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

Konzept und Implementierung eines heuristischen Editors zum Erstellen von Testaufgaben auf einer Crowdsourcing Plattform

Masterarbeit

im Studiengang Informatik

von

Benjamin Riedel

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

Hannover, 30. April 2014

Zusammenfassung

Das Testen in Softwareprojekten wird zunehmend an Spezialisten ausgelagert. Dabei gibt es bisher allerdings nicht die Möglichkeit, nur die Erstellung einzelner automatisierter Testfälle unkompliziert und schnell auszulagern. Ich bin Teil eines Teams, das dies durch Nutzung von Crowdsourcing ermöglichen möchte: Ein Entwickler schreibt aus seiner IDE heraus einen Auftrag für die Erstellung eines automatisierten Tests auf einer Crowdsourcingplattform aus, die ihn verwaltet, bis ein Tester diesen Auftrag übernimmt und bearbeitet. Ein kritischer Punkt dabei ist, dass die Erstellung dieses Testauftrags nicht aufwändiger sein darf, als die Erstellung des Tests selbst, aber dennoch genügend Informationen übergeben werden, um einen projektfremden Testfallentwickler zu befähigen, den Auftrag zufriedenstellend zu bearbeiten.

In dieser Arbeit wird ein Eclipse-Plugin beschrieben, das dem Entwickler bei der Erstellung hilft und die Informationsgewinnung weitestmöglich automatisiert vornimmt. Dies geschieht bei Informationen, die nicht sicher generiert werden können, wie beispielsweise bei der Frage, welche Teile des Programmcodes für einen Test interessant sind, mit Hilfe von Heuristiken. Diese geben Prognosen darüber ab, welcher Inhalt in den einzelnen Informationsfeldern eines Testauftrags sinnvoll sein könnte, sodass der Entwickler diese gegebenenfalls nur noch anpassen muss.

Abstract

Testing in software projects is getting outsourced more and more. However, so far there is no easy and fast way to only outsource the creation of individual automated test cases. I am part of a team that wants to enable this by using crowdsourcing: A developer advertises a task for the creation of an automated test from his integrated development environment on a crowdsourcing plattform, which administrates it, until a tester accepts and executes it. A critical aspect in this process is, that the creation of this task must not be more elaborate than the creation of the test itself. Though it still has to provide sufficient information to enable a developer not associated with the project to complete the task satisfactorily.

In this thesis an Eclipse plugin is described that helps the developer with the creation, and that executes the information gathering automated to the greatest possible extent. This occurs with information that can not be generated reliably, as for example the question which parts of the source code are relevant for a test, by means of heuristics. These give a prognosis on which content could be relevant to the individual fields of the test task, so the developer only has to adjust them if necessary.

Inhaltsverzeichnis

1. Einleitung **1**
1.1. Motivation . . . 1
1.1.1. Crowdsourcing . . . 1
1.1.2. Crowdsourcing im Kontext dieser Arbeit und Abgrenzung gegenüber Crowd-Testing . . . 2
1.1.3. Tester . . . 4
1.1.4. Entwickler . . . 4
1.2. Zielsetzung . . . 5
1.3. Aufbau der Arbeit . . . 5

2. Vision & Lösungsansatz **6**
2.1. Otto und der Webshop . . . 6
2.2. Grundidee . . . 6
2.3. Testaufträge . . . 7
2.3.1. Vergleich Requirements Engineering . . . 8
2.3.2. Felder eines Testauftrags . . . 9
2.4. Szenario dieser Arbeit . . . 11
2.5. Ablauf einer typischen Anwendung von TaskElf . . . 12

3. Grundlegende Konzepte des Testauftragseditors **18**
3.1. Einbettung in die IDE . . . 18
3.2. Verbergen von vertraulichem Code . . . 18
3.3. Aufwandseinschätzung . . . 19
3.4. Erweiterbare Architektur . . . 20

4. Benutzeroberfläche **25**
4.1. Ist-Zustand . . . 25
4.1.1. Formular . . . 26
4.1.2. Steuerbuttons . . . 32
4.2. Erweiterungen . . . 33

5. Heuristische Intentionserkennung **36**
5.1. Verwandte Arbeiten . . . 36
5.2. Grundzüge meines Ansatzes . . . 37
5.3. Kontextunabhängige Annahmen . . . 39
5.4. Nutzung des IDE-Kontextes . . . 40
5.4.1. Statische Quellcodebeobachtung . . . 44
5.4.2. Beobachtung von Quellcodeänderungen . . . 44
5.5. Beobachtung des Nutzerverhaltens . . . 45

5.6. Vereinigung verschiedener Heuristiken 48
5.7. Zielfelder der Heuristiken 49

6. Implementierung von Relevance Neighbors 52
6.1. Algorithmus zur Bestimmung der relevanten Nachbarn 54
6.2. Unterteilung in Klassen und Methoden 56
6.3. Probleme der Umsetzung 57

7. Evaluation 59
7.1. Ziel 59
7.2. Versuchsbedingungen 59
7.3. Versuchsaufbau 60
7.3.1. Aufnahme demographischer Daten 61
7.3.2. Programmieraufgabe 61
7.3.3. Erstellung von Testfällen 62
7.4. Ergebnisse 63
7.4.1. Quantitativer Versuchsteil 63
7.4.2. Qualitativer Versuchsteil 66
7.5. Fehlerquellen 67

8. Fazit 69
8.1. Ergebnisse 69
8.2. Nächste Schritte 70

A. Beispiel RelevanceNeighbors 71

B. Evaluationsdokumente 73

1. Einleitung

Im Folgenden beschreibe ich Motivation, Zielsetzung sowie den Aufbau der Arbeit.

1.1. Motivation

Testen ist ein essentieller Teil der Softwareentwicklung und laut [Haberl2011] fließen im deutsch-sprachigen Raum etwa 20% der Entwicklungskosten in diesen Bereich. Dabei, so die Studie, ist der Testaufwand allerdings nicht gleichmäßig auf ein Projekt verteilt, sondern steigt im Verlauf des Projektes meist deutlich an. Da eine eigene Qualitätssicherungsabteilung bei dieser Art des Vorgehens nicht durchgehend gleichmäßig beschäftigt wäre, unterhalten ein knappes Drittel der befragten Unternehmen keine solchen Abteilungen. Stattdessen wird die Arbeit von „normalen“ Entwicklern mitübernommen, denen oft allerdings eine tiefergehende Expertise im Testen fehlt. Als Reaktion auf diesen Zustand, gibt es die Entwicklung, den Testvorgang vollständig aus dem Unternehmen auszulagern. So kann dieser besser skaliert werden, da nicht die gesamte Zeit entsprechende Mitarbeiter vorgehalten werden müssen. Außerdem können die Erfahrung und das Wissen von Experten genutzt werden. Bisher wird vor allem an spezialisierte Firmen (wie z.B. uTest[1] oder Sogeti[2]) ausgelagert. Wir[3] hingegen befassen uns mit einer alternativen Gruppe von möglichen Testern: Der Crowd.

1.1.1. Crowdsourcing

„Simply defined, crowdsourcing represents the act of a company or institution taking a function once performed by employees and outsourcing it to an undefined (and generally large) network of people in the form of an open call. This can take the form of peer-production (when the job is performed collaboratively), but is also often undertaken by sole individuals. The crucial prerequisite is the use of the open call format and the large network of potential laborers.“[Howe2006]

„Als einfache Definition lässt sich sagen, dass Crowdsourcing die Handlung eines Unternehmens oder einer Einrichtung bezeichnet, eine Aufgabe, die einst von Angestellten ausgeführt wurde, an ein nicht definiertes (und im Allgemeinen großes) Netzwerk von Menschen, in der Form eines offenen Aufrufs, auszulagern. Dies

[1] `http://www.utest.com/`

[2] `http://www.sogeti.de`

[3] Diese Arbeit entsteht zusammen mit [Eiteljörge2014] im Rahmen von Raphael Phams Doktorarbeit. Mit „wir“ meine ich daher in dieser Arbeit diese Gruppe, die wir gemeinsam an der Gesamtvision arbeiten.

> kann die Form von peer-production annehmen (wenn die Arbeit gemeinschaftlich durchgeführt wird), die Arbeit wird aber auch oft von einzelnen Individuen durchgeführt. Die entscheidenden Bedingungen sind der Gebrauch des Formats, des offenen Aufrufs und des großen Netzwerks von potentiellen Arbeitern.“

Crowdsourcing beschreibt also die Praktik einer Firma oder Institution, eine Funktion, die traditionell von Angestellten der Firma ausgeführt wurde, an eine nicht spezifizierte große Gruppe von Menschen in Form eines offenen Aufrufs auszulagern. Es hat sich bereits gezeigt, dass Crowdsourcingansätze für viele Probleme geeignet sind, zufriedenstellende Lösungen zu erbringen. So hat zum Beispiel die Universität von Washington das Computerspiel „Foldit“ entwickelt, dessen Ziel es ist, Proteine räumlich so zu gestalten, dass diese möglichst stabil sind. „Foldit“-Spielern ist es auf diesem Wege bereits gelungen, die Struktur eines für die AIDS-Forschung wichtigen Proteins aufzudecken. Ein Problem, das Forscher bislang erfolglos zu lösen versuchten.[Khatib2011] Während in diesem Beispiel die gegenseitige Beeinflussung der Spieler wichtig für den Erfolg ist, funktioniert Crowdsourcing allerdings auch, wenn die Teilnehmer in der Crowd unabhängig voneinander sind.

Wichtig für Aufgaben, die für Crowdsourcing geeignet sind, ist, dass sie menschliche Kreativität erfordern und es so einen Vorteil gibt, sie an Menschen, statt an Maschinen auszulagern. Im obigen Beispiel sind Computer nicht in der Lage, das Problem mit bekannten Algorithmen in akzeptabler Zeit zu lösen. Die Crowd hat mit ihrer Eigendynamik jedoch Wege gefunden, sich der Lösung anzunähern. In die Crowd ausgelagerte Aufgaben werden auch als „Human Intelligence Tasks“ oder kurz „HIT“ bezeichnet. Der in der Crowd angesiedelte Bearbeiter eines HITs wird „Worker“ genannt.

1.1.2. Crowdsourcing im Kontext dieser Arbeit und Abgrenzung gegenüber Crowd-Testing

Crowdsourcing wird in den Bereichen der mobilen Anwendungen und der Oberflächentests bereits zum Testen genutzt. Während in [Dolstra2012] betrachtet wird, wie gut Crowdsourcing bei Oberflächentests funktioniert, erläutert [Wolter2013], wie Crowdsourcing für das Testen von mobilen Apps genutzt wird. Dabei werden die Worker entweder genutzt, um bereits vorgegebene Testfälle auszuführen, oder aber um explorativ zu testen. Dieses Vorgehen wird Crowd-Testing genannt.

Ein HIT beim Crowd-Testing ist also die Durchführung eines oder mehrerer Tests oder die explorative Suche nach Fehlern ohne konkrete, vorgegebene Testfälle. Im Gegensatz dazu möchten wir die Crowd nutzen, um Testfälle und zu diesen passende automatisierte Tests zu erstellen (beispielsweise als Unit-Test in JUnit). Die Erstellung eines solchen Tests stellt den HIT im Kontext dieser Arbeit dar. Wenn ich im weiteren Verlauf dieser Arbeit von „Test“ spreche, dann meine ich einen solchen automatisierten Test.

Die Nutzung der Crowd erlaubt dem Nutzer eine sehr feine Granularität bei der Wahl,

was er auslagern möchte. Es kann gezielt die Expertise von unternehmensfremden Personen an den Stellen genutzt werden, wo diese benötigt wird, ohne ihnen größere Aufträge oder sogar das gesamte Testen übertragen zu müssen. Stattdessen kann bereits auf Entwicklerebene ausgelagert werden: Wenn ein Entwickler bestimmte Tests nicht gerne macht - sei es aus Zeitgründen, oder weil ihre Erstellung sehr repetitiv ist - oder keine Erfahrung in diesem Bereich vorhanden ist, kann er ein gewisses Budget nutzen, um diese Tests in die Crowd zu verlagern. In unserem Ansatz muss er nur einen Auftrag für einen Test schreiben und diesen auf der Crowdsourcing Plattform (z.B. TestHub [Stoliar2013]) veröffentlichen. Ein Testauftrag soll dabei die Beschreibung eines Tests sowie alle zur Bearbeitung notwendigen Dateien und Informationen umfassen. Der Umfang eines Testauftrages darf die Erstellung weniger Tests nicht übersteigen. Der Worker in der Crowd soll durch eine pauschale Vergütung motiviert werden.

Im weiteren Verlauf dieser Arbeit werde ich für den Worker den Begriff „Tester“ benutzen.

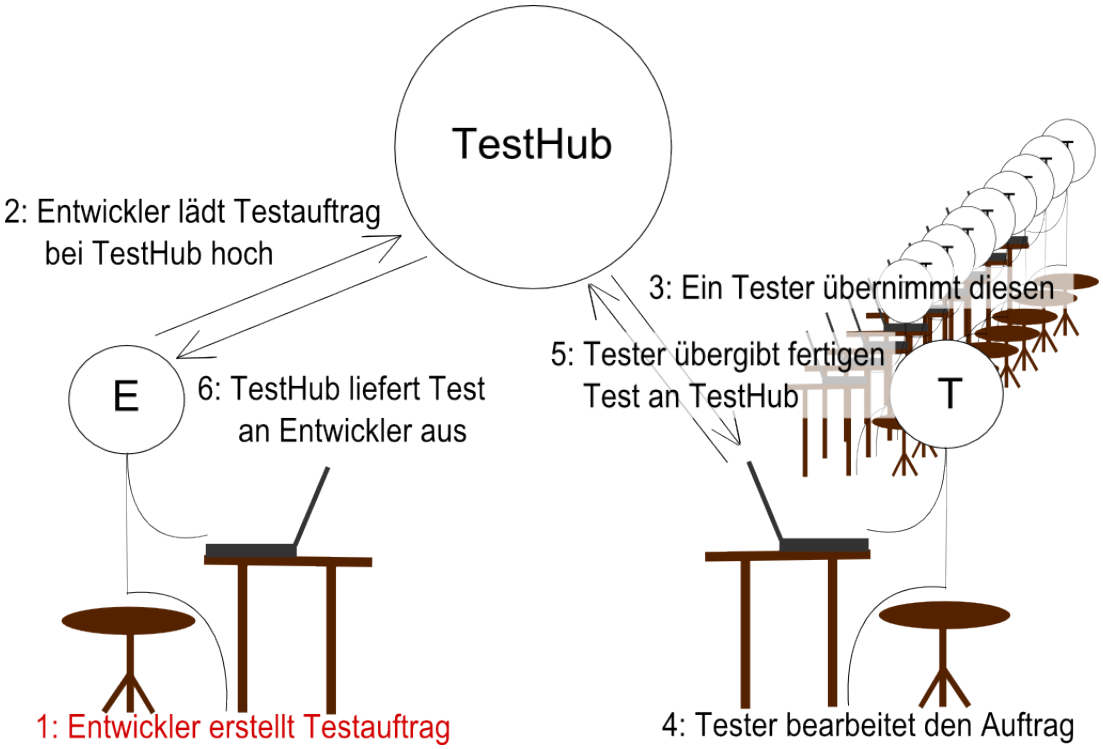

Abbildung 1.1.: Der Gesamtvorgang des Auslagerns eines Tests in die Crowd

Der gesamte Auslagerungsvorgang (Abbildung 1.1) läuft dabei wie folgt ab:

1. Der Entwickler erstellt einen neuen Testauftrag (TA), der den HIT beschreibt
2. Der Entwickler lädt den TA auf die Crowdsourcing Plattform
3. Ein Tester sieht den Testauftrag und lädt ihn sich zur Bearbeitung herunter

4. Der Tester bearbeitet den TA
5. Der Tester lädt seine Lösung des TAs auf den Server
6. Der Server informiert den Entwickler über den bearbeiteten TA und sendet ihm die Lösung zu, wo diese, nach Bestätigung durch den Entwickler, automatisch in den Code eingepflegt wird.

Damit dieser Gesamtvorgang reibungslos und effizient abläuft, sollen Tester und Entwickler softwareseitig unterstützt werden. Die Unterstützung des Testers bei der Erstellung (Punkt 1) ist Inhalt dieser Arbeit.

Im Folgenden möchte ich die Rollen des Testers – der den HIT bearbeitet – und des Entwicklers – der den HIT erstellt – in diesem Crowdsourcingansatz näher betrachten. Dabei lege ich den Schwerpunkt vor allem auf ihre unterschiedlichen Motivationen.

1.1.3. Tester

Der Tester möchte eine Aufgabe so schnell wie möglich abschließen, um mehr Aufgaben pro Stunde zu schaffen und so einen möglichst hohen Verdienst zu erreichen. Das bedeutet, wenn die Dauer der Bearbeitung eines Testauftrags im Verhältnis zur Vergütung zu hoch ist, wird sich kein Tester für diesen Testauftrag finden. Um dem entgegenzuwirken, sollen dem Tester so viele Arbeiten wie möglich durch Automatisierung abgenommen werden. Das gilt insbesondere für das Aufsetzen der Testumgebung. In den meisten Fällen ist es sehr aufwändig, ein in der Entwicklung befindliches Projekt auf einem neuem System zu starten: Es müssen Abhängigkeiten aufgelöst, notwendige Tools und Programme installiert und alles korrekt miteinander verknüpft werden. Dieser Aufwand ist für die Bearbeitung eines einzelnen Testautrags unverhältnismäßig hoch und muss dem Tester daher durch Automatisierung abgenommen werden. Außerdem sollte den Testern alles Wissen, das sie benötigen, direkt zusammen mit dem HIT mitgeteilt werden, da fehlende oder schwammige Anforderungen die Arbeit erschweren.

1.1.4. Entwickler

Der Entwickler soll in diesem Szenario die HITs aus dem Entwicklungsprozess heraus erstellen. Das heißt, dass er in seinem Prozess durch die Aufgabe nicht zu sehr gestört werden sollte. Passiert dies dennoch, benötigt der Entwickler eine kurze Zeit, bis er nach dem Erstellen wieder produktiv arbeiten kann.[LaToza2006] Dies könnte in vielen Fällen dazu führen, dass Entwickler das Erstellen von Testaufträgen vor sich herschieben würden, um ihre produktive Phase nicht beenden zu müssen. Im Endeffekt könnte das Erstellen einzelner Testaufträge vergessen werden, wenn die Hürden, die dem Entwickler entgegenstehen, zu groß sind. Dennoch geschriebene Testaufträge könnten aufgrund der zeitlichen Entfernung zur Erstellung des entsprechenden Codes ungenau beschrieben sein. Die Qualität des Testauftrags würde darunter leiden und damit letztendlich auch die Qualität der erstellten Tests.

Ein anderer Grund das Erstellen von Testaufträgen möglichst einfach zu halten, ist, dass das Crowdsourcing überhaupt nur dann effizient ist, wenn der Aufwand des Auslagerns nicht größer ist, als der Aufwand die Tests selbst zu erstellen. Ziel muss es daher sein, den Aufwand sowohl zeitlich als auch geistig zu minimieren. Dies ist der Fokus dieser Arbeit.

1.2. Zielsetzung

Ziel dieser Arbeit ist es, ein Plugin für die IDE Eclipse zu entwickeln, das den Entwickler bei der Erstellung von Testaufträgen unterstützt. Dieses Plugin heißt TaskElf, nach den mythologischen kleinen Helferwesen, die einem Arbeit abnehmen. Dabei liegt der Fokus darauf, Konzepte zu erarbeiten und zu implementieren, die mit Hilfe von Heuristiken den notwendigen Aufwand für den Entwickler minimieren. Weitere wichtige Punkte sind die Erweiterbarkeit von TaskElf, die es erlauben soll, ohne großen Aufwand weitere dieser Konzepte einzubauen, sowie die Benutzbarkeit.

1.3. Aufbau der Arbeit

Die Arbeit ist wie folgt gegliedert: Zuerst gehe ich näher auf die dieser Arbeit zu Grunde liegende Vision und auf meinen Umsetzungsansatz zu dieser ein (Kapitel 2). Anschließend erläutere ich in Kapitel 3 fundamentale Konzepte von TaskElf, bevor ich in Kapitel 4 seine Benutzeroberfläche sowie mögliche Erweiterungen für die Zukunft vorstelle. Danach erkläre ich in Kapitel 5 die heuristische Intentionserkennung und den Einsatz, welcher Heuristiken ich bei TaskElf sinnvoll halte, bevor ich die komplexeste der von mir umgesetzten Heuristiken im Detail erläutere (Kapitel 6). In Kapitel 7 gehe ich darauf ein, wie und mit welchem Ergebnis TaskElf evaluiert wurde, bevor ich ein abschließendes Fazit ziehe (Kapitel 8).

2. Vision & Lösungsansatz

Auf den folgenden Seiten möchte ich zuerst die Vision beschreiben, der diese Arbeit folgt. Dazu werde ich zunächst ein Beispielszenario einführen, auf das ich im Verlauf der Arbeit immer wieder zurückkommen werde. Danach erläutere ich an diesem Beispielszenario die Grundidee. Anschließend werde ich auf meinen Lösungsansatz eingehen, indem ich zuerst erkläre, was genau ein Testauftrag ist und anschließend den angestrebten Ablauf einer einzelnen Interaktion erläutere. Veranschaulicht wird dies anhand von Mock-Ups am Ende dieses Kapitels.

2.1. Otto und der Webshop

Sei Otto ein prototypischer Entwickler eines Webshops. Seine Firma hat sich entschlossen, einen Teil der Erstellung der Tests mit Hilfe von TaskElf auszulagern. Die Entwickler entscheiden selbst, welche Tests ausgelagert werden und welche sie selbst implementieren. Otto möchte viele der nötigen Tests auslagern. Sein Problem ist nun, dass er jeden Test, den er auslagern möchte, so beschreiben muss, dass der Tester, der ihn versucht zu erstellen, auch in der Lage dazu ist.

2.2. Grundidee

Die Grundidee stellt sich am Beispiel wie folgt dar:

> Otto möchte einen neuen Testauftrag erstellen und öffnet aus dem Entwicklungsprozess heraus die TaskElf-GUI. TaskElf erkennt automatisch relevante Informationen zum Webshop-Projekt und nutzt diese, um verschiedene Testaufträge vorzuschlagen, von denen er für wahrscheinlich hält, dass Otto diese gerade erstellen möchte. Informationen sind zum Beispiel Codeänderungen oder die Dateistruktur des Projektes. In der GUI, die Otto sieht, kann er zwischen verschiedenen vorausgefüllten Vorschlägen wählen, was getestet werden soll. TaskElf erkennt, dass Otto zuletzt an einem HTML-Dokument gearbeitet hat und schlägt vor, dass für dieses ein GUI-Test erstellt werden soll. Otto kann nun entweder einen solchen vorgeschlagenen Testauftrag vervollständigen oder – sollte keiner der Vorschläge seinem momentanen Ziel entsprechen – einen komplett neuen Testauftrag erstellen. Sobald Otto den Testauftrag vervollständigt hat, kann dieser über einen Knopfdruck direkt auf die Crowdsourcing Plattform hochgeladen werden.
>
> Auf der Seite der Crowdsourcing Plattform ist der Testauftrag jetzt für Tester sichtbar. Ein Tester, nennen wir ihn Tom, übernimmt den Testauftrag und bekommt daraufhin einen Editor

angezeigt. In diesem Editor kann der Code eingesehen und ausgeführt werden. Tom löst mit seiner Hilfe den Testauftrag und lädt das Ergebnis auf die Crowdsourcing Plattform.

Die Crowdsourcing Plattform regelt nun die Bezahlung und informiert Otto über den ausgeführten Auftrag. Otto kann das Ergebnis nun aus seiner Entwicklungsumgebung heraus von der Crowdsourcingsplattform anfordern und automatisch einbinden lassen.

Diese Arbeit beschränkt sich auf die Erstellung eines Editors zur Erstellung von Testaufträgen auf Ottos Seite, während die Umsetzung des Editors zum Lösen eines Testauftrags, auf Toms Seite, in [Eiteljörge2014] betrachtet wird. Die Umsetzung der Crowdsourcing Plattform muss in einer gesonderten Arbeit in der Zukunft betrachtet werden.

2.3. Testaufträge

Einzelne auszulagernde Tests werden durch Testaufträge definiert. Ein solcher Testauftrag ist das Einzige, was ein Tester für seine Arbeit zur Verfügung gestellt bekommt und muss daher alles enthalten, was der Tester für seine Arbeit zwingend benötigt (zum Beispiel die Application Under Test (AUT)). Zusätzlich soll dem Tester die Arbeit aber auch noch möglichst leicht gemacht werden, was zu weiteren Anforderungen führt: Dinge, die für den Tester zwar nicht zwingend notwendig, aber doch hilfreich sind (Beispielsweise eine Liste aller Quellcodedateien, die für den Test relevant sind oder Beispiele für die Nutzung der verwendeten Technologie). Auf der anderen Seite möchte der Entwickler möglichst wenig Aufwand dabei haben, den Testauftrag zu erstellen und daher möglichst wenige Angaben machen. Dennoch liegt es auch in seinem Interesse, dass die erfolgreiche Bearbeitung dem Tester einfach gemacht wird. Zwischen diesen beiden Seiten findet ein Tradeoff statt. Der Testauftrag ist die Schnittstelle zwischen dem Editor zum Lösen eines TA, der, parallel zu dieser Arbeit, von Florian Eiteljörge entwickelt wurde ([Eiteljörge2014]), und dieser Arbeit. Er wurde daher von uns beiden gemeinschaftlich, jeweils mit Blick auf die Bedürfnisse der vertretenen Rolle, ausgehandelt. Das Ergebnis werde ich im Folgenden beschreiben.

Testaufträge enthalten sämtliches, für die Erstellung eines spezifischen Tests, notwendiges und förderliches Wissen sowie alle notwendigen Projektdateien. Die Qualität des erstellten Testfalls wird durch die Güte des Testauftrages maßgeblich beeinflusst. Wenn der Auftrag lückenhaft oder ungenau ist, dann sinkt die Wahrscheinlichkeit, dass der Tester einen Testfall erstellt, der dem entspricht, was der Entwickler mit dem Auftrag erreichen wollte.

Es gibt drei Gründe für Informationen, in den Testauftrag aufgenommen zu werden:

1. Sie sind für die Bearbeitung der Aufgabe unumgänglich, wie beispielsweise die AUT.
2. Sie helfen dem Tester bei der Bearbeitung, ohne dabei einen zu großen Mehraufwand für den Ersteller des Testauftrags zu bedeuten.
3. Sie werden von der Crowdsourcing Plattform benötigt, um den Testauftrag zu

veranschaulichen oder anzuzeigen.[1]

Ein Test ist ein kleines Stück Software und ein Teil des Testauftrags lässt sich als Spezifikation für dieses Stück Software begreifen. Genauso wie die Anforderungserhebung in großen Projekten immer eine Herausforderung ist [Cheng2007], so ist sie dies auch bei einem einzelnen Test. Im Folgenden möchte ich die Unterschiede zwischen dem typischen Requirements Engineering (RE) und der Art, wie Testaufträge erstellt werden, herausarbeiten. Dabei werde ich die fünf Teilbereiche der Anforderungsanalyse nach dem RE-Referenzmodell nach [Daimler1999] betrachten.

2.3.1. Vergleich Requirements Engineering

- **Elicitation:** Elicitation findet in der klassischen Weise, eines Dialogs zwischen dem Kunden und dem Software Engineer, nicht statt. Statt vom Software Engineer befragt zu werden, muss der gesamte Testauftrag vom Kunden, hier dem testsuchenden Entwickler, erstellt werden, bevor auch nur feststeht, wem der Auftrag letztlich erteilt wird.
- **Interpretation:** Es werden hier nicht, wie im klassischen RE, die Rohanforderungen interpretiert, da der Tester nur den fertigen Testauftrag erhält. Diesen muss der Tester allerdings interpretieren, da er unter Umständen nicht völlig klar formuliert wurde. Obwohl es sich sowohl beim Entwickler, als auch beim Tester, um Softwareentwickler handelt, kann man nicht davon ausgehen, dass sie das gleiche Vokabular benutzen. Der Entwickler nutzt in seinem Projekt in der Regel projektspezifische Begriffe, deren genaue Bedeutung im Projektkontext dem Tester fremd sind.
- **Negotiation:** Das Verhandeln über Anforderungen entfällt vollkommen, da der Testauftrag, und am Ende der fertige Test, in der Regel die einzige Kommunikation ist, die zwischen Entwickler und Tester stattfindet[2]. Aus diesem Grund ist es besonders wichtig, dass der Entwickler die Anforderungen an den Test klar formuliert; genau so, wie er Anforderungen in einer Spezifikation niederschreiben würde.
- **Documentation:** Der Testauftrag enthält, neben den Projektdateien, die Dokumentation. Aufgrund des geringen Umfangs des Zielprodukts kommt ein Testauftrag mit sehr viel weniger Feldern aus, als die üblichen Spezifikationstemplates, wie z.B. das Volere-Template [Robertson1999].
- **Verfication & Validation:** Da der Entwickler den Testauftrag nur in sehr kurzer Zeit und alleine erstellt, finden diese Punkte wenig Beachtung. Das Höchstmaß an Validation, das man erwarten kann, ist, dass der Entwickler den Testauftrag noch einmal durchliest, bevor er ihn absendet oder der Entwickler den letztlich er-

[1] Da diese Schnittstelle noch nicht feststeht, handelt es sich dabei in dieser Arbeit eher um Informationen, von denen erwartet wird, dass die Crowdsourcing Plattform sie benötigen wird.

[2] Zwar hat der Tester eine E-mail-Adresse für Rückfragen – falls der Entwickler diese angibt – allerdings ist die Antwortbereitschaft von Entwicklern nach [Shah2011] gegenüber ausgelagerten Testern im Allgemeinen sehr gering.

haltenen Test selbst prüft. Allerdings müsste man dann jede Abnahme durch den Kunden als Validation zählen – die dann auch noch durch den Kunden selbst vorgenommen wird. Verification kann lediglich die letztendliche Prüfung zwischen Testauftrag und geschriebenem Test durch den Tester sein, da es keine anderen Dokumente gibt, die gegeneinander geprüft werden können.

Requirements Management spielt bei Testaufträgen keine wirkliche Rolle, da die Anforderungen sich nicht ändern können und es nur sehr wenige gibt.

