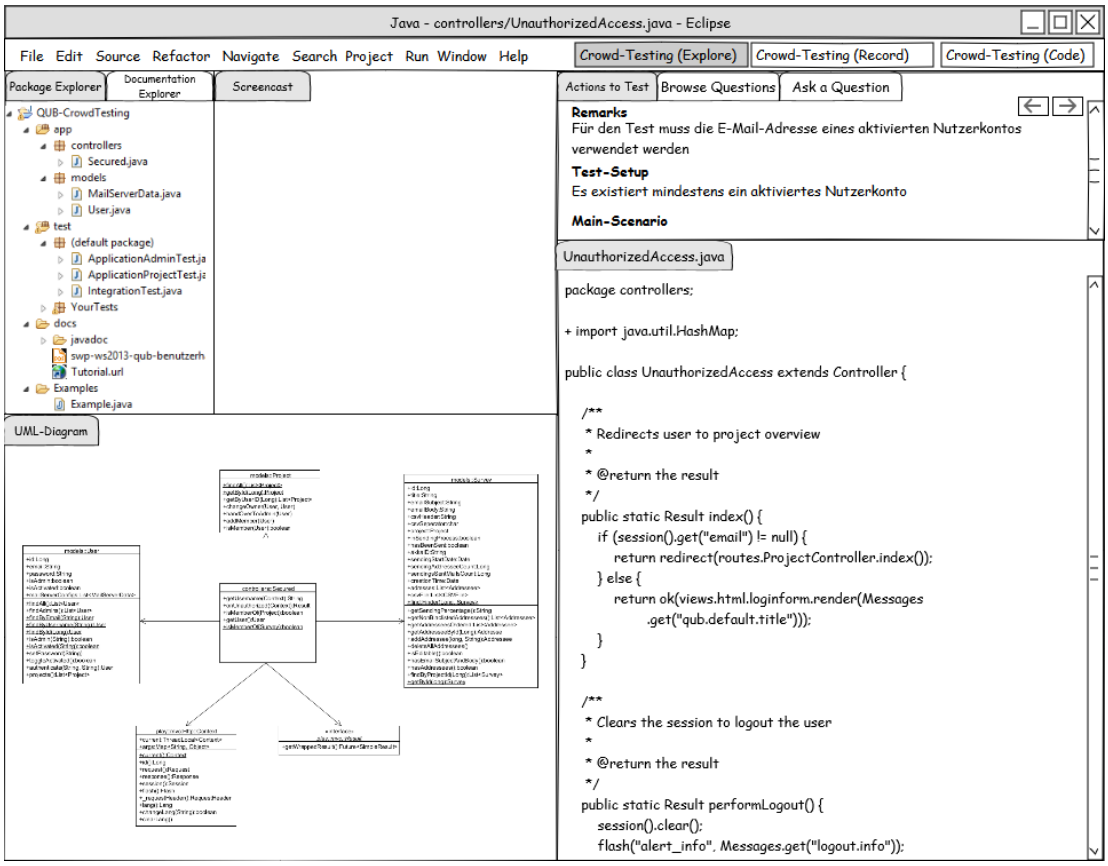

Abbildung 5.2.: Die Explore Perspektive mit geöffnetem Package Explorer und Actions to Test-View

Das Ziel dieser Perspektive ist es, dem Worker einen Überblick über den für ihn relevanten Teil des Projekts zu verschaffen.

Actions to Test

Die „Actions to Test“-View (siehe Abbildung 5.4 oben) enthält alle Informationen darüber, was der bzw. die Tests, die der Worker erstellen soll, genau prüfen sollen. Die View soll dem Worker dabei möglichst klar vorgeben, welche Schritte auszuführen sind. Sie soll ihm außerdem eine Übersicht darüber geben, welcher Teil des Testauftrags bereits abgeschlossen ist. Sie enthält folgende Komponenten:

- **Test-Typ**: Die Art des Tests, der erstellt werden soll. Also zum Beispiel GUI-Test oder Unit-Test.
- **Class under Test**: Die Klasse bzw. der Code der getestet werden soll.
- **Remarks**: Dies ist Freitext, durch den der Provider Hinweise und Anmerkungen zum Testauftrag machen kann, die nicht in die übrige Struktur passen.
- **Test-Setup**: Dies ist die Voraussetzung, unter der der zu entwickelnde Test ausgeführt werden soll.
- **Main-Scenario**: Die Beschreibung des Use Cases, der die Nutzung zu testenden (Web-)Seite beschreibt.
- **Actions To Test**: Dies entspricht dem Erweiterungen-Feld einer Use Case-Tabelle. Eine Erweiterung stellt dabei genau ein Verhalten des Systems dar, dass von einem Test geprüft werden soll. Das Format ist äquivalent zur Use Case-Tabelle im Wenn-Dann-Stil (bzw. englisch If-Then). Eine einzelne Erweiterung kann dabei als abgeschlossen markiert werden (Haken unten rechts), wenn für sie bereits ein Test erstellt wurde. Wird eine Erweiterung vom Worker angeklickt, so werden die Schritte im Hauptszenario hervorgehoben, auf die sich die konkrete Erweiterung auswirkt.

Diese View enthält alle notwendigen Angaben zum Lösen des Testauftrags. Sie ist deshalb in jeder Perspektive enthalten. Steht in einer Perspektive nicht die gesamte Bildschirmbreite zur Verfügung, kann der Worker über zwei Schaltflächen die linke bzw. rechte Seite ausblenden. Alle Angaben, bis auf die Anmerkungen (Freitext), sind so untergliedert, dass der Worker möglichst Schritt für Schritt einzelne Punkte abarbeiten kann. Alle Schritte beim Setup und dem Main-Scenario sind deshalb durchnummeriert.

Screencast

Der Screencast dient dazu, dem Worker innerhalb kurzer Zeit viele Informationen zu übermitteln. Der Task-Provider kann mithilfe der Screencast-Funktion ein Video seines Bildschirms aufnehmen. Er kann währenddessen wesentliche Aspekte des Test-Tasks

auf dem Bildschirm zeigen und parallel dazu Erklärungen über ein Mikrofon aufnehmen. Einerseits spart dies dem Provider Zeit bei der Erstellung des Test-Tasks, da er das, was er übermitteln möchte, nicht erst aufschreiben muss. Andererseits hat dies für den Worker den Vorteil, dass ihm tendenziell mehr Informationen übermittelt werden, da eine Videoaufnahme mit Ton nach Cockburn [Cockburn22008] reichhaltiger und effektiver ist, als eine geschriebene Botschaft.

Zu beachten ist aber auch, dass ein Video, welches gesprochene Sprache enthält, nur schwierig automatisiert geparsed bzw. nach hilfreichen Informationen durchsucht werden kann. Das bedeutet, dass Informationen, die in einem solchen Screencast enthalten sind, an anderer Stelle nicht ohne Weiteres für den Worker aufbereitet dargestellt werden können. Zudem liegt die Strukturierung des Screencasts gänzlich in den Händen des Providers, was sich u.U. nachteilig auswirken kann. Eine mögliche Sprachbarriere zwischen Provider und Worker ist ebenfalls zu beachten. Zeichnet der Provider einen Screencast in einer Sprache auf, die der Worker nicht versteht, sind die darin enthaltenen Informationen für den Worker nahezu nutzlos. Der Screencast sollte deshalb lediglich als Ergänzung zur übrigen Beschreibung und nicht als Ersatz für diese genutzt werden.

Package Explorer

Eine in die Crowd ausgelagerte Testaufgabe wird in der Regel nur einen kleinen Codeausschnitt der gesamten Software betreffen. Aufgaben, bei denen dies nicht der Fall ist und der Worker erst ein tiefergehendes Verständnis der Software erwerben müsste, sind nicht uneingeschränkt für die Auslagerung in die Crowd geeignet. Daher sollte man von ersterem Fall ausgehen, wobei sich dann das Paradigma des Information Hiding aus der objektorientierten Programmierung anwenden ließe. Dieses besagt, dass nur der Teil eines Objekts bekannt sein sollte, der für die Interaktion mit diesem notwendig ist und dass innere Strukturen verborgen werden sollten.

Einen solchen Ansatz verfolgt das bereits vorgestellte Eclipse-Plugin Mylyn. Dieses versteckt alle Elemente, die für die aktuell ausgewählte Aufgabe nicht relevant sind. Diesen halte ich auch im Kontext des Crowdsourcen von Tests für sehr sinnvoll. Bei Mylyn definiert der Nutzer allerdings selber, was er für seine aktuelle Aufgabe als relevant empfindet und kann dies jederzeit anpassen. Im Kontext des Crowdsourcing ist dies jedoch nicht sinnvoll, da der Worker zu Beginn keine Kenntnisse über das Projekt besitzt. Er weiß also nicht, was für ihn relevant ist und was nicht. Daher sollte die Ansicht durch die Software so vorkonfiguriert werden, dass dem Worker automatisch nur relevante Elemente in der Projekt-Übersicht im Package Explorer angezeigt werden.

Als relevant kann beim Crowdsourcing alles betrachtet werden, was dem Worker unmittelbar hilft, die gewünschten Tests zu entwickeln. Dazu zählt auf jeden Fall der Teil der Software, der getestet werden soll, also in der Regel die Dateien, die den zu testenden Code enthalten. Weiterhin sollte Dokumentation, die zur Verfügung gestellt wird, sichtbar sein. Wie in Abbildung 5.2 zu sehen, könnte diese zentral innerhalb eines Verzeichnisses gesammelt aufgelistet werden. Automatisch generierte Java-Doc könnte

diesem Verzeichnis ebenfalls hinzugefügt werden. Auch mitgelieferte Links zur Dokumentation des verwendeten Test-Frameworks (z.B. JUnit) können hier angezeigt werden (siehe dazu auch [Riedel2014]). Bereits existierende Tests des Projekts können bei der Entwicklung eines neuen Tests helfen [Shah2011] (siehe dazu auch Abschnitt 4) und sollten dementsprechend auch im Projekt angezeigt werden. Andere Elemente hingegen, wie weiterer Quelltext, Konfigurationsdateien, Grafiken, Bibliotheken etc., die dem Worker nicht bei der Erstellung der gewünschten Tests helfen, sollten ausgeblendet werden, da sie nur ablenken. Die Gliederung des Projekts in die drei Ordner Source-Code, Tests und Dokumentation halte ich für sinnvoll. Diese Einteilung ist projektunabhängig.

UML-Diagram

Beim Öffnen einer Quellcodedatei wird - sofern diese eine (Java-)Klasse enthält - automatisch ein UML-Klassendiagramm dieser Klasse aus dem Quellcode erzeugt. In dieses Diagramm werden weiterhin die Klassen integriert, die von der geöffneten Klasse verwendet werden. Idealerweise können einzelne Elemente im Klassendiagramm vom Worker angeklickt werden, um diese im Editor zu öffnen. Klassen, die ausschließlich in kompilierter Form im Projekt vorliegen, und damit vom Provider als nicht relevant für Worker eingestuft wurden, werden trotzdem im Diagramm angezeigt, sind aber speziell als nicht relevant gekennzeichnet.

Die Anzeige eines Klassendiagramms soll es dem Worker vereinfachen, sich einen Überblick über die vorhandenen Klassen zu verschaffen. Dieser muss sich so zunächst nicht mit den Details der Implementierung befassen, sondern er kann auf einer abstrakteren Ebene erfassen, welchem Zweck die geöffnete Klasse dient und wie diese sich in das Projektgefüge eingliedert. Parallel dazu hat er aber auch die Möglichkeit sich Details der Implementierung im Editor anzusehen. Auch dabei kann das UML-Diagramm dem Verständnis dienen. Stößt der Worker beim Betrachten des Quellcodes auf eine Klasse, die ihm bisher unbekannt ist (dies wird wahrscheinlich häufig der Fall sein, da dem Worker ja das gesamte Projekt unbekannt ist), so kann er über das Diagramm einen ersten Überblick über die Klasse gewinnen und sich damit möglicherweise schon ihren Zweck erschließen.

Documentation Explorer

Der Documentation Explorer soll gezielt im Projekt enthaltene Dokumentation anzeigen. Da nicht davon auszugehen ist, dass ein Task-Provider Dokumentation speziell für einen spezifischen Test-Task zusammenstellt (dies wäre zeitaufwendig und würde den Nutzen des Crowdsourcings mindern oder sogar beseitigen), kann angenommen werden, dass die Dokumentation, die dem Worker zur Verfügung gestellt wird, mehr umfasst, als dieser eigentlich benötigt. Daher wäre es für den Worker hilfreich, wenn ihm Stellen innerhalb der Dokumentation, die möglicherweise relevant sind, direkt verlinkt würden. So müsste dieser nicht erst danach suchen. Dies soll mit dem Documen-

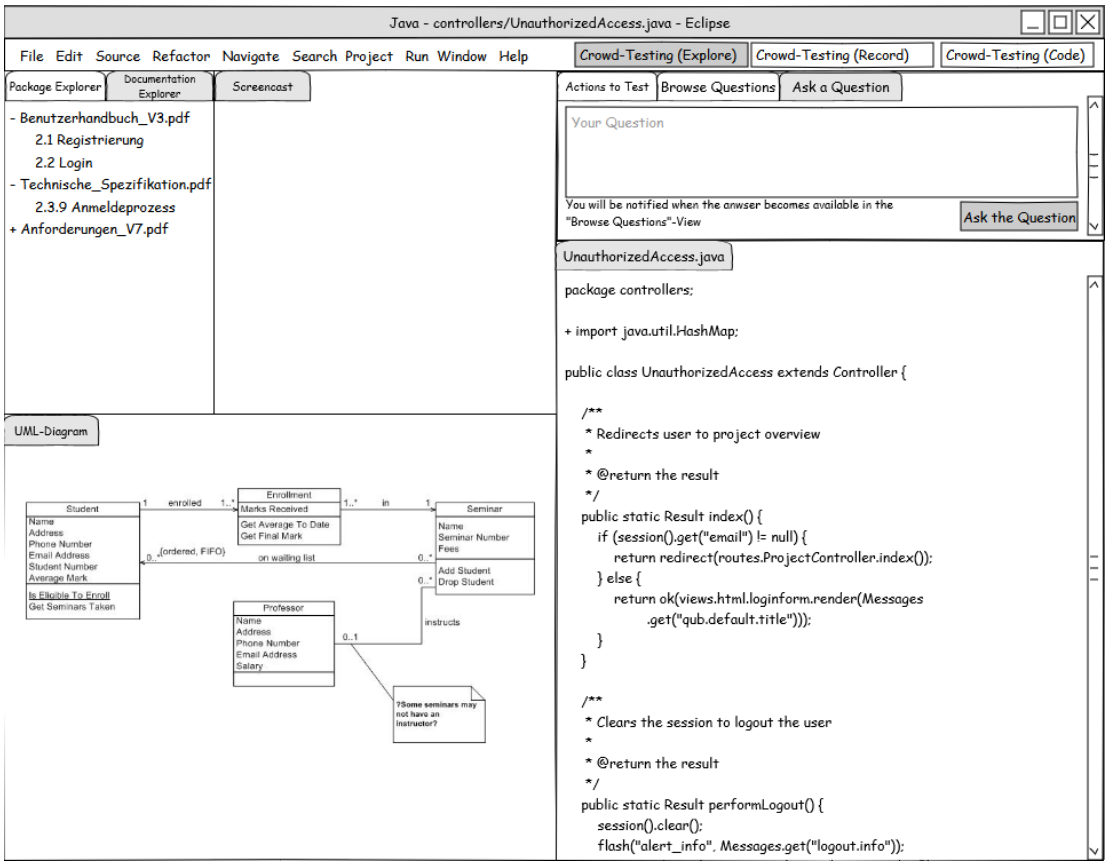

Abbildung 5.3.: Die Explore Perspektive mit geöffnetem Documentation Explorer und Ask a Question-View

tation Explorer geschehen. Dieser listet die vorhandene Dokumentation auf und zeigt Links zu möglicherweise für den Testauftrag relevanten Abschnitten. Durch einen Klick auf einen solchen Link öffnet sich der entsprechende Abschnitt und der Worker muss nicht erst danach suchen.

