Analyse und Refactoring einer Software zur Unterstützung der psychologischen Diagnostik

Bachelorarbeit

von

Nils Kubera
5886728
Am Springintgut 9
21335 Lüneburg

26. Oktober 2010

Betreuer:
Dr. A. Schmolitzky
Prof. Dr. H. Züllighoven

Universität Hamburg
Department Informatik
Arbeitsbereich Softwaretechnik

Inhaltsverzeichnis

1 Einleitung 5

2 Fachliche Grundlagen 6
2.1 Psychologische Diagnostik . . . 6
2.2 Das e-paper-pencil-Verfahren (epp) . . . 6
2.2.1 Die Hardware: *Sahara Slate PC* . . . 7
2.2.2 Das Betriebssystem: *etos* . . . 7
2.2.3 Die Software: *QSheet* . . . 7
2.2.4 Probleme des Prototypen . . . 8

3 Softwaretechnische Grundlagen 10
3.1 Methoden und Ansätze . . . 10
3.1.1 Refactoring . . . 10
3.1.2 Werkzeug & Material-Ansatz (WAM) . . . 11
3.1.3 Agiles Vorgehen . . . 11
3.2 Werkzeuge zur Unterstützung der Analyse . . . 13
3.2.1 XRadar . . . 13
3.2.2 FindBugs . . . 13
3.3 Qualitätssichernde Werkzeuge . . . 13
3.3.1 Unit Tests / JUnit . . . 14
3.3.2 Jemmy . . . 14
3.3.3 Emma . . . 15

4 Analyse und Planung 16
4.1 Architektur . . . 16
4.1.1 Das Paket `qsheet.model` . . . 17
4.1.2 Das Paket `qsheet.pdf` . . . 18
4.1.3 Das Paket `qsheet.ui` . . . 18
4.2 Planung der Umsetzung . . . 18
4.2.1 Agile Praktiken . . . 21

5 Umsetzung und Ergebnisse 23
5.1 Refactoring . . . 23
5.1.1 Vorbereitung . . . 23

5.1.2 I. Iteration 24
5.1.3 Ergebnisse der I. Iteration: 26
5.1.4 II. Iteration 26
5.1.5 Ergebnisse der II. Iteration: 29
5.2 Performanzoptimierungen (III. Iteration) 29
5.2.1 Doppeleingabe bei Navigationsbuttons 30
5.2.2 Zeichengeschwindigkeit 32
5.2.3 Ergebnisse der III. Iteration: 33

6 Zusammenfassung und Ausblick **37**
6.1 Fazit der Methoden und Werkzeuge 37
6.2 Weitere Aufgaben im Projekt 38

Literaturverzeichnis **40**

1 Einleitung

Diese Arbeit beruht auf der Diplomarbeit "Der 'e-paper-pencil' Fragebogen. Äquivalenzüberprüfung einer Papier-Bleistift- und e-paper-pencil-Testung" von Ingmar Böschen aus dem Fachbereich Psychologie des Departements Psychologie der Universität Hamburg aus dem Jahr 2009 [Bös09].
Das in diesem Rahmen vorgestellte Testverfahren "e-paper-pencil" (epp) wurde von Böschen zum Einsatz in der psychologischen Diagnostik entwickelt. Es ähnelt einer Testung mit Papierfragebögen in höchstem Maße und weist dabei die Vorteile computergestützter Testmethoden auf. Umgesetzt wurde dieses Verfahren erstmals durch die speziell dafür entwickelte Software *QSheet*, die auf einem tragbaren Touch-Screen-Computer eingesetzt wurde. Im Feldeinsatz des Verfahrens zeigten sich Performanzschwächen im Prototypen der Software, die die angestrebte Äquivalenz zu Papierfragebögen störten. Zusätzlich sollten weitere Komponenten entwickelt werden, die den Funktionsumfang der Software erweitern sollten.
Im Verlauf dieser Arbeit wurde der Quelltext des Ausgangssystems mit Unterstützung verschiedener Open Source-Werkzeuge analysiert. Anhand der Ergebnisse der Analyse wurde festgestellt, dass die Quelltextbasis des Ausgangssystems in einem Zustand war, der die Erweiterungen und Verbesserungen stark behinderte. Um diese Situation zu beheben wurde ein Refactoring des Ausgangssystems durchgeführt. Anschließend konnten auf der neuen Quelltextbasis zwei Performanzprobleme der Software gelöst werden. Neben dem Refactoring wurden in der Planung und Entwicklung weitere Praktiken agiler Methoden eingesetzt.

Diese Ausarbeitung beginnt mit einer Beschreibung der fachlichen Hintergründe, die den Kontext in dem diese Arbeit durchgeführt wurde vertiefend erläutert. Darauf werden die softwaretechnischen Methoden und Werkzeuge beschrieben, die in der Analyse und dem Refactoring des Ausgangssystems eingesetzt wurden. Im folgenden Abschnitt werden die Ergebnisse der Analyse und die daraus resultierenden Planungen aufgeführt. Anschließend wird die praktische Umsetzung einzelner Schritte des Refactorings und der Performanzverbesserungen inklusive der daraus resultierenden Ergebnisse beschrieben. Als letztes wird ein Fazit über den Erfolg und Nutzen der eingesetzten Werkzeuge und Methoden gezogen und ein Ausblick auf die weiteren softwaretechnischen Aufgaben gegeben, die im Kontext der Arbeit bestehen.

2 Fachliche Grundlagen

In diesem Abschnitt wird der Kontext, in dem diese Arbeit durchgeführt wurde, erläutert.

2.1 Psychologische Diagnostik

Als psychologische Diagnostik wird das Erfassen und Auswerten psychologischer Daten einer Person bezeichnet. Eine in diesem Kontext befragte Person wird als DiagnostikantIn, der/die Befragende als DiagnostikerIn bezeichnet. Herkömmlich werden in diesem Bereich Papier-Bleistift-Fragebögen (paper-pencil) eingesetzt, die in zunehmendem Maße durch computergestützte Methoden ersetzt werden [Wei02, S. 93]. In der standardmäßigen computergestützten Testsituation sitzt der/die DiagnostikantIn vor einem stationären Computer mit Desktop-Monitor [WM03].

Der Übergang zur computergestützten Testdurchführung weist dabei diverse Vorteile auf, wie objektives Erfassen, einfache Kombination verschiedener psychologischer Tests und vor allem Zeiteinsparungen durch automatisierte Auswertungen [Bös09, S. 13]. Auf der anderen Seite konnte nachgewiesen werden, dass in der herkömmlichen computergestützten Diagnostik diverse Faktoren die Ergebnisse beeinflussen. Beispiele dafür können Eigenschaften des/der DiagnostikantIn sein, wie Computerängstlichkeit oder Erfahrung mit Computern [TTMW98], aber auch weniger offensichtliche Faktoren, wie die Wahl der Maus [PHW03] oder Design des Tests [KJVK07].

Dies ist vor allem problematisch, da die Tests regelmäßig in aufwändigen und teuren Verfahren genormt werden müssen. Durch die nachgewiesenen Äquivalenzunterschiede zwischen Papier-Bleistift-Tests und herkömmlichen computergestützten Tests müssen beide Versionen unabhängig voneinander genormt werden, wodurch sich Aufwand und Kosten verdoppeln.

2.2 Das e-paper-pencil-Verfahren (epp)

Das von Böschen vorgestellte e-paper-pencil-Verfahren wurde entwickelt, um die Vorteile der computergestützten Methode nutzen zu können, ohne dass die Tests erneut auf Normwerte geeicht werden müssen. Erreicht wird dies dadurch, dass die computergestützte Testdurchführung im höchsten Maße äquivalent zu der Testdurchführung mit Papier-Bleistift-Fragebögen ist. Dazu wird ein Touch-Screen Tablett-PC eingesetzt, der einen als PDF-Datei digitalisierten Papier-Fragebogen im Vollbildmodus anzeigt. Der Fragebogen wird dabei nicht modifiziert und bleibt in Layout und Schrift identisch mit der

Druckversion. Die Größe des Displays ist ebenfalls identisch mit der Größe des Fragebogens (meist DinA-4). Bedient wird das Gerät mit einem digitalen Stift. Im Testmodus gibt es ganz bewusst keine computertypischen Rückmeldungen, wie Positionskreuz des Stiftes, Meldefenster oder Eingabeunterstützung.
Das Verfahren wurde erstmals mit der im Folgenden beschriebenen Hard- und Software umgesetzt. In den bisherigen Feld- und Labortestreihen konnte Böschen nur marginale Unterschiede zwischen Papier-Bleistift- und epp-Testung feststellen. Weitere Studien zur Äquivalenz beider Versionen sind im Rahmen der Promotion Böschens in Planung.

2.2.1 Die Hardware: *Sahara Slate PC*

Die in der ersten Umsetzung eingesetzte Hardware ist der mobile Touch-Screen Tablet-PC *Sahara Slate PC i215*. Er besitzt einen 1,1 GHz Prozessor, 512MB Arbeitsspeicher und wiegt nach Modifikation etwa 1,3 Kilogramm. Der Rechner lässt sich über den Bildschirm mit einem digitalen Stift, über eine *WACOM*-Schnittstelle bedienen und entspricht in seinen Maßen etwa denen eines DinA-4-Blocks.

2.2.2 Das Betriebssystem: *etos*

Auf dem Gerät wird das Betriebssystem *etos* (electronical test-operatingsystem) eingesetzt. Dies ist ein von dem am Projekt mitwirkenden Fachinformatiker in Ausbildung Gerd Finke speziell für die Bedürfnisse des Verfahrens modifiziertes Gentoo Linux[1].

2.2.3 Die Software: *QSheet*

Die Software *QSheet* wurde von Böschen gemeinsam mit den Diplominformatikern Felix Tille und Tasso Griebel entwickelt. Der erste Prototyp wurde im Zeitraum von zwei Tagen mit einer höheren Priorität auf Funktionsumfang als auf Erweiterbarkeit geschrieben und setzt die Anforderungen des epp-Verfahren um.
Die Software bietet einem/einer DiagnostikerIn die Möglichkeiten, eine Befragung vorzubereiten und durchzuführen. Nach Starten der Software wird ein Vorbereitungsfenster angezeigt, aus dem sich unter anderem ein Fragebogen im PDF-Format einlesen lässt. Weiterhin lassen sich in diesem Modus verschiedene Einstellungen der Präsentation vornehmen. Diese können gemeinsam mit dem eingelesenen Dokument im eigenen QST-Format exportiert werden. Zu den Einstellungen gehört das Gruppieren von Seiten zu

[1] www.gentoo.org/

Seitengruppen, denen eine maximale Bearbeitungszeit zugewiesen werden kann. Außerdem lässt sich angeben, ob eine Seite die Navigation zur nächsten oder vorigen Seite zulässt. In der Papier-Bleistift-Testdurchführung würde dieses beispielsweise dadurch umgesetzt werden, dass der/die DiagnostikerIn dem/der DiagnostikantIn verbal vermittelt, nicht zurück zu blättern.
Aus dem Menü lässt sich die vorbereitete Befragung starten. Dieser Präsentationsmodus zeigt das PDF-Dokument seitenweise im Vollbildmodus. Der einzige Zusatz zu der grafischen Anzeige der Seite sind zwei Navigationsbuttons zum Vor- und Zurückblättern der Seite, die an beiden unteren Bildschirmecken eingeblendet werden. Auf den einzelnen Seiten des Fragebogens lassen sich mit der Maus beliebige Zeichen wie auf Papier malen, die direkt angezeigt werden. Im Einsatz auf dem Tablet-PC werden diese Mauseingaben durch den digitalen Stift getätigt. Läuft die angegebene Zeit einer Seitengruppe ab, bevor der/die DiagnostikantIn diese Seiten verlassen hat, springt die Software automatisch zur ersten Seite der nächsten Gruppe. Auch diese Funktion würde in der Papier-Bleistift-Version von dem/der DiagnostikerIn manuell getätigt werden, beispielsweise mit Hilfe einer Stoppuhr.