Noch ein weiterer wesentlicher Unterschied ist, dass der Testauftrag keine Abnahmetestfälle oder Bedingungen enthält. Dies wäre allerdings für eine spätere Version mit Hilfe von Metriken denkbar. Beispielsweise könnte verlangt werden, dass ein Test eine bestimmte Testabdeckung für eine Funktion erfüllt. Eine echte Abnahmeprüfung wäre bei einem Auftragsvolumen von lediglich einem einstelligen Eurobetrag nicht verhältnismäßig und der Tester muss selbst erkennen, wann er den Testauftrag zufriedenstellend gelöst hat. Wie sichergestellt werden kann, dass Tester dies nicht ausnutzen und absichtlich unbrauchbare Ergebnisse liefern, wird Teil einer anderen Arbeit sein. Denkbar wäre aber zum Beispiel ein Bewertungssystem, ähnlich wie Ebay dies handhabt, wo der Nutzer einen Händler nach einem Geschäft bewertet. Die Bewertung kann anderen Nutzern helfen, „schwarze Schafe“ direkt zu erkennen, und der Plattform selbst ermöglichen, diese bei zu vielen schlechten Bewertungen auszuschließen. Eine Firma könnte sich dann entschließen, nur Tester mit einem gewissen Mindestruf zu beschäftigen.

Die wesentlichen Unterschiede zwischen Testauftrag und gewöhnlichem Anforderungsdokument rühren vom geringen Umfang des zu entwickelnden Produktes und damit der Spezifikation, sowie der geringen Zeit, die in die Erstellung des Testauftrags gesteckt werden darf, ohne dass er ineffizient würde, her. Würde zu viel Zeit investiert, würde die Auslagerung keinen Gewinn mehr darstellen. Damit die wenige investierte Zeit bestmöglich genutzt werden kann, soll TaskElf dem Entwickler so viele Angaben wie möglich abnehmen oder – sofern dies nicht möglich ist – ihn zumindest durch Vorschläge unterstützen. So kann der Entwickler den Großteil der aufgebrachten Zeit in die Beschreibung des Auftrags stecken, die ihm keine Software abnehmen kann.

2.3.2. Felder eines Testauftrags

Ein Testauftrag besteht aus verschiedenen Feldern, die teilweise völlig automatisch, teilweise automatisch vor- und teilweise vom Entwickler vollständig selbst ausgefüllt werden. Im Folgenden erläutere ich die verschiedenen Felder und ihre Aufgabe. Alle mit einem „*“ gekennzeichneten Felder gehören zur Spezifikation.

Automatisch ausgefüllte Felder

- **Software*:** Die zu testende Software (Application Under Test (AUT)) in einer lauffähigen Version. Dazu gehören auch genutzte Bibliotheken.

- **Aufwand:** Eine Abschätzung des Aufwands, den die Erfüllung des Auftrags erfordert (niedrig, mittel, hoch). Sie errechnet sich aus den übrigen Angaben. Eine ausführliche Betrachtung der Aufwandsschätzung findet sich in 3.3.
- **Name:** Der Name des Projektes. Dieser wird auf Testerseite angezeigt und daher benötigt.
- **Tutoriallink:** Ein Link zu einem Tutorial, der durch den Testtyp vorgegebenen Technologie.
- **Standardbeispiel:** Ein vorbereitetes Standardbeispiel, das alle wesentlichen Funktionen der Test-Technologie nutzt. Nach [Pham2013] ist es sehr hilfreich für den Tester, wenn er leichten Zugriff auf ein Tutorial oder ein Beispiel hat, das ihm den Umgang mit der Testtechnologie vereinfacht. Zwar handelt es sich bei den Testern in der Crowd im Wesentlichen um Personen, die viel Erfahrung im Testen haben, aber es ist dennoch hilfreich, ihnen eine gedankliche Stütze zu geben.

Semi-manuell ausgefüllte Felder

- **zu testende Datei*:** Die Klasse oder View, die getestet werden soll.
- **zu testende Methode*:** Die Methode, die getestet werden soll. Nur bei Unit-Tests relevant.
- **Art des Tests*:** Was für eine Art von Test erwartet wird. In dieser Arbeit werden nur Unit-Tests und GUI-Tests betrachtet.
- **Codezugriff:** Regelt, welche Codeteile der Tester sehen darf und welche ihm nicht angezeigt werden dürfen. Ich werde im Abschnitt 3.2 genauer erläutern, was es hiermit auf sich hat.
- **Entlohnung:** Das Honorar, das für den Test gezahlt werden soll.
- **zusätzliche unterstützende Dokumente(*):** Zum Beispiel Teile der Softwarespezifikation oder relevante Diagramme. Diese können Teil der Testfallspezifikation sein, wenn sie im Anmerkungsfeld beschrieben werden, oder auch nur als unterstützendes Nachschlagewerk angehängt sein.
- **Normalverhalten*:** Das normale Verhalten der Methode (bei Unit-Tests), bzw. die Beschreibung des relevanten Use-Cases der zu testenden Oberfläche.
- **Vorbereitung*:** Was getan werden muss, um zur Oberfläche zu gelangen, bzw. welche eventuellen Vorschritte vor dem Methodenaufruf stattfinden müssen.
- **E-Mail-Adresse:** Die E-Mail-Adresse des Entwicklers für eventuelle Rückfragen des Testers.
- **relevante Dateien:** Eine Liste von Quellcodedateien, die für den Testauftrag relevant sind. Diese werden bereitgestellt, damit sie dem Tester hervorgehoben angezeigt werden können. Zu den relevanten Dateien gehören immer die zu testende Datei und alle im Projekt vorhandenen Testklassen.

- **AUT-Version*:** Die derzeitige Version der AUT. Diese ist nicht für den Test selbst, sondern für das Wiedereinbinden der fertigen Testklasse wichtig. Wenn diese in das aktuellere Projekt eingebunden wird, kann es sein, dass die Tests fehlschlagen oder gar nicht mehr funktionieren. Mit Hilfe der Versionsnummer wird es leichter, solchen Fehlern nachzugehen.

Manuell ausgefüllte Felder

- **zu testende Sonderfälle*:** Die Sonderfälle (bei Unit-Tests) bzw. die Erweiterungen des Use-Cases (bei GUI-Tests), die getestet werden sollen.
- **Anmerkungen*:** Zusätzliche Bemerkungen oder Anmerkungen zum Testfall. Alles, was der Entwickler loswerden möchte, für das es keine speziellen Felder gibt.

2.4. Szenario dieser Arbeit

Um die gesamte Thematik in aller Tiefe zu bearbeiten und ein entsprechendes Plugin zu erstellen, wäre deutlich mehr Zeit nötig, als mir im Rahmen dieser Arbeit zur Verfügung steht. Daher möchte ich an dieser Stelle ein paar Annahmen treffen unter deren Vorbehalt diese Arbeit steht.

1. Es werden nur GUI-Tests mit Selenium und Unit-Tests mit JUnit ausgelagert. Andere Testarten finden keine Berücksichtigung.
2. Die AUT ist mit dem play!-Framework (ab V. 2.0) geschrieben worden[3].
3. Die AUT ist in Java und HTML geschrieben.
4. Bei der Implementierung des Java-Teils der AUT wurden die „Java Code Conventions“ von Oracle beachtet[4].
5. Zur Entwicklung wird die IDE Eclipse[5] genutzt.

Außerdem werden psychologische Aspekte und Effekte weitestgehend außer Acht gelassen und nur an den Stellen kurz angeschnitten, an denen sie für Design-Entscheidungen eine größere Rolle spielen.

[3] „Play!“ ist ein Framework zur Erstellung von Webanwendungen mit Java. Siehe `http://www.playframework.com/`

[4] `http://www.oracle.com/technetwork/java/codeconventions-150003.pdf`

[5] `www.ecplise.org`

2.5. Ablauf einer typischen Anwendung von TaskElf

An dieser Stelle möchte ich anhand von Mock-Ups eine typische Anwendung von TaskElf beschreiben.

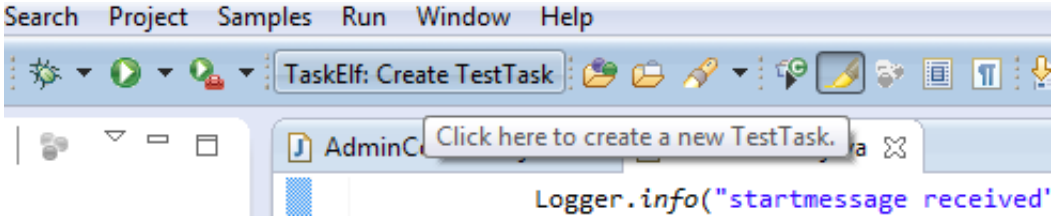

Abbildung 2.1.: TaskElf-Schaltfläche in Eclipse

Otto arbeitet gerade an einer Methode, die dafür sorgen soll, dass bei einem Nutzer beim Login ein Popup erscheint, wenn es eine neue Nachricht für ihn gibt. Eine solche Nachricht könnte beispielsweise der Versand eines bestellten Artikels sein. Die Methode ist soweit fertig und Otto möchte, dass automatische Tests für sie entwickelt werden. Er hat gestern bereits einen Testauftrag für die Grundfunktionalität ausgeschrieben. Heute möchte er einen Testauftrag für einige Sonderfälle ausschreiben. Direkt in seiner Entwicklungsumgebung hat er eine Schaltfläche, um TaskElf zu starten (Abb. 2.1). Otto tut dies.

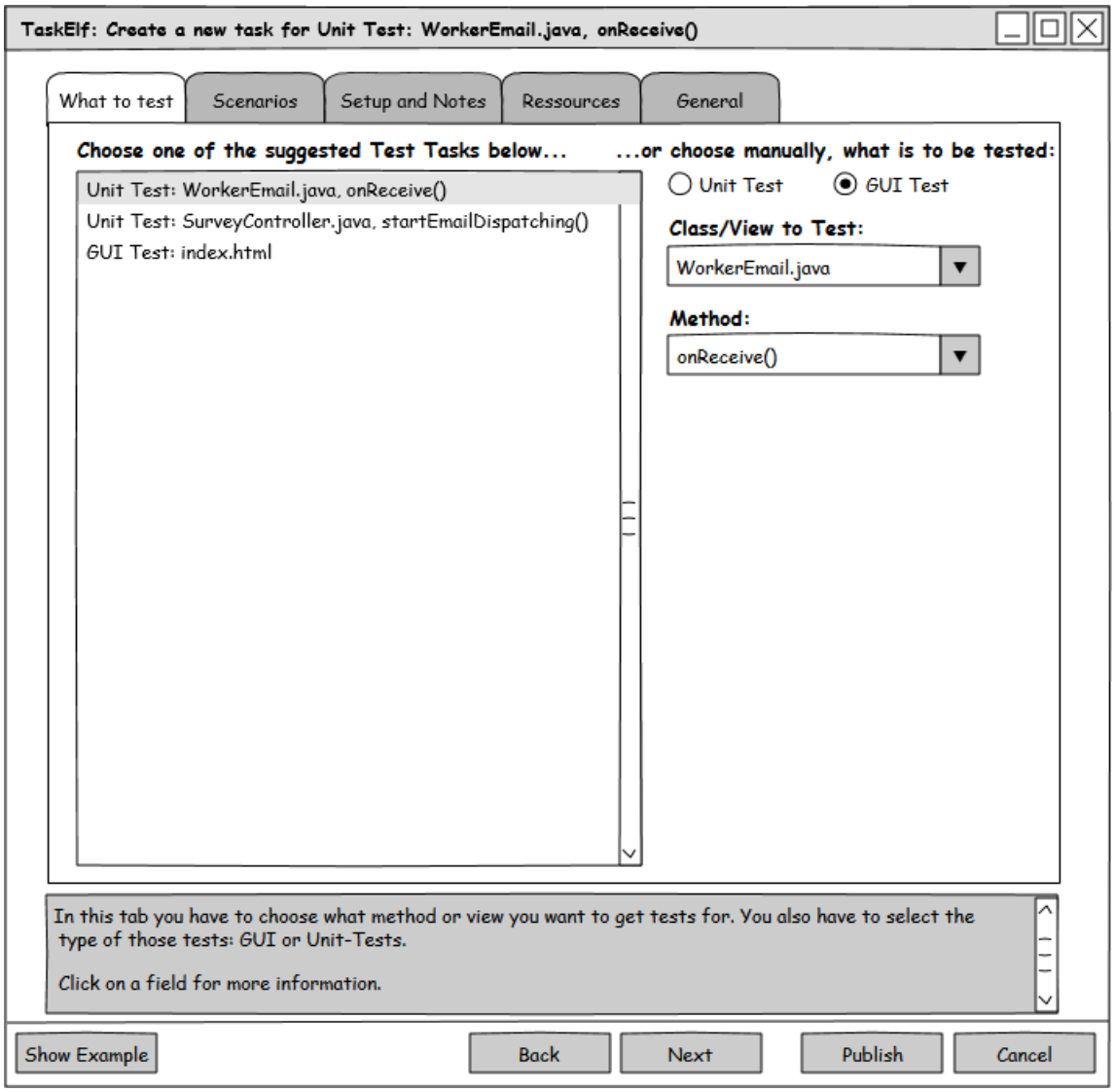

Abbildung 2.2.: Erstellung eines Testauftrags: Reiter „What to test“

Es öffnet sich TaskElf. Otto hat ein Fenster vor sich, in dem er verschiedene Felder ausfüllen soll. Diese Felder sind auf mehrere Reiter verteilt. Zuerst wird Otto aufgefordert zu wählen, für welche Methode oder View er einen Testauftrag ausschreiben möchte (Abb. 2.2). Er kann aus einer Reihe von Vorschlägen wählen, von denen TaskElf es für wahrscheinlich hält, dass Otto diese testen lassen möchte. Diese Vorschläge können zum Beispiel aus der zuletzt geöffneten Dateien („Current File“, 5.4), der Menge der Inline-Kommentare in der entsprechenden Datei („Comment Scan“, 5.4) oder der Tatsache, dass eine Methode noch nicht getestet wurde („Untested First“, 5.4) prognostiziert werden.

Otto möchte die Methode onReceive() der Klasse WorkerEmail testen. Diese steht in der Liste ganz oben und ist schon ausgewählt, da TaskElf sie als am wahrscheinlichsten eingeschätzt hat. Otto muss in diesem Reiter daher nichts weiter tun.

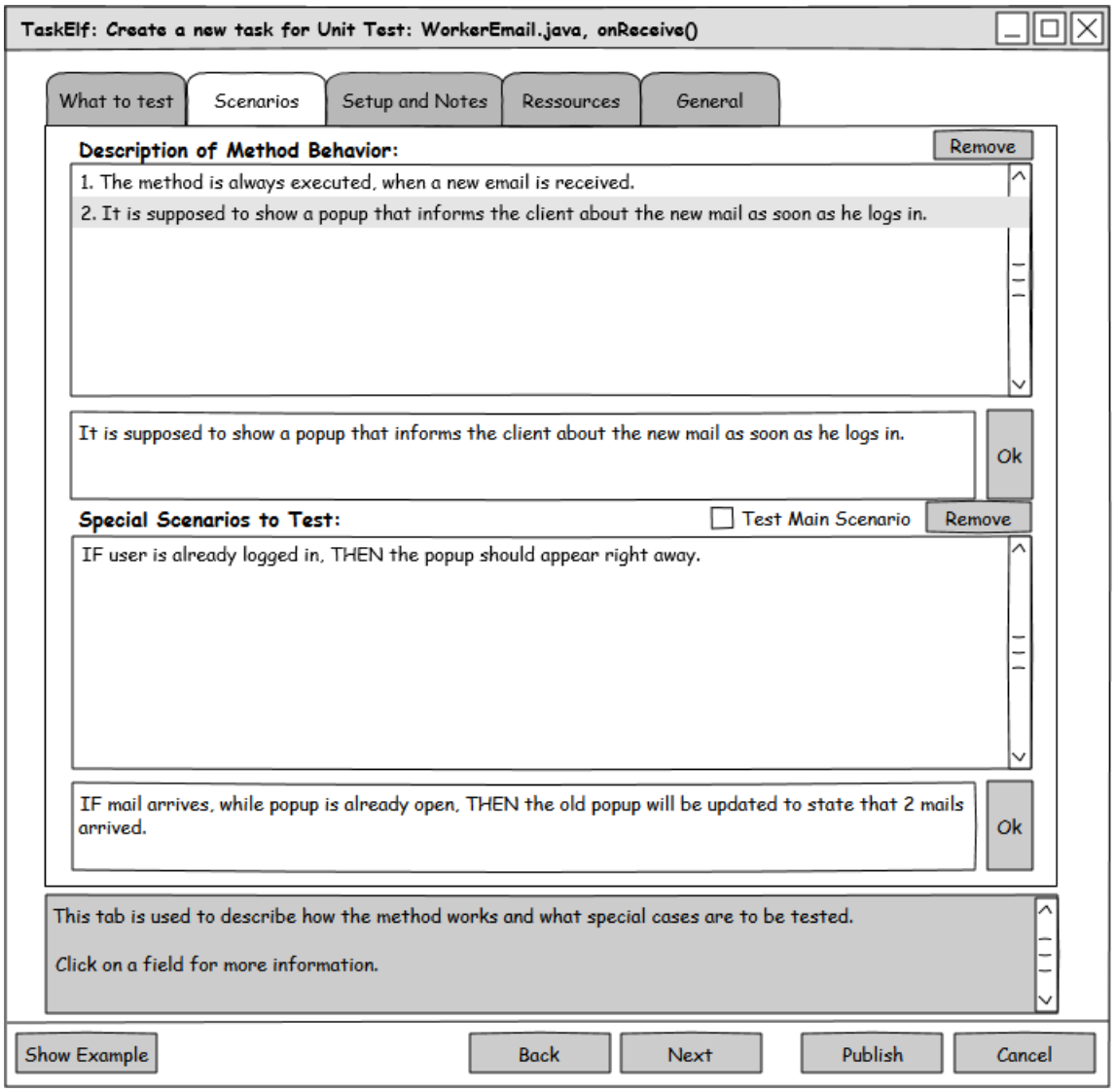

Abbildung 2.3.: Erstellung eines Testauftrags: Reiter „Scenarios“

Otto wählt nun den nächsten Reiter (Abb. 2.3). Hier wird er aufgefordert das allgemeine Methodenverhalten zu beschreiben und aufzulisten, welche Sonderfälle eintreten können und getestet werden sollen. Da Otto das Normalverhalten der Methode bereits am Vortag im letzten Testauftrag beschrieben hat, ist dieses Feld bereits ausgefüllt („Last Entry“, 5.5). Otto möchte zwei Sonderfälle durch Tests abgedeckt haben: Zum einen möchte er, dass das Popup-Fenster direkt erscheint, falls der Nutzer zum Zeitpunkt der neuen Nachricht bereits eingeloggt ist. Zum anderen soll ein gegebenenfalls schon existentes Popup aktualisiert werden, wenn eine weitere Nachricht eintrifft. Er gibt diese beiden Sonderszenarien im unteren Fenster in einer „If [...], then“-Schreibweise an.

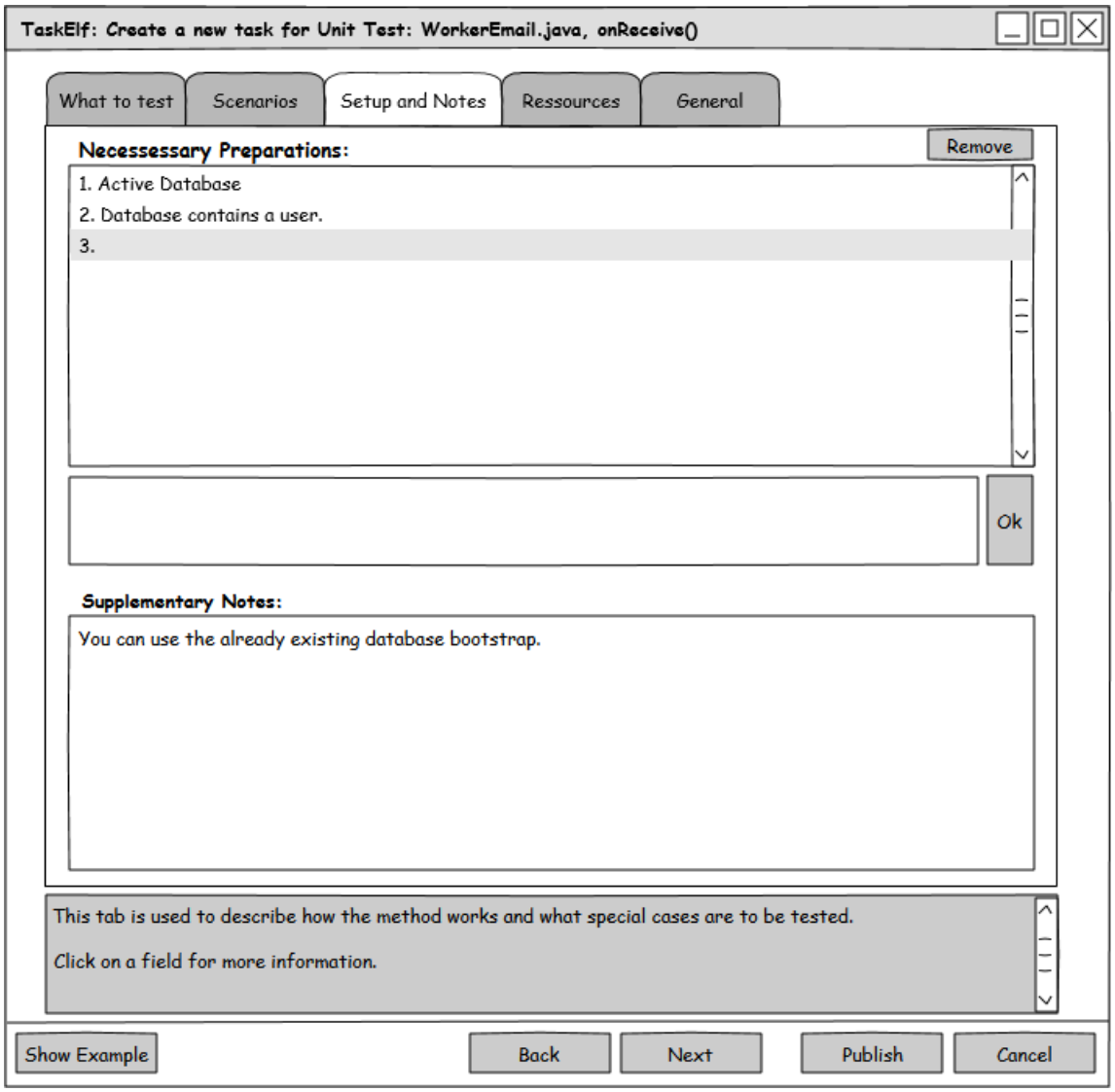

Abbildung 2.4.: Erstellung eines Testauftrags: Reiter „Setup and Notes“

Im nächsten Reiter (Abb. 2.4) wird Otto aufgefordert, nötige Vorbereitungen aufzulisten. Auch diese wurden bereits aus dem Testauftrag vom Vortag übernommen. Außerdem bekommt Otto die Möglichkeit dem Tester noch eine Freitext-Nachricht zukommen zu lassen. Er nutzt dies, um den Tester darauf hinzuweisen, dass es bereits einen für die Tests ausreichenden Datenbank-Bootstrap gibt, welche der Tester nutzen kann.

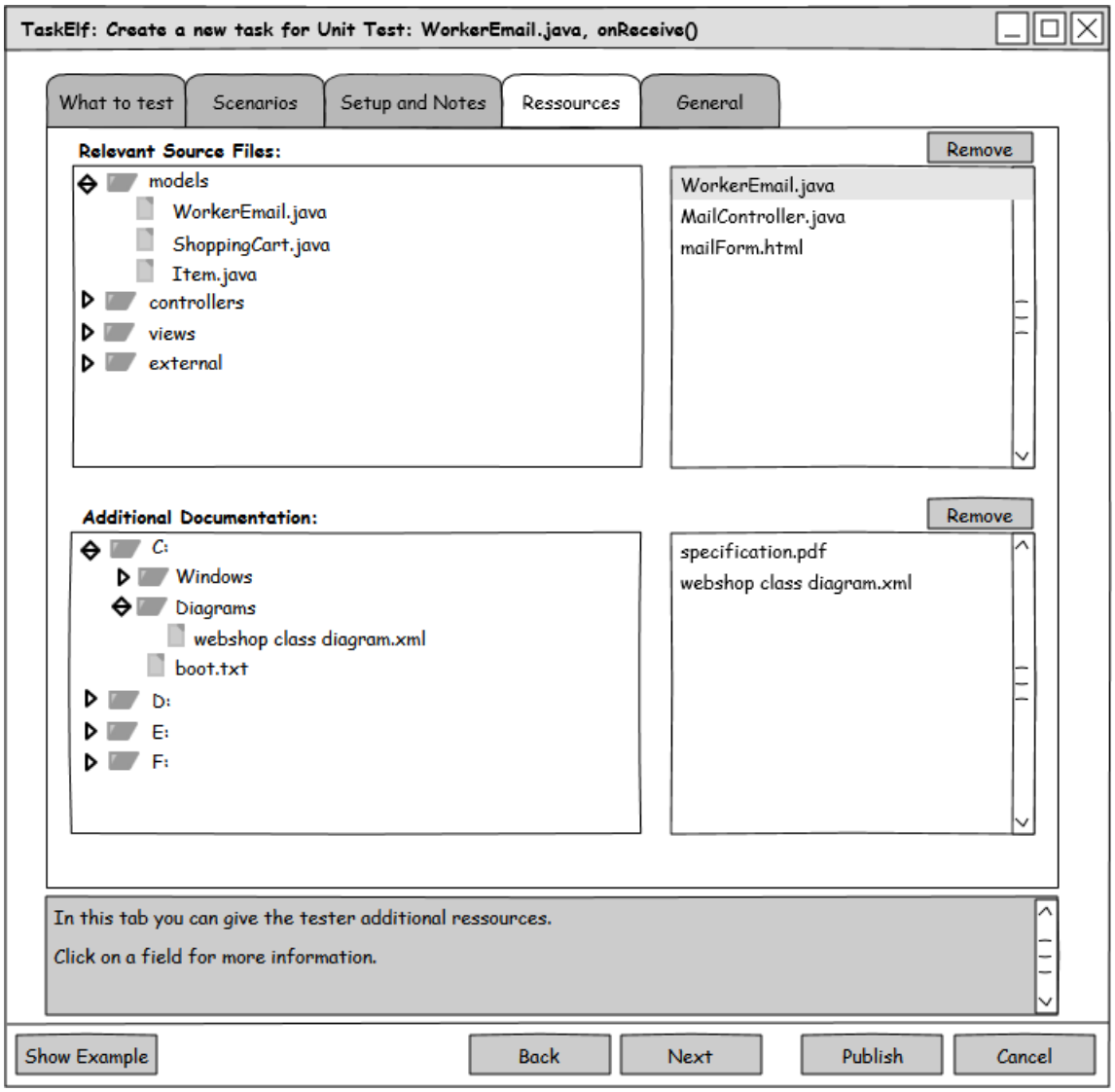

Abbildung 2.5.: Erstellung eines Testauftrags: Reiter „Ressources“

Nun gelangt Otto zum vierten Reiter (Abb. 2.5). Dieser dient dazu, dem Tester zusätzliche Ressourcen zukommen zu lassen und Codedateien als relevant zu kennzeichnen. TaskElf hat bereits eine zufriedenstellende Vorausfüllung vorgenommen. Die relevanten Dateien wurden aus der Projektstruktur bestimmt („Relevance Neighbors“, 6). Da die Datei „specification.pdf“ im Projektordner lag, wurde diese bereits zur zusätzlichen Dokumentation hinzugefügt („Documents from Project“, 5.3). Otto möchte zusätzlich noch ein Klassendiagramm anhängen. Dieses sucht er im Dateibaum auf der linken Seite und zieht es per Drag and Drop in das entsprechende Feld.

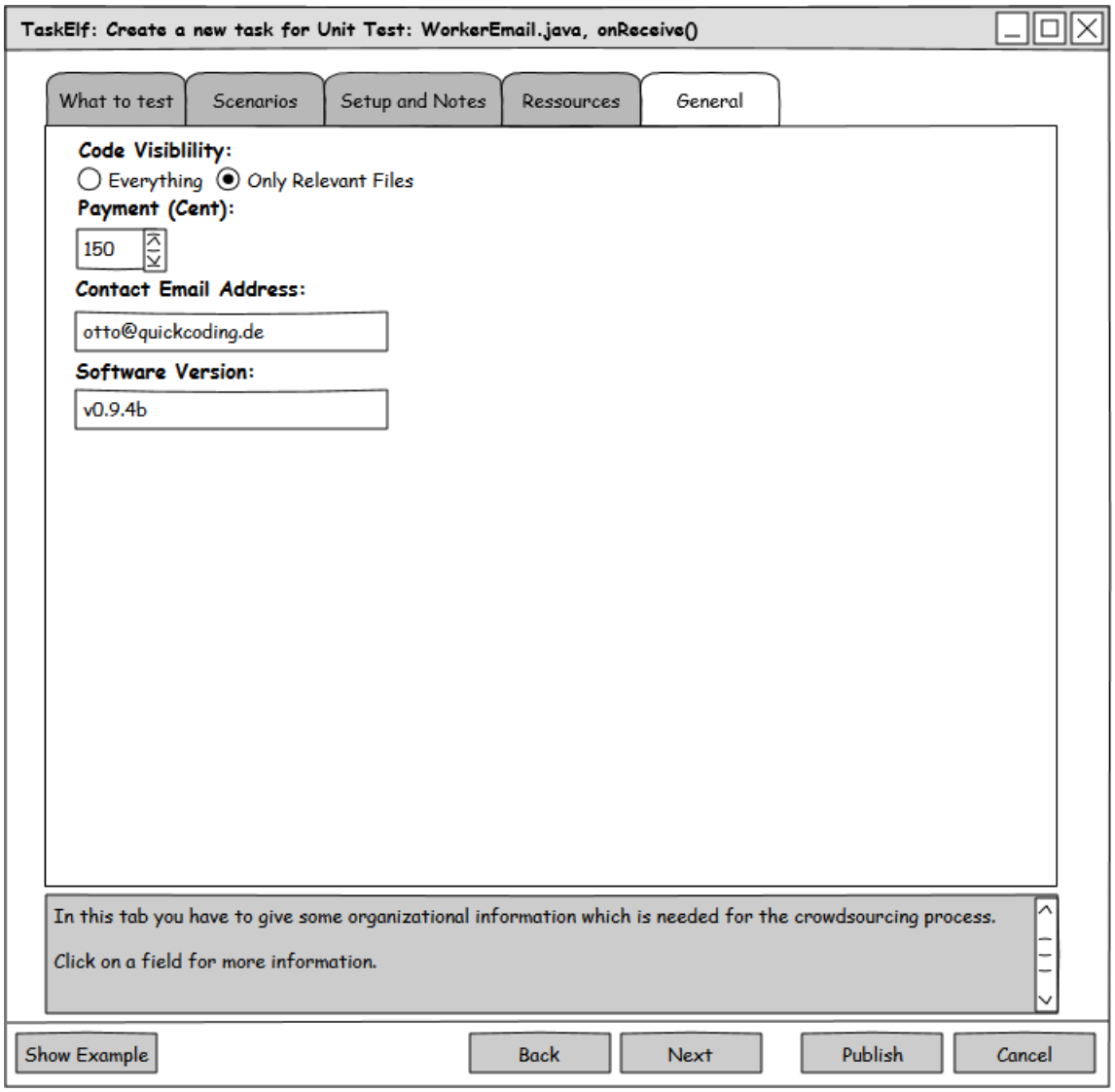

Abbildung 2.6.: Erstellung eines Testauftrags: Reiter „General“

Im letzten Reiter (Abb. 2.6) soll Otto noch einige organisatorische Angaben machen. Code Visibility, die E-Mail-Adresse und die Version sind bereits aktuell. Als Honorar wird ihm ebenfalls das des selbe vom letzten Testauftrag vorgeschlagen. Otto verdoppelt dieses, da er am Vortag lediglich ein Szenario zum Test ausgeschrieben hatte und heute zwei.