Inwieweit sich eine solche Übersicht wirklich automatisiert aus den Eingaben des Task-Provider generieren lässt, ist fraglich. Einen möglichen Ansatz dazu stellt Riedel unter dem Stichwort „Specification Scanner“ vor [Riedel2014].

Browse Questions und Ask a Question

Diese beiden Views sollen Möglichkeiten bieten, zusätzliche Informationen zum Testauftrag zu erhalten. Ein Worker hat dazu die Möglichkeit, bei einer Unklarheit eine Frage direkt an den Provider zu stellen (über die Ask a Question-View). Der Provider kann die Frage im Anschluss entweder über Mail oder - je nach Ausgestaltung - über die Crowdsourcing Plattform beantworten. Alle gestellten Fragen werden inklusive eventuell vorhandener Antworten in der Browse Questions-View aufgelistet. Ant-

wortet der Provider, wird der Worker direkt über Eclipse benachrichtigt. Denkbar wäre auch, dass die Crowdsourcing Plattform den Worker über eine E-Mail benachrichtigt, da nicht unbedingt davon auszugehen ist, dass dieser das Projekt ständig geöffnet hat. Inwieweit diese Funktionalität wirklich sinnvoll ist, bleibt unklar, da die Ergebnisse aus [Shah2011] darauf hindeuten, dass Entwickler oftmals nicht dazu bereit sind weitere Informationen zu liefern. Da die Autoren dieser Studie aber auch darauf hinweisen, dass beim ausgelagerten Testen fehlende Informationen zu den Hauptproblemen zählen, halte ich es trotzdem für wichtig, eine Kontaktmöglichkeit zum Test-Task-Provider bereitzustellen.

5.3. Phase 2: Recording-Perspective

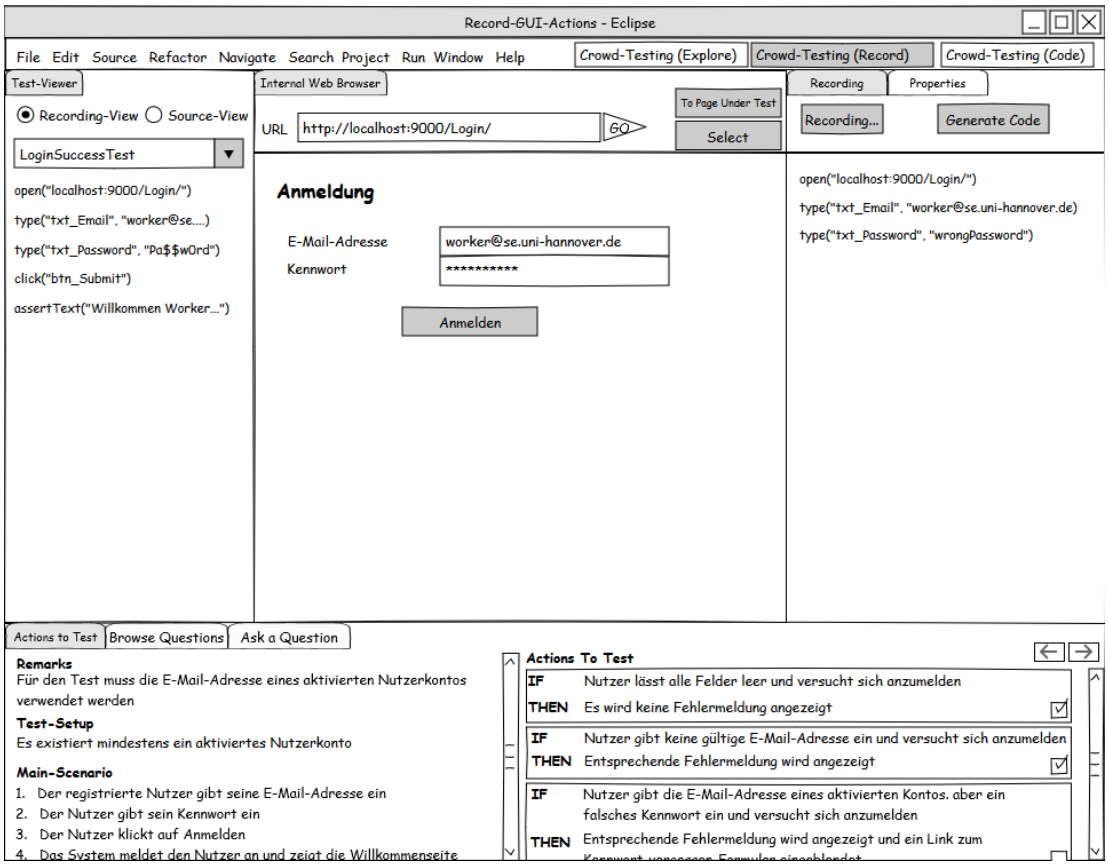

Abbildung 5.4.: Die Recording Perspektive in Eclipse mit geöffneter Actions to Test-View und Recording View. Im Test-Viewer ist der bereits vorhandene GUI-Test geöffnet, der prüfen soll, ob ein Benutzer sich am System anmelden kann.

Da ich mich in dieser Arbeit auf die Erstellung von GUI-Tests für Webanwendungen beschränke, ist die Recording Perspektive entsprechend darauf optimiert. Bei einer

Desktopapplikation müsste diese sicherlich etwas anders gestaltet werden. Einen ersten Hinweis kann hier die Testoberfläche von Visual Studio geben (vgl. Kapitel 3.4), die aber wiederum als Basis für die im Folgenden vorgestellte Perspektive diente.

Die Recording Perspektive enthält als zentrales Element einen integrierten Browser zur Anzeige der zu testenden Webanwendung. Alle Aktionen, die in diesem Browser durchgeführt werden, werden von der Entwicklungsumgebung aufgezeichnet und in der rechts befindlichen Recording-View angezeigt. Dabei werden zusammen mit der Aktion ggf. genutzte Parameter (z.B. welches Element betrifft die Aktion) angezeigt. Die Notation ist in Abbildung 5.4 an die Schreibweise von Selenium-Tests angelehnt. Gleichzeitig ist links der sogenannte Test-Viewer zu finden. Über diesen können sich im Projekt befindliche GUI-Tests angezeigt werden. Diese können entweder in der gleichen Darstellungsweise wie in der Recording-View angezeigt werden oder als Quellcodedateien. Bei ersterer Darstellungsform können ein- oder mehrere Schritte vom Worker ausgewählt und abgespielt werden, was zu den entsprechenden Aktionen im Browser führt. Dieses Vorgehen ist hilfreich, wenn der Task-Provider einen Test anfordert, der einem anderen ähnelt, wie das nachfolgende Beispiel zeigt.

> Willy soll einen Test erstellen, der bei einem Anmeldeformular prüft, ob bei einer falsch formatierten E-Mail-Adresse, die korrekte Fehlermeldung angezeigt wird. Er öffnet im Test-Viewer den mitgelieferten Test, der prüft, ob bei korrekten Eingaben im Anmeldeformular der Nutzer am System angemeldet und die Willkommensseite angezeigt wird. Er spielt den dort vorhandenen Ablauf bis auf den Schritt zum Absenden des Formulars ab. Im Browser ist das Formular nun vollständig ausgefüllt und in der Recording-View sind die entsprechenden Schritte aufgenommen. Aus dieser löscht Willy nun die Eingabe für das E-Mail-Feld und tippt im Browser eine fehlerhafte E-Mail-Adresse ein. Anschließend klickt er auf den Absenden-Button des Formulars. Die Webseite weist ihn mit einer Fehlermeldung darauf hin, dass die E-Mail-Adresse fehlerhaft ist. Willy klickt nun auf den Select-Button in der oberen Leiste des Browsers und bewegt die Maus über die Fehlermeldung, die nun blau umrahmt wird. Auf der rechten Seite neben dem Browser werden automatisch alle erkannten Eigenschaften des ausgewählten Elements angezeigt. Willy klickt auf den kleinen Button, neben der Eigenschaft, die den aktuellen Text der Fehlermeldung zeigt. Daraufhin wird ihm ein Fenster angezeigt, über das er Assertions für diese Eigenschaften erstellen kann. Willy erstellt nun eine Assertion, die auf das Vorhandensein der entsprechenden Fehlermeldung prüft. Nach einem Klick auf den „Generate Code“-Button wird aus den aufgenommen Interaktionen und Assertions automatisch der entsprechende Java-Code generiert. Damit ist dieser Teil des Testauftrags für Willy abgeschlossen.

Im vorangegangenen Beispiel ist beschrieben, wie ein Worker Assertions direkt über die IDE erstellen kann. Dies soll die Notwendigkeit für den Worker, sich mit den Elementen der GUI auf Quellcodeebene auseinandersetzen zu müssen, noch weiter reduzieren und Zeit sparen. Diese Funktionalität existiert so bereits in Visual Studio (siehe dazu Abschnitt 3.4).

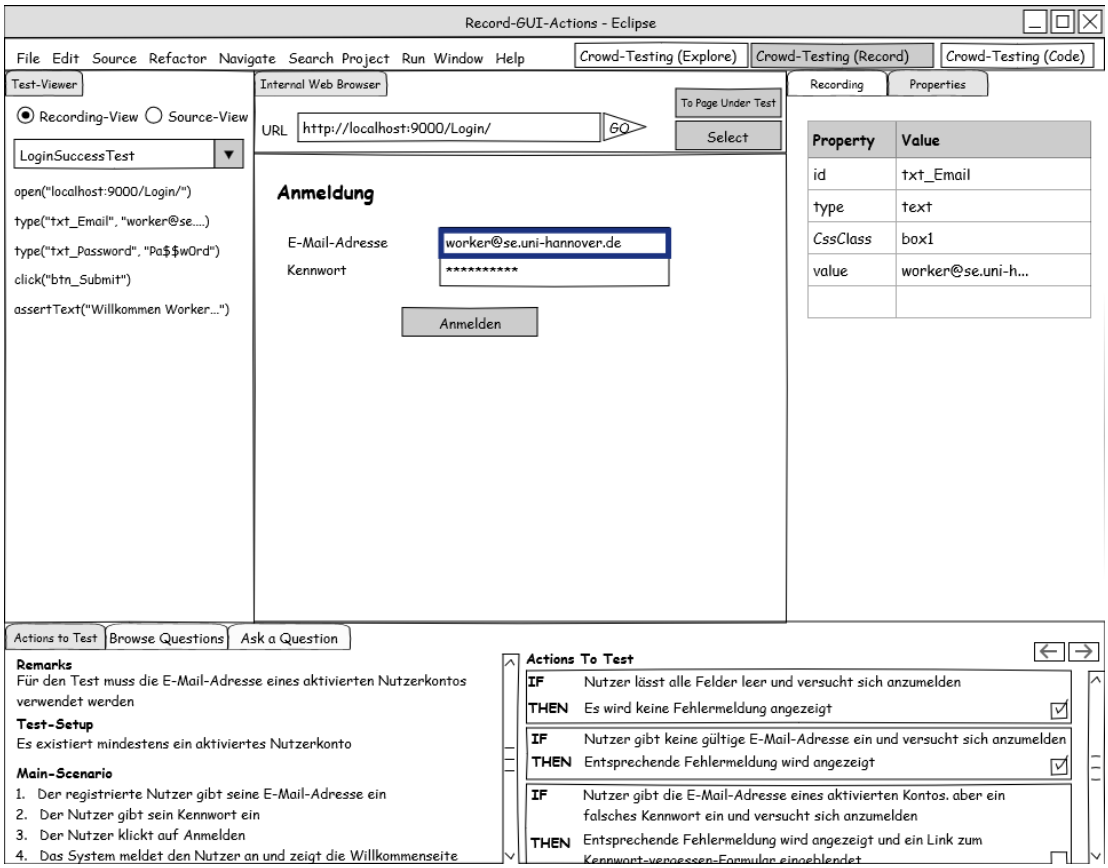

Abbildung 5.5.: Die Recording Perspektive in Eclipse mit geöffneter „Actions to Test“-View. Außerdem wurde das Feld E-Mail-Adresse im Browser ausgewählt (blauer Rahmen). Rechts sind die erkannten Eigenschaften des GUI-Elements zu sehen.

5.4. Phase 3: Coding-Perspective

Aus den in der Recording-Perspektive aufgezeichneten Schritten kann per Knopfdruck Java-Code erzeugt werden. Dieser wird in eine automatisch angelegte Datei integriert, die beim Öffnen der Coding-Perspektive angezeigt wird. Soll der Worker einen Unit-Test erstellen, könnte an dieser Stelle der Ansatz von NetBeans aufgegriffen werden. Dieses erzeugt bei Erstellung eines Unit-Tests bereits einen rudimentären Methodenrumpf für die Testmethode, in der wahrscheinlich benötigte Variablen bereits deklariert werden (vgl. Abschnitt 3.2).