2.2.4 Probleme des Prototypen

Beim Feldeinsatz des Prototypen im Rahmen der Diplomarbeit Böschens traten zwar keine elementaren Fehler auf, die sofortiges Ändern verlangt hätten, er stellte jedoch im Präsentationsmodus einige Schwächen in der Performanz fest. Exemplarisch waren zwei Probleme: Die nächste Seite wurde gelegentlich, nach Klicken des Buttons zum Umblättern, zu langsam nachgeladen. Da aufgrund der Prinzipien des epp-Verfahrens keine Rückmeldung darüber erfolgte, dass der Klick auf den Umblättern-Button registriert wurde, war dem/der DiagnostikantIn nicht bewusst, dass das Wechseln der Seite in diesem Zeitraum intern ausgeführt wurde. Deshalb wurde von einigen DiagnostikantInnen angenommen, der Klick wäre nicht registriert worden und sie klickten erneut auf den Button zum Umblättern. Das System blätterte darauf korrekterweise zwei Seiten um, was nicht den Erwartungen des/der DiagnostikantIn entsprach. Ein anderes Problem bestand darin, dass die Linien beim Zeichnen mit durchschnittlicher Geschwindigkeit auf den Seiten merklich langsamer dargestellt wurden, als sie eingegeben wurden. Dadurch fühlte es sich an, als folge die Zeichnung dem Stift mit leichter Verzögerung. Eine geringe Verzögerung ist natürlich in einer derartigen digitalen Umsetzung zu einem gewissen Grad immer gegeben. Da bei Bleistiftzeichnungen auf Papier keine Verzögerungszeit zwischen Malen und Anzeigen existiert, sollte diese in der Umsetzung des epp-Verfahrens möglichst nicht zu bemerken

sein.
Zusätzlich sollte die Software weiterentwickelt werden. In Planung waren ergänzende Komponeten die eine teil-automatisierte Auswertung der bearbeiteten Fragebögen ermöglichen sollten. Dazu zählte ein Editor, in dem DiagnostikerInnen auf einem PDF-Fragebogen Variablenbereiche und Auswertungsformeln definieren können. Eine weitere Komponente sollte anhand dieser Informationen in der Lage sein die grafischen Eingaben in psychologische Variablen zu überführen. Da es aufgrund der Ungenauigkeit der Eingaben einiger DiagnostikantInnen oder anderer Einflüsse in dieser Phase leicht zu Fehlinterpretationen kommen kann, sollte diese Phase nur teil-automatisiert durchgeführt und manuell durch den/die DiagnostikerIn überprüft werden.
Um diese softwaretechnischen Aufgaben umzusetzen, wurde von Böschen eine Zusammenarbeit mit dem Arbeitsberich SWT der Universität Hamburg angeregt zu der diese Arbeit zu zählen ist. Als erster Schritt musste geklärt werden, in wie weit sich der bestehende Prototyp der Software für die geplante Weiterentwicklung eignet oder welche vorbereitenden Schritte notwendig sind, um diese durchführen zu können. Diese Frage sollte durch eine Analyse des Prototypen beantwortet werden.

3 Softwaretechnische Grundlagen

Um die Analyse und das darauf folgende Refactoring durchzuführen, wurden in dieser Arbeit unterstützende Werkzeuge und Methoden der Softwaretechnik eingesetzt. Diese werden in diesem Abschnitt kurz vorgestellt.

3.1 Methoden und Ansätze

Die folgenden beschriebenen softwaretechnischen Methoden und Ansätze hatten Einfluss auf den praktischen Teil dieser Arbeit oder wurden in diesem verwendet.

3.1.1 Refactoring

Als Refactoring wird die Veränderung an der internen Struktur einer Software bezeichnet, ohne dabei ihr sichtbares Verhalten zu ändern. Ziel ist es die interne Struktur leichter zu verstehen und diese mit geringerem Aufwand modifizieren zu können [Fow07, S. 53]. Refactoring grenzt sich begrifflich von Quelltextänderungen mit anderen Motivationen ab, wie beispielsweise einer Performanzverbesserung.
Beim Entwickeln oder durch Anpassen einer Software entstehen häufig so genannte "schlechte Gerüche" (Original: Bad Smells). Dabei handelt es sich um Stellen, die das intuitive oder schnelle Verständnis des Quelltextes erschweren, beispielsweise Methoden, die zu lang sind, um auf einen Blick erfasst zu werden. Für die meisten schlechten Gerüche existieren empfohlene Muster, die beschreiben, mit welchem Vorgehen sich diese optimal in eine besser zu erfassende Form umwandeln lassen. Das Einhalten dieser Vorgehensweisen erhöht die Sicherheit, keine Änderungen am äußeren Verhalten und damit auch keine Fehler durch das Refactoring zu produzieren. Ein Katalog solcher Muster findet sich in [Fow07]. Eine grundlegende Regel bei Refactorings besagt, in kleinen Schritten vorzugehen. Das bedeutet, dass kleinstmögliche Änderungen vorgenommen werden, die ausführbaren Quelltext erzeugen und jede dieser Änderungen durch Testdurchläufe abgesichert wird. Diese Tests überprüfen das äußere Verhalten auf Veränderungen und erhöhen dadurch die Sicherheit diese rechtzeitig zu bemerken. Manche dieser Muster lassen sich von Entwicklungsumgebungen, wie der Eclipse IDE[2] für einige Programmiersprachen automatisiert durchführen, beispielsweise das Umbenennen von Methoden und Klassen.
Neben einer Quelltextbasis, die Entwicklern die Möglichkeit gibt Modifikationen schneller und einfacher durchzuführen, bringt ein Refactoring häufig auch weitere Vorteile mit sich:

[2] www.eclipse.org

Die Person, die das Refactoring durchführt, setzt sich notwendigerweise intensiv mit dem Quelltext auseinander, den sie überarbeitet. Ist sie selbst nicht dessen AutorIn, oder liegt die Entwicklung eine längere Zeit zurück, kann durch das Refactoring der Zugang zur Funktionsweise der Software stark verbessert werden. Auch potentielle Fehlerquellen lassen sich auf diesem Weg aufspüren [Fow07].

3.1.2 Werkzeug & Material-Ansatz (WAM)

Der Werkzeug & Material-Ansatz (WAM) ist ein Ansatz der objektorientierten Softwareentwicklung, "die Gegenstände und Konzepte des Anwendungsbereichs als Grundlagen des softwaretechnischen Modells zu nehmen" [Zü98, S. 4]. Hierzu werden auch jenseits der Softwaretechnik verwendete Metaphern in die Begriffswelt der Softwareentwicklung übertragen. Dies kann die Kommunikation zwischen EntwicklerInnen und AnwenderInnen aufgrund der gemeinsamen Basis vereinfachen und zwischen EntwicklerInnen aufgrund gemeinsam bekannter Konzepte.
Kern des Ansatzes ist es die speziellen Entwurfsmuster im Anwendungsbereichs auszumachen und in der Software zu modellieren. Die wichtigsten Elemente sind dabei: Materialien, also "Gegenstände die [...] Teil eines Arbeitsergebnisses werden" [Zü98, S. 86] und Werkzeuge, "Gegenstände, [die] Materialen verändern und sondieren können" [Zü98, S. 84].

3.1.3 Agiles Vorgehen

Agilität ist der Oberbegriff einer Denkweise in der Softwareentwicklung, die erstmals 2001 im agilen Manifest[3] zusammengefasst wurde. In diesem wird ein Wertesystem formuliert, das Menschen und Zusammenhänge über Prozesse und Werkzeuge, lauffähige Software über umfangreiche Dokumentation, Zusammenarbeiten mit Auftraggebern über Vertragsverhandlungen und Reagieren auf Änderungen über Befolgen eines Plans stellt. Basierend auf diesem Wertesystem existieren verschiedene Methoden, wie Extreme Programming (XP) [BA04] oder Crystal [Coc07], die diese Ideen durch Maxime und Praktiken für die Softwareentwicklung konkretisieren [HRS09]. Keine der agilen Methoden soll dabei mit allen ihren Praktiken als die beste Lösung für jedes Softwareprojekt gesehen werden. Welche Praktiken eingesetzt werden, sollte mit Blick auf die aktuelle Situation entschieden werden. Ändert sich diese, ist es durchaus im Sinn eines agilen Vorgehens,

[3] www.agilemanifesto.org

auch die eingesetzten Praktiken anzupassen [BA04, S. 35].
Einige der Praktiken, die in dieser Arbeit verwendet wurden, sind Folgende:

- **Iterativ inkrementelle Entwicklung** - Im Gegensatz zu klassischen Vorgehensweisen der Softwareentwicklung, wie dem "Wasserfallmodell", in dem alle Schritte, wie Analyse, Entwurf, Implementation, Tests hintereinander ausgeführt werden, wird in agilen Methoden eine iterativ inkrementelle Entwicklung eingesetzt. Dies bedeutet, dass die gleichen Schritte in möglichst kurzen Zyklen (iterativ) eingebunden werden, durch die der Umfang der Software und ihr Wert für den/die KundIn stetig wächst (inkrementell). Dadurch lässt sich in jedem Teilschritt auf Änderungen, Missverständnisse oder Fehler reagieren und dadurch die Gefahr, ein ganzes Projekt zu entwickeln, das dem/der KundIn keinen Nutzen bringt, minimieren [Coc07, S. 72-73] [BA04, S. 94].

- **"Customer Involvement"** - *(dt.: Kundeneinbezogenheit)* Ein wichtiges Element agiler Methoden ist es, die KundInnen, also Menschen, deren Leben und Arbeit direkt von der Entwicklung betroffen sind, in den Prozess einzubeziehen. In XP wird sogar empfohlen, dass KundInnen Teil des Entwicklungsteams werden und in jeder Besprechung anwesend sein sollten [BA04, S. 61]. Im Kontext dieser Arbeit sind potentielle KundInnen PsychologInnen als DiagnostikerInnen und im Prinzip alle Menschen, die in der Lage sind mit Stift und Papier einen Testbogen auszufüllen, als DiagnostikantInnen.

- **Regelmäßiges Testen** - Beck empfiehlt Tests früh, häufig und automatisiert ("Early, often, automated") durchzuführen [BA04, S. 97]. Dadurch lassen sich Fehler rechtzeitig entdecken, bevor sie sich im Quelltext so stark verfestigt haben, dass sie nur schwer zu beheben sind. Gleichzeitig ist bei regelmäßigen Tests, die letzte Änderung präsenter, wodurch sich die Fehlerquelle leichter finden lässt. Um die Tests früh durchzuführen eignet sich am besten eine testgetriebene Entwicklung [Lin05, S. 11]. Eine ausreichende Testhäufigkeit kann dadurch erreicht werden, sie als Teilschritt in der iterativ inkrementellen Entwicklung einzuplanen (Integrationstests) oder sie nach jeder lauffähigen Änderung durchzuführen (Unit Tests). Zur Automatisierung eignen sich Unit Test-Frameworks wie JUnit.

- **Refactoring** gehört ebenfalls zu den Praktiken agiler Methoden und wird beispielsweise in XP regelmäßig angewendet, um den Quelltext kontinuierlich zu verbessern und gleichzeitig leicht les- und wartbar zu halten [HRS09, S. 86].

3.2 Werkzeuge zur Unterstützung der Analyse

In diesem Abschnitt werden die Open Source-Werkzeuge beschrieben, die in der Analyse des Ausgangssystems eingesetzt wurden. Sie sollten die manuelle Bewertung der Eigenschaften des Systems durch automatisch generierte Ergebnisse unterstützen.