Anschließend drückt er den Button „Publish“. Er wird nach seinem Passwort gefragt. Otto gibt dieses ein und bekommt eine Meldung, dass der Testauftrag nun im Hintergrund auf die Crowdsourcing Plattform geladen wird. Otto bestätigt dies. TaskElf schließt sich und unten rechts im Eclipse wird ein Fortschrittsbalken angezeigt, um Otto mitzuteilen, wie das Erstellen und Hochladen des neuen Testauftrags fortschreitet. Otto setzt nun die Entwicklungsarbeit fort. Sobald der Testauftrag fertig auf die Crowdsourcing Plattform hochgeladen ist bekommt Otto in einem eclipse-typischen Balloon-Fenster eine Erfolgsmeldung darüber.

3. Grundlegende Konzepte des Testauftragseditors

In diesem Kapitel beschreibe ich grundlegende Konzepte von TaskElf, die weder das Benutzerinterface noch die Intentionserkennung betreffen.

3.1. Einbettung in die IDE

Um einen möglichst einfachen Zugriff auf Kontextinformationen und das Entwicklungsprojekt zu haben, soll TaskElf als Plugin für die IDE des Nutzers realisiert werden. Kontextinformationen, die innerhalb der IDE abgegriffen werden können, sind zum Beispiel das Verhalten des Entwicklers während der Entwicklung oder die Änderungshistorie des Quellcodes. Ein weiterer Grund für diese Entscheidung ist, dass die Erstellung von Testaufträgen auf diesem Wege erfolgen kann, ohne den Nutzer zu zwingen, das Programm zu wechseln. So wird der Eindruck abgeschwächt, die Entwicklung unterbrechen zu müssen, um Testaufträge zu erstellen. Viel mehr empfindet der Entwickler die Erstellung so als Teil des Prozesses. Dies soll die Akzeptanz erhöhen. Mein Prototyp ist für die Entwicklungsumgebung Eclipse erstellt.

3.2. Verbergen von vertraulichem Code

Ein großes Problem bei der Auslagerung des Testvorgangs ist, dass es Code gibt, von dem der Entwickler nicht möchte, dass er öffentlich wird. Dies ist ein häufiger Einwand von Firmenchefs gegen die Auslagerung ([Riungu2010]). Während bei der Auslagerung an entsprechende Testfirmen mit Hilfe von Verschwiegenheitserklärungen gearbeitet werden kann, ist dies beim Crowdsourcing nicht möglich[1]. Der Tester hat unter Umständen keinen Ruf zu verlieren und könnte sich mit Hilfe entsprechender Anonymisierungsvorkehrungen vor rechtlicher Verfolgung, bei Verstoß gegen eine solche Erklärung weitgehend schützen. Stattdessen gibt es aber die Möglichkeit den Code im Testauftrag nur als bereits kompilierte Bibliothek mitzusenden und lediglich die Dateien, in die der Tester zum Erstellen der Tests Einblick erhalten soll, zusätzlich in nicht kompilierter Form mitzusenden ([Eiteljörge2014]). Dies funktioniert, da der Tester in die Funktion der AUT nicht eingreifen und deren Code damit nicht verändern

[1] Dies gilt zumindest dann, wenn man nah an der Definition von Crowdsourcing bleibt und an ein „undefiniertes Netzwerk“ von Menschen auslagert. Im Crowd-Testing beispielsweise wird dies oft nicht getan und die Tester sind Angestellte der Crowd-Testing-Firma. Dort können dann auch Verschwiegenheitserklärungen effektiv genutzt werden.

muss. Einzig sein geschriebener Test kommt hinzu und muss neu kompiliert werden. So hat der Tester nur Kenntnis über den Code, den er wirklich kennen muss. Testfälle, die sich auf Code beziehen, der nicht veröffentlicht werden darf, können so naturgemäß jedoch nicht ausgelagert werden. Um Reverse Engineering vorzubeugen, kann der kompilierte Code zusätzlich durch einen Obfuscator[2] verschleiert werden.

3.3. Aufwandseinschätzung

Externe Tester haben nach [Shah2011] gerne eine Abschätzung des Schwierigkeitsgrades und damit des Aufwands eines Auftrages, bevor sie diesen übernehmen. Diese Abschätzung lässt sich automatisiert oder manuell durch den Provider vorgeben. Die manuelle Festlegung hat in unserem Fall den Nachteil, dass der Entwickler andere Ziele als eine korrekte Abschätzung verfolgt. Die oberste Motivation des Entwicklers ist es, seine Testaufträge möglichst schnell und möglichst korrekt bearbeitet zu bekommen. Eine Aufwandsabschätzung wäre – neben dem Honorar – ein zweites Werkzeug, um die Dauer zu beeinflussen, die ein Testauftrag benötigt, um von einem Tester übernommen zu werden. Der Entwickler ist also grundsätzlich bestrebt den Aufwand möglichst niedrig anzusetzen, da er so schneller von einem Tester übernommen wird. Eine negative Auswirkung von zu niedrig angesetzten Aufwandsschätzungen ist, dass Tester Aufträge, die ihnen zu schwierig oder zeitaufwändig erscheinen, unbearbeitet zurückgeben. Dieser Grund für das Zurückgeben ist allerdings für den Entwickler nur ersichtlich, wenn er vom Tester angegeben wurde und auch dann erst eine Weile nach der Erstellung des Testauftrags. Dies führt daher insgesamt zu Verzögerungen und senkt damit die Akzeptanz des Systems.

Ein weiterer Nachteil ist, dass dem Entwickler unter Umständen Erfahrung und Know-How fehlen, um den Aufwand korrekt einzuschätzen. Dies wird umso wahrscheinlicher, da Entwickler eher Tests auslagern werden, deren Erstellung ihnen weniger liegt.

Ein Weg die Probleme mit der manuellen Abschätzung zu umgehen, ist es, die Alternative zu wählen und diese Abschätzung durch TaskElf automatisiert vornehmen zu lassen. In eine solche Metrik können verschiedene Faktoren einfließen:

- **Anzahl der relevanten Quelldateien:** Je mehr Dateien relevant sind, desto mehr muss beim Test beachtet werden[3].
- **Anzahl der Vorbereitungsschritte:** Müssen mehr Vorbereitungen getroffen werden, um den Testauftrag zu bearbeiten, macht es diesen aufwändiger.
- **Anzahl der zu testenden Szenarien:** Im Mittel steigt der Aufwand linear mit der Anzahl an Szenarien, die getestet werden müssen. Dies ist das verlässlichste

[2] Bei Obfuscatorn handelt es sich um Programme, die den Code für den menschlichen Betrachter möglichst unverständlich gestalten, ohne dass ein zusätzlicher Aufwand für die Maschine besteht, die ihn ausführen muss.

[3] Solche Parameter sind gefährlich, weil viele Nutzer beim Bekanntwerden derer dazu übergehen würden, sich das Wissen zu nutze zu machen und die Liste beispielsweise einfach leer lassen würden. Dieses „Gaming the System“ ist allerdings nicht Fokus dieser Arbeit.

Kriterium, auch wenn natürlich Fälle denkbar sind, wo der Test eines einzelnen Szenarios aufwändiger ist, als der Test von mehreren, einfacheren Szenarien.

- **Honorar:** Das Honorar beeinflusst maßgeblich die Wahrscheinlichkeit, dass ein Tester einen Testauftrag übernimmt. Der Entwickler hat im Wesentlichen nur zwei Gründe, das Honorar zu erhöhen: Er benötigt einen Testauftrag besonders schnell oder er schätzt ihn selbst als komplex ein. Ich denke, dass letzterer Grund überwiegt. Unter dieser Prämisse lässt sich die relative Honorarhöhe im Vergleich zum durchschnittlichen Honorar für Testaufträge in diesem Projekt als Indikator des Aufwands nutzen. Auf diesem Wege kann die manuelle Einschätzung des Nutzers auch in die automatisierte Einschätzung mit einfließen.
- **Existenz von Beispielen im Projekt:** Wenn bereits viele ähnliche Tests im Projekt existieren, an denen sich der Tester orientieren kann, dann wird im Allgemeinen weniger Zeit für den Testauftrag benötigt.

Verwendete Metrik

Ich halte die Anzahl der Vorbereitungsschritte und der Szenarien für am geeignetsten, da jeder Eintrag direkt eine nötige Handlung für den Tester darstellt. Dabei ist aber zu beachten, dass ein zu testendes Szenario in der Regel mehr Aufwand bedeutet, als ein einzelner Vorbereitungsschritt (Ausnahmen sind allerdings denkbar). Daher sollten die Szenarien mit einem zusätzlichen Gewicht versehen werden, um diesem Umstand Rechnung zu tragen. So kann man einen relativen Aufwandswert errechnen:

$$Aufwand_{rel} = \alpha * Anzahl_{Szenarios} + Anzahl_{Vorbereitungen}$$

Der resultierende Wert muss danach noch auf den Wertebereich (niedrig, mittel, hoch) gemappt werden. TaskElf verwendet als Konfiguration:

$$\alpha = 3$$
$$niedrig \leq 8 < mittel \leq 17 < hoch$$

Zusätzlich sollte in Zukunft der Umstand Beachtung finden, dass die Vorbereitung gegebenenfalls von bereits existenten Tests übernommen werden kann. In diesem Fall dürften die Vorbereitungspunkte nicht mehr in den Aufwand einfließen.

3.4. Erweiterbare Architektur

Die Aufgabenstellung sieht als Anforderung an TaskElf eine leichte Erweiterbarkeit vor. Insbesondere neue Konzepte zur Informationsgewinnung sollen schnell und ohne größere Veränderungen eingebunden werden können. Dies ist deshalb von so zentraler Bedeutung, da die heuristische Erkennung der Nutzerintention sehr viele Möglichkeiten bietet und im Rahmen dieser Arbeit nur ein Bruchteil davon umgesetzt werden

kann. Daher soll es die Möglichkeit geben, TaskElf auch später noch zu erweitern, ohne die Architektur ändern zu müssen.

Umsetzung

In diesem Abschnitt beschreibe ich das verwendete Datenmodell und erläutere im Anschluss, wieso ich bestimmte Entscheidungen getroffen habe und es besonders gut erweiterbar ist.

Beschreibung des Datenmodells

Das Datenmodell von TaskElf (Abbildung 3.1) wird von Außen über die Klasse TestTask angesprochen. TestTask speichert alle Informationen, die für die Erstellung der Testauftragsdatei nötig sind. Die einzelnen Felder (im Sinne von 2.3.2) des Testauftrags erben von der Klasse Field. In der Abbildung sind stellvertretend für die Gesamtmenge der Felder, die Felder Example (enthält ein Beispiel für die Nutzung der zu verwendenden Technologie – zum Beispiel JUnit), Description (enthält die Beschreibung des Testauftrages) und RelevantCode (enthält eine Liste von Pfaden zu allen relevanten Quellcodedateien) zu sehen.

Viele Felder werden auch in der GUI angezeigt und können dort editiert werden. Auf den Inhalt des Feldes wird über die toString-Methode zugegriffen. Diesen Inhalt speichern die Felder in Form einer Liste von Strings (contentList). Jedes Element dieser Liste enthält einen Eintrag für das Feld. Der Umgang mit diesen Einträgen hängt von der Art des Feldes ab. Jedes Feld kann eine Liste sein oder Dateipfade enthalten. Wenn eine Liste angezeigt werden soll, werden alle Elemente in der Ausgabe durch Kommata konkateniert, bei Pfaden wird nur der Dateiname mitsamt Endung zurückgegeben. Ist ein Feld keine Liste, wird hingegen nur das erste Element der contentList zurückgegeben. Analog verändert TestTask die Liste über die setContent-Methode mit einem String, der der Form der Ausgabe von toString entsprechen muss.

Die initialen Einträge der contentList erhält ein Feld aus seinen Informationsquellen. Eine Informationsquelle ist eine Konkretisierung der abstrakten Klasse AbstractInformationSource. Jede Informationsquelle hat genau ein Feld für das sie Informationen bereitstellt. Informationen bestehen aus ihrem Inhalt und einem Gewicht, das eine Einschätzung darüber abgibt, wie verlässlich diese Information ist. Ein Feld kann beliebig viele Informationsquellen besitzen und ermittelt seine contentList mit der Methode extractContent. In dieser Funktion werden die Informationen aus allen Informationsquellen dieses Feldes angefordert (über extractInformation) und verknüpft, worauf ich später im Abschnitt 5.6 noch genauer eingehen werde. Die resultierende sortierte Liste von Informationen wird anschließend in die contentList übernommen.

Ein vorgeschlagener Testauftrag wird über die Kombination aus Test-Typ, zu testen-

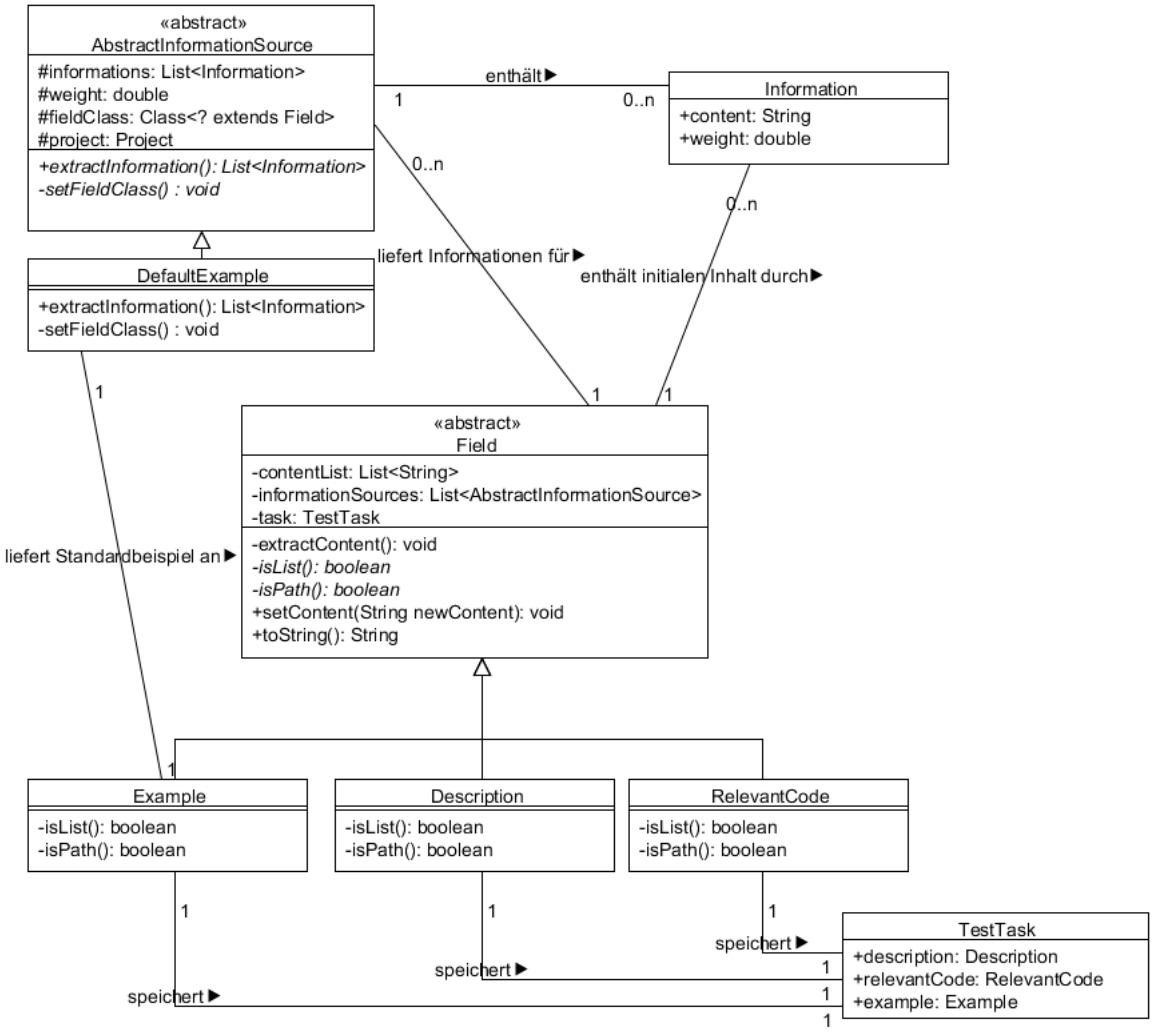

Abbildung 3.1.: Ein Ausschnitt aus dem Datenmodell von TaskElf

der Klasse/View und gegebenenfalls der zu testenden Methode eindeutig identifiziert[4]. Diese Daten stehen in einem besonderen Feld, das zum Aufbau der Datenstruktur genutzt wird. Es wird als erstes erstellt und für alle Informationen, die es extrahiert und die über einem vorgegebenen Schwellwert liegen, wird ein TestTask-Objekt erstellt. Die entstehende Liste von TestTasks bildet die Menge an vorgeschlagenen Testaufträgen, aus denen der Entwickler bei der Erstellung auswählen kann. Bei der Erstellung eines TestTasks werden seine Field-Klassen direkt miterstellt. Als nächstes werden alle Konkretisierungen von AbstractInformationSource instanziiert und ihren entsprechenden Feldern hinzugefügt. Eine Instanz einer Field-Klasse ruft seine extractContent-

[4]Zwar kann es mehrere Testaufträge zu ein- und derselben Methode oder View geben, diese unterscheiden sich dann allerdings nur in Feldern, die nicht automatisch ausgefüllt werden können. Somit sind die entsprechenden Vorschläge identisch.

Methode selbstständig auf, sobald ein anderes Objekt den Inhalt dieses Feldes anfordert und dieser noch leer ist.

Begründung der Entwurfsentscheidungen

In diesem Teil werde ich einige Entwurfsentscheidungen, die der Architektur zu Grunde liegen, erläutern.

- **Informationsspeicherung als String:** In der verwendeten Architektur werden alle Informationen in den Feldern als Strings gespeichert. Dies gilt auch für Felder, die andere Datentypen, wie Pfade oder Integer, speichern. Dies mag ungewöhnlich wirken, hat aber den Vorteil, dass alle Felder dieselbe Informationsklasse und Konkretisierungen derselben abstrakten Informationsquelle nutzen können. So wird der Aufwand für die Erstellung eines neuen Feldes sehr gering gehalten. Um den korrekten Umgang mit den Daten sicherzustellen, müssen nur die beiden boolean-Methoden isPath und isList in der Kindklasse implementiert werden. Da die Werte für die Oberfläche sowieso als Strings angezeigt werden, ist hier kein weiterer Aufwand nötig (die einzige Ausnahme bilden hier Zahlen, die für die Anzeige, zum Beispiel in einem Spinner, gecastet werden müssen).
- **Verwendung von isList und isPath statt eigener Klasse:** Sowohl bei Listen, als auch bei Pfaden werden die Daten anders verarbeitet. Bei Pfaden wird lediglich die Ausgabe auf den Dateinamen beschnitten. Bei Feldern, die Listen anzeigen, wird als Besonderheit die gesamte contentList genutzt, während diese bei Nicht-Listen grundsätzlich auf ein Element reduziert wird. Die Ein- und Ausgabe verläuft dementsprechend unterschiedlich. Bei einem Nicht-Listen Feld würde das Setzen der contentListe mit dem String "1, 2, 3" zu der Liste {"1, 2, 3"} führen. Dies ist zum Beispiel beim „Anmerkungen“-Feld des Testauftrages der Fall, wo nur ein Freitext existiert und dieser beliebig formatiert sein kann. Dieselbe Aktion würde bei einem Listenfeld in {"1", "2", "3"} als contentList resultieren. Diese Besonderheiten sind beliebig miteinander kombinierbar, weshalb eine eigene Feldklasse für jede Kombination direkt vier Klassen erfordern würde, die sich nur sehr geringfügig voneinander unterscheiden. Hier würde sich prinzipiell der Einsatz des Decorator-Patterns anbieten. Da dies bei nur zwei verschiedenen Sonderfällen allerdings unnötig komplex wäre, weitere kombinierbare Sonderfälle unwahrscheinlich erscheinen und das Problem sich auch durch ein einfaches IF-Konstrukt in der toString- und der setContent-Methode lösen ließe, habe ich diesen Weg gewählt.
- **Dynamische Ermittlung der konkreten Informationsquellen:** Die Namen der Informationsquellenklassen werden mit Hilfe der Reflections-API von Java ermittelt und anschließend über diesen instanziiert. Bei der Erstellung von neuen Informationsquellen hat dieses Verfahren den Vorteil, dass nichts weiter nötig ist, als eine neue Klasse zu erstellen, die AbstractInformationSource konkretisiert, um eine neue Informationsquelle einzubinden. Es müssen lediglich zwei Methoden realisiert werden: setFieldClass wird bei der Instanziierung aufgerufen und

muss nur mit einer Zuweisung der Variable fieldClass auf [Zielklasse].class gesetzt werden; extractInformation enthält die Informationsgewinnung. Damit gibt es keinen Overhead bei der Erstellung einer neuen Informationsquelle. Die neue Klasse hat Zugriff auf den aktuellen Task und die Klasse Project, die Metainformationen, wie den Eclipse-Ressourcen zum aktuellen Java-Projekt, enthält. So stehen zahlreiche Informationen zur Verfügung, aus denen die Informationen gewonnen werden können.

- **Eigene Klassen für alle unterschiedlichen Felder:** Die meisten Kindklassen von Field sind, abgesehen von dem Methoden isList und isPath, fast vollkommen identisch mit Field. Daher hätte sich angeboten, statt eigenen Klassen, direkt die Klasse Field zu instanziieren. Während der Entwicklung hat sich allerdings gezeigt, dass bei Änderungen und Erweiterungen oft kleine spezifische Anpassungen für spezifische Felder nötig werden. Um diesem Umstand Rechnung zu tragen, haben alle Felder eine eigene Klasse. So sind sie leichter individuell anpassbar und es gibt ein konsistenteres Gesamtbild, als wenn nur ein Teil der Felder eine eigene Klasse hätte.

4. Benutzeroberfläche

In diesem Kapitel möchte ich die Benutzeroberfläche und die ihr zu Grunde liegenden Entscheidungen erläutern. Dazu werde ich zuerst den momentanen Zustand der Oberfläche in TaskElf besprechen und anschließend erläutern, welche Erweiterungen ich darüber hinaus noch für sinnvoll erachte.

4.1. Ist-Zustand

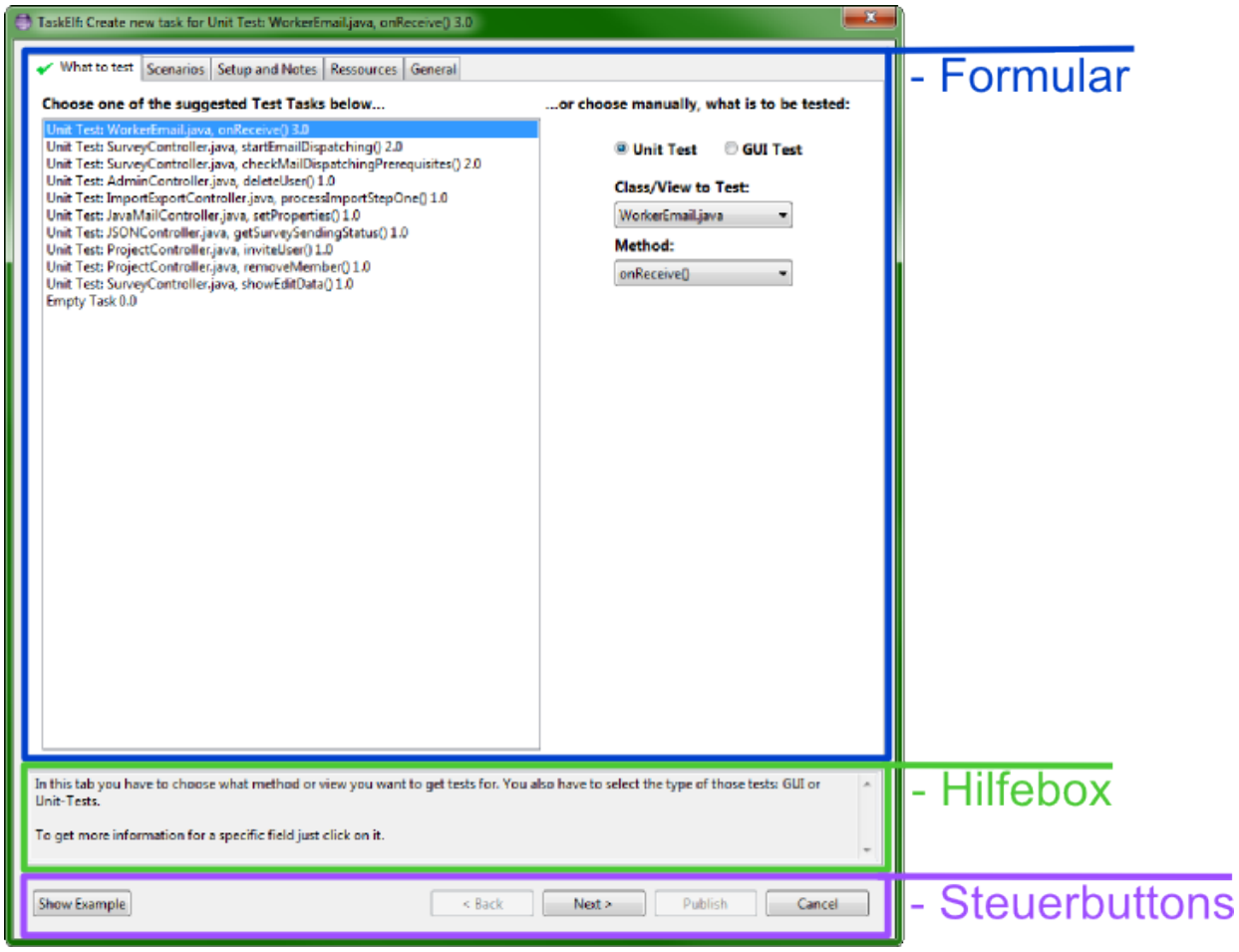

Abbildung 4.1.: Aufteilung des Inferfaces von TaskElf.

Den Kern von TaskElf bildet der Formularteil, welcher der Eintragung der für den Tes-

tauftrag nötigen Informationen dient. Er besteht aus fünf Reitern. Darunter befindet sich eine Hilfebox, die Erläuterungen zum gerade in den Reitern aktiven GUI-Element gibt oder - falls kein Element ausgewählt ist - die grundsätzliche Intention des aktiven Reiters erläutert. Schließlich befindet sich am unteren Ende ein Bereich mit verschiedenen Steuerbuttons. Die Aufteilung ist in Abbildung 4.1 dargestellt.

Im Folgenden werde ich zunächst auf den Formularbereich und die Steuerbuttons näher eingehen.

4.1.1. Formular

Das Formular besteht aus fünf Reitern, die verschiedene Felder des Testauftrags enthalten. Der erste Reiter dient der Wahl, für welche View oder Methode überhaupt Tests erstellt werden sollen. In den nächsten beiden Reitern muss der Entwickler vermitteln, was durch die geforderten Tests genau getestet werden soll. Im vierten Reiter kann der Entwickler dem Tester Hilfestellungen für seine Arbeit geben. Der fünfte und letzte Reiter dient organisatorischen Angaben, die für den Crowdsourcing-Vorgang benötigt werden.

Wenn ein Reiter vom Nutzer geöffnet wurde, wird er durch ein kleines grünes Häkchen markiert. Ein Testauftrag kann nur ausgeschrieben werden, wenn der Nutzer zuvor alle Reiter betrachtet hat. Die Häkchen ermöglichen es, dem Nutzer jederzeit zu signalisieren, welche Reiter vor der Veröffentlichung noch von ihm bearbeitet werden müssen.

Dass der Nutzer alle Reiter einmal betrachtet haben muss, hat einen einfachen Grund: Auch wenn viele Felder durch Heuristiken vorausgefüllt werden können, kann nur der Nutzer selbst entscheiden, ob diese automatisiert bestimmten Angaben auch korrekt sind. Fehler sind durch das Crowdsourcing potentiell direkt mit finanziellen Einbußen verbunden. Dadurch, dass der Nutzer gezwungen wird, alle Eingaben einmal zu betrachten, wird ihm das Gefühl vermittelt, dass er die Verantwortung dafür trägt, dass alle Eingaben richtig sind.

Die einzelnen Reiter beschreibe ich im folgenden in der Tiefe.