Der Worker hat nun die Möglichkeit, die Details des Tests zu codieren. Dabei soll ihn die Coding-Perspektive durch einige Views unterstützen. Die Ansicht ähnelt ein wenig der „normalen“ Java-Perspektive aus Eclipse. Im Fokus steht der Java-Editor zur Quellcodeeingabe. Rechts daneben ist eine spezielle View zur Anzeige von sich im Projekt befindlichen Tests vorhanden. Diese sollen dem Worker im Idealfall als Vorlage

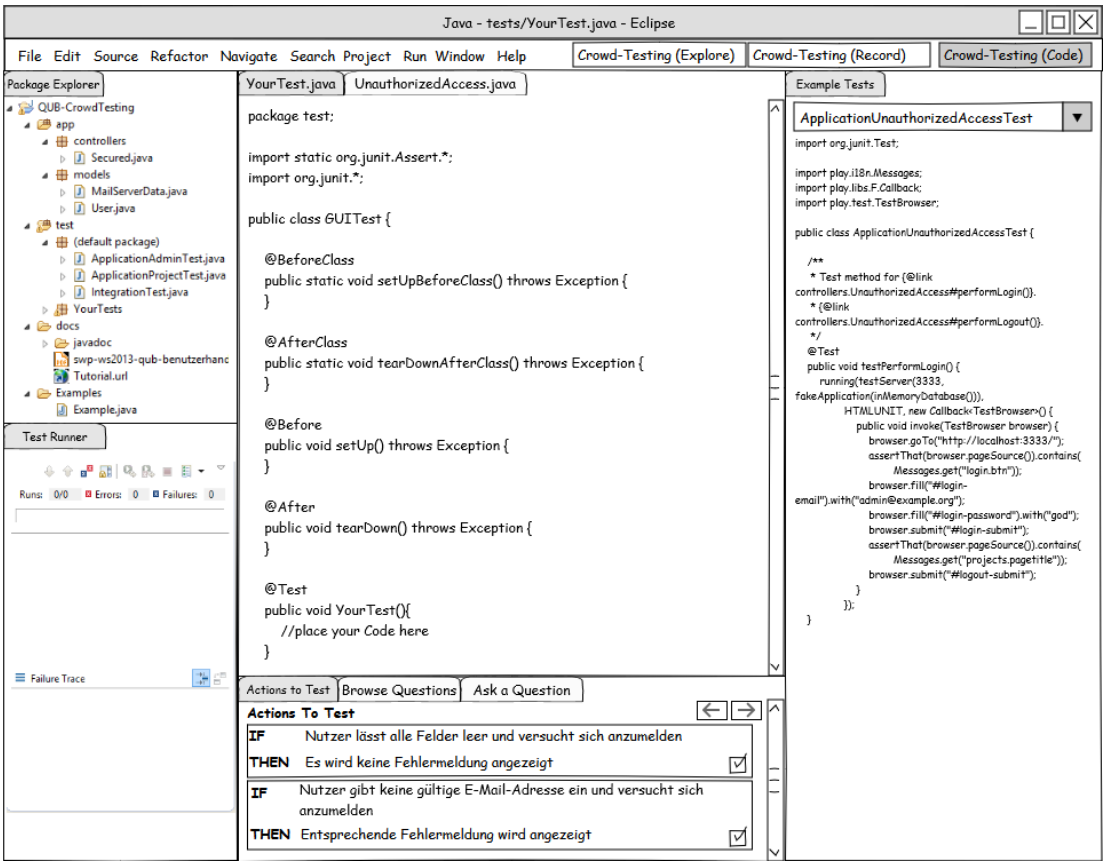

Abbildung 5.6.: Die Coding-Perspektive: rechts sind die Tests auswählbar, die durch den Provider als relevant eingestuft sind.

dienen und ihm bei der Programmierung der Details seines Tests helfen. So ist zum Beispiel vorstellbar, dass der Worker etwa die Setup-Phase seines Tests aus einem anderen Test des Projekts weitestgehend übernimmt und nur leicht für seine eigenen Zwecke modifiziert. Da die View ständig neben dem Editor eingeblendet ist, muss der Worker beim Betrachten der Beispiele nicht zwischen verschiedenen Tabs hin und her wechseln, wie es der Fall wäre, würde er die Tests einfach mit dem Editor öffnen. Die View sollte, wie der Editor selbst auch, über Syntaxhighlighting verfügen. Die dort angezeigte Testdatei kann über ein Dropdownmenü, welches die Tests des Projektes enthält, ausgewählt werden.

Darüber hinaus sollte die Perspektive über eine View zur Anzeige der Testergebnisse verfügen. Wünschenswert wäre es hier, wenn die Ergebnisse sämtlicher ausgeführter Tests in einer gemeinsamen View angezeigt werden, unabhängig davon, ob z.B. Unit-Tests oder GUI-Tests ausgeführt werden. In Abbildung 5.6 entspricht diese „Test Runner“ genannte View der JUnit-Ergebnisansicht aus Eclipse. In den in Kapitel 3 untersuchten IDEs, unterschieden sich die Ansichten zur Anzeige der Testergebnisse nur unwesentlich voneinander. Dies deutet darauf hin, dass die Ansicht, so wie sie existiert, alle notwendigen Komponenten enthält. Meines Erachtens nach, besteht

deshalb kein Grund Modifikationen daran vorzunehmen oder gar eine Implementierung zu erstellen. Daher habe ich die Ergebnis-Ansicht von JUnit aus Eclipse ohne Veränderungen übernommen.

Weitere Elemente dieser Perspektive sind der bereits vorgestellte, angepasste Package Explorer, sowie die Views zur Anzeige von Informationen zum Testauftrag und zum Durchsuchen bzw. zum Stellen einer Frage an den Provider.

5.5. Weitere Konzepte

Vermeiden von Konfiguration

In der Vision hat der Worker Willy einen Testauftrag in seiner IDE geöffnet und konnte unmittelbar mit dem Entwickeln von Tests beginnen. Er musste keinerlei Konfiguration vornehmen. Bei den betrachteten IDEs verhält sich dies anders. Möchte man dort Tests entwickeln, müssen zunächst Schritte ausgeführt werden, die nicht unmittelbar mit dem Entwickeln von Tests zusammenhängen. Bei Eclipse und IntelliJ muss zunächst eine Run Configuration angelegt werden, bevor Tests ausgeführt werden können. In NetBeans können nicht ohne weiteres Zutun mehrere Testklassen ausgeführt werden. Bei Visual Studio muss zunächst ein spezielles Testprojekt angelegt werden und darin muss das Projekt, welches die AUT enthält, referenziert werden. Im Crowdsourcing-Umfeld wäre es wünschenswert, wenn keiner dieser Schritte unternommen werden müsste und die IDE entsprechend vorkonfiguriert wäre. Eine entsprechende automatische Konfiguration ist sinnvoll, da bereits beim Öffnen eines entsprechenden Projekts feststeht, dass Tests entwickelt und diese auch ausgeführt werden sollen. Dem Worker kann an dieser Stelle wiederkehrende und überflüssige Arbeit abgenommen werden.

Ein weiterer Aspekt bezieht sich auf die Ausführung der AUT. Für das Verständnis einer Software kann es sehr hilfreich sein, diese selbst auszuführen und zu bedienen. Der Worker sollte daher die Möglichkeit haben, dies zu tun. Spätestens für das Erstellen von Oberflächentests, ist die Ausführung der Software aber unabdingbar. Daher sollte das Projekt so vorkonfiguriert werden, dass der Worker dieses direkt starten kann. Außerdem ist der Worker unter Umständen gar nicht in der Lage, eine korrekte Startkonfiguration zu erstellen, etwa wenn gewisse Startparameter festgelegt werden müssen und nicht der vollständige Quelltext zur Verfügung steht. Auch für die Ausführung notwendige Bibliotheken müssen mitgeliefert oder zumindest in einer Form angegeben sein, dass diese heruntergeladen werden können. Das Einbinden dieser Bibliotheken gehört allerdings nicht zu den Aufgaben des Workers und sollte daher auch durch die IDE erfolgen.

Vertraulichkeit von Quellcode

Im Unternehmensumfeld spielt die Vertraulichkeit des Quellcodes eine Rolle. Unternehmen wollen nicht, dass Konkurrenzunternehmen Einblicke in die Details der eigenen Software erhalten, da so unter Umständen Vorsprünge beim Know How verloren gehen können. Beim ausgelagerten Testen können sich die Unternehmen dadurch behelfen, in dem sie sich von den Unternehmen, die das Testen übernehmen, vertraglich zusichern lassen können, dass der übergebene Quellcode keiner dritten Partei zugänglich gemacht wird (siehe auch Abschnitte 2.2 und 2.3). Dies ist zwar keine Garantie für die Vertraulichkeit, jedoch können im Zweifelsfall rechtliche Schritte gegen das entsprechende Unternehmen eingeleitet werden.

Beim Auslagern in die Crowd kann die Vertraulichkeit auf eine andere Art und Weise sichergestellt werden. Der Worker erhält vom Provider nicht die vollständige AUT als Quellcode, sondern nur die Teile, die getestet werden sollen und die für das Verständnis des Workers wichtig sind. Alle anderen Bestandteile der AUT liegen dem Worker nur in kompilierter Form vor. Einschränkungen für den Worker innerhalb seiner IDE bestehen nicht. Die Syntaxergänzung und die Fehleranzeige funktionieren trotzdem, da die kompilierte Software in das Testprojekt des Workers eingebunden werden kann. So werden auch Klassen gefunden, die nur in kompilierte Form vorliegen. Diese können dann jedoch nicht als Quelltext angezeigt werden. Anders sieht dies aus, falls vor dem Kompilieren ein Obfuscator eingesetzt wird. Dies würde zwar die Vertraulichkeit der Daten ein Stück weit erhöhen, allerdings würde es dadurch für den Worker schwieriger, da er z.B. auf Syntaxergänzung bezüglich der nur binär vorhandenen Dateien verzichten müsste.

Problematisch wird es, wenn ein Unternehmen sämtliche Tests zu einem Projekt in die Crowd auslagert. Prinzipiell könnte ein Konkurrenzunternehmen dann sämtliche Testaufträge, die das Projekt betreffen, annehmen und sich so die gesamte Software Stück für Stück zusammensetzen (zumindest wenn man davon ausgeht, dass alle Bestandteile der Software getestet werden). Dieses Problem könnte eventuell auf Seite der Crowdsourcing Plattform zumindest entschärft werden. Nähere Betrachtungen müssten aber in einer Arbeit erfolgen, die sich mit der Organisation des gesamten Prozesses näher beschäftigt.

Project Workflow-View

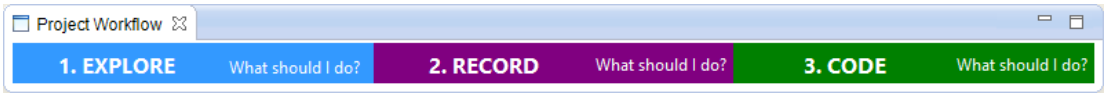

Abbildung 5.7.: Die Project Workflow-View. Bei Klick auf die farbigen Felder, öffnet sich die zugehörige Eclipse-Perspektive. Bei Klick auf „What should I do?“ öffnet sich ein Hilfe-Fenster.

„Project Workflow“ ist eine View, die in allen Perspektiven direkt unterhalb der Sym-

bolleiste von Eclipse platziert ist. Die View soll dem Worker eine Übersicht darüber geben, wie der Testprozess grundsätzlich aufgebaut ist. Sie soll in erster Linie der Orientierung dienen. Aus diesem Grund enthält die View auch Links zu Hilfefenstern. Beim Klick auf „What should I do?“ öffnet sich ein Pop-Up-Fenster, dass eine Anleitung zu der jeweiligen Phase enthält. In der Anleitung ist beschrieben, was der Worker tun und worauf er achten sollte. Diese Anleitungen sind vor allem für Entwickler gedacht, die zum ersten Mal einen Testauftrag bearbeiten. Darüber hinaus hat die View aber noch einen ganz praktischen Nutzen. Die farblich hinterlegten Felder repräsentieren einerseits die entsprechenden Phasen des Testprozesses, dienen aber gleichzeitig auch dazu, die zugeordnete Perspektive in Eclipse zu öffnen. So muss der Worker nicht den Umweg über das Anwendungsmenü nehmen. Insbesondere neue Nutzer wissen zu Beginn auch nicht unbedingt, dass es speziell designte Perspektiven für die Testerstellung gibt.

6. Realisierung der Konzepte in Eclipse

Ein wesentlicher Teil meiner Arbeit war es, die entwickelten Konzepte prototypisch in Form eines Plugins für Eclipse umzusetzen. In diesem Kapitel möchte ich darauf eingehen, wie verschiedene Konzepte realisiert wurden und an welchen Stellen der Prototyp noch unvollständig ist bzw. nur mit Mocks arbeitet. Meine Implementierung geht dabei an einigen Stellen über das hinaus, was durch den Test-Task-Editor zur Verfügung gestellt wird [Riedel2014]. Dies ist aufgrund der angestrebten Evaluation nötig geworden, damit ich während dieser, die von mir entwickelten Konzepte evaluieren konnte. Ich werde an den entsprechenden Stellen darauf hinweisen.

6.1. Übersicht über die Implementierung

Explore-Perspektive

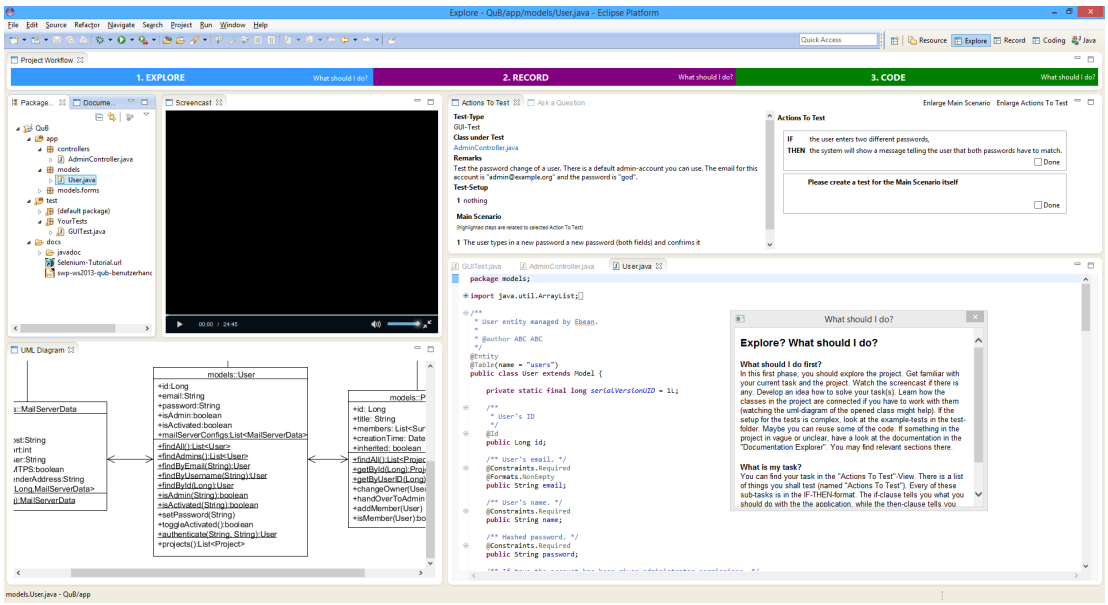

Abbildung 6.1.: Die fertig implementierte Explore-Perspektive in Eclipse mit geöffnetem Testauftrag und Hilfefenster.