3.2.1 XRadar

XRadar[4] ist ein Werkzeug, das Metriken erstellt, anhand derer sich die Qualität einer Software bewerten lässt. Dazu werden verschiedene Open Source-Analysewerkzeuge, wie Checkstyle[5], ckjm[6], FindBugs oder JDepend[7] eingesetzt. Die Metriken werden in die Bereiche Architektur, Design, Quelltext-Quallität und Unit-Test-Suite zusammengefasst und lassen sich als Diagramme und Tabellen in einem erstellten HTML-Dokument betrachten. Um ein Projekt von XRadar analysieren zu lassen, müssen Variablen in verschiedenen Konfigurationsdateien und ein ANT[8]-Script auf die Eigenschaften des Projekts angepasst werden.

3.2.2 FindBugs

Das Open Source-Werkzeug FindBugs[9] wurde entwickelt, um Programmierfehler in Java-Programmen aufzuspüren. Es durchsucht die Klassen eines Projekts nach etwa 300 bekannten Fehlermustern und bietet Hinweise darauf, wie ein gefundener Fehler behoben werden kann [APM+07]. In [CHH+06] wurde ermittelt, dass FindBugs etwa jede 800-2400. Zeile in einem typischen Java-Programm, bei der es sich nicht um kommentierenden Quelltext handelt, als fehlerhaft meldet. Der ermittelte Anteil an Fehlermeldungen, bei denen es sich nicht um wirkliche Programmierfehler handelt, ist im Normalfall weit geringer als 50%.

3.3 Qualitätssichernde Werkzeuge

Die folgenden Werkzeuge wurden eingesetzt, um die Veränderungen am Quelltext gegen Fehler abzusichern und zur Qualitätssicherung des Programms.

[4] xradar.sourceforge.net
[5] checkstyle.sourceforge.net
[6] www.spinellis.gr/sw/ckjm/
[7] www.clarkware.com/software/JDepend.html
[8] ant.apache.org/
[9] findbugs.sourceforge.net

3.3.1 Unit Tests / JUnit

Unit Tests sind spezielle Tests, die sich im Gegensatz zu anderen Arten dadurch auszeichnen, dass sie nicht das gesamte System, sondern die Funktionalität einzelner Bestandteile, genannt Einheiten (Unit), testen. Bei objektorientierten Sprachen entspricht die getestete Einheit meist einer Klasse [Lin05, S. 5].
In der Entwicklung von Software ist es wichtig Unit Tests automatisch verwenden zu können, da sie im Vergleich zu manuellen Tests meist gründlicher und vor allem deutlich zeitsparender sind. Dies ist wichtig, da oft mehrmals am Tag auf potentielle Fehler getestet werden sollte. Speziell im Refactoringprozess ist es wichtig automatische Tests zu haben, da nach jedem Schritt ein vollständiger Testdurchlauf verlangt wird.
Für die Programmiersprache Java wurde von Beck und Gamma das Open Source-Framework JUnit[10] zur Automatisierung von Unit Tests entwickelt. Die Programmiersprache des Frameworks entspricht bei JUnit der zu testenden Sprache, was gemeinsame Verwaltung erleichtert. Einzelne JUnit-Tests lassen sich in eigene Klassen auslagern, wodurch der Quelltext der Anwendung vom Quelltext der Tests getrennt wird. Durch Einhalten von Namenskonventionen lassen sich beide Teile einander wieder leicht zuordnen. Zusätzlich können einzelne Unit Tests in Testsuiten zusammengefasst werden, was den Vorteil hat, dass sich mit einem einzelnen Durchlauf die Auswirkungen der Änderung einer Klasse auf das gesamte System testen lassen, sowie mit unterstützenden Werkzeugen, wie Emma die Testabdeckung [Lin05, S. 21].

3.3.2 Jemmy

Jemmy[11] ist eine Java Bibliothek unter der Sun Public License zum Testen von Anwendungen des Java GUI-Frameworks Swing. Es wurde ursprünglich als Modul für die NetBeans IDE entwickelt, lässt sich allerdings auch außerhalb von Netbeans verwenden. Es stellt dem Entwickler Methoden bereit, die das Testen der grafischen Benutzerschnittstelle unterstützen. Darunter fallen das Aufspüren und Lokalisieren einzelner GUI-Komponenten und Fenster, oder das Simulieren von Ereignissen, wie Klicken eines Buttons, durch einen AWT-Roboter. Im Vergleich zu jfcUnit, einem anderen populären Werkzeug zum Testen von Swing-Anwendungen, empfiehlt Link Jemmy, da es in der Verwendung einfacher erscheint [Lin05, S. 309].

[10] www.junit.org
[11] jemmy.netbeans.org

3.3.3 Emma

Emma[12] ist ein in Java für Java-Projekte geschriebenes Open Source-Werkzeug, mit dem sich die Abdeckung des Quelltextes durch eine ausführbare Applikation bestimmen lässt. Diese Informationen lassen sich dabei prozentual für das gesamte System oder kleinere Abstraktionseinheiten, wie Pakete, Klassen oder Methoden anzeigen. Durch grafische Hervorhebung (Grün für ausgeführte, Rot für nicht ausgeführte Zeilen) lässt sich einfach bestimmen, welche Quelltextzeilen von der Applikation nicht ausgeführt werden. Ein wichtiges Einsatzgebiet für Werkzeuge, die die Quelltext-Abdeckung bestimmen, ist die Überprüfung der Testabdeckung. Neben den Vorteilen einer hohen Testabdeckung in der Entwicklung, ist sie speziell für ein Refactoring sehr wichtig, da mit ihr die Wahrscheinlichkeit steigt, dass eine Änderung am äußeren Verhalten der Software rechtzeitig bemerkt wird.

[12] emma.sourceforge.net

4 Analyse und Planung

Um die gewünschten Modifikationen an der Software durchzuführen war es grundlegend, Architektur und Zustand des Quelltextes des bestehenden Prototypen zu analysieren. Dieser Schritt sollte als Basis einer fundierten Entscheidung über das weitere Vorgehen dienen.
Dazu habe ich den Quelltext ausführlich gelesen und manuell UML-Klassendiagramme der einzelnen Pakete, mit Hilfe des Open Source-Werkzeugs Graphor[13], erstellt. Zusätzlich wurden unterstützende Werkzeuge benutzt, mit denen sich eine automatisch Analyse der Software durchführen ließ. Das Einlesen in den Quelltext wurde durch die geringe Quantität an Javadoc-Kommentaren stark behindert; nur etwa 7% der Klassen und Methoden des Ausgangssystems waren dokumentiert.
Um von XRadar erstellte Analysen zu erhalten, mussten unterschiedliche Script-Dateien an verschiedenen Stellen auf das Projekt angepasst werden. Dafür habe ich mehr Einarbeitungszeit als geplant benötigt. Im Endeffekt erwies sich der Einsatz von XRadar jedoch als vorteilhaft. Anhand der als HTML-Dokument erstellten grafische Ausgabe hatte ich viele Daten über die Software, wie beispielsweise durchschnittliche Größe der Klassen eines Pakets, und die Ergebnisse verschiedener Metriken der Softwarequalität übersichtlich und verständlich zur Verfügung. (Siehe Abbildung 1) Daraus konnte ich einige Stellen des Prototypen, an denen Änderungen sinnvoll waren, leicht ermitteln.
Emma wurde eingesetzt, um zu erkennen, welche Bestandteile des Quelltextes für bestimmte Funktionalitäten benutzt werden. Dadurch konnte ich ungenutzte Klassen und Methoden ausfindig machen. Die Testabdeckung mit Emma zu überprüfen war nicht notwendig, da der Prototyp keine Testklassen besaß.
Der Einsatz des Werkzeugs FindBugs half an einigen Stellen des Quelltextes potentielle Fehlerquellen oder auch einfach überflüssige Bestandteile des Quelltextes zu erkennen, wie im Fall einer if-Else-Verweigung, die in beiden Fällen die gleiche Anweisung ausführte.

4.1 Architektur

Der Prototyp ist in der Programmiersprache Java geschrieben und benutzt das Paket Swing für die grafische Benutzerschnittstelle. Er besteht aus insgesamt zehn Klassen und 17 Hilfsklassen, die in drei Pakete mit den Namen `qsheet.model`, `qsheet.pdf` und `qsheet.ui` unterteilt sind. Er benutzt neun externe Bibliotheken, die überwiegend das Umwandeln von

[13] gaphor.sourceforge.net/

		Gesamtsystem	qsheet.model	qsheet.pdf	qsheet.ui
Werte	Pakete	3	1	1	1
	Klassen	54	3	1	50
	Anweisungen	1640	287	173	1180
	Externe Pakete	13	1	4	9
	Ungenutzte private Methoden	1	1	0	0
	Ungenutzte Importe	18	3	0	15
Metriken	Methoden pro Klasse (WMT)	1.64	18.33	19	3.74
	Modularisation (MOD)	0.67	1	0	1
	Kohäsion (COH)	33	1	0	0
	Objektkopplung (CBO)	0.83	1	0	0.84
	Methodenanzahl (NOM)	0.93	0.33	1	0.96

Abbildung 1: XRadar-Ergebnisse (Auswahl)

PDF-Dokumenten und Layoutgestaltung unterstützen. Zwischen den Paketen `qsheet.pdf` und `qsheet.ui` existiert eine zyklische Referenz. Diese Referenz führt zu einer starken Kopplung beider Pakete. Es ist durchaus denkbar, dass in der Zukunft des Projekts, angeregt durch neue Hardware-Geräte, die auf dem Markt erscheinen und eine andere Schnittstelle besitzen, der Wunsch auftritt, die grafische Benutzerschnittstelle getrennt von den PDF-Funktionen auszutauschen zu können. Daher sollten diese Pakete eine möglichst lose Kopplung aufweisen.

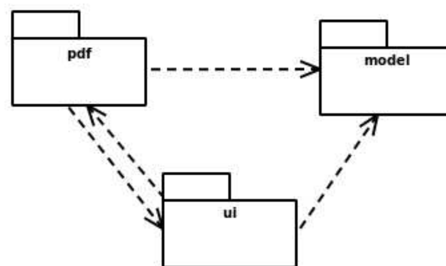

Abbildung 2: Paketstruktur des Prototypen

Im Folgenden werden die einzelnen Pakete und ihre Funktionen beschrieben.

4.1.1 Das Paket `qsheet.model`

Das Paket `qsheet.model` enthält drei Klassen, die nach dem WAM-Ansatz der Material-Metapher entsprechen. Sie werden von den anderen Paketen benutzt und manipuliert, ohne eine Klasse außerhalb des Pakets zu benutzen. Im einzelnen

repräsentieren sie eine Einzelseite des Fragebogens, als Grafik mit zusätzlichen Informationen (`qsheet.model.QPage`), eine Seitengruppe (`qsheet.model.QPageGroup`) und den gesamten Fragebogen, als Sammlung von Seiten und Seitengruppen (`qsheet.model.PageManager`).

4.1.2 Das Paket `qsheet.pdf`

Das Paket `qsheet.pdf` enthält nur eine einzige Klasse, `qsheet.pdf.PDFManager`, die das Singleton-Muster [GHJV96, S. 157] implementiert und verschiedene Dienste bezüglich PDF-Dokumenten anbietet, wie das Umwandeln der Seiten in Rastergrafiken.