Reiter „What to test“

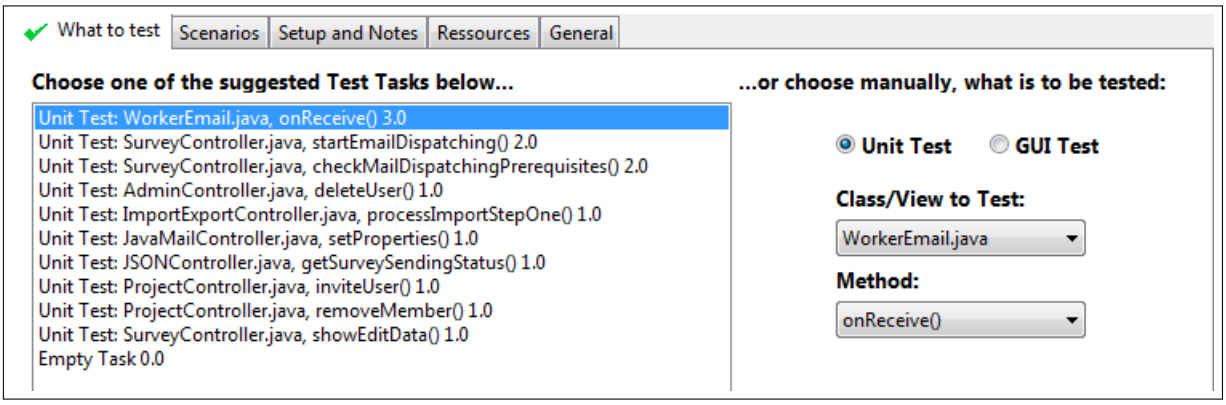

Abbildung 4.2.: Der Ist-Zustand des Reiters „What to test“

Die Entscheidung, welche Komponente der AUT vom Testauftrag abgedeckt werden soll, nimmt der Nutzer über eine Listenauswahl vor (links, Abb. 4.2). In dieser Liste stehen alle Testaufträge, die durch die Intentionserkennung als wahrscheinlich gewertet wurden sowie ein leerer Testauftrag, für den Fall, dass alle Prognosen falsch sind. Die Testaufträge werden in der Liste durch den Typ des Tests, der zu testenden Datei sowie gegebenenfalls der zu testenden Methode repräsentiert[1].

Auf der rechten Seite können die Angaben manuell vorgenommen werden. Dabei gibt es Radio-Buttons für den geforderten Testtyp und Dropdown-Listen für die Klasse bzw. View und gegebenenfalls die Methode, die getestet werden soll. Die Dropdown-Liste für letztere ist dabei nur sichtbar, wenn ein Unit-Test als Typ gewählt wird. Die obere Dropdownliste enthält alle Klassen und HTML-Dokumente des Projektes, während die untere nur die Methoden zur Auswahl stellt, die sich in der momentan gewählten Klasse befinden.

Von der Wahl des Testkandidaten hängen die Ergebnisse einiger Heuristiken ab, weshalb diese zuerst stattfinden muss. Die Wahl des Testtyps hat darüber hinaus auch noch Einfluss auf Beschriftungen und Hilfetexte einiger Felder in den nächsten Reitern.

[1] Ich gehe in dieser Arbeit vom einfachen Fall aus, dass keine Klassennamen bzw. Viewnamen und keine Methodennamen innerhalb von Klassen doppelt vorkommen. In der Praxis kommt dies jedoch vor und die Bezeichner sollten gegebenenfalls durch weitere Teile der Signatur eindeutig gemacht werden.

Reiter „Scenarios“

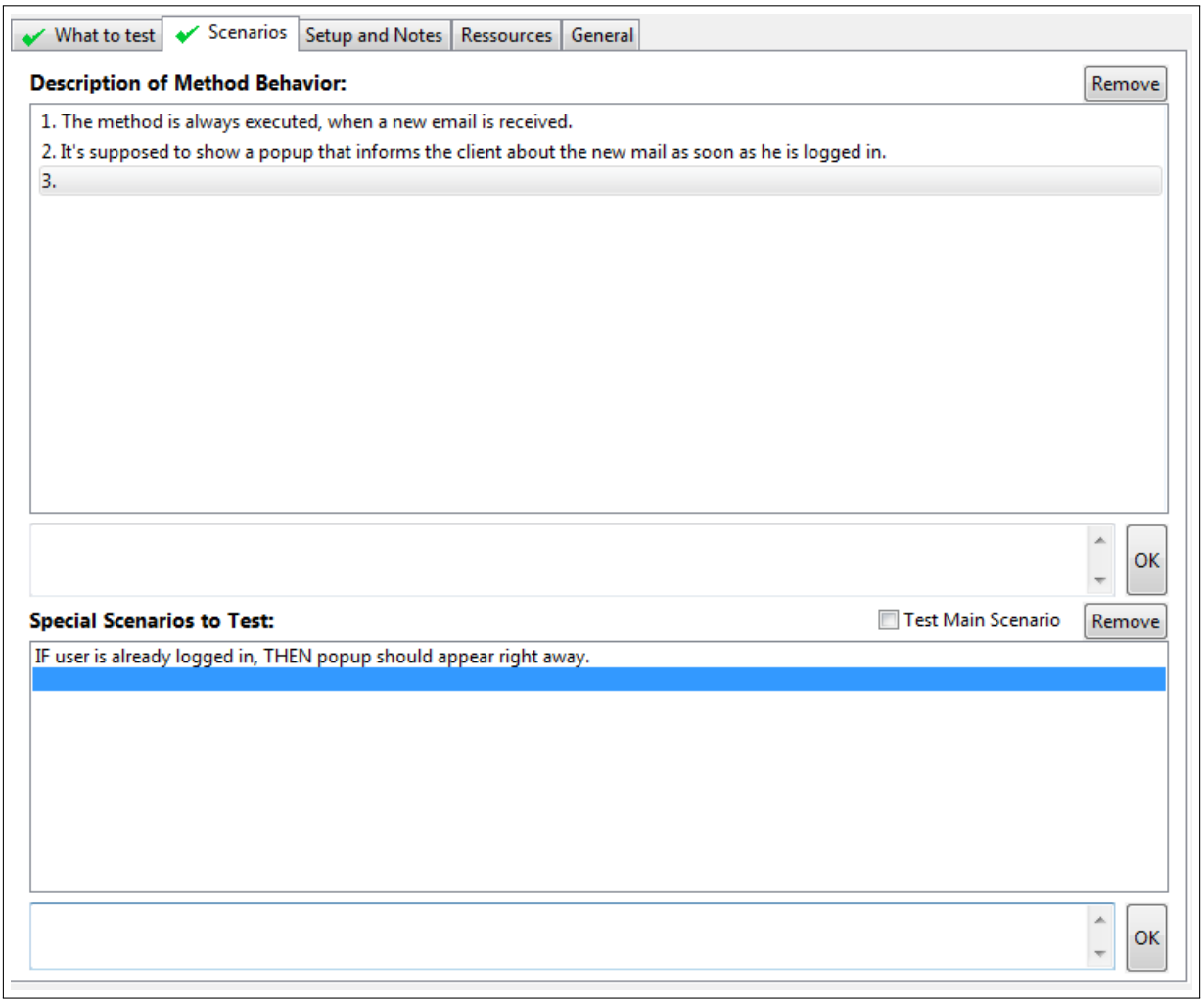

Abbildung 4.3.: Der Ist-Zustand des Reiters „Scenarios“

Der Reiter „Scenarios“ dient der Beschreibung das Normalverhaltens des Testkandidaten und von Abweichungen bzw. Sonderfällen, die getestet werden sollen.

Das obere Feld (Abb. 4.3) enthält dabei des Normalverhalten. Wie genau dieses dargestellt wird, hängt dabei vom Testtyp ab. Bei Unit-Tests muss hier eine Beschreibung erfolgen, womit die zu testende Methode aufgerufen werden muss, was sie tut und was sie zurückgibt. Bei GUI-Tests muss hier der normale Handlungsablauf im Umgang mit der GUI beschrieben werden. Dies sollte ähnlich wie die Beschreibung eines Use-Cases aussehen: Der normale Ablauf soll Schritt für Schritt chronologisch beschrieben werden. Dabei ist jeweils der Handelnde und was er tut, hervorzuheben. Sonderfälle sollen hier nicht beschrieben werden. Zusätzlich soll allerdings auch angegeben werden, was im Erfolgsfall nach vollständiger Durchführung passieren sollte.

Im unteren Feld werden nun diejenigen Sonderfälle oder Abweichungen vom obe-

ren Normverhalten eingetragen, für die im Rahmen dieses Testauftrags automatische Tests vom Tester geschrieben werden sollen. Im Fall von Unit-Tests sind dies Sonderfälle der Methode. So wäre es zum Beispiel für eine Methode, die mit einem Laufwerksbuchstaben aufgerufen wird und das entsprechende Laufwerk formatieren soll, ein Sonderfall, wenn das vorgegebene Laufwerk nicht existiert. Bei GUI-Tests sind die Einträge in dieses Feld äquivalent zu den Erweiterungen eines Use-Cases. In beiden Fällen kann der Sonderfall in der Form „Wenn [Bedingung], dann [Sonderverhalten]“ beschrieben werden. Diese Schreibweise hat den Vorteil, dass sie den Nutzer zwingt, sich Gedanken über beide Teile zu machen und diese explizit anzugeben. Außerdem können Bedingung und Sonderverhalten durch die Signalwörter „If“ und „then“ automatisiert auf der Testerseite ausgelesen werden. Um diese Vorteile zu nutzen, muss der Anwender allerdings dazu aufgefordert werden, genau diese Schreibweise zu nutzen. Dies geschieht in TaskElf. Zusätzlich ist es bei GUI-Tests optional auch möglich, für jeden Sonderfall diejenigen Punkte des Normablaufs zu markieren, die sich durch diesen Sonderfall verändern. Dies geschieht durch strg-linksklicken auf die entsprechenden Punkte in der oberen Liste. Sie werden dadurch grün hinterlegt. Ein nochmaliger Strg-Linksklick löst die Verknüpfung wieder. Es sind immer nur die Verknüpfungen des momentan in der unteren Liste markierten Sonderfalls sichtbar und editierbar.

Die genaue Beschreibung, was in welche Felder eingetragen werden muss und was es dabei zu beachten gibt, wird für jedes Feld in der Hilfebox erläutert, an deren Inhalt der Nutzer sich orientieren kann. Das verwendete Verfahren mit Normalverhalten und Sonderfällen funktioniert nicht für alle Fälle zufriedenstellend. Potentiell sind auch Sonderfälle von Sonderfällen (beliebigen Grades) möglich. TaskElf beschränkt sich in der vorliegenden Version allerdings auf die einfachen Fälle. Eine weitere Einschränkung an dieser Stelle ist, dass das System keinen Testauftrag ermöglicht, der sich auf verschiedene Use-Cases bezieht, auch wenn eine View grundsätzlich beliebig viele Use-Cases haben kann, die auf ihr ablaufen. In diesem Fall müsste der Nutzer aber mit dem vorliegenden System für jeden Use-Case einen eigenen Testauftrag ausschreiben.

Die Listen lassen sich über das daruntergelegene Textfeld editieren. Es enthält immer den Text des gerade gewählten Listenfeldes. Durch einen Klick auf den rechts neben dem Textfeld gelegenen „OK“-Button oder Drücken der Return-Taste auf der Tastatur wird der Text des Textfeldes in die Liste übernommen. Wenn dieser Eintrag sich am Ende der Liste befand, entsteht in der Liste direkt unter diesem wieder ein neuer leerer Eintrag, der direkt automatisch selektiert wird. Der Nutzer kann ohne Verzögerung weitertippen, um diesen neuen Eintrag zu füllen. Dies ermöglicht es schnell, ausschließlich durch Tastatureingaben eine Menge von aufeinanderfolgenden Einträgen anzulegen.

Durch einen „Remove“-Button am rechten oberen Rand einer Liste lässt sich das momentan ausgewählte Element der Liste entfernen. Soll der Normalfall auch getestet werden, kann dies über eine Checkbox angegeben werden.

Reiter „Setup and Notes“

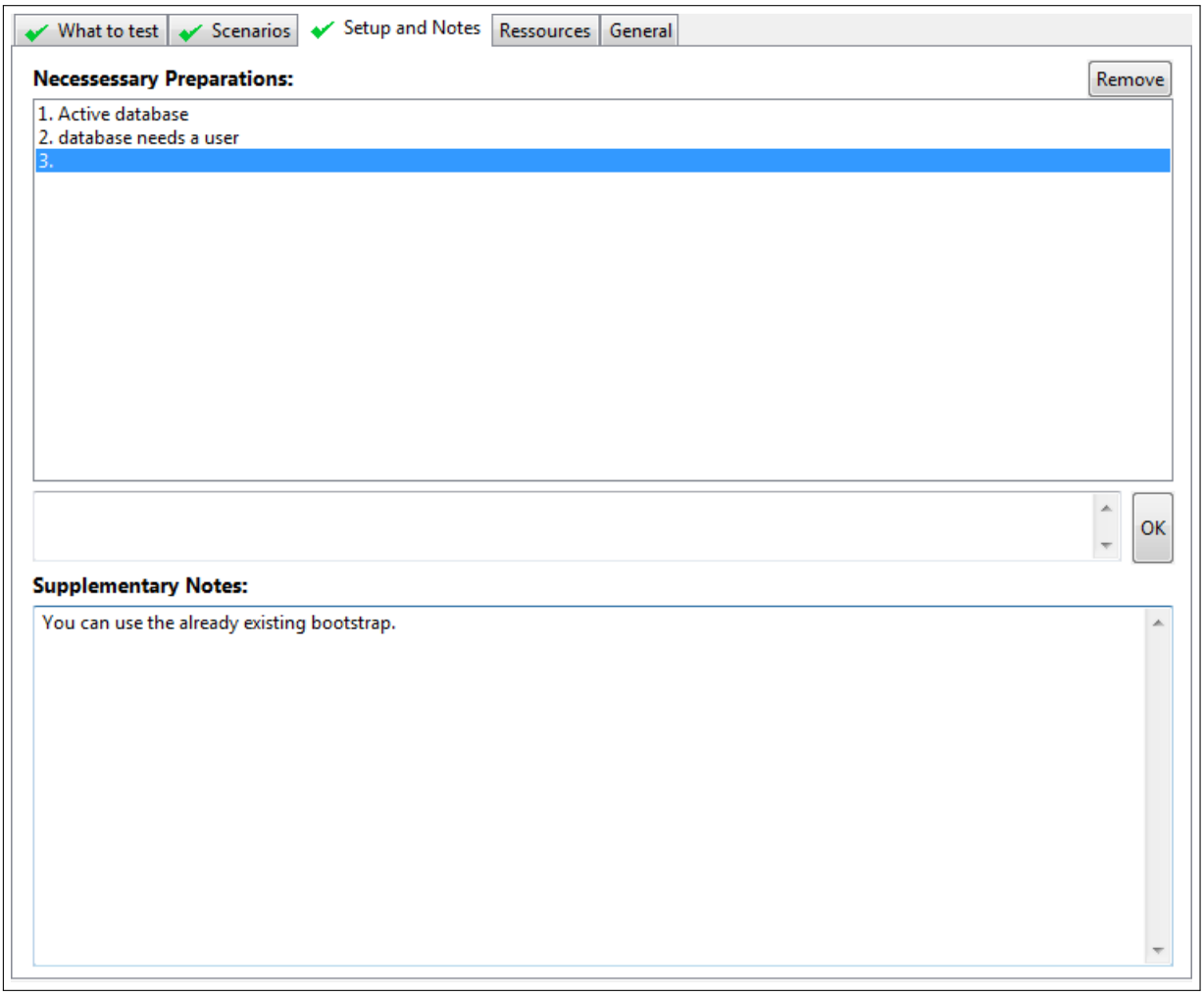

Abbildung 4.4.: Der Ist-Zustand des Reiters „Setup and Notes“

Der Reiter „Setup and Notes“ besteht aus zwei Teilen. Im oberen Teil werden Vorbereitungsschritte in einer Liste/Textbox/„OK“-Button-Anordnung, wie ich sie bereits beim vorherigen Reiter beschrieben habe, angegeben. Im unteren Teil können zusätzliche Anmerkungen als Freitext eingetragen werden.

Die Vorbereitungen müssen alle Informationen und Punkte enthalten, die für die erfolgreiche Ausführung des Normverhaltens im vorherigen Reiter nötig sind.

Das Anmerkungsfeld dient zum einen dazu, dem Nutzer eine Stelle zu geben, an der er alles loswerden kann, was er dem Tester noch mitteilen möchte, ohne dafür ein anderen Feld zweckentfremden zu müssen. Zum anderen können in diesem Feld auch Angaben für Testaufträge, die sich mit dem Schema aus Normalverhalten und Sonderfall nicht zufriedenstellend beschreiben lassen, gemacht werden. Ein Beispiel wäre ein explorativer Testauftrag, also ein Testauftrag der Art „Fällt noch jemandem ein Sonder-

fall ein, der noch nicht durch einen Test abgedeckt ist?“. Bei so einem Fall können das Wissen und vor allem die Erfahrung der Crowd sehr nützlich sein.

Reiter „Ressources“

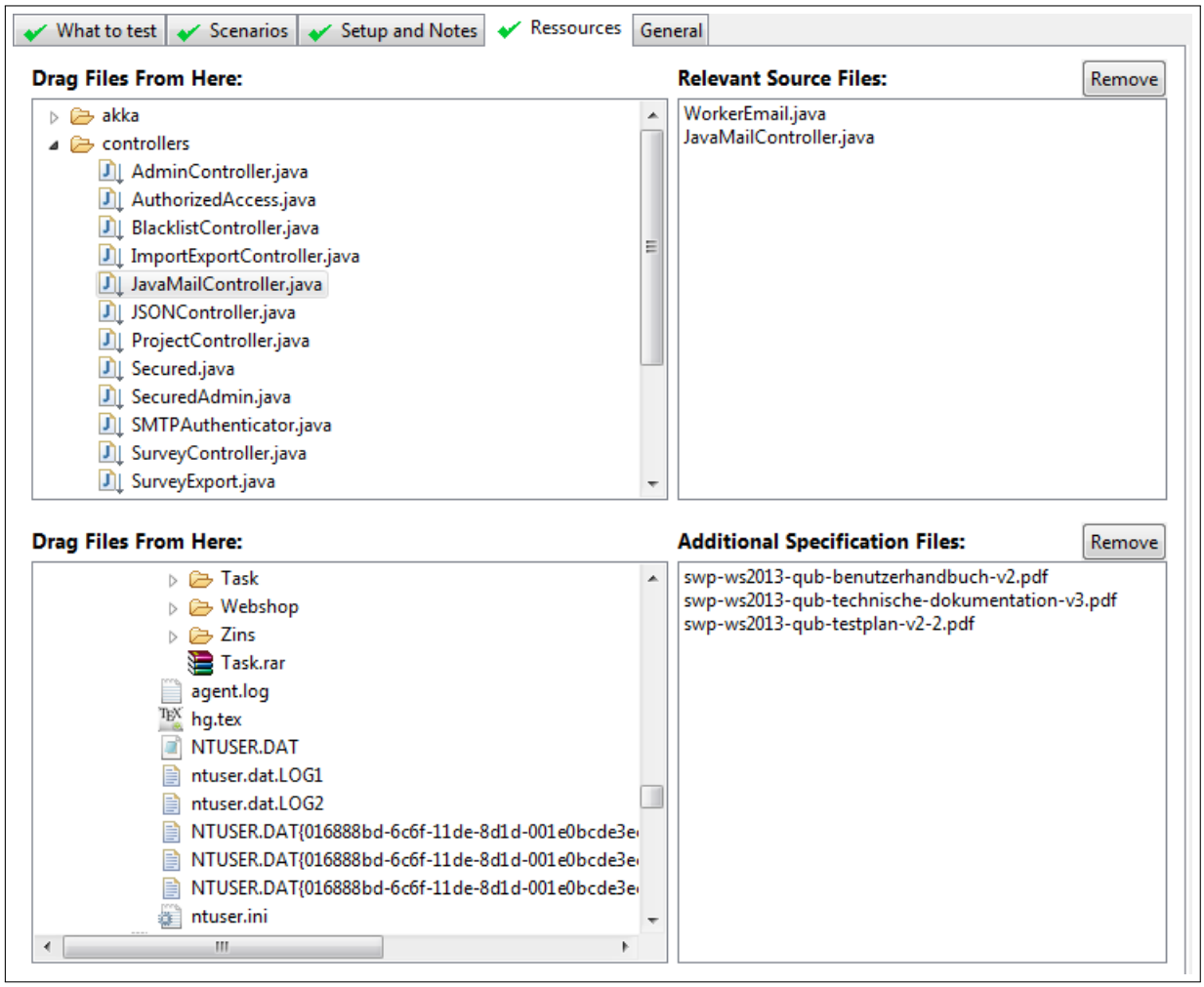

Abbildung 4.5.: Der Ist-Zustand des Reiters „Ressources“

In diesem Reiter können Quellcodedateien zur Liste der relevanten Dateien (oben, Abb. 4.5) und zusätzliche Dokumente, wie zum Beispiel UML-Diagramme oder Spezifikationsdokumente dem Testauftrag hinzugefügt werden (unten, Abb. 4.5).

Das Hinzufügen funktioniert jeweils per Drag und Drop aus einem Baum links von der entsprechenden Liste in die Liste rechts. Bei den relevanten Dateien beschränkt sich der Baum auf das Projekt. Dokumente können hingegen von überall im System hinzugefügt werden. Das Entfernen von Elementen funktioniert wie zuvor: durch „Remove“-Buttons oben rechts.

Reiter „General“

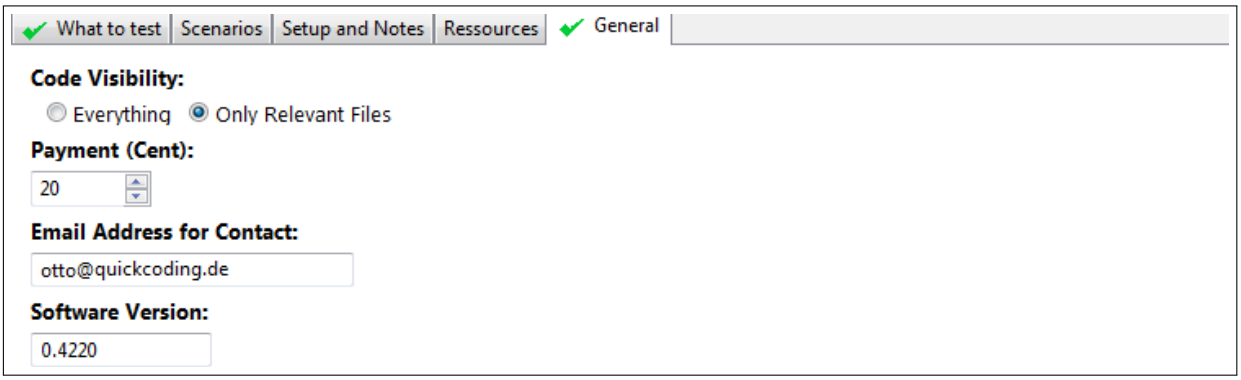

Abbildung 4.6.: Der Ist-Zustand des Reiters „General“

Im fünften Reiter befinden sich organisatorische Angaben. Es gibt Radiobuttons, um zu bestimmen, ob nur relevante Quelldateien unkompiliert mitgesendet werden sollen, einen Spinner, um das Honorar festzulegen, ein Feld für die E-Mail-Adresse des Entwicklers, unter der er für Rückfragen zur Verfügung steht sowie ein Feld, in die der Entwickler die Versionsnummer der AUT eintragen kann.

4.1.2. Steuerbuttons

Abbildung 4.7.: Die Steuerbuttons von TaskElf.

Im untersten Bereich befinden sich die Buttons zum Abbrechen der Erstellung und zum Veröffentlichen eines Testauftrags. „Cancel“ schließt das Plugin, „Publish“ erstellt einen neuen Testauftrag mit den vorhandenen Informationen und legt diesen im Projekt im Ordner /tmp ab[2]. Über Erfolg oder Misserfolg der Erstellung wird durch eine Dialogbox unterrichtet, um dem Nutzer zu bestätigen, dass der Vorgang abgeschlossen wurde.

Links daneben finden sich ein „Back“ und ein „Next“-Button, über die zwischen den Reitern navigiert werden kann. In der linken unteren Ecke gibt es einen „Show Example“--Button, der allerdings noch ohne Funktion ist (Sein Zweck wird in 4.2 erklärt).

[2]Da die genaue Ausgestaltung der Crowdsourcingfunktionalität von TestHub noch nicht feststeht und erst in einer späteren Arbeit behandelt werden soll, findet an dieser Stelle kein Upload statt.

4.2. Erweiterungen

Im Folgenden beschreibe ich, welche Erweiterungen ich mir für die Oberfläche noch vorstelle, die ich im Rahmen dieser Arbeit aber nicht mehr umsetzen konnte.

Testauftragsvorschläge

Die Testauftragswahl ist um eine Filterfunktion erweitert, die es ermöglicht, bestimmte Konzepte zu ignorieren oder obligatorisch zu machen. So können beispielsweise nur Testaufträge angezeigt werden, die neuen oder veränderten Code testen oder es kann auch das Alter des Codes für die Auswahl vollkommen außer Acht gelassen werden. Gesteuert wird dies über ein ausklappbares Feld an der linken Seite des Plugin-Fensters, in dem es für jede Heuristik, die Testaufträge vorschlägt, eine Dropdown-Liste gibt, mit der man festlegen kann, ob sie „activated“, „deactivated“ oder „required“ ist. Die Modi werden über Tooltips erklärt und bedeuten das Folgende:

- **activated:** Die Heuristik wird ganz normal für die Prognose der möglichen Testaufträge genutzt.
- **deactivated:** Die Heuristik wird für die Prognose der möglichen Testaufträge nicht genutzt.
- **required:** Es werden nur Testaufträge angezeigt, die auch von dieser Heuristik prognostiziert werden. Im Gegensatz zu „activated“ bedeutet dies, dass keine Testaufträge vorgeschlagen werden, die nicht auch von allen Heuristiken vorgeschlagen wurden, die „required“ sind. Dieser Unterschied ist vergleichbar mit dem zwischen einem logischen „Oder“ („activated“) und einem logischen „Und“ („required“).

Auf diese Weise kann gerade bei vielen Prognosen die Übersicht erhöht werden, außerdem bekommen erfahrene Nutzer die Möglichkeit die Auswahl nach ihren Wünschen anzupassen.

Zusätzlich bekommen Nutzer jetzt angezeigt, wieso ein Vorschlag in der Liste steht, wenn sie mit der Maus über ihn fahren. Im aufgehenden Tooltip stehen Informationen wie „zu 63% ungetestet; zuletzt bearbeitet; hohe Komplexität“. Dies gibt dem Nutzer ein Verständnis dafür, wie die Vorschläge zu Stande kommen.

Kontextmenü von Listenelementen

Bei Rechtsklick auf ein Listenelement in den Reitern „Scenarios“ und „Setup and Notes“ erscheint ein Kontextmenü, das die Auswahl gibt, vor oder nach dem Element, ein neues Element einzufügen. Außerdem wird das Löschen von Elementen (in allen Listen) in dieses Kontextmenü verschoben.

Speichern und Laden

Zusätzlich finden sich in diesem Bereich nun noch die Buttons „Save“ und „Load“, mit denen sich Testaufträge lokal speichern und wieder laden lassen, ohne sie ausschreiben zu müssen.

Wird das Fenster geschlossen, ohne dass ein Testauftrag erstellt wurde, so wird der gerade aktive Testauftrag gespeichert und beim späteren Öffnen des Plugins in der Testauftragsliste an hervorgehobener Position zur Auswahl gestellt. Dies ermöglicht dem Entwickler, die Erstellung im Bedarfsfall zu unterbrechen, ohne einen Informationsverlust zu erleiden.

Show Example

Der „Show Example“-Button lässt TaskElf nun bei Betätigung in einen Beispielmodus wechseln: Eine Ansicht, in der der Nutzer keine Formulareinträge ändern kann und alle Felder bereits mit einem, dem aktuellen Testtyp angepassten Beispiel, ausgefüllt sind. Ein weiterer Druck auf den Button (der nun mit „Hide Example“ beschriftet ist) bringt den Nutzer zurück in die Ansicht seines momentanen Testauftrags, die er vor dem Druck auf den Button hatte.

Funktionierende Veröffentlichung

Der Button „Publish“ öffnet nun ein weiteres Fenster, in das die Login-Informationen für die Crowdsourcing Plattform eingegeben werden müssen. Der Nutzername ist bereits vorausgefüllt, das Passwort muss der Nutzer aber von Hand eintragen. Zusätzlich ist in diesem Fenster eine Warnung zu sehen, dass mit Bestätigung des Testauftrags rechtsverbindlich ausgeschrieben wird. Wird dies bestätigt und ist das Passwort korrekt eingetragen, wird der Testauftrag erstellt. Während dieses Vorgangs informiert ein Fortschrifttsbalken darüber, wie der Gesamtfortschritt aussieht. Die Erstellung wird abschließend wieder bestätigt und es wird mitgeteilt, dass der Testauftrag nun im Hintergrund zur Crowdsourcing Plattform hochgeladen wird. Wird dies bestätigt, schließt sich das Plugin und der Upload beginnt. Gibt es Probleme mit der Verbindung, wird dies über einen Dialog mitgeteilt, zusammen mit der Information, dass der Upload nachgeholt wird, sobald die Crowdsourcing Plattform wieder erreicht werden kann. Ist der Upload vollständig, wird darüber ebenfalls informiert. Die lokale Kopie im /tmp-Ordner wird in diesem Fall gelöscht.

Ein Fortschrittsbalken ist sehr wichtig, da der Nutzer vermittelt bekommen muss, dass das Programm noch etwas tut. Der Vorgang des Erstellens des Testauftrags dauert einige Sekunden und ohne eine Rückmeldung ist für den Nutzer nicht ersichtlich, ob das Plugin noch arbeitet und die Wartezeit beabsichtigt ist oder, ob es sich um einen Fehler handelt.

Screencast-Funktion

Als Alternative ist es im Reiter „Scenarios“ nun möglich, die Informationen statt oder zusätzlich zu der textuellen Darstellung, in einem Screencast darzulegen. Dazu gibt es einen Button „Screencast“ am oberen Rand, der bei Betätigung die AUT startet, wenn GUI-Tests ausgewählt sind und bei Unit-Tests in Eclipse in die entsprechende Methode springt. Zusätzlich wird eine Aufnahme des aktuellen Bildschirms und des Mikrofons gestartet. Hier kann der Entwickler direkt anhand der AUT erläutern, was passieren und was getestet werden soll. Wenn der Entwickler die Erstellung des Screencasts abgeschlossen hat, kann der Nutzer ihn über einen Knopfdruck in einem seitlichen Menü beenden. In diesem Menü hat er ebenfalls die Möglichkeit, den Screencast zu pausieren. Nach dem Abschluss kann er sich den Screencast in TaskElf im Reiter „Scenarios“ ansehen und gegebenenfalls löschen oder neu erstellen.

Der Screencast ermöglicht es, komplexe Sachverhalte einfacher und klarer darzulegen, als dies in einem Text möglich wäre ([Cockburn2000]). Dafür erfordert er tendenziell mehr Zeit und ist für den Tester schwieriger strukturiert auszuwerten, da er keine feste Struktur aufweist. Zusätzlich gibt es hier sprachliche Hürden zu beachten. Auch wenn die Englischkenntnisse des Entwicklers ausreichen, um eine verständliche, textuelle Beschreibung abzugeben, bedeutet dies nicht unbedingt, dass auch sein gesprochenes, englisches Wort für den Tester verständlich ist. Zu diesem Problem kann sowohl der Entwickler beitragen, beispielsweise, wenn er einen starken Akzent hat oder nuschelt, als auch der Tester, zum Beispiel wenn sein Hörverstehen im Englischen nicht sehr ausgeprägt ist.