Abbildung 6.1 zeigt die fertig implementierte Explore-Perspektive. Dem Package Explorer wurde ein standardmäßig aktivierter Filter hinzugefügt, der alle nicht relevanten Teile des Projekts (u.a. Bibliotheken, diverse Arbeitsverzeichnisse des Plugins) ausblendet. In der rechts daneben platzierten Screencast-View kann, sofern der aktuell

geöffnete Testauftrag einen solchen enthält, der Screencast angesehen werden. Wiederum rechts davon, befindet sich in der oberen Bildschirmhälfte die Anzeige der eigentlich zu bearbeitenden Aufgabe. Für jeden Test, der erstellt werden soll, befindet sich in der rechten Hälfte der View eine einzelne if-then-Box, die den Inhalt des Tests beschreibt. Jede dieser Boxen kann angeklickt werden, um im Main-Scenario in der linken View-Hälfte die Schritte zu markieren, die von diesem Test betroffen sind. Jede einzelne Box und damit jeder zu erstellende Test, können einzeln als abgeschlossen markiert werden (kleine Done-Checkbox unten rechts). Sowohl die linke, als auch die rechte Seite der View können ausgeblendet werden, um dem Inhalt der jeweils anderen Seiten mehr Platz zu bieten (z.B. für Bildschirme mit geringer Auflösung). Wird eine im Projekt enthaltene Klasse (keine Testklasse) geöffnet, so zeigt die View unten links das zur Klasse gehörige UML-Diagramm an. Bisher muss das UML-Diagramm im Testauftrag enthalten sein (vgl. Kapitel 6.2), sollte konzeptionell jedoch on-the-fly beim Öffnen der Klasse generiert werden.

Record-Perspektive

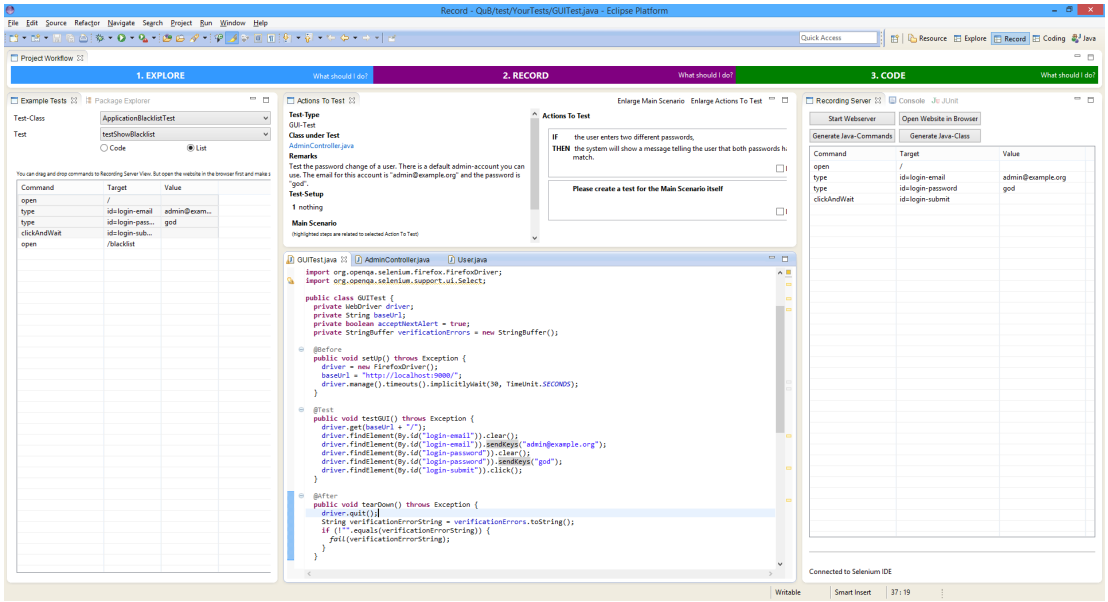

Abbildung 6.2.: Die fertig implementierte Record-Perspektive in Eclipse mit geöffnetem Testauftrag. Im Recording-Server (rechts) sind bereits einige Schritte aufgenommen. Die geöffnete Testklasse enthält den daraus generierten Java-Code.

Abbildung 6.2 zeigt die fertig implementierte Record-Perspektive. In der Example Tests-View (links) können im Testauftrag enthaltene Tests angesehen werden. Dabei kann zwischen einer Code-Ansicht und einer Listenansicht unterschieden werden. Die Code-Ansicht zeigt die vollständige Klasse bzw. Datei an. In der Listenansicht muss der Nutzer eine Testmethode (mit @Test annotierte Methode) auswählen. Er bekommt dann

die dazugehörigen Selenium-Befehle angezeigt, sofern im Testauftrag im Ordner der ausgewählten Testklasse eine Selenium-Datei (im html-Format) hinterlegt ist. Diese Datei muss dem Namensschema „[Testklasse].[Testmethode].html“ entsprechen, wobei die Klammerausdrücke Platzhalter für die entsprechenden Namen der ausgewählten Testklasse bzw. -methode sind. Die Befehle der Listenansicht können per Drag & Drop in die Recording Server-View (rechts) gezogen werden. Dazu muss allerdings eine Firefox mit entsprechend modifiziertem Selenium IDE-Plugin geöffnet sein (vgl. Kapitel 6.3). Im Recording Server kann die Position von Befehlen per Drag & Drop verändert werden. Befehle können per Druck auf die Entf-Taste gelöscht werden. Über eine Schaltfläche im oberen Bereich kann der Webserver, der die AUT hostet, gestartet werden. Der Firefox kann ebenfalls per Schaltfläche mit der Startseite der AUT aufgerufen werden. Des Weiteren kann der Nutzer, basierend auf den Befehlen im Recording Server, eine vollständige Testklasse (mit Setup, Teardown etc.) in Java oder nur die Java-Äquivalente der vorhandenen Selenium-Befehle generieren, die dann an der Cursorposition im Editor erstellt werden.

Code-Perspektive

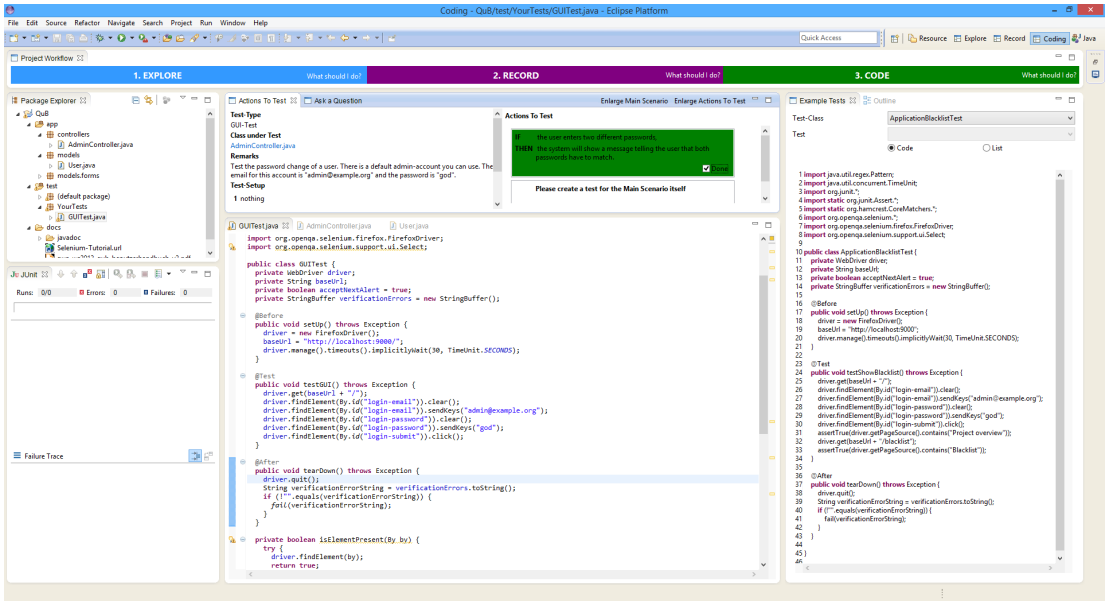

Abbildung 6.3.: Die fertig implementierte Code-Perspektive in Eclipse mit geöffnetem Testauftrag. In der Example Test-View (rechts) ist ein im Testauftrag mitgelieferter Test geöffnet. In der Actions to Test-View (Mitte oben) ist einer von zwei zu erstellenden Tests als abgeschlossen markiert.

Abbildung 6.3 zeigt die fertig implementierte Code-Perspektive. Diese enthält, neben der Anzeige des Aufgabestellung (oben) und der bereits beschriebenen Example Tests-View (rechts), den JUnit-Runner, der die Ergebnisse der Testausführung anzeigt.

6.2. Der Testauftrag

Der Testauftrag wird vom Provider mithilfe des Test-Task-Editors erzeugt. Die Ausgabe dieses Plugins ist eine zip-Datei mit der Dateiendung „.cwproject“. Eine solche Datei kann vom Plugin des Workers (also dem hier vorgestellten Prototypen) gelesen und interpretiert werden. Der Inhalt der Datei sieht wie folgt aus:

- **.task**: Eine XML-Datei, die alle von Eclipse benötigten Informationen des Testauftrags enthält. Dies umfasst die meisten in Abschnitt 4 beschriebenen Felder. Ausgenommen davon sind natürlich Quellcode, Dokumentation und Binärdaten. Die Datei dient außerdem zur Speicherung des Status von einzelnen AUT-Verhalten, für die Tests erstellt werden sollen (vgl. Abschnitt 5.2). Das Erstellen bzw. Lesen dieser Datei erfolgt über eine Java-Bibliothek, die ich gemeinsam mit Benjamin Riedel entworfen und implementiert habe. Auf diese Weise haben wir sichergestellt, dass die .task-Datei kompatibel mit beiden Plugins ist.
- **play.zip**: Eine Zip-Datei, die eine ausführbare binäre Version der AUT enthält. Die Datei wurde mit dem Distributions-Befehl des Play!-Frameworks (play dist projektname) vom Plugin des Providers erzeugt.
- **docs-Verzeichnis**: Ein Verzeichnis, das sämtliche, zum Testauftrag gehörige Dokumentation enthält. Zu jeder pdf-Datei kann außerdem eine gleichnamige .content-Datei existieren, die auf Abschnitte innerhalb der pdf-Datei verweist. Die .content-Datei enthält zeilenweise lediglich eine Seitenzahl und den Titel des Abschnitts, auf den die Seitenzahl verweist. Diese Datei ist für das Funktionieren des Documentation Explorer essentiell, jedoch werden diese Dateien bisher weder durch den Test-Task-Editor, noch durch mein Plugin erzeugt (siehe dazu auch Abschnitt 5.2).
- **screencast-Verzeichnis**: Dieses Verzeichnis enthält die Video-Datei eines vom Provider erstellten Screencasts. Diese Funktion wird bisher nicht durch den Test-Task-Editor unterstützt.
- **src-Verzeichnis**: Dieses Verzeichnis enthält die Unterverzeichnisse app und test. Ersteres enthält entweder den vollständigen Quellcode oder nur die relevanten Teile des Quellcodes der AUT, während Letzteres die für den Worker relevanten Tests enthält. Diese bestehen in der Regel lediglich aus Java-Dateien. Für das vollständige Funktionieren der im nächsten Abschnitt beschriebenen Recording-Perspektive, sind jedoch auch Selenium-Tests notwendig. Liegt im Testverzeichnis neben der Java-Datei eine gleich benannte html-Datei vor, wird diese von einer speziellen View als Selenium Test interpretiert. Dieser kann als Basis für einen zu entwickelnden Test verwendet werden (vgl. Abschnitt 5.3). Diese html-Dateien könnten einerseits vom Provider unter Nutzung der Selenium IDE erstellt und in den Testauftrag integriert werden[1], alternativ wäre aber auch eine automatische Erstellung denkbar. Dazu müssten die vorhandenen Java-Test-Klassen entsprechend geparsed werden, was an dieser Stelle aber nicht

[1] Dies ist im Test-Task-Editor bisher nicht vorgesehen.

implementiert wurde.

- **uml-Verzeichnis**: Dieses Verzeichnis enthält die gleiche Verzeichnisstruktur, wie das src/app-Verzeichnis. Für jede dort vorhandene Java-Klasse, liegt am entsprechenden Ort im uml-Verzeichnis eine gleich benannte png-Datei. Diese Datei enthält ein UML-Klassendiagramm, das die entsprechende Java-Klasse und die von dieser Klasse genutzten Klassen aufzeigt. Das Verzeichnis gehört konzeptionell eigentlich nicht in den Testauftrag. Idealerweise sollten die UML-Diagramme beim Importieren des Testauftrags in Eclipse dynamisch erzeugt werden. Da ich allerdings kein geeignetes, kostenloses Plugin bzw. Programm gefunden habe, das sich über eine API ansprechen lässt und gezielt genau die benötigten UML-Klassendiagramme erzeugt, habe ich mich für die Platzierung dieses Verzeichnisses im Testauftrag entschieden. Eine eigene Implementierung wäre zeitlich zu aufwändig gewesen und hätte auch nur einen geringen Mehrwert bedeutet. Das Funktionieren der UML-View während der Evaluation konnte durch dieses Verzeichnis auf eine einfache Weise sichergestellt werden. Die benötigten UML-Diagramme müssen so allerdings von Hand an der richtigen Stelle im Testauftrag eingefügt werden.

6.3. Recording-Perspektive

Problematisch bei der Erstellung des Prototypen war das Entwickeln der Recording-Perspektive, die die zweite Phase des Gesamtprozesses repräsentiert. Um das Funktionieren des Gesamtkonzeptes während der Evaluation überprüfen zu können, muss der Prototyp dieses Konzept natürlich möglichst vollständig implementieren. Der Aufwand für die Entwicklung einer vollständig funktionierenden, in Eclipse integrierten Testoberfläche, wie sie in Kapitel 5.1 beschrieben ist, hätte jedoch den Umfang dieser Arbeit deutlich überstiegen. Zudem existiert mit Visual Studio bereits eine meinem Entwurf sehr ähnliche Testumgebung mit erheblichem Funktionsumfang. Somit wurde bereits gezeigt, dass sich eine entsprechende Umgebung erstellen lässt.

Ich habe mich deshalb dazu entschieden, für die Recording-Perspektive zu einem gewissen Grad vom ursprünglichen Konzept abzuweichen, dafür aber einen Prototypen zu entwickeln, mit dem sich das Funktionieren des Gesamtkonzeptes überprüfen lässt. Statt selbst eine Umgebung zu entwickeln, mit der GUI-Tests aufgenommen werden können, verwende ich dafür das Open Source-Programm Selenium IDE. Selenium IDE ist ein Plugin für den Browser Mozilla Firefox, mit dessen Hilfe GUI-Tests für Webanwendungen, sogenannte Selenium-Tests, aus dem Browser heraus aufgenommen werden können.