4.1.3 Das Paket `qsheet.ui`

Das Paket `qsheet.ui` enthält die Werkzeuge zur Manipulation des Dokuments, sowie ihre grafischen Benutzerschnittstellen. Es ist mit insgesamt 23 Klassen und 75% des Quelltextes, das größte der drei Pakete. Im Gegensatz zu den anderen Paketen ist es überhaupt nicht dokumentiert. Auch existiert keine Trennung zwischen Präsentation und Steuerung innerhalb der Werkzeuge.
Es enthält die größte Klasse des Ausgangssystems (etwa 1000 Zeilen), `qsheet.ui.QSheet`. Diese Klasse enthält die `main`-Methode des Programms und übernimmt zusätzlich, als von `javax.swing.JFrame` erbende Klasse, die Darstellung des Hauptfensters. Zwischen ihr und einigen GUI-Klassen existieren weitere zyklische Referenzen.
Zusammenfassend ist `qsheet.ui` das Paket, das zum Verständnis der Funktionsweise des Prototypen am wichtigsten und gleichzeitig am schwersten zugänglich ist.

4.2 Planung der Umsetzung

Aufgrund der Analyse ließ sich bestimmen, dass die in geplanten Weiterentwicklungen an der Software, die in Abschnitt 2.2.4 beschrieben werden, mit der aktuellen Quelltextbasis nur mit großem Aufwand durchzuführen gewesen wären. Fowler schlägt in einer solchen Situation drei mögliche Vorgehensweisen vor: Eine Reimplementation, also die Software mit gleichem Funktionsumfang komplett neu schreiben, ein klassisches Refactoring wie in Abschnitt 3.1.1 beschrieben, oder durch ein erstes Refactoring das Gesamtsystem in einzelne, stark gekapselte Komponenten zu überführen und für jede Komponente eine eigene Entscheidung zwischen Refactoring und Reimplementation zu treffen [Fow07, S. 66]. Die dritte Möglichkeit habe ich ausgeschlossen, da sie mir für die Größe der Software zu

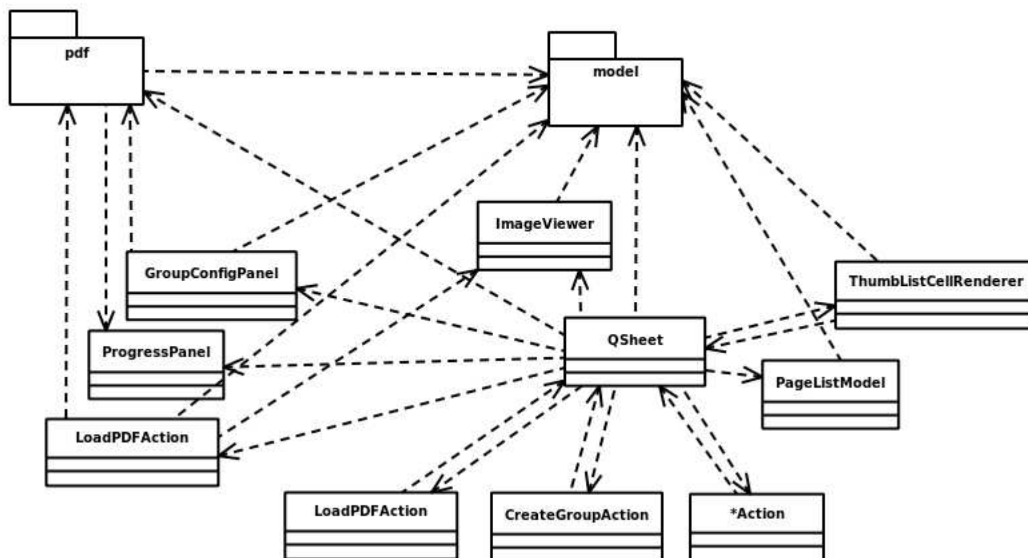

Abbildung 3: UML-Klassendiagramm des Pakets `qsheet.ui` im Prototyp. (Die Klassenbezeichnung *Action steht stellvertretend für sieben weitere Erweiterungen der Klasse `swing.AbstractAction`, die zur besseren Übersichtlichkeit weggelassen wurden)

aufwendig erschien.
Um eine Entscheidung zwischen den beiden anderen potentiellen Vorgehensweisen treffen zu können, habe ich Argumente aus der Literatur und dem Kontext des Projekts zusammengestellt und beide Seiten bewertet.

Folgende Argumente sind in die Bewertung der Vorgehensweisen eingeflossen:

A Das Verhalten des Prototyps ist nicht durch eine Testumgebung abgedeckt. Das Fehlen von Unit-Tests spricht gegen ein Refactoring, da ohne eine ausreichende Testabdeckung ein hohes Risiko besteht, produzierte Änderungen im Verhalten nicht zu bemerken. Eine Testumgebung in ein bestehendes Projekt nachträglich einzuführen wird von Link kritisch gesehen, da es dazu führen kann Fehler oder ungewolltes Verhalten des Programms durch die Tests festzuschreiben [Lin05, S. 345]. Bei einer Reimplementation kann die Testumgebung parallel mit der restlichen Funktionalität erstellt werden, oder dieser, durch testgetriebene Entwicklung, sogar vorhergehen [Lin05, S. 11].

B Meine Bachelorarbeit ist auf den Zeitraum von drei Monaten begrenzt, in denen ich als einziger Entwickler an dem Projekt arbeite. Wäre der Prototyp zu umfangreich,

um eine Reimplementation unter diesen Umständen Erfolgreich abschließen zu können, würde ich eine Reimplementation nicht in Betracht ziehen. Dies ist bei dem Prototyp, mit seinem Umfang von zehn Klassen, nicht der Fall.

C Weist der bestehende Quelltext essentielles Fehlverhalten auf, rät Fowler von einem Refactoring ab [Fow07, S. 66]. Um ein Refactoring durchzuführen muss eine korrekt funktionierende Quelltextbasis vorhanden sein, die sich auf Verhaltensänderung testen lässt. Der aktuelle Prototyp weißt zwar Schwächen in der Performanz auf, hat aber bis jetzt kein Fehlverhalten gezeigt.

D Diese Arbeit steht im Kontext eines größeren Projektes, des epp-Verfahrens, dessen zukünftiger Einsatz von der Weiterentwicklung der Software abhängt. Wird der Quelltext des Prototypen verworfen und die Software durch eine Reimplementation komplett neu geschrieben, ist die Weiterentwicklung erst dann möglich, wenn die Reimplementation vollständig abgeschlossen ist. Schlägt die Reimplementation fehl, weil beispielsweise der Aufwand unterschätzt wurde oder unvorhergesehene Änderungen auftreten, wäre mit der Arbeit kein Nutzen für das Projekt geschaffen. Ein Refactoring dagegen wird immer in kleinstmöglichen Schritten durchgeführt, wobei nach Abschluss jedes Schrittes eine funktionierende Version der Software entsteht, die gegenüber ihrem Vorgänger verbessert wurde. Unter diesem Aspekt ist ein Refactoring also ein geringeres Risiko für das Vorankommen des Projektes als eine Reimplementation.

E Durch seinen schlechten Zustand ist es zwar schwer sich in den Quelltext des Ausgangssystems einzulesen, die dabei gesammelten Erfahrungen können jedoch für die Arbeit am Projekt sehr nützlich sein. Er enthält letztendlich die Designentscheidungen der Entwickler des Prototypen. Bei einer Reimplementation des Prototypen würde ich unumgänglich auf einige der Probleme stoßen, die bei der ersten Entwicklung bereits gelöst wurden. Ohne eine intensive und aufwendige Einarbeitung in den schwer zu lesenden Quelltext bestünde keine Möglichkeit, auf die Lösungen meiner Vorgänger zurückzugreifen und sämtliche Entscheidungen müssten ein zweites mal getroffen werden. Ein Refactoring dagegen bringt immer eine notwendigerweise intensive Auseinandersetzung mit dem bestehenden Quelltext mit sich und hilft, ihn besser zu verstehen und leichter verändern zu können [Fow07, S. 56]. Dadurch kann von der vorhergegangenen Entwicklungsgeschichte des Prototypen bei der weiteren Entwicklung profitiert werden. Zusätzlich kann ich selbst als Entwickler

einen Nutzen aus dieser Vorgehensweise ziehen, da ich mir dadurch die Lösungswege anderer Entwickler erarbeite und daraus lerne.

In der Aufzählung sind Für- und Gegenargumente für beide möglichen Vorgehensweisen enthalten. Diese Argumente habe ich gegeneinander abgewogen. Dabei habe ich den letzten beiden Argumenten (D und E) höheres Gewicht eingeräumt, da sie starken Bezug zum Projekthintergrund haben. Unter diesen Voraussetzungen zeigt die Abwägung der Argumente, dass in dieser Situation ein Refactoring sicherer und profitabler für das Projekt ist, als eine Reimplementation, jedoch zuerst einer Vorbereitung des Quelltextes bedarf.

4.2.1 Agile Praktiken

Zu Beginn der im folgenden Abschnitt beschriebenen Arbeiten am Quelltext war die grundlegende Entscheidung zu treffen, ob und wenn ja, welche weiteren agilen Praktiken, neben dem Refactoring, eingesetzt werden sollen.
Um besser auf Änderungen, beispielsweise in der Priorität einzelner Probleme oder auch auf neue Ideen reagieren zu können, habe ich mich dazu entschieden gegenüber einer einstufigen Planung nach dem Wasserfallmodell, eine iterativ inkrementelle Vorgehensweise einzusetzen. Dazu war es notwendig den Gesamtprozess der Umsetzung in Iterationen zu unterteilen und gleichzeitig regelmäßige Rückmeldungen über den Fortschritt zu erhalten. Diese sollten möglichst von einer/einem KundIn stammen um eine hohe Einbezogenheit der KundInnen zu erreichen. Auf die Vorbereitungsphase folgend, habe ich drei Iterationen angesetzt. In den ersten zwei Iterationen habe ich das Refactoring des Quellocdes durchgeführt. Die dritte Iteration wurde für Verbesserungen der Performazschwächen vorgesehen. Jede dieser Iterationen bestand aus einer Planungsphase, einer Entwicklungsphase in der parallel regelmäßige Tests durchgeführt wurden, einer Analyse der Ergebnisse, die jeweils eine aktuelle XRadar-Auswertung und das Erstellen aktueller Klassendiagramme enthielten, und einem Treffen, in dem Rückmeldung zu den Ergebnissen der Iteration eingeholt wurde.
Die KundInnen eines Refactorings, also Menschen, deren Leben und Arbeiten davon betroffen sind, sind eher zukünftige Entwickler als die AnwenderInnen der Software. Diese waren nicht greifbar, beziehungsweise ich selbst im Fall der folgenden Iterationen. Um trotzdem qualifizierte Rückmeldungungen zu den Entwicklungen zu erhalten, habe ich die Treffen mit dem Erstbetreuer dieser Arbeit, Axel Schmolitzky, abgehalten.

In der dritten Iteration, in der das äußere Verhalten der Software geändert wurde, habe ich mir in zeitlich kurzen Abständen Rückmeldungen vom Projektleiter Böschen eingeholt, der Auftraggeber- und als Diagnognostiker auch Anwenderrolle ausfüllt.
Rückmeldungen von DiagnostikantInnen habe ich erhalten, indem ich die Änderungen Personen vorgeführt habe, die nicht an dem Projekt beteiligt waren und testhalber einen Fragebogen ausfüllten.

5 Umsetzung und Ergebnisse

In diesem Abschnitt wird die Umsetzung der aus der Analyse resultierenden Planung in drei Iterationen beschrieben. Zu jeder Iteration werden die Ergebnisse der darauf folgenden Analyse kurz aufgeführt.

5.1 Refactoring

Im Folgenden werden die vorbereitende Phase und die beiden Refactoring-Iterationen beschrieben. In der Beschreibung der Iterationen werden einzelne Änderungen, die im Rahmen des Refactorings durchgeführt wurden, aufgelistet. Dabei werden einige Änderungen mit geringerer Wirkung, wie das Umbenennen einzelner Methoden und Exemplarvariablen, nicht mit aufgeführt, da sie unter dem Aspekt der Übersichtlichkeit nicht genügend Relevanz besaßen. Aufgrund der speziellen Gegebenheiten, die jeder Quelltext mit sich bringt, sind nicht alle vorgenommenen Änderungen auf ein einzelnes definitiv passendes Refactoring-Muster zurückzuführen. In diesen Fällen habe ich mich an das Schema gehalten, in sehr kleinen Schritten vorzugehen und die Änderungen regelmäßig durch die aufgesetzte JUnit-Testsuite zu überprüfen.