5. Heuristische Intentionserkennung

Die Intentionen eines Entwicklers können nicht zweifelsfrei algorithmisch bestimmt werden.[1] TaskElf kann nur eine Prognose seiner wirklichen Intentionen bestimmen und auch diese kann völlig falsch sein – ein wenig, wie bei der Vorhersage des Wetters in drei Wochen.

Im folgenden werde ich zuerst darstellen, was in diesem Bereich bereits an Arbeiten existiert, bevor ich dann meinen eigenen Ansatz darlege.

5.1. Verwandte Arbeiten

Es gibt einige Ansätze, den IDE-Kontext zu nutzen, um den Entwickler mit Hilfestellungen zu seiner Aufgabe zu versorgen. So wird in [Heinemann2011] ein Ansatz erläutert, um mithilfe der Bezeichnernamen, die ein Entwickler wählt, Vorschläge für mögliche, für ihn interessante, APIs zu machen. In [Rahman2013] wird der Kontext genutzt, um eine bessere Auswahl der Suchergebnisse zu Exceptions zu ermöglichen und in [Antunes2012] wird die allgemeine Relevanz von Codeabschnitten anhand des Kontextes eingeschätzt.

Einen anderen Ansatz verfolgt [Kersten2006]: Hier wird der Kontext zu einer Aufgabe gespeichert, um ihn im Falle einer Unterbrechung später, bei Fortsetzung der Aufgabe, wieder abrufen zu können.

All diesen Ansätzen ist gemein, dass sie den Kontext erfassen und modellieren müssen. Während [Rahman2013] und [Heinemann2011] sich dabei auf Datenbanken verlassen, die feste Teile des Kontextes auswerten (beispielsweise Bezeichnernamen), werten [Antunes2012] und [Kersten2006] den Kontext dynamisch, abhängig vom Nutzerverhalten, aus. Gerade das Vorgehen in [Antunes2012] ist interessant für TaskElf, da ein Modell vorgestellt wird, das dabei hilft, das Nutzerverhalten zu codieren und zu verwalten. Leider ist mir eine Umsetzung der vollständigen Aufnahme des Nutzerverhaltens allerdings nicht möglich, da es den zeitlichen Rahmen dieser Arbeit übertreffen würde und ich daher einzelne, abgeschlossene Heuristiken verwende. Es ist aber in jedem Fall als Erweiterung für die Zukunft denkbar.

Von der Nutzung von Datenbanken sehe ich in TaskElf ab (gerade auch weil die Erstellung dieser sehr aufwändig ist), allerdings finden derart vorgefertigte Informationen vereinzelt Eingang in die Heuristiken in Form von kontextunabhängigen Annahmen

[1] Ob es möglich wäre, die Intentionen sicher zu bestimmen, wenn man alle Einflüsse auf den Entwickler hätte, ist eine Frage mit der sich Philosophen beschäftigen. Da TaskElf allerdings niemals Zugriffe auf alle Informationen haben kann, ist die Antwort auf diese Frage hier nicht von Bedeutung.

(5.3).

5.2. Grundzüge meines Ansatzes

Eine Prognose der Intentionen des Entwicklers kann aus Heuristiken abgeleitet werden. Eine Heuristik ist ein Verfahren, das mit unvollständigem Wissen und geringem zeitlichen Aufwand zu guten Ergebnissen führt [Todd1999]. Das begrenzte Wissen, das zur Verfügung steht, unterteile ich nach der jeweiligen Herkunft in drei Klassen:

1. **Kontextunabhängige Annahmen**
2. **Nutzung des IDE-Kontextes**
3. **Beobachtung des Verhaltens des Nutzers**

Auf diese drei Klassen werde ich in diesem Kapitel näher eingehen. Das Wissen, das das Programm aus diesen Quellen gewinnt, muss bewertet und kombiniert werden (Abbildung 5.1), damit es in die Felder des Testauftrags hineinpasst. Dies geschieht in einem eigenen Verarbeitungsschritt, bevor die letztlich extrahierten Daten in das Formular, für die Erstellung des Testauftrags, fließen. Was genau in diesem Verarbeitungsschritt geschieht, wird ebenfalls in diesem Kapitel erläutert.

Außerdem werden in diesem Kapitel die verschiedenen Konzepte zur Intentionserkennung vorgestellt. Dies gliedere ich durch die Aufteilung der Konzepte in die zuvor erläuterten Klassen: Kontextunabhängige Annahmen, Beobachtung des Quellcodes und Beobachtung des Verhaltens des Nutzers. Davon abgesehen können die Konzepte auch danach unterteilt werden, bei welchem Problem sie dem Nutzer helfen. Dabei fallen folgende Probleme in der Auftragserstellung an:

1. **Dem System mitteilen, welche Methode oder View getestet werden soll:** Für einige Felder des Testaufauftrags muss TaskElf die Information besitzen, wofür genau der Test vorgesehen ist und wie getestet werden soll. Die Unterscheidung, beispielsweise, welche Art von Test geschrieben werden soll, hat auch Auswirkungen auf das vorgeschlagene Beispiel. Daher muss der Nutzer dies dem Plugin irgendwie mitteilen. TaskElf bietet dem Nutzer beim Öffnen des Plugins eine Liste vorgeschlagener Tests, die ihn unterstützen soll, dem Plugin dieses „Wofür und Wie?“ zu beantworten.
2. **In den Testauftrag eintragen was getestet werden soll:** Der Tester muss wissen, was genau er an einer Methode oder View testen soll. Das bedeutet, dass der Entwickler (also der Nutzer von TaskElf) dieses „Was?“ irgendwie im Testauftrag festhalten muss. Dazu gehört unter anderem das Ausfüllen der Felder „Workflow Description“ und „Special Case To Test“.
3. **Dem Tester seine Aufgabe erleichtern:** Je leichter ein Testauftrag für einen Tester zu bearbeiten ist, desto eher wird er von einem Tester übernommen und desto schneller ist er von diesem bearbeitet. Daher liegt es im Interesse des Entwicklers, dem Tester seine Aufgabe möglichst einfach zu machen und sich zu

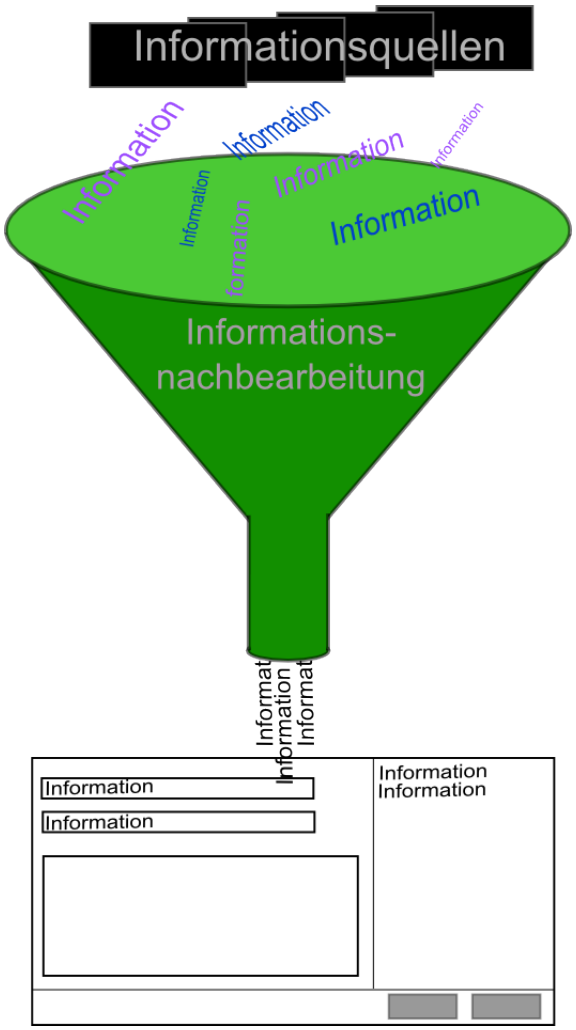

Abbildung 5.1.: Der Weg der Informationen

fragen, welche Hilfsmittel er ihm zur Verfügung stellen kann. Welche Hilfsmittel prinzipiell unterstützt werden und – wenn möglich – angegeben werden sollten, ist bereits durch die, mit der Testerseite ausgehandelten, Struktur des Testauftrags vorgegeben. Was genau in den entsprechenden Feldern sinnvoll ist, ist dennoch eine Frage, die der Entwickler zu beantworten hat. Ein Beispiel wäre die Liste der Spezifikationsdokumente: Es ist dem Entwickler vorgegeben, dass zusätzliche Dokumente sinnvoll sein können. Ob dies im konkreten Testauftrag der Fall ist, und wenn ja, welches Dokument oder welche Dokumente explizit für diesen Testauftrag sinnvoll sind, muss er selbst entscheiden.

4. **Dem System alle organisatorischen Informationen mitteilen:** Es gibt einen gewissen Overhead, der durch die Nutzung der Crowdsourcing Plattform entsteht. Dies drückt sich in der Notwendigkeit von organisatorischen Angaben aus, die mit dem eigentlichen Testen nichts zu tun haben. Beispiele sind die Anga-

be der Honorarhöhe oder die Angabe, ob der Tester den gesamten Quellcode einsehen darf.

5. **Es dem Tester möglich machen das Programm auszuführen:** Der Tester sollte das Programm ausführen können, um es zu testen. Dazu muss er neben dem (kompilierten) Programmcode auch alle nötigen Bibliotheken haben und wissen, wie genau das Programm gestartet werden kann. Dieses Problem sei nur der Vollständigkeit halber erwähnt, da es im Gegensatz zu den anderen Problemen keines ist, bei dem TaskElf den Entwickler unterstützt. Stattdessen wird ihm dieses Problem vollständig von TaskElf abgenommen und es läuft im Hintergrund ohne Zutun des Nutzers ab.

Die Unterstützung bei den Punkten zwei, drei und vier betrifft lediglich das (Vor)Ausfüllen der vom Editor vorgegebenen Felder, während es bei der Behandlung des ersten Problems zusätzlich darum geht, eine Liste von Vorschlägen für neue Testfälle zu erstellen.

5.3. Kontextunabhängige Annahmen

Die erste Klasse bilden kontextunabhängige Annahmen. Diese umfassen all solches Wissen, welches nicht vom Kontext abhängt und nicht zur Laufzeit generiert wird, sondern das bereits zum Zeitpunkt der Implementierung vorliegt. Ein Beispiel für eine solche Annahme wäre:

HTML-Dokumente werden mit GUI-Tests getestet.

Die Annahmen können dabei prinzipiell von Allgemeinwissen bis hin zum Ergebnis komplexer Studien unterschiedlich trivial oder erstaunlich sein. Kontextunabhängige Annahmen bilden, neben direkt anwendbaren Konzepten, auch die Grundlage für alle anderen hier besprochenen Konzepte.

Folgende Konzepte fallen in den Bereich der kontextunabhängigen Annahmen:

- **Dateiendungen:** Hier ist die Annahme, dass die Endung einer Datei Rückschlüsse auf den Testtyp zulässt, mit dem ihr Inhalt getestet werden kann. Auf das play!-Framework bezogen heißt das, dass ein html-Dokument durch einen GUI-Test und ein java-Dokument durch einen Unit-Test getestet wird. Dies kann als Richtwert gesehen werden, auch wenn ein GUI-Test ebenfalls die einer Oberfläche zu Grunde liegende Logik testen kann. TaskElf nutzt dies zum Beispiel bei „Current File“, welches ich im folgenden Abschnitt erläutere.
- **Documents in Project:** Die Annahme ist hier, dass Dokumente wie pdf- oder Worddateien, die im Projekt liegen, Anforderungsdokumente sind. Als Konzept kann man diese automatisch zu den Spezifikationsdokumenten hinzufügen. Wenn dem Nutzer dies mitgeteilt wird, kann er sich auch darauf einrichten und entsprechende Dokumente dort ablegen. Allerdings sollte TaskElf sich idealerweise

auch merken, welche der automatisch hinzugefügten Dokumente vom Entwickler wieder aus der Liste entfernt wurden, um diese in Zukunft nicht mehr automatisch hinzuzufügen. Gegebenenfalls könnte es besser sein, ein Dokument erst nach zwei oder dreimaligem manuellen Entfernen zu ignorieren, da es immer sein kann, dass einzelne Dokumente nur für einen einzelnen Test nicht relevant sind.

5.4. Nutzung des IDE-Kontextes

Eine weitere Informationsquelle ist der IDE-Kontext. Die IDE gibt Zugriff auf den Quellcode, in dem sich sowohl Veränderungen, als auch Besonderheiten erkennen und Schlüsse aus diesen ziehen lassen. Darüber hinaus ermöglicht die IDE es aber auch, auf Plugins und Funktionen zuzugreifen, die hilfreiche Informationen liefern können. Ein Beispiel wäre das Auslesen der Änderungshistorie von Eclipse.

Ich schlage die folgenden Konzepte vor:

- **Current File:** Die Grundannahme dieses Konzeptes beruht auf der Annahme, dass der Entwickler mit einer erhöhten Wahrscheinlichkeit die gerade im aktiven Editor geöffnete Datei testen möchte. Dies rührt aus der Einschätzung her, dass ein Entwickler sich oft vor der Erstellung eines Tests die betreffende Klasse noch einmal ansehen wird oder einen Test für etwas möchte, an dem er gerade gearbeitet hat. Auch in letzterem Fall ist die Wahrscheinlichkeit, dass die zu testende Datei aktuell aktiv ist, erhöht. Dies lässt sich nutzen, indem die aktuell offene Datei als Indiz für die zu testende Klasse verarbeitet wird. Als Indiz deshalb, weil der Entwickler ebenfalls eine andere Datei geöffnet haben könnte. Der Anteil der Testaufträge, die der Entwickler erstellt, ohne die Zieldatei im aktuell fokussierten Editor offen zu haben, ist der Fehler dieser Heuristik.

  Obwohl durch die letzte Position des Eingabecursors innerhalb der Datei theoretisch auch ein Hinweis auf eine möglicherweise zu testende Methode möglich ist, verwendet TaskElf nur den Hinweis auf die Klasse. Da in der Methodenebene ein weiterer Fehler hinzukommt (der Entwickler könnte die richtige Datei offen haben, der Cursor sich aber zuletzt in der falschen Methode befunden haben), der sich mit dem Fehler auf Klassenebene multipliziert, wäre die Deutung meiner Meinung nach, zu ungenau, um sie nützlich einsetzen zu können. Ob diese Einschätzung korrekt ist und wie hoch die Fehler der Ebenen genau sind, müsste durch einen Versuch ermittelt werden.

- **Untested First:** Bei „Untested First“ geht es darum, die Testabdeckung des Codes zu betrachten. Diesem Konzept liegt die Annahme zu Grunde, dass die Wahrscheinlichkeit in einem neuen Testauftrag berücksichtigt zu werden, für solchen Code erhöht ist, der bisher noch von keinem Test abgedeckt ist. Die Testabdeckung kann mit Hilfe von Plugins wie Cobertura[2] bestimmt werden. Dabei ist es wichtig, ein Plugin zu wählen, das über einen API Zugriff auf die Testabde-

[2] http://cobertura.github.io/cobertura/

ckung bietet. Liegen die Daten vor, kann jede Methode zusätzlich zum Gewicht des Konzeptes noch mit ihrer umgekehrten Testabdeckung gewichtet werden. Umgekehrt deshalb, weil eine höhere Testabdeckung ein geringeres Gewicht nach sich ziehen muss. Die Formel für das Gewicht einer Information, sieht dann so aus:

$$Gewicht_{gesamt} = Gewicht_{Informationsquelle} * (1 - \text{Testabdeckung}_{Methode})$$

wobei

$$Testabdeckung_{Methode} = \frac{AnzahlGetesterZeilen_{Methode}}{AnzahlZeilen_{Methode}}$$

Das volle Gewicht der Informationsquelle wird so nur denjenigen Informationen zugeordnet, die Methoden zum Inhalt haben, für die es keinerlei automatisierte Tests gibt (die Testabdeckung also bisher null ist).

- **Change Observation:** „Change Observation“ sieht vor, neuen Code als zu testenden Code zu prognostizieren. Dieses Konzept profitiert besonders von einem Entwickler, der für neu geschriebenen Code direkt Tests ausschreibt. Ein Entwickler, der Tests erst am Ende der Entwicklung eines größeren Abschnitts ausschreibt, wäre den Ergebnissen von Change Observation hingegen abträglich. Um Veränderungen wahrnehmen zu können, muss man einen Zeitpunkt wählen, demgegenüber die Veränderung gemessen wird sowie den Zustand der Dateien zu diesem Zeitpunkt kennen. Ein Weg dies zu erreichen, ist die Nutzung des Versionskontrollsystems, das von Haus aus in der Lage ist, die Unterschiede seit dem letzten Commit anzuzeigen. Diese Daten können genutzt werden, um, analog zu Untested First, für jede Methode (und jedes HTML-Dokument) einen Wert zwischen 0 und 1 zu berechnen, wobei 0 keine Änderung und 1 eine völlig neue Methode oder Datei bedeutet. Das Gesamtgewicht einer Information ist dann das Produkt aus diesem Wert und dem Gewicht der Informationsquelle:

$$Gewicht_{gesamt} = Gewicht_{Informationsquelle} * AnteilVeränderterZeilenMethode)$$

wobei

$$AnteilVeränderterZeilen_{Methode} = \frac{AnzahlZeilenMitÄnderungen_{Methode} + AnzahlNeueZeilen_{Methode}}{AnzahlZeilen_{Methode}}$$

- **Example From Test:** Neben den Standardbeispielen kann es auch sein, dass manche, der bereits im Projekt existenten Tests, besonders hilfreich für den Tester wären. Bei „Example From Test“ geht es darum, rauszufinden, welche Tests besonders nah am ausgeschriebenen Test sind und diese Information dem Testauftrag hinzuzufügen. Als Metrik könnte zum Beispiel die Anzahl der identischen Felder der Testaufträge zweier Tests verwendet werden. Allerdings würde dies nur bei Tests funktionieren, die ebenfalls durch Testaufträge entstanden sind und es müssten alle vergangenen Testaufträge archiviert und mit ihren Tests verknüpft werden. Außerdem sollte solch eine Metrik nicht ausnahmslos alle Felder vergleichen, sondern nur solche, die auch etwas über den Inhalt aussagen. Eine geänderte E-Mail-Adresse zum Beispiel, sagt nichts über die Ähnlichkeit des zu

schreibenden Tests aus. Ein anderer Weg wäre es, nur die relevanten Dateien zu prüfen und festzustellen, welche Tests wie viele dieser Dateien nutzen.

Beide Wege könnten allerdings auch zu einer hohen Ähnlichkeit bei Tests führen, die technisch nicht viel gemein haben. Solche False Positives unter den ähnlichen Tests bergen die Gefahr, den Tester mehr zu verwirren, wenn sie ihm als besonders nahes Beispiel dargebracht werden. Insgesamt lässt sich sagen, dass die Bestimmung von ähnlichen Tests schwierig ist.

- **Comment Scan:** „Comment Scan“ liegen zwei Annahmen zu Grunde: Erstens, dass besonders lange Kommentarblöcke in einer Funktion in der Regel vor komplizierten Codestücken stehen und zweitens, dass insbesondere komplizierte Codestücke ausgiebig getestet werden müssen. Das Konzept besagt nun, dass Methoden, die besonders viele solcher Kommentarblöcke enthalten, wahrscheinliche Testkandidaten sind. Ein Kommentarblock besteht dabei aus mehreren Zeilen Codekommentaren direkt hintereinander innerhalb einer Methode. Außen vor gelassen werden Javadoc-Kommentare, die eine Schnittstellenbeschreibung darstellen. Eine ausgiebig beschriebene Schnittstelle bedeutet nicht, dass der Code hinter der Schnittstelle kompliziert ist.

  Zur Berechnung des Gewichts sollte die Anzahl der Codekommentare als Faktor auf das Informationsquellengewicht einwirken, da jeder einzelner Kommentarblock ein unabhängiges Indiz für die Methode ist. Äquivalent wäre es, für jeden entdeckten Kommentarblock eine eigene Information zu erstellen, die dann später in der Nachbearbeitung mit den anderen fusioniert würde (da sie sich ja auf die selbe Methode bezögen). Ergebnis dieser Fusion wäre eine einzelne Information mit der Summe der Informationen der einzelnen, vorherigen Informationen.

- **Behavior From Javadoc:** Die zuvor ignorierten Javadoc-Kommentare können von „Behavior From Javadoc“ genutzt werden. Ein ausführlicher Javadoc-Kommentar für eine Methode enthält bereits eine Erklärung, was sie tut und wie sie zu benutzen ist; inklusive der Eingabe- und Rückgabewerte. Dies sind alle wesentlichen Punkte, die in die Beschreibung des Normalverhaltens, im gleichnamigen Feld des Testauftrags, einfließen sollten. Wenn also noch kein Normalverhalten für eine Methode vorliegt (weil für die Methode bisher noch kein Testauftrag geschrieben wurde), so kann gegebenenfalls die Javadoc der Methode in das Feld voreingetragen werden.

- **Relevance Neighbors:** Bei „Relevance Neighbors“ geht es darum, die Liste der relevanten Quellcodedateien aus der zentralen Datei, die getestet werden soll, abzuleiten. Hierzu kann ein Nachbarschaftsgraph für diese Datei aufgebaut werden. Jede Projektklasse oder Ansicht, die die zentrale Datei aufruft oder von ihr aufgerufen wird, kann als Nachbarschaftsknoten hinzugefügt werden. Wie genau dieser Ansatz in TaskElf verfolgt wird, behandle ich in 6.

- **Specification Scanner:** Bei diesem Konzept geht es darum, in den angegebenen Spezifikationsdokumenten nach für den Testauftrag relevanten Passagen zu suchen. Dazu muss TaskElf in der Lage sein, die relevanten Formate zu lesen, in

denen Spezifikationsdokumente angehängt werden können. Das Vorgehen sieht dann wie folgt aus:

1. **Extraktion relevanter Schlagwörter:** Zuerst müssen für den Testauftrag relevante Begriffe bestimmt werden. Diese können aus den verschiedenen Feldern des Testauftrags extrahiert werden. So sind zum Beispiel der Name der zu testenden Methode und Klasse relevant. Erstrebenswert wäre es auch, eine Möglichkeit zu finden, relevante Begriffe aus der Testbeschreibung zu erhalten. Dabei ist es schwierig, relevante Begriffe von Begriffen zu trennen, die einfach nur der Beschreibung dienen, wie zum Beispiel „Ergebnis“ oder „Methode“. Ein erster Ansatz dieses Problem zu lösen, wäre eine Blacklist mit den n häufigsten Begriffen, die in einer Beispielmenge von Testbeschreibungen vorkommen.

   Eine weitere Quelle für die Schlagwörter könnten ggf. vorhandene Javadoc Kommentare sein.

   Die Quellen sollte man gewichten, da zum Beispiel der Name der zu testenden Methode wichtiger und zentraler ist, als der einer mit beliebigen relevanten Klasse. Statt der einzelnen Schlagwörter würde man also ein Tupel aus Schlagwort und Gewicht generieren.

2. **Vereinigung der Schlagwörter:** Gleiche und ähnliche Schlagwörter sollten gruppiert werden und die Gruppe das maximale Gewicht aus der Menge der Ursprungsschlagwörter erhalten.

3. **Suche in den Dokumenten:** Als nächstes sollten die Dokumente mithilfe einer unscharfen Suche[3] durchsucht werden, um die Schlagwörter oder ihnen ähnliche Wörter zu finden.

4. **Suche nach Abschnitten mit hohem Gesamtgewicht:** Zum Schluss werden diejenigen Abschnitte eines Dokuments gesucht, bei denen die Summe der Gewichte aller Treffer aus Schritt 4, einen vordefinierten Schwellwert überschreitet. Dies sind die letztlichen Treffer, die in den Testauftrag aufgenommen werden können.

Dieses Vorgehen ist fehleranfällig und gerade durch die Suche nach ähnlichen Begriffen bei größeren Dokumenten unter Umständen auch recht langsam. Der vorgeschlagene Algorithmus ist nur eine erste Annäherung an das Problem des automatischen Auffindens von relevanten Textstellen auf Grundlage eines Testauftrags und würde, für sich genommen, genug Material für eine eigene Arbeit geben.

[3] Bei einer unscharfen Suche handelt es sich um eine Textsuche, bei der nicht der exakte Suchbegriff gefunden werden muss, um einen Treffer zu erzielen, sondern auch ähnliche Begriffe gewertet werden.

5.4.1. Statische Quellcodebeobachtung

Informationen direkt aus dem Quellcode zu gewinnen, ist technisch vergleichsweise einfach, da sie bereits in parsbarer Form vorliegen und TaskElf über die IDE direkten Zugriff auf den Quellcode hat. Ein Beispiel für die statische Quellcodebeobachtung ist die Betrachtung von Kommentaren. Komplizierter Code ist im Allgemeinen stärker kommentiert als unkomplizierter und erfordert gleichzeitig auch mehr Tests, um seine korrekte Funktion sicherzustellen.

5.4.2. Beobachtung von Quellcodeänderungen

Veränderungen sind schwieriger zu beobachten. Dies liegt daran, dass neben dem bereits vorliegenden, aktuellen Quellcode, auch eine ältere Version benötigt wird, die als Vergleichsbasis dient, um Änderungen zu erkennen. Dabei stellt sich unweigerlich die Frage, von wann diese Vergleichsbasis stammen soll. Möchte man Änderungen der letzten Stunde oder der letzten Woche? Möchte man Änderungen seit dem letzten erstellten Testauftrag oder seit dem letzten Commit? Um diese Ressource vollständig ausbeuten zu können, würde man eine vollständige Historie des Programmiervorgangs benötigen. Allerdings kann auch die weniger aufwändige Ermittlung der Änderungen seit einem festen Zeitpunkt bereits hilfreiche Informationen zu Tage fördern. So kann man in Eclipse beispielsweise das vom Entwickler genutzte Version Control System nutzen, um die Änderungen seit dem letzten Commit zu extrahieren.

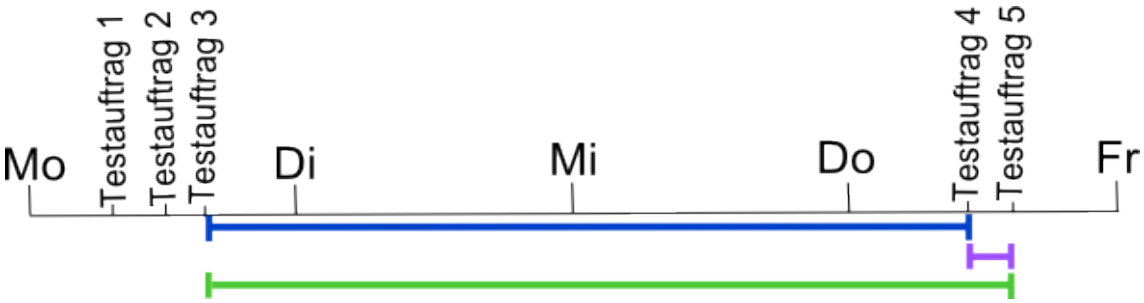

Abbildung 5.2.: Zeiträume, in denen Änderungen erfasst werden (Blau: Erfasste Änderungen für Testauftrag 4; Rot: Erfasste Änderungen für Testauftrag 5, bei Aufzeichnung der Änderungen seit letztem Testauftrag; Grün: Erfasste Änderungen für Testauftrag 5, bei Aufzeichnung der Änderungen seit der letzten Erstellungsphase)

Die Nutzung des letzten Commits als Zeitpunkt hat allerdings Nachteile: Es kann vorkommen, dass der Entwickler zuerst committet und erst im Anschluss Testaufträge für den neuen Code schreibt.[4]. Besser wäre es Änderungen, seit dem letzten Ausschreiben von Tests zu messen. Dabei ist wichtig, nicht einzelne Ausschreibungen

[4] Das Committen von Code, zu dem noch keine Tests existieren, ist zwar prinzipiell kritisch zu sehen, aber beim Outsourcing von Tests nicht immer zu vermeiden, da nicht beliebig lange darauf gewartet werden kann, dass Tests für den neuen Code geschrieben sind.

zu betrachten, sondern Blöcke. Dies wird deutlich, wenn man ein Beispiel betrachtet (5.4.2):

> Es ist Donnerstag Nachmittag. Otto hat gerade eine neue Funktion fertiggestellt und möchte für diese Testaufträge schreiben. Zuletzt hat er am Montag Testaufträge ausgeschrieben. Beim ersten Testauftrag, den er heute erstellt (Testauftrag 4), betrachtet TaskElf die Änderungen, die es seit Montag im Code gab (blauer Zeitraum) und findet direkt die Stellen, an denen Otto seitdem gearbeitet hat. Nun möchte Otto noch einen weiteren Testauftrag für diese Funktion ausschreiben (Testauftrag 5). Wenn TaskElf nun lediglich die Änderungen seit dem letzten Testauftrag ausliest, würde es keinerlei Änderungen finden, da der letzte Testauftrag nur wenige Minuten zurückliegt (roter Zeitraum). Betrachtet TaskElf allerdings die Änderungen seit der letzten Erstellungsphase, werden die Änderungen erkannt (grüner Zeitraum).

Um zu gewährleisten, dass der gesamte relevante Zeitraum betrachtet wird, müssen die Testauftragserstellungen geclustert werden. Der einfachste Weg wäre es, einen festen Schwellzeitraum festzulegen, ab dem zwei aufeinander folgende Testaufträge nicht mehr dem selben Block zugeordnet werden.

Für die, von Versionskontrollsystemen unabhängige, Änderungsbetrachtung, muss TaskElf die alten Daten allerdings selbst vorhalten, was den Speicherplatzbedarf des Projektes erhöht. Zum Feststellen der Unterschiede zwischen den Dateien, kann in diesem Szenario zum Beispiel die „Compare With“ Funktion von Eclipse genutzt werden.