SeleniumIDE

Wird das Selenium IDE-Plugin im Browser gestartet, werden im Anschluss alle Interaktionen mit der geöffneten Webseite, wie Anklicken von Elementen, Texteingaben oder

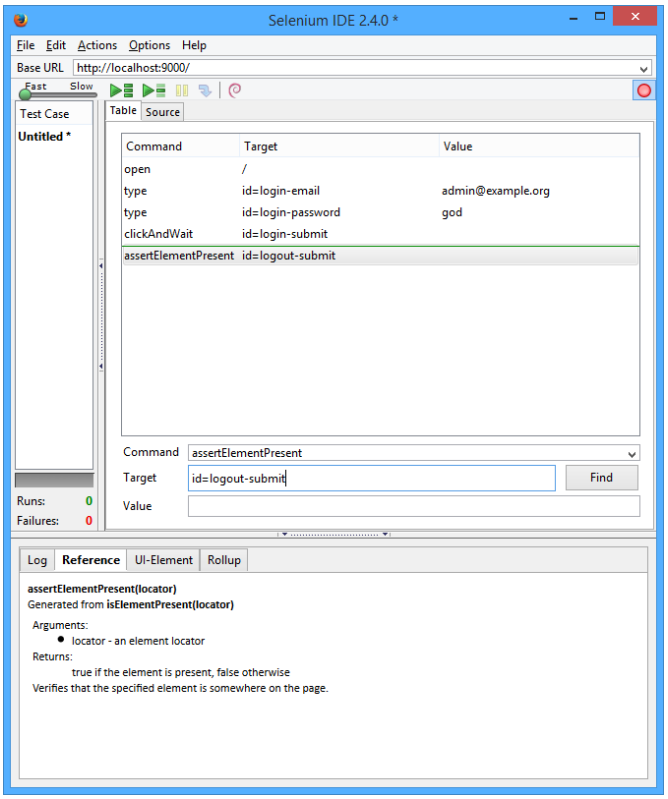

Abbildung 6.4.: Selenium IDE mit einigen aufgenommenen Interaktionen und einem eingefügten Assert.

Seitenwechsel, aufgezeichnet und im Plugin-Fenster angezeigt. Über dieses Fenster hat der Benutzer die Möglichkeit, die Aufzeichnung nachträglich zu bearbeiten. Dabei können Befehle modifiziert, ihre Reihenfolge verändert oder neue Befehle eingefügt werden. Auch Assertions können über das Menü in den Test eingefügt werden. Das Plugin ermöglicht es weiterhin, Tests für eine mögliche spätere Verwendung abzuspeichern. Aus dem aufgenommenen Testablauf kann anschließend Quellcode für verschiedene Programmiersprachen erzeugt werden. Bei der Installation des Plugins werden sogenannte Formatter für die Sprachen C#, Python, Ruby und Java mitgeliefert. Der Aufbau des jeweils generierten Quellcodes kann dabei über ein Optionsmenü angepasst werden.

Bei Selenium-IDE kann der Nutzer auf zwei Arten Quellcode in der gewünschten Sprache generieren. Die erste Möglichkeit ist der Export des Tests über das Menü des Plugins. Der Nutzer wählt dazu lediglich einen Speicherort für den Quellcode aus, woraufhin dieser an der entsprechenden Stelle erzeugt wird. Der zweite Weg Programmcode zu generieren, ist die Zwischenablage. Der Nutzer wählt in der Liste die gewünschten Einträge des Tests aus und drückt Strg + C oder wählt Kopieren im Kontextmenü aus. Der entsprechende Quellcode befindet sich dann in der Zwischenablage des Computers und kann vom Nutzer an jede gewünschte Stelle kopiert werden. Beide Möglichkeiten wären zwar auch für eine Anbindung an Eclipse prinzipiell praktikabel,

sind aber für den Nutzer umständlich, insbesondere wenn der Nutzer den Test nach Erzeugung des Quellcodes im Browser modifiziert. Außerdem können so bereits vom Provider mitgelieferte Tests (bzw. Auszüge daraus) nur über Umwege genutzt werden.

Die grundsätzlich vorhandenen Möglichkeiten von Selenium IDE decken bereits einen großen Teil des benötigten Funktionsumfangs für den Eclipse-Prototypen ab. Nachteilig ist jedoch, dass Selenium IDE ein Firefox-Plugin ist und daher ein Teil der Testentwicklung nicht mehr - wie von meinem ursprünglichen Konzept vorgesehen - innerhalb der Eclipse-Entwicklungsumgebung abläuft, sondern in einem externen Browserfenster. Dies sehe ich, gerade was den Arbeitsfluss des Workers angeht, als kritisch an, da dieser so gezwungen ist, zwischen verschiedenen Fenstern hin und her zu wechseln. Um die Nachteile, die aus der Verwendung des Plugins resultieren, möglichst auszugleichen, habe ich mich dazu entschieden, eine direkte Verbindung zwischen Selenium IDE und Eclipse herzustellen.

Verbindung von Selenium IDE und Eclipse

Es stellt sich nun zunächst einmal die Frage, wie umfangreich eine Verbindung zwischen beiden Programmen gestaltet sein sollte. Maßgeblich ist hierbei meiner Meinung nach wieder die Aufgabenstellung dieser Arbeit, die besagt, dass die Einstiegshürde für den Worker möglichst gering ausfallen soll. Die erste Hürde, die ein Worker zu überwinden hat, ist der Start des Plugins, bzw. des Browsers. Ohne Integration in Eclipse müsste ein Worker zunächst einmal das Wissen erlangen, dass er bei Nutzung des Prototypen in der zweiten Phase des Testprozesses mit dem Browser bzw. dem dort integrierten Plugin weiterarbeiten sollte. Dies kann dem Worker abgenommen werden. Dazu habe ich in Eclipse eine Schaltfläche zum Öffnen der Website Under Test platziert. Beim Betätigen der Schaltfläche öffnet sich der Firefox-Browser auf der Startseite der Website, für die der Worker Tests erstellen soll. Zudem öffnet sich jetzt direkt beim Starten des Browsers das Selenium IDE-Fenster.

Im nächsten Schritt möchte der Worker unter Umständen einen im Testauftrag mitgelieferten Test als Basis für seinen Test verwenden. Für Eclipse ist dazu im Konzept für die Recording-Perspektive zum einen eine View zur Anzeige bereits existierender Tests vorgesehen („Example Tests“-View) und zum anderen eine View, die den Ablauf des aktuellen, gerade zu erstellenden Tests zeigt. Befehle des vorhandenen Tests sollten dann per Drag & Drop in den aktuellen Test gezogen werden können. Dieses Konzept behalte ich an dieser Stelle bei, erweitere es aber um eine Synchronisation der Eclipse-View, die den aktuellen Testaufbau anzeigt, mit der Selenium IDE. Das bedeutet, wenn der Worker per Drag & Drop einen oder mehrere Befehle aus einem bestehenden Test in die Ansicht des aktuellen Tests zieht, werden diese Befehle auch im Selenium IDE-Fenster im Browser angezeigt. Er kann dann über das Selenium Plugin diese Schritte ausführen lassen und anschließend testspezifische weitere Interaktionen aufnehmen. Diese weiteren Interaktionen, die im Browser aufgezeichnet werden, werden automatisch an Eclipse übertragen. Dadurch hat der Worker jederzeit die Möglichkeit, direkt in Eclipse, aus dem aufgezeichneten Testablauf (Java-)Quellcode zu generieren. Er muss nicht erst über das Selenium-Plugin den entsprechenden Test

an die richtige Stelle im Eclipse-Projekt exportieren. Dabei kann der Worker entscheiden, ob er eine vollständige Java-Klasse (inklusive der übersetzen Selenium-Befehle) mit dem für Selenium-Tests nötigen Setup generieren will, oder nur die Übersetzung der Selenium-Befehle in den Code generieren möchte.

Technische Umsetzung

Die Kommunikation zwischen Selenium IDE und Eclipse basiert auf einer lokalen Netzwerkverbindung zwischen beiden Programmen. Dazu habe ich zunächst das auf Java Script, XUL[2] und der Firefox-API basierende Plugin erweitert. Beim Aufruf des Plugins, welcher nun zusammen mit dem Browser-Start erfolgt, öffnet das Plugin einen Server-Socket, mit dem sich Eclipse automatisch innerhalb von fünf Sekunden verbindet. Über ein einfaches, selbst entwickeltes Kommunikationsprotokoll können so Nachrichten zwischen Eclipse und Selenium IDE ausgetauscht werden (siehe auch Abbildung 6.5). Dieses Kommunikationsprotokoll unterstützt im Wesentlichen drei verschiedene Nachrichtentypen:

- **Selenium-Befehle (bidirektional)**: Ein Selenium-Befehl besteht, je nach Art des Befehls, aus einem Zwei- oder Dreitupel. Die erste Komponente gibt den Befehlstyp an (also z.B. Öffnen einer Seite, Tastatureingabe oder Mausklick), die Zweite das Ziel des Befehls (also z.B. ein Formularfeld), während die dritte Komponente den Wert des Befehls festlegt (bei einer Tastatureingabe beispielsweise der eingegebene String). Selenium-Befehle (bzw. deren Reihenfolge) können sowohl in Selenium IDE, als auch in Eclipse modifiziert werden. Aus diesem Grund wird dieser Nachrichtentyp bidirektional versendet, also von Eclipse zur Selenium IDE und umgekehrt.
- **Testbefehle in Java (nur Selenium IDE zu Eclipse)**: Hierbei werden aus den Selenium-Befehlen, die in Selenium IDE vorhanden sind, über den Java-Formatter von Selenium, Java-Statements generiert. Ein Selenium-Befehl entspricht dabei meist einem einzigen Java-Statement.
- **vollständige Testklasse in Java (nur Selenium IDE zu Eclipse)**: Bei diesem Nachrichtentyp wird eine vollständige Java-Klasse aus den vorhandenen Selenium-Befehlen generiert und übertragen. Die Klasse enthält alle notwendigen Elemente (wie Imports, Variablen- und Methodendeklarationen, Annotationen), die für die Ausführung in JUnit notwendig sind. Die Java-Befehle des vorherigen Nachrichtentyps sind ebenfalls bereits in die Klasse integriert.

Wie aus der Aufzählung hervorgeht, werden nur Selenium-Befehle in beide Richtungen (Eclipse zu Selenium-IDE und Selenium-IDE zu Eclipse) versendet. Dies hat einen einfachen Grund. Die Selenium-Befehle bilden die Grundlage für den Test. Wird in Eclipse der Test modifiziert, werden alle Selenium-Befehle des Tests an die Selenium-IDE gesendet. Die Reaktion darauf besteht aus zwei Schritten. Zunächst einmal werden die empfangenen Befehle in das interne Datenmodell integriert und die Änderungen in

[2] Eine auf XML basierende Beschreibungssprache zur Gestaltung von Benutzeroberflächen.

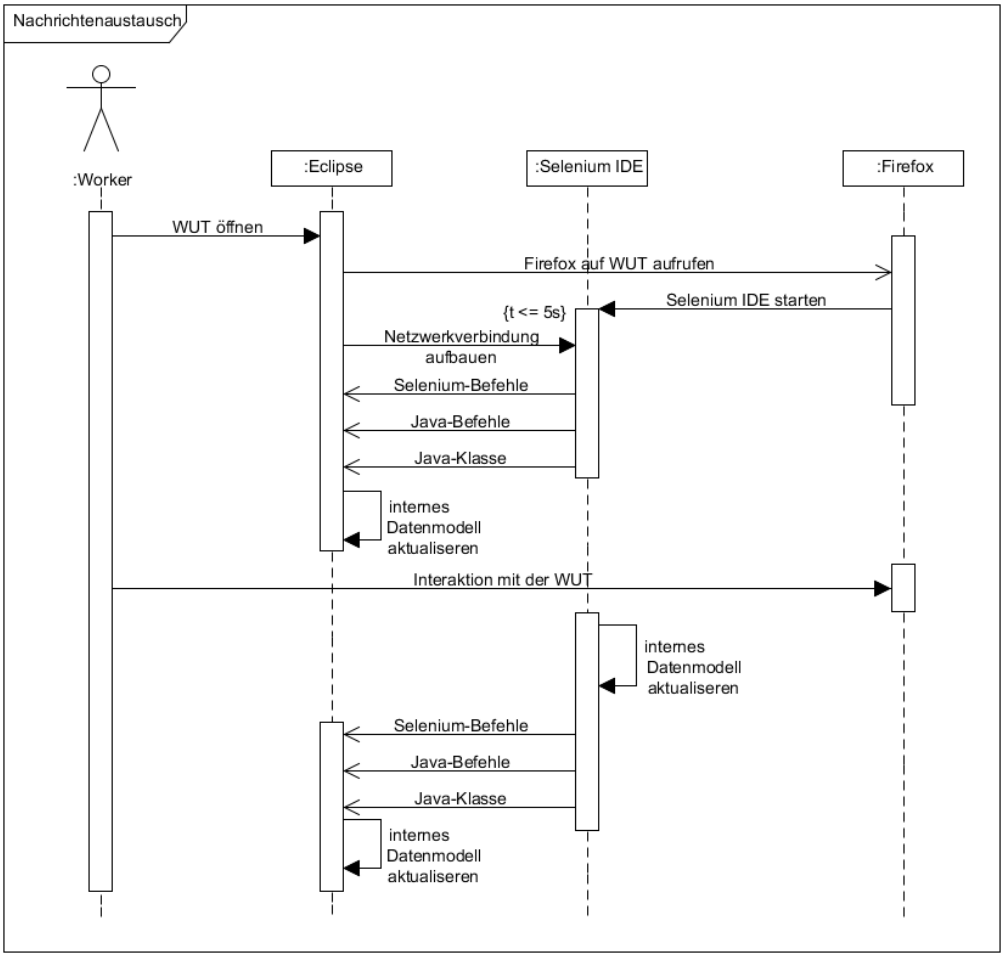

Abbildung 6.5.: Dargestellt ist der Verbindungsaufbau zwischen Eclipse und Selenium IDE. Weiterhin ist der Nachrichtenaustausch bei Modifikation der Webpage Under Test (WUT) dargestellt.

der Benutzeroberfläche dargestellt. Anschließend sendet Selenium IDE die Selenium-Befehle, die Testbefehle in Java und eine vollständige Java-Klasse an Eclipse zurück.[3] Diese Daten werden nun in Eclipse vorgehalten. Will der Worker in Eclipse auf den Java-Code des Tests zugreifen, muss dieser nur noch an die gewünschte Stelle ausgegeben werden.

[3] Die Rückübertragung der unmodifizierten Selenium-Befehle an Eclipse ist an dieser Stelle redundant. Der Vorteil ist allerdings eine einfachere Struktur innerhalb des Plugins. Dieses sendet alle drei Nachrichtentypen an Eclipse, sobald der Test modifziert wurde. Dadurch muss nicht unterschieden werden, ob die Änderung durch Eclipse ausgelöst wurde oder durch eine Benutzerinteraktion mit der Selenium IDE-Oberfläche.