5.1.1 Vorbereitung

Das wichtigste Ziel der vorbereitenden Phase bestand darin, eine Testumgebung zu erhalten, die das äußere Verhalten prüft, um die Sicherheit zu erhöhen, ungewollte Veränderungen im Laufe des Refactorings zu bemerken. Zusätzlich wurden Veränderungen an den Elementen des Quelltextes, die keinen Einfluss auf das Verhalten haben, beispielsweise Kommentare und unerreichbare Anweisungen, durchgeführt. Dadurch sollte die allgemeine Lesbarkeit und mein Verständnis des Quelltextes verbessert werden, um das Refactoring leichter durchführen zu können.
Folgende Schritte habe ich zur Vorbereitung durchgeführt:

- Die von FindBugs bestimmten potentiellen Fehlerquellen habe ich vor dem Aufsetzen der Testsuite behoben, damit diese nicht durch die Unit-Tests festgeschrieben werden konnten.

- Ich habe eine Unit-Testumgebung für das Ausgangssystem aufgesetzt. Hierfür wurden JUnit-Tests für sämtliche Klassen erstellt. Das äußere Verhalten der grafischen Benutzerschnittelle wurde überwiegend durch Klick-Roboter des Jemmy-Frameworks

geprüft. Die durch das Werkzeug Emma ermittelte Testabdeckung am Ende der vorbereitenden Phase betrug etwa 95,8%.

- Für die Elemente der Schnittstelle aller Klassen habe ich Javadoc-Kommentare geschrieben, die ihr Verhalten beschreiben. Dies wurde parallel zum Aufsetzen der Tests der entsprechenden Bereiche im Quelltext durchgeführt, da häufig erst durch das Testen dieses Bereichs das tatsächliche Verhalten bestimmt werden konnte und dieses nicht in jedem Fall dem erwarteten entsprach.
- Mit Emma und der Funktion der Eclipse IDE zum Finden von Referenzen auf Methoden und Klassen konnte ich die Schnittstellen aller Klassen so weit reduzieren, dass sie nur noch solche Elemente enthalten, auf die von außen zugegriffen wird. Weiterhin konnte ich Stellen im Quelltext des Ausgangssystems ausmachen und entfernen, die nicht verwendet wurden. Dadurch hat sich die Übersichtlichkeit des Quelltextes verbessert.

5.1.2 I. Iteration

Der Schwerpunkt der Änderungen, die im Rahmen der ersten Refactoring-Iteration durchgeführt wurden, lag darauf, die Probleme zu beheben, die durch die Analyse aufgezeigt wurden. Folgende Änderungen waren Teil dieser Iteration:

- Die enge Kopplung zwischen den Paketen `qsheet.pdf` und `qsheet.ui`, sollte aus den in Abschnitt 4.1 genannten Gründen, reduziert werden. Dazu war es nötig die zyklische Referenz zwischen ihnen zu lösen. Diese Referenz existiert, da die Klasse `qsheet.pdf.PDFManager`, die von verschiedenen Klassen im Paket `qsheet.ui` zur Manipulation von PDF-Dateien verwendet wird, eine Referenz auf ein Exemplar der Klasse `qsheet.ui.ProgressPanel` als Exemplarvariable hält. Diese wird über den Fortschritt der aktuellen Manipulation informiert und stellt ihn durch einen prozentualen Fortschrittsbalken an der grafischen Benutzerschnittstelle dar.
  Als Lösung für entsprechende Probleme existiert das Entwurfsmuster des Beobachters (Observer) [GHJV96, S. 287], in dem die Kopplung über Implementation von Interfaces abstrahiert wird. Um dieses Muster in einem Refactoring zu implementieren existiert das Refactoring-Muster "Replace Hard-Coded Notifications with Observer" [Ker04, S. 236]. Durch Befolgen dieser Muster und Verwenden der bereits für diesen Fall existierenden Java-Klassen `util.Observable` und `util.Observer` habe ich diese Referenz aufgelöst.

- Die Klasse `qsheet.ui.QSheet` war mit etwa 1000 Quelltextzeilen im Ausgangssystem groß genug, um eine Teilung zu rechtfertigen, um dadurch ihr Verhalten einfacher erfassen zu können. Zusätzlich übernimmt sie mit dem Darstellen und Manipulieren des Materials aus dem Paket `qsheet.model` Aufgaben, die von zwei Klassen übernommen werden sollten.
  Nach dem Refactoring-Muster "Extract Class" [Fow07, S. 149] habe ich die grafischen Bestandteile der Klasse in eine neue Klasse mit dem Namen `qsheet.ui.QSheetView` extrahiert. Die Funkion "Refactoring.Extract Class" der Eclipse-IDE ist in der Lage dieses Refactoring-Muster zu unterstützen, indem sie die neue Klasse automatisch erstellt und auswählbare Exemplarvariablen in sie übertragen werden. Um eine einheitliche Trennung zwischen den Aufgaben beider Klassen herzustellen, müssen allerdings einige Schritte manuell nachgearbeitet werden, wie die Aufteilung von Methoden, die Quelltext beider Aufgabengebiete beinhalten.

- In einigen Methodennamen, sowie in der Dokumentation des Ausgangssystems wird der Begriff "Index einer Seite" für zwei unterschiedliche Bedeutungen verwendet. Die erste Bedeutung ist die Positionsnummer in dem Array, in dem die Seite gespeichert ist und über die auf sie zugegriffen werden kann. Die zweite Bedeutung ist die Seitenzahl der Seite im ursprünglichen PDF-Dokument. Da die Position im Array mit 0, die Seitenzahl meistens mit 1 beginnt und die Seite im Array verschoben werden kann, unterscheiden sich diese Zahlen in der Regel. Besonders auffällig ist die gemeinsame Verwendung des Begriffs in der Klasse `qsheet.model.PageManager` in der auf eine Seite auf beide Arten zugegriffen werden kann. Die Methode, mit der über die Position im Array auf eine Seite zugegriffen werden kann, hat folgende Schnittstelle:

  ```
  public QPage getPage(int index)
  ```

  Dagegen gibt folgende Methode eine Seite per Seitenzahl zurück:

  ```
  public QPage getPageWithIndex(int index)
  ```

  Da aufgrund dieser Uneindeutigkeit der Quelltext schwerer zu verstehen war, habe ich der Bedeutung "Seitenzahl" den Begriff "page number" zugeordnet. Dies habe ich im gesamten Quelltext durch Umbenennen einzelner Methoden [Fow07, S. 273] und Exemplarvariablen, so wie Anpassen der Dokumentation durchgeführt. Auch hierbei konnte die automatisierte Refactoring-Unterstützung der Eclipse IDE erfolgreich einsetzt werden.

5.1.3 Ergebnisse der I. Iteration:

Durch die erste Iteration konnten vor allem die Durchschnittsgröße der einzelnen Klassen reduziert und zyklische Referenzen aufgelöst werden. Die Analyse mit XRadar zeigte eine Verbesserung in den Metriken des Bereichs Architektur. Der Gesamtwert, der sich aus den Werten für Modularisierung und Kohäsion zusammensetzt, stieg aufgrund des nicht mehr vorhandenen Paketzyklus von $0,47$ auf das Maximum 1. Entgegen den Erwartungen verringerten sich die Werte im Bereich Design um wenige Prozentpunkte. Dies hat unter anderem damit zu tun, dass Verbesserungen in einigen Metriken erst ab einem festgelegten Wert erfasst werden. Ein Beispiel ist die Metrik *Coupling Between Objects*, der die Kopplung zwischen Objekten angibt. Auf den Wert der Metrik wirken alle Objekte des betrachteten Systems negativ ein, die Beziehungen zu mehr als vier anderen Klassen besitzen.

Coupling Between Objects
Die Metrik Coupling Between Objects wird von XRadar mit der Formel $CBO = \frac{count_classes(cbo<5)}{total\text{-}classes}$ berechnet. Mit:

- CBO: Der Wert der Metrik für das betrachtete System.
- $count_classes(cbo < 5)$: Anzahl der Klassen im betrachteten System, die zu weniger als fünf Klassen Beziehungen haben.
- *total-classes*: Anzahl aller Klassen im betrachteten System.

Die Klasse `qsheet.ui.QSheet` hat im Ausgangssystem eine sehr hohe Anzahl von 29 Beziehungen. Durch ihre Teilung entstanden zwar zwei Klassen mit einer geringeren Kopplung, allerdings erreicht keine einen Wert unter vier Beziehungen. Dadurch hat sich die XRadar-Metrik dieser Version mit einer zusätzlichen Klasse, die diese Bedingung nicht erfüllt, im Vergleich zum Ausgangssystem verschlechtert.

5.1.4 II. Iteration

In der zweiten Refactoring-Iteration wurden vorwiegend große Methoden und Klassen weiter aufgeteilt. Ein Weg dazu waren noch ausstehende Trennungen zwischen grafischem und manipulierenden Quelltextbereichen.
Durch die Reflexion der ersten Iteration wurde ich dazu angeregt, die Software aus dem

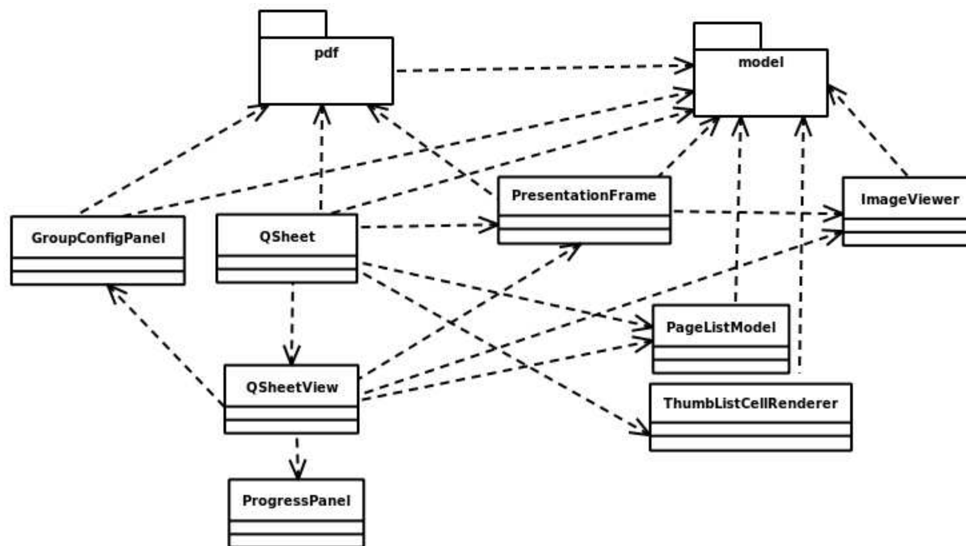

Abbildung 4: UML-Klassendiagramm des Pakets `qsheet.ui` nach der I. Iteration

Blickwinkel des Werkzeug & Material-Ansatzes zu betrachten und gegebenenfalls neu zu strukturieren.
Folgende Änderungen wurden in dieser Iteration umgesetzt:

- Nach der Betrachtungsweise des Werkzeug & Material-Ansatzes lassen sich in der Software zwei Werkzeuge ausmachen, die das Material manipulieren, das sich im Paket `qsheet.model` befindet. Das erste dient dem/der DiagnostikerIn zur Vorbereitung des Präsentationsmodus. Es unterstützt also beispielsweise Aufgaben, wie das Gruppieren von Seiten oder das Einstellen der Bearbeitungszeit einzelner Seitengruppen. Der Quelltext dieses Werkzeugs und seiner grafischen Benutzerschnittstelle ist im Ausgangssystem auf unterschiedliche Klassen verteilt; der größte Teil befindet sich in der Klasse `qsheet.ui.QSheet`. Das zweite Werkzeug dient zur Durchführung der Präsentation. Es stellt klassisch die einzelnen Seiten dar und lässt diese von dem/der DiagnostikantIn manipulieren, indem Zeichnungen des Stiftes auf ihnen gespeichert und angezeigt werden. Auch diese Funktionen lassen sich im Ausgangssystem, verteilt auf unterschiedliche Klassen, finden.
  Eine leichter erkennbare Strukturierung dieser Werkzeuge kann dabei helfen, das System einfacher zu verstehen und passt damit in den Kontext eines Refactorings. Deshalb habe ich durch Verschieben von Methoden und Exemplarvariablen, die Funktionalität des Vorbereitens in der Klasse `qsheet.ui.QSheet` und die zur Durchführung einer Präsentation in `qsheet.ui.PresentationFrame` gebündelt. Die

anderen Klassen des `qsheet.ui`-Paketes übernehmen dagegen nur noch grafische Aufgaben zur Unterstützung dieser Klassen.