5.5. Beobachtung des Nutzerverhaltens

Eine weitere Informationsquelle ist das Verhalten des Entwicklers selbst. Durch die Überwachung seiner Nutzung von TaskElf kann entweder eine Datenbasis erstellt werden, die genutzt werden kann, um zukünftiges Verhalten vorherzusagen, oder Nutzerhandlungen können direkt in Gewichte einfließen und diese verändern. Während bei ersterer Variante die gewonnenen Informationen flexibler genutzt werden können, um zukünftiges Verhalten vorauszusagen, muss bei letzterer Variante keine Historie geführt werden. Die Gewichte helfen bei der Abwägung zwischen den Ergebnissen verschiedener Heuristiken. Wenn ein Konzept in der Vergangenheit bessere Ergebnisse erzielt hat als ein anderes, dann sollten im Zweifel die Ergebnisse des erfolgreicheren Konzeptes bevorzugt werden.

Mit diesem System kann TaskElf zwei Dinge berücksichtigen: Zum einen die persönlichen Präferenzen des Nutzers: Bei einem Entwickler zum Beispiel, der sehr ausgiebig kommentiert, sodass auch viele einfache Codestellen mit größeren Kommentaren versehen sind, wird „Comment Scan“ schlechtere Ergebnisse liefern, als bei einem Entwickler, der nur komplizierte Codestellen mit größeren Inlinekommentarblöcken versieht. Der zweite Aspekt, der mit einem solchen System berücksichtigt werden kann, ist die Tatsache, dass verschiedene Heuristiken unterschiedlich gut sein in ihren Vorhersagen können. Auf die Gewichtung der Informationsquellen werde ich im nächsten

Abschnitt noch näher eingehen.

Ein solches selbst-optimierendes System hat den Vorteil, dass es mit der Zeit besser auf einen einzelnen Nutzer zugeschnitten sein wird als ein starres System. Dies resultiert daraus, dass starre Systeme eine gute Lösung für möglichst viele Benutzer bieten und damit die kleinste Schnittmenge der Präferenzen verschiedener Nutzertypen implementieren müssen. Natürlich ist es auch möglich, ein eigentlich starres System durch Konfiguration auf einen Nutzer zuzuschneiden. Allerdings hat diese Herangehensweise, neben dem zusätzlichen Aufwand für den Nutzer (er muss die Konfiguration auf sich einstellen), den Nachteil, dass der Nutzer viele seiner Präferenzen vielleicht selbst nicht im Detail bewusst kennt oder sie sich im Laufe der Zeit ändern, sodass die Gefahr einer falschen Konfiguration besteht. Die regelmäßige Anpassung der Konfiguration kann vom Nutzer schnell als störend empfunden werden und könnte in der Folge vernachlässigt werden. Dies würde die Performance der Software und damit des Entwicklers senken. Ein Beispiel für eine solche Änderung der Präferenzen wäre es, wenn ein Entwickler anfänglich nur GUI-Tests auslagert, aber im Laufe der Zeit auch immer mehr dazu übergeht, Unit-Tests auszulagern:

Szenario 1 – Anpassbare Konfiguration:

> Otto hat gerade TaskElf installiert. Er startet das Programm und es öffnet sich eine Konfiguration. Mit Schiebereglern soll er festlegen, wie wichtig ihm die Auslagerung einzelner Testtypen ist. Da er nur GUI-Tests auslagern möchte, schiebt er den Regler für GUI-Tests auf 1 und die anderen Regler auf 0. Dies hat zur Folge, dass Otto nur GUI-Tests vorgeschlagen werden. Mit der Zeit hat Otto sich sehr an das Auslagern von Testfällen gewöhnt und beginnt auch Unit-Tests auszulagern. Durch seine Konfigurationseinstellungen werden Otto von TaskElf keine Unit-Tests für Testaufträge vorgeschlagen und er muss diese jedes Mal vollständig selbst erstellen. An die Angaben, die er Monate zuvor in der Konfiguration vorgenommen hat, erinnert sich Otto nicht mehr. Soweit er sich ersinnt, hat TaskElf immer nur GUI-Tests vorgeschlagen. So kommt Otto auch nicht auf die Idee die Konfiguration zu ändern.

Szenario 2 – Selbstoptimierendes System:

> Otto hat gerade TaskElf installiert. Er muss nichts konfigurieren und das Programm ist direkt betriebsbereit. Otto schreibt nur Testaufträge für GUI-Tests. Mit jedem neu erstellten GUI-Test werden Tests dieses Typs in der Liste der vorgeschlagenen Testaufträge immer höher eingeordnet. Im Laufe der Zeit hat Otto sich sehr an das Auslagern von Testfällen gewöhnt und beginnt auch Unit-Tests auszulagern. Während zu Beginn seiner Verhaltensänderung Unit-Tests in der Vorschlagsliste immer nur ganz unten auftauchten, sind sie jetzt zunehmend auch weiter oben zu finden. Otto nimmt diese Veränderung des Verhaltens kaum zur Kenntnis, da sie sehr langsam von statten geht.

Dass genau dieses Szenario eintritt, mag unwahrscheinlich wirken, aber das Prinzip ist für alle anderen Präferenzänderungen dasselbe.

Auch eine Kombination dieser beiden Varianten ist möglich. Eine Ausgangskonfiguration, die dann im Laufe der Zeit vom System optimiert wird. Ein solches System verbindet den Vorteil des ersten Szenarios mit dem Vorteil des zweiten: Das System ist bereits zu Beginn personalisiert und bleibt durch die automatische Selbst-Optimierung

auch personalisiert. Selbst dann, wenn der Nutzer mit der Zeit seine Präferenzen ändert.

Das System sollte sich bei Selbstkonfiguration nicht zu stark oder zu plötzlich verändern. Beides kann zu vom Nutzer unerwartetem Verhalten führen und ihm das Gefühl nehmen, die Kontrolle über die Software zu haben, was die Akzeptanz und damit die Benutzbarkeit der Software senkt [Shneiderman2010]. Der vielleicht größte Nachteil eines personalisierten Systems ist, dass es nur für den Nutzer richtig funktioniert, auf den es personalisiert ist. An einem Arbeitsplatz mit wechselnden Nutzern ist Personalisierung sogar nachteilhaft, da das System sich für identische Angaben je nach Konfiguration unterschiedlich verhalten kann. Um diesen Nachteil zu umgehen, sollte Personalisierung immer deaktivierbar sein. In der prototypischen Implementierung im Rahmen dieser Arbeit gehe ich davon aus, dass das System immer nur von der Person genutzt wird, die es auch einrichtet.

Automatische Personalisierung, wie hier beschrieben, wird in TaskElf nur genutzt, um die Gewichte der verschiedenen Informationsquellen zu verändern („Flexible Weights“) und eine Verhaltensänderung tritt nur in der Reihenfolge der verschiedenen vorausgefüllten Testaufträge und den vorausgefüllten Informationen ein. Die genaue Herkunft und Sortierung der Informationen ist für den Nutzer nicht ersichtlich, was grundsätzlich ein Benutzbarkeitsproblem ist, da der Nutzer nicht völlig versteht, was das Plugin macht. Es wäre zu untersuchen, inwieweit ein Nutzer das Gefühl hat, nicht die Kontrolle über das Verhalten von TaskElf zu haben. Ein Vorteil davon, dass der Nutzer nicht weiß, wie die Vorausfüllungen zustande kommen, ist aber, dass kleine, durch die Parameteränderungen hervorgerufene, Änderungen dem Nutzer kaum auffallen dürften.

Folgende Heuristiken berücksichtigen das Nutzerverhalten:

- **Last Entry:** Bei einigen Feldern des Testauftrags kann man davon ausgehen, dass sie nicht immer auftragsspezifisch sind. Diese können mit dem letzten Eintrag vorausgefüllt werden, wie es bei Formularen generell sehr verbreitet ist. Beispielsweise wird sich die E-Mail-Adresse nicht mit jedem neuen Testauftrag ändern. Dies ist ein sehr schlichtes Konzept, aber bei derartigen Feldern dennoch sehr effektiv.
- **Flexible Weights:** „Flexible Weights“ greift den Gedanken der Personalisierung wieder auf. Anhand der Auswahl des Nutzers kann mit der Erstellung eines Testauftrags eine Aussage darüber getroffen werden, wie gut eine Vorhersage und damit die entsprechende, durch Gewichte vorgegebene, Entscheidung war. Wenn der Nutzer eine Vorhersage nutzt und nicht ändert, dann kann angenommen werden, dass diese zutreffend war. Eine Vorhersage, die der Nutzer ändert, kann entweder unvollständig oder falsch sein. Diese Information kann erfasst und genutzt werden. TaskElf hebt die Gewichte der Quellen aller korrekten Vorhersagen um einen festgelegten Prozentwert an und senkt die Gewichte der Quellen aller falschen Vorhersagen um einen Prozentwert ab. Ich habe dabei gute Erfahrungen mit einem Wert von 10% gemacht. Betrachten wir als Beispiel das Feld „Related Documentation“:

  Otto möchte einen neuen Testauftrag erstellen. Er wählt einen vorausgefüllten Test-

auftrag für die View login.html. TaskElf extrahiert verschiedene Informationen für das Feld „Related Documentation“: Zum einen wird die Datei „Specification.pdf“ dem Feld hinzugefügt, da sie sich im Projektordner befindet. Zum anderen wird die Datei AdministrationView.pdf hinzugefügt, da Otto diese beim letzten erstellten Testauftrag angehängt hatte. Otto sieht beide Einträge und löscht AdministrationView.pdf aus der Liste, da dieses Dokument bei diesem Testauftrag nicht relevant ist. Anschließend füllt Otto noch die restlichen Felder aus und veröffentlicht den Testauftrag. TaskElf erkennt nun, dass Specification.pdf ein Treffer und AdministrationView.pdf ein Fehlschlag war. Daher wird das Gewicht der Heuristik von Specification.pdf um 10% angehoben, während das der „AdministrationView.pdf“ zu Grunde liegenden Heuristik um 10% gesenkt wird.

Wichtig ist, die Werte nach der Anpassung zu normalisieren, da sonst die Gefahr besteht, mit der Zeit zu große oder zu kleine Werte zu bekommen und so an die Grenzen der gewählten Zahlendarstellung zu gelangen.

5.6. Vereinigung verschiedener Heuristiken

Wenn Informationen aus mehreren Quellen vorliegen, müssen diese verknüpft und zu Folgerungen zusammengefasst werden. Dies gilt umso mehr dann, wenn die Informationen sich gegenseitig widersprechen. Nehmen wir als Bespiel die zwei Heuristiken „Comment Scan“ und „Change Observation“: Beide schlagen jeweils eine Menge von möglichen zu testenden Dateien vor. Diese Mengen können sich überschneiden, gleich oder aber auch disjunkt sein. Aus diesem Grund muss es einen Nachbearbeitungsschritt geben, der die verschiedenen Informationen bewertet, zusammenführt und sortiert.

Wie bereits in dem Beispiel zur Beobachtung des Nutzerverhaltens angedeutet, lassen sich hierzu sehr gut variable Gewichtungen für die einzelnen Heuristiken nutzen. Jede Abwägung, die das System vornehmen muss, kann durch die Gewichte der verschiedenen Möglichkeiten entschieden werden. Ich möchte anhand eines Beispieles verdeutlichen:

Nehmen wir die folgenden drei Heuristiken als Informationsquellen an:

- **„Comment Scan“ (Q_1):** Gewicht von 0.3
- **„Change Observation“ (Q_2):** Gewicht von 0.7
- **„Untested First“ (Q_3):** Gewicht von 0.5

Diese extrahieren bei der Erstellung eines Testauftrags folgende vier Informationen (Erinnerung: Eine Information ist ein Tupel aus dem Inhalt in Form eines Strings und eines Gewichts in Form eines double-Wertes)):

- I_1 **= ["Unit Test, Global.java: onStop()", 0.3] aus** Q_1
- I_2 **= ["Unit Test, Login.java: calculateAgeDifference()", 0.7] aus** Q_2

- I_3 **= ["Gui Test, Login.html", 0.5] aus** Q_3
- I_4 **= ["Unit Test, Global.java: onStop()", 0.5] aus** Q_3

Informationen bekommen standardmäßig die Gewichte, der ihnen zu Grunde liegenden Informationsquellen zu gewiesen.[5] Im nächsten Schritt werden Informationen verschmolzen, die denselben Inhalt haben. In diesem Fall betrifft dies I_1 und I_4. Um dem Umstand Rechnung zu tragen, dass die neue Information von verschiedenen Informationsquellen prognostiziert wurden, werden die Gewichte von I_1 und I_4 dabei aufaddiert. Die resultierende Information sieht dann so aus:

- I_5 **= ["Unit Test, Global.java: onStop()", 0.8] aus** I_1 **und** I_4

Die Liste der Informationen wird nun nach ihrem Gewicht sortiert:

- I_5 **= ["Unit Test, Global.java: onStop()", 0.8] aus** I_1 **und** I_4
- I_2 **= ["Unit Test, Login.java: calculateAgeDifference()", 0.7] aus** Q_2
- I_3 **= ["Gui Test, Login.html", 0.5] aus** Q_3

Obwohl I_2 aus der – laut Gewicht – besten Quelle kam, ist sie durch Fusion doch auf den zweiten Platz gerutscht.

Die jetzt vorliegende sortierte Liste von Informationen wird vom entsprechenden Feld genutzt, um die Vorausfüllung vorzunehmen. In diesem Fall ist es das Feld „TestTypeAndFunction“, das für das Füllen der Liste mit den prognostizierten Testaufträgen zuständig ist. Die Inhalte der Informationen würden daher in der vorliegenden Sortierung direkt in das Feld übernommen werden. Wäre das entsprechende Feld stattdessen ein Feld, in dem nur eine einzelne Information steht, wie dies zum Beispiel beim Honorar-Feld der Fall ist, dann würde nur die oberste Information der Liste beachtet werden.

5.7. Zielfelder der Heuristiken

In diesem Abschnitt möchte ich einen Überblick darüber geben, an welchen Stellen die Heuristiken in TaskElf wirken und welche ich implementiert habe.

[5] Die zusätzliche Gewichtung nach Umfang der Änderungen bzw. des ungetesteten Teils bei „Change Observation“ respektive „Untested First“ ignoriere ich an dieser Stelle, da sie das Prinzip im Sinn nicht verändern und das Beispiel nur unnötig verkomplizieren würden.

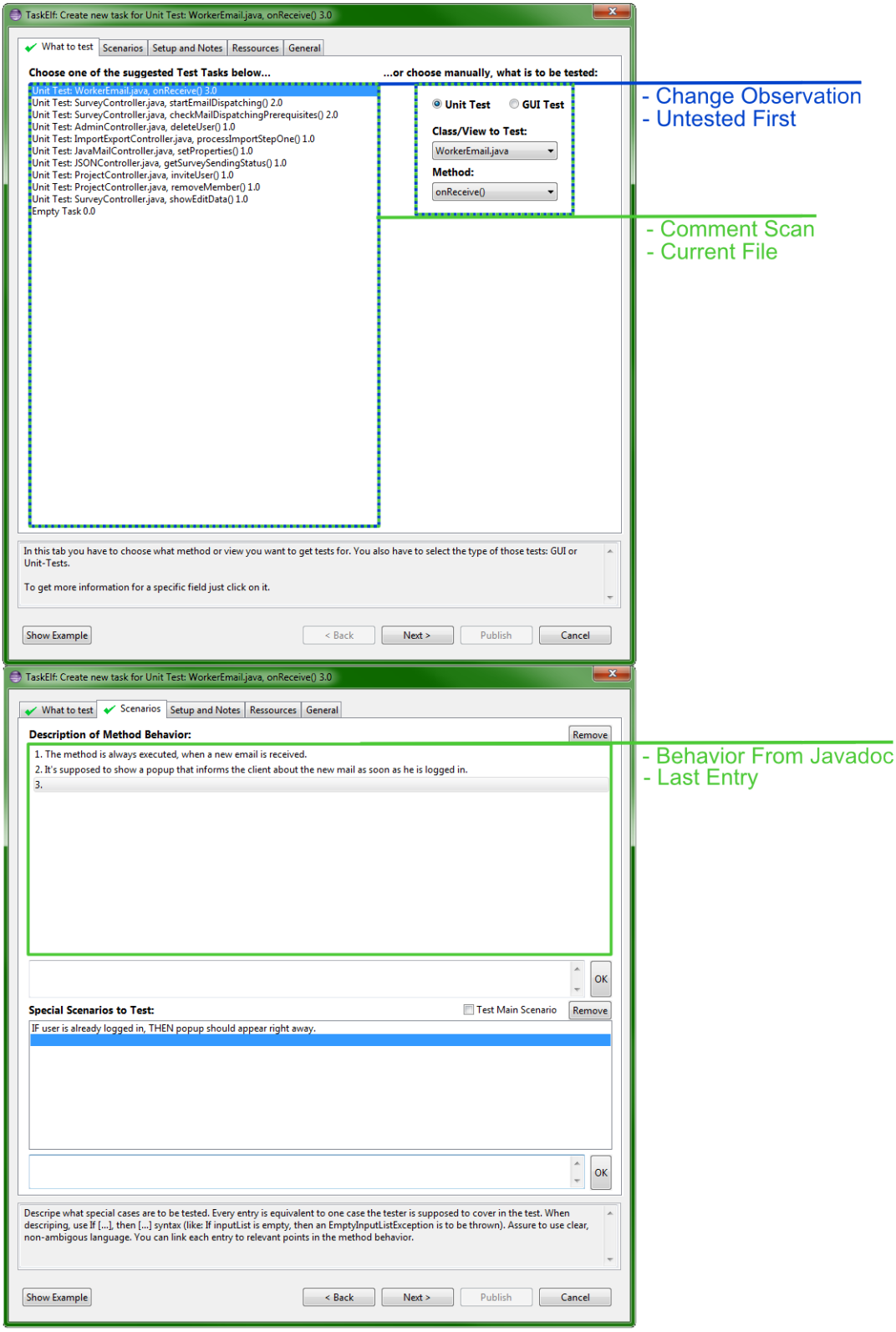

Abbildung 5.3.: Heuristiken in den Reitern „What to test“ und „Scenarios“. Grüne Heuristiken sind in TaskElf bereits implementiert, während blaue lediglich Konzepte sind.

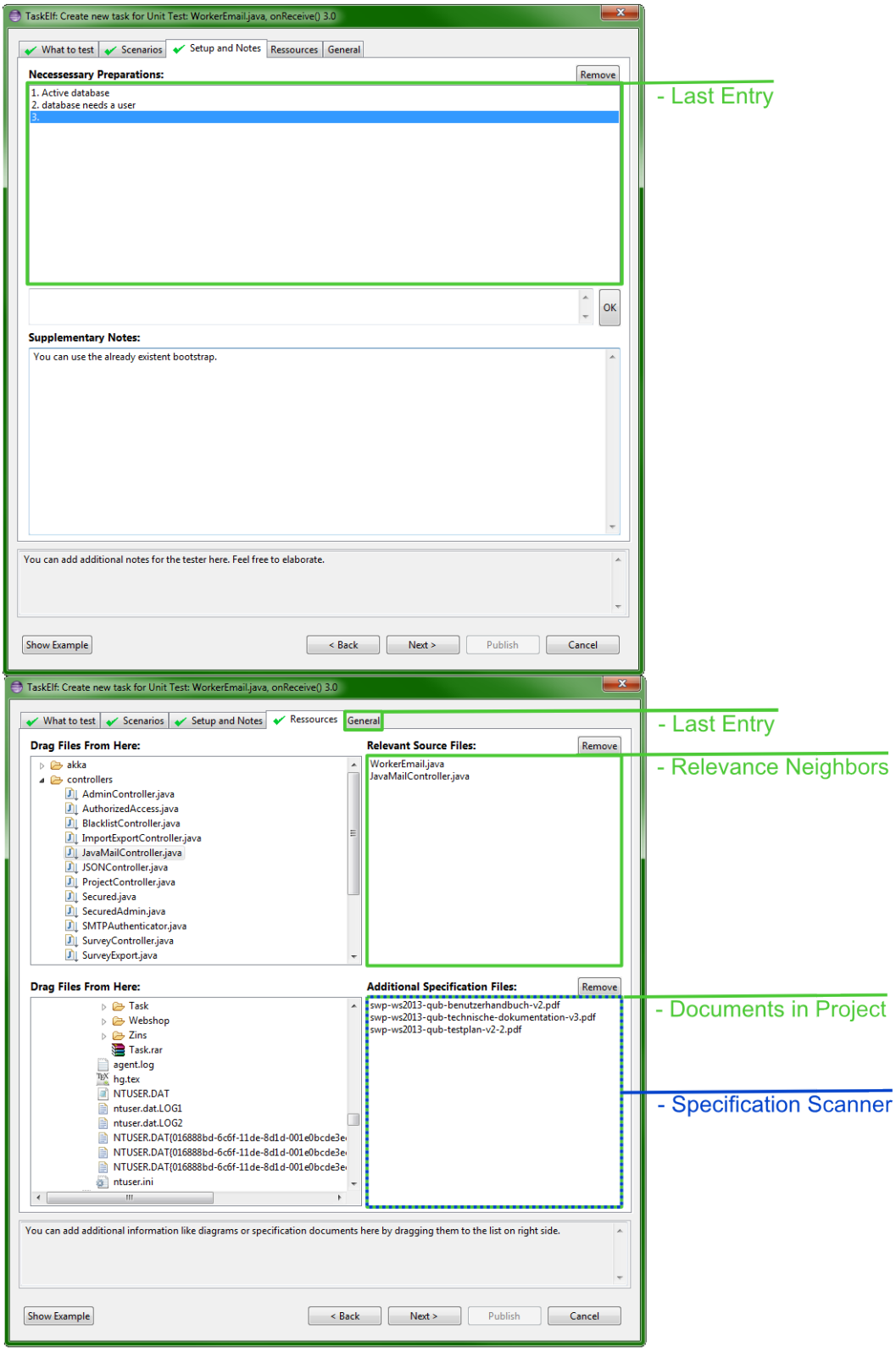

Abbildung 5.4.: Heuristiken in den Reitern „Setup and Notes“, „Ressources“ und „General“. Grüne Heuristiken sind in TaskElf bereits implementiert, während blaue lediglich Konzepte sind.

6. Implementierung von Relevance Neighbors

Kurze Erinnerung: Die relevanten Quellcodedateien sind all jene, die für den Tester möglicherweise wichtig sind und deshalb für ihn hervorgehoben dargestellt werden sollen. Wichtig sind sie, wenn der Tester ihren Inhalt für die erfolgreiche Bearbeitung des Testauftrags kennen muss oder weil ihr Inhalt für das Verständnis von Dateien aus der ersten Gruppe besonders hilfreich ist.

Das bedeutet, dass es zwischen diesen Dateien Zusammenhänge geben muss. Wie bereits im vorherigen Kapitel erwähnt, geht es bei „Relevance Neighbors“ darum, einen Graphen aus diesen Zusammenhängen zu erstellen und mit seiner Hilfe eine Prognose abzugeben, welche Dateien relevant sind. Ein Verfahren, dies zu tun wird in [Antunes2012] vorgestellt. Dabei werden die Beziehungen zwischen den einzelnen Objekten sehr genau modelliert und es gibt 13 verschiedene Zusammenhangstypen zwischen Java-Objekten. Da TaskElf neben Java auch noch HTML betrachten muss und dies zu einer weiteren Erhöhung der Komplexität führen würde, betrachte ich an dieser Stelle nur die in meinen Augen wesentlichen fünf Beziehungen[1]

- **Methodenaufrufe:** Direkte Aufrufe von einer Methode auf eine andere. Diese lassen sich mit Hilfe der Eclipse-Call-Hierarchy extrahieren.[2] Dabei erhält man für jede Methode die Information, welche anderen Methoden im Projekt diese aufrufen.
- **Hyperlinks:** Hyperlinks sind Verknüpfungen zwischen HTML-Dokumenten. Sie können mit Hilfe eines Parsers extrahiert und ausgewertet werden.
- **Routes:** In play! werden Verknüpfungen von HTML-Dokumenten auf Java-Methoden mit speziellen Codes festgehalten, die in der Datei „routes“ stehen. Diese können dort mit Hilfe eines Parsers extrahiert und dann in den HTML-Dokumenten gesucht werden.
- **HTML-Render Aufrufe:** Umgekehrt rufen Java-Methoden HTML-Dokumente auf, indem sie die render()-Methode auf ihnen aufrufen[3]. Dieser Aufruf signalisiert der play!-Applikation, das betreffende HTML-Dokument zu öffnen. Diese Aufrufe können ebenfalls mit einem Parser extrahiert werden.
- **Packagestruktur:** Es lässt sich auslesen, ob Klassen im selben Package liegen oder wie weit sie in der Hierarchie voneinander entfernt sind.

[1] Weitere Beziehungen könnten später in Erweiterungen hinzugefügt werden.

[2] Allerdings müssen dafür interne Eclipse-Ressourcen verwendet werden, die nicht zur API gehören und sich in zukünftigen Versionen ändern können. Für ein Produktivsystem wäre es daher empfehlenswert, die Nutzung des Eclipse-Codes durch ein eigenes Extrahieren der Aufrufe zu ersetzen.

[3] Technisch ist der Prozess komplexer und hier nur sehr vereinfacht dargestellt, da die Technik dahinter von play! übernommen wird und hier keine Rolle spielt.

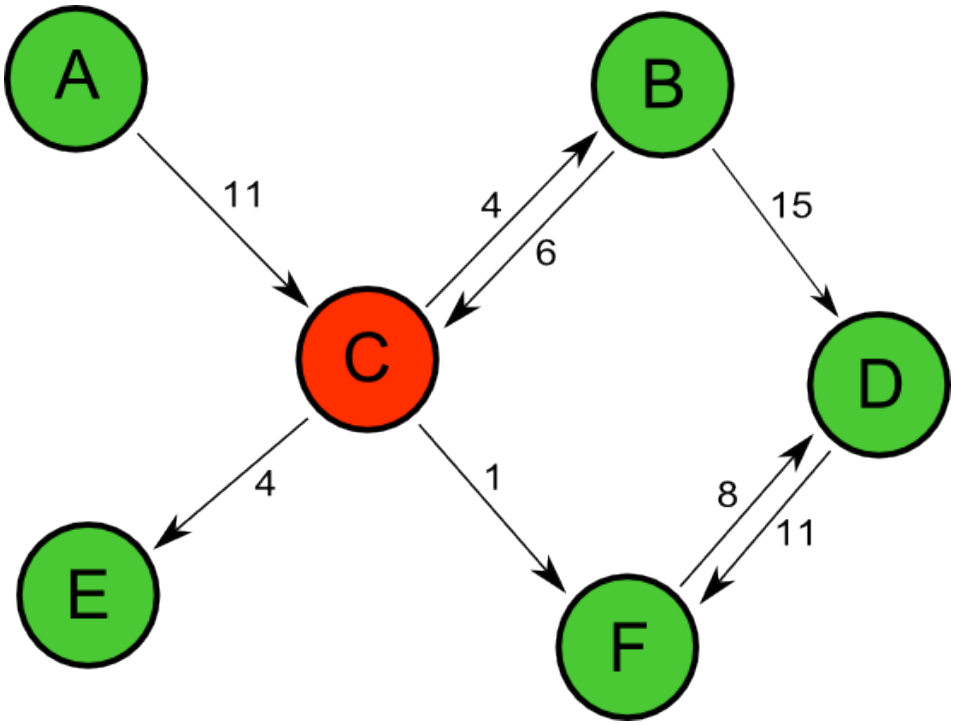

Abbildung 6.1.: Ein Nachbarschaftsgraph, der die Nähe verschiedener Sourcecode-Elemente darstellt.

Abgesehen von der Packagestruktur sind all diese Beziehungen gerichtet und lassen sich direkt in einem gerichteten Graphen umsetzen, bei dem die Knoten die Quellcodeeinheiten und die Kanten die Verbindungen zwischen diesen darstellen (Abb. 6.1). Die beidseitigen Packagebeziehungen zwischen zwei Dateien[4] kann man auf beide gerichteten Kanten zwischen ihren Knoten aufteilen. Die Problematik, dass der Graph Methoden betrachtet, von denen in der Regel mehrere in einer Datei oder Klasse zu finden sind, soll vorerst ignoriert werden. Auf diese wird später noch eingegangen.

Beim Aufbau dieses Graphen kann es sinnvoll sein, verschiedene Beziehungen stärker zu gewichten als andere. So wird vermutlich ein HTML-Dokument, das von der zu testenden Methode geöffnet wird, interessanter für den Tester sein als die toString-Methode einer Klasse, die sich im selben Package befindet. Um herauszufinden, genau welche Gewichtung sinnvoll ist, müssen Versuche durchgeführt und die Ergebnisse bewertet werden. In TaskElf werden nur Methodenaufrufe betrachtet. Diese sind leicht zu erhalten und können zumindest für Unit-Tests bereits erste, zufriedenstellende Ergebnisse liefern. Abgesehen von der nötigen unterschiedlichen Gewichtung der verschiedenen Aufruftypen beim Aufbau des Graphen, gibt es im weiteren Vorgehen aber keine Unterschiede zwischen der Variante, die sämtliche Verknüpfungstypen (Methode → Method, Methode → HTML, HTML → Methode, HTML → HTML) berücksichtigt und der implementierten, beschränkten Variante (nur Methode → Method).

[4] In play! wird die Packagestruktur aus Java auch auf HTML-Dokumente ausgedehnt.

6.1. Algorithmus zur Bestimmung der relevanten Nachbarn

Aus dem so aufgebauten Graphen (Abb. 6.1), muss nun erst noch bestimmt werden, was relevant sein könnte. Das in [Antunes2012] vorgeschlagene Verfahren dies zu tun, betrachtet nur Verbindungen in eine Richtung und dient außerdem in erster Linie dazu, ein Bild davon abzugeben, welche Klassen allgemein für den Entwickler momentan interessant sein könnten. Zusätzlich arbeitet das Verfahren mit Gewichten zwischen 0 und 1. Dies sind alles Abweichungen von der Problemstellung, wie sie in TaskElf auftritt. Stattdessen schlage ich einen eigenen Algorithmus vor.

Die Informationen liegen lediglich implizit vor. Die Situation dabei ist die, dass ein Knoten des Graphen als Eingabe vorgegeben wird (die zentrale Methode oder View des Testauftrags). Im Beispielgraph ist dies Knoten C. Nun gilt es zu entscheiden, welche der anderen Knoten als relevant betrachtet werden sollen.