6.4. Vermeidung von Konfiguration

Wie bereits in Abschnitt 5.5 erwähnt, ist es wünschenswert, dem Worker jedwede Konfiguration des Projekts abzunehmen. Dazu zählt unter anderem die Entwicklungsumgebung so einzurichten, dass der Worker die AUT starten kann. In Eclipse muss dafür eine sogenannte Run Configuration erstellt werden. Diese legt u.a. fest, welche Klasse als Startpunkt dienen soll, wo benötigte Bibliotheken sind oder mit welchen Parametern das Programm gestartet werden soll. Da ich mich in diesem Prototypen auf Play!-Anwendungen beschränke, ist das Anlegen einer solchen Konfiguration nicht schwierig. Die Play!-AUT wird durch den Test-Task-Editor immer im gleichen vorkompilierten Format im Testauftrag mitgeliefert. Daher muss in der Run Configuration lediglich der Play!-eigene Webserver als Startklasse festgelegt werden.

Etwas schwieriger ist das Einbinden der Bibliotheken. Play! benötigt zum einfachen Starten bereits eine Vielzahl von Bibliotheken. Dazu kommen noch einmal vom Entwickler zusätzlich verwendete Bibliotheken. Die im Testauftrag enthaltene, von Play! erzeugte zip-Datei enthält allerdings all diese Dateien und weist auch immer die gleiche Struktur auf. Daher können einfach alle Bibliotheken, die in einem bestimmten Ordern innerhalb der zip-Datei vorhanden sind, in das Projekt eingebunden werden und alle Abhängigkeiten des Projekts können danach korrekt aufgelöst werden.

Entfiele die Beschränkung auf ein bestimmtes Framework, wäre eine Erweiterung des Testauftrags notwendig. In diesem Fall müssten innerhalb des Test-Task-Editors bei der Erstellung des Testauftrags sämtliche Abhängigkeiten bestimmt und die entsprechenden Bibliotheken in einer vordefinierten Form mitgeliefert werden. Auch die Parameter zur Erzeugung einer Run Configuration müssten erfasst werden.[4]

Bereits beim Anlegen des Test-Projekts steht fest, dass der Worker seine Tests auch ausführen wird. Daher kann dem Worker die Aufgabe abgenommen werden, eine Run Configuration zur Ausführung seiner Tests zu erstellen. Durch die Beschränkung auf das Play!-Framework fallen die Tests JUnit-kompatibel aus. Daher kann eine Run Configuration für JUnit vordefiniert werden. Diese bezieht sich auf ein Verzeichnis namens „YourTests“ im Projektordner, in dem der Worker seine Tests anlegen soll. Dadurch werden ausschließlich die vom Worker angelegten Tests gestartet. Beachtet werden müssen noch die Abhängigkeiten bezüglich der Testbibliotheken. Da JUnit allerdings bei den Java unterstützenden Versionen von Eclipse integriert ist, kann diese Abhängigkeit durch Eclipse selbst aufgelöst werden. Beschränkt man sich an dieser Stelle nicht auf JUnit, müssten auch hier Erweiterungen des Testauftrags erfolgen.

6.5. Einschränkungen des Prototyps

In diesem Abschnitt möchte ich kurz auf Einschränkungen des Prototypen eingehen, die bisher keine Erwähnung fanden.

[4] Dies könnte durch Auslesen der vom Provider genutzten Run Configuration geschehen.

- **Browse Questions**: Diese View konnte nicht implementiert werden, da sie technisch auf die Struktur einer Crowdsourcing Plattform angewiesen ist, welche bisher aber nicht existiert.
- **Ask a Question**: Diese View ist vollständig implementiert. Sie verwendet allerdings einen SMTP-Mail-Server, der über die Eclipse-Einstellungen vom Nutzer festgelegt werden muss. Idealerweise sollte hier ein Mailserver der unterstützten Crowdsourcing Plattform Verwendung finden.
- **Unterstützte Dateitypen im Documentation Explorer**: Der Documentation Explorer unterstützt lediglich die Verwendung von pdf-Dateien. Als Reader werden bisher nur der Adobe Reader, sowie der Foxit Reader unterstützt, da sie beide einheitliche Kommandozeilenaufrufe zum Öffnen bestimmter Seiten in pdf-Dateien verwenden.[5] Der Pfad zum Reader muss in den Einstellungen von Eclipse angegeben werden. Wünschenswert wäre hier einerseits eine automatische Erkennung des vorinstallierten pdf-Readers, andererseits die Unterstützung von weiteren Dateiformaten, wie z.B. von Office-Dokumenten.
- **Import des Testauftrags**: Ein Testauftrag wird bisher über einen Import-Assistenten in den Eclipse-Workspace geladen. Bei diesem Vorgehen müsste der Worker einen Testauftrag mit seinem Browser von der Crowdsourcing Plattform herunterladen, um die entsprechende Datei dann importieren zu können. Dies ist ein durchaus valider Ablauf. Denkbar wäre aber auch, Projekte über eine in Eclipse integrierte View anzuzeigen und direkt darüber herunterladen zu können.
- **Upload des Ergebnisses**: Hat der Worker den Testauftrag gelöst, muss dieser auf die Crowdsourcing Plattform hochgeladen werden. Dies könnte natürlich über einen Browser erfolgen. Ideal wäre hier allerdings, wenn das Ergebnis direkt über Eclipse hochgeladen werden könnte. Dies erfordert jedoch Wissen über die bisher nicht existente Crowdsourcing Plattform.

[5] Die Reader können unter `http://get.adobe.com/de/reader/` bzw. `http://www.foxitsoftware.com/Secure_PDF_Reader/` heruntergeladen werden.

7. Evaluation

Wie in Kapitel 1.4 erläutert, sollte in dieser Arbeit eine Entwicklungsumgebung konzipiert und implementiert werden, die es einem Entwickler ermöglicht, automatisierte Tests für ein ihm unbekanntes Projekt zu erstellen. Um zu überprüfen, ob das im Rahmen dieser Arbeit entwickelte Plugin dazu beitragen kann, soll an dieser Stelle eine Evaluation erfolgen. Durch diese sollen die folgenden Fragen geklärt werden:

1. Beschleunigt die Verwendung des Plugins einen Entwickler/Worker bei der Bearbeitung eines Testauftrags?
2. Ist die Dreiteilung des Testprozesses sinnvoll bzw. wird das Konzept von den Workern angenommen?
3. Welche Aspekte des Plugins funktionieren, bzw. wo besteht Verbesserungsbedarf?

7.1. Aufbau und Ablauf

Die Evaluation wurde im letzten Monat der Arbeit vorbereitet, durchgeführt und ausgewertet und verwendet dabei die aktuellste Version des hier vorgestellten Plugins. Die Durchführung, sowie Teile der Vorbereitung erfolgten in Zusammenarbeit mit Benjamin Riedel, der für seine Arbeit [Riedel2014] ebenfalls eine Evaluation durchgeführt hat. Die ausgewählten Probanden nahmen jeweils an beiden Evaluationen teil.

Die Evaluation wurde nach dem Between-Group Design gestaltet. Die Probanden wurden nach dem Zufallsprinzip einer von zwei Gruppen zugewiesen, wobei darauf geachtet wurde, dass am Ende beide Gruppen gleich groß waren. Die erste Gruppe sollte die gestellte Aufgabe mit dem hier vorgestellten Eclipse-Plugin lösen (Plugin-Gruppe), während die zweite Gruppe (die Kontrollgruppe) die Aufgabe ohne Plugin, mit einem nicht veränderten Eclipse, zu lösen hatte.

Aufgabenstellung

Beide Gruppen sollten die gleiche Aufgabe lösen. Diese sah es vor, GUI-Tests mit Selenium für eine bereitgestellte, den Probanden unbekannte, auf dem Play!-Framework basierende Webapplikation zu erstellen. Die Tests sollten dabei zwei Sachverhalte überprüfen:

1. Kann ein Benutzer sein Kennwort ändern?

2. Zeigt das System eine Fehlermeldung an, wenn das eingegebene Kennwort und seine Wiederholung nicht übereinstimmen?

Der Plugin-Gruppe wurde die Aufgabe in Form eines Testauftrags vorgelegt, während die Kontrollgruppe die Aufgabenstellung in Form eines kurzen Textes erhielt (siehe Anhang A.3). Um die Frage nach der Dreiteilung des Testprozesses beantworten zu können, wurde ein GUI-Test als Testart ausgewählt. Bei einem Unit-Test hätte die gesamte zweite Perspektive des Plugins nicht evaluiert werden können, da sie - zumindest in der aktuellen Version - für Unit-Tests irrelevant ist.

Ablauf

Der Ablauf der Evaluation war bei allen Probanden der gleiche und sah wie folgt aus:

1. Begrüßung
2. Ausfüllen des Demographie-Fragebogens (siehe Anhang A.3)
3. Evaluation Benjamin Riedel (siehe [Riedel2014])
4. kurze Erklärung zum weiteren Ablauf
5. Ansehen des Tutorialvideos
6. Klärung von eventuell aufgetretenen Fragen zum Tutorialvideo
7. Bearbeitung der Aufgabenstellung
8. Interview (nur Plugin-Gruppe)
9. Verabschiedung/Überreichen eines Schokoladenosterhasen zum Dank

Nach einer kurzen Begrüßung wurden die Probanden gebeten, einen Demographie-Fragebogen auszufüllen, in dem sie ihre eigenen Fähigkeiten auf einer Likert-Skala bewerten sollten. Anschließend fand die Evaluation durch Benjamin Riedel statt. Diese musste zwingend vor der meinigen geschehen, um ein Bias auf seiner Seite zu vermeiden. So hätte die Verwendung des hier vorgestellten Plugins den Probanden aufgezeigt, wie ein Testauftrag sinnvollerweise aufgebaut wird und wie ein solcher dem Empfänger präsentiert wird. Dies hätte höchst wahrscheinlich die Art und Weise beeinflusst, wie die Probanden auf sein Plugin reagiert hätten. Der detaillierte Ablauf dieser Evaluation ist in [Riedel2014] beschrieben.

Nach Abschluss dieses ersten Teils, wurden die Probanden kurz über den weiteren Ablauf informiert. Ihnen wurde im Anschluss daran ein Tutorialvideo gezeigt. Dazu habe ich mich aufgrund der Aufgabenstellung und der darin enthaltenen Verwendung von Selenium entschlossen. Ich ging vor der Evaluation nicht davon aus, eine ausreichende Zahl von Probanden zu finden, die Grundkenntnisse über Selenium-Tests besitzen. Das Zeigen des Videos sollte es Probanden, die bisher keine Kenntnisse über Selenium hatten, ermöglichen, die gestellte Aufgabe überhaupt in akzeptabler Zeit lösen zu können. Die Form des Tutorialvideos habe ich gewählt, um für alle Probanden möglichst gleiche Voraussetzungen zu schaffen. Die Probanden der Plugin-Gruppe erhielten dabei ein anderes Video als die Probanden der Kontrollgruppe. Das

Video der letzteren Gruppe erläutert ausschließlich die grundsätzliche Funktionsweise von Selenium IDE (vgl. Abschnitt 6.3) und wie der damit erzeugte Java-Code in Eclipse verwendet werden kann. Der Plugin-Gruppe wurde ein Video gezeigt, welches die Verwendung des hier vorgestellten Eclipse-Plugins und das damit einhergehende Zusammenspiel mit Selenium IDE erläutert. In beiden Videos wurde beispielhaft der gleiche Test erstellt. Das Tutorialvideo der Plugin-Gruppe war 9:56 Minuten lang, während das der Kontrollgruppe 6:41 Minuten lang war.

Nachdem die Probanden das Video vollständig angeschaut hatten, hatten sie die Möglichkeit, Fragen bezüglich des Gesehenen zu stellen, die ihnen von mir auch beantwortet wurden. Unmittelbar danach wurden die Probanden aufgefordert, die eigentliche Aufgabenstellung zu bearbeiten. Dazu war bei beiden Gruppen das zu bearbeitende Projekt in Eclipse importiert und geöffnet. Beiden Gruppen stand ein Firefox mit installierter Selenium IDE zur Verfügung. Die Probanden hatten während der gesamten Bearbeitungszeit die Möglichkeit, auf das Tutorialvideo zurückzugreifen und Fragen bezüglich Selenium zu stellen. Der Zugriff auf das Internet war ebenfalls möglich. Falls Probanden an einer Stelle mit Selenium nicht weiter kamen, habe ich unterstützende Hinweise gegeben. Der Kontrollgruppe standen darüber hinaus noch zwei Hilfen zur Verfügung. In einem Verzeichnis auf der Festplatte standen eine Batch-Datei, die den für die Ausführung der Webseite benötigten Server startete, und eine Verknüpfung (.lnk-Datei) zur Startseite der Webapplikation zur Verfügung. Auf beide Hilfen wurde auf dem Aufgabenblatt hingewiesen (siehe A.3). Die Dauer, die die Probanden für die Bearbeitung der Aufgabe benötigten, wurde jeweils gemessen und erfasst. Die Zeitmessung wurde beendet, sobald die Probanden der Meinung waren, die Aufgabe abgeschlossen zu haben.

Im Anschluss daran wurden die Probanden der Plugin-Gruppe interviewt. Die Fragen, die dabei gestellt wurden, sind im Anhang zu finden (siehe Abschnitt A.4). Diese dienten hauptsächlich der Klärung von Frage zwei und drei der Evaluation. Am Ende wurden die Probanden verabschiedet und erhielten als Belohnung einen Schokoladenosterhasen.

7.2. Ergebnisse

An der Evaluation nahmen insgesamt zehn Personen teil. Darunter befanden sich sechs Informatik-Studenten, drei wissenschaftliche Mitarbeiter des Fachgebiets Software Engineering der Leibniz Universität Hannover und ein professioneller Softwareentwickler.

Eine ideale Stichprobe für die Evaluation hätte lediglich aus erfahrenen Softwareentwicklern bestanden, die auf technischer Ebene bereits umfangreiche Erfahrungen mit den in der Aufgabestellung der Evaluation verwendeten Technologien aufgewiesen hätten. Solche Personen hätten dem Nutzerbild, welches in der Aufgabenstellung dieser Arbeit beschrieben wird, am ehesten entsprochen. Eine solche Stichprobe stand jedoch nicht zur Verfügung, weshalb Personen für die Evaluation ausgewählt wurden, die grundsätzlich Erfahrung in der Softwareentwicklung besitzen.