- Durch das Umstrukturieren der einzelnen Klassen des Pakets `qsheet.ui` hat sich der Umfang der Klasse `qsheet.ui.PresentationFrame` stark vergrößert. Entsprechend dem Vorgehen bei der Klasse `qsheet.ui.QSheet` in der ersten Iteration habe ich in dieser Klasse den Quelltext der grafischen Benutzerschnittstelle in eine eigene Klasse `qsheet.ui.PresentationFrameView` extrahiert.

- Die Klasse `qsheet.pdf.PDFManager` übernimmt unter anderem Aufgaben zum Importieren und Exportieren von PDF-Dateien. Die interne Repräsentation von PDF-Dateien als Material befindet sich in der Klasse `qsheet.model.PageManager`. Eine QST-Datei, die Einstellungen einer Präsentation speichert, wird ebenfalls im `qsheet.model.PageManager` intern repräsentiert. Im Gegensatz zu dem internen Umgang mit PDF-Dateien, befinden sich die Import- und Exportfunktionen der QST-Datei in statischen Methoden, die in der selben Klasse stehen, wie die Repräsentation. Dadurch existieren bezüglich des Aufteilens von Repräsentation und Import/Export eines Materials, zwei unterschiedliche Umsetzungskonzepte, was das einfache Verstehen erschweren kann. Um diese auf ein einheitliches Konzept zu reduzieren, habe ich die Import/Export-Funktionen für QST-Dateien in eine neue Klasse `qsheet.pdf.PageManagerIO` extrahiert.

- Der Name des Pakets `qsheet.pdf` war aufgrund der letzten Änderung nicht mehr für beide beinhaltenden Klassen passend. Beide sind Klassen, die Dienste anbieten, welche die Werkzeuge bei ihrer Arbeit unterstützen, und damit dem Elementtyp "Service" in [BPKL06] entsprechen. Deshalb habe ich das Paket in `qsheet.services` umbenannt.

- Die Verwendung von Klassen, die das Singleton-Muster umsetzen, wird vermehrt kritisch betrachtet. Singletons sollten verwendet werden, wenn von einer Klasse genau ein Exemplar existieren soll, auf das jederzeit zugegriffen werden kann [GHJV96, S. 157]. Unbedacht eingesetzte Singletons können dagegen ähnliche Probleme wie globale Variablen haben, indem sie zum Beispiel Abhängigkeiten in weit entfernten Teilen eines Systems herstellen. Dabei können Seiteneffekte auftreten, deren Ursprung schwer auszumachen ist [Rai01]. Zudem lassen sich Singletons im Vergleich zu anderen Klassen schwerer testen, da das Erzeugen eines neuen, unveränderten Exemplars

unterbunden wird [Lin05, S. 121].

Für das im Ausgangssystem verwendete Singleton `qsheet.services.PDFManager` besteht keine Notwendigkeit, dieses Muster umzusetzen. Ein Exemplar des Objekts ist nicht aufwendig zu initialisieren, noch beinhaltet sie Variablen, die global verfügbar sein müssen.

Für das Entfernen eines Singletons in einer Recht ähnlichen Situation existiert das Refactoring-Muster "Inline Singleton" [Ker04, S. 114]. Nach diesem Muster wird der Quelltext des Singletons, entsprechend dem Muster "Inline Class" [Fow07, S. 154], von benutzenden Klassen aufgenommen. Dadurch verschwindet das Singelton und mindestens eine benutzende Klasse wird entsprechend größer. Da die PDF-Funktionen als Service gekapselt bleiben sollte, habe ich dieses Muster leicht variiert, indem kein Quelltext zwischen Klassen verschoben wurde, sondern nur die ursprünglich Klasse das Singleton-Muster nicht mehr erfüllt. Dazu habe ich den spezifischen Singleton-Quelltext aus der Klasse entfernt und durch einen öffentlichen Konstruktor ersetzt. Die externen Aufrufe auf dem Singleton habe ich entsprechend umgeschrieben, so dass vor dem Aufruf neue Exemplare erzeugt werden, auf denen die Anweisungen ausgeführt werden können.

5.1.5 Ergebnisse der II. Iteration:

Die zweite Iteration schließt die in der ersten Iteration begonnenen Verbesserungen an der Klassenstruktur des Paketes `qsheet.ui` ab. Durch die Umverteilung der Aufgaben einzelner Klassen und die dadurch reduzierte Anzahl an Benutzt-Beziehungen, lässt sich das Klassendiagramm nach dieser Iteration leichter verstehen und kommunizieren.

Im Gegensatz zur ersten Iteration, ließen sich die Verbesserungen im Gesamtsystem nach der zweiten Iteration auch in minimalen Verbesserungen in den Design-Metriken der XRadar-Analyse ablesen.

5.2 Performanzoptimierungen (III. Iteration)

Die dritte Iteration umfasst Änderungen an der Software, welche die direkten Aufträge Böschens in der Rolle des Kunden umsetzten. Zusammengefasst dienen sie der Verbesserung der Performanz der Software. Sie sind von einem Refactoring begrifflich zu trennen, da sie nicht mit der Motivation, die Les- und Wartbarkeit des Quelltextes zu verbessern durchgeführt wurden und letztendlich das äußere Verhalten der Software verändern.

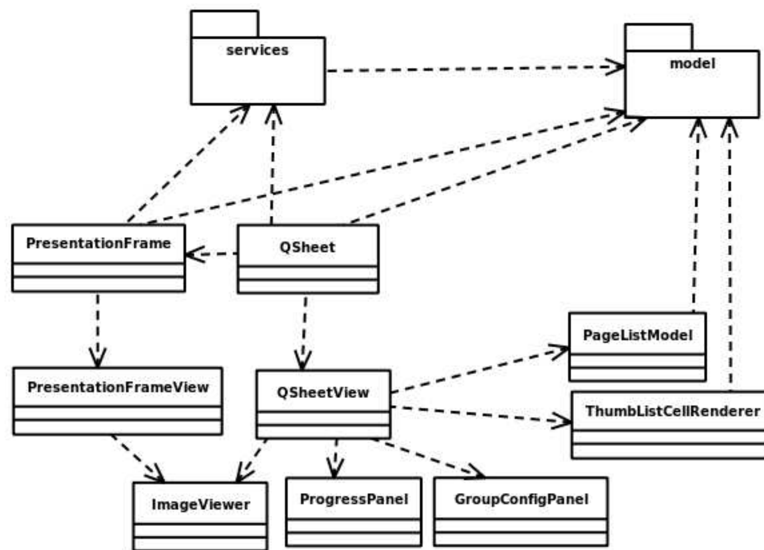

Abbildung 5: UML-Klassendiagramm des Pakets `qsheet.ui` nach der II. Iteration

Ihr problemloses und erfolgreiches Durchführen, an einer Software, deren Quelltext vor dem Refactoring sehr schwer zu verändern war, dient in dieser Arbeit, den Erfolg des Refactorings zu zeigen.

5.2.1 Doppeleingabe bei Navigationsbuttons

Das am höchsten priorisierte Performanzproblem war das Problem der Doppeleingabe bei den Navigationsbuttons (Vgl. Abschnitt 2.2.4). Diese Doppeleingaben konnten auftreten, wenn das Nachladen einer Seite zu viel Zeit in Anspruch nahm. In dieser Situation entsprach die Umsetzung des epp-Verfahrens durch die Software nicht den formulierten Anforderungen, denn dieses in der Papier-Bleistift-Testung nicht vorkommende Verhalten offenbart dem/der DiagnostikantIn unvermeidlich, dass er/sie mit einem Computer arbeitet, auf dem Bedienfehler gemacht werden können. Diese Störung der Papieräquivalenz begründete die hohe Priorität des Problems. Deshalb war dessen Lösung bereits in der Planung am Ende der ersten Analyse als als Inhalt der dritten Iteration angesetzt worden. Damit nicht unnötig Arbeit in eine Lösung investiert wurde, die nicht den Vorstellungen des Kunden entspricht, sollte dieser Teil der Arbeit, im Sinne hoher Kundeneinbezogenheit, noch stärker durch gemeinsames Erarbeiten des Vorgehens abgesichert werden. Dazu habe ich eine Liste möglicher Änderungen erstellt, die das Problem lösen könnten. In dieser wurden absehbare Vor- und Nachteile der einzelnen Lösungen aufgeführt und gemeinsam präsentiert. Die vorgeschlagenen Optionen waren folgende:

A Die Navigationsbuttons werden so lange ausgeblendet, bis die gewünschte Seite vollständig geladen ist.
Diese Option ist einfach umzusetzen. Es reicht, vier Anweisungen im Quelltext hinzuzufügen.

B Die/Der DiagnostikantIn wird über ein Fenster oder einen ähnlichen Hinweis darüber informiert, dass die Seite im Moment geladen wird.
Durch diese Form der Rückmeldung weiß der/die DiagnostikantIn, dass seine Eingabe registriert wurde und der Computer im Ladezustand ist. Allerdings widerspricht diese Option dem Konzept es epp-Verfahrens, da dadurch der Computer zu stark präsent würde.

C Die Software agiert schneller und auch die Seite erscheint ausreichend schnell. (Erreicht durch allgemeine Performanzverbesserungen im Quelltext.)
Diese Möglichkeit würde bei Erfolg den größten Vorteil für das Projekt bringen. Es ist jedoch auch möglich, dass sie viel Zeit und Arbeit in Anspruch nimmt, ohne das ursprüngliche Problem in ausreichendem Maße zu beheben.

D Die Seite erscheint schneller. (Erreicht durch internes Vorladen der nächsten und der vorhergehenden Seite.)
Diese Option könnte den Regelfall, der darin besteht, dass eine einzelne Seite umgeblättert wird, wahrscheinlich verbessern. Geht die/der DiagnostikantIn allerdings zwei oder mehr Seiten schnell hintereinander weiter, oder springt das Programm selbst aufgrund einer abgelaufenen Bearbeitungszeit zu einer späteren als der übernächsten Seite, so besteht das Problem weiterhin.

In der Präsentation und anschließenden Abwägung dieser Vorschläge wurde eine neue Option formuliert.

E Die Navigationsbuttons werden so lange ausgeblendet, bis die gewünschte Seite vollständig geladen ist. Zusätzlich werden sie beim erscheinen einer neuen Seite für eine Viertelsekunde ausgeblendet.