Für dieses Problem habe ich folgenden Algorithmus entwickelt:

//initialisiere neighbors als Liste von Tupeln

List<node | closeness> neighbors := {}

1. **function getRelevantNodesFor(inputNode):**
2. *List<node> cloud := {}*

   //cloud startet mit zentraler Datei
3. *cloud.add(inputNode | 1)*
4. *currentNode := inputNode*
5. ***FOR ALL** Nachbarn von currentNode, die aus ausgehenden Kanten stammen:*

   //Weit entfernte Kanten (höherer Grad) werden weniger gewichtet
6. *addOrReplace(Nachbar,* $Kantengewicht * \frac{1}{currentNode.degree}$*, currentNode.degree)*
7. ***FOR ALL** Nachbarn von currentNode, die aus eingehenden Kanten stammen:*

   //α erlaubt eingehende Kanten anders zu gewichten
8. *addOrReplace(Nachbar,* $\alpha * Kantengewicht * \frac{1}{currentNode.degree}$ *, currentNode.degree))*
9. *e :=* $max_{closeness}(neighbor)$
10. ***IF** (*$e.closeness = MIN_ELEMENT_CLOSENESS$*) **THEN***
11. *currentNode := e.node*

12. *cloud.add(e.node, e.degree)*
13. ***GOTO** 5.*
14. ***RETURN**: alle Knoten aus cloud*

1. **function addOrReplace(inputNode, closeness, degreeOfCaller):**
2. ***IF** $(inputNode \in neighbors)$ **THEN***

   //degree ist nach der beigetragenen closeness gewichteter degree der Nachbarn in der Wolke
3. *inputNode.degree* := $inputNode.degree * inputNode.closeness + (degreeOfCaller + 1) * closeness / inputNode.closeness + closeness$
4. *inputNode.closeness* := $inputNode.closeness + closeness$
5. ***ELSE***
6. *inputNode.degree := degreeOfCaller + 1*
7. *neighbors.add(inputNode, closeness)*

Der Algorithmus nutzt, ähnlich wie der Algorithmus von Dijkstra ([Dijkstra1959]), eine Menge von Knoten, die bereits abgearbeitet wurden – im nachfolgenden „Wolke“ genannt – und betrachtet die Entfernung der Knoten außerhalb der Wolke zu dieser. Dabei ist allerdings zu beachten, dass die Kantengewichte in diesem Algorithmus die Nähe und nicht die Entfernung beschreiben. Das heißt, dass hier ein sehr hohes Kantengewicht eine besonders große Nähe zwischen zwei Knoten beschreibt, während beim Algorithmus von Dijkstra eine besonders große Entfernung beschrieben würde.

Zu Beginn ist nur der Startknoten in dieser Wolke. Mit jeder Iteration wird der, der Wolke jeweils nächste, Knoten zu dieser hinzugefügt und die Liste der Nachbarn der Wolke aktualisiert. Dies geschieht so lange bis entweder alle Knoten Teil der Wolke sind oder alle Knoten außerhalb der Wolke eine Nähe haben, die unter einem gewissen Threshold (MIN_CLOSENESS) liegt. Alle Knoten in der Wolke werden dann als relevant angesehen.

Der schwierige Teil des Algorithmus ist die Berechnung der Nähe der Nachbarn (Zeilen 6-9). Diese berücksichtigt zwei Sachverhalte. Zum einen sind Aufrufe, die von einem Knoten ausgehen im Allgemeinen relevanter als solche, die auf ihn ausgeführt werden. Eine Methode kann nur dann vollständig verstanden werden, wenn klar ist, was die Funktionen tun, die sie aufruft. Umgekehrt müssen aber nicht alle Methoden bekannt sein, die sie aufrufen. Diesem Umstand wird mit dem Faktor α Rechnung getragen (Zeile 9), der zwischen 0 und 1 liegen sollte. Bei 0 werden eingehende Aufrufe ignoriert, bei 1 werden eingehende und ausgehende Aufrufe gleich behandelt.

Zum anderen muss beachtet werden, dass nicht einfach nur die Entfernung eines Nachbarn zum nächsten Knoten der Wolke relevant ist, sondern dass es immer um

das Verständnis der Eingabemethode geht. Das heißt, dass eine Datei C, die für das Verständnis einer Datei B essentiell ist, durchaus relevant für einen Nutzer sein kann, wenn dieser für das Verständnis der zentralen Datei A die Datei B benötigt. Allerdings wird die Relevanz im Bezug auf Datei A je unwahrscheinlicher, über desto mehr Ecken die Verbindung zu Datei C besteht. Diesem Umstand trage ich Rechnung, indem ich jedem Knoten einen Grad zuweise, der aussagt, wie weit er von der zentralen Datei entfernt ist, die per definitionem den Grad 1 hat. Der Kehrwert des Grades fließt als Faktor in die Berechnung der Nähe der Nachbarn ein, sodass bei einem Grad von 3 an einem Knoten beispielsweise nur noch ein drittel der Nähe dieses Knotens zu seinen Nachbarn außerhalb der Wolke, in die Rechnung einfließt. Der Grad eines Knotens berechnet sich aus dem nach der beigetragenen Nähe gewichteten Mittel der um eins inkrementierten Grade seiner Nachbarn (addOrReplace, Zeile 3). Das bedeutet, dass ein Knoten B, dessen Nähe vollständig von Knoten A, mit dem Grad 2, stammt, den Grad 3 hat. Hat B hingegen nur die Hälfte der Nähe von Knoten A erhalten und die andere Hälfte von Knoten C, mit Grad 1, so hätte Knoten B den Grad 2,5.

Ein ausführliches Beispiel befindet sich in Anhang A.

6.2. Unterteilung in Klassen und Methoden

Bisher wurden die Knoten gleichbedeutend mit Dateien betrachtet. Für Methoden stimmt dies allerdings zumeist nicht. Eine Datei enthält nur eine öffentliche Klasse, die wiederum 0-n Methoden enthält (Abb. 6.2). Einzelne, aus einer Klasse heraus getrennte Methoden an den Tester zu verschicken ist fehleranfällig, da Klassenattribute und Imports maßgeblich Einfluss auf eine Methode nehmen. Daher werden an den Testauftrag nur vollständige Klassen angehängt. Um dies zu berücksichtigen, habe ich am zuvor betrachteten Algorithmus zwei Dinge verändert. Zum einen erhält jeder Knoten eine Datei, zu der er gehört, die bei Rückgabe beachtet wird:

14. *Bilde für alle Dateien* D_i *die closeness, wobei*
    closeness := $\sum node.closeness | node \in D_i \wedge node \in cloud$
15. ***RETURN****:* $alleDateien | closeness \geq MIN_FILE_CLOSENESS$

Zum anderen wähle ich MIN_ELEMENT_CLOSENESS etwas niedriger, sodass am Ende mehr Knoten in der Wolke sind. Wenn MIN_FILE_CLOSENESS größer als MIN_ELEMENT_CLOSENESS gewählt wird, werden in diesem letzten Schritt wieder Knoten aussortiert. Dabei muss allerdings beachtet werden, dass bei Knoten, die wieder aussortiert werden, auch die Nähewerte ihrer Nachbarn noch einmal angepasst werden müssen. Ich habe gute Erfahrungen mit der Konfiguration MIN_ELEMENT_CLOSENESS = 1, MIN_FILE_CLOSENESS = 2 und α=0,5 gemacht.

Diese 2-Ebenen-Variante lässt sich mit der flachen Variante, die bei HTML-Dokumenten zum Einsatz kommen würde, kombinieren: Die HTML-Dokumente sind dann einfach

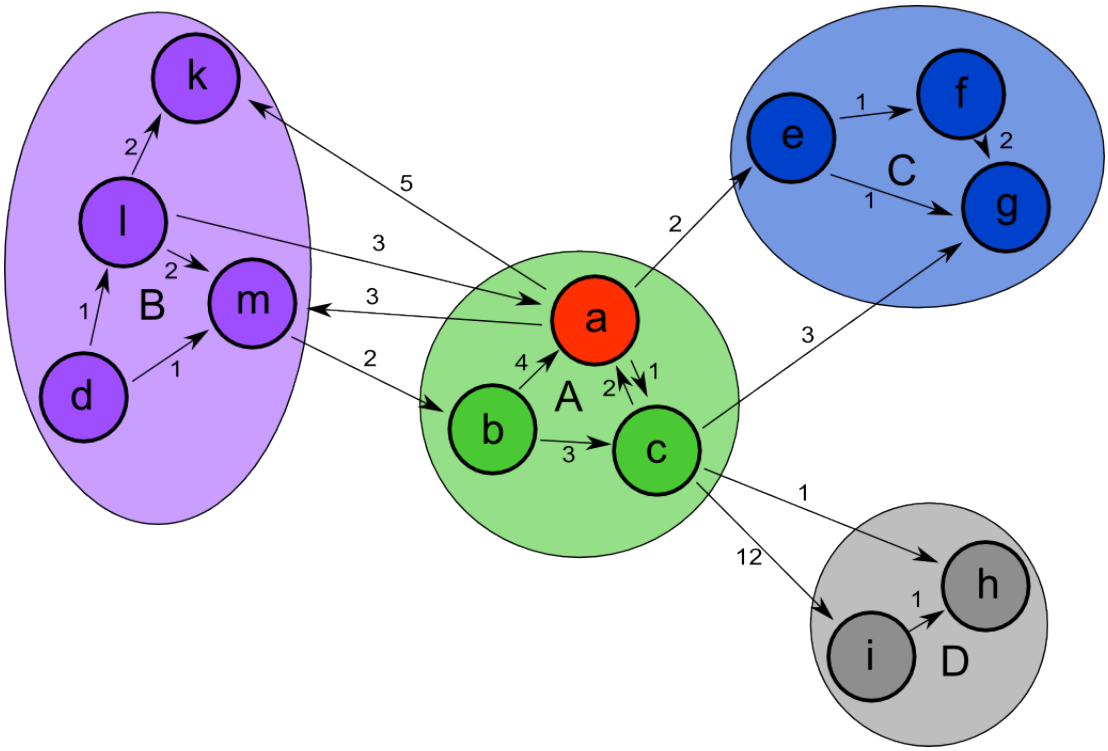

Abbildung 6.2.: Erweiterung des Nachbarschaftsgraphen, um eine neue Ebene. Es wird zwischen Methoden (Kleinbuchstaben) und Klassen (Großbuchstaben) unterschieden.

Knotenmengen, die nur einen einzelnen Knoten enthalten.

6.3. Probleme der Umsetzung

In der aktuellen Version von TaskElf wird der Graph erst aufgebaut, sobald der Nutzer das Plugin über die entsprechende Schaltfläche in der Entwicklungsumgebung betätigt. Da Eclipse für jede Methode im Projekt gesondert die Aufrufhierarchie abrufen muss, dauert dieser Vorgang je nach Rechner ein paar Sekunden (auf meinem Testsystem circa 13) und führt zu einer Wartezeit für den Entwickler. Dies ist schlecht, da es seinen Arbeitsfluss stört. Zusätzlich gilt zu bedenken, dass diese Wartezeit bereits im vorliegenden Prototypen besteht, in dem lediglich eine von vier derartigen Verknüpfungsarten betrachtet wird (die Packagestruktur kann außer Acht gelassen werden, da keine zeitaufwändigen Operationen nötig sind, um diese zu erhalten).

Der Graph selbst ist universell im Projekt und muss nicht für einzelne zentrale Methoden – zu denen die relevanten Dateien bestimmt werden sollen – erstellt werden. Dies ermöglicht es, die Aufrufhierarchien – zum Beispiel im XML-Format – im Projekt vorrätig zu halten und beim Aufruf von TaskElf lediglich diejenigen Teile neu zu bestimmen, die sich seit dem letzten Aufruf geändert haben. Dabei müsste allerdings bekannt

sein, welche Änderungen es gibt, was mit Problemen verbunden ist, wie es bereits in 5.4.2 betrachtet wurde. Alternativ könnte dies auch im Hintergrund geschehen, während der Entwickler noch programmiert. Immer wenn er eine Datei speichert, könnten die Einträge, die mit dieser Datei zusammenhängen, aktualisiert werden. Da die Ermittlung der Aufrufe einzelner Methoden kaum Rechenzeit erfordert und die Wartezeit nur durch die große Zahl von Methoden entsteht (im Testsystem 152), kann so eine Ausführung ohne Wartezeiten für die Erstellung des Graphen sichergestellt werden.

7. Evaluation

In diesem Kapitel beschreibe ich die Evaluation der aktuellen Version von TaskElf. Dazu werde ich zuerst beschreiben, was die Ziele der Evaluation sind. Anschließend werde ich den dafür gewählten Versuchsaufbau und die Rahmenbedingungen erläutern, bevor ich den Ablauf des Versuchs beschreibe. Danach stelle ich die Ergebnisse vor und werte diese aus. Abschließend betrachte ich mögliche Fehlerquellen im Versuch.

7.1. Ziel

Die Evaluation hat zwei Hauptziele und ein Nebenziel. Im Kern geht es darum, herauszufinden, ob die Erstellung von aussagekräftigen[1] Testaufträgen mit Hilfe von TaskElf schneller ist als die eigenständige Erstellung der entsprechenden Tests. Daraus leiten sich die beiden Kernfragen ab:

1. **Ist die Erstellung eines Testauftrags durch einen Nutzer schneller als die Erstellung dieses Tests durch den Nutzer selbst?**
2. **Erstellen Nutzer aussagekräftige Testaufträge?**

Neben diesen beiden Kernfragen nutze ich die Evaluation außerdem, um allgemeine Eindrücke einzufangen, die Nutzer von TaskElf haben. Dieser zusätzliche, explorative Ansatz soll, neben dem Finden von Verbesserungsmöglichkeiten, dabei helfen, Informationen zur Akzeptanz der Nutzer gegenüber dem Crowdsourcingprozess im Allgemeinen und TaskElf im Speziellen zu sammeln.

7.2. Versuchsbedingungen

Der Versuch fand in Zusammenarbeit mit Florian Eiteljörge statt und war in zwei Teile gegliedert. Im ersten Teil wurde TaskElf und im zweiten das Schwesterplugin aus [Eiteljörge2014] evaluiert. Dies erlaubte es uns, die vorhandenen Probanden möglichst effizient zu nutzen, ohne in eine Konkurrenzsituation zu geraten.

Die Evaluation fand am Ende der Arbeit statt und war vom zeitlichen Umfang her sehr eingeschränkt. Die zu untersuchende Population bestände idealerweise nur aus professionellen Softwareentwicklern. Da diese allerdings nicht in ausreichender Zahl zur

[1] „aussagekräftig“ bedeutet in diesem Zusammenhang, dass der Testauftrag genug Informationen enthält, um dem Tester die Erstellung des gewünschten Tests zu ermöglichen.

Verfügung standen, haben wir die Population auf Personen ausgedehnt, die regelmäßig programmieren. Unsere Stichprobe besteht größtenteils aus Studenten und wissenschaftlichen Mitarbeitern der Universität. Um eine Motivation zu geben, wurde allen Probanden ein Schokoladenosterhase in Aussicht gestellt.

Die Alternative, eine Firma zu finden, die sich bereit erklärt, die Plugins im Praxiseinsatz zu testen, war für die uns zur Verfügung stehende Zeit zu aufwändig. Da die Vorlesungszeit allerdings erst zwei Wochen vor Abgabe dieser Arbeit wieder startete und sich vorher nur ein sehr kleiner Teil der Population in der Universität befand, musste der Versuch innerhalb einer Woche ablaufen, da sonst keine Zeit mehr für die Auswertung und Niederschrift geblieben wäre. Dies ist ein bestimmender Faktor für die Evaluation und führt dazu, dass wir nur eine sehr kleine Stichprobe der Population betrachten konnten.

Ein weiterer Faktor ist die begrenzte Zeit, die wir einem einzelnen Probanden abverlangen konnten. Um wirklich realistische Szenarien in den Versuchen zu testen, hätten diese mehrere Stunden dauern müssen – eine Zeit, die wir freiwilligen Probanden nicht abverlangen konnten. Daher musste der Versuchsaufbau sehr vereinfacht werden, was die Übertragbarkeit der Ergebnisse auf den realen Einsatz stark einschränkt. Die einzelnen Versuche dauerten allerdings auch in dieser Minimalfassung noch insgesamt etwa 80 Minuten pro Proband.

Außerdem ist die momentane Beschränkung von TaskElf auf play!-Projekte ein limitierender Aspekt: Ein Versuch, der Erfahrungen mit play! voraussetzt, hätte die Populationsgröße maßgeblich verringert.

7.3. Versuchsaufbau

Ich beschränke mich bei der Beschreibung des Versuchs auf den ersten Teil, der TaskElf betrifft. Durch die zeitliche Abfolge ist die Evaluation von TaskElf vom zweiten Teil des Versuchs unabhängig. TaskElf wurde mit der Erstellung von Unit-Tests evaluiert, da für die Erstellung einer GUI tiefgehende play!-Erfahrungen sowie bedeutend mehr Zeit nötig gewesen wären.

Der Versuch war in vier Phasen gegliedert:

1. **Aufnahme demographischer Daten:** Dies erfolgte durch einen Fragebogen, den die Probanden selbst ausfüllten. Hierin wurden neben allgemeinen Angaben wie Alter oder Studiengang auch allgemeine Erfahrungen mit verschiedenen Technologien und Testen abgefragt. Dies erfolgte mit Hilfe einer fünf Punkte Likert-Skala (siehe Anhang B.1).

2. **Programmieraufgabe:** Die Probanden sollten eine kurze Methode mit vorgegebenem Verhalten schreiben (siehe Anhang B.2).

3a. **Erstellung von Testfällen:** Die Probanden sollten einen JUnit-Test mit zwei Testfällen schreiben, die jeweils unterschiedliche – vom Probanden selbst gewählte – Fälle abdecken sollten. Die benötigte Zeit wurde gemessen. Anschlie-

ßend wurden sie aufgefordert, zwei neue Testfälle zu schreiben. Die Zeit wurde abermals gemessen.

3b. **Erstellung von Testaufträgen:** Die Probanden sollten einen Testauftrag für zwei JUnit-Tests zu zwei selbst gewählten Fällen ausschreiben. Dazu sollten sie TaskElf nutzen. Die benötigte Zeit wurde gemessen. Anschließend wurden sie aufgefordert, einen weiteren Testauftrag für zwei neue Tests zu erstellen. Die Zeit wurde abermals gemessen.

4. **Interview** Diejenigen Probanden, die TaskElf genutzt haben, wurden zu ihren Erfahrungen interviewt.

Ob ein Proband Testfälle oder Testaufträge erstellen sollte, wurde mit Hilfe eines einfachen Algorithmus bestimmt: Wenn ein Proband viel Erfahrung mit Unit-Tests hatte (Likert-Skala 4 oder 5), wurde er derjenigen Gruppe zugeordnet, die bisher eine geringere Anzahl solcher Probanden enthielt. War dies ausgeglichen, wurde er der kleineren Gruppe zugewiesen. Probanden mit weniger Erfahrung wurden der kleineren Gruppe zugeordnet. Immer, wenn sich ein Proband mit den vorherigen Regeln nicht zuordnen ließ, entschied ein Münzwurf. Ich werde im Folgenden den Ablauf der einzelnen Phasen im Detail beschreiben.

7.3.1. Aufnahme demographischer Daten

Zu Beginn sollten die Probanden einen Fragebogen (Anhang B.1) ausfüllen, in dem sie ihr Alter, ihren Studiengang, ihr Semester und einige Selbsteinschätzungen zu ihrer Erfahrung mit verschiedenen Technologien und Programmieren sowie Testen im Allgemeinen angeben sollten. Ursprünglich sollten lediglich Studenten befragt werden. Als sich dies änderte, wurden die Felder „Studiengang“ und „Semester“ handschriftlich durch die Angabe des Berufs ersetzt. Die gesammelten Daten zu „Semester“ sind außerdem auch für die Studenten nicht vergleichbar und wurden gestrichen, da nicht fest vorgegeben war, ob Gesamtsemester oder nur Semester im momentanen Studiengang gefordert waren.

7.3.2. Programmieraufgabe

Bei der Programmieraufgabe sollten die Probanden einen vorgefertigten Methodenmock nach Anforderungen aus einem pdf-Dokument (Anhang B.2) vervollständigen. Die Methode sollte den größten gemeinsamen Teiler zweier Zahlen berechnen und dabei eine Reihe von Nebenbedingungen berücksichtigen. Eine dieser Nebenbedingungen war es, dass die Eingabezahlen mindestens fünfstellig sein sollten. Dies sollte das Erstellen von Testfällen etwas verkomplizieren.

Die Probanden wurden vor Beginn der Aufgabe gebeten, wartbaren Code zu erzeugen und sich vorzustellen, die Aufgabe im Rahmen eines größeren Projektes auszuführen. Außerdem wurden sie darauf hingewiesen, dass dieser Teil der Aufgabe nur vorbereitend ist, nicht kontrolliert wird und lediglich dazu dient, dass sie den Code, mit dem im

weiteren Verlauf gearbeitet würde, gut kennen. Auf diese Weise habe ich versucht, die Probanden zu möglichst von der Prüfungssituation unabhängigem, Programmierverhalten zu motivieren.

Die Probanden hatten während der Aufgabe Zugriff auf das Internet.

7.3.3. Erstellung von Testfällen

Die Kontrollgruppe, die Testfälle erstellen sollte, fand bereits eine vorgefertigte Testklasse in Eclipse geöffnet vor und musste lediglich die Assert-Prüfungen einfügen. Bei Bedarf konnten sie die Benutzung von JUnit im Internet nachlesen. Als zusätzliche Schwierigkeit bei der Erstellung, wurde als Rückgabetyp der Programmieraufgabe der Datentyp „Integer“ gewählt, sodass für den Vergleich entweder gecastet oder ein neues Objekt erstellt werden musste. Wenn ein Proband mehr als zwei Minuten an diesem Problem verbrachte, wurde ihm geholfen. Die Zeit für den gesamten Erstellvorgang wurde gemessen.

Um Lernzeiten aus der Zeitmessung zu entfernen, wurde die Aufgabe im Anschluss mit zwei neuen Testfällen wiederholt. Die neuen Testfälle sollten sich die Probanden vor Beginn der Zeitmessung überlegen.

Erstellung von Testaufträgen

Den Mitgliedern der Gruppe, die Testaufträge erstellen sollte, wurde erklärt, dass sie die Tests von einer projektfremden Person erstellen lassen und dazu einen Auftrag ausschreiben sollten. Sie wurden explizit darauf hingewiesen, dass diese Person projektfremd sei und damit nur über die Informationen verfüge, die sie der Person zukommen ließen. Während der Erstellung des Testauftrags, waren ihre einzigen Hilfen der Inhalt der Hilfebox in TaskElf sowie der „Show Example“-Button, dessen Funktion mit Hilfe von ausgedruckten Bildern von TaskElf gemockt wurde. Es wurde die Zeit zwischen Betätigung des „TaskElf: Create Test Task“-Buttons und dem Drücken des „Publish“-Buttons in TaskElf gemessen.

Auch hier wurde der Lernaufwand wieder berücksichtigt, indem die Aufgabe wiederholt wurde. Zwischen den Wiederholungen wurden Bedienfragen beantwortet und gegebenenfalls Missverständnisse ausgeräumt, sodass die Probanden beim zweiten Durchlauf wussten wie genau sie die Felder auszufüllen hatten.

Vor jedem neuen Probanden wurde TaskElf in einen Ausgangszustand zurückversetzt, indem die Konfigurationsdateien gelöscht wurden. Damit war das Verhalten von TaskElf vollkommen unabhängig von den Vorprobanden. Zwischen dem ersten und dem zweiten Durchgang wurde TaskElf hingegen nicht zurückgesetzt, sodass Eingaben wie die E-Mail-Adresse und das Methodenverhalten, erhalten blieben. Da das pdf-Dokument mit der Aufgabenstellung den Probanden im Projekt zur Verfügung gestellt wurde, wurde dieses von TaskElf automatisch dem Testauftrag hinzugefügt.

Interview

Während die Kontrollgruppe lediglich die Schwierigkeit der Programmieraufgabe beurteilen sollte, wurden den Probanden, die TaskElf genutzt haben, einige ausführlichere Fragen gestellt (Anhang B.3). Die Frage nach einer potentiellen privaten Nutzung wurde im Verlauf der Evaluation gestrichen, da sie sich als nicht zielführend herausstellte und somit nur unnötig Zeit kostete. Die Frage nach der Schwierigkeit der Programmieraufgabe wurde vor Beginn der Studie vorgezogen und ebenfalls in eine Likert-Skala-Frage gewandelt.

Das Interview diente dazu, Eindrücke der Probanden aufzunehmen und Verbesserungsansätze zu sammeln.

7.4. Ergebnisse

Wir haben die Plugins über einen Zeitraum von drei Tagen mit zehn Probanden evaluiert. Diese Probandengruppe bestand aus Informatikstudenten (6), wissenschaftlichen Mitarbeitern des Fachgebiets für Software Engineering (3) und einem professionellen Softwareentwickler. Bis auf den Softwareentwickler, der 54 Jahre alt war, waren alle anderen Probanden zwischen 24 und 30 Jahren.

Die Aufschlüsselung der Demographie ist in Tabelle 7.1 sichtbar. Die Kontrollgruppe ist durch den Softwareentwickler im Schnitt fünf Jahre älter. Außerdem hat sie laut Selbsteinschätzung etwas weniger Erfahrung als die Plugin-Gruppe, vor allem mit Eclipse, aber auch allgemein beim Testen. Dies könnte sich auf die Ergebnisse ausgewirkt haben. Die Nutzung von Eclipse spielte allerdings für das Erstellen der Testfälle und -aufträge keine Rolle, da die Testklasse bereits angelegt war und nur ausgefüllt werden musste. Mit der Verwendung der Syntaxergänzung hatte augenscheinlich keiner der Probanden Probleme.

Attribut	Ø Plugin-Gruppe	Ø Kontrollgruppe	Ø Gesamt
Alter (Jahre)	27,2	32,4	29,8
Programmiererfahrung	4	4,2	4,1
Testerfahrung	3,2	2,8	3
Erfahrung mit Eclipse	4	3,2	3,6
Erfahrung mit Unit-Tests	3	2,8	2,9

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

7.4.1. Quantitativer Versuchsteil

In Tabelle 7.2 sind die Ergebnisse des Versuchs sichtbar. Bei beiden Gruppen wurde die Erstellungszeit in der Wiederholung signifikant kürzer. Während durch die nötige

Lernzeit die durchschnittliche Zeit im ersten Versuch noch bei beiden Gruppen über sieben Minuten lag, ging sie im zweiten Teil auf unter zweieinhalb Minuten herunter. Zwischen den Gruppen sind die Unterschiede bei den durchschnittlichen Zeiten sehr gering. Die Werte für die Plugin-Gruppe werden allerdings sehr stark von einem Ausreißer bestimmt wie der Median es schon andeutet. Während der Proband mit der zweithöchsten Zeit in der Wiederholung mit 2:18 Minuten sogar noch unter dem Durchschnitt liegt, brauchte der Proband mit der höchsten Zeit 4:56 Minuten. Es ist allerdings bei der sehr kleinen Stichprobe nicht möglich, zu beurteilen, ob es sich tatsächlich um einen Ausreißer handelt oder ob die anderen Probanden einfach besonders schnell waren. Dazu wäre ein größerer Versuch notwendig.

Attribut	**Plugin-Gruppe**	**Kontrollgruppe**
Ø Zeit, Versuch 1 (min)	7:20	7:49
Ø Zeit, Versuch 2 (min)	2:23	2:07
Median Zeit, Versuch 1 (min)	6:30	7:12
Median Zeit, Versuch 2 (min)	1:35	2:15
# Anzahl Fehler	8	4
# Anzahl Fehler Typ 2 und 47.4.1	3	1

Tabelle 7.2.: Quantitative Ergebnisse des Versuchs.

Die Frage, ob die Erstellung eines Testauftrags schneller ist als das Erstellen der Tests, lässt sich mit diesen Ergebnissen nicht vollends klären. Während die Erstellung auch in der momentanen TaskElf-Version beim getesteten Szenario zumindest keine schlechtere Leistung gezeigt hat als die direkte eigenständige Erstellung der Tests, war sie aber auch nicht schneller. Aufgrund der Versuchsbeschränkungen war die Aufgabe, für die TaskElf in diesem Versuch benutzt wurde, sehr einfach. Dies hat auf der einen Seite dazu geführt, dass die eigenständige Erstellung von Testfällen sehr einfach war und daher sehr schnell ging. Beispielsweise musste für den Test keine Datenstruktur aufgebaut und initialisiert werden, sondern es konnten direkt Asserts geschrieben werden. Auf der anderen Seite war die Erstellung des Testauftrags aber auch nicht sehr komplex. Es mussten keine Vorbereitungen, zusätzlichen Anmerkungen oder relevanten Dateien angegeben werden. Bei einer komplexeren Aufgabe würde also beides steigen: der Bearbeitungsaufwand und der Erstellaufwand. Was davon schneller steigt, müsste ein entsprechender Versuch klären.

Die zweite Frage bezog sich auf die Qualität der resultierenden Testaufträge. Da die Versuchszeit es nicht zuließ, die resultierenden Testaufträge bearbeiten zu lassen und dafür auch kein Plugin existiert hätte, da in [Eiteljörge2014] lediglich GUI-Tests behandelt werden, habe ich die Testaufträge und Tests von Hand geprüft. Ich habe dabei nicht betrachtet, ob die Testaufträge genau so geschrieben sind wie das Plugin es vorsieht. Das Kriterium war einzig und allein die Frage, ob zwei korrekte Testszenarien im Testauftrag stehen und ob diese mit Hilfe der mitgelieferten Informationen lösbar sind. Bei dieser Untersuchung sind mir vier Arten von Fehlern begegnet:

1. **Verständnis der Anforderungen:** Dies war der häufigste Fehler. Er trat sowohl in den Testaufträgen als auch in den Tests auf. Der Proband hat einen Teil der

Anforderungen entweder nicht gelesen oder falsch verstanden und daraufhin bereits Fehler in die Methode programmiert. Da das Erstellen von Tests oder Testaufträgen durch den Entwickler nur eine Verifikation ist, dass der Code das tut, was der Entwickler möchte, können diese Fehler nicht durch solche Tests gefunden werden.

2. **Fehler beim Ausschreiben:** Hierbei handelt es sich um „echte“ Fehler beim Schreiben des Tests oder Testauftrags. Der Entwickler schreibt etwas anderes auf als er eigentlich beabsichtigt. Ein Beispiel wäre ein vergessenes Minus vor einer Zahl.
3. **Sinnhaftigkeit des Testszenarios:** Dies ist kein Fehler im engeren Sinne. Ein Testszenario, das nicht sinnvoll ist, kann dennoch richtige Ergebnisse liefern, aber genügt dem Minimalitätsprinzip des Testens nicht. Ein Beispiel wäre: „Der größte gemeinsame Teiler (ggT) zweier Zahlen, bei denen die zweite Zahl doppelt so groß ist wie die erste, muss kleiner sein als die größere Zahl.“ Dies ist zwar zweifellos richtig, jedoch ist in dem Fall immer die kleinere Zahl der ggT. Daher bietet es sich an, dies direkt zu testen. Oder noch allgemeiner: „Wenn eine der beiden Zahlen das Vielfache der anderen Zahl ist, so ist der ggT diese andere Zahl.“
4. **Verständnis der Programmiersprache:** Hierbei handelt es sich um einen Fehler, der aus falschen Annahmen beim Verhalten der Programmiersprache resultiert. Ein Beispiel wäre die Annahme, dass der Rückgabewert „null“ automatisch zu einer NullPointerException führt. Wenn nun „null“ zurückgegeben wird, im Testauftrag aber geschrieben wird, dass eine NullPointerException auftritt, so führt dies zu einem falschen Test.