Attribut	Ø Plugin-Gruppe	Ø Kontrollgruppe	Ø Geamt
Alter (in Jahren)	27,4	32,2	29,8
Programmiererfahrung	3,8	4,4	4,1
Testerfahrung	2,6	3,4	3
Erfahrung mit Eclipse	3,8	3,4	3,6
Erfahrung mit Unit-Tests	2,6	3,2	2,9
Erfahrung mit Selenium-Tests	1,4	1,4	1,4
Bearbeitungszeit (in Minuten)	24:41	23:52	24:16

Tabelle 7.1.: Ergebnisse des Demographie-Fragebogens und Bearbeitungszeiten für die Aufgabe. Wertebereich bei den Erfahrungen: 1 = keine Erfahrung, 5 = viel Erfahrung

Die Werte in Tabelle 7.1 zeigen ein großes Problem der durchgeführten Evaluation auf. Zwar hatten die Probanden große Erfahrung beim Programmieren (4,1) und auch der Umgang mit Tests war den meisten nicht unbekannt. Jedoch war entweder gar keine oder nur wenig Erfahrung mit Selenium-Tests vorhanden (bei den Probanden kamen nur die Werte 1 und 2 vor). Da aber gerade das Erstellen eines solchen Tests die zentrale Aufgabe war, ist die ermittelte Bearbeitungszeit nur wenig aussagekräftig. Sie hängt zu wesentlichen Teilen davon ab, wie gut der jeweilige Proband mit Selenium zurechtkam. So konnte der schnellste Proband, der keine Erfahrung mit Selenium hatte, die Aufgabe in 12:45 Minuten vollkommen ohne Hilfestellung lösen[1], während der langsamste Proband mit Erfahrung für die Aufgabe 29:19 Minuten[2] benötigte und dabei Hilfestellung erhielt. Außerdem repräsentiert die Stichprobe mit ihrer geringen Erfahrung bezüglich Selenium-Tests die eigentliche Population nicht besonders gut. Laut Aufgabenstellung dieser Arbeit, würden solche Testaufträge von Personen gelöst, die grundsätzlich über Erfahrung mit den verwendeten Technologien besitzen. Die Aussage der Bearbeitungszeit wird weiterhin dadurch geschmälert, dass die Kontrollgruppe insgesamt etwas älter war und über einen etwas größeren Erfahrungsschatz verfügte (zumindest laut Selbsteinschätzung), als die Plugin-Gruppe.

Ein weiteres Problem bezüglich der Aussagekraft der hier ermittelten Zeiten, ist die Aufgabe, die den Probanden gestellt wurde. Diese sah es vor, dass ein reiner GUI-Test erstellt wird. Ein Problem, das dabei auftritt, ist der Datenbankzustand. Laut Aufgabenstellung sollte der Test prüfen, ob die Änderung des eigenen Kennworts ordnungsgemäß funktioniert. Dazu musste der Test einen Login enthalten. Wurde der Test dann einmal erfolgreich ausgeführt, so besaß das im Test verwendete Benutzerkonto danach ein anderes Kennwort als zu Beginn der Ausführung, sodass eine erneute Ausführung des Tests fehlschlug. Das Problem wurde von allen Probanden erkannt. In einer realen Situation hätte daher im Testsetup der Datenbankzustand entsprechend so modifiziert werden müssen, dass der Test auch bei mehrmaliger Durchführung funktioniert. Dafür hätte sich der Testentwickler jedoch in den Quellcode der Applikation einarbeiten müssen, um auf die Datenbank zuzugreifen. Bei dieser Tätigkeit hätte ein

[1] Dies war mit großem Abstand die insgesamt schnellste Zeit aller Teilnehmer.

[2] Diese Zeit müsste eigentlich noch höher liegen. Der entsprechende Proband hat nur einen Teil der Aufgabe gelöst, wollte aus Zeitgründen den anderen Teil aber nicht weiter bearbeiten.

Entwickler in der Plugin-Gruppe Unterstützung durch das in der Arbeit vorgestellte Plugin erhalten, während einem Entwickler in der Kontrollgruppe diese Unterstützung nicht zur Verfügung gestanden hätte.

Ein weiterer Aspekt, der in einer realen Situation die Bearbeitungszeit beeinflusst hätte, wäre die Konfiguration des Eclipse-Projekts gewesen. Entwickler in der Plugin-Gruppe hätten das entsprechende Projekt lediglich importieren müssen, während es für Entwickler der Kontrollgruppe nötig gewesen wäre, die benötigten Bibliotheken korrekt in das Projekt einzubinden.

Sowohl für die nicht durchzuführende Modifikation der Datenbank, als auch für die Vermeidung des Projektimports und ggf. anschließender Konfiguration, habe ich mich bei der Planung der Evaluation bewusst entschieden. Dies sollte die bei der Bearbeitung der Aufgabe auftretenden Probleme begrenzen, um den zeitlichen Rahmen der Evaluation nicht zu sehr auszuweiten. Trotz dieser Einschränkungen haben die Probanden im Schnitt bereits etwa 50 bis 60 Minuten für meinen Teil der Evaluation benötigt.[3] Dazu kamen noch einmal etwa 20-30 Minuten für die Bearbeitung des Evaluationsteils von Benjamin Riedel, sodass die Gesamtzeit, die die Probanden für die Evaluation aufgewendet haben (inklusive Begrüßung und Verabschiedung) im Schnitt bei etwa 80 Minuten lag.

Insgesamt lässt sich daher sagen, dass die durchgeführte Evaluation nicht ausgereicht hat, um festzustellen, ob die Bearbeitung eines Testauftrags mit der Unterstützung durch mein Plugin effektiver und schneller ist, als ohne Plugin. Außerdem liegt die Größe der Stichprobe mit zehn Probanden in einem Bereich, der sich nicht für statistisch relevante Aussagen nutzen lässt. Die Evaluation mit wesentlich mehr Probanden wäre zum einen zeitlich nicht möglich gewesen. Zum anderen gestaltete sich die Beschaffung von geeigneten Probanden als schwierig, da die Dauer von mehr als einer Stunde viele geeignete Kandidaten abgeschreckt hat.

Weitere Beobachtungen und Sachverhalte

Durch Beobachtung der Probanden bei Lösung der Aufgabe konnten weitere Beobachtungen gemacht werden, die im Folgenden aufgelistet sind.

- Zwei Probanden (einer aus der Plugin-Gruppe, einer aus der Kontrollgruppe) haben nicht das Kennwort des angemeldeten Nutzers geändert, sondern haben im Admin-Menü in der Benutzerübersicht das Kennwort eines anderen Nutzers geändert.
- Zwei Probanden aus der Plugin-Gruppe haben den Screencast vollkommen ignoriert. Ein weiterer Proband hat nicht herausgefunden, wie der Screencast in akzeptabler Größe betrachtet werden kann und das dort eine Tonaufnahme vorhanden ist.
- Ein Proband aus der Plugin-Gruppe hat bei den Tests keine Asserts verwendet und somit nur geprüft, ob der Ablauf des Tests keine Fehler verursacht, aber

[3] 7-10 Minuten Tutorialvideo, 13-35 Minuten Bearbeitungszeit, 10-15 Minuten Interview

nicht den eigentlichen Testauftrag erfüllt.

- Ein Proband aus der Plugin-Gruppe hat die eigentlich zu entwickelnden Tests nicht gesehen und nur das Hauptszenario getestet.
- Ein Proband aus der Plugin-Gruppe hat das Hauptszenario zunächst übersehen. Nachdem erkannt wurde, dass ein weiterer Test zu schreiben ist, hat der Proband dies aus Zeitgründen abgelehnt.
- Keiner der Probanden der Plugin-Gruppe hat den Documentation Explorer genutzt oder der UML-Ansicht Aufmerksamkeit geschenkt.

Weitere mögliche Fehlerquellen

Neben den bereits genannten Sachverhalten, die das Ergebnis möglicherweise beeinflusst haben, gab es noch einige weitere Punkte, die bei der Interpretation des Ergebnisses Berücksichtigung finden sollten.

- **Kritikfähigkeit**: Den Probanden war bei der Bearbeitung der Aufgabe klar, dass das verwendete Plugin von mir stammt. Einige Probanden haben sich deshalb dafür entschuldigt, wenn sie im Abschlussinterview das Plugin kritisiert haben. Trotz Versicherung, dass Kritik absolut erwünscht ist, hat dies möglicherweise dazu geführt, dass die Probanden nicht vollkommen offen ihre Meinung zum Ausdruck gebracht haben.
- **Laborsituation**: Die Probanden haben während der Evaluation nicht in ihrer gewohnten Umgebung gearbeitet, sondern waren an einem ungewohnten Ort. Hinzu kam, dass alle Probanden während ihrer Arbeit von mir beobachtet wurden und ich während ihrer Tätigkeiten Notizen darüber angefertigt habe. Außerdem wurde die Bearbeitungszeit erfasst. Ob, und wenn ja, wie stark sich diese Einflüsse ausgewirkt haben, kann nicht beurteilt werden. Zumindest ein Proband merkte aber an, dass er während meiner Evaluation ruhiger gewesen wäre, als im Evaluationsteil von Benjamin Riedel und er sich nicht mehr ganz so stark in einer Prüfungssituation wähnte.
- **Unterschiede in der Aufgabenstellung**: Durch das hier präsentierte Plugin und dem Plugin aus [Riedel2014] wird eine feste Struktur vorgegeben, wie ein Testauftrag aufgebaut ist und präsentiert wird. Ohne die Verwendung dieser Plugins sähe die Beschreibung einer Testaufgabe mit Sicherheit anders aus. Ich habe versucht, beiden Gruppen möglichst die gleichen Informationen zur Verfügung zu stellen. Bei der Kontrollgruppe lagen alle Informationen lediglich in Form eines Textblocks vor, während die Plugin-Gruppe einen strukturierten Testauftrag erhielt (vgl dazu A.2 und A.3). Inwieweit diese Form realistisch ist und welche Informationen jemand tatsächlich mitliefern würde, wenn er einen Testauftrag ohne Softwarehilfe erstellen würde, bleibt jedoch offen.

7.3. Erkenntnisse

Obwohl durch die Evaluation nicht geklärt werden konnte, wie effektiv die Nutzung des Plugins letztendlich ist, so konnten durch das Interviewen der Probanden (siehe Abschnitt A.4) und die Beobachtung dieser bei der Aufgabenbearbeitung, trotzdem zahlreiche Erkenntnisse gewonnen werden. Die nachfolgenden Erkenntnisse stammen ausschließlich aus der Plugin-Gruppe.

Von den Probanden positiv angemerkt:

- die Möglichkeit, Code zu generieren (5 Probanden)
- die Einteilung in drei verschiedene Perspektiven (4 Probanden)
- die Übersichtlichkeit insgesamt (3 Probanden)
- die Möglichkeit, einen Screencast direkt in Eclipse anzuschauen (2 Probanden)
- die Möglichkeit, GUI-Interaktionen im Browser aufzuzeichnen (2 Probanden)
- die Möglichkeit, Aufgaben als erledigt zu markieren (2 Probanden)
- die Möglichkeit, Selenium-Befehle aus vorhandenen Tests per Drag & Drop in die aktuelle Aufnahme zu ziehen (1 Proband)
- die if-then-Struktur der einzelnen Tests in der Aufgabenstellung (1 Proband)
- die Code-Beispiele/die Anzeige der vorhandenen Tests in eigener Ansicht (1 Proband)

Von den Probanden kritisiert, bzw. als Verbesserungsvorschlag geäußert:

- Bug im Selenium IDE-Plugin, der dazu führt, dass im Codegenerat in der URL am Ende ein Slash (/) zu viel generiert wird (führt z.B. zum Aufruf „http://localhost:9000//“, was einen Fehler verursacht) (3 Probanden)[4]
- Die Größenänderung beim Screencast ist unintuitiv.[5] (3 Probanden)
- Der Unterschied zwischen der Funktionalität der Schaltflächen „Generate Java-Commands“ und „Generate Java-Class“ ist unklar.[6] (2 Probanden)
- Texte der Aufgabenstellung (Actions-to-Test-View) können nicht ausgewählt und kopiert werden (z.B. darin enthaltene Logindaten). (2 Probanden)
- Die Möglichkeit, Aktionen im Recording-Server (z.B. Löschen von Befehlen) rückgängig zu machen, fehlt. (1 Proband)

[4] Davon waren allerdings auch alle Probanden der Kontrollgruppe betroffen.

[5] Der Screencast wird in einer Eclipse-View mithilfe des eclipseinternen Browsers angezeigt, der diese View ausfüllt. Der eigentliche Video-Player wird dabei durch Java-Script erzeugt und enthält eine Vergrößern-Funktion. Ist jedoch die Eclipse-View klein, wird das Video durch das Vergrößern des Players nicht größer. Dazu muss die Eclipse-View an sich vergrößert werden.

[6] Durch einen Klick auf „Generate Java-Class“ wird eine vollständige Java-Klasse, inklusive Test-Setup und übersetzter Selenium-Befehle, generiert, während durch einen Klick auf „Generate Java-Commands“ nur die Java-Kommandos generiert werden, die aus den aktuellen Selenium-Befehlen resultieren.

- Die Möglichkeit, den Webserver zu beenden, fehlt. (1 Proband)
- Die Namen bei der Anzeige von vorhandenen Tests sind nichtssagend. (1 Proband)
- Das Code-Generat (aus den Selenium-Befehlen) ist von sich aus nicht korrekt formatiert. (1 Proband)
- Die Aufgabenstellung sollte stärker hervorgehoben werden. (1 Proband)
- Es wird beim Generieren von Code nur Code hinzugefügt. Es wäre besser, wenn bereits generierter Code nach einer Änderung ersetzt wird. (1 Proband)
- Es ist nicht ersichtlich, welche Perspektive gerade aktiv ist (Workflow-View (vgl. Abschnitt 5.5)). (1 Proband)
- Die Möglichkeit Code zu generieren, wäre in der Code-Perspektive besser platziert (statt in der Recording-Perspektive). (1 Proband)
- Die Möglichkeit, einzelne Selenium-Befehle in der Recording-Perspektive auszuwählen und nur den Code zu diesen Befehlen generieren zu können, fehlt. (1 Proband)
- Klickt man in der Workflow-View auf die dort vorhandene Schrift, wird die Perspektive nicht gewechselt. (1 Proband)
- Es fehlt eine Schaltfläche, bei der beim Anklicken geprüft wird, ob alle Aufgaben erledigt sind und die Arbeit am Projekt ggf. abschließt. (1 Proband)
- Es gibt keine Einführung. „Man fühlt sich etwas allein gelassen.“ (1 Proband)
- Eine Rückwärtssuche im Generat wäre wünschenswert. Das bedeutet, wenn man ein Java-Kommando im Quelltext auswählt, sollte in der Übersicht der Selenium-Befehle angezeigt werden, durch welchen Selenium-Befehl der ausgewählte Java-Befehl zustande gekommen ist.

Insgesamt lässt sich sagen, dass das Plugin durchaus positiv aufgenommen wurde. Die Möglichkeit Code direkt in der IDE zu generieren und die Aufteilung in die drei verschiedenen Perspektiven wurden fast durchgängig positiv angemerkt. Bei der Bedienung des Plugins traten Schwierigkeiten meist bei Detailfragen auf. Dies kann teilweise auf mangelnde Erfahrung mit dem Plugin zurückgeführt werden. Im Tutorialvideo wurde die Bedienung nur relativ oberflächlich erläutert, sodass es nicht verwunderlich ist, dass sich die Probanden bei ihrer ersten Benutzung meist zunächst einen Überblick verschaffen mussten. Ein weiterer Grund für die Schwierigkeiten, ist in der mangelnden Reife des Plugins bzw. dessen prototypischer Umsetzung zu suchen, was sich auch in den oben aufgelisteten Kritikpunkten widerspiegelt. Trotzdem bewerteten die Probanden die Bedienung tendenziell als eher intuitiv und die Unterstützung als hilfreich (siehe Tabelle A.4).