Mit dem ersten Teil von Vorschlag E, der Vorschlag A entspricht, wird verhindert, dass im Ladezustand mehr Mausklicks auf den Navigationsbuttons als gewünscht registriert werden. Der zweite Teil verhindert, dass Seiten umgeblättert werden, die nicht betrachtet

wurden. Ein Mausklick innerhalb der ersten Viertelsekunde der Darstellungszeit einer Seite, kann nicht als Reaktion auf die betrachtete Seite gewertet werden. Es ist davon auszugehen, dass die Seite überblättert wurde ohne betrachtet worden zu sein oder ein ungewollter Klick ausgeführt wird. Beides ist in der Testdurchführung unerwünscht.
Diese Lösung entsprach den Vorstellungen des Kunden. In der Klasse `qsheet.ui.PresentationFrame`, welche die Funktionen des Präsentationsmodus fasst, habe ich die Lösung ohne größere Probleme implementiert, indem ich eine Methode zum Ein- und Ausblenden der Navigationsbuttons und ein Exemplar der Klasse `javax.swing.Timer`, die eine Viertelsekunde abzählt, eingesetzt habe.

5.2.2 Zeichengeschwindigkeit

Neben der Doppeleingabe bei den Navigationsbuttons bestand eine weitere Schwachstelle in der Performanz des Ausgangssystems, die den Präsentationsmodus negativ beeinflusste: Die Geschwindigkeit mit der die eingegeben Linien dargestellt wurden, war zu langsam. (Vgl. Abschnitt 2.2.4) Dieses Verhalten brach ebenfalls mit der Forderung des epp-Verfahrens nach höchster Äquivalenz zur Papier-Bleistift-Testung.
In der Ausgangsplanung wurde auch dieses Problem hoch priorisiert. Es war zu dem damaligen Zeitpunkt jedoch noch nicht klar, ob die Zeichengeschwindigkeit durch die Hard- oder Software begrenzt würde. Die ursprüngliche Planung der dritten Iteration enthielt zunächst lediglich die Lösung des Problems der Doppeleingaben. Aufgrund des vorhergehenden Refactorings hatte sich die Wartbarkeit des Quelltextes und meine Kenntnis dessen entschieden verbessert, so dass dieses erste Problem schneller behoben werden konnte, als erwartet. Dadurch war die Lösung des Problems der Verzögerten Darstellung von Eingaben, im Zeitrahmen der dritten Iteration möglich.
An dieser Stelle konnte ich ebenfalls von der agilen Vorgehensweise profitieren, da sie es erlaubt in entsprechenden Situationen das Vorgehen flexibel anzupassen. Die Stellen im Quelltext auszumachen, die für dieses Problem verantwortlich waren, und sie zu ändern machte ebenfalls aufgrund des vorhergehenden Refactorings nur einen vergleichsweise geringen Zeit- und Arbeitsaufwand aus.
Das Problem habe ich durch die folgenden Änderungen behoben:

Die Eingaben des Stiftes, die die Software intern als Mauseingaben behandelt, werden von Listenern registriert und verarbeitet. Diese befinden sich in der Klasse `qsheet.ui.PresentaionFrame`. Wird eine Linie beendet wird die Methode `MouseDragged`

aufgerufen. Diese fügt zuerst die neu gezeichnete Linie zu den bestehenden hinzu und ruft als letzte Anweisung die Methode `repaint` an dem Exemplar der Klasse `qsheet.ui.ImageViewer` auf, welche die aktuelle Seite anzeigt. Der `ImageViewer` erbt die Methode `repaint` von der Klasse `JComponent` des SWING-Frameworks. Ein Aufruf von `repaint` ohne Parameter bewirkt ein komplettes Neuzeichnen der Komponente. Wird diese Methode mit einem rechteckiger Bereich, als Parameter aufgerufen, beschränkt sich das Neuzeichnen auf den übergebenen Bereich.
Die Methode `MouseDragged` habe ich so angepasst, dass zuerst das umfassende Rechteck der Linie bestimmt und bei der `repaint`-Anweisung als Parameter übergeben wird. Dadurch wird nicht mehr nach jeder neuen Linie der Bereich des gesamten Displays neu gezeichnet, sondern nur noch der Bereich, in dem die neue Linie hinzugefügt wurde. Diese Änderung bewirkt, dass dieser Prozess, der in der durchschnittlichen Anwendung des Präsentationsmodus sehr oft ausgeführt wird, weniger rechenintensiv und letztendlich schneller ist.
Zusätzlich habe ich Änderungen am Schriftbild des Ausgangssystems vorgenommen. Das Schriftbild, das aus den Einzellinien besteht, war sehr hart, da die Linien ohne Glättung der Kanten gezeichnet wurden. Dieses habe ich dadurch geändert, dass beim Zeichnen der Linien, der in der `awt`-Bibliothek enthaltene Antialiasing-Algorithmus verwendet wird. Dadurch erscheint das Schriftbild nun weicher und natürlicher. Die Berechnung der Kantenglättung bedeutet jedoch zusätzlichen Rechenaufwand, der bei jeder gezeichneten Linie anfällt. Da sich die Zeichengeschwindigkeit dadurch wieder reduziert, wurde Antialiasing als optionale Funktion eingebaut und kann über das Menü aktiviert und deaktiviert werden. Somit lassen sich beide Optionen im Feldeinsatz testen und der Anwender kann eine Entscheidung treffen, welche für das Projekt angemessener ist.

5.2.3 Ergebnisse der III. Iteration:

Um die Veränderungen an der Software bewerten zu können, war es notwendig vergleichende Tests zwischen der letzten Iteration und dem Ausgangssystem durchzuführen. Neben den Aussagen betroffener Personen, deren Evaluation in den geplanten Studien zu Böschens Promotion folgen soll, wurden automatisierte Vergleichstests aufgesetzt. Dabei sind Eigenschaften der Umgebung und Durchführung der Tests beider Versionen von Bedeutung: Sie sollten möglichst identisch sein, damit die Ergebnisse vergleichbar sind, und sie sollten dem tatsächlichen Einsatz möglichst ähnlich sein, damit die Ergebnisse

aussagekräftig sind.
Um den tatsächlichen Einsatz zu simulieren, habe ich beide Versionen auf dem eingesetzten Touch-Screen Tablet-PC installiert. Damit die Durchführung identisch ist, habe ich mit Hilfe der Jemmy-Bibliothek einen Test-Roboter geschrieben, der automatisiert identische Benutzereingaben an der Software simuliert und dokumentiert.
Der Roboter beginnt, in dem er eine Präsentation vorbereitet, also eine PDF-Datei einliest und ein Verzeichnis, in das die Ausgaben exportiert werden sollen, erstellt. Daraufhin startet er den Präsentationsmodus und führt die vergleichenden Operationen aus. Als letzten Schritt löscht er die erstellten Verzeichnisse und exportiert die in einem `java.lang.StringBuffer` gesammelte Dokumentation der Ereignisse in einer Textdatei.
Die Testoperationen, die der Roboter ausführt, sind dabei zuerst simulierte Mausklicks auf den Navigationsbutton zum Umblättern einer Seite, um das Problem der Doppeleingabe zu überprüfen, und anschließend simulierte Zeichnungen der Maus auf die angezeigte Seite, um die Zeichengeschwindigkeit zu vergleichen.
Im ersten Fall simuliert der Roboter erst einen einfachen und anschließend einen doppelten Klick, ohne Verzögerungszeit dazwischen. Dabei wird nach jedem Schritt die aktuelle Seite protokolliert. Die Protokolldaten zeigen (Siehe Abbildung 6), dass im Ausgangssystem auf alle drei Klicks reagiert und drei Seiten umgeblättert wurden, wogegen in der neuen Version, den Anforderungen entsprechend, nur eine Seite weiter geblättert wird. Ein schnelles Durchblättern wird also durch die neue Version verhindert.

Ausgangssystem	Iteration 3

`Klicke next...`
`Neue Seite 2`
`Klicke next doppelt...`
`Neue Seite 3`
`Neue Seite 4`
	`Klicke next...`
	`Neue Seite 2`
	`Klicke next doppelt...`

Abbildung 6: Protokoll des Seitenblätterns

Die Zeichengeschwindigkeit wird mit Hilfe von drei unterschiedlichen Linien gemessen. Getestet wird das Ausgangssystem gegenüber dem System der dritten Iteration mit und ohne Antialiasing. Die Dauer des Zeichnens einer Linie wird dabei durch Zeitmessungen ermittelt, die vor und nach dem Zeichnen durchgeführt und im `StringBuffer` protokolliert werden. Dieser Schritt wurde von dem Roboter etwa 100 mal automatisiert wiederholt. Anschließend habe ich aus den Ergebnissen die mittlere Dauer in jeder der drei Versionen

ermittelt.

Die für den Test gezeichneten Linien unterscheiden sich dabei in der Fläche des umschließenden Rechtecks (Bounding Box). Das Rechteck zu $Linie_1$ fasst 6.400 Pixel ($Linie_1 = \{\binom{10}{10} + \gamma \binom{80}{80} | \gamma \in [0,1]\}$), zu $Linie_2$ 310 Pixel ($Linie_2 = \{\binom{90}{90} + \gamma \binom{310}{0} | \gamma \in [0,1]\}$) und zu $Linie_3$ etwa 150.000 Pixel ($Linie_3 = \{\binom{10}{10} + \gamma \binom{390}{390} | \gamma \in [0,1]\}$) .

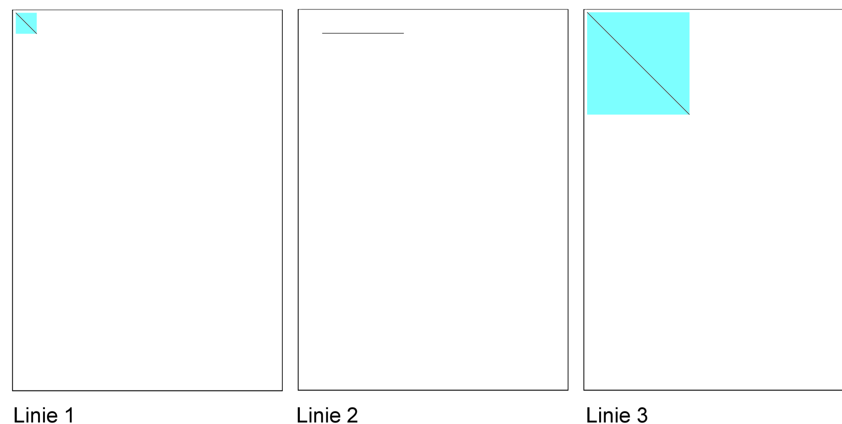

Abbildung 7: Die Linien mit umschließenden Rechtecken auf einer DIN-A4-Seite.

Im Ausgangssystem wird für jede neue Linie die volle Fläche der Seite neu gezeichnet, daher war zu erwarten, dass die Zeichendauer aller drei Linien etwa gleich lang ist. In den Versionen der dritten Iteration, in denen beim Zeichnen einer Linie nur das umfassende Rechteck neu gezeichnet wird, sollte die Dauer von der Größe des Rechtecks abhängen. Die Zeichendauer einer Linie sollte sich dabei der in etwa konstanten Zeichendauer des Ausgangssystems nähern, je ähnlicher sie einer Diagonalen des Bildschirms ist. Aufgrund der zusätzlichen Berechnung könnte sie im diesem Fall etwas höher sein. Eine solche Linie wird aber in der normalen Anwendung nicht gezeichnet werden. Die Standardeingaben bestehen aus kleinen Zeichen, wie Kreuzen oder Kreisen, und Schrift, die bei hoher Abtastungsrate des Stifts aus sehr vielen kurzen Linien bestehen. Diese zu erwartenden Linien, besitzen umfassende Rechtecke die kleiner als das Rechteck von $Linie_1$ sind. Sehr selten werden Linien vorkommen, die ein umfassendes Rechteck der Größe von $Linie_3$ besitzen; sie ist also als Extremfall, beispielsweise willkürliches Malen auf der ganzen Seite, zu sehen.