All diese Fehler können sowohl in Tests als auch in Testaufträgen auftreten. Dabei kann man allerdings zwei Umstände unterscheiden: Fehler der Typen 2 und 4 werden, sofern sie beim Erstellen des Tests entstanden sind werden wahrscheinlich sofort, wenn die entsprechenden Tests fehlschlagen. Bei Testaufträgen können sie zwar theoretisch entdeckt werden, führen aber vermutlich in der Regel zu falschen Ergebnissen. Diese Fehlertypen sollten daher in Testaufträgen möglichst nicht auftreten. Fehler der Typen 1 oder 3 hingegen können beim eigenständigen Erstellen der Tests nicht entdeckt werden. Beim Erstellen der Testaufträge gibt es hingegen zumindest die Chance, dass ein aufmerksamer Tester diese entdeckt.[2] Da diese beiden Typen von Fehlern bereits vor der Erstellung des Testauftrags passieren, sind sie von TaskElf unabhängig und bei der Bewertung nicht relevant.

Das Gesamtauftreten der Fehler der Typen 2 und 4 ist bei beiden Gruppen ähnlich (3/20 in der Plugin-Gruppe, 1/20 in der Kontrollgruppe) und unterscheidet sich nicht signifikant. Ob dies bei größeren Stichproben weiterhin gilt, müsste in einem weiteren Versuch geprüft werden. Alles in allem lässt sich sagen, dass die resultierenden Testaufträge qualitativ mit den erstellten Tests vergleichbar sind.

Eine Aufstellung der gesamten quantitativen Ergebnisse findet sich in Anhang B.4.

[2] Bei Typ 1 gilt dies nur unter der Voraussetzung, dass der Entwickler dem Tester auch ein entsprechendes Anforderungsdokument zur Verfügung gestellt hat.

7.4.2. Qualitativer Versuchsteil

Die Grundlage für die qualitative Analyse von TaskElf bilden die Antworten der Probanden (Anhang B.6) im Interview sowie die Beobachtung der Probanden während der Bearbeitung der Aufgabenstellung. Insgesamt ist aufgefallen, dass die Probanden sehr schnell eine Vorstellung davon hatten, was sie in einem Reiter der Oberfläche tun sollten. Dies führte allerdings dazu, dass sie die Kontexthilfe weitestgehend ignorierten und erst zu Rate zogen, beziehungsweise bemerkten, wenn sie an einer Stelle unsicher wurden. Dies geschah – wie es zu erwarten war – meist im zweiten Reiter, in dem die Szenarien angegeben werden müssen. Gerade die Eingabe der Testszenarien ist durch die „If [...], then [...]"-Schreibweise etwas, was man ohne Lesen der Erklärung oder des Beispiels nicht richtig machen kann. Es wäre daher sehr sinnvoll, die Hilfen, beispielsweise durch Icons oder Farben, deutlicher hervorzuheben und speziell bei der erstmaligen Eingabe der Szenarien auf die Hilfen hinzuweisen. Dies könnte Nutzern den ersten Einstieg erleichtern. Es war allerdings trotz dieser Probleme zu beobachten, dass die meisten Nutzer dennoch nach ein paar Minuten in der Lage waren, die Szenarien im Wesentlichen so aufzuschreiben wie sie gedacht waren. Jedoch wurde dann das „then" oft weggelassen, da dies grammatikalisch gebräuchlicher ist. Das Plugin auf Seiten von [Eiteljörge2014] benötigt allerdings zwingend diese If-then-Syntax. Daher muss in einer zukünftigen Version bei der Eingabe eine Prüfung erfolgen, ob sie syntaktisch korrekt ist und der Nutzer gegebenenfalls zu einer Korrektur gezwungen wird.

Am häufigsten wurde in den Interviews die Testauftragsliste im ersten Reiter erwähnt. Interessanterweise gilt dies sowohl für die Frage, was den Probanden gefallen hat als auch, was den Probanden nicht gefallen hat. Ein Proband fand es sehr gut, dass die gewünschte Testklasse beim zweiten Starten weiter oben in der Liste stand, während zwei andere Probanden dies kritisierten. Teilweise wurden auch Funktionen wie die Wahl der verwendeten Heuristiken oder die Kenntlichmachung der einem Vorschlag zu Grunde liegenden Heuristiken, gewünscht, die für die Zukunft bereits angedacht sind (siehe Kapitel 4). Auch die Möglichkeit einer alphabetischen Sortierung wurde gewünscht. Dies ließe sich in der Zukunft leicht umsetzen.

Außerdem gab es noch zwei weitere Anregungen: Zum einen wurde angesprochen, dass bei Unit-Tests der erste Reiter verlassen werden kann, ohne dass eine Methode gewählt ist – dies ist unvorteilhaft, da eine sehr wichtige Information so vergessen anzugeben werden könnte. Zum anderen fiel einem Probanden auf, dass das Beschreibungsfeld im vierten Reiter leer genauso aussieht wie die leeren Listenfelder zuvor („Preparation", „Main Scenario", „Scenarios to Test"). Dies sollte optisch besser voneinander abgetrennt werden.

Insgesamt wurde TaskElf sehr positiv aufgenommen. Es wurden vor allem die gute Führung durch den Prozess, die gute Hilfe und die Tatsache, dass viele Felder vorausgefüllt waren, gelobt.

Auch die Idee des Auslagerns von Tests in die Crowd wie sie in dieser Arbeit beschrieben wurde, hat den Probanden gut gefallen. Prinzipiell können sich alle fünf Plugin-Probanden vorstellen, TaskElf im Firmenumfeld zu nutzen. Entscheidend für

einen Einsatz wäre allerdings, ob die Gesamtprozedur des Crowdsourcings der Tests funktioniert und die Tester gute Ergebnisse produzieren.

7.5. Fehlerquellen

Es gibt einige Faktoren, die sich auf die Qualität der Ergebnisse ausgewirkt haben könnten. Während ich auf einige bereits zuvor eingegangen bin, fanden andere noch keine Erwähnung. An dieser Stelle möchte ich eine Auflistung möglicher Fehlerquellen geben.

- **Laborsituation:** Eine Laborsituation führt immer zu einem gewissen Bias. Probanden verhalten sich anders, wenn sie wissen, dass sie beobachtet werden als sie es normalerweise tun würden.
- **Sehr einfache Aufgabe:** Wie bereits zuvor erwähnt, ist dies die größte Verzerrung der Ergebnisse. Die Aufgabe war nicht mit dem Rest des Projekts verknüpft und die Testfälle so einfach, dass Probanden sich fragten wieso sie das nicht direkt selbst schnell testen.
- **Falsche Gruppenzuordnung bei Proband 8:** Ich habe Proband 8 versehentlich der Plugin-Gruppe zugeordnet, obwohl er laut Algorithmus in die Kontrollgruppe gehört hätte. Dies geschah, da ich mich in meinen Aufzeichnungen verzählt hatte. Ich gehe allerdings nicht davon aus, dass dies einen kontraproduktiven Einfluss auf die Evaluation hatte.
- **Zeitdruck:** Ich habe die Probanden bei der Erstellung nicht explizit auf die Zeitmessung hingewiesen. Stattdessen habe ich die Stoppuhr offen vor mir liegen gehabt und offen betätigt. Die Probanden haben diese Messung allerdings unterschiedlich schnell bemerkt. Diese Vorgehensweise ist jedoch rückblickend als problematisch einzustufen, da das Gefühl, dass die Zeit genommen wird, ein Bias bei den Probanden bildet. Die Zeitmessung hätte entweder verdeckt vorgenommen oder explizit angesagt werden müssen.
- **Between-Groups-Ansatz:** Die Kontrollgruppe und die Plugin-Gruppe waren voneinander getrennt. Dies war notwendig, da beide Aufgaben einander sehr stark beschleunigen würden, würden sie direkt nacheinander von denselben Personen ausgeführt. Der Nachteil des Between-Groups-Ansatzes ist allerdings, dass die Gruppen – gerade bei kleinen Stichproben – nicht homogen sind. Diese Heterogenität ist bereits in den Ergebnissen des Demographiefragebogens ersichtlich, aber nicht auf ihn beschränkt. Es konnte nicht sichergestellt werden, dass in beiden Gruppen gleich viele Personen sind, die beispielsweise eine besonders schnelle Auffassunggabe haben, oder in Prüfungssituationen sehr nervös reagieren.
- **Eigene Arbeit evaluiert:** Da wir unsere eigenen Arbeiten evaluiert und dies den Probanden auch deutlich gemacht haben, waren die Probanden prinzipiell wohlwollender gegenüber den Plugins eingestellt. Dies stellt ein Bias dar. Deutlich

wurde dieser Umstand beispielsweise daran, dass Probanden sich entschuldigten, wenn sie etwas kritisierten. Auch der Hinweis, dass wir an Kritik interessiert seien, konnte nicht verhindern, dass die Probanden ihre Kritik deutlich zurückhaltender vorbrachten als sie es in einer anonymisierten Testsituation getan hätten.

8. Fazit

Zu Beginn der Arbeit stand die Vision beschrieben, einzelne Tests mit Hilfe von Crowdsourcing auszulagern. Im Verlauf der Arbeit habe ich einige Erkenntnisse und Eindrücke zu der Frage gewonnen, ob dies effizient möglich ist. Diese möchte ich zunächst erläutern, bevor ich einen Ausblick gebe, wohin die weitere Entwicklung meiner Meinung nach gehen sollte.

8.1. Ergebnisse

Die Evaluation hat gezeigt, dass das Erstellen von Testaufträgen bei Unit-Tests in einfachen Fällen zeitlich mit der Erstellung von Testfällen konkurrieren kann. Meiner Einschätzung nach dürfte der Aufwand für die Erstellung der Testfälle stärker steigen als der der Testaufträge. Dies rührt daher, dass die Erstellung von Datenbankmocks und Eingabedatenstrukturen verhältnismäßig aufwändig ist, während die Beschreibung im Testauftrag zu einem guten Teil bereits in Form von Javadoc vorliegen sollte. Diese Einschätzung müsste allerdings, wie im Vorkapitel erwähnt, erst noch durch Versuche bestätigt werden.

Ein sichereres Anwendungsgebiet sehe ich im Bereich der GUI-Tests. Wie sich in [Eiteljörge2014] gezeigt hat, erfordert selbst die Bearbeitung eines Testauftrags, der weniger Inhalt hat als die in meiner Evaluation erstellten Testaufträge, bereits einen zweistelligen Aufwand in Minuten. Der Autor versicherte mir gegenüber, dass die Arbeit auch für einen geübten Testentwickler noch mindestens eine hohe einstellige Minutenzahl erfordere. Die Erstellung des Testauftrags wäre in einem solchen Fall mehr als drei mal so schnell wie die Erstellung des Tests. Bei komplexeren Fällen könnte dieser Faktor noch weiter steigen. Ein weiterer Vorteil bei GUI-Tests ist, dass die Funktion der GUI in aller Regel in Form des Benutzerhandbuchs bereits dokumentiert ist und man die Crowd so direkt gegen diese Anforderungen testen lassen kann. Wenn das Benutzerhandbuch im Vorfeld validiert worden ist, können sogar Fehler gefunden werden, in denen gegen die eigentlichen Anforderungen verstoßen wurde. Diese Fehlerart machte in meiner Evaluation etwa die Hälfte aller Fehler aus. Auch wenn das frühe Anfertigen der Benutzerdokumentation und ihre Validierung eine gewisse Umstellung und einen gewissen Zusatzaufwand bedeuten, so bietet das Crowdsourcing doch an dieser Stelle eine Chance, die Software unabhängig vom Entwickler zu verifizieren und diesen Fehlertypus zu bekämpfen.

Der Nutzen der einzelnen Heuristiken ließ sich im Rahmen dieser Arbeit leider nicht zufriedenstellend bestimmen. In der Evaluation wurde nur eine einfache, aber mächtige Heuristik wirklich geprüft, nämlich die Nutzung von zuvor erstellten Einträgen. Das

Verhalten einer Methode oder die E-Mail-Adresse eines Entwicklers ändern sich nicht allzu oft und dass diese Felder beim zweiten Aufruf bereits ausgefüllt waren, wurde sehr positiv bewertet. Die anderen implementierten Heuristiken waren entweder aufgrund des vorgegebenen Szenarios (z.B. „Current File“ oder „Documents in Project“) oder aber aufgrund der Einfachheit der Aufgabe („Relevance Neighbors“) nicht objektiv bewertbar. Zwar trugen diese Heuristiken auch zum Gesamteindruck von TaskElf bei, jedoch steht der Beweis aus, dass sie auch im Feldeinsatz einen positiven Beitrag leisten können und ihre Einsatzszenarien nicht konstruiert sind. Dasselbe gilt für die nicht implementierten Heuristiken.

8.2. Nächste Schritte

Bevor das Crowdsourcing von Tests in einem realen Kontext genutzt werden kann, müssen vorher noch einige Schritte unternommen werden. Zunächst einmal müssen die genauen Abläufe erarbeitet und eine zu diesen passende Crowdsourcing Plattform entworfen werden. Zu den zu klärenden Fragen gehört zum Beispiel, ob Lösungen abgelehnt werden können, oder ob mehr als ein Tester einen Testauftrag übernehmen kann. Ist dies geschehen, müssen die Plugins auf beiden Seiten mit der Plattform verbunden und gegebenenfalls an die Rahmenbedingungen angepasst werden. Dies würde erstmal eine grundsätzliche Einsatzbereitschaft herstellen und könnte eine Grundlage für eine eventuelle Kooperation mit einer Entwicklungsfirma für einen Testbetrieb sein. Dabei wäre allerdings noch zu klären, wer für die Erfüllung der Testaufträge sorgen würde.

Auf Seiten von TaskElf sollten zuvor noch die in Kapitel 4 vorgeschlagenen sowie in der Evaluation angeregten, Erweiterungen vorgenommen werden. Die Implementierung weiterer Heuristiken könnte dann während des Testbetriebs geschehen. Als Anhalt für ihre Güte könnten ihre Gewichte ausgewertet werden, sodass kein zusätzlicher Aufwand entstände, um die Qualität einer neuen Heuristik zu bewerten.

Dies alles wäre ein beträchtlicher Aufwand, aber ich denke, dass er es wert wäre und crowdsourcing-basiertes Testen, wie es in dieser Arbeit beschrieben wurde, einen deutlichen Einfluss auf das Outsourcing des Testens haben könnte.

A. Beispiel RelevanceNeighbors

An dieser Stelle gebe ich ein ausführliches Beispiel für den Algorithmus der relevanten Nachbarn. Dazu werde ich beschreiben, wie der Algorithmus auf dem Graphen in Abbildung 6.1 abläuft.

Eingabeknoten ist der Knoten C. Nach der Initialisierung ist die Liste der Nachbarn leer und in der Liste der Wolke befindet sich nur Knoten C, mit Grad 1.

Wolke:

Knoten	Grad
C	1

Nachbarn:

Knoten	Grad	Nähe
-	-	-

Als nächstes werden die Nachbarn von C zur Nachbarliste hinzugefügt. Da sie neu hinzugefügt werden, bekommen sie den um Eins inkrementierten Grad von C zugewiesen. Ihre Nähewerte werden berechnet, indem jeweils das Gewicht der Kante von C zum neuen Nachbarknoten zum durch α verminderten Gewicht des entgegengesetzten Link addiert wird. Für α verwende ich im Beispiel 0,5. Eingehende Kanten werden also mit halbiertem Gewicht betrachtet. Normalerweise würde der Wert noch durch den Grad von C verringert werden, da C aber als zentraler Knoten den Grad 1 hat und mit dem Kehrwert des Grades multipliziert wird, hat dies in der ersten Iteration keine Auswirkung.

Wolke:

Knoten	Grad
C	1

Nachbarn:

Knoten	Grad	Nähe
A	2	$0{,}5 \cdot 11 = 5{,}5$
B	2	$4 + 0{,}5 \cdot 6 = 7$
E	2	4
F	2	1

Nun wird der Knoten aus den Nachbarn gewählt, der die höchste Nähe zur Wolke hat. Dies ist Knoten B. Jetzt muss geprüft werden, ob B mindestens den Nähewert MIN_CLOSENESS hat. Im Beispiel nutze ich den Wert 5. B ist also nah genug an der Wolke, um zu ihr hinzugefügt zu werden. Es werden nun alle Nachbarn von B betrachtet, die außerhalb der Wolke liegen, genauso wie es in der vorherigen Iteration mit den Nachbarn von C gemacht wurde. Der einzige neue Nachbar ist Knoten D. Ein- und ausgehende Kanten werden gewertet wie zuvor. D hätte damit eine Nähe von 15. Nun wird diese aber noch mit dem Kehrwert des Grades von B multipliziert und sinkt auf 7,5. Damit hat D einen höheren Nähewert als zuvor B. Dies macht deutlich, dass es sich bei Nähe nicht um die Nähe zu C handelt, sondern um die Nähe zur Wolke. Anschaulich bedeutet die Situation: Wenn man Datei B als relevant ansieht und dem

Tester vorlegt, dann sollte man ihm auch Datei D geben, da B ohne D nicht verständlich ist.

Wolke:

Knoten	Grad
C	1
B	2

Nachbarn:

Knoten	Grad	Nähe
A	2	0,5 · 11 = 5,5
D	3	15 · (1/2) = 7,5
E	2	4
F	2	1

Der Schritt wiederholt sich mit Knoten D als neuem Teil der Wolke. D ist mit F verknüpft, welches sich schon in der Liste der Nachbarn befindet. Die Nähe, die Knoten F durch Knoten D erhält, wird zur bisherigen Nähe von F hinzu addiert. Die größte Änderung zu den vorherigen Schritten stellt die Berechnung des neuen Grades von F dar. Da F sowohl über C als auch über D mit der Wolke benachbart ist, funktioniert ein einfaches Inkrementieren nicht: Bisher wurde der inkrementierte Wert von C (2) für F geführt. Andererseits ist F an D viel näher, was aber einen Grad von 4 bedeuten würde (3 + 1). Sinnvoller ist es, einen Wert dazwischen zu wählen. Ich bestimme den Anteil eines Knotens am Grad seines Nachbarn durch den Anteil, den der Knoten zur Nähe dieses Nachbarn beigetragen hat. Dies entspricht dem gewichteten Mittel der vorgeschlagenen (d.h. inkrementierten) Grade. In diesem Fall hat F durch C den Grad 2 als Vorschlag, wobei C 1 zur Nähe von F beigetragen hat. D schlägt 4 als Grad vor und hat 5 ((0,5 · 8 + 11) / 3) zur Nähe beigetragen. Der Gesamtgrad von F ist also 3,67 ((1 · 2 + 5 · 4) / 6).

Wolke:

Knoten	Grad
C	1
B	2
D	3

Nachbarn:

Knoten	Grad	Nähe
A	2	0,5 · 11 = 5,5
E	2	4
F	3,67	6

Als nächstes würden nun F und anschließend A der Wolke hinzugefügt werden. Danach würde der Algorithmus abbrechen, da E als nächster (und einziger) Nachbar den Threshold MIN_CLOSENESS nicht erreicht. Relevant wären in diesem Beispiel also alle Knoten außer E.

B. Evaluationsdokumente

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

Abbildung B.1.: Demographiefragebogen

Aufgabe:

Die Funktion "public Integer getGCD(int firstNumber, int secondNumber)" soll den größten, von 1 verschiedenen, gemeinsamen Teiler zweier mind. 5-stelliger Zahlen zurückgeben. Dies ist der größte Integerwert durch den man beide Zahlen teilen kann und eine ganze Zahl als Ergebnis erhält, die nicht 1 ist. Sollte es keine solche geben, so soll null zurückgegeben werden. Wenn das Ergebnis eine der beiden Zahlen selbst ist, so wird es negiert.

Hinweis: Der Operator Modulo berechnet den Rest einer ganzzahligen Division zwischen zwei Zahlen. In Java ist die Syntax hierfür a % b. Bsp: 15 % 4 = 3

Abbildung B.2.: Die Programmieraufgabe

Abschlussinterview Task-Elf

1. Es war eine angenehme Erfahrung: 1-5
2. Das Auslagern von Tests kann meine Arbeit beschleunigen: 1-5
3. Ich hatte das Gefühl zu wissen, was ich tue: 1-5
4. Würden Sie TaskElf im Firmenumfeld nutzen? Warum?
5. Würden Sie TaskElf privat nutzen? Warum?
6. Was hat Ihnen gefallen?
7. Was hat Ihnen nicht gefallen?
8. Haben Sie sofort verstanden, was Ihre Aufgabe ist?
9. Wie schwierig war die Lösung der Programmieraufgabe?
10. Was halten Sie allgemein von Crowdsourcing?

Abbildung B.3.: Die Interviewfragen

Ergebnisse

	Demographie		
Teilnehmer	**Alter**	**Beruf**	~~**Semester**~~
1	24	Informatik Student	~~8~~
2	28	Informatik Student	~~12~~
3	28	Inf. WiMi	~~5~~
4	30	Inf. WiMi	
5	27	Informatik Student	~~12~~
6	29	Informatik Student	
7	26	Informatik Student	~~4~~
8	27	Inf. WiMi	
9	25	Informatik Student	~~4~~
10	54	Softwareentwickler	

	Erfahrungen				
Teilnehmer	**Programmieren**	**Testen**	**Eclipse**	**Unit-Tests**	**Selenium**
1	5	4	4	4	1
2	4	2	3	2	1
3	4	3	3	4	2
4	3	3	5	3	2
5	4	3	4	3	2
6	5	4	4	4	1
7	4	2	4	2	1
8	3	3	3	2	2
9	4	3	4	2	1
10	5	3	2	3	1

	Versuch				
Teilnehmer	**Plugin**	**Zeit 1 (min)***	**Zeit 2 (min)***	**Güte 1**	**Güte 2**
1	true	06:30	01:35	kk	kk
2	false	12:35	02:29	kk	kk
3	false	07:58	02:15	kk	kk
4	true	09:11	01:35	kk	fF
5	false	04:20	01:05	kk	kk
6	true	04:58	01:33	ak	kk
7	true	03:59	02:18	aa	sk
8	true	12:01	04:56	kf	kk
9	false	06:59	03:02	aa	ka
10	false	07:12	01:42	fk	kk

*15 Sekunden Startzeit des Plugins bei Pluginprobanden abgezogen

Legende Güte:

k	Test(auftrag) ist konsitent mit den Anforderungen und sinnvoll
f	Test(auftrag) ist inkonsistent zu den Anforderungen und der Implementierung
F	Test(auftrag) ist aus Fehlverständnis der Programmiersprache falsch
a	Test(auftrag) ist inkonsistent zu den Anforderungen, aber passt zur Implementierung
s	Test(auftrag) ist nicht sinnvoll

Page 1

Abbildung B.4.: Die Ergebnisse des quantitativen Versuchsteils, Seite 1

Ergebnisse

	Interview Likert			
Teilnehmer	**angenehm**	**beschleunigen**	**Sicherheit**	**Aufgabe**
1	4	3	2/4	2
2				3
3				2
4	4	4	5	2
5				2
6	3	4	4	1
7	4	5	4	3
8	3	4	3	1
9				2
10				1

Page 2

Abbildung B.5.: Die Ergebnisse des quantitativen Versuchsteils, Seite 2

4. Würden Sie TaskElf im Firmenumfeld nutzen? Warum?

- Ja, wenn es Arbeit einspart
- Kommt auf die Firma an, Vertraulichkeit des Codes, Qualität der Crowd, Preise, Testwilligkeit der Entwickler
- Ja, nimmt Arbeit ab, muss Tests nicht selbst schreiben; von Außen kann anderer Ansatz kommen -> es werden evtl. mehr Fehler entdeckt
- vermutlich schon, aber keine Erfahrung mit großen Projekten; spart Tester
- Ja, prinzipiell schon

6. Was hat Ihnen gefallen?

- Einzelne Schritte; Assistenz; Führung
- Führung; Examples; Vorausfüllung; Hilfefeld
- die Idee; Vorschlagsliste im ersten Tab
- viel vorausgefüllt; veränderte Gewichte (Tab 1), Drag 'n Drop
- erster Tab; Hilfetexte; Eingeben + Enter -> nächste Eingabe

7. Was hat Ihnen nicht gefallen?

- Sprung des Testfalls in Tab 1 weiter nach oben in der Liste [veränderliche Gewichte – Suchfunktion oder alphabetische Sortierung vorgeschlagen]
- Inkonsistenz (Freitextfeld neben den Listen nicht als solches erkennbar); if-then Form beim ersten Mal unintuitiv; Eingabefelder wirken zu klein, bevor man verstanden hat, dass nur einzelne, kurze Listeneinträge erstellt werden sollen; erster Tab erschlägt einen etwas; veränderliche Gewichte in Tab 1
- sichtbar machen, woher die Vorschläge kommen (Tab 1)
- Tab 1 verlassbar, auch wenn keine Methode gewählt ist (Hinweis wäre gut, vllt. ne Warnung, wenn es leer ist und eigener Eintrag für "ganze Klasse testen"); Hilfeboxen auffälliger
- Ein Rechtschreibfehler + ein Grammatikfehler; unklar was "Method Behavior" genau ist (vllt. Beschriftung ändern): Granulat unklar – heißt "keine Sonderfälle" nun garkeine Sonderfälle, oder nur keine Sonderfälle, die nicht in den Anforderungen stehen?

8. Haben Sie sofort verstanden, was Ihre Aufgabe ist?

- Ja
- Joar
- Ja, hätte etwas besser sein können, ähnlich wie Javadoc (doppelte Arbeit) [Nutzer hat viel Javadoc geschrieben, aber durch einen Bug wurde nur ein Teil davon automatisch in den Testauftrag übernommen]
- Ja
- Ja

Abbildung B.6.: Die Antworten auf die Interviewfragen, Seite 1

10. Was halten Sie allgemein von Crowdsourcing?

- Viel
- Keine Erfahrung, Wikipedia funktioniert, aber Qualität kommt durch Wenige; Qualität kritisch sonst aber Gewinn
- finde ich gut; von Außen kommen andere Ideen
- noch nie genutzt; nützlich, mehr Leute sehen mehr -> unbekanntes wird entdeckt
- gute Sache; Qualität könnte Problem sein; man bekommt Dinge an die man selbst nicht denkt

Abbildung B.7.: Die Antworten auf die Interviewfragen, Seite 2

Literaturverzeichnis

[Antunes2012] Antunes, B., Cordeiro, J., Gomes, P.,
An Approach to Context-Based Recommendation in Software Development,
RecSys'12, 2012

[Cheng2007] Cheng, B.H.C., Atlee, J.M.,
Research Directions in Requirements Engineering
Future of Software Engineering, 2007

[Cockburn2000] Cockburn, A., Highsmith, J.
Agile Software Development, The Agile Software Development Series, S.84f Alistair Cockburn, 2000

[Daimler1999] DaimlerChrysler
Beitrag
Dagstuhl-Seminar „Requirements Capture, Documentation and Validation“, 1998

[Dijkstra1959] Dijkstra, E. W.,
A note on two problems in connexion with graphs Numerische Mathematik 1, 1959, S.269 ff.

[Dolstra2012] Dolstra, E., Vliegendhart, R., Pouwelse, J.,
Crowdsourcing GUI Tests ICSE'12, 2012

[Eiteljörge2014] Eiteljörge, F.,
Konzept und Implementierung einer aufgabenfokussierten Testumgebung für eine Crowdsourcing Plattform
Institut für Praktische Informatik, Leibniz Universität Hannover, 2014

[Haberl2011] Haberl, P., Spillner, A., Vosseberg, K., Winter, M.,
Umfrage 2011: Softwaretest in der Praxis, 1. Auflage 2012, überarb.
dpunkt.verlag, 2012

[Heinemann2011] Heinemann, L., Hummel, B.,
Recommending API Methods Based on Identifier Contexts,
SUITE'11, 2011

[Howe2006] Howe, J.,
Crowdsourcing: A Definition
`http://www.crowdsourcing.com/cs/2006/06/crowdsourcing_a.html`, 2006, Stand 1. Februar 2014

[Khatib2011] Khatib, F., DiMaio F., Foldit Contenders Group, Foldit Void Crushers

Group et al.,
Crystal structure of a monomeric retroviral protease solved by protein folding game players
Nature Structural & Molecular Biology, 2011

[Kersten2006] Kersten, M., Murphy, G.C.,
Using Task Context to Improve Programmer Productivity,
SIGSOFT'06, 2006

[LaToza2006] LaToza, T.D., Venolia, G., DeLine, R.,
Maintaining Mental Models: A Study of Developer Work Habits
ICSE'06, 2006

[Pham2013] Pham, R., Singer, L., Liskin, O., Filho, F.F., Schneider, K.,
textitCreating a Shared Understanding of Testing Culture on a Social Coding Site
ICSE'13, 2013

[Rahman2013] Rahman, M.M., Yeasmin, S., Roy, C.K.,
textitAn IDE-Based Context-Aware Meta Search Engine
WCRE 2013, 2013

[Riungu2010] Riungu, L.M., Taipale, O., Smolander, K.,
Software Testing as an Online Service: Observations from Practice
ICST 2010, 2010

[Robertson1999] Robertson, S., Robertson, J.,
Mastering the Requirements Process
Addison-Wesley Longman, 1999

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

[Shneiderman2010] Plaisant, C., Shneiderman, B.,
Designing the User Interface: Strategies for Effective Human-Computer Interaction: Fifth Edition
Addison-Wesley, 2010

[Stoliar2013] Stoliar, Y.,
Entwurf und Entwicklung einer Ranking Website von Open Source Projekten für Crowdsourcing Zwecke
Institut für Praktische Informatik, Gottfried Wilhelm Leibniz Universität Hannover, 2013

[Todd1999] Todd, P.M., Gigerenzer, G.,
textitSimple Heuristics That Make Us Smart
Oxford University Press, 1999

[Wolter2013] Wolter, J.,

Crowdtesting: Anwendungsfälle und Grenzen einer neuen Methode
OBJEKTspectrum, Ausgabe Testing/2013, 2013

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. April 2014

_____

Benjamin Riedel