8. Fazit

In dieser Arbeit konnte gezeigt werden, dass die technischen Hürden, die das Crowdsourcen von automatisierten Tests erschweren, teilweise gelöst werden können. Die Testerstellung für Projekte, die ohne komplexe Infrastruktur auskommen, kann aus technischer Sicht ohne Weiteres in die Crowd verlagert werden. Einem Worker kann die Einrichtung solcher Projekte fast vollständig abgenommen werden, sodass dieser sich auf die Lösung seiner eigentlichen Aufgabe - dem Erstellen von Tests - konzentrieren kann.

Auch die hier entwickelte Struktur eines Testauftrags scheint grundsätzlich zu funktionieren. Wie in der Evaluation gezeigt, sind auch ungeübte Nutzer ohne Probleme in der Lage, die Aufgabenstellung eines Testauftrags korrekt zu interpretieren und entsprechend zu handeln. Gleichzeitig konnte in [Riedel2014] gezeigt werden, dass Benutzer mit sehr wenig Übung und entsprechender Softwareunterstützung Testaufträge in relativ kurzer Zeit erstellen können. Durch die vordefinierte Struktur wird es den Autoren der Testaufträge außerdem erleichtert, die relevanten Informationen in den Auftrag einfließen zu lassen, während auf Seiten der Worker die Arbeit durch eine einheitliche Präsentation der Informationen vereinfacht wird. Inwiefern die gewählte Auftragsstruktur neben GUI-Tests auch für andere Testarten funktioniert, müsste in zukünftigen Arbeiten untersucht werden.

GUI-Tests scheinen sich besonders gut für die Auslagerung in die Crowd zu eignen. Ein Grund dafür ist die Differenz zwischen der Zeit, die für die Erstellung eines entsprechenden Testauftrags, und der Zeit die zur Lösung eines solchen Auftrags benötigt wird. Diese fällt - wie die Evaluation gezeigt hat - bei GUI-Tests relativ groß aus. So benötigt selbst die Bearbeitung eines Testauftrags mit äußert minimalem Umfang deutlich mehr Zeit, als für die Erstellung eines solchen Testauftrags notwendig ist. Damit kann die Auslagerung eines solchen Tests aus Providersicht durchaus Sinn machen.

Ein weiterer Vorteil, den GUI-Tests in diesem Kontext bieten, sind die geringen Kenntnisse der AUT, die ein Worker erwerben muss. In der Evaluation hat nicht ein einziger Proband den Quelltext der AUT näher angesehen. Trotzdem waren fast alle Probanden in der Lage, korrekte GUI-Tests zu erstellen. Dies lag mit Sicherheit unter anderem an der Möglichkeit große Teile des Tests aufzunehmen, statt diese durch das manuelle Schreiben von Code zu erzeugen. Doch auch wenn eine Auseinandersetzung mit Teilen des Quellcodes der AUT notwendig ist, kann dies durch das Mitliefern von bereits bestehenden Tests erleichtert werden. Diese können einem Worker als Einstiegspunkt in die Applikation dienen und so seine Arbeit vereinfachen. Auch der in meinem Plugin bereits integrierte Screencast würde sich besonders gut eignen, um darin GUI-spezifische Sachverhalte kurz zu erläutern. In wie weit die Erstellung eines solchen Screencasts auf Seiten des Providers, insbesondere unter Berücksichtigung

des Zeitaspekts, realisiert werden kann, ist aber noch offen.

Eine Frage die ebenfalls nicht vollständig beantwortet werden konnte, ist die Frage in wieweit das Konzept bezüglich der Dreiteilung des Testprozesses wirklich funktioniert. Die Aufteilung der IDE-Hilfen auf drei Perspektiven wurde in der Evaluation zwar positiv gesehen, jedoch wurden in dieser hauptsächlich die zweite und dritte Perspektive genutzt. Die dort implementierten Konzepte scheinen dabei zu funktionieren, auch wenn im Detail sicher Verbesserungsbedarf besteht. Wie hilfreich die erste Perspektive ist, konnte jedoch aufgrund der einfachen Aufgabenstellung, die eine Nutzung dieser Perspektive nicht nötig gemacht hat, nicht geklärt werden. Konzeptionell halte ich diese Perspektive jedoch nach wie vor für sinnvoll, da sich ein Worker in einem realen Umfeld sicherlich häufiger zumindest in Teile des Quellcodes einarbeiten muss. Fraglich ist jedoch in wieweit eine Dreiteilung bei anderen Testarten (z.B. Unit oder Integrationstests) sinnvoll ist. Hier müsste untersucht werden, ob die Recording-Perspektive entfallen sollte oder ob in ihr andere testartspezifische Hilfen untergebracht werden können bzw. sollten.

Um das im Rahmen dieser Arbeit erstellte Plugin in einem realen Kontext nutzen zu können, müsste dieses vorher erweitert werden. Das Plugin dürfte nicht nur das Play!-Framework unterstützen, da bei diesem die Nutzerzahl sicherlich nur sehr überschaubar ist. Hier müsste entweder eine breitere Unterstützung für eine Vielzahl von Frameworks geboten werden oder besser noch, das Plugin so generisch gestaltet werden, dass prinzipiell jedes beliebige Framework genutzt werden kann. Bei der Aufnahme von GUI-Tests sollten nicht nur Selenium-Tests bzw. Webapplikationen unterstützt werden. Hier kann Visual Studio als Referenz gelten, welches prinzipiell GUI-Tests für jede beliebige Applikation anbietet.

Ein weiterer wichtiger Punkt für eine etwaige reale Nutzung, ist die Verbindung zu einer Crowdsourcing Plattform. So müsste in das Plugin auf jeden Fall eine Möglichkeit eingefügt werden, das Ergebnis eines fertig bearbeiteten Testauftrags auf eine entsprechende Plattform hochzuladen. Wie eine solche Plattform aber überhaupt aussehen könnte, ist eine für das Funktionieren des Gesamtprozesses enorm wichtige Frage. Dabei müssten zunächst einmal zahlreiche Rahmenbedingungen geklärt werden. Zum Beispiel müsste festgelegt werden, ob ein Testauftrag nur einziges Mal vergeben wird, oder es grundsätzliche mehrere parallele Bearbeiter geben kann. Auch die Frage nach der Bezahlung dürfte einen enormen Einfluss auf Erfolg oder Misserfolg einer solchen Plattform haben. Werden bei paralleler Bearbeitung eines Auftrags alle Bearbeiter bezahlt, oder muss der Test-Task-Provider sich für eine Lösung entscheiden?

Abschließend lässt sich sagen, dass die in [Riedel2014] und dieser Arbeit vorgestellten Plugins die grundsätzliche technische Machbarkeit demonstrieren, automatisierte Testfälle zu crowdsourcen. Jedoch können beide Masterarbeiten lediglich als Fundament für zukünftige Arbeiten dienen, in denen der Gesamtprozess näher definiert und in Form einer konkreten Crowdsourcing Plattform technisch umgesetzt werden müsste.

A. Anhang

A.1. Demographie-Fragebogen

Teilnehmer Nr. ◯

Fragebogen

1. Alter: _____
2. Studiengang:

   _____

3. Semester: _____

4. Ich habe Erfahrung beim Programmieren:

   keine ◯ ◯ ◯ ◯ ◯ viel
   1 2 3 4 5

5. Ich habe Erfahrung im Testen:

   keine ◯ ◯ ◯ ◯ ◯ viel
   1 2 3 4 5

6. Ich habe Erfahrung mit Eclipse:

   keine ◯ ◯ ◯ ◯ ◯ viel
   1 2 3 4 5

7. Ich habe Erfahrung mit Unit-Tests:

   keine ◯ ◯ ◯ ◯ ◯ viel
   1 2 3 4 5

8. Ich habe Erfahrung mit Selenium-Tests:

   keine ◯ ◯ ◯ ◯ ◯ viel
   1 2 3 4 5

A.2. Aufgabenstellung der Plugin-Gruppe

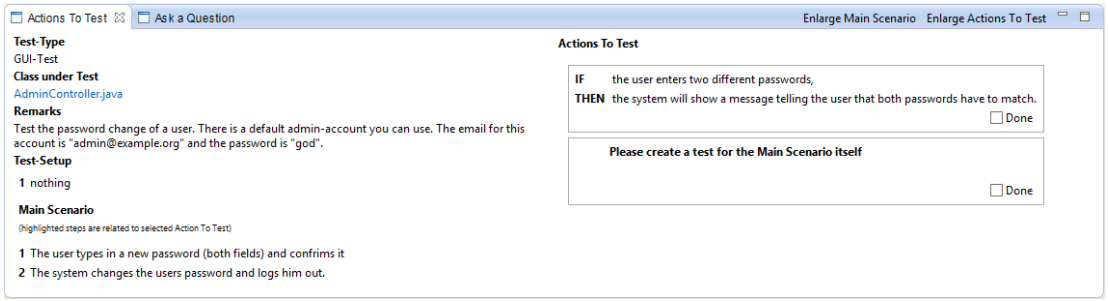

Abbildung A.1.: Aufgabenstellung der Plugin-Gruppe. Der Screenshot stammt direkt aus Eclipse. Dies war die einzige Art, wie den Probanden die Aufgabe präsentiert wurde.

A.3. Aufgabenstellung und Hinweise der Kontrollgruppe

Your Task

Create GUI-tests for the password-change feature. There is a default admin-account you can use. The email for this account is "admin@example.org" and the password is "god". Please test if the system changes the user-password if the user enters a new one. Please test also if there is an error-message when the password and the password repetition are not equal.

Hints

- The project-files can be found under: *C:\DATA\project*
- There are some shortcuts under C:\DATA\
  - *Start Website.bat*: will start the webserver; you must start the server before you can open the page in the browser
  - *Open Website*: will open the website in your browser (http://localhost:9000/)
- There are some gui-tests in the test-folder of the project. You can open the html-files with the Selenium IDE Browser-Plugin
- There are some Java-Classes containing gui-tests in the test-folder of the project

A.4. Abschlussinterviews für Probanden der Plugin-Gruppe

1. Ich wusste was meine Aufgabe war.
   Likert-Skala 1-5, 1 trifft gar nicht zu, 5 trifft voll zu
2. Wie schwierig war die Lösung der Aufgabe auf technischer Ebene?
   Likert-Skala 1-5, 1 sehr leicht, 5 sehr schwer
3. Die Bedienung von Eclipse war intuitiv.
   Likert-Skala 1-5, 1 trifft gar nicht zu, 5 trifft voll zu
4. Ich empfand die Unterstützung durch Eclipse als hilfreich.
   Likert-Skala 1-5, 1 trifft gar nicht zu, 5 trifft voll zu
5. Was hat Ihnen an Eclipse/Plugin gefallen?
6. Was hat Ihnen an Eclipse/Plugin nicht gefallen?
7. Welche Funktionen fanden Sie besonders hilfreich?
8. Empfanden Sie die drei verschiedenen Perspektiven als hilfreich?
9. Fehlten Ihnen an einer Stelle Hilfen?

Frage/Antwort	**1**	**2**	**3**	**4**	**5**
1.	0	0	2	0	3
2.	1	2	1	0	1
3.	0	0	2	3	0
4.	0	0	2	2	1

Tabelle A.1.: Antworten auf die Likert-Skala-Fragen des Abschlussinterviews. Zeilenweise sind die Fragen aufgetragen, während in den Spalten das jeweilige Vorkommen der Antwort notiert ist.

Literaturverzeichnis

[Dijkstra1972] E. W. Dijkstra
The Humble Programmer
ACM Turin Lecture 1972, S. 10

[Howe2006] J. Howe
Crowdsourcing: A Definition
von http://www.crowdsourcing.com/cs/2006/06/crowdsourcing_a.html, 11.02.2014

[Wolter2013] J. Wolter
Crowdtesting: Anwendungsfälle und Grenzen einer neuen Methode
OBJEKTspectrum, Ausgabe Testing/2013, 2013

[Deak2013] A. Deak, T. Stålhane
Organization of testing activities in Norwegian Software Companies
IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, 2013

[Riedel2014] B. Riedel
Konzept und Implementierung eines heuristischen Editors zum Erstellen von Testaufgaben auf einer Crowdsourcing Plattform
Gottfried Wilhelm Leibniz Unisverität Hannover, 2014

[Dolstra2013] E. Dolstra, R. Vliegendhart, J. Pouwelse
Crowdsourcing GUI Tests
IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, 2013

[Pham2013] R. Pham, L. Singer, K. Schneider
Building Test Suites in Social Coding Sites by Leveraging Drive-By Commits
35th International Conference on Software Engineering (ICSE), 2013

[Rooksby2009] J Rooksby, M. Rouncefield, I. Sommerville
Testing in the Wild: The Social and Organisational Dimensions of Real World Practice
Computer Supported Cooperative Work archive, Volume 18, Issue 5-6, 2009

[Dustin2002] E. Dustin
Effective Software Testing: 50 Specific Ways to Improve Your Testing
Addison-Wesley Professional, 2002, S. 192-194

[Garousi] V. Garousi, T. Varma

A Replicated Survey of Software Testing Practices in the Canadian Province of Alberta: What has Changed from 2004 to 2009?
Journal of Systems and Software 83 (11), 2010

[Kasurinen2009] J. Kasurinen, O. Taipale, K. Smolander
Analysis of Problems in Testing Practices 16th Asia-Pacific Software Engineering Conference, 2009

[Haberl2012] P. Haberl, A. Spillner, K. Vosseberg, M. Winter
Umfrage 2011: Softwaretest in der Praxis
2012

[Ng2004] S.P. Ng, T. Murnane, K. Reed, D. Grant, T.Y. Chen
A Preliminary Survey on Software Testing Practices in Australia
Proceedings of the 2004 Australian Software Engineering Conference, 2004

[Griffiths2001] D. Griffiths
The Theory and Practice of Outsourcing
2001

[Tervonen2009] I. Tervonen, T. Mustonen
Test Automation: Observations and Lessons Learned
Fourth IEEE International Conference on Global Software Engineering, 2009

[Shah2011] H. Shah, S. Sinha, M.J. Harrold
Outsourced, Offshored Software-Testing Practice: Vendor-Side Experiences
Sixth IEEE International Conference on Global Software Engineering, 2011

[tiobe2013] *TIOBE Index for December 2013*
von http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 16.12.2013

[redmonk2013] *The RedMonk Programming Language Rankings: June 2013*
von http://redmonk.com/sogrady/2013/07/25/language-rankings-6-13/, 16.12.2013

[Cockburn22008] A. Cockburn, J. Highsmith
Agile Software Development, The Agile Software Development Series Alistair Cockburn, 2000, S.84f

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 30.04.2014

_____

Florian Eiteljörge