Die Messergebnisse (Siehe Abbildung 8) zeigen, dass die Erwartungen zutreffend

waren: Die Version des Ausgangssystems braucht zum Zeichen jeder Linie unabhängig von der Größe des umfassenden Rechtecks eine relativ konstante Zeit zwischen 33 und 43 Millisekunden. Die neue Version braucht maximal 10 ms für das sehr große Rechteck zu $Linie_3$. Die beste Zeit erreicht $Linie_2$ mit einem Durchschnittswert unter 3 ms. Die Ergebnisse mit eingeschaltetem Antialiasing zeigen eine etwas längere Zeichendauer. Diese liegt bei $Linie_3$ mit höchstens 15 ms etwa bei der Hälfte der Zeit, die das Ausgangssystem benötigt.
Zusammenfassend lässt sich sagen, dass die Version der dritten Iteration, mit und ohne Antialiasing, das Zeichnen der Linien deutlich schneller durchführt. Vor allem der normale Anwendungsfall, bei dem viele kurze Linien gezeichnet werden, wurde stark verbessert. Der anfängliche Effekt der verzögerten Darstellung einer mit dem Stift gezogenen Linie wurde so stark reduziert, dass er nun kaum mehr spürbar ist.

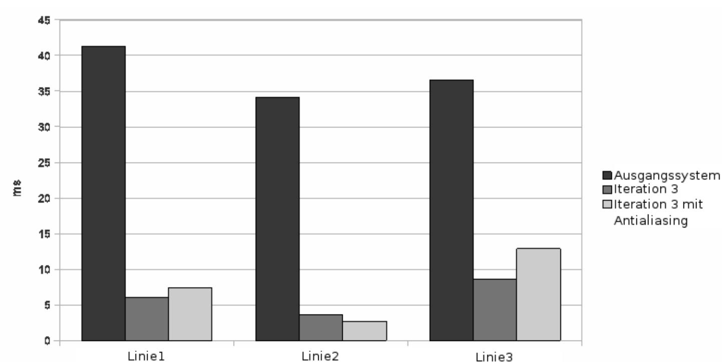

Abbildung 8: Geschwindigkeitsmessung der Linien

6 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit habe ich den Prototypen der Software *QSheet* analysiert und auf Basis der gewonnenen Ergebnisse ein Refactoring durchgeführt. Aufgrund der dadurch erworbenen Kenntnisse und der verbesserten Struktur des Quelltextes war ich anschließen in der Lage, für das Projekt wichtige Verbesserungen in der Performanz schnell und einfach durchzuführen.
Die erreichten Performanzverbesserungen bringen der Software und damit dem ganzen Projekt einen deutlichen Vorteil. Ein schnelleres Reagieren der Anzeige auf die Eingaben des Stiftes bringt die Arbeitsumgebung näher an das Ideal des Verfahrens heran - eine höchstmögliche Papieräquivalenz, in der keine Zeit zwischen dem Zeichnen mit einem Stiftes und dem Erscheinen der gemalten Linie vergeht. Wichtiger als ein Nachweis einer Geschwindigkeitssteigerung durch Auswertung der Testläufe ist dabei die erfolgreiche Abnahme in den regelmäßigen Rücksprachen mit Böschen, da dies zeigt, dass die Modifikationen nicht das Abarbeiten eines Plans, sondern die Lösung eines Problems des Auftraggebers bedeutet. Damit sind die Änderungen eindeutig als erfolgreiche Verbesserungen zu deuten, die dem Projekt einen Nutzen bringen.
Sie zeigen darüber hinaus auch den Erfolg der eingesetzten Methoden und Werkzeuge.

6.1 Fazit der Methoden und Werkzeuge

Es ist nicht auszuschließen, dass die Verbesserungen ohne die vorhergehende Analyse und das Refactoring möglich gewesen wären. Sie wären aber sicher komplizierter und zeitaufwendiger gewesen. Dazu existiert nun eine im Vergleich zum Ausgangssystem besser Les- und Wartbare Quelltextbasis mit einer stehenden Testumgebung, auf der die zukünftigen Erweiterungen einfacher durchzuführen sein werden.
Durch das iterativ inkrementelle Vorgehen konnte ich direkt auf die Änderungen reagieren, die selbst in diesem kurzen Entwicklungszeitraum auftraten, wie beispielsweise die neue Bewertung der Priorität der beiden Performanzprobleme. Auch das Refactoring wurde durch die Rückmeldungen, die ich zwischen den Iterationen erhalten habe, positiv beeinflusst. Ein vollständiger Entwicklungsplan, der zum Zeitpunkt der ersten Analyse aufgestellt und nicht mehr angepasst worden wäre, hätte definitiv andere Ergebnisse geliefert, da mein Wissen über den Quelltext vermehrt und die Prioritäten des Anwenders sich während der Arbeit verändert haben. Diese anderen Ergebnisse wären nicht

unbedingt qualitativ schlechtere Software, aber weniger nutzbringend für den Kunden und das gesamte Projekt gewesen. Als Beispiel lässt sich die dritte Iteration aufführen, in der ich auf die veränderte Situation reagieren und neben dem ursprünglichen Plan, die Zeichengeschwindigkeit verbessern konnte.

Auch die eingesetzten Open Source-Werkzeuge haben sich als nützlich erwiesen. Emma und FindBugs konnten das Projekt in dem Umfang unterstützen, den ich von ihnen erwartete und ließen sich ohne größere Aufwände als Plug-In-Module der Eclipse IDE einrichten und zu benutzen. Auch den Einsatz von XRadar bewerte ich als vorteilhaft für mein Arbeiten in diesem Kontext, obwohl es deutlich komplizierter und zeitaufwändiger einzurichten und zu benutzen war, als ich erwartet hatte. Die regelmäßigen Analysen konnte ich als zusätzliche Rückversicherung der akuten Änderung verwenden. Neben den Unit-Test, die Änderungen im Verhalten aufzeigen konnten, konnte ich durch XRadar viele potentiell ungewollte Veränderungen in Architektur oder Design abfangen, wie neue zyklische Referenzen oder Programmierfehler, die im Musterkatalog von FindBugs enthalten sind. Zu beachten ist dabei, dass die Metriken, die von XRadar automatisiert erstellt werden, eine von einem Entwickler durchgeführte Analyse hilfreich unterstützen, aber nicht ersetzen können. Das Beispiel aus den Ergebnissen der ersten Iteration (Siehe Abschnitt 5.1.3) zeigt, wie eine leichte Verbesserung des Designs eine Verschlechterung der Metriken bewirken kann. XRadar war in diesem Fall also nicht in der Lage die Veränderung so zu interpretieren, wie es ein menschlicher Entwickler würde. Daher sind alleinstehende Metriken für eine äquivalente Analyse nicht ausreichend.

6.2 Weitere Aufgaben im Projekt

Softwaretechnisch existiert in dem Projekt weiterhin Handlungsbedarf, der möglicherweise im Rahmen weiterer Arbeiten abgedeckt werden kann.
Das zurzeit am höchsten priorisierte Ziel des Projektes ist das Erstellen einer Softwarekomponente, die die mit epp-Verfahren gewonnen Daten (teil-) automatisiert auswertet. Das Verfahren wurde kürzlich auf dem *47. Kongresses der Deutschen Gesellschaft für Psychologie* in Bremen präsentiert. Dabei zeigte sich, dass ein Werkzeug zur Auswertung der Daten von zentraler Bedeutung für die AnwenderInnen ist ist, in diesem Fall PsychologInnen und SozialwissenschaftlerInnen. Dabei ist unter anderem zu klären, zu welchen Teilen es sinnvoll ist neue Auswertungsalgorithmen zu schreiben,

beziehungsweise die Berechnungen der Variablen so weit wie möglich an andere Werkzeuge, wie die frei Statistiksoftware R[14], zu delegieren.

[14]/www.r-project.org

Literatur

[APM+07] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Using findbugs on production software. In *Conference on Object Oriented Programming Systems Languages and Applications. Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion.*, pages 805 – 806, Montreal, Quebec, Canada, 2007. ACM New York.

[BA04] Kent Beck and Cynthia Andres. *Extreme programming explained. Embrace change.* Addison-Wesley, Boston, 2 edition, 2004.

[Bös09] Ingmar Böschen. Der e-paper-pencil Fragebogen. Äquivalenzüberprüfung einer Papier-Bleistift- und e-paper-pencil-Testung. 2009. Diplomarbeit im Studiengang Psychologie des Fachbereichs Psychologie der Universität Hamburg.

[BPKL06] Petra Becker-Pechau, Bettina Karstens, and Carola Lilienthal. Automatisierte Softwareüberprüfung auf der Basis von Architekturregeln. In *Software Engineering*, pages 27–37, 2006.

[CHH+06] Brian Cole, Daniel Hakim, David Hovemeyer, Reuven Lazarus, William Pugh, and Kristin Stephens. Improving your software using static analysis to find bugs. In *Conference on Object Oriented Programming Systems Languages and Applications. Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications.*, pages 673 – 674, Portland, Oregon, USA, 2006. ACM New York.

[Coc07] Alistair Cockburn. *Agile software development. The cooperative game.* Addison-Wesley, Upper Saddle River, NJ u.a., 2 edition, 2007.

[Fow07] Martin Fowler. *Refactoring. Improving the design of existing code.* Addison-Wesley, Boston, 20 edition, 2007.

[GHJV96] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software.* Addison-Wesley, München [u.a], 1996.

[HRS09] Peter Hruschka, Chris Rupp, and Gernot Starke. *Agility kompakt. Tipps für erfolgreiche Systementwicklung.* Spektrum Akademischer Verlag, Heidelberg, 2 edition, 2009.

[Ker04] Joshua Kerievsky. *Refactoring to patterns.* Addison-Wesley, Boston, 2004.

[KJVK07] P. Květon, M. Jelinek, D. Vobořil, and K. Klimusova. Computer-based tests: the impact of design and problem of equivalency. *Computers in Human Behaviour*, 23:32–51, 1 2007.

[Lin05] Johannes Link. *Softwaretests mit JUnit. Techniken der testgetriebenen Entwicklung.* dpunkt-Verlag, Heidelberg, 2 edition, 2005.

[PHW03] Richard R. Plant, Nick Hammond, and Tom Whitehouse. How choice of mouse may affect responsetiming in psychological studies. *Behavior Research Methods, Instruments and Computers*, 35:276–284, 2003.

[Rai01] Joe B. Rainsberger. Use your singletons wisely. *IBM developerWorks*, 6 2001. Verfügbar unter: www-106.ibm.com/developerworks/components/library/co-single.html.

[TTMW98] Hsu-Min Tseng, Brian Tiplady, Hamish A. Macleod, and Peter Wright. Computer anxiety: A comparsion of pen-based personal digital assistants, conventional computer and paper assessment of mood and performance. *British Journal of Psychology*, 89:599–610, 1998.

[Wei02] Martin Weichbold. Touchscreenbefragungen. *Östereichische Zeitschrift für Sozialpsychologie*, pages 93–107, 3 2002.

[WM03] M. M. Wagner-Menghin. Computerdiagnostik. In K.D. Kubinger and R.S. Jäger, editors, *Schlüsselbegriffe der Psychologischen Diagnostik*, pages 68–82. Beltz PVU, Weinheim, 2003.

[Zü98] Heinz Züllighoven. *Das objektorientierte Konstruktionshandbuch: nach dem Werkzeug & Material-Ansatz.* dpunkt-Verlag, Heidelberg, 1 edition, 1998.

Erklärung

Ich versichere, dass ich die Arbeit selbstständig verfasst und keine anderen, als die angegebenen Hilfsmittel - insbesondere keine im Quellenverzeichnis nicht benannten Internetquellen – benutzt habe, die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Lüneburg, 26. Oktober 2010, Nils Jochen Kubera